FIFTH EDITION

CRACKING THE CODING INTERVIEW

150 Programming Questions and Solutions

Gayle Laakmann McDowell
Founder / CEO, CareerCup.com

CRACKING THE CODING INTERVIEW

5th Edition

Also by Gayle Laakmann McDowell

The Google Resume

How to Prepare for a Career and Land a Job at Apple, Microsoft, Google, or any Top Tech Company

CRACKING THE CODING INTERVIEW

5th Edition

150 Programming Questions and Solutions

GAYLE LAAKMANN MCDOWELL
Founder and CEO, CareerCup.com

CareerCup, LLC
Palo Alto, CA

CRACKING THE CODING INTERVIEW, FIFTH EDITION

Published by CareerCup, LLC, Palo Alto, CA. Version 5.01390210100131.

For more information, contact support@careercup.com.

978-0984782802 (ISBN 13)

To Pauline "Oma" Venti
for her eternal support

Table of Contents

Foreword 1

Introduction 2

I. The Interview Process 5

Overview 6

How Questions are Selected 7

Timeline and Preparation Map 8

The Evaluation Process 10

Incorrect Answers 11

Dress Code 12

Top 10 Mistakes 13

Frequently Asked Questions 15

II. Behind the Scenes 17

The Microsoft Interview 19

The Amazon Interview 20

The Google Interview 21

The Apple Interview 22

The Facebook Interview 23

The Yahoo! Interview 24

III. Special Situations 25

Experienced Candidates 26

Testers and SDETs 27

Program and Product Managers 28

Dev Leads and Managers 30

Start-Ups 31

IV. Before the Interview 33

Getting the Right Experience 34

Building a Network 35

Writing a Great Resume 37

V. Behavioral Questions 39

Behavioral Preparation 40

Handling Behavioral Questions 43

VI. Technical Questions 45

Technical Preparation 46
Handling Technical Questions 49
Five Algorithm Approaches 52
What Good Coding Looks Like 56
VII. The Offer and Beyond 61
Handling Offers and Rejection 62
Evaluating the Offer 63
Negotiation 65
On the Job 66
VIII. Interview Questions 67
Data Structures 69
Chapter 1 | Arrays and Strings 71
Chapter 2 | Linked Lists 75
Chapter 3 | Stacks and Queues 79
Chapter 4 | Trees and Graphs 83
Concepts and Algorithms 87
Chapter 5 | Bit Manipulation 89
Chapter 6 | Brain Teasers 93
Chapter 7 | Mathematics and Probability 97
Chapter 8 | Object-Oriented Design 103
Chapter 9 | Recursion and Dynamic Programming 107
Chapter 10 | Scalability and Memory Limits 111
Chapter 11 | Sorting and Searching 117
Chapter 12 | Testing 123
Knowledge Based 131
Chapter 13 | C and C++ 133
Chapter 14 | Java 141
Chapter 15 | Databases 147
Chapter 16 | Threads and Locks 153
Additional Review Problems 161
Chapter 17 | Moderate 163
Chapter 18 | Hard 167

IX. **Solutions** 169

Data Structures 171

Chapter 1 | Arrays and Strings 171

Chapter 2 | Linked Lists. 183

Chapter 3 | Stacks and Queues. 201

Chapter 4 | Trees and Graphs 219

Concepts and Algorithms 241

Chapter 5 | Bit Manipulation 241

Chapter 6 | Brain Teasers. 257

Chapter 7 | Mathematics and Probability 263

Chapter 8 | Object-Oriented Design 279

Chapter 9 | Recursion and Dynamic Programming 315

Chapter 10 | Scalability and Memory Limits. 341

Chapter 11 | Sorting and Searching. 359

Chapter 12 | Testing. 377

Knowledge Based 385

Chapter 13 | C and C++ 385

Chapter 14 | Java. 399

Chapter 15 | Databases. 407

Chapter 16 | Threads and Locks 415

Additional Review Problems 429

Chapter 17 | Moderate 429

Chapter 18 | Hard 461

X. **Acknowledgements** 491

XI. **Index** 492

XII. **About the Author** 500

*Join us at **www.CrackingTheCodingInterview.com** to download full, compilable Java / Eclipse solutions, discuss problems from this book with other readers, report issues, view this book's errata, post your resume, 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. And I've debated in Google's Hiring Committee whether or not a candidate did well enough to merit an offer. I understand and have experienced the full hiring circle.

And you, reader, are probably preparing for an interview, perhaps tomorrow, next week, or next year. You likely have or are working towards a Computer Science or related degree. I am not here to re-teach you the basics of what a binary search tree is, or how to traverse a linked list. You already know such things, and if not, there are plenty of other resources to learn them.

I am here to help you take your understanding of Computer Science fundamentals to the next level, to learn how to apply those fundamentals to crack the coding interview.

The 5th edition of *Cracking the Coding Interview* updates the 4th edition with over 200 pages of additional questions, revised solutions, new chapter introductions, and other content. Be sure to check out our website, www.careercup.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!), but 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.

Study hard, practice, and good luck!

Gayle L. McDowell

Founder / CEO, CareerCup.com
Author of *The Google Resume* and *Cracking the Coding Interview*

Something's Wrong

We walked out of the hiring meeting frustrated, again. Of the ten "passable" 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 who 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 worked hard. He was sharp. 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 would sway them to reconsider, he would surely get rejected in the later stages of the hiring process. There were just too many red flags.

Though the interviewers generally believed that he was quite intelligent, he had 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 then 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, but they're not going to help you much in an interview. 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.

Cracking the Coding Interview is the result of my first-hand experience interviewing at top companies. 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 150 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 where there's more to learn.

My Passion

Teaching is my passion. I love helping people understand new concepts and giving them tools so that they can 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 CS course at the university focused on "hands-on" skills.

As an engineer at Google, training and mentoring "Nooglers" (yes, that's really what they call new Google employees!) were some of the things I enjoyed most. I went on to use my "20% time" to teach two Computer Science courses at the University of Washington.

Cracking the Coding Interview, ***The Google Resume***, and **CareerCup.com** reflect my passion for teaching. Even now, you can often find me "hanging out" at CareerCup.com, helping users who stop by for assistance.

Join us.

Gayle L. McDowell

The Interview Process

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. 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 interviewers before being brought on-site.

In an on-site interview round, you usually have 4 to 6 in-person interviews. One of these will be 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 and algorithm questions. You should also expect some questions about your resume.

Afterwards, the interviewers meet to discuss your performance and/or submit written feedback. At most companies, your recruiter should respond to you within a week with an update on your status.

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.

Candidates frequently ask me what the "recent" interview questions are at a specific company, assuming that the questions change over time. The reality is that the company itself has typically little to do with selecting the questions. It's all up to the interviewer. Allow me to explain.

At a large company, interviewers typically go through an interviewer training course. My "training" course at Google was outsourced to another company. Half of the one-day training course focused on the legal aspects of interviewing: don't ask a candidate if they're married, don't ask about their ethnicity, and so on. The other half discussed "troublesome" candidates: the ones who get angry or rude when asked a coding question or other questions they think are "beneath" them. After the training course, I "shadowed" two real interviews to show me what happened in an interview—as though I didn't already know! After that, I was sent off to interview candidates on my own.

That's it. That's all the training we got—and that's very typical of companies.

There was no list of "official Google interview questions." No one ever told me that I should ask a particular question, and no one told me to avoid certain topics.

So where did my questions come from? From the same places as everyone else's.

Interviewers borrow questions that they were asked as candidates. Some swap questions amongst each other. Others look on the internet for questions, including—yes—on CareerCup.com. And some interviewers take questions from the earlier mentioned resources and tweak them in minor or major ways.

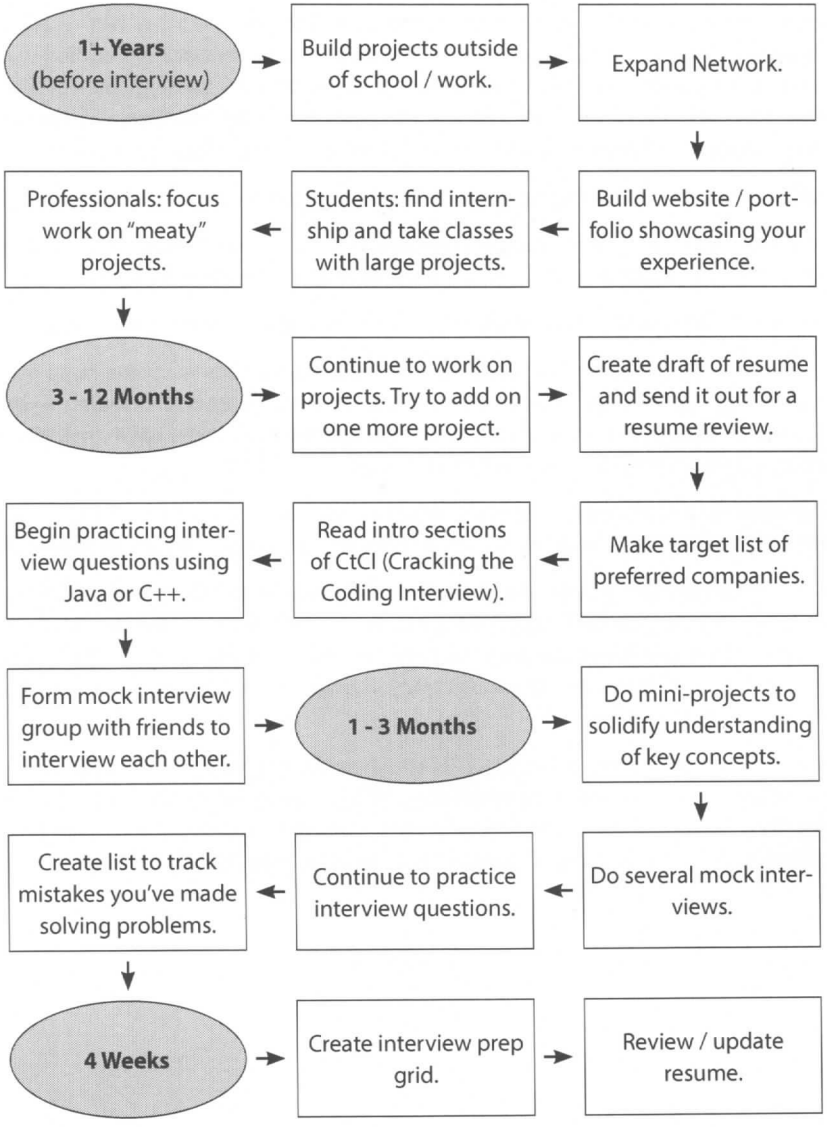
It's unusual for a company to ever give interviewers a list of questions. Interviewers pick their own questions and tend to each have a pool of five or so questions that they prefer.

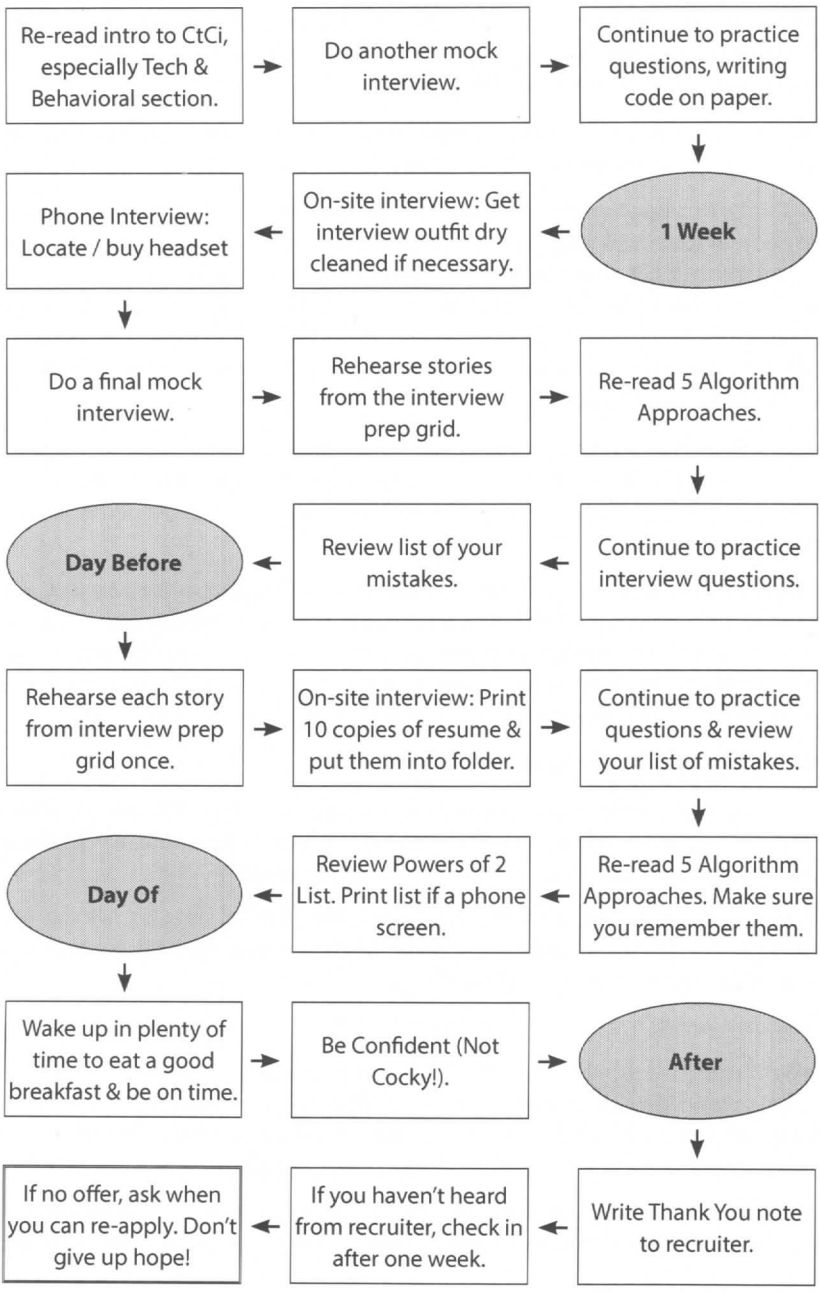
So next time you want to know what the "recent" Google interview questions are, stop and think. Google interview questions are really no different from Amazon interview questions since the questions aren't decided at a company-wide level. The "recent" questions are also irrelevant. Questions don't change much over time since no one's telling anyone what to do.

There are some differences in broad terms across companies. Web-based companies are more likely to ask system design questions, and a company using databases heavily is more likely to ask you database questions. Most questions, however, fall into the broad "data structures and algorithms" category and could be asked by any company.

Acing an interview starts well before the interview itself—years before, in fact. You need to get the right technical experience, apply to companies, and begin preparing to actually solve questions. 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!




Re-read intro to CtCi, especially Tech & Behavioral section.
Do another mock interview.
Continue to practice questions, writing code on paper.
1 Week
On-site interview: Get interview outfit dry cleaned if necessary.
Phone Interview: Locate / buy headset
Do a final mock interview.
Rehearse stories from the interview prep grid.
Re-read 5 Algorithm Approaches.
Continue to practice interview questions.
Review list of your mistakes.
Day Before
Rehearse each story from interview prep grid once.
On-site interview: Print 10 copies of resume & put them into folder.
Continue to practice questions & review your list of mistakes.
Re-read 5 Algorithm Approaches. Make sure you remember them.
Review Powers of 2 List. Print list if a phone screen.
Day Of
Wake up in plenty of time to eat a good breakfast & be on time.
Be Confident (Not Cocky!).
After
Write Thank You note to recruiter.
If you haven't heard from recruiter, check in after one week.
If no offer, ask when you can re-apply. Don't give up hope!

Most recruiters will probably tell you that candidates are evaluated on four aspects: prior experience, culture fit, coding skills, and analytical ability. These four components are certainly all in play, but typically, the decision comes down to your coding skills and your analytical ability (or intelligence). This is why most of this book is devoted to improving your coding and algorithm skills.

However, just because the decision usually comes down to coding and algorithm skills doesn't mean you should overlook the other two as factors.

At bigger tech companies, your prior experience tends not to be a direct deciding factor once you're actually interviewing, but it may bias an interviewer's perception of the rest of your interview. For example, if you demonstrate brilliance when you discuss some tricky program you wrote, your interviewer is more likely to think, "Wow, she's brilliant!" And once he's decided that you're smart, he's more likely to subconsciously overlook your little mistakes. Interviewing, after all, is not an exact science. Preparing for "softer" questions is well worth your time.

Culture fit (or your personality, particularly with relation to the company) tends to matter more at smaller companies than at big companies. One way it might come up is if the company's culture is to let employees make decisions independently, and you need direction.

It's not unusual for a candidate to get rejected because they appear too arrogant, argumentative, or defensive. I once had a candidate blame his struggling with a question on my wording of the problem, and later, on the way that I'd coached him through it. I recorded this defensiveness as a potential red flag—and, as it turns out, so did the other interviewers. He was rejected. Who wants to work with a teammate like that?

What this means for you is the following:

- If people often perceive you as arrogant or argumentative, or with any other nasty adjectives, keep an eye on this behavior in an interview. Even an otherwise superstar candidate may get rejected if people don't want to work with them.
- Spend some time preparing for questions about your resume. It's not the most important factor, but it matters. Even a little bit of time here can help you improve in major ways. It's a great "bang for your buck."
- Focus mainly on coding and algorithm questions.

Finally, it's worth noting that interviewing is not a perfect science. There is some randomness not only in your performance, but also in the decision of the hiring committee (or whoever decides on your offer). Like any group, the hiring committee is easily swayed by the most outspoken individuals. It may not be fair, but that's the way it is.

And remember—a rejection is not a life sentence. You can almost always reapply within a year, and many candidates get offers from companies that previously rejected them.

Don't get discouraged. Keep at it.

One of the most pervasive—and dangerous—rumors is that candidates need to get every question right. That's not even close to 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 ask myself, how many questions did they get right? Rather, it's about how optimal your final solution was, how long it took you to get there, and how clean your code was. It's not a binary right vs. wrong; there are 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.

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.

Software engineers and those in similar positions typically dress less formally. This is reflected in the appropriate interview attire. A good rule of thumb for any interview is to dress one small notch better than the employees in your position.

More specifically, I would recommend the following attire for software engineering (and testing) interviews. These rules are designed to put you in the "safe zone": not too dressy, and not too casual. Many people interview at start-ups and big companies in jeans and a t-shirt and don't face any problems. After all, your coding skills matter far more than your sense of style.

	Start-Ups	Microsoft, Google, Amazon, Facebook, e.t.c.	Non-Tech Companies (including banks)
Men	Khakis, slacks, or nice jeans. Polo shirt or dress shirt.	Khakis, slacks, or nice jeans. Polo shirt or dress shirt.	Suit, no tie. (Consider bringing a tie just in case.)
Women	Khakis, slacks, or nice jeans. Nice top or sweater.	Khakis, slacks, or nice jeans. Nice top or sweater.	Suit, or nice slacks with a nice top.

These are just good advisements, and you should consider the culture of the company with which you're interviewing. If you are interviewing for a Program Manager, Dev Lead, or any role closer to management or the business side, you should lean towards the more dressy side.

#1 | Practicing on a Computer

If you were training for an ocean swim race, would you practice only by swimming in a pool? Probably not. You'd want to get a feel for the waves and other "terrain" differences. I bet you'd want to practice in the ocean, too.

Using a compiler to practice interview questions is like doing all your training in the pool. Put away the compiler and get out the old pen and paper. Use a compiler only to verify your solutions *after* you've written and hand-tested your code.

#2 | Not Rehearsing Behavioral Questions

Many candidates spend all their time prepping for technical questions and overlook the behavioral questions. Guess what? Your interviewer is judging those too!

And, not only that—your performance on behavioral questions might bias your interviewer's perception of your technical performance. Behavioral prep is relatively easy and well-worth your time. Look over your projects and positions and rehearse your key stories.

#3 | Not Doing a Mock Interview

Imagine you're preparing for a big speech. Your whole school, company, or whatever will be there. Your future depends on this. You'd be crazy to only practice the speech silently in your head.

Not doing a mock interview to prepare for your real one is just like this. If you're an engineer, you must know other engineers. Grab a buddy and ask him/her to do a mock interview with you. You can even return the favor!

#4 | Trying to Memorize Solutions

Memorizing the solution to a specific problem will help you solve that one if it comes up in an interview, but it won't help you to solve new problems. It's very unlikely that all, or even most, of your interview questions will come from this book.

It's much more effective to try to struggle through the problems in this book yourself, without flipping to the solutions. This will help you develop strategies to approach new problems. Even if you review fewer problems in the end, this kind of preparation will go much further. Quality beats quantity.

#5 |Not Solving Problems Out Loud

Psst—let me tell you a secret: I don't know what's going on in your head. So if you aren't talking, I don't know what you're thinking. If you don't talk for a long time, I'll assume that you aren't making any progress. Speak up often, and try to talk your way through a solution. This shows your interviewer that you're tackling the problem and aren't stuck.

And it lets them guide you when you get off-track, helping you get to the answer faster. Best of all, it demonstrates your awesome communication skills. What's not to love?

#6 | Rushing

Coding is not a race, and neither is interviewing. Take your time when working on a coding problem. Rushing leads to mistakes and suggests that you are careless. Go slowly and methodically, testing often and thinking through the problem thoroughly. In the end, you'll finish the problem in less time and with fewer mistakes.

#7 | Sloppy Coding

Did you know that you can write bug-free code but still perform horribly on a coding question? It's true! Duplicated code, messy data structures (i.e., lack of object-oriented design), and so on. Bad, bad, bad! When you write code, imagine you're writing for real-world maintainability. Break code into sub-routines, and design data structures to link appropriate data.

#8 | Not Testing

You probably wouldn't write code in the real world without testing it, so why do that in an interview? When you finish writing code in an interview, "run" (or walk through) the code to test it. Or, on more complicated problems, test the code while writing it.

#9 | Fixing Mistakes Carelessly

Bugs will happen; they're just a matter of life, or of coding. If you're testing your code carefully, then you will probably discover bugs. That's okay.

The important thing is that when you find a bug, you think through why it occurred before fixing it. Some candidates, when they find that their function returns `false` for particular parameters, will just flip the return value and check if that fixes the issue. Of course, it rarely does; in fact, it tends to create even more bugs and demonstrates that you're careless.

Bugs are acceptable, but changing your code randomly to fix the bugs is not.

#10 | Giving 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.

Should I tell my interviewer if I know a question?

Yes! You should definitely tell your interviewer if you've previously heard the question. This seems silly to some people—if you already know the question (and answer), you could ace the question, right? Not quite.

Here's why we strongly recommend that you tell your interviewer that you've heard the problem before:

1. Big honesty points. This shows a lot of integrity—that's huge! Remember that the interviewer is evaluating you as a potential teammate. I don't know about you, but I personally prefer to work with honest people.
2. The question might have changed ever so slightly. You don't want to risk repeating the wrong answer.
3. If you easily belt out the right answer, it's obvious to the interviewer. They know how difficult a problem is supposed to be. If you instead try to pretend to struggle through a problem, you may very well wind up "struggling" *too* much and coming off unqualified.

What language should I use?

Many people will tell you to use whatever language you're most comfortable with, but ideally you want to use a language that your interviewer is comfortable with. I'd usually recommend coding in either C, C++ or Java, as the vast majority of interviewers will be comfortable in one of these languages. My personal preference for interviews is Java (unless it's a question requiring C / C++), because it's quick to write and almost everyone can read and understand Java, even if they code mostly in C++. For this reason, almost all the solutions in this book are written in Java.

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

No. Almost every company intends to tell candidates when they're rejected. Not hearing back quickly could mean almost anything. You might have done very well, but the recruiter is on vacation and can't process your offer yet. The company might be going through a re-org and be unclear what their head count is. Or, it might be that you did poorly, but you got stuck with a lazy or overworked recruiter who hasn't gotten a chance to notify you. It would be a strange company that actually decides, "Hey, we're rejecting this person, so we just won't respond." It's in the company's best interest to notify you of your ultimate decision. Always follow up.

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

Almost always, but you typically have to wait a bit (6 months – 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.

Behind the Scenes

II

II. Behind the Scenes

For many candidates, interviewing is a bit of a black box. You walk in, you get pounded with questions from a variety of interviewers, and then somehow, you return with an offer... or not.

Have you ever wondered:

- How do decisions get made?
- Do your interviewers talk to each other?
- What does the company really care about?

Well, wonder no more!

For this book, we sought out interviewing experts from five top companies–Microsoft, Amazon, Google, Apple, Facebook, and Yahoo!–to show you what really happens "behind the scenes."

These experts will guide us through a typical interview day, describing what takes place outside of the interviewing room and what transpires after you leave.

Our interviewing experts also told us what's different about their interview process. From bar raisers (Amazon) to Hiring Committees (Google), each company has its own quirks. Knowing these idiosyncrasies will help you to react better to a super-tough interviewer (Amazon), or to avoid being intimidated when two interviewers show up at the door (Apple).

In addition, our specialists offered insight as to what their company stresses in their interviews. While almost all software firms care about coding and algorithms, some companies focus more than others on specific aspects of the interview. Whether this is because of the company's technology or its history, now you'll know what and how to prepare.

So, join us as we take you behind the scenes at Microsoft, Facebook, Google, Amazon, Yahoo! and Apple.

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. 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!

Amazon's recruiting process typically begins with two phone screens in which a candidate interviews with a specific team. A small portion of the time, a candidate may have three 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, such as CollabEdit. 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 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 is a web-based company, and that means they care 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 Scalability and Memory Limits 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.

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 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 "Scalability and Memory Limits." Additionally, many Google interviewers will ask questions involving Bit Manipulation, so you are advised to brush up on these topics as well.

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.

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 1-on-1 and 2-on-1 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 2-on-1 interviews often, but don't get stressed out about them–it's the same as a 1-on-1 interview!

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

Though Facebook's online engineering puzzles get a lot of hype, they're merely one more way to get noticed. You can still apply without solving these puzzles, through the traditional avenues like an online job application or your university career fair.

Once selected for an interview, candidates will generally do a minimum of two phone screens. Local candidates, however, will often do just one interview before being invited on-site. Phone screens will be technical and will involve coding, usually via Etherpad or another online document editor.

If you are in college and are interviewing on your campus, you will also do coding. This will be done either on a whiteboard (if one is available) or on a sheet of paper.

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. Questions are broken down into algorithm / coding skills, architecture / design skills, and the ability to be successful in Facebook's fast-paced environment.

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.

Facebook looks for "ninja skills"–the ability to 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.

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.

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.

While Yahoo! tends to only recruit from the top twenty schools, other candidates can still get interviewed through Yahoo's job board (or—better yet—through an internal referral). If you are selected for an interview, your interview process will start off with a phone screen. Your phone screen will be with a senior employee such as a tech lead or manager.

During your on-site interview, you will typically interview with 6 – 7 people on the same team for 45 minutes each. Each interviewer will have an area of focus. For example, one interviewer might focus on databases, while another interviewer might focus on your understanding of computer architecture. Interviews will often be composed as follows:

- *5 minutes:* General conversation. Tell me about yourself, your projects, etc.
- *20 minutes:* Coding question. For example, implement merge sort.
- *20 minutes:* System design. For example, design a large distributed cache. These questions will often focus on an area from your past experience or on something your interviewer is currently working on.

At the end of the day, you will likely meet with a Program Manager or someone else for a general conversation. This may include a product demos or a discussion about potential concerns about the company or your competing offers. This is usually not a factor in the decision.

Meanwhile, your interviewers will discuss your performance and attempt to come to a decision. The hiring manager has the ultimate say and will weigh the positive feedback against the negative.

If you have done well, you will often get a decision that day, but this is not always the case. There can be many reasons that you might not be told for several days—for example, the team may feel it needs to interview several other people.

Definitely Prepare:

Yahoo!, almost as a rule, asks questions about system design, so make sure you prepare for that. They want to know that you can not only write code, but can also design software. Don't worry if you don't have a background in this–you can still reason your way through it!

What's Unique:

Your phone interview will likely be performed by someone with more influence, such as a hiring manager.

Yahoo! is also unusual in that it often gives a decision (if you're hired) on the same day. Your interviewers will discuss your performance while you meet with a final interviewer.

Special Situations

III

If you read the prior section carefully, the following shouldn't surprise you: experienced candidates are asked very similar questions as inexperienced candidates, and the standards don't vary significantly.

Most questions, as you may know, are general questions covering data structures and algorithms. The major companies feel that this is a good test of one's abilities, so they hold everyone to that test.

Some interviewers may hold experienced candidates to a slightly higher standard on those questions. After all, an experienced candidate has many more years of experience and *should* perform better, right?

It turns out that other interviewers see things in exactly the opposite way. Experienced candidates are years out of school and may not have touched some of these concepts since then. It's expected that they would forget some details, so we *should* hold them to a lower standard.

On average, it balances out. If you're an experienced candidate, you'll be asked roughly the same types of questions and held to roughly the same standard.

The exception to this rule is system design and architecture questions, as well as questions on 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.

SDETs are in a tough spot. Not only do they have to be great coders, but they must also be great testers.

We recommend the following preparation process:

- *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 SDETs get rejected for is coding skills. Although coding standards are typically lower for an SDET than for an SDE, SDETs 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, "OK, 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.

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 will typically 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? 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. An intimate knowledge of instant messengers is probably not necessary to work on the MSN Messenger team, 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 challenges of the job.

- *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.

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.

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

The Application Process

Many start-ups might post job listings, but for the hottest start-ups, often the best way in is through a personal referral. This referrer 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, most smaller start-ups 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 start-up, your best bet is to reach out to a professional recruiter who works with many start-ups or to focus your search on bigger start-ups.

Resume Selection Factors

Start-ups are 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.

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, start-ups 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 start-ups 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 start-up works with, make sure to brush up on the details.
- *Prior Experience:* Start-ups are likely to ask you a lot of questions about your prior 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.

Before the Interview

IV

Although offer decisions are typically based more on the interview than anything else, it's your resume—and therefore your prior experience—that gets you the interview. You should think actively about how to enhance your technical (and non-technical) experience. Both students and professionals will benefit greatly by adding additional coding experience.

For current students, this may mean the following:

- *Take the Big Project Classes:* If you're in school, don't shy away from the classes with big, "meaty" projects. These projects belong on your resume and will greatly improve your chances at getting an interview with the top companies. The more relevant the project is to the real world, the better.
- *Get an Internship:* Even at a relatively early stage in school, students can get relevant professional experience. Freshmen and sophomores can get early experience through programs like Microsoft Explorer and Google Summer of Code. If you can't score one of these internships, start-ups are also a great option.
- *Start Something:* Trying your hand at something more entrepreneurial will impress almost any company. It develops your technical experience and shows that you have initiative and can "get things done." Use your weekends and breaks to build some software on your own. If you get to know a professor, you might even get her to agree to "sponsor" your work as an independent study.

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.

You probably already know that many people get jobs from their friends. But what you may not know is that, in fact, even more people get jobs from their *friends of friends*. And this really makes perfect sense. To drop into geek-speak for a second, you may have N friends, but you have N^2 friends of friends.

So what does this all mean for your job-finding possibilities? It means that both your immediate and your extended network is critical to finding a job.

What Makes a Network a "Good Network"

A good network is one that is both broad and close. If it seems contradictory, that's because it is (somewhat).

- *Broad:* A network should be somewhat focused on your own industry (technology), but it should also be broad and cover many industries. An accountant, for example, can be valuable to you career-wise simply because he or she probably has lots of friends outside of accounting. Some of those friends—who are *your* friends of friends—will probably be looking for someone just like you at some point. Be open to connecting with anyone you meet.
- *Close:* It's much easier to access a friend of a friend via a close friend of yours than it is through a more distant acquaintance. Moreover, people who are seen as "professional networkers" or "card collectors" are often written off by others as being too fake. Make your connections deep and meaningful.

The trick to finding a balance here is to meet whoever you can, but to be open and genuine to everyone. When you just try to "collect cards," you often wind up with nothing at all.

How to Build a Strong Network

Some people argue that you "just" need to get out and meet people, and that's mostly true. But where? And how do you go from an introduction to a real connection?

The following basic steps will help:

1. Use websites like Meetup.com or your alumni network to discover events that are relevant to your interests and goals. Bring business cards. If you don't have them because you're unemployed or a student, make some.
2. Walk up and say, "Hello," to people. It may seem scary to you, but honestly, no one will hold it against you. Most people will even appreciate your assertiveness. What's the worst that can happen? They don't like you, don't establish a connection with you, and you never see them again?
3. Be open about your interests, and talk to people about theirs. If they're running a start-up or something else you might have some interest in, ask to grab coffee to chat more.

4. Follow-up after the events by adding the person on LinkedIn and by emailing them. Or, even better, ask to meet them for coffee to discuss their start-up, or whatever they're working on that could be mutually interesting.
5. And, most importantly, *be helpful*. By lending a hand in some way to people, you will be seen as generous and friendly. People will want to help you if you have helped them.

And remember, your network is more than just your face-to-face network. In this day and age, your network can extend to strictly online interactions through blogs, Twitter, Facebook, and email.

However, when your interaction has been strictly online, you must work much harder to actually establish a bond.

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, and no more than two pages otherwise. Why is this? Here are two great reasons to do this:

- Recruiters only spend a fixed amount of time (about 20 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 page, trust me, *you can*. Everyone says this at first. 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. Your job serving ice cream, for example, will not show that you're smart or that you can code. You should include only the relevant positions.

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.

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.

Programming Languages and Software

Software

Generally speaking, I do not recommend listing that you're familiar with Microsoft Office. Everyone is, and it just dilutes the "real" information. Familiarity with highly technical software (e.g., Visual Studio, Linux) *can* be useful, but, frankly, it usually doesn't make much of a difference.

Languages

Knowing which languages to include on your resume is always a tricky thing. Do you list everything you've ever worked with, or do you shorten the list to just the ones that you're most comfortable with? I recommend the following compromise: 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).

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.

Behavioral Questions

V

Behavioral questions are asked for a variety of reasons. They can be asked to get to know your personality, to understand your resume more deeply, or just to ease you into an interview. Either way, these questions are important and can be prepared for.

How to Prepare

Behavioral questions are usually of the form "Tell me about a time when you...," and may require an example from a specific project or position. I recommend filling in the following "preparation grid" as shown below:

Common Questions	Project 1	Project 2	Project 3	Project 4
Most Challenging				
What You Learned				
Most Interesting				
Hardest Bug				
Enjoyed Most				
Conflicts with Teammates				

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 questions: what you enjoyed most, what you enjoyed least, what you considered most challenging, what you learned, what the hardest bug was, and so on. In each cell, put the corresponding story.

In your interview, when you're asked about a project, you'll be able to come up with an appropriate story effortlessly. Study this grid before your interview.

I recommend reducing each story to just a couple of keywords that you can write in each cell. This will make the grid easier to study and remember.

If you're doing a phone interview, you should have this grid out in front of you. When each story has just a couple of keywords to trigger your memory, it will be much easier to give a fluid response than if you're trying to re-read a paragraph.

It may also be useful to extend this grid to "softer" questions, such as conflicts on a team, failures, or times you had to persuade someone. Questions like these are very common outside of strictly software engineer roles, such as dev lead, PM or even testing role. If you are applying for one of these positions, I would recommend making a second grid covering these softer areas.

When answering these questions, you're not just trying to find a story that matches their question. You're telling them about yourself. Think deeply about what each story communicates about you.

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. No one wants to work with someone like that. A better 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 was the most challenging part of that project?

When asked what the most challenging part was, don't say "I had to learn a lot of new languages and technologies." This is the "cop out" answer when you don't know what else to say. It tells the interviewer that nothing was really very hard.

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.

Some questions may come to you during the interview, but you can—and should—prepare questions in advance. Doing research on the company or team may help you with preparing questions.

Questions can be divided into three different categories.

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. "How much of your day do you spend coding?"
2. "How many meetings do you have every week?"
3. "What is the ratio of testers to developers to program managers? What is the interaction like? How does project planning happen on the team?"

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

Insightful Questions

These questions are designed to demonstrate your deep knowledge of programming or technologies, and they also demonstrate your passion for the company or product.

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. Did you come in with a background in this, or what opportunities are there to learn about it?"
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?"

As stated earlier, interviews usually start and end with "chit chat" or "soft skills." This is a time for your interviewer to ask questions about your resume or general questions, and it's also a time for you ask your interviewer questions about the company. This part of the interview is targeted at getting to know you, as well as relaxing you.

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. Consider an example:

- Candidate #1: "I basically did all the hard work for the team."
- Candidate #2: "I implemented the file system, which was considered one of the most challenging components because ..."

Candidate #2 not only sounds more impressive, but she also appears less arrogant.

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. That is, consider something like this: "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.

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 in 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. If you have a tendency to ramble, 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 chipped in during email discussions, and struggled to complete his components.
- **Action:** One day after class, I pulled him aside to speak about the course and then moved the discussion into talking about the project. I asked him open-ended questions about how he felt it was going and which components he was most excited about tackling. He suggested all the easiest components, and yet offered to do the write-up. I realized then that he wasn't lazy—he was actually just really confused about the project and lacked confidence. I worked with him after that to break down the components into smaller pieces, and I made sure to compliment him a lot on his work to boost his confidence.
- **Result:** He was still the weakest member of the team, but he got a lot better. He was able to finish 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 very 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.

Technical Questions VI

You've purchased this book, so you've already gone a long way towards good technical preparation. Nice work!

That said, there are better and worse ways to prepare. Many candidates just read through problems and solutions. That's like trying to learn calculus by reading a problem and its solution. You need to practice solving problems. Memorizing solutions won't help you.

How to Practice a Question

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.* I mean, *really* try to solve it. Many questions are designed to be tough—that's ok! When you're solving a problem, make sure to think about the space and time efficiency. Ask yourself if you could improve the time efficiency by reducing the space efficiency, or vice versa.
2. *Write the code for the algorithm on paper.* You've been coding all your life on a computer, and you've gotten used to the many nice things about it. But, in your interview, you won't have the luxury of syntax highlighting, code completion, or compiling. Mimic this situation 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 in the real interview.

In addition, mock interviews are extremely useful. CareerCup.com offers mock interviews with actual Microsoft, Google and Amazon employees, but you can also practice with friends. You and your friend can trade 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.

What You Need To Know

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
Binary Trees	Depth First Search	Singleton Design Pattern
Tries	Binary Search	Factory Design Pattern
Stacks	Merge Sort	Memory (Stack vs. Heap)
Queues	Quick Sort	Recursion
Vectors / ArrayLists	Tree Insert / Find / e.t.c.	Big-O Time
Hash Tables		

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

For the data structures and algorithms, be sure to practice implementing them from scratch. You might be asked to implement one directly, or you might be asked to implement a modification of one. Either way, the more comfortable you are with implementations, the better.

In particular, hash tables are an extremely important topic. You will find that you use them frequently in solving interview questions.

Powers of 2 Table

Some people have already committed this to memory, but if you haven't, you should before your interview. The table comes in handy often in scalability questions in computing how much space a set of data will take up.

Power of 2	Exact Value (X)	Approx. Value	X Bytes into MB, GB, e.t.c.
7	128		
8	256		
10	1024	1 thousand	1 K
16	65,536		64 K
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

Using this table, you could easily compute, for example, that a hash table mapping every 32-bit integer to a boolean value could fit in memory on a single 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.

Do you need to know details of C++, Java, or other programming languages?

While I personally never liked asking these sorts of questions (e.g., "what is a vtable?"), many interviewers do ask them.

For big companies like Microsoft, Google, and Amazon, I wouldn't stress too much about these questions. You should understand the main concepts of any language you claim to know, but you should focus more on data structures and algorithms preparation.

At smaller companies and non-software companies, these questions can be more important. Look up your company on CareerCup.com to decide for yourself. If your company isn't listed, find a similar company as a reference. In general, start-ups tend to look for skills in "their" programming language.

Interviews are supposed to be difficult. If you don't get every—or any—answer immediately, that's ok! In fact, in my experience, maybe only 10 people out of the 120+ that I've interviewed have gotten my favorite questions right instantly.

So when you get a hard question, don't panic. Just start talking aloud about how you would solve it. Show your interviewer how you're tackling the problem so that he doesn't think you're stuck.

And, one more thing: you're not done until the interviewer says you're done! What I mean here is that when you come up with an algorithm, start thinking about the problems accompanying it. When you write code, start trying to find bugs. If you're anything like the other 110 candidates that I've interviewed, you probably made some mistakes.

Five Steps to a Technical Question

A technical interview question can be solved utilizing a five step approach:

1. Ask your interviewer questions to resolve ambiguity.
2. Design an Algorithm.
3. Write pseudocode first, but make sure to tell your interviewer that you'll eventually write "real" code.
4. Write your code at a moderate pace.
5. Test your code and *carefully* fix any mistakes.

We will go through each of these steps in more detail below.

Step 1: Ask Questions

Technical problems are more ambiguous than they might appear, so make sure to ask questions to resolve anything that might be unclear. You may eventually wind up with a very different—or much easier—problem than you had initially thought. In fact, many interviewers (especially at Microsoft) will specifically test to see if you ask good questions.

Good questions might be ones like: What are the data types? How much data is there? What assumptions do you need to solve the problem? Who is the user?

Example: "Design an algorithm to sort a list."

- Question: What sort of list? An array? A linked list?

  Answer: An array.
- Question: What does the array hold? Numbers? Characters? Strings?

  Answer: Numbers.
- Question: And are the numbers integers?

Answer: Yes.

- Question: Where did the numbers come from? Are they IDs? Values of something?

  Answer: They are the ages of customers.

- Question: And how many customers are there?

  Answer: About a million.

We now have a pretty different problem: sort an array containing a million integers between 0 and 130 (a reasonable maximum age). How do we solve this? Just create an array with 130 elements and count the number of ages at each value.

Step 2: Design an Algorithm

Designing an algorithm can be tough, but our Five Approaches to Algorithms (see next section) can help you out. While you're designing your algorithm, don't forget to ask yourself the following:

- What are its space and time complexity?
- What happens if there is a lot of data?
- Does your design cause other issues? For example, if you're creating a modified version of a binary search tree, did your design impact the time for insert, find, or delete?
- If there are other issues or limitations, did you make the right trade-offs? For which scenarios might the trade-off be less optimal?
- If they gave you specific data (e.g., mentioned that the data is ages, or in sorted order), have you leveraged that information? Usually there's a reason that an interviewer gave you specific information.

It's perfectly acceptable, and even recommended, to first mention the brute force solution. You can then continue to optimize from there. You can assume that you're always expected to create the most optimal solution possible, but that doesn't mean that the first solution you give must be perfect.

Step 3: Pseudocode

Writing pseudocode first can help you outline your thoughts clearly and reduce the number of mistakes you commit. But, make sure to tell your interviewer that you're writing pseudocode first and that you'll follow it up with "real" code. Many candidates will write pseudocode in order to "escape" writing real code, and you certainly don't want to be confused with those candidates.

Step 4: Code

You don't need to rush through your code; in fact, this will most likely hurt you. Just go

at a nice, slow, methodical pace. Also, remember this advice:

- *Use Data Structures Generously:* Where relevant, use a good data structure or define your own. For example, if you're asked a problem involving finding the minimum age for a group of people, consider defining a data structure to represent a Person. This shows your interviewer that you care about good object-oriented design.
- *Don't Crowd Your Coding:* This is a minor thing, but it can really help. When you're writing code on a whiteboard, start in the upper left hand corner rather than in the middle. This will give you plenty of space to write your answer.

Step 5: Test

Yes, you need to test your code! Consider testing for:

- Extreme cases: 0, negative, null, maximums, minimums.
- User error: What happens if the user passes in null or a negative value?
- General cases: Test the normal case.

If the algorithm is complicated or highly numerical (bit shifting, arithmetic, etc.), consider testing while you're writing the code rather than just at the end.

When you find mistakes (which you will), carefully think through *why* the bug is occurring before fixing the mistake. You do not want to be seen as a "random fixer" candidate: one who, for example, finds that their function returns `true` instead of `false` for a particular value, and so just flips the return value and tests to see if the function works. This might fix the issue for that particular case, but it inevitably creates several new issues.

When you notice problems in your code, really think deeply about why your code failed before fixing the mistake. You'll create beautiful, clean code much, much faster.

There's no surefire approach to solving a tricky algorithm problem, but the approaches below can be useful. Keep in mind that the more problems you practice, the easier it will be to identify which approach to use.

The five approaches below can be "mixed and matched." That is, after you've applied "Simplify & Generalize," you may want to try "Pattern Matching" next.

Approach I: Examplify

We'll start with an approach you are probably familiar with, even if you've never seen it labeled. Under Examplify, you write out specific examples of the problem and see if you can derive a general rule from there.

Example: Given a time, calculate the angle between the hour and minute hands.

Let's start with an example like 3:27. We can draw a picture of a clock by selecting where the 3 hour hand is and where the 27 minute hand is.

For the below solution, we'll assume that `h` is the hour and `m` is the minute. We'll also assume that the hour is specified as an integer between 0 and 23, inclusive.

By playing around with these examples, we can develop a rule:

- Angle between the minute hand and 12 o'clock: `360 * m / 60`
- Angle between the hour hand and 12 o'clock: `360 * (h % 12) / 12 + 360 * (m / 60) * (1 / 12)`
- Angle between hour and minute: `(hour angle - minute angle) % 360`

By simple arithmetic, this reduces to `(30h - 5.5m) % 360`.

Approach II: Pattern Matching

Under the Pattern Matching approach, we consider what problems the algorithm is similar to and try to modify the solution to the related problem to develop an algorithm for this problem.

Example: A sorted array has been rotated so that the elements might appear in the order 3 4 5 6 7 1 2. How would you find the minimum element? You may assume that the array has all unique elements.

There are two problems that jump to mind as similar:

- Find the minimum element in an array.
- Find a particular element in a sorted array (i.e., binary search).

The Approach

Finding the minimum element in an array isn't a particularly interesting algorithm (you

could just iterate through all the elements), nor does it use the information provided (that the array is sorted). It's unlikely to be useful here.

However, binary search is very applicable. You know that the array is sorted, but rotated. So, it must proceed in an increasing order, then reset, and increase again. The minimum element is the "reset" point.

If you compare the middle and last element (6 and 2), you will know the reset point must be between those values, since `MID > RIGHT`. This wouldn't be possible unless the array "reset" between those values.

If `MID` were less than `RIGHT`, then either the reset point is on the left half, or there is no reset point (the array is truly sorted). Either way, the minimum element could be found there.

We can continue to apply this approach, dividing the array in half in a manner much like binary search. We will eventually find the minimum element (or the reset point).

Approach III: Simplify and Generalize

With Simplify and Generalize, we implement a multi-step approach. First, we change a constraint such as the data type or amount of data. Doing this helps us simplify the problem. Then, we solve this new simplified version of the problem. Finally, once we have an algorithm for the simplified problem, we generalize the problem and try to adapt the earlier solution 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.

Approach IV: Base Case and Build

Base Case and Build is a great approach for certain types of problems. With Base Case and Build, we solve the problem first for a base case (e.g., `n = 1`). This usually means just recording the correct result. Then, we try to solve the problem for `n = 2`, assuming that you have the answer for `n = 1`. Next, we try to solve it for `n = 3`, assuming that you have the answer for `n = 1` and `n = 2`,

Eventually, we can build a solution that can always compute the result for N if we know the correct result for N-1. It may not be until N equals 3 or 4 that we get an instance that's interesting enough to try to build the solution based on the previous result.

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") = insert "c" into all locations of all strings in P("ab")
P("abc") = insert "c" into all locations of all strings in
{"ab","ba"}
P("abc") = 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 $s_1 \ldots s_n$ by "chopping off" the last character and generating all permutations of $s_1 \ldots s_{n-1}$. Once we have the list of all permutations of $s_1 \ldots s_{n-1}$, we iterate through this list, and 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.

Approach V: 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.

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 $Ax^a + Bx^b + \ldots$ (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 kth 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] * x^exponents[i]`.

Under this implementation, if `coefficients[p] = k` and `exponents[p] = m`, then the pth 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, since you need to return two arrays.

```
1  ??? sum(double[] coeffs1, double[] expon1,
2          double[] coeffs2, double[] expon2) {
3     ...
4  }
```

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  public boolean compareBinToHex(String binary, String hex) {
2     int n1 = convertToBase(binary, 2);
3     int n2 = convertToBase(hex, 16);
4     if (n1 < 0 || n2 < 0) {
5        return false;
```

```
    } else {
      return n1 == n2;
    }
  }

  public int digitToValue(char c) {
    if (c >= '0' && c <= '9') return c - '0';
    else if (c >= 'A' && c <= 'F') return 10 + c - 'A';
    else if (c >= 'a' && c <= 'f') return 10 + c - 'a';
    return -1;
  }

  public int convertToBase(String number, int base) {
    if (base < 2 || (base > 10 && base != 16)) return -1;
    int value = 0;
    for (int i = number.length() - 1; i >= 0; i--) {
      int digit = digitToValue(number.charAt(i));
      if (digit < 0 || digit >= base) {
        return -1;
      }
      int exp = number.length() - 1 - i;
      value += digit * Math.pow(base, exp);
    }
    return value;
  }
```

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 `convertToBase` 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:

```
public void swapMinMax(int[] array) {
  int minIndex = 0;
  for (int i = 1; i < array.length; i++) {
    if (array[i] < array[minIndex]) {
      minIndex = i;
    }
  }

  int maxIndex = 0;
  for (int i = 1; i < array.length; i++) {
    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   public static int getMinIndex(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      return minIndex;
9   }
10
11  public static int getMaxIndex(int[] array) {
12     int maxIndex = 0;
13     for (int i = 1; i < array.length; i++) {
14        if (array[i] > array[maxIndex]) {
15           maxIndex = i;
16        }
17     }
18     return maxIndex;
19  }
20
21  public static void swap(int[] array, int m, int n) {
22     int temp = array[m];
23     array[m] = array[n];
24     array[n] = temp;
25  }
26
27  public static void swapMinMaxBetter(int[] array) {
28     int minIndex = getMinIndex(array);
29     int maxIndex = getMaxIndex(array);
30     swap(array, minIndex, maxIndex);
31  }
```

While the non-modular isn't particularly awful, the nice thing about the modular code is that it's easily testable since 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 constants instead of variables 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 to this. 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  public 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.

The Offer and Beyond

VII

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.

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

How you decline an offer matters. 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 move to another 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, offer a reason that is non-offensive and inarguable. For example, if you were declining a big company for a start-up, you could explain that you feel a start-up is the right choice for you at this time. The big company can't suddenly "become" a start-up, so they can't argue about your reasoning.

Handling Rejection

The big tech companies reject around 80% of their interview candidates and recognize that interviews are not a perfect test of skills. 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, you should see this as a setback but not a life sentence. 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.

Finding out why you were rejected is incredibly difficult. Recruiters are unlikely to reveal the reason, though you might have slightly better luck if you instead ask where you should focus your preparation. You can try to reflect on your performance yourself, but in my experience, candidates can rarely properly analyze their performance. You may think that you struggled on a question, but it's all relative; did you struggle more or less than other candidates on the same question? Instead, just remember that candidates are typically rejected because of their coding and algorithm skills, and so you should focus your preparation there.

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:* If you've received offers in multiple locations, do not overlook the impact of cost of living differences. Silicon Valley, for example, is about 20 to 30% more expensive than Seattle (in part due to a 10% California state income tax). A variety of online sources can estimate the cost of living difference.
- *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're working for Microsoft in Silicon Valley and wish to leave, you'll get your choice of almost any other company. In Microsoft-Seattle, however, you're limited to Amazon, Google and a handful of other smaller tech companies. If you go to AOL in Dulles, Virginia, your options are even more limited. You may find yourself forced to stay at a company simply because there's nowhere else to go without uprooting your whole life.

Company Stability

Everyone's situation is a little bit different, but I typically encourage candidates to not focus too much on the stability of a company. If they do let you go, you can usually find an offer from an equivalent company. The question for you to answer is: what will happen if you get laid off? Do you feel reasonably comfortable in your ability to find a new job?

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 factor, 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), you'll have an opportunity to find a team and product that matches you well.

In late 2010, 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.

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 nothing to lose.
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 that you'll decline their offer much more real.
3. *Have a Specific "Ask":* It's much 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.

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 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.

Interview Questions VIII

*Join us at **www.CrackingTheCodingInterview.com** to download full, compilable Java / Eclipse solutions, discuss problems from this book with other readers, report issues, view this book's errata, post your resume, and seek additional advice.*

Data Structures

Interview Questions and Advice

1

Arrays and Strings

Hopefully, all readers of this book are familiar with what arrays and strings are, 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. In a very simple implementation of a hash table, the hash table has an underlying array and a *hash function*. When you want to insert an object and its key, the hash function maps the key to an integer, which indicates the index in the array. The object is then stored at that index.

Typically, though, this won't quite work right. In the above implementation, the hash value of all possible keys must be unique, or we might accidentally overwrite data. The array would have to be extremely large—the size of all possible keys—to prevent such "collisions."

Instead of making an extremely large array and storing objects at index `hash(key)`, we can make the array much smaller and store objects in a linked list at index `hash(key) % array_length`. To get the object with a particular key, we must search the linked list for this key.

Alternatively, we can implement the hash table with a binary search tree. We can then guarantee an `O(log n)` lookup time, since we can keep the tree balanced. Additionally, we may use less space, since a large array no longer needs to be allocated in the very beginning.

Prior to your interview, we recommend you practice both implementing and using hash tables. They are one of the most common data structures for interviews, and it's almost

a sure bet that you will encounter them in your interview process.

Below is a simple Java example of working with a hash table.

```
1  public HashMap<Integer, Student> buildMap(Student[] students) {
2     HashMap<Integer, Student> map = new HashMap<Integer, Student>();
3     for (Student s : students) map.put(s.getId(), s);
4     return map;
5  }
```

Note that while the use of a hash table is sometimes explicitly required, more often than not, it's up to you to figure out that you need to use a hash table to solve the problem.

ArrayList (Dynamically Resizing Array)

An ArrayList, or a dynamically resizing array, 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 time is still O(1).

```
1  public 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  }
```

StringBuffer

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  public 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 $O(x + 2x + \ldots + nx)$. This reduces to $O(xn^2)$. (Why isn't it $O(xn^n)$? Because $1 + 2 + \ldots + n$ equals $n(n+1)/2$, or $O(n^2)$.)

`StringBuffer` can help you avoid this problem. `StringBuffer` simply creates an array of all the strings, copying them back to a string only when necessary.

```
1  public String joinWords(String[] words) {
2     StringBuffer sentence = new StringBuffer();
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 `StringBuffer`.

Interview Questions

1.1 Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

pg 172

1.2 Implement a function void reverse(char* str) in C or C++ which reverses a null-terminated string.

pg 173

1.3 Given two strings, write a method to decide if one is a permutation of the other.

pg 174

1.4 Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end of the string 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    "`

Output: `"Mr%20John%20Smith"`

pg 175

1.5 Implement a method to perform basic string compression using the counts of repeated characters. For example, the string `aabcccccaaa` would become `a2b1c5a3`. If the "compressed" string would not become smaller than the original string, your method should return the original string.

pg 176

1.6 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?

pg 179

1.7 Write an algorithm such that if an element in an `MxN` matrix is 0, its entire row and column are set to 0.

pg 180

1.8 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 181

Additional Questions: Bit Manipulation (#5.7), Object-Oriented Design (#8.10), Recursion (#9.3), Sorting and Searching (#11.6), C++ (#13.10), Moderate (#17.7, #17.8, #17.14)

2

Linked Lists

Because of the lack of constant time access and the frequency of recursion, linked list questions can stump many candidates. The good news is that there is comparatively little variety in linked list questions, and many problems are merely variants of well-known questions.

Linked list problems rely so much on the fundamental concepts, so it is essential that you can implement a linked list from scratch. We have provided the code below.

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 }
```

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 are implementing 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 $a_1->a_2->...->a_n->b_1->b_2->...->b_n$ and you wanted to rearrange it into $a_1->b_1->a_2->b_2->...->a_n->b_n$. 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 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 184

2.2 Implement an algorithm to find the kth to last element of a singly linked list.

pg 185

2.3 Implement an algorithm to delete a node in the 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

Result: nothing is returned, but the new linked list looks like a->b->d->e

pg 187

2.4 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.

pg 188

2.5 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-> 1 -> 6) + (5 -> 9 -> 2). That is, 617 + 295.

Output: 2 -> 1 -> 9. That is, 912.

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

EXAMPLE

Input: (6 -> 1 -> 7) + (2 -> 9 -> 5). That is, 617 + 295.

Output: 9 -> 1 -> 2. That is, 912.

pg 190

2.6 Given a circular linked list, implement an algorithm which 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 193

2.7 Implement a function to check if a linked list is a palindrome.

pg 196

Additional Questions: Trees and Graphs (#4.4), Object-Oriented Design (#8.10), Scalability and Memory Limits (#10.7), Moderate (#17.13)

3

Stacks and Queues

Like linked list questions, 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

Recall that a stack uses the LIFO (last-in first-out) ordering. That is, like a stack of dinner plates, the most recent item added to the stack is the first item to be removed.

We have provided simple sample code to implement a stack. Note that a stack can also be implemented using a linked list. In fact, they are essentially the same thing, except that a stack usually prevents the user from "peeking" at items below the top node.

```
1  class Stack {
2     Node top;
3
4     Object pop() {
5        if (top != null) {
6           Object item = top.data;
7           top = top.next;
8           return item;
9        }
10       return null;
11    }
12
13    void push(Object item) {
14       Node t = new Node(item);
15       t.next = top;
16       top = t;
17    }
18
19    Object peek() {
20       return top.data;
21    }
22 }
```

Implementing a Queue

A queue implements FIFO (first-In first-out) ordering. Like a line or queue at a ticket stand, items are removed from the data structure in the same order that they are added.

A queue can also be implemented with a linked list, with new items added to the tail of the linked list.

```
1  class Queue {
2     Node first, last;
3
4     void enqueue(Object item) {
5        if (first == null) {
6           last = new Node(item);
7           first = last;
8        } else {
9           last.next = new Node(item);
10          last = last.next;
11       }
12    }
13
14    Object dequeue() {
15       if (first != null) {
16          Object item = first.data;
17          first = first.next;
18          return item;
19       }
20       return null;
21    }
22 }
```

Interview Questions

3.1 Describe how you could use a single array to implement three stacks.

pg 202

3.2 How would you design a stack which, in addition to push and pop, also has a function `min` which returns the minimum element? `Push`, `pop` and `min` should all operate in O(1) time.

pg 206

3.3 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 208

3.4 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 the next tower.

(3) A disk can only be placed on top of a larger disk.

Write a program to move the disks from the first tower to the last using stacks.

pg 211

3.5 Implement a `MyQueue` class which implements a queue using two stacks.

pg 213

3.6 Write a program to sort a stack in ascending order (with biggest items on top). You may use at most one additional stack to hold items, 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 215

3.7 An animal shelter holds only dogs and cats, and 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 217

Additional Questions: Linked Lists (#2.7), Mathematics and Probability (#7.7)

4

Trees and Graphs

Many interviewees find trees and graphs problems to be some of the trickiest. Searching the data structure is more complicated than in a linearly organized data structure like 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.

Potential Issues to Watch Out For

Trees and graphs questions are ripe for ambiguous details and incorrect assumptions. Be sure to watch out for the following issues and seek clarification when necessary.

Binary Tree vs. Binary Search Tree

When given a binary tree question, many candidates assume that the interviewer means binary *search* tree. Be sure to ask whether or not the tree is a binary search tree. A binary search tree imposes the condition that, for all nodes, the left children are less than or equal to the current node, which is less than all the right nodes.

Balanced vs. Unbalanced

While many trees are balanced, not all are. Ask your interviewer for clarification on this issue. If the tree is unbalanced, you should describe your algorithm in terms of both the average and the worst case time. Note that there are multiple ways to balance a tree, and balancing a tree implies only that the depth of subtrees will not vary by more than a certain amount. It does not mean that the left and right subtrees are exactly the same size.

Full and Complete

Full and complete trees are trees in which all leaves are at the bottom of the tree, and all non-leaf nodes have exactly two children. Note that full and complete trees are *extremely* rare, as a tree must have exactly 2^n - 1 nodes to meet this condition.

Binary Tree Traversal

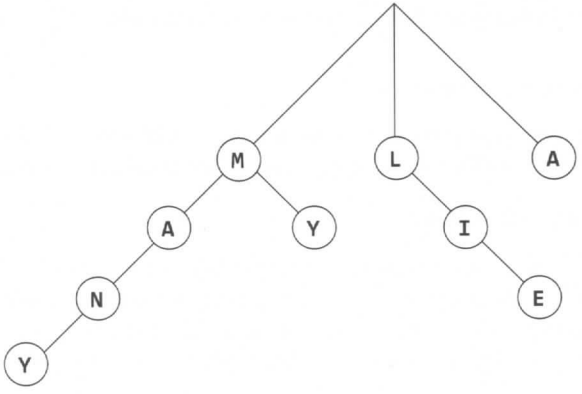
Prior to your interview, you should be comfortable implementing in-order, post-order, and pre-order traversal. The most common of these, in-order traversal, works by visiting the left side, then the current node, then the right.

Tree Balancing: Red-Black Trees and AVL Trees

Though learning how to implement a balanced tree may make you a better software engineer, it's very rarely asked during an interview. You should be familiar with the runtime of operations on balanced trees, and vaguely familiar with how you might balance a tree. The details, however, are probably unnecessary for the purposes of an interview.

Tries

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. A simple trie might look something like:



Graph Traversal

While most candidates are reasonably comfortable with binary tree traversal, graph traversal can prove more challenging. Breadth First Search is especially difficult.

Note that Breadth First Search (BFS) and Depth First Search (DFS) are usually used in different scenarios. DFS is typically the easiest if we want to visit every node in the graph, or at least visit every node until we find whatever we're looking for. However, if we have a very large tree and want to be prepared to quit when we get too far from the original node, DFS can be problematic; we might search thousands of ancestors of the node, but never even search all of the node's children. In these cases, BFS is typically preferred.

Depth First Search (DFS)

In DFS, we visit a node `r` and then iterate through each of `r`'s adjacent nodes. When visiting a node `n` that is adjacent to `r`, we visit all of `n`'s adjacent nodes before going

on to r's other adjacent nodes. That is, n is exhaustively searched before r moves on to searching its other children.

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 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     foreach (Node n in root.adjacent) {
6        if (n.visited == false) {
7           search(n);
8        }
9     }
10 }
```

Breadth First Search (BFS)

BFS is considerably less intuitive, and most interviewees struggle with it unless they are already familiar with the implementation.

In BFS, we visit each of a node r's adjacent nodes before searching any of r's "grandchildren." An iterative solution involving a queue usually works best.

```
1  void search(Node root) {
2     Queue queue = new Queue();
3     root.visited = true;
4     visit(root);
5     queue.enqueue(root); // Add to end of queue
6
7     while (!queue.isEmpty()) {
8        Node r = queue.dequeue(); // Remove from front of queue
9        foreach (Node n in r.adjacent) {
10          if (n.visited == false) {
11             visit(n);
12             n.visited = true;
13             queue.enqueue(n);
14          }
15       }
16    }
17 }
```

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.

Interview Questions

4.1 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 220

4.2 Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

pg 221

4.3 Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

pg 222

4.4 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 224

4.5 Implement a function to check if a binary tree is a binary search tree.

pg 225

4.6 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 229

4.7 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 230

4.8 You have two very large binary trees: T1, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide 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.

pg 235

4.9 You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum to a given value. The path does not need to start or end at the root or a leaf.

pg 237

Additional Questions: Scalability and Memory Limits (#10.2, #10.5), Sorting and Searching (#11.8), Moderate (#17.13, #17.14), Hard (#18.6, #18.8, #18.9, #18.10, #18.13)

Concepts and Algorithms

Interview Questions and Advice

5

Bit Manipulation

Bit manipulation is used in a variety of problems. Sometimes, the question explicitly calls for bit manipulation, while at other times, it's simply a useful technique to optimize your code. You should be comfortable with bit manipulation by hand, as well as with code. But be very careful; it's easy to make little mistakes on bit manipulation problems. Make sure to test your code thoroughly after you're done writing it, or even while writing it.

Bit Manipulation By Hand

The practice exercises below will be useful if you have the oh-so-common fear of bit manipulation. When you get stuck or confused, try to work these operations through as a base 10 number. You can then apply the same process to a binary number.

Remember that ^ indicates an XOR operation, and ~ is a not (negation) operation. For simplicity, assume that these are four-bit numbers. The third column can be solved manually, or with "tricks" (described below).

`0110 + 0010`	`0011 * 0101`	`0110 + 0110`
`0011 + 0010`	`0011 * 0011`	`0100 * 0011`
`0110 - 0011`	`1101 >> 2`	`1101 ^ (~1101)`
`1000 - 0110`	`1101 ^ 0101`	`1011 & (~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. Since `0100` equals 4, we are just multiplying `0011` by 4. Multiplying by 2^n just shifts a number by n. 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^(~a)` will be a sequence of 1s.

4. An operation like x & (~0 << n) clears the n rightmost bits of x. The value ~0 is simply a sequence of 1s, so by shifting it left by n, we have a bunch of ones followed by n zeros. By doing an AND with x, we clear the rightmost n bits of x.

For more problems, open the Windows calculator and go to View > Programmer. From this application, you can perform many binary operations, including AND, XOR, and shifting.

Bit Facts and Tricks

In solving bit manipulation problems, it's useful to understand the following facts. 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.

```
x ^ 0s = x        x & 0s = 0        x | 0s = x
x ^ 1s = ~x       x & 1s = x        x | 1s = 1s
 x ^ x = 0         x & x = x         x | x = x
```

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.

Common Bit Tasks: Get, Set, Clear, and Update Bit

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 do:

```
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 do:

```
1  int clearBitsIthrough0(int num, int i) {
2     int mask = ~((1 << (i+1)) - 1);
3     return num & mask;
4  }
```

Update Bit

This method merges the approaches of `setBit` and `clearBit`. First, we 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, int v) {
2     int mask = ~(1 << i);
3     return (num & mask) | (v << i);
4  }
```

Interview Questions

5.1 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 242

5.2 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 243

5.3 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 244

5.4 Explain what the following code does: `((n & (n-1)) == 0)`.

pg 250

5.5 Write a function to determine the number of bits required to convert integer A to integer B.

EXAMPLE

Input: 31, 14

Output: 2

pg 250

5.6 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 251

5.7 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?

pg 252

5.8 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 `drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y)` which draws a horizontal line from `(x1, y)` to `(x2, y)`.

pg 255

Additional Questions: Arrays and Strings (#1.1, #1.7), Recursion (#9.4, #9.11), Scalability and Memory Limits (#10.3, #10.4), C++ (#13.9), Moderate (#17.1, #17.4), Hard (#18.1)

6

Brain Teasers

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 a brain teaser. Why? Because no one can agree on a definition of what a brain teaser is.

The good news is that if you are asked a brain teaser, 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 brain teasers even have their foundations in mathematics or computer science.

We'll go through some common approaches for tackling brain teasers.

Start Talking

Don't panic when you get a brain teaser. 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 `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-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 brain teasers 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 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 five approaches for solving algorithm questions. Brain teasers are often nothing more than algorithm questions with the technical aspects removed. Examplify, Simplify and Generalize, Pattern Matching, and Base Case and Build can be especially useful.

Interview Questions

6.1 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 258

6.2 There is an 8x8 chess board 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 258

6.3 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 259

6.4 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 260

6.5 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 261

6.6 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 262

7

Mathematics and Probability

Although many mathematical problems given during an interview read as brain teasers, most can be tackled with a logical, methodical approach. They are typically rooted in the rules of mathematics or computer science, and this knowledge can facilitate either solving the problem or validating your solution. We'll cover the most relevant mathematical concepts in this section.

Prime Numbers

As you probably know, every positive integer can be decomposed into a product of primes. For example:

$$84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * \ldots$$

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\y, or `mod(y, x) = 0`), all primes in x's prime factorization must be in y's prime factorization. Or, more specifically:

Let $x = 2^{j0} * 3^{j1} * 5^{j2} * 7^{j3} * 11^{j4} * \ldots$

Let $y = 2^{k0} * 3^{k1} * 5^{k2} * 7^{k3} * 11^{k4} * \ldots$

If x\y, then for all `i`, `ji <= ki`.

In fact, the greatest common divisor of x and y will be:

$$gcd(x, y) = 2^{min(j0, k0)} * 3^{min(j1, k1)} * 5^{min(j2, k2)} * \ldots$$

The least common multiple of x and y will be:

$$lcm(x, y) = 2^{max(j0, k0)} * 3^{max(j1, k1)} * 5^{max(j2, k2)} * \ldots$$

As a fun exercise, stop for a moment and think what would happen if you did `gcd * lcm`:

$$gcd * lcm = 2^{min(j0, k0)} * 2^{max(j0, k0)} * 3^{min(j1, k1)} * 3^{max(j1, k1)} * \ldots$$

$$= 2^{\min(j0,\ k0)\ +\ \max(j0,\ k0)} * 3^{\min(j1,\ k1)\ +\ \max(j1,\ k1)} * \ldots$$
$$= 2^{j0\ +\ k0} * 3^{j1\ +\ k1} * \ldots$$
$$= 2^{j0} * 2^{k0} * 3^{j1} * 3^{k1} * \ldots$$
$$= xy$$

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` is sufficient because, for every number a which divides n evenly, there is a complement b, where `a * b = n`. If `a > sqrt`, then `b < sqrt` (since `sqrt * sqrt = 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.

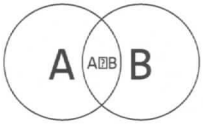
```
boolean[] sieveOfEratosthenes(int max) {
   boolean[] flags = new boolean[max + 1];
   int count = 0;

   init(flags); // Set all flags to true other than 0 and 1
   int prime = 2;

   while (prime <= Math.sqrt(max)) {
      /* Cross off remaining multiples of prime */
      crossOff(flags, prime);

      /* Find next value which is true */
      prime = getNextPrime(flags, prime);

      if (prime >= flags.length) {
         break;
      }
   }

   return flags;
}

void crossOff(boolean[] flags, int prime) {
   /* Cross off remaining multiples of prime. We can start with
    * (prime*prime), because if we have a k * prime, where
    * k < prime, this value would have already been crossed off in
    * a prior iteration. */
   for (int i = prime * prime; i < flags.length; i += prime) {
      flags[i] = false;
   }
}

int getNextPrime(boolean[] flags, int prime) {
   int next = prime + 1;
   while (next < flags.length && !flags[next]) {
      next++;
   }
   return next;
}
```

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 and B) = P(B 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:

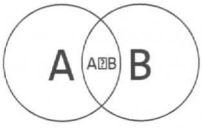
```
P(x is even and x <= 5)
    = P(x is even given x <= 5) P(x <= 5)
    = (2/5) * (1/2)
    = 1/5
```

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 or B) = P(A) + P(B) - P(A 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:

```
P(x is even or x <=5)
    = P(x is even) + P(x <= 5) - P(x is even and x <= 5)
    = (1/2) + (1/2) - (1/5)
    = 4/5
```

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 and B) = P(A) P(B)`. This rule simply comes from recognizing that `P(B 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 or B) = P(A) + P(B)`. This is because `P(A and B) = 0`, so this term is removed from the earlier `P(A 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.

Things to Watch Out For

1. Be careful with the difference in precision between floats and doubles.
2. Don't assume that a value (such as the slope of a line) is an `int` unless you've been told so.
3. Unless otherwise specified, do not assume that events are independent (or mutually exclusive). You should be careful, therefore, of blindly multiplying or adding probabilities.

Interview Questions

7.1 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 264

7.2 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 265

7.3 Given two lines on a Cartesian plane, determine whether the two lines would intersect.

pg 266

7.4 Write methods to implement the multiply, subtract, and divide operations for integers. Use only the add operator.

pg 267

7.5 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.

pg 269

7.6 Given a two-dimensional graph with points on it, find a line which passes the most number of points.

pg 271

7.7 Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7.

pg 274

Additional Questions: Moderate (#17.11), Hard (#18.2)

8

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 Object-Oriented Design Questions

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 especially useful for interviews, so we will cover them here.

There are far more design patterns than this book could possibly discuss. A fantastic way to improve your software engineering skills is to pick up a book that focuses on this area specifically.

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     public static Restaurant getInstance() {
4        if (_instance == null) {
5           _instance = new Restaurant();
6        }
7        return _instance;
8     }
9  }
```

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

8.1 Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

pg 280

8.2 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 283

8.3 Design a musical jukebox using object-oriented principles.

pg 286

8.4 Design a parking lot using object-oriented principles.

pg 289

8.5 Design the data structures for an online book reader system.

pg 292

8.6 Implement a 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 pieces, returns true if the two pieces belong together.

pg 296

8.7 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 300

8.8 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 305

8.9 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 308

8.10 Design and implement a hash table which uses chaining (linked lists) to handle collisions.

pg 311

Additional Questions: Threads and Locks (#16.3)

9

Recursion and Dynamic Programming

While there is a wide variety of recursive problems, many follow similar patterns. A good hint that a problem is recursive is that it can be built off of sub-problems.

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..."; etc..

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 solutions to sub-problems. 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 have to do something more complicated.

You should consider both bottom-up and top-down recursive solutions. The Base Case and Build approach works quite well for recursive problems.

Bottom-Up Recursion

Bottom-up recursion 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, and 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.

Top-Down Recursion

Top-Down Recursion can be more complex, but it's sometimes necessary for problems. In these problems, we think about how we can divide the problem for case `N` into sub-problems. Be careful of overlap between the cases.

Dynamic Programming

Dynamic programming (DP) problems are rarely asked because, quite simply, they're too difficult for a 45-minute interview. Even good candidates would generally do so poorly on these problems that it's not a good evaluation technique.

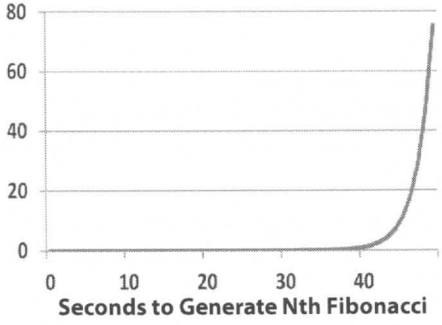
If you're unlucky enough to get a DP problem, you can approach it much the same way as a recursion problem. The difference is that intermediate results are "cached" for future calls.

Simple Example of Dynamic Programming: Fibonacci Numbers

As a very simple example of dynamic programming, imagine you're asked to implement a program to generate the nth Fibonacci number. 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? Computing the nth Fibonacci number depends on the previous `n-1` numbers. But *each* call does two recursive calls. This means that the runtime is $O(2^n)$. The below graph shows this exponential increase, as computed on a standard desktop computer.



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[] fib = new int[max];
2  int fibonacci(int i) {
3    if (i == 0) return 0;
4    if (i == 1) return 1;
5    if (fib[i] != 0) return fib[i]; // Return cached result.
6    fib[i] = fibonacci(i - 1) + fibonacci(i - 2); // Cache result
7    return fib[i];
8  }
```

While the first recursive one may take over a minute to generate the 50th Fibonacci number on a standard 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.)

Dynamic programming, as you can see, is nothing to be scared of. It's little more than recursion where you cache the results. A good way to approach such a problem is often to implement it as a normal recursive solution, and then to add the caching part.

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 has `O(n)` recursive calls, then it uses `O(n)` memory. Ouch!

All recursive code 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 trade-offs with your interviewer.

Interview Questions

9.1 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 316

9.2 Imagine a robot sitting on the upper left corner of an X by Y grid. The robot can only move in two directions: right and down. How many possible paths are there for the robot to go from `(0,0)` to `(X,Y)`?

FOLLOW UP

Imagine certain spots 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 317

9.3 A magic index in an array `A[0...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 319

9.4 Write a method to return all subsets of a set.

pg 321

9.5 Write a method to compute all permutations of a string.

pg 324

9.6 Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n-pairs of parentheses.

EXAMPLE

Input: 3

Output: `((()))`, `(()())`, `(())()`, `()(())`, `()()()`

pg 325

9.7 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 327

9.8 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 328

9.9 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 331

9.10 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 build the tallest stack possible, where the height of a stack is the sum of the heights of each box.

pg 333

9.11 Given a boolean expression consisting of the symbols `0`, `1`, `&`, `|`, and `^`, 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

Expression: `1^0|0|1`

Desired result: `false (0)`

Output: 2 ways. `1^((0|0)|1)` and `1^(0|(0|1))`.

pg 335

Additional Questions: Linked Lists (#2.2, #2.5, #2.7), Stacks and Queues (#3.3), Trees and Graphs (#4.1, #4.3, #4.4, #4.5, #4.7, #4.8, #4.9), Bit Manipulation (#5.7), Brain Teasers (#6.4), Sorting and Searching (#11.5, #11.6, #11.7, #11.8), C++ (#13.7), Moderate (#17.13, #17.14), Hard (#18.4, #18.7, #18.12, #18.13)

10

Scalability and Memory Limits

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. You don't need any courses in distributed systems, nor do you need any experience in system design. With a bit of practice, any thorough and intelligent software engineer can handle these questions with ease.

The Step-By-Step Approach

Interviewers are not trying to test your knowledge of system design; in fact, interviewers rarely try to test knowledge of anything but the most basic Computer Science concepts. Instead, they are evaluating your ability to break down a tricky problem and to solve problems using what you do know. The following approach works well for many system design problems.

Step 1: Make Believe

Pretend that the data can all fit on one machine and there are no memory limitations. How would you solve the problem? This answer to this question will provide the general outline for your solution.

Step 2: 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 the data up? 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 3: 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 to use (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 2, 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.

What You Need to Know: Information, Strategies and Issues

A Typical System

Though super-computers are still in use, most web-based companies use large systems of interconnected machines. You can almost always assume that you will be working in such a system.

Prior to your interview, you should fill in the following chart. This chart will help you to ballpark how much data a computer can store.

Component	Typical Capacity / Value
Hard Drive Space	
Memory	
Internet Transfer Latency	

Dividing Up Lots of Data

Though we can sometimes increase hard drive space in a computer, there comes a point where data simply must be divided up across machines. The question, then, is what data belongs on which machine? There are a few strategies for this.

- *By Order of Appearance:*

  We could simply divide up data by order of appearance. That is, as new data comes in, we wait for our current machine to fill up before adding an additional machine. This has the advantage of never using more machines than are necessary. However, depending on the problem and our data set, our lookup table may be more complex and potentially very large.

- *By Hash Value:*

  An alternative approach is to store the data on the machine corresponding to the hash value of the data. More specifically, we do the following: (1) pick some sort of *key* relating to the data, (2) hash the key, (3) mod the hash value by the number of machines, and (4) store the data on the machine with that value. That is, the data is stored on machine `#[mod(hash(key), N)]`.

  The nice thing about this is that there's no need for a lookup table. Every machine

will automatically know where a piece of data is. The problem, however, is that a machine may get more data and eventually exceed its capacity. In this case, we may need to either shift data around the other machines for better load balancing (which is very expensive), or split this machine's data into two machines (causing a tree-like structure of machines).

- *By Actual Value:*

  Dividing up data by hash value is essentially arbitrary; there is no relationship between what the data represents and which machine stores the data. In some cases, we may be able to reduce system latency by using information about what the data represents.

  For example, suppose you're designing a social network. While people do have friends around the world, the reality is that someone living in Mexico will probably have a lot more friends from Mexico than an average Russian citizen. We could, perhaps, store "similar" data on the same machine so that looking up the Mexican person's friends requires fewer machine hops.

- *Arbitrarily:*

  Frequently, data just gets arbitrarily broken up and we implement a lookup table to identify which machine holds a piece of data. While this does necessitate a potentially large lookup table, it simplifies some aspects of system design and can enable us to do better load balancing.

Example: Find all documents that contain a list of words

Given a list of millions of documents, how would you find all documents that contain a list of words? The words do not need to appear in any particular 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 we can therefore accept the burden of pre-processing.

Step 1

The first step is to forget about the millions of documents and pretend we just had 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 to 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 will move to the next machine.

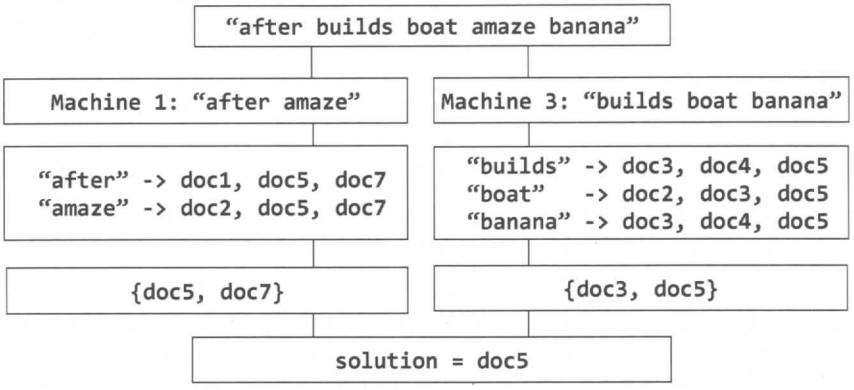
The advantage of this approach is that the lookup table is small and simple (since it must only specify a range of values), and each machine can store a copy of the lookup table. The disadvantage, however, is that if new documents or words are added, we may need to perform an expensive shift of keywords.

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 would look up the documents containing "after" and "amaze," and perform an intersection on these document lists. Machine 3 would do the same for {"banana", "boat", "builds"}, and intersect 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

10.1 Imagine you are building some sort of service that will be called by up to 1000 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 which 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 342

10.2 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 connection, or path, between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 344

10.3 Given an input file with four billion non-negative integers, provide an algorithm to generate an integer which 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.

pg 347

10.4 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?

pg 350

10.5 If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 351

10.6 You have 10 billion URLs. How do you detect the duplicate documents? In this case, assume that "duplicate" means that the URLs are identical.

pg 353

10.7 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 can not 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.

pg 354

Additional Questions: Object-Oriented Design (#8.7)

11

Sorting and Searching

Understanding the common sorting and searching algorithms is incredibly valuable, as many of 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.

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,
16            int high) {
17    /* Copy both halves into a helper array */
18    for (int i = low; i <= high; i++) {
19       helper[i] = array[i];
20    }
21
22    int helperLeft = low;
23    int helperRight = middle + 1;
24    int current = low;
25
26    /* Iterate through helper array. Compare the left and right
27     * half, copying back the smaller element from the two halves
28     * into the original array. */
29    while (helperLeft <= middle && helperRight <= high) {
30       if (helper[helperLeft] <= helper[helperRight]) {
31          array[current] = helper[helperLeft];
32          helperLeft++;
33       } else { // If right element is smaller than left element
34          array[current] = helper[helperRight];
35          helperRight++;
```

```
36        }
37        current++;
38     }
39
40     /* Copy the rest of the left side of the array into the
41      * target array */
42     int remaining = middle - helperLeft;
43     for (int i = 0; i <= remaining; i++) {
44        array[current + i] = helper[helperLeft + i];
45     }
46  }
```

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.

Quick Sort | Runtime: O(n Log(n)) average, O(n²) 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  }
29
```

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

11.1 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.

pg 360

11.2 Write a method to sort an array of strings so that all the anagrams are next to each other.

pg 361

11.3 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)

pg 362

11.4 Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

pg 364

11.5 Given a sorted array of strings which is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: find "ball" in {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Output: 4

pg 364

11.6 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.

pg 365

11.7 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)`

pg 371

11.8 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
```

pg 374

Additional Questions: Arrays and Strings (#1.3), Recursion (#9.3), Moderate (#17.6, #17.12), Hard (#18.5)

12

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

12.1 Find the mistake(s) in the following code:

```
1 unsigned int i;
2 for (i = 100; i >= 0; --i)
3   printf("%d\n", i);
```

pg 378

12.2 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?

pg 378

12.3 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.

pg 379

12.4 How would you load test a webpage without using any test tools?

pg 380

12.5 How would you test a pen?

pg 381

12.6 How would you test an ATM in a distributed banking system?

pg 382

Knowledge Based

Interview Questions and Advice

13

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.

```
1  Person(int a) {
2     id = a;
3  }
```

Fields within the class can also be initialized as follows:

```
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. It is particularly useful when we have constant fields that can only be assigned a value once.

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 {
```

```
    ...
    virtual void aboutMe() {
      cout << "I am a person.";
    }
};

class Student : public Person {
 public:
   void aboutMe() {
      cout << "I am a student.";
   }
};
```

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.

```
class Person {
   int id; // all members are private by default
   char name[NAME_SIZE];
  public:
   virtual void aboutMe() {
      cout << "I am a person." << endl;
   }
   virtual bool addCourse(string s) = 0;
};

class Student : public Person {
  public:
   void aboutMe() {
      cout << "I am a student." << endl;
   }

   bool addCourse(string s) {
      cout << "Added course " << s << " to student." << endl;
      return true;
   }
};

int main() {
   Person * p = new Student();
   p->aboutMe(); // prints "I am a student."
   p->addCourse("History");
   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  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>
2  class ShiftedList {
3    T* array;
4    int offset, size;
5   public:
6    ShiftedList(int sz) : offset(0), size(sz) {
7      array = new T[size];
8    }
9
10   ~ShiftedList() {
11     delete [] array;
12   }
13
14   void shiftBy(int n) {
15     offset = (offset + n) % size;
16   }
```

```
17
18    T getAt(int i) {
19       return array[convertIndex(i)];
20    }
21
22    void setAt(T item, int i) {
23       array[convertIndex(i)] = item;
24    }
25
26 private:
27    int convertIndex(int i) {
28       int index = (i - offset) % size;
29       while (index < 0) index += size;
30       return index;
31    }
32 };
33
34 int main() {
35    int size = 4;
36    ShiftedList<int> * list = new ShiftedList<int>(size);
37    for (int i = 0; i < size; i++) {
38       list->setAt(i, i);
39    }
40    cout << list->getAt(0) << endl;
41    cout << list->getAt(1) << endl;
42    list->shiftBy(1);
43    cout << list->getAt(0) << endl;
44    cout << list->getAt(1) << endl;
45    delete list;
46 }
```

Interview Questions

13.1 Write a method to print the last K lines of an input file using C++.

pg 386

13.2 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?

pg 387

13.3 How do virtual functions work in C++?

pg 388

13.4 What is the difference between deep copy and shallow copy? Explain how you would use each.

pg 389

13.5 What is the significance of the keyword "volatile" in C?

pg 389

13.6 Why does a destructor in base class need to be declared `virtual`?

pg 391

13.7 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`.

pg 391

13.8 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.

pg 392

13.9 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`.

pg 395

13.10 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]`.

pg 396

Additional Questions: Arrays and Strings (#1.2), Linked Lists (#2.7), Testing (#12.1), Java (#14.4), Threads and Locks (#16.3)

14

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..."

final keyword

The `final` keyword in Java has a different meaning depending on whether it is applied to a variable, class or method.

- *Variable:* The value cannot be changed once initialized.
- *Method:* The method cannot be overridden by a subclass.
- *Class:* The class cannot be subclassed.

finally keyword

The `finally` keyword is used in association with a `try/catch` block and guarantees that a section of code will be executed, even if an exception is thrown. The `finally` block will be executed after the `try` and `catch` blocks, but before control transfers back to its origin.

Watch how this plays out in the example below.

```
1  public static String lem() {
2     System.out.println("lem");
3     return "return from lem";
4  }
5
6  public static String foo() {
7     int x = 0;
8     int y = 5;
9     try {
10       System.out.println("start try");
11       int b = y / x;
12       System.out.println("end try");
13       return "returned from try";
14    } catch (Exception ex) {
15       System.out.println("catch");
16       return lem() + " | returned from catch";
17    } finally {
18       System.out.println("finally");
19    }
20 }
21
22 public static void bar() {
23    System.out.println("start bar");
24    String v = foo();
25    System.out.println(v);
26    System.out.println("end bar");
27 }
28
29 public static void main(String[] args) {
30    bar();
31 }
```

The output for this code is the following:

```
1  start bar
2  start try
3  catch
```

```
4  lem
5  finally
6  return from lem | returned from catch
7  end bar
```

Look carefully at lines 3 to 5 in the output. The `catch` block is fully executed (including the function call in the `return` statement), then the `finally` block, and then the function actually returns.

finalize method

The automatic garbage collector calls the `finalize()` method just before actually destroying the object. A class can therefore override the `finalize()` method from the `Object` class in order to define custom behavior during garbage collection.

```
1  protected void finalize() throws Throwable {
2     /* Close open files, release resources, etc */
3  }
```

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  78.75
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 languages, since the rest of the book is filled with Java programming questions.

14.1 In terms of inheritance, what is the effect of keeping a constructor private?

pg 400

14.2 In Java, does the `finally` block get executed if we insert a `return` statement inside the try block of a `try-catch-finally`?

pg 400

14.3 What is the difference between `final`, `finally`, and `finalize`?

pg 400

14.4 Explain the difference between templates in C++ and generics in Java.

pg 401

14.5 Explain what object reflection is in Java and why it is useful.

pg 403

14.6 Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. The class should use a generic type, and should support iteration via the standard `for (Obj o : circularArray)` notation.

pg 404

Additional Questions: Arrays and Strings (#1.4), Object-Oriented Design (#8.10), Threads and Locks (#16.3)

15

Databases

Candidates who profess experience with databases may be asked to demonstrate this knowledge by implementing SQL queries or designing a database for an application. 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

Developers commonly use both the implicit join and the explicit join in SQL queries. Both syntaxes are shown below.

```
1  /* Explicit Join */
2  SELECT CourseName, TeacherName
3  FROM   Courses INNER JOIN Teachers
4  ON     Courses.TeacherID = Teachers.TeacherID
5
6  /* Implicit Join */
7  SELECT CourseName, TeacherName
8  FROM   Courses, Teachers
9  WHERE     Courses.TeacherID = Teachers.TeacherID
```

The two statements above are equivalent, and it's a matter of personal preference which one you choose. For consistency, we will stick to the explicit join.

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,
5            count(StudentCourses.CourseID) as [Cnt]
6    FROM Students LEFT JOIN StudentCourses
7    ON Students.StudentID = StudentCourses.StudentID
8    GROUP BY Students.StudentID
9  ) 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,
2         count(StudentCourses.CourseID) as [Cnt]
3  FROM Students LEFT JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  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,
3         count(StudentCourses.CourseID) as [Cnt]
4  FROM Students LEFT JOIN StudentCourses
5  ON Students.StudentID = StudentCourses.StudentID
6  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 `NULL` values in the `SELECT` statement to convert the `NULL` 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:

Buildings		
BuildingID	BuildingName	BuildingAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

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:

Tenants		
TenantID	TenantName	TenantAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

TenantApartments	
TenantID	ApartmentID

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 below database schema:

Apartments		Buildings		Tenants	
AptID	int	BuildingID	int	TenantID	int
UnitNumber	varchar	ComplexID	int	TenantName	varchar

Apartments		Buildings		Tenants	
BuildingID	int	BuildingName	varchar		
		Address	varchar		

Complexes		AptTenants		Requests	
ComplexID	int	TenantID	int	RequestID	int
ComplexName	varchar	AptID	int	Status	varchar
				AptID	int
				Description	varchar

Note that 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.

15.1 Write a SQL query to get a list of tenants who are renting more than one apartment.

pg 408

15.2 Write a SQL query to get a list of all buildings and the number of open requests (`Requests` in which `status` equals `'Open'`).

pg 408

15.3 Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

pg 409

15.4 What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

pg 409

15.5 What is denormalization? Explain the pros and cons.

pg 411

15.6 Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

pg 412

15.7 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.

pg 412

Additional Questions: Object-Oriented Design (#8.6)

16

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        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 }
18
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) {
11        /* same as before */
12     }
13
14     public static synchronized void bar(String name) {
15        /* same as foo */
16     }
17  }
```

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:

1. *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.)
2. *Hold and Wait:* Processes already holding a resource can request additional resources, without relinquishing their current resources.
3. *No Preemption:* One process cannot forcibly remove another process' resource.
4. *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

16.1 What's the difference between a thread and a process?

pg 416

16.2 How would you measure the time spent in a context switch?

pg 416

16.3 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.

pg 418

16.4 Design a class which provides a lock only if there are no possible deadlocks.

pg 420

16.5 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`.

pg 425

16.6 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?

pg 427

Additional Review Problems

Interview Questions and Advice

17

Moderate

17.1 Write a function to swap a number in place (that is, without temporary variables).

pg 430

17.2 Design an algorithm to figure out if someone has won a game of tic-tac-toe.

pg 431

17.3 Write an algorithm which computes the number of trailing zeros in n factorial.

pg 434

17.4 Write a method which finds the maximum of two numbers. You should not use if-else or any other comparison operator.

pg 436

17.5 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.

pg 438

17.6 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)`

pg 439

17.7 Given any integer, print an English phrase that describes the integer (e.g., "One Thousand, Two Hundred Thirty Four").

pg 442

17.8 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})`

pg 443

17.9 Design a method to find the frequency of occurrences of any given word in a book.

pg 445

17.10 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 END
```

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).

pg 446

17.11 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).

pg 447

17.12 Design an algorithm to find all pairs of integers within an array which sum to a specified value.

pg 450

17.13 Consider a simple node-like data structure called `BiNode`, which has pointers to two other nodes.

```
1 public class BiNode {
2    public BiNode node1, node2;
3    public int data;
4 }
```

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).

pg 451

17.14 Oh, no! You have just completed a lengthy document when you have an unfortunate Find/Replace mishap. You have accidentally removed all spaces, punctuation, and capitalization in the document. A sentence like "`I reset the computer. It still didn't boot!`" would become "`iresetthecomputeritstilldidntboot`". You figure that you can add back in the punctation and capitalization later, once you get the individual words properly separated. Most of the words will be in a dictionary, but some strings, like proper names, will not.

Given a dictionary (a list of words), design an algorithm to find the optimal way of "unconcatenating" a sequence of words. In this case, "optimal" is defined to be the parsing which minimizes the number of unrecognized sequences of characters.

For example, the string "`jesslookedjustliketimherbrother`" would be optimally parsed as "`JESS looked just like TIM her brother`". This parsing has seven unrecognized characters, which we have capitalized for clarity.

pg 455

18

Hard

18.1 Write a function that adds two numbers. You should not use + or any arithmetic operators.

pg 462

18.2 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.

pg 463

18.3 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.

pg 464

18.4 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.)

pg 465

18.5 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?

pg 468

18.6 Describe an algorithm to find the smallest one million numbers in one billion numbers. Assume that the computer memory can hold all one billion numbers.

pg 469

18.7 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`

pg 471

18.8 Given a string `s` and an array of smaller strings `T`, design a method to search `s` for each small string in `T`.

pg 473

18.9 Numbers are randomly generated and passed to a method. Write a program to find and maintain the median value as new values are generated.

pg 474

18.10 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

pg 476

18.11 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.

pg 477

18.12 Given an `NxN` matrix of positive and negative integers, write code to find the submatrix with the largest possible sum.

pg 481

18.13 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.

pg 485