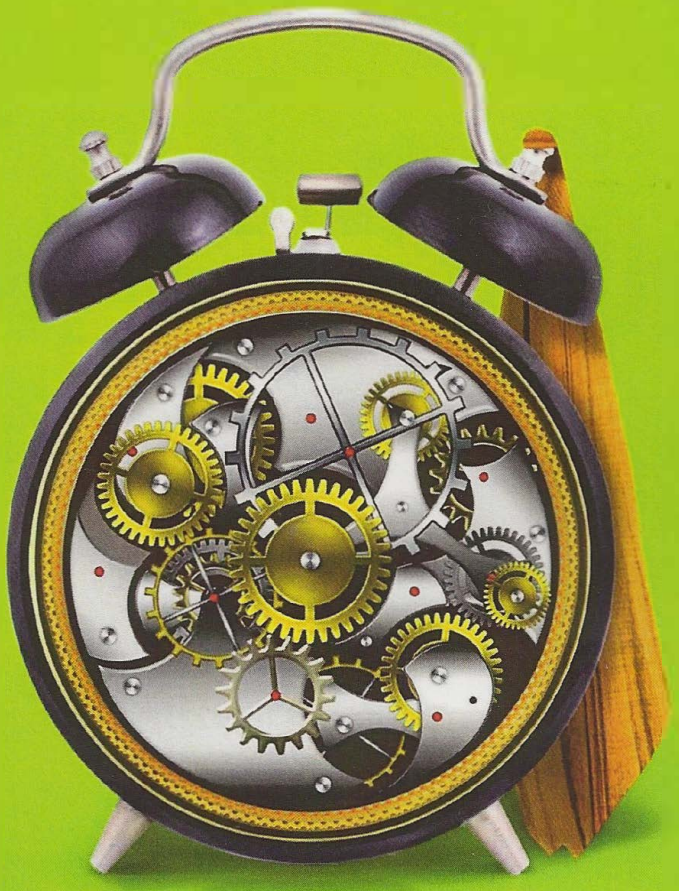
CRACKING *the* CODING INTERVIEW

189 PROGRAMMING QUESTIONS & SOLUTIONS



GAYLE LAAKMANN MCDOWELL
Author of Cracking the PM Interview and Cracking the Tech Career

6TH EDITION

CRACKING THE CODING INTERVIEW, SIXTH EDITION

Copyright © 2015 by CareerCup.

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

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

For more information, contact support@careercup.com.

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

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

Dear Reader,

Let's get the introductions out of the way.

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

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

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

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

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

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

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

Study hard, practice—and good luck!

Gayle L. McDowell

Founder/CEO, CareerCup.com
Author of *Cracking the PM Interview* and *Cracking the Tech Career*

Something's Wrong

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

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

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

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

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

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

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

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

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

My Approach

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

My Passion

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

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

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

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

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

Join us.

Gayle L. McDowell

1

The Interview Process

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

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

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

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

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

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

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

▸ Why?

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

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

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

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

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

False negatives are acceptable.

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

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

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

Problem-solving skills are valuable.

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

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

Basic data structure and algorithm knowledge is useful.

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

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

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

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

Whiteboards let you focus on what matters.

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

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

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

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

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

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

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

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

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

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

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

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

▸ How Questions are Selected

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

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

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

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

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

It's All Relative

If there's no grading system, how are you evaluated? How does an interviewer know what to expect of you?

Good question. The answer actually makes a lot of sense once you understand it.

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

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

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

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

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

Frequently Asked Questions

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

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

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

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

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

11

Behind the Scenes

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

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

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

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

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

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

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

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

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

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

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

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

The Microsoft Interview

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

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

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

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

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

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

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

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

Definitely Prepare:

"Why do you want to work for Microsoft?"

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

What's Unique:

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

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

The Amazon Interview

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

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

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

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

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

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

Definitely Prepare:

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

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

What's Unique:

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

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

The Google Interview

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

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

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

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

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

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

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

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

Definitely Prepare:

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

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

What's Different:

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

▸ The Apple Interview

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

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

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

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

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

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

Definitely Prepare:

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

What's Unique:

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

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

The Facebook Interview

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

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

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

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

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

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

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

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

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

Definitely Prepare:

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

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

What's Unique:

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

The Palantir Interview

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

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

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

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

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

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

Definitely Prepare:

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

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

What's Unique:

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

III

Special Situations

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

Experienced Candidates

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

More experienced engineers might see slightly less focus on algorithm questions—but only slightly

If a company asks algorithm questions to inexperienced candidates, they tend to ask them to experienced candidates too. Rightly or wrongly, they feel that the skills demonstrated in these questions are important for all developers.

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

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

On average, it balances out.

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

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

Testers and SDETs

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

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

- *Prepare the Core Testing Problems:* For example, how would you test a light bulb? A pen? A cash register? Microsoft Word? The Testing chapter will give you more background on these problems.
- *Practice the Coding Questions:* The number one thing that SDETs get rejected for is coding skills. Although coding standards are typically lower for an SDET than for a traditional developer, 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, "Okay, now test it." Even when the question doesn't specifically require this, you should ask yourself, "How would I test this?" Remember: any problem can be an SDET problem!

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

Career Advice

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

Never let your coding skills atrophy.

▸ Product (and Program) Management

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

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

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

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

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

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

▸ Dev Lead and Managers

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

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

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

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

Startups

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

The Application Process

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

Visas and Work Authorization

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

Resume Selection Factors

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

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

The Interview Process

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

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

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

Acquisitions and Acquihires

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

Which startups go through this? And why?

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

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

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

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

How important are these interviews?

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

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

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

Which employees go through the interviews?

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

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

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

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

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

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

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

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

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

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

Your "best" (and worst) employees might surprise you.

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

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

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

Are employees held to the same standards as typical candidates?

Essentially yes, although there is a bit more leeway.

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

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

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

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

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

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

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

What happens to the team after an acquisition?

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

How should you prepare your team for acquisition interviews?

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

You're all in this together.

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

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

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

Some people may be less prepared than others.

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

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

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

Don't wait until the last minute.

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

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

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

▸ For Interviewers

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

Don't actually ask the exact questions in here.

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

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

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

Ask Medium and Hard Problems

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

Look for questions with multiple hurdles.

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

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

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

Use hard questions, not hard knowledge.

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

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

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

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

Avoid "scary" questions.

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

- Math or probability.

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

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

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

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

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

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

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

Offer positive reinforcement.

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

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

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

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

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

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

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

Probe deeper on behavioral questions.

Many candidates are poor at articulating their specific accomplishments.

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

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

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

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

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

Coach your candidates.

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

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

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

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

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

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

If they want silence, give them silence.

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

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

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

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

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

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

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

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

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

IV

Before the Interview

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

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

Getting the Right Experience

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

For current students, this may mean the following:

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

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

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

for fun."

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

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

Writing a Great Resume

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

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

Appropriate Resume Length

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

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

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

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

Employment History

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

Writing Strong Bullets

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

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

Here's another example with an alternate wording:

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

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

Projects

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

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

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

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

Programming Languages and Software

Software

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

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

Languages

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

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

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

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

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

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

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

Advice for Non-Native English Speakers and Internationals

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

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

Beware of (Potential) Stigma

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

A few stigmas you should be aware of:

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

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

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

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

- **Certifications:** Certifications for software engineers can be anything from a positive, to a neutral, to a negative. This goes hand-in-hand with being too language focused; the companies that are biased against candidates with a very lengthy list of technologies tend to also be biased against certifications. This means that in some cases, you should actually remove this sort of experience from your resume.

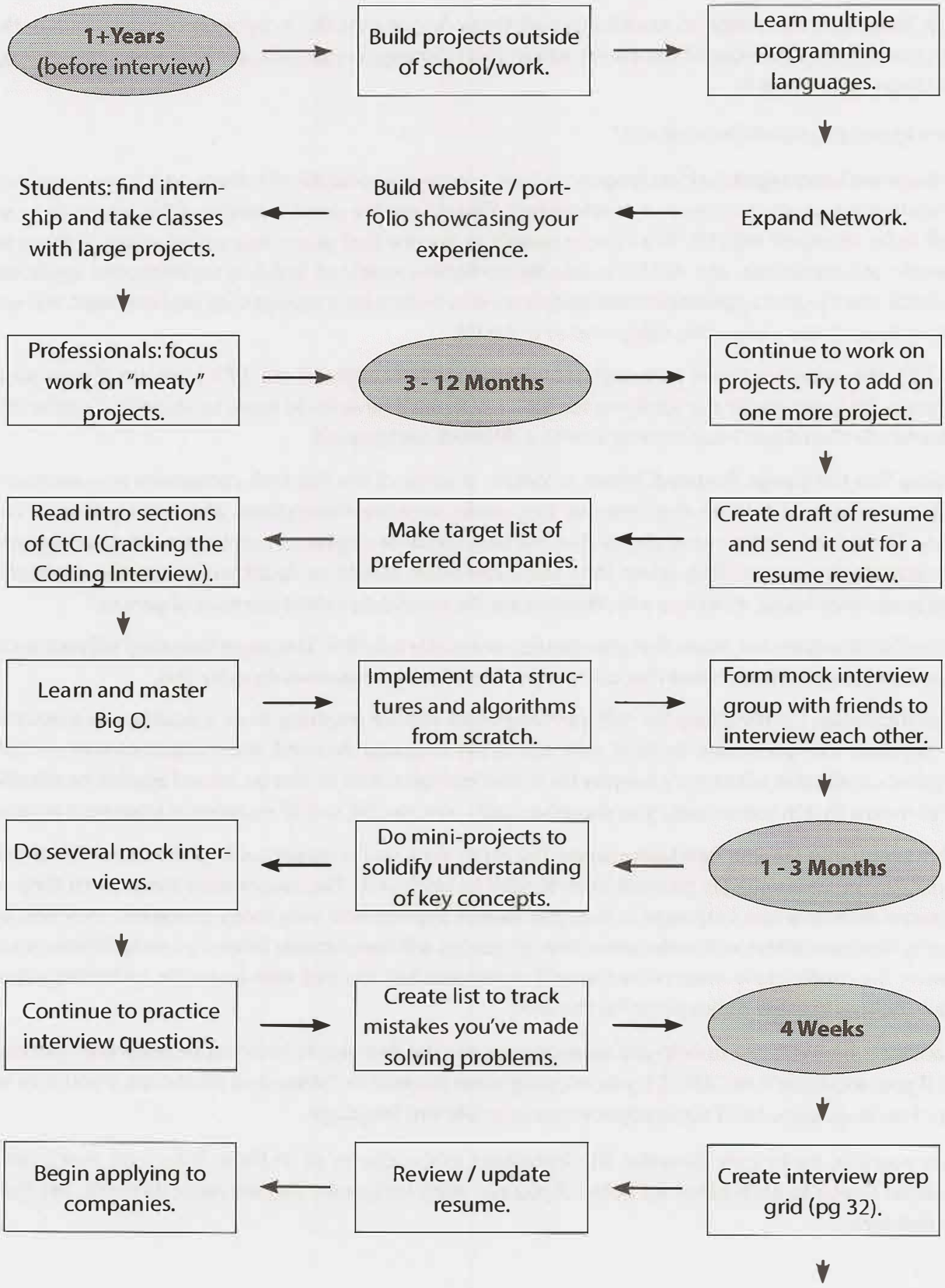
- **Knowing Only One or Two Languages:** The more time you've spent coding, the more things you've built, the more languages you will have tended to work with. The assumption then, when they see a resume with only one language, is that you haven't experienced very many problems. They also often worry that candidates with only one or two languages will have trouble learning new technologies (why hasn't the candidate learned more things?) or will just feel too tied with a specific technology (potentially not using the best language for the task).

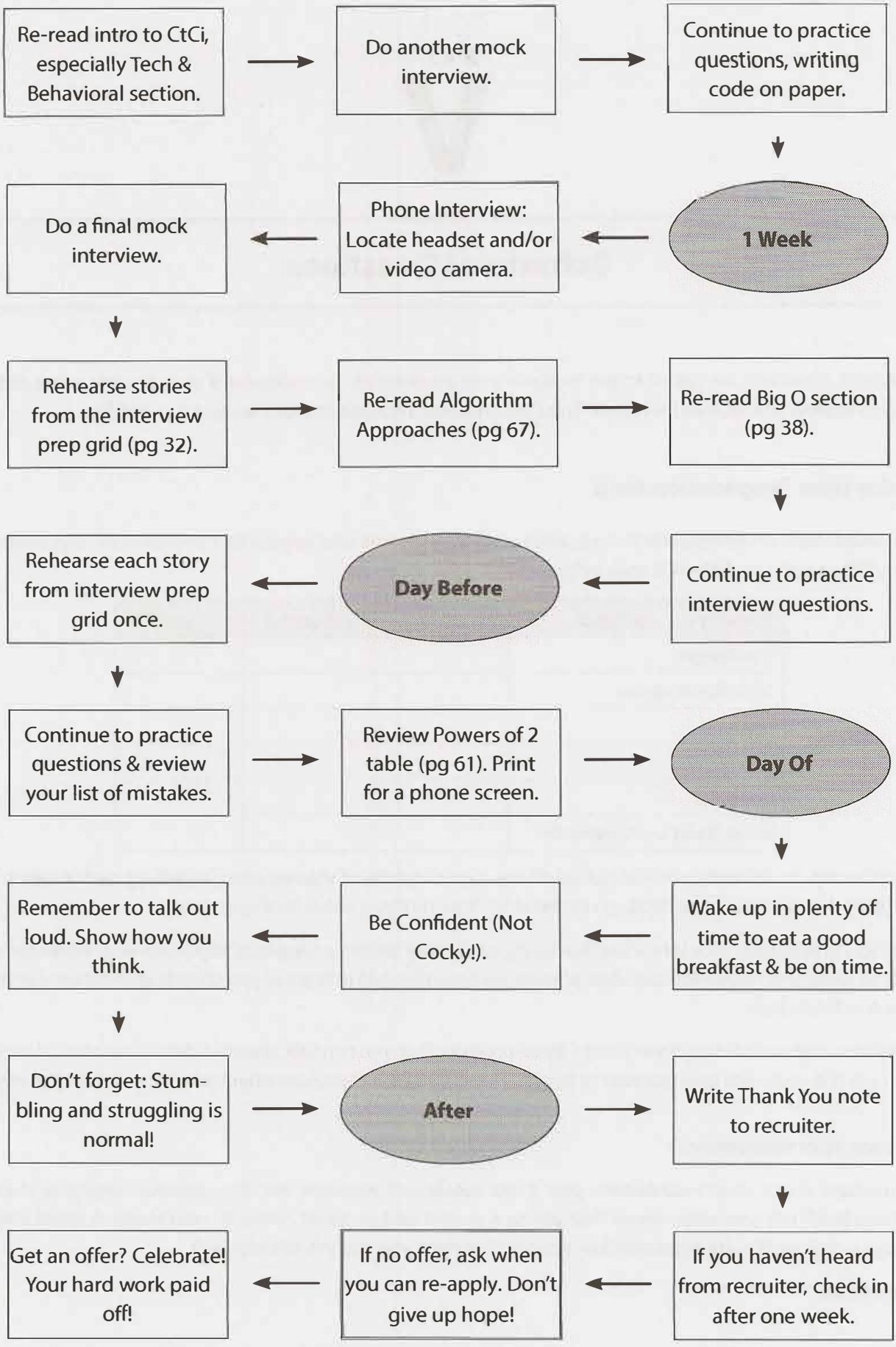
This advice is here not just to help you work on your resume, but also to help you develop the right experience. If your expertise is in C#.NET, try developing some projects in Python and JavaScript. If you only know one or two languages, build some applications in a different language.

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

Preparation Map

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




Re-read intro to CtCi, especially Tech & Behavioral section.
Do another mock interview.
Continue to practice questions, writing code on paper.
1 Week
Phone Interview: Locate headset and/or video camera.
Do a final mock interview.
Rehearse stories from the interview prep grid (pg 32).
Re-read Algorithm Approaches (pg 67).
Re-read Big O section (pg 38).
Continue to practice interview questions.
Day Before
Rehearse each story from interview prep grid once.
Continue to practice questions & review your list of mistakes.
Review Powers of 2 table (pg 61). Print for a phone screen.
Day Of
Wake up in plenty of time to eat a good breakfast & be on time.
Be Confident (Not Cocky!).
Remember to talk out loud. Show how you think.
Don't forget: Stumbling and struggling is normal!
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!
Get an offer? Celebrate! Your hard work paid off!

V

Behavioral Questions

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

▸ Interview Preparation Grid

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

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

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

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

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

What are your weaknesses?

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

For example:

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

What questions should you ask the interviewer?

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

You can think about three general types of questions.

Genuine Questions

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

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

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

Insightful Questions

These questions demonstrate your knowledge or understanding of technology.

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

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

Passion Questions

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

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

▸ Know Your Technical Projects

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

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

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

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

Responding to Behavioral Questions

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

Be Specific, Not Arrogant

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

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

Limit Details

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

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

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

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

Focus on Yourself, Not Your Team

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

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

Give Structured Answers

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

Nugget First

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

For example:

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

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

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

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

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

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

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

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

  Third and finally, I asked him to help me with breaking out some of the components of the project. We sat down together and designed a thorough spec for one of the big component, in much more detail than we had before. Once he could see all the pieces, it helped show him that the project wasn't as scary as he'd assumed.
- **Result:** With his confidence raised, he now offered to take on a bunch of the smaller coding work, and then eventually some of the biggest parts. He finished all his work on time, and he contributed more in discussions. We were happy to work with him on a future project.

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

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

Consider putting your stories into the following grid:

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

Explore the Action

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

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

Think About What It Says

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

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

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

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

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

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

▸ So, tell me about yourself...

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

Structure

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

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

entrepreneurial environment.

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

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

Hobbies

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

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

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

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

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

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

Sprinkle in Shows of Successes

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

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

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

VI

Big O

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

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

Master this concept.

An Analogy

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

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

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

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

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

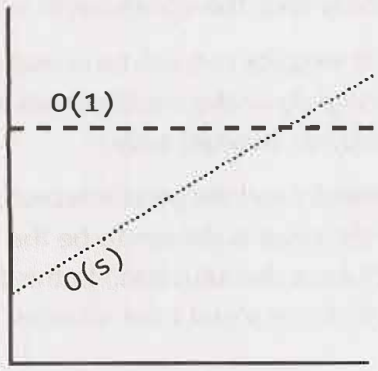
What if there were no flights, and instead you had to drive across the country? Even then, for a really huge file, it would be faster to drive.

Time Complexity

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

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

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



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

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

Big O, Big Theta, and Big Omega

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

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

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

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

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

Best Case, Worst Case, and Expected Case

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

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

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

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

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

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

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

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

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

▸ Space Complexity

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

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

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

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

Each call adds a level to the stack.

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

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

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

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

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

▸ Drop the Constants

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

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

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

Consider the below code:

Min and Max 1

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

Min and Max 2

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

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

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

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

Drop the Non-Dominant Terms

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

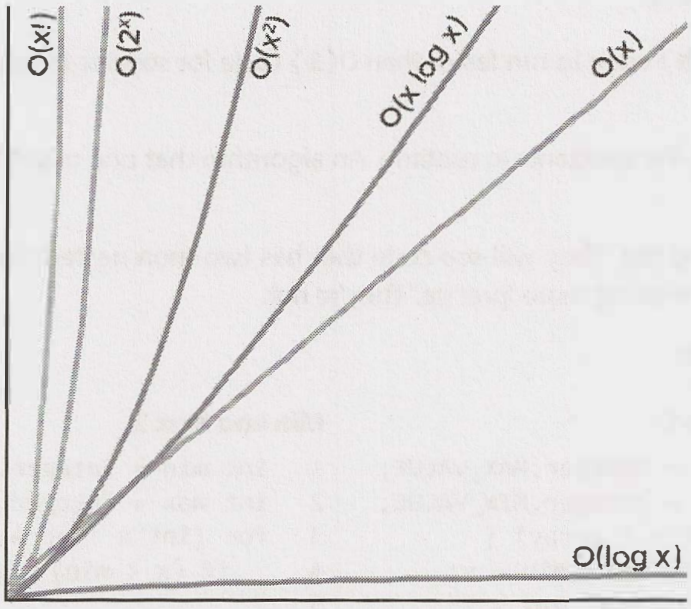
We already said that we drop constants. Therefore, $O(N^2 + N^2)$ would be $O(N^2)$. If we don't care about that latter N^2 term, why would we care about N? We don't.

You should drop the non-dominant terms.

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

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

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



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

Multi-Part Algorithms: Add vs. Multiply

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

This is a common source of confusion for candidates.

Add the Runtimes: O(A + B)

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

Multiply the Runtimes: O(A*B)

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

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

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

In other words:

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

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

▸ Amortized Time

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

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

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

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

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

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

In this case, what is the amortized time?

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

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

What then is the sum of $X + X/2 + X/4 + X/8 + ... + 1$? This is roughly 2X.

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

Log N Runtimes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Recursive Runtimes

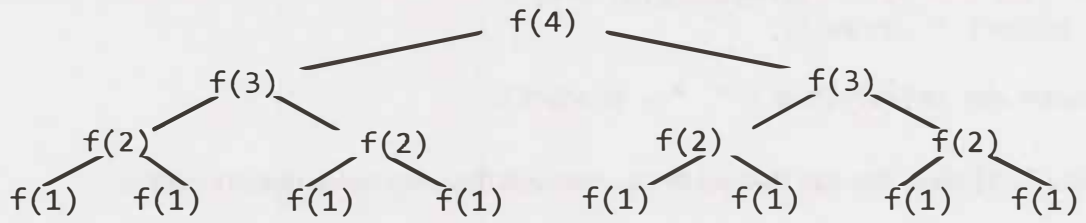
Here's a tricky one. What's the runtime of this code?

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

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

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

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



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

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

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

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

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

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

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

Examples and Exercises

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

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

Example 1

What is the runtime of the below code?

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

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

Example 2

What is the runtime of the below code?

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

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

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

Example 3

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

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

We can derive the runtime several ways.

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

Counting the Iterations

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

Therefore, the number of steps total is:

```
(N-1) + (N-2) + (N-3) + ... + 2 + 1
```

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

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

What It Means

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

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

Visualizing What It Does

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

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

This looks like half of an NxN matrix, which has size (roughly) $N^2/2$. Therefore, it takes $O(N^2)$ time.

Average Work

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

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

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

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

Example 4

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

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

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

We now have this:

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

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

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

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

Example 5

What about this strange bit of code?

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

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

Example 6

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

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

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

Example 7

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

- `O(N + P)`, where $P < N/2$
- `O(2N)`
- `O(N + log N)`
- `O(N + M)`

Let's go through these.

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

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

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

Example 8

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

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

This is completely incorrect. Did you catch the error?

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

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

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

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

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

Now we can work through this in parts:

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

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

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

Example 9

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

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

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

We can look at this two ways.

What It Means

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

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

Recursive Pattern

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

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

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

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

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

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

Recall what $\log_2$ means:

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

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

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

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

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

Example 10

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

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

What is the time complexity of this function?

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

```
8   }
```

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

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

The for loop will start when x = 2 and end when x*x = n. Or, in other words, it stops when x = √n (when x equals the square root of n).

This for loop is really something like this:

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

This runs in O(√n) time.

Example 11

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

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

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

Example 12

This code counts all permutations of a string.

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

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

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

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

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

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

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

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

How long does each function call take?

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

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

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

What is the total runtime?

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

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

Example 13

The following code computes the Nth Fibonacci number.

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

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

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

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

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

Example 14

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

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

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

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

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

Instead, let's walk through each call.

```
fib(1) -> 2^1 steps
fib(2) -> 2^2 steps
fib(3) -> 2^3 steps
fib(4) -> 2^4 steps
...
fib(n) -> 2^n steps
```

Therefore, the total amount of work is:

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

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

Example 15

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

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

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

Let's walk through what this algorithm does.

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

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

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

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

Example 16

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

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

There are several ways we could compute this runtime.

What It Does

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

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

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

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

What It Means

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

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

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

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

Rate of Increase

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

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

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

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

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

Additional Problems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            }
        }
        return -1;
    }
```

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

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

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

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

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

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

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

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

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

```
int numChars = 26;

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

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

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

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

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

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

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

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

    return intersect;
}
```

Solutions

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

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

VII

Technical Questions

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

How to Prepare

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

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

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

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

What You Need To Know

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

Core Data Structures, Algorithms, and Concepts

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

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

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

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

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

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

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

Powers of 2 Table

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

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

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

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

Walking Through a Problem

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

A Problem-Solving Flowchart

1 Listen

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

2 Example

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

BUD Optimization

Bottlenecks

Unnecessary Work

Duplicated Work

3 Brute Force

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

4 Optimize

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

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

5 Walk Through

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

6 Implement

Your goal is to **write beautiful code.** Modularize your code from the beginning and refactor to clean up anything that isn't beautiful.

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

7 Test

Test in this order:

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

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

We'll go through this flowchart in more detail.

What to Expect

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

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

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

1. Listen Carefully

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

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

But I'm saying something more than that.

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

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

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

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

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

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

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

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

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

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

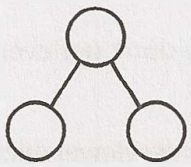
2. Draw an Example

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

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

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

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



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

Instead, you want to create an example that is:

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

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

3. State a Brute Force

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

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

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

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

4. Optimize

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

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

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

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

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

5. Walk Through

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

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

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

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

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

6. Implement

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

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

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

Beautiful code means:

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

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

  Long variable names can also be slow to write though. A good compromise that most interviewers will be okay with is to abbreviate it after the first usage. You can use `startChild` the first time, and then explain to your interviewer that you will abbreviate this as `sc` after this.

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

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

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

7. Test

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

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

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

Instead, try this approach:

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

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

Optimize & Solve Technique #1: Look for BUD

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

- **B**ottlenecks
- **U**nnecessary work
- **D**uplicated work

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

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

Bottlenecks

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

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

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

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

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

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

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

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

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

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

Unnecessary Work

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

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

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

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

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

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

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

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

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

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

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

Duplicated Work

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
s: abbc
b: cbabadcbbabbcbabaabccbabc
```

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

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

I've underlined below each permutation.

```
s: abbc
b: cbabadcbbabbcbabaabccbabc
     ____    ____      ____
         ____ ____      ____
              ____
```

Did you find these? How?

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

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

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

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

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

Optimize & Solve Technique #3: Simplify and Generalize

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

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

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

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

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

Optimize & Solve Technique #4: Base Case and Build

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

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

Consider a test string abcdefg.

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

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

```
P("abc") = 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. For each string in it, we insert s_n into every location of the string.

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

Optimize & Solve Technique #5: Data Structure Brainstorm

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

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

Our data structure brainstorm might look like the following:

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

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

Best Conceivable Runtime (BCR)

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

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

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

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

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

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

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

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

An Example of How to Use BCR

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

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

```
A: 13  27  35  40  49  55  59
B: 17  35  39  40  55  58  60
```

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

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

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

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

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

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

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

If we imagine our current algorithm's runtime as $O(N \times N)$, then getting to $O(N)$ or $O(N \times \log N)$ might mean reducing that second $O(N)$ in the equation to $O(1)$ or $O(\log N)$.

This is one way that BCR can be useful. We can use the runtimes to get a "hint" for what we need to reduce.

That second O(N) comes from searching. The array is sorted. Can we search in a sorted array in faster than O(N) time?

Why, yes. We can use binary search to find an element in a sorted array in O(log N) time.

We now have an improved algorithm: O(N log N).

```
Brute Force:        O(N²)
Improved Algorithm: O(N log N)
Optimal Algorithm:  ?
BCR:                O(N)
```

Can we do even better? Doing better likely means reducing that O(log N) to O(1).

In general, we cannot search an array—even a sorted array—in better than O(log N) time. This is *not* the general case though. We're doing this search over and over again.

The BCR is telling us that we will never, ever have an algorithm that's faster than O(N). Therefore, any work we do in O(N) time is a "freebie"—it won't impact our runtime.

Re-read the list of optimization tips on page 64. Is there anything that can help us?

One of the tips there suggests precomputing or doing upfront work. Any upfront work we do in O(N) time is a freebie. It won't impact our runtime.

> This is another place where BCR can be useful. Any work you do that's less than or equal to the BCR is "free," in the sense that it won't impact your runtime. You might want to eliminate it eventually, but it's not a top priority just yet.

Our focus is still on reducing search from O(log N) to O(1). Any precomputation that's O(N) or less is "free."

In this case, we can just throw everything in B into a hash table. This will take O(N) time. Then, we just go through A and look up each element in the hash table. This look up (or search) is O(1), so our runtime is O(N).

Suppose our interviewer hits us with a question that makes us cringe: Can we do better?

No, not in terms of runtime. We have achieved the fastest possible runtime, therefore we cannot optimize the big O time. We could potentially optimize the space complexity.

> This is another place where BCR is useful. It tells us that we're "done" in terms of optimizing the runtime, and we should therefore turn our efforts to the space complexity.

In fact, even without the interviewer prompting us, we should have a question mark with respect to our algorithm. We would have achieved the exact same runtime if the data wasn't sorted. So why did the interviewer give us sorted arrays? That's not unheard of, but it is a bit strange.

Let's turn back to our example.

```
A: 13  27  35  40  49  55  59
B: 17  35  39  40  55  58  60
```

We're now looking for an algorithm that:

- Operates in O(1) space (probably). We already have an O(N) space algorithm with optimal runtime. If we want to use less additional space, that probably means no additional space. Therefore, we need to drop the hash table.

- Operates in O(N) time (probably). We'll probably want to at least match the current best runtime, and we know we can't beat it.
- Uses the fact that the arrays are sorted.

Our best algorithm that doesn't use extra space was the binary search one. Let's think about optimizing that. We can try walking through the algorithm.

1. Do a binary search in B for A[0] = 13. Not found.
2. Do a binary search in B for A[1] = 27. Not found.
3. Do a binary search in B for A[2] = 35. Found at B[1].
4. Do a binary search in B for A[3] = 40. Found at B[5].
5. Do a binary search in B for A[4] = 49. Not found.
6. . . .

Think about BUD. The bottleneck is the searching. Is there anything unnecessary or duplicated?

It's unnecessary that A[3] = 40 searched over all of B. We know that we just found 35 at B[1], so 40 certainly won't be before 35.

Each binary search should start where the last one left off.

In fact, we don't need to do a binary search at all now. We can just do a linear search. As long as the linear search in B is just picking up where the last one left off, we know that we're going to be operating in linear time.

1. Do a linear search in B for A[0] = 13. Start at B[0] = 17. Stop at B[0] = 17. Not found.
2. Do a linear search in B for A[1] = 27. Start at B[0] = 17. Stop at B[1] = 35. Not found.
3. Do a linear search in B for A[2] = 35. Start at B[1] = 35. Stop at B[1] = 35. Found.
4. Do a linear search in B for A[3] = 40. Start at B[2] = 39. Stop at B[3] = 40. Found.
5. Do a linear search in B for A[4] = 49. Start at B[3] = 40. Stop at B[4] = 55. Found.
6. ...

This algorithm is very similar to merging two sorted arrays. It operates in O(N) time and O(1) space.

We have now reached the BCR and have minimal space. We know that we cannot do better.

> This is another way we can use BCR. If you ever reach the BCR and have O(1) additional space, then you know that you can't optimize the big O time or space.

Best Conceivable Runtime is not a "real" algorithm concept, in that you won't find it in algorithm textbooks. But I have found it personally very useful, when solving problems myself, as well as while coaching people through problems.

If you're struggling to grasp it, make sure you understand big O time first (page 38). You need to master it. Once you do, figuring out the BCR of a problem should take literally seconds.

Handling Incorrect Answers

One of the most pervasive—and dangerous—rumors is that candidates need to get every question right. That's not quite true.

First, responses to interview questions shouldn't be thought of as "correct" or "incorrect." When I evaluate how someone performed in an interview, I never think, "How many questions did they get right?" It's not a binary evaluation. Rather, it's about how optimal their final solution was, how long it took them to get there, how much help they needed, and how clean was their code. There is a range of factors.

Second, your performance is evaluated *in comparison to other candidates*. For example, if you solve a question optimally in 15 minutes, and someone else solves an easier question in five minutes, did that person do better than you? Maybe, but maybe not. If you are asked really easy questions, then you might be expected to get optimal solutions really quickly. But if the questions are hard, then a number of mistakes are expected.

Third, many—possibly most—questions are too difficult to expect even a strong candidate to immediately spit out the optimal algorithm. The questions I tend to ask would take strong candidates typically 20 to 30 minutes to solve.

In evaluating thousands of hiring packets at Google, I have only once seen a candidate have a "flawless" set of interviews. Everyone else, including the hundreds who got offers, made mistakes.

When You've Heard a Question Before

If you've heard a question before, admit this to your interviewer. Your interviewer is asking you these questions in order to evaluate your problem-solving skills. If you already know the question, then you aren't giving them the opportunity to evaluate you.

Additionally, your interviewer may find it highly dishonest if you don't reveal that you know the question. (And, conversely, you'll get big honesty points if you do reveal this.)

The "Perfect" Language for Interviews

At many of the top companies, interviewers aren't picky about languages. They're more interested in how well you solve the problems than whether you know a specific language.

Other companies though are more tied to a language and are interested in seeing how well you can code in a particular language.

If you're given a choice of languages, then you should probably pick whatever language you're most comfortable with.

That said, if you have several good languages, you should keep in mind the following.

Prevalence

It's not required, but it is ideal for your interviewer to know the language you're coding in. A more widely known language can be better for this reason.

Language Readability

Even if your interviewer doesn't know your programming language, they should hopefully be able to basically understand it. Some languages are more naturally readable than others, due to their similarity to other languages.

For example, Java is fairly easy for people to understand, even if they haven't worked in it. Most people have worked in something with Java-like syntax, such as C and C++.

However, languages such as Scala or Objective C have fairly different syntax.

Potential Problems

Some languages just open you up to potential issues. For example, using C++ means that, in addition to all the usual bugs you can have in your code, you can have memory management and pointer issues.

Verbosity

Some languages are more verbose than others. Java for example is a fairly verbose language as compared with Python. Just compare the following code snippets.

Python:

```
1   dict = {"left": 1, "right": 2, "top": 3, "bottom": 4};
```

Java:

```
1   HashMap<String, Integer> dict = new HashMap<String, Integer>().
2   dict.put("left", 1);
3   dict.put("right", 2);
4   dict.put("top", 3);
5   dict.put("bottom", 4);
```

However, some of the verbosity of Java can be reduced by abbreviating code. I could imagine a candidate on a whiteboard writing something like this:

```
1   HM<S, I> dict = new HM<S, I>().
2   dict.put("left", 1);
3   ...      "right", 2
4   ...      "top", 3
5   ...      "bottom", 4
```

The candidate would need to explain the abbreviations, but most interviewers wouldn't mind.

Ease of Use

Some operations are easier in some languages than others. For example, in Python, you can very easily return multiple values from a function. In Java, the same action would require a new class. This can be handy for certain problems.

Similar to the above though, this can be mitigated by just abbreviating code or presuming methods that you don't actually have. For example, if one language provides a function to transpose a matrix and another language doesn't, this doesn't necessarily make the first language much better to code in (for a problem that needs such a function). You could just assume that the other language has a similar method.

▸ What Good Coding Looks Like

You probably know by now that employers want to see that you write "good, clean" code. But what does this really mean, and how is this demonstrated in an interview?

Broadly speaking, good code has the following properties:

- **Correct:** The code should operate correctly on all expected and unexpected inputs.
- **Efficient:** The code should operate as efficiently as possible in terms of both time and space. This "efficiency" includes both the asymptotic (big O) efficiency and the practical, real-life efficiency. That is, a

constant factor might get dropped when you compute the big O time, but in real life, it can very much matter.

- **Simple:** If you can do something in 10 lines instead of 100, you should. Code should be as quick as possible for a developer to write.
- **Readable:** A different developer should be able to read your code and understand what it does and how it does it. Readable code has comments where necessary, but it implements things in an easily understandable way. That means that your fancy code that does a bunch of complex bit shifting is not necessarily *good* code.
- **Maintainable:** Code should be reasonably adaptable to changes during the life cycle of a product and should be easy to maintain by other developers, as well as the initial developer.

Striving for these aspects requires a balancing act. For example, it's often advisable to sacrifice some degree of efficiency to make code more maintainable, and vice versa.

You should think about these elements as you code during an interview. The following aspects of code are more specific ways to demonstrate the earlier list.

Use Data Structures Generously

Suppose you were asked to write a function to add two simple mathematical expressions which are of the form $Ax^a + Bx^b + \ldots$ (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 because you need to return two arrays.

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

Good Implementation

A good implementation for this problem is to design your own data structure for the expression.

```
class ExprTerm {
   double coefficient;
   double exponent;
}

ExprTerm[] sum(ExprTerm[] expr1, ExprTerm[] expr2) {
   ...
}
```

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.

```
boolean compareBinToHex(String binary, String hex) {
   int n1 = convertFromBase(binary, 2);
   int n2 = convertFromBase(hex, 16);
   if (n1 < 0 || n2 < 0) {
      return false;
   }
   return n1 == n2;
}

int convertFromBase(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;
}

int digitToValue(char c) { ... }
```

We could have implemented separate code to convert a binary number and a hexadecimal code, but this just makes our code harder to write and harder to maintain. Instead, we reuse code by writing one `convertFromBase` method and one `digitToValue` method.

Modular

Writing modular code means separating isolated chunks of code out into their own methods. This helps keep the code more maintainable, readable, and testable.

Imagine you are writing code to swap the minimum and maximum element in an integer array. You could implement it all in one method like this:

```
1  void swapMinMax(int[] array) {
2    int minIndex = 0;
3    for (int i = 1; i < array.length; i++) {
4      if (array[i] < array[minIndex]) {
5        minIndex = i;
6      }
7    }
8
9    int maxIndex = 0;
10   for (int i = 1; i < array.length; i++) {
11     if (array[i] > array[maxIndex]) {
12       maxIndex = i;
13     }
14   }
15
16   int temp = array[minIndex];
17   array[minIndex] = array[maxIndex];
18   array[maxIndex] = temp;
19 }
```

Or, you could implement in a more modular way by separating the relatively isolated chunks of code into their own methods.

```
1  void swapMinMaxBetter(int[] array) {
2    int minIndex = getMinIndex(array);
3    int maxIndex = getMaxIndex(array);
4    swap(array, minIndex, maxIndex);
5  }
6
7  int getMinIndex(int[] array) { ... }
8  int getMaxIndex(int[] array) { ... }
9  void swap(int[] array, int m, int n) { ... }
```

While the non-modular code isn't particularly awful, the nice thing about the modular code is that it's easily testable because each component can be verified separately. As code gets more complex, it becomes increasingly important to write it in a modular way. This will make it easier to read and maintain. Your interviewer wants to see you demonstrate these skills in your interview.

Flexible and Robust

Just because your interviewer only asks you to write code to check if a normal tic-tac-toe board has a winner, doesn't mean you *must* assume that it's a 3x3 board. Why not write the code in a more general way that implements it for an `NxN` board?

Writing flexible, general-purpose code may also mean using variables instead of hard-coded values or using templates / generics to solve a problem. If we can write our code to solve a more general problem, we should.

Of course, there is a limit. If the solution is much more complex for the general case, and it seems unnecessary at this point in time, it may be better just to implement the simple, expected case.

Error Checking

One sign of a careful coder is that she doesn't make assumptions about the input. Instead, she validates that the input is what it should be, either through ASSERT statements or if-statements.

For example, recall the earlier code to convert a number from its base `i` (e.g., base 2 or base 16) representation to an `int`.

```
1   int convertToBase(String number, int base) {
2     if (base < 2 || (base > 10 && base != 16)) return -1;
3     int value = 0;
4     for (int i = number.length() - 1; i >= 0; i--) {
5       int digit = digitToValue(number.charAt(i));
6       if (digit < 0 || digit >= base) {
7         return -1;
8       }
9       int exp = number.length() - 1 - i;
10      value += digit * Math.pow(base, exp);
11    }
12    return value;
13  }
```

In line 2, we check to see that `base` is valid (we assume that bases greater than 10, other than base 16, have no standard representation in string form). In line 6, we do another error check: making sure that each digit falls within the allowable range.

Checks like these are critical in production code and, therefore, in interview code as well.

Of course, writing these error checks can be tedious and can waste precious time in an interview. The important thing is to point out that you *would* write the checks. If the error checks are much more than a quick if-statement, it may be best to leave some space where the error checks would go and indicate to your interviewer that you'll fill them in when you're finished with the rest of the code.

Don't Give Up!

I know interview questions can be overwhelming, but that's part of what the interviewer is testing. Do you rise to a challenge, or do you shrink back in fear? It's important that you step up and eagerly meet a tricky problem head-on. After all, remember that interviews are supposed to be hard. It shouldn't be a surprise when you get a really tough problem.

For extra "points," show excitement about solving hard problems.

VIII

The Offer and Beyond

Just when you thought you could sit back and relax after your interviews, now you're faced with the post-interview stress: Should you accept the offer? Is it the right one? How do you decline an offer? What about deadlines? We'll handle a few of these issues here and go into more details about how to evaluate an offer, and how to negotiate it.

Handling Offers and Rejection

Whether you're accepting an offer, declining an offer, or responding to a rejection, it matters what you do.

Offer Deadlines and Extensions

When companies extend an offer, there's almost always a deadline attached to it. Usually these deadlines are one to four weeks out. If you're still waiting to hear back from other companies, you can ask for an extension. Companies will usually try to accommodate this, if possible.

Declining an Offer

Even if you aren't interested in working for this company right now, you might be interested in working for it in a few years. (Or, your contacts might one day move to a more exciting company.) It's in your best interest to decline the offer on good terms and keep a line of communication open.

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

Handling Rejection

Getting rejected is unfortunate, but it doesn't mean that you're not a great engineer. Lots of great engineers do poorly, either because they don't "test well" on these sort of interviewers, or they just had an "off" day.

Fortunately, most companies understand that these interviews aren't perfect and many good engineers get rejected. For this reason, companies are often eager to re-interview previously rejected candidate. Some companies will even reach out to old candidates or expedite their application *because* of their prior performance.

When you do get the unfortunate call, use this as an opportunity to build a bridge to re-apply. Thank your recruiter for his time, explain that you're disappointed but that you understand their position, and ask when you can reapply to the company.

You can also ask for feedback from the recruiter. In most cases, the big tech companies won't offer feedback, but there are some companies that will. It doesn't hurt to ask a question like, "Is there anything you'd suggest I work on for next time?"

Evaluating the Offer

Congratulations! You got an offer! And—if you're lucky—you may have even gotten multiple offers. Your recruiter's job is now to do everything he can to encourage you to accept it. How do you know if the company is the right fit for you? We'll go through a few things you should consider in evaluating an offer.

The Financial Package

Perhaps the biggest mistake that candidates make in evaluating an offer is looking too much at their salary. Candidates often look so much at this one number that they wind up accepting the offer that is *worse* financially. Salary is just one part of your financial compensation. You should also look at:

- *Signing Bonus, Relocation, and Other One Time Perks:* Many companies offer a signing bonus and/or relocation. When comparing offers, it's wise to amortize this cash over three years (or however long you expect to stay).
- *Cost of Living Difference:* Taxes and other cost of living differences can make a big difference in your take-home pay. Silicon Valley, for example, is 30+% more expensive than Seattle.
- *Annual Bonus:* Annual bonuses at tech companies can range from anywhere from 3% to 30%. Your recruiter might reveal the average annual bonus, but if not, check with friends at the company.
- *Stock Options and Grants:* Equity compensation can form another big part of your annual compensation. Like signing bonuses, stock compensation between companies can be compared by amortizing it over three years and then lumping that value into salary.

Remember, though, that what you learn and how a company advances your career often makes far more of a difference to your long term finances than the salary. Think very carefully about how much emphasis you really want to put on money right now.

Career Development

As thrilled as you may be to receive this offer, odds are, in a few years, you'll start thinking about interviewing again. Therefore, it's important that you think right now about how this offer would impact your career path. This means considering the following questions:

- How good does the company's name look on my resume?
- How much will I learn? Will I learn relevant things?
- What is the promotion plan? How do the careers of developers progress?
- If I want to move into management, does this company offer a realistic plan?
- Is the company or team growing?
- If I do want to leave the company, is it situated near other companies I'm interested in, or will I need to move?

The final point is extremely important and usually overlooked. If you only have a few other companies to pick from in your city, your career options will be more restricted. Fewer options means that you're less likely to discover really great opportunities.

Company Stability

All else being equal, of course stability is a good thing. No one wants to be fired or laid off.

However, all else isn't actually equal. The more stable companies are also often growing more slowly.

How much emphasis you should put on company stability really depends on you and your values. For some candidates, stability should not be a large factor. Can you fairly quickly find a new job? If so, it might be better to take the rapidly growing company, even if it's unstable? If you have work visa restrictions or just aren't confident in your ability to find something new, stability might be more important.

The Happiness Factor

Last but not least, you should of course consider how happy you will be. Any of the following factors may impact that:

- *The Product:* Many people look heavily at what product they are building, and of course this matters a bit. However, for most engineers, there are more important 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 and Facebook), you'll have an opportunity to find a team and product that matches you well.

▸ Negotiation

Years ago, I signed up for a negotiations class. On the first day, the instructor asked us to imagine a scenario where we wanted to buy a car. Dealership A sells the car for a fixed $20,000—no negotiating. Dealership B allows us to negotiate. How much would the car have to be (after negotiating) for us to go to Dealership B? (Quick! Answer this for yourself!)

On average, the class said that the car would have to be $750 cheaper. In other words, students were willing to pay $750 just to avoid having to negotiate for an hour or so. Not surprisingly, in a class poll, most of these students also said they didn't negotiate their job offer. They just accepted whatever the company gave them.

Many of us can probably sympathize with this position. Negotiation isn't fun for most of us. But still, the financial benefits of negotiation are usually worth it.

Do yourself a favor. Negotiate. Here are some tips to get you started.

1. *Just Do It.* Yes, I know it's scary; (almost) no one likes negotiating. But it's so, so worth it. Recruiters will not revoke an offer because you negotiated, so you have little to lose. This is especially true if the offer is from a larger company. You probably won't be negotiating with your future teammates.
2. *Have a Viable Alternative.* Fundamentally, recruiters negotiate with you because they're concerned you may not join the company otherwise. If you have alternative options, that will make their concern much more real.
3. *Have a Specific "Ask":* It's more effective to ask for an additional $7000 in salary than to just ask for "more."

After all, if you just ask for more, the recruiter could throw in another $1000 and technically have satisfied your wishes.

4. *Overshoot:* In negotiations, people usually don't agree to whatever you demand. It's a back and forth conversation. Ask for a bit more than you're really hoping to get, since the company will probably meet you in the middle.
5. *Think Beyond Salary:* Companies are often more willing to negotiate on non-salary components, since boosting your salary too much could mean that they're paying you more than your peers. Consider asking for more equity or a bigger signing bonus. Alternatively, you may be able to ask for your relocation benefits in cash, instead of having the company pay directly for the moving fees. This is a great avenue for many college students, whose actual moving expenses are fairly cheap.
6. *Use Your Best Medium:* Many people will advise you to only negotiate over the phone. To a certain extent, they're right; it is better to negotiate over the phone. However, if you don't feel comfortable on a phone negotiation, do it via email. It's more important that you attempt to negotiate than that you do it via a specific medium.

Additionally, if you're negotiating with a big company, you should know that they often have "levels" for employees, where all employees at a particular level are paid around the same amount. Microsoft has a particularly well-defined system for this. You can negotiate within the salary range for your level, but going beyond that requires bumping up a level. If you're looking for a big bump, you'll need to convince the recruiter and your future team that your experience matches this higher level—a difficult, but feasible, thing to do.

▸ On the Job

Navigating your career path doesn't end at the interview. In fact, it's just getting started. Once you actually join a company, you need to start thinking about your career path. Where will you go from here, and how will you get there?

Set a Timeline

It's a common story: you join a company, and you're psyched. Everything is great. Five years later, you're still there. And it's then that you realize that these last three years didn't add much to your skill set or to your resume. Why didn't you just leave after two years?

When you're enjoying your job, it's very easy to get wrapped up in it and not realize that your career is not advancing. This is why you should outline your career path before starting a new job. Where do you want to be in ten years? And what are the steps necessary to get there? In addition, each year, think about what the next year of experience will bring you and how your career or your skill set advanced in the last year.

By outlining your path in advance and checking in on it regularly, you can avoid falling into this complacency trap.

Build Strong Relationships

When you want to move on to something new, your network will be critical. After all, applying online is tricky; a personal referral is much better, and your ability to do so hinges on your network.

At work, establish strong relationships with your manager and teammates. When employees leave, keep in touch with them. Just a friendly note a few weeks after their departure will help to bridge that connection from a work acquaintance to a personal acquaintance.

This same approach applies to your personal life. Your friends, and your friends of friends, are valuable connections. Be open to helping others, and they'll be more likely to help you.

Ask for What You Want

While some managers may really try to grow your career, others will take a more hands-off approach. It's up to you to pursue the challenges that are right for your career.

Be (reasonably) frank about your goals with your manager. If you want to take on more back-end coding projects, say so. If you'd like to explore more leadership opportunities, discuss how you might be able to do so.

You need to be your best advocate, so that you can achieve goals according to your timeline.

Keep Interviewing

Set a goal of interviewing at least once a year, even if you aren't actively looking for a new job. This will keep your interview skills fresh, and also keep you in tune with what sorts of opportunities (and salaries) are out there.

If you get an offer, you don't have to take it. It will still build a connection with that company in case you want to join at a later date.

Interview Questions

IX

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

1

Arrays and Strings

Hopefully, all readers of this book are familiar with arrays and strings, so we won't bore you with such details. Instead, we'll focus on some of the more common techniques and issues with these data structures.

Please note that array questions and string questions are often interchangeable. That is, a question that this book states using an array may be asked instead as a string question, and vice versa.

▸ Hash Tables

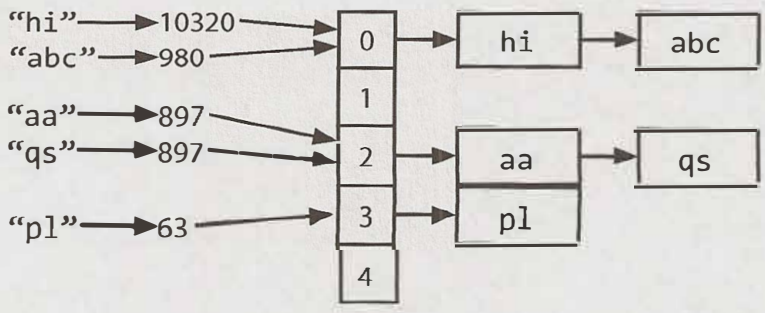
A hash table is a data structure that maps keys to values for highly efficient lookup. There are a number of ways of implementing this. Here, we will describe a simple but common implementation.

In this simple implementation, we use an array of linked lists and a hash code function. To insert a key (which might be a string or essentially any other data type) and value, we do the following:

1. First, compute the key's hash code, which will usually be an `int` or `long`. Note that two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of ints.
2. Then, map the hash code to an index in the array. This could be done with something like `hash(key) % array_length`. Two different hash codes could, of course, map to the same index.
3. At this index, there is a linked list of keys and values. Store the key and value in this index. We must use a linked list because of collisions: you could have two different keys with the same hash code, or two different hash codes that map to the same index.

To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. Then, search through the linked list for the value with this key.

If the number of collisions is very high, the worst case runtime is `O(N)`, where `N` is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is `O(1)`.



Alternatively, we can implement the hash table with a balanced binary search tree. This gives us an O(log N) lookup time. The advantage of this is potentially using less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

▸ ArrayList & Resizable Arrays

In some languages, arrays (often called lists in this case) are automatically resizable. The array or list will grow as you append items. In other languages, like Java, arrays are fixed length. The size is defined when you create the array.

When you need an array-like data structure that offers dynamic resizing, you would usually use an ArrayList. An ArrayList is an array that resizes itself as needed while still providing O(1) access. A typical implementation is that when the array is full, the array doubles in size. Each doubling takes O(n) time, but happens so rarely that its amortized insertion time is still O(1).

```
1  ArrayList<String> merge(String[] words, String[] more) {
2    ArrayList<String> sentence = new ArrayList<String>();
3    for (String w : words) sentence.add(w);
4    for (String w : more) sentence.add(w);
5    return sentence;
6  }
```

This is an essential data structure for interviews. Be sure you are comfortable with dynamically resizable arrays/lists in whatever language you will be working with. Note that the name of the data structure as well as the "resizing factor" (which is 2 in Java) can vary.

Why is the amortized insertion runtime O(1)?

Suppose you have an array of size N. We can work backwards to compute how many elements we copied at each capacity increase. Observe that when we increase the array to K elements, the array was previously half that size. Therefore, we needed to copy $K/2$ elements.

```
final capacity increase   : n/2 elements to copy
previous capacity increase: n/4 elements to copy
previous capacity increase: n/8 elements to copy
previous capacity increase: n/16 elements to copy
...
second capacity increase  : 2 elements to copy
first capacity increase   : 1 element to copy
```

Therefore, the total number of copies to insert N elements is roughly $N/2 + N/4 + N/8 + \ldots + 2 + 1$, which is just less than N.

> If the sum of this series isn't obvious to you, imagine this: Suppose you have a kilometer-long walk to the store. You walk 0.5 kilometers, and then 0.25 kilometers, and then 0.125 kilometers, and so on. You will never exceed one kilometer (although you'll get very close to it).

Therefore, inserting N elements takes O(N) work total. Each insertion is O(1) on average, even though some insertions take O(N) time in the worst case.

▸ StringBuilder

Imagine you were concatenating a list of strings, as shown below. What would the running time of this code be? For simplicity, assume that the strings are all the same length (call this x) and that there are n strings.

```
1  String joinWords(String[] words) {
2    String sentence = "";
3    for (String w : words) {
4      sentence = sentence + w;
5    }
6    return sentence;
7  }
```

On each concatenation, a new copy of the string is created, and the two strings are copied over, character by character. The first iteration requires us to copy x characters. The second iteration requires copying 2x characters. The third iteration requires 3x, and so on. The total time therefore is $O(x + 2x + ... + nx)$. This reduces to $O(xn^2)$.

Why is it $O(xn^2)$? Because $1 + 2 + ... + n$ equals $n(n+1)/2$, or $O(n^2)$.

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

```
1  String joinWords(String[] words) {
2    StringBuilder sentence = new StringBuilder();
3    for (String w : words) {
4      sentence.append(w);
5    }
6    return sentence.toString();
7  }
```

A good exercise to practice strings, arrays, and general data structures is to implement your own version of `StringBuilder`, `HashTable` and `ArrayList`.

Additional Reading: Hash Table Collision Resolution (pg 636), Rabin-Karp Substring Search (pg 636).

Interview Questions

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

Hints: #44, #117, #132

pg 192

1.2 **Check Permutation:** Given two strings, write a method to decide if one is a permutation of the other.

Hints: #1, #84, #122, #131

pg 193

1.3 **URLify:** Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string. (Note: If implementing in Java, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: `"Mr John Smith    ", 13`

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

Hints: #53, #118

pg 194

1.4 **Palindrome Permutation:** Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: `Tact Coa`

Output: `True (permutations: "taco cat", "atco cta", etc.)`

Hints: #106, #121, #134, #136

pg 195

1.5 **One Away:** There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.

EXAMPLE

```
pale,  ple  -> true
pales, pale -> true
pale,  bale -> true
pale,  bake -> false
```

Hints: #23, #97, #130

pg 199

1.6 **String Compression:** Implement a method to perform basic string compression using the counts of repeated characters. For example, the string `aabcccccaaa` would become `a2b1c5a3`. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a - z).

Hints: #92, #110

pg 201

1.7 **Rotate Matrix:** Given an image represented by an `NxN` matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

Hints: #51, #100

pg 203

1.8 **Zero Matrix:** Write an algorithm such that if an element in an `MxN` matrix is 0, its entire row and column are set to 0.

Hints: #17, #74, #102

pg 204

1.9 **String Rotation:** Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").

Hints: #34, #88, #104

pg 206

Additional Questions: Object-Oriented Design (#7.12), Recursion (#8.3), Sorting and Searching (#10.9), C++ (#12.11), Moderate Problems (#16.8, #16.17, #16.22), Hard Problems (#17.4, #17.7, #17.13, #17.22, #17.26).

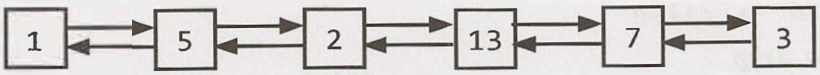
Hints start on page 653.

2

Linked Lists

A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next node and the previous node.

The following diagram depicts a doubly linked list:



Unlike an array, a linked list does not provide constant time access to a particular "index" within the list. This means that if you'd like to find the Kth element in the list, you will need to iterate through K elements.

The benefit of a linked list is that you can add and remove items from the beginning of the list in constant time. For specific applications, this can be useful.

Creating a Linked List

The code below implements a very basic singly linked list.

```
1  class Node {
2    Node next = null;
3    int data;
4
5    public Node(int d) {
6      data = d;
7    }
8
9    void appendToTail(int d) {
10     Node end = new Node(d);
11     Node n = this;
12     while (n.next != null) {
13       n = n.next;
14     }
15     n.next = end;
16   }
17 }
```

In this implementation, we don't have a `LinkedList` data structure. We access the linked list through a reference to the head `Node` of the linked list. When you implement the linked list this way, you need to be a bit careful. What if multiple objects need a reference to the linked list, and then the head of the linked list changes? Some objects might still be pointing to the old head.

We could, if we chose, implement a `LinkedList` class that wraps the `Node` class. This would essentially just have a single member variable: the head `Node`. This would largely resolve the earlier issue.

Remember that when you're discussing a linked list in an interview, you must understand whether it is a singly linked list or a doubly linked list.

Deleting a Node from a Singly Linked List

Deleting a node from a linked list is fairly straightforward. Given a node n, we find the previous node `prev` and set `prev.next` equal to `n.next`. If the list is doubly linked, we must also update `n.next` to set `n.next.prev` equal to `n.prev`. The important things to remember are (1) to check for the null pointer and (2) to update the head or tail pointer as necessary.

Additionally, if you implement this code in C, C++ or another language that requires the developer to do memory management, you should consider if the removed node should be deallocated.

```
1  Node deleteNode(Node head, int d) {
2     Node n = head;
3
4     if (n.data == d) {
5        return head.next; / *moved head */
6     }
7
8     while (n.next != null) {
9        if (n.next.data == d) {
10          n.next = n.next.next;
11          return head; / *head didn't change */
12       }
13       n = n.next;
14    }
15    return head;
16 }
```

The "Runner" Technique

The "runner" (or second pointer) technique is used in many linked list problems. The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other. The "fast" node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.

For example, suppose you had a linked list $a_1->a_2->...->a_n->b_1->b_2->...->b_n$ 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 **Remove Dups:** Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

Hints: #9, #40

pg 208

2.2 **Return Kth to Last:** Implement an algorithm to find the kth to last element of a singly linked list.

Hints: #8, #25, #41, #67, #126

pg 209

2.3 **Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list a->b->c->d->e->f

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

Hints: #72

pg 211

2.4 **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

Input: 3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1 [partition = 5]

Output: 3 -> 1 -> 2 -> 10 -> 5 -> 5 -> 8

Hints: #3, #24

pg 212

2.5 **Sum Lists:** You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in *reverse* order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: (7-> 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.

Hints: #7, #30, #71, #95, #109

pg 214

2.6 **Palindrome:** Implement a function to check if a linked list is a palindrome.

Hints: #5, #13, #29, #61, #101

pg 216

2.7 **Intersection:** Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the kth node of the first linked list is the exact same node (by reference) as the jth node of the second linked list, then they are intersecting.

Hints: #20, #45, #55, #65, #76, #93, #111, #120, #129

pg 221

2.8 **Loop Detection:** Given a circular linked list, implement an algorithm that returns the node at the beginning of the loop.

DEFINITION

Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: A -> B -> C -> D -> E -> C [the same C as earlier]

Output: C

Hints: #50, #69, #83, #90

pg 223

Additional Questions: Trees and Graphs (#4.3), Object-Oriented Design (#7.12), System Design and Scalability (#9.5), Moderate Problems (#16.25), Hard Problems (#17.12).

Hints start on page 653.

3

Stacks and Queues

Questions on stacks and queues will be much easier to handle if you are comfortable with the ins and outs of the data structure. The problems can be quite tricky, though. While some problems may be slight modifications on the original data structure, others have much more complex challenges.

Implementing a Stack

The stack data structure is precisely what it sounds like: a stack of data. In certain types of problems, it can be favorable to store data in a stack rather than in an array.

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

It uses the following operations:

- `pop()`: Remove the top item from the stack.
- `push(item)`: Add an item to the top of the stack.
- `peek()`: Return the top of the stack.
- `isEmpty()`: Return true if and only if the stack is empty.

Unlike an array, a stack does not offer constant-time access to the `i`th item. However, it does allow constant-time adds and removes, as it doesn't require shifting elements around.

We have provided simple sample code to implement a stack. Note that a stack can also be implemented using a linked list, if items were added and removed from the same side.

```
1   public class MyStack<T> {
2     private static class StackNode<T> {
3       private T data;
4       private StackNode<T> next;
5
6       public StackNode(T data) {
7         this.data = data;
8       }
9     }
10
11    private StackNode<T> top;
12
13    public T pop() {
14      if (top == null) throw new EmptyStackException();
15      T item = top.data;
```

```
16        top = top.next;
17        return item;
18     }
19
20     public void push(T item) {
21        StackNode<T> t = new StackNode<T>(item);
22        t.next = top;
23        top = t;
24     }
25
26     public T peek() {
27        if (top == null) throw new EmptyStackException();
28        return top.data;
29     }
30
31     public boolean isEmpty() {
32        return top == null;
33     }
34  }
```

One case where stacks are often useful is in certain recursive algorithms. Sometimes you need to push temporary data onto a stack as you recurse, but then remove them as you backtrack (for example, because the recursive check failed). A stack offers an intuitive way to do this.

A stack can also be used to implement a recursive algorithm iteratively. (This is a good exercise! Take a simple recursive algorithm and implement it iteratively.)

Implementing a Queue

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

It uses the operations:

- `add(item)`: Add an item to the end of the list.
- `remove()`: Remove the first item in the list.
- `peek()`: Return the top of the queue.
- `isEmpty()`: Return true if and only if the queue is empty.

A queue can also be implemented with a linked list. In fact, they are essentially the same thing, as long as items are added and removed from opposite sides.

```
1   public class MyQueue<T> {
2      private static class QueueNode<T> {
3         private T data;
4         private QueueNode<T> next;
5
6         public QueueNode(T data) {
7            this.data = data;
8         }
9      }
10
11     private QueueNode<T> first;
12     private QueueNode<T> last;
13
14     public void add(T item) {
```

```
15        QueueNode<T> t = new QueueNode<T>(item);
16        if (last != null) {
17           last.next = t;
18        }
19        last = t;
20        if (first == null) {
21           first = last;
22        }
23     }
24
25     public T remove() {
26        if (first == null) throw new NoSuchElementException();
27        T data = first.data;
28        first = first.next;
29        if (first == null) {
30           last = null;
31        }
32        return data;
33     }
34
35     public T peek() {
36        if (first == null) throw new NoSuchElementException();
37        return first.data;
38     }
39
40     public boolean isEmpty() {
41        return first == null;
42     }
43  }
```

It is especially easy to mess up the updating of the first and last nodes in a queue. Be sure to double check this.

One place where queues are often used is in breadth-first search or in implementing a cache.

In breadth-first search, for example, we used a queue to store a list of the nodes that we need to process. Each time we process a node, we add its adjacent nodes to the back of the queue. This allows us to process nodes in the order in which they are viewed.

Interview Questions

3.1 **Three in One:** Describe how you could use a single array to implement three stacks.

Hints: #2, #12, #38, #58

pg 227

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

Hints: #27, #59, #78

pg 232

3.3 **Stack of Plates:** Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

Hints: #64, #81

pg 233

3.4 **Queue via Stacks:** Implement a `MyQueue` class which implements a queue using two stacks.

Hints: #98, #114

pg 236

3.5 **Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

Hints: #15, #32, #43

pg 237

3.6 **Animal Shelter:** An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as `enqueue`, `dequeueAny`, `dequeueDog`, and `dequeueCat`. You may use the built-in `LinkedList` data structure.

Hints: #22, #56, #63

pg 239

Additional Questions: Linked Lists (#2.6), Moderate Problems (#16.26), Hard Problems (#17.9).

Hints start on page 653.

4

Trees and Graphs

Many interviewees find tree and graph problems to be some of the trickiest. Searching a tree is more complicated than searching in a linearly organized data structure such as an array or linked list. Additionally, the worst case and average case time may vary wildly, and we must evaluate both aspects of any algorithm. Fluency in implementing a tree or graph from scratch will prove essential.

Because most people are more familiar with trees than graphs (and they're a bit simpler), we'll discuss trees first. This is a bit out of order though, as a tree is actually a type of graph.

> Note: Some of the terms in this chapter can vary slightly across different textbooks and other sources. If you're used to a different definition, that's fine. Make sure to clear up any ambiguity with your interviewer.

Types of Trees

A nice way to understand a tree is with a recursive explanation. A tree is a data structure composed of nodes.

- Each tree has a root node. (Actually, this isn't strictly necessary in graph theory, but it's usually how we use trees in programming, and especially programming interviews.)
- The root node has zero or more child nodes.
- Each child node has zero or more child nodes, and so on.

The tree cannot contain cycles. The nodes may or may not be in a particular order, they could have any data type as values, and they may or may not have links back to their parent nodes.

A very simple class definition for Node is:

```
1  class Node {
2    public String name;
3    public Node[] children;
4  }
```

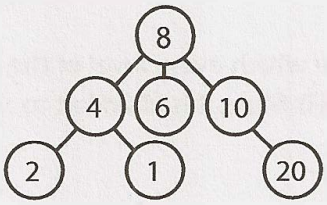
You might also have a `Tree` class to wrap this node. For the purposes of interview questions, we typically do not use a `Tree` class. You can if you feel it makes your code simpler or better, but it rarely does.

```
1  class Tree {
2    public Node root;
3  }
```

Tree and graph questions are rife with ambiguous details and incorrect assumptions. Be sure to watch out for the following issues and seek clarification when necessary.

Trees vs. Binary Trees

A binary tree is a tree in which each node has up to two children. Not all trees are binary trees. For example, this tree is not a binary tree. You could call it a ternary tree.



There are occasions when you might have a tree that is not a binary tree. For example, suppose you were using a tree to represent a bunch of phone numbers. In this case, you might use a 10-ary tree, with each node having up to 10 children (one for each digit).

A node is called a "leaf" node if it has no children.

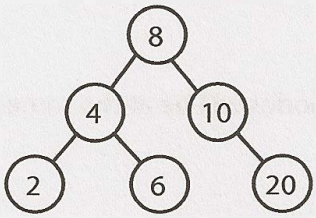
Binary Tree vs. Binary Search Tree

A binary search tree is a binary tree in which every node fits a specific ordering property: `all left descendents <= n < all right descendents`. This must be true for each node n.

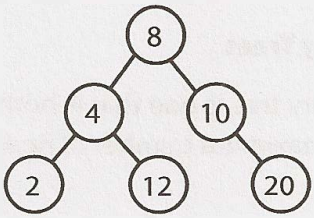
> The definition of a binary search tree can vary slightly with respect to equality. Under some definitions, the tree cannot have duplicate values. In others, the duplicate values will be on the right or can be on either side. All are valid definitions, but you should clarify this with your interviewer.

Note that this inequality must be true for all of a node's descendents, not just its immediate children. The following tree on the left below is a binary search tree. The tree on the right is not, since 12 is to the left of 8.

A binary search tree.



Not a binary search tree.



When given a tree question, many candidates assume the interviewer means a binary *search* tree. Be sure to ask. A binary search tree imposes the condition that, for each node, its left descendents are less than or equal to the current node, which is less than the right descendents.

Balanced vs. Unbalanced

While many trees are balanced, not all are. Ask your interviewer for clarification here. Note that balancing a tree does not mean the left and right subtrees are exactly the same size (like you see under "perfect binary trees" in the following diagram).

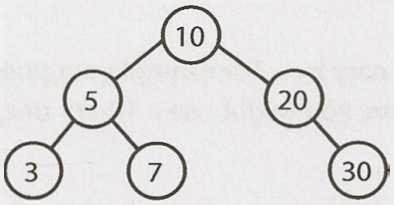
One way to think about it is that a "balanced" tree really means something more like "not terribly imbalanced." It's balanced enough to ensure `O(log n)` times for `insert` and `find`, but it's not necessarily as balanced as it could be.

Two common types of balanced trees are red-black trees (pg 639) and AVL trees (pg 637). These are discussed in more detail in the Advanced Topics section.

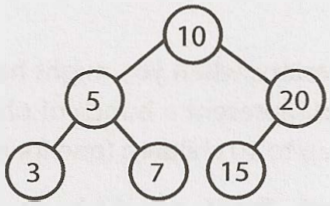
Complete Binary Trees

A complete binary tree is a binary tree in which every level of the tree is fully filled, except for perhaps the last level. To the extent that the last level is filled, it is filled left to right.

not a complete binary tree



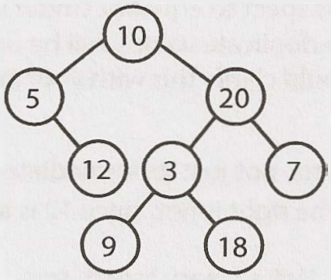
a complete binary tree



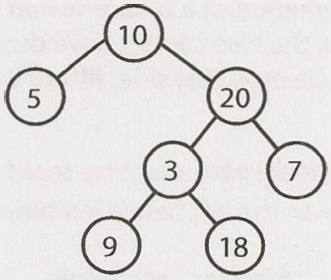
Full Binary Trees

A full binary tree is a binary tree in which every node has either zero or two children. That is, no nodes have only one child.

not a full binary tree

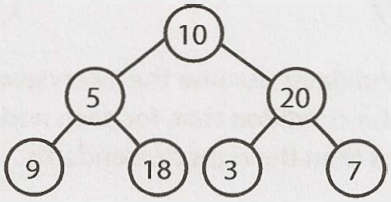


a full binary tree



Perfect Binary Trees

A perfect binary tree is one that is both full and complete. All leaf nodes will be at the same level, and this level has the maximum number of nodes.



Note that perfect trees are rare in interviews and in real life, as a perfect tree must have exactly $2^k - 1$ nodes (where k is the number of levels). In an interview, do not assume a binary tree is perfect.

Binary Tree Traversal

Prior to your interview, you should be comfortable implementing in-order, post-order, and pre-order traversal. The most common of these is in-order traversal.

In-Order Traversal

In-order traversal means to "visit" (often, print) the left branch, then the current node, and finally, the right branch.

```
1  void inOrderTraversal(TreeNode node) {
2    if (node != null) {
3      inOrderTraversal(node.left);
4      visit(node);
5      inOrderTraversal(node.right);
6    }
7  }
```

When performed on a binary search tree, it visits the nodes in ascending order (hence the name "in-order").

Pre-Order Traversal

Pre-order traversal visits the current node before its child nodes (hence the name "pre-order").

```
1  void preOrderTraversal(TreeNode node) {
2    if (node != null) {
3      visit(node);
4      preOrderTraversal(node.left);
5      preOrderTraversal(node.right);
6    }
7  }
```

In a pre-order traversal, the root is always the first node visited.

Post-Order Traversal

Post-order traversal visits the current node after its child nodes (hence the name "post-order").

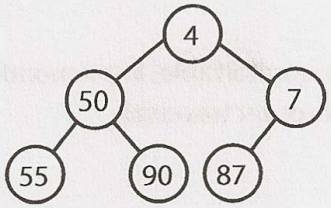
```
1  void postOrderTraversal(TreeNode node) {
2    if (node != null) {
3      postOrderTraversal(node.left);
4      postOrderTraversal(node.right);
5      visit(node);
6    }
7  }
```

In a post-order traversal, the root is always the last node visited.

Binary Heaps (Min-Heaps and Max-Heaps)

We'll just discuss min-heaps here. Max-heaps are essentially equivalent, but the elements are in descending order rather than ascending order.

A min-heap is a *complete* binary tree (that is, totally filled other than the rightmost elements on the last level) where each node is smaller than its children. The root, therefore, is the minimum element in the tree.



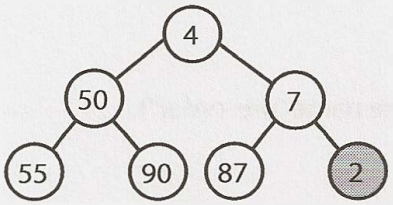
We have two key operations on a min-heap: `insert` and `extract_min`.

Insert

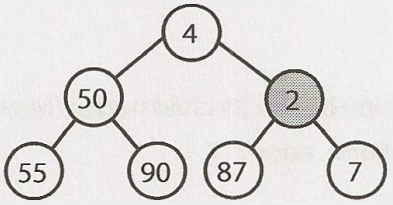
When we insert into a min-heap, we always start by inserting the element at the bottom. We insert at the rightmost spot so as to maintain the complete tree property.

Then, we "fix" the tree by swapping the new element with its parent, until we find an appropriate spot for the element. We essentially bubble up the minimum element.

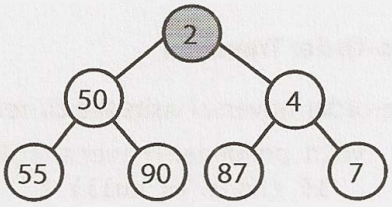
Step 1: Insert 2



Step 2: Swap 2 and 7



Step 3: Swap 2 and 4



This takes `O(log n)` time, where n is the number of nodes in the heap.

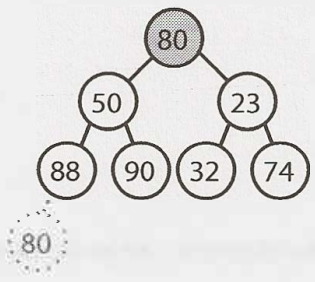
Extract Minimum Element

Finding the minimum element of a min-heap is easy: it's always at the top. The trickier part is how to remove it. (In fact, this isn't that tricky.)

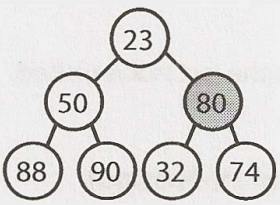
First, we remove the minimum element and swap it with the last element in the heap (the bottommost, rightmost element). Then, we bubble down this element, swapping it with one of its children until the min-heap property is restored.

Do we swap it with the left child or the right child? That depends on their values. There's no inherent ordering between the left and right element, but you'll need to take the smaller one in order to maintain the min-heap ordering.

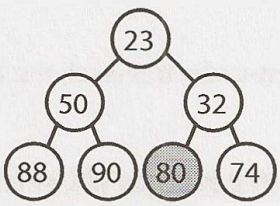
Step 1: Replace min with 80



Step 2: Swap 23 and 80



Step 3: Swap 32 and 80



This algorithm will also take `O(log n)` time.

Tries (Prefix Trees)

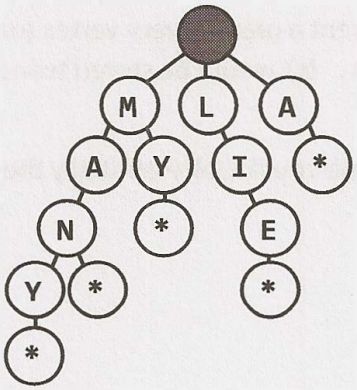
A trie (sometimes called a prefix tree) is a funny data structure. It comes up a lot in interview questions, but algorithm textbooks don't spend much time on this data structure.

A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word.

The * nodes (sometimes called "null nodes") are often used to indicate complete words. For example, the fact that there is a * node under MANY indicates that MANY is a complete word. The existence of the MA path indicates there are words that start with MA.

The actual implementation of these * nodes might be a special type of child (such as a `TerminatingTrieNode`, which inherits from `TrieNode`). Or, we could use just a boolean flag `terminates` within the "parent" node.

A node in a trie could have anywhere from 1 through `ALPHABET_SIZE + 1` children (or, 0 through `ALPHABET_SIZE` if a boolean flag is used instead of a * node).



Very commonly, a trie is used to store the entire (English) language for quick prefix lookups. While a hash table can quickly look up whether a string is a valid word, it cannot tell us if a string is a prefix of any valid words. A trie can do this very quickly.

> How quickly? A trie can check if a string is a valid prefix in `O(K)` time, where K is the length of the string. This is actually the same runtime as a hash table will take. Although we often refer to hash table lookups as being `O(1)` time, this isn't entirely true. A hash table must read through all the characters in the input, which takes `O(K)` time in the case of a word lookup.

Many problems involving lists of valid words leverage a trie as an optimization. In situations when we search through the tree on related prefixes repeatedly (e.g., looking up M, then MA, then MAN, then MANY), we might pass around a reference to the current node in the tree. This will allow us to just check if Y is a child of MAN, rather than starting from the root each time.

Graphs

A tree is actually a type of graph, but not all graphs are trees. Simply put, a tree is a connected graph without cycles.

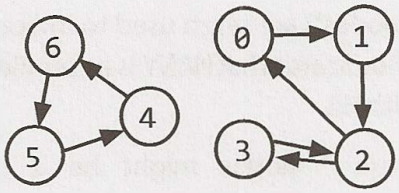
A graph is simply a collection of nodes with edges between (some of) them.

- Graphs can be either directed (like the following graph) or undirected. While directed edges are like a

one-way street, undirected edges are like a two-way street.

- The graph might consist of multiple isolated subgraphs. If there is a path between every pair of vertices, it is called a "connected graph."
- The graph can also have cycles (or not). An "acyclic graph" is one without cycles.

Visually, you could draw a graph like this:



In terms of programming, there are two common ways to represent a graph.

Adjacency List

This is the most common way to represent a graph. Every vertex (or node) stores a list of adjacent vertices. In an undirected graph, an edge like `(a, b)` would be stored twice: once in a's adjacent vertices and once in b's adjacent vertices.

A simple class definition for a graph node could look essentially the same as a tree node.

```
1  class Graph {
2    public Node[] nodes;
3  }
4
5  class Node {
6    public String name;
7    public Node[] children;
8  }
```

The Graph class is used because, unlike in a tree, you can't necessarily reach all the nodes from a single node.

You don't necessarily need any additional classes to represent a graph. An array (or a hash table) of lists (arrays, arraylists, linked lists, etc.) can store the adjacency list. The graph above could be represented as:

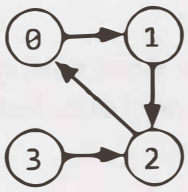
```
0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5
```

This is a bit more compact, but it isn't quite as clean. We tend to use node classes unless there's a compelling reason not to.

Adjacency Matrices

An adjacency matrix is an `NxN` boolean matrix (where `N` is the number of nodes), where a `true` value at `matrix[i][j]` indicates an edge from node `i` to node `j`. (You can also use an integer matrix with 0s and 1s.)

In an undirected graph, an adjacency matrix will be symmetric. In a directed graph, it will not (necessarily) be.



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

The same graph algorithms that are used on adjacency lists (breadth-first search, etc.) can be performed with adjacency matrices, but they may be somewhat less efficient. In the adjacency list representation, you can easily iterate through the neighbors of a node. In the adjacency matrix representation, you will need to iterate through all the nodes to identify a node's neighbors.

Graph Search

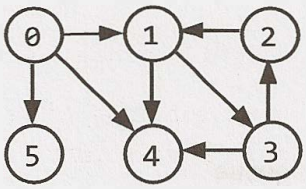
The two most common ways to search a graph are depth-first search and breadth-first search.

In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first (hence the name *depth*-first search) before we go wide.

In breadth-first search (BFS), we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide (hence *breadth*-first search) before we go deep.

See the below depiction of a graph and its depth-first and breadth-first search (assuming neighbors are iterated in numerical order).

Graph



Depth-First Search

```
1  Node 0
2    Node 1
3      Node 3
4        Node 2
5        Node 4
6  Node 5
```

Breadth-First Search

```
1  Node 0
2  Node 1
3  Node 4
4  Node 5
5  Node 3
6  Node 2
```

Breadth-first search and depth-first search tend to be used in different scenarios. DFS is often preferred if we want to visit every node in the graph. Both will work just fine, but depth-first search is a bit simpler.

However, if we want to find the shortest path (or just any path) between two nodes, BFS is generally better. Consider representing all the friendships in the entire world in a graph and trying to find a path of friendships between Ash and Vanessa.

In depth-first search, we could take a path like `Ash -> Brian -> Carleton -> Davis -> Eric -> Farah -> Gayle -> Harry -> Isabella -> John -> Kari...` and then find ourselves very far away. We could go through most of the world without realizing that, in fact, Vanessa is Ash's friend. We will still eventually find the path, but it may take a long time. It also won't find us the shortest path.

In breadth-first search, we would stay close to Ash for as long as possible. We might iterate through many of Ash's friends, but we wouldn't go to his more distant connections until absolutely necessary. If Vanessa is Ash's friend, or his friend-of-a-friend, we'll find this out relatively quickly.

Depth-First Search (DFS)

In DFS, we visit a node a and then iterate through each of a's neighbors. When visiting a node b that is a neighbor of a, we visit all of b's neighbors before going on to a's other neighbors. That is, a exhaustively searches b's branch before any of its other neighbors.

Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited. If we don't, we risk getting stuck in an infinite loop.

The pseudocode below implements DFS.

```
1  void search(Node root) {
2    if (root == null) return;
3    visit(root);
4    root.visited = true;
5    for each (Node n in root.adjacent) {
6      if (n.visited == false) {
7        search(n);
8      }
9    }
10 }
```

Breadth-First Search (BFS)

BFS is a bit less intuitive, and many interviewees struggle with the implementation unless they are already familiar with it. The main tripping point is the (false) assumption that BFS is recursive. It's not. Instead, it uses a queue.

In BFS, node a visits each of a's neighbors before visiting any of *their* neighbors. You can think of this as searching level by level out from a. An iterative solution involving a queue usually works best.

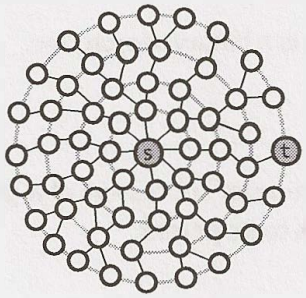
```
1  void search(Node root) {
2    Queue queue = new Queue();
3    root.marked = true;
4    queue.enqueue(root); // Add to the end of queue
5
6    while (!queue.isEmpty()) {
7      Node r = queue.dequeue(); // Remove from the front of the queue
8      visit(r);
9      foreach (Node n in r.adjacent)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

If you are asked to implement BFS, the key thing to remember is the use of the queue. The rest of the algorithm flows from this fact.

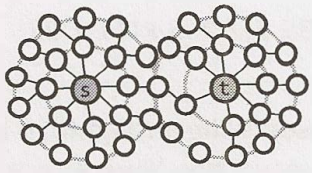
Bidirectional Search

Bidirectional search is used to find the shortest path between a source and destination node. It operates by essentially running two simultaneous breadth-first searches, one from each node. When their searches collide, we have found a path.

Breadth-First Search
Single search from s to t that collides after four levels.



Bidirectional Search
Two searches (one from s and one from t) that collide after four levels total (two levels each).



To see why this is faster, consider a graph where every node has at most k adjacent nodes and the shortest path from node s to node t has length d.

- In traditional breadth-first search, we would search up to k nodes in the first "level" of the search. In the second level, we would search up to k nodes for each of those first k nodes, so k^2 nodes total (thus far). We would do this d times, so that's $O(k^d)$ nodes.
- In bidirectional search, we have two searches that collide after approximately $^d/_2$ levels (the midpoint of the path). The search from s visits approximately $k^{d/2}$, as does the search from t. That's approximately $2\ k^{d/2}$, or $O(k^{d/2})$, nodes total.

This might seem like a minor difference, but it's not. It's huge. Recall that $(k^{d/2})*(k^{d/2}) = k^d$. The bidirectional search is actually faster by a factor of $k^{d/2}$.

Put another way: if our system could only support searching "friend of friend" paths in breadth-first search, it could now likely support "friend of friend of friend of friend" paths. We can support paths that are twice as long.

Additional Reading: Topological Sort (pg 632), Dijkstra's Algorithm (pg 633), AVL Trees (pg 637), Red-Black Trees (pg 639).

Interview Questions

4.1 **Route Between Nodes:** Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

Hints: #127

pg 241

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

Hints: #19, #73, #116

pg 242

4.3 **List of Depths:** Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).

Hints: #107, #123, #135

pg 243

4.4 **Check Balanced:** Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

Hints: #21, #33, #49, #105, #124

pg 244

4.5 **Validate BST:** Implement a function to check if a binary tree is a binary search tree.

Hints: #35, #57, #86, #113, #128

pg 245

4.6 **Successor:** Write an algorithm to find the "next" node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

Hints: #79, #91

pg 248

4.7 **Build Order:** You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

```
projects: a, b, c, d, e, f
dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)
```

Output: `f, e, a, b, d, c`

Hints: #26, #47, #60, #85, #125, #133

pg 250

4.8 **First Common Ancestor:** Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

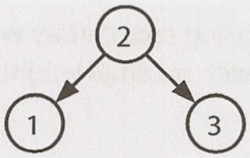
Hints: #10, #16, #28, #36, #46, #70, #80, #96

pg 257

4.9 **BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:



Output: `{2, 1, 3}, {2, 3, 1}`

Hints: #39, #48, #66, #82

pg 262

4.10 Check Subtree: T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

Hints: #4, #11, #18, #31, #37

pg 265

4.11 Random Node: You are implementing a binary tree class from scratch which, in addition to `insert`, `find`, and `delete`, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

Hints: #42, #54, #62, #75, #89, #99, #112, #119

pg 268

4.12 Paths with Sum: You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

Hints: #6, #14, #52, #68, #77, #87, #94, #103, #108, #115

pg 272

Additional Questions: Recursion (#8.10), System Design and Scalability (#9.2, #9.3), Sorting and Searching (#10.10), Hard Problems (#17.7, #17.12, #17.13, #17.14, #17.17, #17.20, #17.22, #17.25).

Hints start on page 653.

5

Bit Manipulation

Bit manipulation is used in a variety of problems. Sometimes, the question explicitly calls for bit manipulation. Other times, it's simply a useful technique to optimize your code. You should be comfortable doing bit manipulation by hand, as well as with code. Be careful; it's easy to make little mistakes.

Bit Manipulation By Hand

If you're rusty on bit manipulation, try the following exercises by hand. The items in the third column can be solved manually or with "tricks" (described below). For simplicity, assume that these are four-bit numbers.

If you get confused, work them through as a base 10 number. You can then apply the same process to a binary number. Remember that ^ indicates an XOR, and ~ is a NOT (negation).

`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. `0100` equals 4, and multiplying by 4 is just left shifting by 2. So we shift `0011` left by 2 to get `1100`.
3. Think about this operation bit by bit. If you XOR a bit with its own negated value, you will always get 1. Therefore, the solution to `a^(~a)` will be a sequence of 1s.
4. `~0` is a sequence of 1s, so `~0 << 2` is 1s followed by two 0s. ANDing that with another value will clear the last two bits of the value.

If you didn't see these tricks immediately, think about them logically.

Bit Facts and Tricks

The following expressions are useful in bit manipulation. Don't just memorize them, though; think deeply about why each of these is true. We use "1s" and "0s" to indicate a sequence of 1s or 0s, respectively.

```
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.

Two's Complement and Negative Numbers

Computers typically store integers in two's complement representation. A positive number is represented as itself while a negative number is represented as the two's complement of its absolute value (with a 1 in its sign bit to indicate that a negative value). The two's complement of an N-bit number (where N is the number of bits used for the number, *excluding* the sign bit) is the complement of the number with respect to 2^N.

Let's look at the 4-bit integer `-3` as an example. If it's a 4-bit number, we have one bit for the sign and three bits for the value. We want the complement with respect to 2^3, which is 8. The complement of 3 (the absolute value of `-3`) with respect to 8 is 5. 5 in binary is `101`. Therefore, -3 in binary as a 4-bit number is 1101, with the first bit being the sign bit.

In other words, the binary representation of -K (negative K) as a N-bit number is `concat(1, 2^(N-1) - K)`.

Another way to look at this is that we invert the bits in the positive representation and then add 1. 3 is `011` in binary. Flip the bits to get `100`, add `1` to get `101`, then prepend the sign bit (`1`) to get `1101`.

In a four-bit integer, this would look like the following.

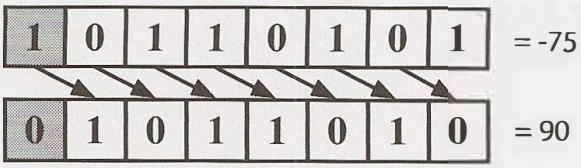
Positive Values		Negative Values	
7	0 111	-1	1 111
6	0 110	-2	1 110
5	0 101	-3	1 101
4	0 100	-4	1 100
3	0 011	-5	1 011
2	0 010	-6	1 010
1	0 001	-7	1 001
0	0 000		

Observe that the absolute values of the integers on the left and right always sum to 2^3, and that the binary values on the left and right sides are identical, other than the sign bit. Why is that?

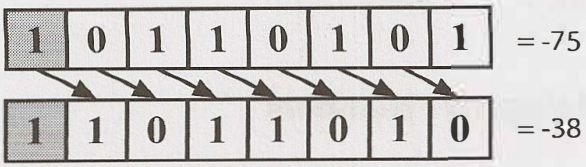
Arithmetic vs. Logical Right Shift

There are two types of right shift operators. The arithmetic right shift essentially divides by two. The logical right shift does what we would visually see as shifting the bits. This is best seen on a negative number.

In a logical right shift, we shift the bits and put a `0` in the most significant bit. It is indicated with a `>>>` operator. On an 8-bit integer (where the sign bit is the most significant bit), this would look like the image below. The sign bit is indicated with a gray background.



In an arithmetic right shift, we shift values to the right but fill in the new bits with the value of the sign bit. This has the effect of (roughly) dividing by two. It is indicated by a >> operator.



What do you think these functions would do on parameters `x = -93242` and `count = 40`?

```
1  int repeatedArithmeticShift(int x, int count) {
2    for (int i = 0; i < count; i++) {
3      x >>= 1; // Arithmetic shift by 1
4    }
5    return x;
6  }
7
8  int repeatedLogicalShift(int x, int count) {
9    for (int i = 0; i < count; i++) {
10     x >>>= 1; // Logical shift by 1
11   }
12   return x;
13 }
```

With the logical shift, we would get `0` because we are shifting a zero into the most significant bit repeatedly.

With the arithmetic shift, we would get `-1` because we are shifting a one into the most significant bit repeatedly. A sequence of all 1s in a (signed) integer represents `-1`.

▸ Common Bit Tasks: Getting and Setting

The following operations are very important to know, but do not simply memorize them. Memorizing leads to mistakes that are impossible to recover from. Rather, understand *how* to implement these methods, so that you can implement these, and other, bit problems.

Get Bit

This method shifts `1` over by `i` bits, creating a value that looks like `00010000`. By performing an AND with num, we clear all bits other than the bit at bit `i`. Finally, we compare that to `0`. If that new value is not zero, then bit `i` must have a 1. Otherwise, bit `i` is a 0.

```
1  boolean getBit(int num, int i) {
2    return ((num & (1 << i)) != 0);
3  }
```

Set Bit

`SetBit` shifts `1` over by `i` bits, creating a value like `00010000`. By performing an OR with num, only the value at bit `i` will change. All other bits of the mask are zero and will not affect num.

```
1  int setBit(int num, int i) {
2    return num | (1 << i);
3  }
```

Clear Bit

This method operates in almost the reverse of `setBit`. First, we create a number like `11101111` by creating the reverse of it (`00010000`) and negating it. Then, we perform an AND with `num`. This will clear the `i`th bit and leave the remainder unchanged.

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

To clear all bits from the most significant bit through `i` (inclusive), we create a mask with a `1` at the `i`th bit (`1 << i`). Then, we subtract `1` from it, giving us a sequence of `0`s followed by `i` `1`s. We then AND our number with this mask to leave just the last `i` bits.

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

To clear all bits from `i` through `0` (inclusive), we take a sequence of all `1`s (which is `-1`) and shift it left by `i + 1` bits. This gives us a sequence of `1`s (in the most significant bits) followed by `i` `0` bits.

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

Update Bit

To set the `i`th bit to a value `v`, we first clear the bit at position `i` by using a mask that looks like `11101111`. Then, we shift the intended value, `v`, left by `i` bits. This will create a number with bit `i` equal to `v` and all other bits equal to `0`. Finally, we OR these two numbers, updating the `i`th bit if `v` is `1` and leaving it as `0` otherwise.

```
1  int updateBit(int num, int i, boolean bitIs1) {
2    int value = bitIs1 ? 1 : 0;
3    int mask = ~(1 << i);
4    return (num & mask) | (value << i);
5  }
```

Interview Questions

5.1 **Insertion:** You are given two 32-bit numbers, `N` and `M`, and two bit positions, `i` and `j`. Write a method to insert `M` into `N` such that `M` starts at bit `j` and ends at bit `i`. You can assume that the bits `j` through `i` have enough space to fit all of `M`. That is, if `M = 10011`, you can assume that there are at least 5 bits between `j` and `i`. You would not, for example, have `j = 3` and `i = 2`, because `M` could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: `N = 10000000000, M = 10011, i = 2, j = 6`

Output: `N = 10001001100`

Hints: #137, #169, #215

pg 276

5.2 **Binary to String:** Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

Hints: #143, #167, #173, #269, #297

pg 277

5.3 **Flip Bit to Win:** You have an integer and you can flip exactly one bit from a `0` to a `1`. Write code to find the length of the longest sequence of `1`s you could create.

EXAMPLE

Input: `1775 (or: 11011101111)`

Output: `8`

Hints: #159, #226, #314, #352

pg 278

5.4 **Next Number:** Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

Hints: #147, #175, #242, #312, #339, #358, #375, #390

pg 280

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

Hints: #151, #202, #261, #302, #346, #372, #383, #398

pg 285

5.6 **Conversion:** Write a function to determine the number of bits you would need to flip to convert integer A to integer B.

EXAMPLE

Input: `29 (or: 11101), 15 (or: 01111)`

Output: `2`

Hints: #336, #369

pg 286

5.7 **Pairwise Swap:** Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit `0` and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on).

Hints: #145, #248, #328, #355

pg 286

5.8 **Draw Line:** A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width `w`, where `w` is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function that draws a horizontal line from `(x1, y)` to `(x2, y)`.

The method signature should look something like:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

Hints: #366, #381, #384, #391

pg 287

Additional Questions: Arrays and Strings (#1.1, #1.4, #1.8), Math and Logic Puzzles (#6.10), Recursion (#8.4, #8.14), Sorting and Searching (#10.7, #10.8), C++ (#12.10), Moderate Problems (#16.1, #16.7), Hard Problems (#17.1).

Hints start on page 662.

6

Math and Logic Puzzles

So-called "puzzles" (or brain teasers) are some of the most hotly debated questions, and many companies have policies banning them. Unfortunately, even when these questions are banned, you still may find yourself being asked one of them. Why? Because no one can agree on a definition of what a brainteaser is.

The good news is that if you are asked a puzzle or brainteaser, it's likely to be a reasonably fair one. It probably won't rely on a trick of wording, and it can almost always be logically deduced. Many have their foundations in mathematics or computer science, and almost all have solutions that can be logically deduced.

We'll go through some common approaches for tackling these questions, as well as some of the essential knowledge.

Prime Numbers

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

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

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:

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

$$= xy$$

Checking for Primality

This question is so common that we feel the need to specifically cover it. The naive way is to simply iterate from 2 through n-1, checking for divisibility on each iteration.

```
1  boolean primeNaive(int n) {
2    if (n < 2) {
3      return false;
4    }
5    for (int i = 2; i < n; i++) {
6      if (n % i == 0) {
7        return false;
8      }
9    }
10   return true;
11 }
```

A small but important improvement is to iterate only up through the square root of n.

```
1  boolean primeSlightlyBetter(int n) {
2    if (n < 2) {
3      return false;
4    }
5    int sqrt = (int) Math.sqrt(n);
6    for (int i = 2; i <= sqrt; i++) {
7      if (n % i == 0) return false;
8    }
9    return true;
10 }
```

The $\sqrt{n}$ is sufficient because, for every number a which divides n evenly, there is a complement b, where a * b = n. If a > $\sqrt{n}$, then b < $\sqrt{n}$ (since $(\sqrt{n})^2 = n$). We therefore don't need a to check n's primality, since we would have already checked with b.

Of course, in reality, all we *really* need to do is to check if n is divisible by a prime number. This is where the Sieve of Eratosthenes comes in.

Generating a List of Primes: The Sieve of Eratosthenes

The Sieve of Eratosthenes is a highly efficient way to generate a list of primes. It works by recognizing that all non-prime numbers are divisible by a prime number.

We start with a list of all the numbers up through some value `max`. First, we cross off all numbers divisible by 2. Then, we look for the next prime (the next non-crossed off number) and cross off all numbers divisible by it. By crossing off all numbers divisible by 2, 3, 5, 7, 11, and so on, we wind up with a list of prime numbers from 2 through `max`.

The code below implements the Sieve of Eratosthenes.

```
1  boolean[] sieveOfEratosthenes(int max) {
2    boolean[] flags = new boolean[max + 1];
3    int count = 0;
4
5    init(flags); // Set all flags to true other than 0 and 1
6    int prime = 2;
7
8    while (prime <= Math.sqrt(max)) {
```

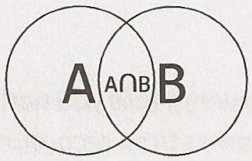
```
9        /* Cross off remaining multiples of prime */
10       crossOff(flags, prime);
11
12       /* Find next value which is true */
13       prime = getNextPrime(flags, prime);
14     }
15
16     return flags;
17   }
18
19   void crossOff(boolean[] flags, int prime) {
20     /* Cross off remaining multiples of prime. We can start with (prime*prime),
21      * because if we have a k * prime, where k < prime, this value would have
22      * already been crossed off in a prior iteration. */
23     for (int i = prime * prime; i < flags.length; i += prime) {
24       flags[i] = false;
25     }
26   }
27
28   int getNextPrime(boolean[] flags, int prime) {
29     int next = prime + 1;
30     while (next < flags.length && !flags[next]) {
31       next++;
32     }
33     return next;
34   }
```

Of course, there are a number of optimizations that can be made to this. One simple one is to only use odd numbers in the array, which would allow us to reduce our space usage by half.

Probability

Probability can be a complex topic, but it's based in a few basic laws that can be logically derived.

Let's look at a Venn diagram to visualize two events A and B. The areas of the two circles represent their relative probability, and the overlapping area is the event {A and B}.



Probability of A and B

Imagine you were throwing a dart at this Venn diagram. What is the probability that you would land in the intersection between A and B? If you knew the odds of landing in A, and you also knew the percent of A that's also in B (that is, the odds of being in B given that you were in A), then you could express the probability as:

```
P(A 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
```

Observe that since P(A and B) = P(B given A) P(A) = P(A given B) P(B), you can express the probability of A given B in terms of the reverse:

```
P(A given B) = P(B given A) P(A) / P(B)
```

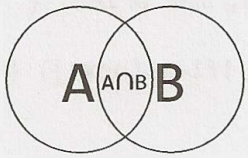
The above equation is called Bayes' Theorem.

Probability of A or B

Now, imagine you wanted to know what the probability of landing in A or B is. If you knew the odds of landing in each individually, and you also knew the odds of landing in their intersection, then you could express the probability as:

```
P(A 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.

Start Talking

Don't panic when you get a brainteaser. Like algorithm questions, interviewers want to see how you tackle a problem; they don't expect you to immediately know the answer. Start talking, and show the interviewer how you approach a problem.

Develop Rules and Patterns

In many cases, you will find it useful to write down "rules" or patterns that you discover while solving the problem. And yes, you really should write these down—it will help you remember them as you solve the problem. Let's demonstrate this approach with an example.

You have two ropes, and each takes exactly one hour to burn. How would you use them to time exactly 15 minutes? Note that the ropes are of uneven densities, so half the rope length-wise does not necessarily take half an hour to burn.

> *Tip: Stop here and spend some time trying to solve this problem on your own. If you absolutely must, read through this section for hints—but do so slowly. Every paragraph will get you a bit closer to the solution.*

From the statement of the problem, we immediately know that we can time one hour. We can also time two hours, by lighting one rope, waiting until it is burnt, and then lighting the second. We can generalize this into a rule.

Rule 1: Given a rope that takes x minutes to burn and another that takes y minutes, we can time x+y minutes.

What else can we do with the rope? We can probably assume that lighting a rope in the middle (or anywhere other than the ends) won't do us much good. The flames would expand in both directions, and we have no idea how long it would take to burn.

However, we can light a rope at both ends. The two flames would meet after 30 minutes.

Rule 2: Given a rope that takes x minutes to burn, we can time $x/2$ 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 brainteasers are worst-case minimization problems, worded either in terms of *minimizing* an action or in doing something at most a specific number of times. A useful technique is to try to "balance" the worst case. That is, if an early decision results in a skewing of the worst case, we can sometimes change the decision to balance out the worst case. This will be clearest when explained with an example.

The "nine balls" question is a classic interview question. You have nine balls. Eight are of the same weight, and one is heavier. You are given a balance which tells you only whether the left side or the right side is heavier. Find the heavy ball in just two uses of the scale.

A first approach is to divide the balls in sets of four, with the ninth ball sitting off to the side. The heavy ball is in the heavier set. If they are the same weight, then we know that the ninth ball is the heavy one. Replicating this approach for the remaining sets would result in a worst case of three weighings—one too many!

This is an imbalance in the worst case: the ninth ball takes just one weighing to discover if it's heavy, whereas others take three. If we *penalize* the ninth ball by putting more balls off to the side, we can lighten the load on the others. This is an example of "worst case balancing."

If we divide the balls into sets of three items each, we will know after just one weighing which set has the heavy one. We can even formalize this into a *rule*: given N balls, where N is divisible by 3, one use of the scale will point us to a set of $N/3$ balls with the heavy ball.

For the final set of three balls, we simply repeat this: put one ball off to the side and weigh two. Pick the heavier of the two. Or, if the balls are the same weight, pick the third one.

Algorithm Approaches

If you're stuck, consider applying one of the approaches for solving algorithm questions (starting on page 67). Brainteasers are often nothing more than algorithm questions with the technical aspects removed. Base Case and Build and Do It Yourself (DIY) can be especially useful.

Additional Reading: Useful Math (pg 629).

Interview Questions

6.1 **The Heavy Pill:** You have 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.

Hints: #186, #252, #319, #387

pg 289

6.2 **Basketball:** You have a basketball hoop and someone says that you can play one of two games.

Game 1: You get one shot to make the hoop.

Game 2: You get three shots and you have to make two of three shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

Hints: #181, #239, #284, #323

pg 290

6.3 **Dominos:** There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).

Hints: #367, #397

pg 291

6.4 **Ants on a Triangle:** There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle? Assume that each ant randomly picks a direction, with either direction being equally likely to be chosen, and that they walk at the same speed.

Similarly, find the probability of collision with n ants on an n-vertex polygon.

Hints: #157, #195, #296

pg 291

6.5 **Jugs of Water:** You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly "half" of the jug would be impossible.

Hints: #149, #379, #400

pg 292

6.6 **Blue-Eyed Island:** A bunch of people are living on an island, when a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00 pm every evening. Each person can see everyone else's eye color, but they do not know their own (nor is anyone allowed to tell them). Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?

Hints: #218, #282, #341, #370

pg 293

6.7 **The Apocalypse**: In the new post-apocalyptic world, the world queen is desperately concerned about the birth rate. Therefore, she decrees that all families should ensure that they have one girl or else they face massive fines. If all families abide by this policy—that is, they have continue to have children until they have one girl, at which point they immediately stop—what will the gender ratio of the new generation be? (Assume that the odds of someone having a boy or a girl on any given pregnancy is equal.) Solve this out logically and then write a computer simulation of it.

Hints: #154, #160, #171, #188, #201

pg 293

6.8 **The Egg Drop Problem:** There is a building of 100 floors. If an egg drops from the N th floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.

Hints: #156, #233, #294, #333, #357, #374, #395

pg 296

6.9 **100 Lockers:** There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass `i`, the man toggles every `i`th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

Hints: #139, #172, #264, #306

pg 297

6.10 **Poison**: You have 1000 bottles of soda, and exactly one is poisoned. You have 10 test strips which can be used to detect poison. A single drop of poison will turn the test strip positive permanently. You can put any number of drops on a test strip at once and you can reuse a test strip as many times as you'd like (as long as the results are negative). However, you can only run tests once per day and it takes seven days to return a result. How would you figure out the poisoned bottle in as few days as possible?

FOLLOW UP

Write code to simulate your approach.

Hints: #146, #163, #183, #191, #205, #221, #230, #241, #249

pg 298

Additional Problems: Moderate Problems (#16.5), Hard Problems (#17.19)

Hints start on page 662.

7

Object-Oriented Design

Object-oriented design questions require a candidate to sketch out the classes and methods to implement technical problems or real-life objects. These problems give—or at least are believed to give—an interviewer insight into your coding style.

These questions are not so much about regurgitating design patterns as they are about demonstrating that you understand how to create elegant, maintainable object-oriented code. Poor performance on this type of question may raise serious red flags.

How to Approach

Regardless of whether the object is a physical item or a technical task, object-oriented design questions can be tackled in similar ways. The following approach will work well for many problems.

Step 1: Handle Ambiguity

Object-oriented design (OOD) questions are often intentionally vague in order to test whether you'll make assumptions or if you'll ask clarifying questions. After all, a developer who just codes something without understanding what she is expected to create wastes the company's time and money, and may create much more serious issues.

When being asked an object-oriented design question, you should inquire *who* is going to use it and *how* they are going to use it. Depending on the question, you may even want to go through the "six Ws": who, what, where, when, how, why.

For example, suppose you were asked to describe the object-oriented design for a coffee maker. This seems straightforward enough, right? Not quite.

Your coffee maker might be an industrial machine designed to be used in a massive restaurant servicing hundreds of customers per hour and making ten different kinds of coffee products. Or it might be a very simple machine, designed to be used by the elderly for just simple black coffee. These use cases will significantly impact your design.

Step 2: Define the Core Objects

Now that we understand what we're designing, we should consider what the "core objects" in a system are. For example, suppose we are asked to do the object-oriented design for a restaurant. Our core objects might be things like `Table`, `Guest`, `Party`, `Order`, `Meal`, `Employee`, `Server`, and `Host`.

Step 3: Analyze Relationships

Having more or less decided on our core objects, we now want to analyze the relationships between the objects. Which objects are members of which other objects? Do any objects inherit from any others? Are relationships many-to-many or one-to-many?

For example, in the restaurant question, we may come up with the following design:

- `Party` should have an array of `Guests`.
- `Server` and `Host` inherit from `Employee`.
- Each `Table` has one `Party`, but each `Party` may have multiple `Tables`.
- There is one `Host` for the `Restaurant`.

Be very careful here—you can often make incorrect assumptions. For example, a single `Table` may have multiple `Parties` (as is common in the trendy "communal tables" at some restaurants). You should talk to your interviewer about how general purpose your design should be.

Step 4: Investigate Actions

At this point, you should have the basic outline of your object-oriented design. What remains is to consider the key actions that the objects will take and how they relate to each other. You may find that you have forgotten some objects, and you will need to update your design.

For example, a `Party` walks into the `Restaurant`, and a `Guest` requests a `Table` from the `Host`. The `Host` looks up the `Reservation` and, if it exists, assigns the `Party` to a `Table`. Otherwise, the `Party` is added to the end of the list. When a `Party` leaves, the `Table` is freed and assigned to a new `Party` in the list.

Design Patterns

Because interviewers are trying to test your capabilities and not your knowledge, design patterns are mostly beyond the scope of an interview. However, the Singleton and Factory Method design patterns are widely used in interviews, so we will cover them here.

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

Be careful you don't fall into a trap of constantly trying to find the "right" design pattern for a particular problem. You should create the design that works for that problem. In some cases it might be an established pattern, but in many other cases it is not.

Singleton Class

The Singleton pattern ensures that a class has only one instance and ensures access to the instance through the application. It can be useful in cases where you have a "global" object with exactly one instance. For example, we may want to implement `Restaurant` such that it has exactly one instance of `Restaurant`.

```
1  public class Restaurant {
2     private static Restaurant _instance = null;
3     protected Restaurant() { ... }
4     public static Restaurant getInstance() {
5        if (_instance == null) {
6           _instance = new Restaurant();
7        }
```

```
8         return _instance;
9      }
10  }
```

It should be noted that many people dislike the Singleton design pattern, even calling it an "anti-pattern." One reason for this is that it can interfere with unit testing.

Factory Method

The Factory Method offers an interface for creating an instance of a class, with its subclasses deciding which class to instantiate. You might want to implement this with the creator class being abstract and not providing an implementation for the Factory method. Or, you could have the Creator class be a concrete class that provides an implementation for the Factory method. In this case, the Factory method would take a parameter representing which class to instantiate.

```
1   public class CardGame {
2      public static CardGame createCardGame(GameType type) {
3         if (type == GameType.Poker) {
4            return new PokerGame();
5         } else if (type == GameType.BlackJack) {
6            return new BlackJackGame();
7         }
8         return null;
9      }
10  }
```

Interview Questions

7.1 **Deck of Cards:** Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

Hints: #153, #275

pg 305

7.2 **Call Center:** Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.

Hints: #363

pg 307

7.3 **Jukebox:** Design a musical jukebox using object-oriented principles.

Hints: #198

pg 310

7.4 **Parking Lot:** Design a parking lot using object-oriented principles.

Hints: #258

pg 312

7.5 **Online Book Reader:** Design the data structures for an online book reader system.

Hints: #344

pg 318

7.6 **Jigsaw:** Implement an NxN jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a `fitsWith` method which, when passed two puzzle edges, returns true if the two edges belong together.

Hints: #192, #238, #283

pg 318

7.7 **Chat Server:** Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

Hints: #213, #245, #271

pg 326

7.8 **Othello:** Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.

Hints: #179, #228

pg 326

7.9 **Circular Array:** Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. If possible, the class should use a generic type (also called a template), and should support iteration via the standard `for (Obj o : circularArray)` notation.

Hints: #389

pg 329

7.10 **Minesweeper:** Design and implement a text-based Minesweeper game. Minesweeper is the classic single-player computer game where an NxN grid has B mines (or bombs) hidden across the grid. The remaining cells are either blank or have a number behind them. The numbers reflect the number of bombs in the surrounding eight cells. The user then uncovers a cell. If it is a bomb, the player loses. If it is a number, the number is exposed. If it is a blank cell, this cell and all adjacent blank cells (up to and including the surrounding numeric cells) are exposed. The player wins when all non-bomb cells are exposed. The player can also flag certain places as potential bombs. This doesn't affect game play, other than to block the user from accidentally clicking a cell that is thought to have a bomb. (Tip for the reader: if you're not familiar with this game, please play a few rounds online first.)

This is a fully exposed board with 3 bombs. This is not shown to the user.

	1	1	1			
	1	*	1			
	2	2	2			
	1	*	1			
	1	1	1			
			1	1	1	
			1	*	1	

The player initially sees a board with nothing exposed.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Clicking on cell (row = 1, col = 0) would expose this:

	1	?	?	?	?	?
	1	?	?	?	?	?
	2	?	?	?	?	?
	1	?	?	?	?	?
	1	1	1	?	?	?
			1	?	?	?
			1	?	?	?

The user wins when everything other than bombs has been exposed.

	1	1	1			
	1	?	1			
	2	2	2			
	1	?	1			
	1	1	1			
			1	1	1	
			1	?	1	

Hints: #351, #361, #377, #386, #399

pg 332

7.11 **File System:** Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

Hints: #141, #216

pg 337

7.12 **Hash Table:** Design and implement a hash table which uses chaining (linked lists) to handle collisions.

Hints: #287, #307

pg 339

Additional Questions: Threads and Locks (#16.3)

Hints start on page 662.

8

Recursion and Dynamic Programming

While there are a large number of recursive problems, many follow similar patterns. A good hint that a problem is recursive is that it can be built off of subproblems.

When you hear a problem beginning with the following statements, it's often (though not always) a good candidate for recursion: "Design an algorithm to compute the nth ...", "Write code to list the first n...", "Implement a method to compute all...", and so on.

> Tip: In my experience coaching candidates, people typically have about 50% accuracy in their "this sounds like a recursive problem" instinct. Use that instinct, since that 50% is valuable. But don't be afraid to look at the problem in a different way, even if you initially thought it seemed recursive. There's also a 50% chance that you were wrong.

Practice makes perfect! The more problems you do, the easier it will be to recognize recursive problems.

How to Approach

Recursive solutions, by definition, are built off of solutions to subproblems. Many times, this will mean simply to compute `f(n)` by adding something, removing something, or otherwise changing the solution for `f(n-1)`. In other cases, you might solve the problem for the first half of the data set, then the second half, and then merge those results.

There are many ways you might divide a problem into subproblems. Three of the most common approaches to develop an algorithm are bottom-up, top-down, and half-and-half.

Bottom-Up Approach

The bottom-up approach is often the most intuitive. We start with knowing how to solve the problem for a simple case, like a list with only one element. Then we figure out how to solve the problem for two elements, then for three elements, and so on. The key here is to think about how you can *build* the solution for one case off of the previous case (or multiple previous cases).

Top-Down Approach

The top-down approach can be more complex since it's less concrete. But sometimes, it's the best way to think about the problem.

In these problems, we think about how we can divide the problem for case `N` into subproblems.

Be careful of overlap between the cases.

Half-and-Half Approach

In addition to top-down and bottom-up approaches, it's often effective to divide the data set in half.

For example, binary search works with a "half-and-half" approach. When we look for an element in a sorted array, we first figure out which half of the array contains the value. Then we recurse and search for it in that half.

Merge sort is also a "half-and-half" approach. We sort each half of the array and then merge together the sorted halves.

Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n, it uses at least O(n) memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex. Before diving into recursive code, ask yourself how hard it would be to implement it iteratively, and discuss the tradeoffs with your interviewer.

Dynamic Programming & Memoization

Although people make a big deal about how scary dynamic programming problems are, there's really no need to be afraid of them. In fact, once you get the hang of them, these can actually be very easy problems.

Dynamic programming is mostly just a matter of taking a recursive algorithm and finding the overlapping subproblems (that is, the repeated calls). You then cache those results for future recursive calls.

Alternatively, you can study the pattern of the recursive calls and implement something iterative. You still "cache" previous work.

> A note on terminology: Some people call top-down dynamic programming "memoization" and only use "dynamic programming" to refer to bottom-up work. We do not make such a distinction here. We call both dynamic programming.

One of the simplest examples of dynamic programming is computing the nth Fibonacci number. A good way to approach such a problem is often to implement it as a normal recursive solution, and then add the caching part.

Fibonacci Numbers

Let's walk through an approach to compute the nth Fibonacci number.

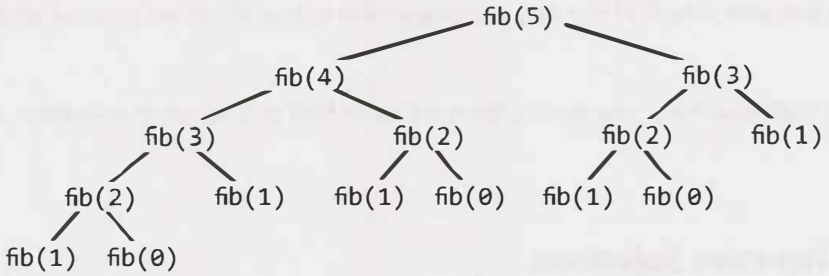
Recursive

We will start with a recursive implementation. Sounds simple, right?

```
1  int fibonacci(int i) {
2    if (i == 0) return 0;
3    if (i == 1) return 1;
4    return fibonacci(i - 1) + fibonacci(i - 2);
5  }
```

What is the runtime of this function? Think for a second before you answer.

If you said $O(n)$ or $O(n^2)$ (as many people do), think again. Study the code path that the code takes. Drawing the code paths as a tree (that is, the recursion tree) is useful on this and many recursive problems.



Observe that the leaves on the tree are all `fib(1)` and `fib(0)`. Those signify the base cases.

The total number of nodes in the tree will represent the runtime, since each call only does $O(1)$ work outside of its recursive calls. Therefore, the number of calls is the runtime.

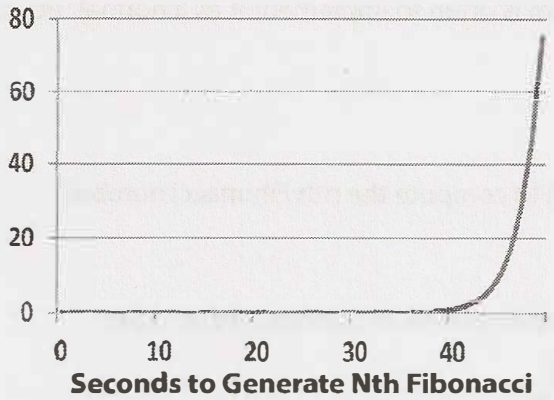
> Tip: Remember this for future problems. Drawing the recursive calls as a tree is a great way to figure out the runtime of a recursive algorithm.

How many nodes are in the tree? Until we get down to the base cases (leaves), each node has two children. Each node branches out twice.

The root node has two children. Each of those children has two children (so four children total in the "grandchildren" level). Each of those grandchildren has two children, and so on. If we do this n times, we'll have roughly $O(2^n)$ nodes. This gives us a runtime of roughly $O(2^n)$.

> Actually, it's slightly better than $O(2^n)$. If you look at the subtree, you might notice that (excluding the leaf nodes and those immediately above it) the right subtree of any node is always smaller than the left subtree. If they were the same size, we'd have an $O(2^n)$ runtime. But since the right and left subtrees are not the same size, the true runtime is closer to $O(1.6^n)$. Saying $O(2^n)$ is still technically correct though as it describes an upper bound on the runtime (see "Big O, Big Theta, and Big Omega" on page 39). Either way, we still have an exponential runtime.

Indeed, if we implemented this on a computer, we'd see the number of seconds increase exponentially.



We should look for a way to optimize this.

Top-Down Dynamic Programming (or Memoization)

Study the recursion tree. Where do you see identical nodes?

There are lots of identical nodes. For example, `fib(3)` appears twice and fib(2) appears three times. Why should we recompute these from scratch each time?

In fact, when we call `fib(n)`, we shouldn't have to do much more than `O(n)` calls, since there's only `O(n)` possible values we can throw at `fib`. Each time we compute `fib(i)`, we should just cache this result and use it later.

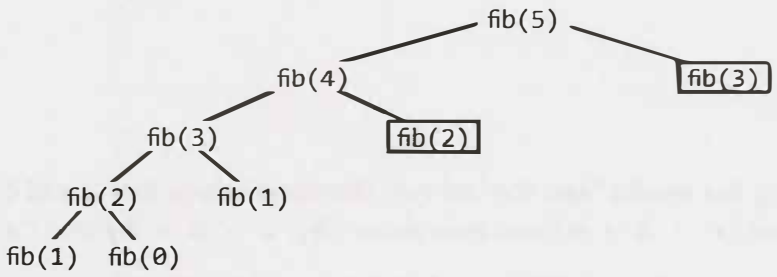
This is exactly what memoization is.

With just a small modification, we can tweak this function to run in `O(n)` time. We simply cache the results of `fibonacci(i)` between calls.

```
1  int fibonacci(int n) {
2    return fibonacci(n, new int[n + 1]);
3  }
4
5  int fibonacci(int i, int[] memo) {
6    if (i == 0 || i == 1) return i;
7
8    if (memo[i] == 0) {
9      memo[i] = fibonacci(i - 1, memo) + fibonacci(i - 2, memo);
10   }
11   return memo[i];
12 }
```

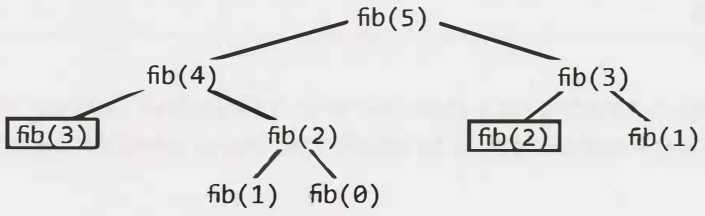
While the first recursive function may take over a minute to generate the 50th Fibonacci number on a typical computer, the dynamic programming method can generate the 10,000th Fibonacci number in just fractions of a millisecond. (Of course, with this exact code, the `int` would have overflowed very early on.)

Now, if we draw the recursion tree, it looks something like this (the black boxes represent cached calls that returned immediately):



How many nodes are in this tree now? We might notice that the tree now just shoots straight down, to a depth of roughly n. Each node of those nodes has one other child, resulting in roughly 2n children in the tree. This gives us a runtime of `O(n)`.

Often it can be useful to picture the recursion tree as something like this:



This is *not* actually how the recursion occurred. However, by expanding the further up nodes rather than the

lower nodes, you have a tree that grows wide before it grows deep. (It's like doing this breadth-first rather than depth-first.) Sometimes this makes it easier to compute the number of nodes in the tree. All you're really doing is changing which nodes you expand and which ones return cached values. Try this if you're stuck on computing the runtime of a dynamic programming problem.

Bottom-Up Dynamic Programming

We can also take this approach and implement it with bottom-up dynamic programming. Think about doing the same things as the recursive memoized approach, but in reverse.

First, we compute `fib(1)` and `fib(0)`, which are already known from the base cases. Then we use those to compute `fib(2)`. Then we use the prior answers to compute `fib(3)`, then `fib(4)`, and so on.

```
1  int fibonacci(int n) {
2    if (n == 0) return 0;
3    else if (n == 1) return 1;
4
5    int[] memo = new int[n];
6    memo[0] = 0;
7    memo[1] = 1;
8    for (int i = 2; i < n; i++) {
9      memo[i] = memo[i - 1] + memo[i - 2];
10   }
11   return memo[n - 1] + memo[n - 2];
12 }
```

If you really think about how this works, you only use `memo[i]` for `memo[i+1]` and `memo[i+2]`. You don't need it after that. Therefore, we can get rid of the memo table and just store a few variables.

```
1  int fibonacci(int n) {
2    if (n == 0) return 0;
3    int a = 0;
4    int b = 1;
5    for (int i = 2; i < n; i++) {
6      int c = a + b;
7      a = b;
8      b = c;
9    }
10   return a + b;
11 }
```

This is basically storing the results from the last two Fibonacci values into a and b. At each iteration, we compute the next value (`c = a + b`) and then move (`b, c = a + b`) into (`a, b`).

This explanation might seem like overkill for such a simple problem, but truly understanding this process will make more difficult problems much easier. Going through the problems in this chapter, many of which use dynamic programming, will help solidify your understanding.

Additional Reading: Proof by Induction (pg 631).

Interview Questions

8.1 **Triple Step:** A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

Hints: #152, #178, #217, #237, #262, #359

pg 342

8.2 **Robot in a Grid:** Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are "off limits" such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

Hints: #331, #360, #388

pg 344

8.3 **Magic Index:** A magic index in an array `A[0...n-1]` is defined to be an index such that `A[i] = i`. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.

FOLLOW UP

What if the values are not distinct?

Hints: #170, #204, #240, #286, #340

pg 346

8.4 **Power Set:** Write a method to return all subsets of a set.

Hints: #273, #290, #338, #354, #373

pg 348

8.5 **Recursive Multiply:** Write a recursive function to multiply two positive integers without using the * operator. You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

Hints: #166, #203, #227, #234, #246, #280

pg 350

8.6 **Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

(1) Only one disk can be moved at a time.

(2) A disk is slid off the top of one tower onto another tower.

(3) A disk cannot be placed on top of a smaller disk.

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

Hints: #144, #224, #250, #272, #318

pg 353

8.7 **Permutations without Dups:** Write a method to compute all permutations of a string of unique characters.

Hints: #150, #185, #200, #267, #278, #309, #335, #356

pg 355

8.8 **Permutations with Dups:** Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

Hints: #161, #190, #222, #255

pg 357

8.9 **Parens:** Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n pairs of parentheses.

EXAMPLE

Input: 3

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

Hints: #138, #174, #187, #209, #243, #265, #295

pg 359

8.10 **Paint Fill:** Implement the "paint fill" function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

Hints: #364, #382

pg 361

8.11 **Coins:** Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

Hints: #300, #324, #343, #380, #394

pg 362

8.12 **Eight Queens:** Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

Hints: #308, #350, #371

pg 364

8.13 **Stack of Boxes:** You have a stack of n boxes, with widths w_i, heights h_i, and depths d_i. The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

Hints: #155, #194, #214, #260, #322, #368, #378

pg 366

8.14 **Boolean Evaluation:** Given a boolean expression consisting of the symbols `0` (false), `1` (true), `&` (AND), `|` (OR), and `^` (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`.

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

Hints: #148, #168, #197, #305, #327

pg 368

Additional Questions: Linked Lists (#2.2, #2.5, #2.6), Stacks and Queues (#3.3), Trees and Graphs (#4.2, #4.3, #4.4, #4.5, #4.8, #4.10, #4.11, #4.12), Math and Logic Puzzles (#6.6), Sorting and Searching (#10.5, #10.9, #10.10), C++ (#12.8), Moderate Problems (#16.11), Hard Problems (#17.4, #17.6, #17.8, #17.12, #17.13, #17.15, #17.16, #17.24, #17.25).

Hints start on page 662.

9

System Design and Scalability

Despite how intimidating they seem, scalability questions can be among the easiest questions. There are no "gotchas," no tricks, and no fancy algorithms—at least not usually. What trips up many people is that they believe there's something "magic" to these problems—some hidden bit of knowledge.

It's not like that. These questions are simply designed to see how you would perform in the real world. If you were asked by your manager to design some system, what would you do?

That's why you should approach it just like this. Tackle the problem by doing it just like you would at work. Ask questions. Engage the interviewer. Discuss the tradeoffs.

We will touch on some key concepts in this chapter, but recognize it's not really about memorizing these concepts. Yes, understanding some big components of system design can be useful, but it's much more about the process you take. There are good solutions and bad solutions. There is no perfect solution.

Handling the Questions

- **Communicate**: A key goal of system design questions is to evaluate your ability to communicate. Stay engaged with the interviewer. Ask them questions. Be open about the issues of your system.
- **Go broad first**: Don't dive straight into the algorithm part or get excessively focused on one part.
- **Use the whiteboard**: Using a whiteboard helps your interviewer follow your proposed design. Get up to the whiteboard in the very beginning and use it to draw a picture of what you're proposing.
- **Acknowledge interviewer concerns**: Your interviewer will likely jump in with concerns. Don't brush them off; validate them. Acknowledge the issues your interviewer points out and make changes accordingly.
- **Be careful about assumptions**: An incorrect assumption can dramatically change the problem. For example, if your system produces analytics / statistics for a dataset, it matters whether those analytics must be totally up to date.
- **State your assumptions explicitly**: When you do make assumptions, state them. This allows your interviewer to correct you if you're mistaken, and shows that you at least know what assumptions you're making.
- **Estimate when necessary**: In many cases, you might not have the data you need. For example, if you're designing a web crawler, you might need to estimate how much space it will take to store all the URLs. You can estimate this with other data you know.
- **Drive**: As the candidate, you should stay in the driver's seat. This doesn't mean you don't talk to your interviewer; in fact, you *must* talk to your interviewer. However, you should be driving through the ques-

tion. Ask questions. Be open about tradeoffs. Continue to go deeper. Continue to make improvements.

These questions are largely about the process rather than the ultimate design.

Design: Step-By-Step

If your manager asked you to design a system such as TinyURL, you probably wouldn't just say, "Okay", then lock yourself in your office to design it by yourself. You would probably have a lot more questions before you do it. This is the way you should handle it in an interview.

Step 1: Scope the Problem

You can't design a system if you don't know what you're designing. Scoping the problem is important because you want to ensure that you're building what the interviewer wants and because this might be something that interviewer is specifically evaluating.

If you're asked something such as "Design TinyURL", you'll want to understand what exactly you need to implement. Will people be able to specify their own short URLs? Or will it all be auto-generated? Will you need to keep track of any stats on the clicks? Should the URLs stay alive forever, or do they have a timeout?

These are questions that must be answered before going further.

Make a list here as well of the major features or use cases. For example, for TinyURL, it might be:

- Shortening a URL to a TinyURL.
- Analytics for a URL.
- Retrieving the URL associated with a TinyURL.
- User accounts and link management.

Step 2: Make Reasonable Assumptions

It's okay to make some assumptions (when necessary), but they should be reasonable. For example, it would not be reasonable to assume that your system only needs to process 100 users per day, or to assume that you have infinite memory available.

However, it might be reasonable to design for a max of one million new URLs per day. Making this assumption can help you calculate how much data your system might need to store.

Some assumptions might take some "product sense" (which is not a bad thing). For example, is it okay for the data to be stale by a max of ten minutes? That all depends. If it takes 10 minutes for a just-entered URL to work, that's a deal-breaking issue. People usually want these URLs to be active immediately. However, if the statistics are ten minutes out of date, that might be okay. Talk to your interviewer about these sorts of assumptions.

Step 3: Draw the Major Components

Get up out of that chair and go to the whiteboard. Draw a diagram of the major components. You might have something like a frontend server (or set of servers) that pull data from the backend's data store. You might have another set of servers that crawl the internet for some data, and another set that process analytics. Draw a picture of what this system might look like.

Walk through your system from end-to-end to provide a flow. A user enters a new URL. Then what?

It may help here to ignore major scalability challenges and just pretend that the simple, obvious approaches will be okay. You'll handle the big issues in Step 4.

Step 4: Identify the Key Issues

Once you have a basic design in mind, focus on the key issues. What will be the bottlenecks or major challenges in the system?

For example, if you were designing TinyURL, one situation you might consider is that while some URLs will be infrequently accessed, others can suddenly peak. This might happen if a URL is posted on Reddit or another popular forum. You don't necessarily want to constantly hit the database.

Your interviewer might provide some guidance here. If so, take this guidance and use it.

Step 5: Redesign for the Key Issues

Once you have identified the key issues, it's time to adjust your design for it. You might find that it involves a major redesign or just some minor tweaking (like using a cache).

Stay up at the whiteboard here and update your diagram as your design changes.

Be open about any limitations in your design. Your interviewer will likely be aware of them, so it's important to communicate that you're aware of them, too.

Algorithms that Scale: Step-By-Step

In some cases, you're not being asked to design an entire system. You're just being asked to design a single feature or algorithm, but you have to do it in a scalable way. Or, there might be one algorithm part that is the "real" focus of a broader design question.

In these cases, try the following approach.

Step 1: Ask Questions

As in the earlier approach, ask questions to make sure you really understand the question. There might be details the interviewer left out (intentionally or unintentionally). You can't solve a problem if you don't understand exactly what the problem is.

Step 2: Make Believe

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

Step 3: Get Real

Now go back to the original problem. How much data can you fit on one machine, and what problems will occur when you split up the data? Common problems include figuring out how to logically divide the data up, and how one machine would identify where to look up a different piece of data.

Step 4: Solve Problems

Finally, think about how to solve the issues you identified in Step 2. Remember that the solution for each issue might be to actually remove the issue entirely, or it might be to simply mitigate the issue. Usually, you

can continue using (with modifications) the approach you outlined in Step 1, but occasionally you will need to fundamentally alter the approach.

Note that an iterative approach is typically useful. That is, once you have solved the problems from Step 3, new problems may have emerged, and you must tackle those as well.

Your goal is not to re-architect a complex system that companies have spent millions of dollars building, but rather to demonstrate that you can analyze and solve problems. Poking holes in your own solution is a fantastic way to demonstrate this.

Key Concepts

While system design questions aren't really tests of what you know, certain concepts can make things a lot easier. We will give a brief overview here. All of these are deep, complex topics, so we encourage you to use online resources for more research.

Horizontal vs. Vertical Scaling

A system can be scaled one of two ways.

- Vertical scaling means increasing the resources of a specific node. For example, you might add additional memory to a server to improve its ability to handle load changes.
- Horizontal scaling means increasing the number of nodes. For example, you might add additional servers, thus decreasing the load on any one server.

Vertical scaling is generally easier than horizontal scaling, but it's limited. You can only add so much memory or disk space.

Load Balancer

Typically, some frontend parts of a scalable website will be thrown behind a load balancer. This allows a system to distribute the load evenly so that one server doesn't crash and take down the whole system. To do so, of course, you have to build out a network of cloned servers that all have essentially the same code and access to the same data.

Database Denormalization and NoSQL

Joins in a relational database such as SQL can get very slow as the system grows bigger. For this reason, you would generally avoid them.

Denormalization is one part of this. Denormalization means adding redundant information into a database to speed up reads. For example, imagine a database describing projects and tasks (where a project can have multiple tasks). You might need to get the project name and the task information. Rather than doing a join across these tables, you can store the project name within the task table (in addition to the project table).

Or, you can go with a NoSQL database. A NoSQL database does not support joins and might structure data in a different way. It is designed to scale better.

Database Partitioning (Sharding)

Sharding means splitting the data across multiple machines while ensuring you have a way of figuring out which data is on which machine.

A few common ways of partitioning include:

- **Vertical Partitioning**: This is basically partitioning by feature. For example, if you were building a social network, you might have one partition for tables relating to profiles, another one for messages, and so on. One drawback of this is that if one of these tables gets very large, you might need to repartition that database (possibly using a different partitioning scheme).
- **Key-Based (or Hash-Based) Partitioning**: This uses some part of the data (for example an ID) to partition it. A very simple way to do this is to allocate N servers and put the data on `mod(key, n)`. One issue with this is that the number of servers you have is effectively fixed. Adding additional servers means reallocating all the data—a very expensive task.
- **Directory-Based Partitioning**: In this scheme, you maintain a lookup table for where the data can be found. This makes it relatively easy to add additional servers, but it comes with two major drawbacks. First, the lookup table can be a single point of failure. Second, constantly accessing this table impacts performance.

Many architectures actually end up using multiple partitioning schemes.

Caching

An in-memory cache can deliver very rapid results. It is a simple key-value pairing and typically sits between your application layer and your data store.

When an application requests a piece of information, it first tries the cache. If the cache does not contain the key, it will then look up the data in the data store. (At this point, the data might—or might not—be stored in the data store.)

When you cache, you might cache a query and its results directly. Or, alternatively, you can cache the specific object (for example, a rendered version of a part of the website, or a list of the most recent blog posts).

Asynchronous Processing & Queues

Slow operations should ideally be done asynchronously. Otherwise, a user might get stuck waiting and waiting for a process to complete.

In some cases, we can do this in advance (i.e., we can pre-process). For example, we might have a queue of jobs to be done that update some part of the website. If we were running a forum, one of these jobs might be to re-render a page that lists the most popular posts and the number of comments. That list might end up being slightly out of date, but that's perhaps okay. It's better than a user stuck waiting on the website to load simply because someone added a new comment and invalidated the cached version of this page.

In other cases, we might tell the user to wait and notify them when the process is done. You've probably seen this on websites before. Perhaps you enabled some new part of a website and it says it needs a few minutes to import your data, but you'll get a notification when it's done.

Networking Metrics

Some of the most important metrics around networking include:

- **Bandwidth**: This is the maximum amount of data that can be transferred in a unit of time. It is typically expressed in bits per second (or some similar ways, such as gigabytes per second).
- **Throughput**: Whereas bandwidth is the maximum data that can be transferred in a unit of time, throughput is the actual amount of data that is transferred.
- **Latency**: This is how long it takes data to go from one end to the other. That is, it is the delay between the sender sending information (even a very small chunk of data) and the receiver receiving it.

Imagine you have a conveyor belt that transfers items across a factory. Latency is the time it takes an item to go from one side to another. Throughput is the number of items that roll off the conveyor belt per second.

- Building a fatter conveyor belt will not change latency. It will, however, change throughput and bandwidth. You can get more items on the belt, thus transferring more in a given unit of time.
- Shortening the belt will decrease latency, since items spend less time in transit. It won't change the throughput or bandwidth. The same number of items will roll off the belt per unit of time.
- Making a faster conveyor belt will change all three. The time it takes an item to travel across the factory decreases. More items will also roll off the conveyor belt per unit of time.
- Bandwidth is the number of items that can be transferred per unit of time, in the best possible conditions. Throughput is the time it really takes, when the machines perhaps aren't operating smoothly.

Latency can be easy to disregard, but it can be very important in particular situations. For example, if you're playing certain online games, latency can be a very big deal. How can you play a typical online sports game (like a two-player football game) if you aren't notified very quickly of your opponent's movement? Additionally, unlike throughput where at least you have the option of speeding things up through data compression, there is often little you can do about latency.

MapReduce

MapReduce is often associated with Google, but it's used much more broadly than that. A MapReduce program is typically used to process large amounts of data.

As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.

- `Map` takes in some data and emits a `<key, value>` pair.
- `Reduce` takes a key and a set of associated values and "reduces" them in some way, emitting a new key and value. The results of this might be fed back into the `Reduce` program for more reducing.

MapReduce allows us to do a lot of processing in parallel, which makes processing huge amounts of data more scalable.

For more information, see "MapReduce" on page 642.

Considerations

In addition to the earlier concepts to learn, you should consider the following issues when designing a system.

- **Failures**: Essentially any part of a system can fail. You'll need to plan for many or all of these failures.
- **Availability and Reliability**: Availability is a function of the percentage of time the system is operational. Reliability is a function of the probability that the system is operational for a certain unit of time.
- **Read-heavy vs. Write-heavy**: Whether an application will do a lot of reads or a lot of writes impacts the design. If it's write-heavy, you could consider queuing up the writes (but think about potential failure here!). If it's read-heavy, you might want to cache. Other design decisions could change as well.
- **Security**: Security threats can, of course, be devastating for a system. Think about the types of issues a system might face and design around those.

This is just to get you started with the potential issues for a system. Remember to be open in your interview about the tradeoffs.

There is no "perfect" system.

There is no single design for TinyURL or Google Maps or any other system that works perfectly (although there are a great number that would work terribly). There are always tradeoffs. Two people could have substantially different designs for a system, with both being excellent given different assumptions.

Your goal in these problems is to be able to understand use cases, scope a problem, make reasonable assumptions, create a solid design based on those assumptions, and be open about the weaknesses of your design. Do not expect something perfect.

Example Problem

Given a list of millions of documents, how would you find all documents that contain a list of words? The words can appear in any order, but they must be complete words. That is, "book" does not match "bookkeeper."

Before we start solving the problem, we need to understand whether this is a one time only operation, or if this `findWords` procedure will be called repeatedly. Let's assume that we will be calling `findWords` many times for the same set of documents, and, therefore, we can accept the burden of pre-processing.

Step 1

The first step is to pretend we just have a few dozen documents. How would we implement `findWords` in this case? (Tip: stop here and try to solve this yourself before reading on.)

One way to do this is to pre-process each document and create a hash table index. This hash table would map from a word to a list of the documents that contain that word.

```
"books" -> {doc2, doc3, doc6, doc8}
"many"  -> {doc1, doc3, doc7, doc8, doc9}
```

To search for "`many books`," we would simply do an intersection on the values for "`books`" and "`many`", and return `{doc3, doc8}` as the result.

Step 2

Now go back to the original problem. What problems are introduced with millions of documents? For starters, we probably need to divide up the documents across many machines. Also, depending on a variety of factors, such as the number of possible words and the repetition of words in a document, we may not be able to fit the full hash table on one machine. Let's assume that this is the case.

This division introduces the following key concerns:

1. How will we divide up our hash table? We could divide it up by keyword, such that a given machine contains the full document list for a given word. Or, we could divide by document, such that a machine contains the keyword mapping for only a subset of the documents.
2. Once we decide how to divide up the data, we may need to process a document on one machine and push the results off to other machines. What does this process look like? (Note: if we divide the hash table by document, this step may not be necessary.)
3. We will need a way of knowing which machine holds a piece of data. What does this lookup table look like, and where is it stored?

These are just three concerns. There may be many others.

Step 3

In Step 3, we find solutions to each of these issues. One solution is to divide up the words alphabetically by keyword, such that each machine controls a range of words (e.g., "after" through "apple").

We can implement a simple algorithm in which we iterate through the keywords alphabetically, storing as much data as possible on one machine. When that machine is full, we can move to the next machine.

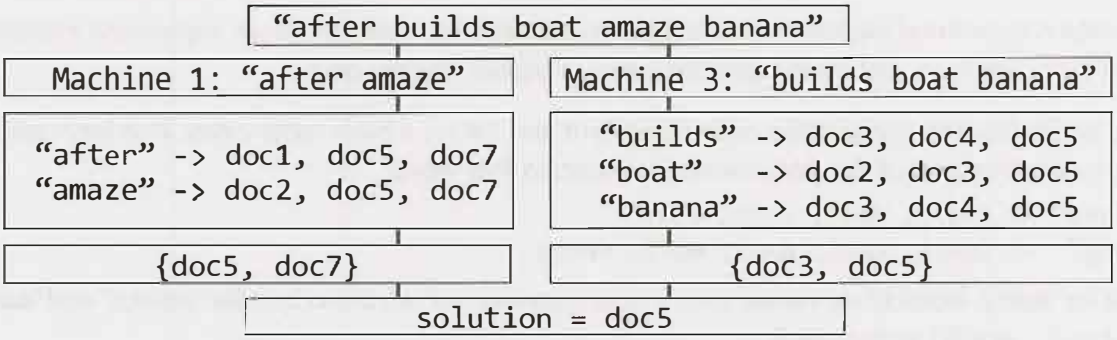
The advantage of this approach is that the lookup table is small and simple (since it must only specify a range of values), and each machine can store a copy of the lookup table. However, the disadvantage is that if new documents or words are added, we may need to perform an expensive shift of keywords.

To find all the documents that match a list of strings, we would first sort the list and then send each machine a lookup request for the strings that the machine owns. For example, if our string is "after builds boat amaze banana", machine 1 would get a lookup request for {"after", "amaze"}.

Machine 1 looks up the documents containing "after" and "amaze," and performs an intersection on these document lists. Machine 3 does the same for {"banana", "boat", "builds"}, and intersects their lists.

In the final step, the initial machine would do an intersection on the results from Machine 1 and Machine 3.

The following diagram explains this process.



Interview Questions

These questions are designed to mirror a real interview, so they will not always be well defined. Think about what questions you would ask your interviewer and then make reasonable assumptions. You may make different assumptions than us, and that will lead you to a very different design. That's okay!

9.1 **Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service that provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

Hints: #385, #396

pg 372

9.2 **Social Network:** How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

Hints: #270, #285, #304, #321

pg 374

9.3 **Web Crawler:** If you were designing a web crawler, how would you avoid getting into infinite loops?

Hints: #334, #353, #365

pg 378

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

Hints: #326, #347

pg 380

9.5 **Cache:** Imagine a web server for a simplified search engine. This system has 100 machines to respond to search queries, which may then call out using `processSearch(string query)` to another cluster of machines to actually get the result. The machine which responds to a given query is chosen at random, so you cannot guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism for the most recent queries. Be sure to explain how you would update the cache when data changes.

Hints: #259, #274, #293, #311

pg 381

9.6 **Sales Rank**: A large eCommerce company wishes to list the best-selling products, overall and by category. For example, one product might be the #1056th best-selling product overall but the #13th best-selling product under "Sports Equipment" and the #24th best-selling product under "Safety." Describe how you would design this system.

Hints: #142, #158, #176, #189, #208, #223, #236, #244

pg 385

9.7 **Personal Financial Manager**: Explain how you would design a personal financial manager (like Mint.com). This system would connect to your bank accounts, analyze your spending habits, and make recommendations.

Hints: #162, #180, #199, #212, #247, #276

pg 388

9.8 **Pastebin**: Design a system like Pastebin, where a user can enter a piece of text and get a randomly generated URL to access it.

Hints: #165, #184, #206, #232

pg 392

Additional Questions: Object-Oriented Design (#7.7)

Hints start on page 662.

10

Sorting and Searching

Understanding the common sorting and searching algorithms is incredibly valuable, as many sorting and searching problems are tweaks of the well-known algorithms. A good approach is therefore to run through the different sorting algorithms and see if one applies particularly well.

For example, suppose you are asked the following question: Given a very large array of `Person` objects, sort the people in increasing order of age.

We're given two interesting bits of knowledge here:

1. It's a large array, so efficiency is very important.
2. We are sorting based on ages, so we know the values are in a small range.

By scanning through the various sorting algorithms, we might notice that bucket sort (or radix sort) would be a perfect candidate for this algorithm. In fact, we can make the buckets small (just 1 year each) and get $O(n)$ running time.

Common Sorting Algorithms

Learning (or re-learning) the common sorting algorithms is a great way to boost your performance. Of the five algorithms explained below, Merge Sort, Quick Sort and Bucket Sort are the most commonly used in interviews.

Bubble Sort | Runtime: $O(n^2)$ **average and worst case. Memory:** $O(1)$.

In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly "bubble" up to the beginning of the list.

Selection Sort | Runtime: $O(n^2)$ **average and worst case. Memory:** $O(1)$.

Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this until all the elements are in place.

Merge Sort | Runtime: $O(n \log(n))$ **average and worst case. Memory:** Depends.

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single-element arrays. It is the "merge" part that does all the heavy lifting.

The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be (`helperLeft` and `helperRight`). We then iterate through `helper`, copying the smaller element from each half into the array. At the end, we copy any remaining elements into the target array.

```
1  void mergesort(int[] array) {
2    int[] helper = new int[array.length];
3    mergesort(array, helper, 0, array.length - 1);
4  }
5
6  void mergesort(int[] array, int[] helper, int low, int high) {
7    if (low < high) {
8      int middle = (low + high) / 2;
9      mergesort(array, helper, low, middle); // Sort left half
10     mergesort(array, helper, middle+1, high); // Sort right half
11     merge(array, helper, low, middle, high); // Merge them
12   }
13 }
14
15 void merge(int[] array, int[] helper, int low, int middle, int high) {
16   /* Copy both halves into a helper array */
17   for (int i = low; i <= high; i++) {
18     helper[i] = array[i];
19   }
20
21   int helperLeft = low;
22   int helperRight = middle + 1;
23   int current = low;
24
25   /* Iterate through helper array. Compare the left and right half, copying back
26    * the smaller element from the two halves into the original array. */
27   while (helperLeft <= middle && helperRight <= high) {
28     if (helper[helperLeft] <= helper[helperRight]) {
29       array[current] = helper[helperLeft];
30       helperLeft++;
31     } else { // If right element is smaller than left element
32       array[current] = helper[helperRight];
33       helperRight++;
34     }
35     current++;
36   }
37
38   /* Copy the rest of the left side of the array into the target array */
39   int remaining = middle - helperLeft;
40   for (int i = 0; i <= remaining; i++) {
41     array[current + i] = helper[helperLeft + i];
42   }
43 }
```

You may notice that only the remaining elements from the left half of the helper array are copied into the target array. Why not the right half? The right half doesn't need to be copied because it's *already* there.

Consider, for example, an array like `[1, 4, 5 || 2, 8, 9]` (the "`||`" indicates the partition point). Prior to merging the two halves, both the helper array and the target array segment will end with `[8, 9]`. Once we copy over four elements (`1, 4, 5,` and 2) into the target array, the `[8, 9]` will still be in place in both arrays. There's no need to copy them over.

The space complexity of merge sort is O(n) due to the auxiliary space used to merge parts of the array.

Quick Sort | Runtime: $O(n \log(n))$ **average,** $O(n^2)$ **worst case. Memory:** $O(\log(n))$.

In quick sort, we pick a random element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps (see below).

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow. This is the reason for the $O(n^2)$ worst case runtime.

```
1  void quickSort(int[] arr, int left, int right) {
2    int index = partition(arr, left, right);
3    if (left < index - 1) { // Sort left half
4      quickSort(arr, left, index - 1);
5    }
6    if (index < right) { // Sort right half
7      quickSort(arr, index, right);
8    }
9  }
10
11 int partition(int[] arr, int left, int right) {
12   int pivot = arr[(left + right) / 2]; // Pick pivot point
13   while (left <= right) {
14     // Find element on left that should be on right
15     while (arr[left] < pivot) left++;
16
17     // Find element on right that should be on left
18     while (arr[right] > pivot) right--;
19
20     // Swap elements, and move left and right indices
21     if (left <= right) {
22       swap(arr, left, right); // swaps elements
23       left++;
24       right--;
25     }
26   }
27   return left;
28 }
```

Radix Sort | Runtime: $O(kn)$ **(see below)**

Radix sort is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In radix sort, we iterate through each digit of the number, grouping numbers by each digit. For example, if we have an array of integers, we might first sort by the first digit, so that the 0s are grouped together. Then, we sort each of these groupings by the next digit. We repeat this process sorting by each subsequent digit, until finally the whole array is sorted.

Unlike comparison sorting algorithms, which cannot perform better than $O(n \log(n))$ in the average case, radix sort has a runtime of $O(kn)$, where n is the number of elements and k is the number of passes of the sorting algorithm.

Searching Algorithms

When we think of searching algorithms, we generally think of binary search. Indeed, this is a very useful algorithm to study.

In binary search, we look for an element x in a sorted array by first comparing x to the midpoint of the array. If x is less than the midpoint, then we search the left half of the array. If x is greater than the midpoint, then we search the right half of the array. We then repeat this process, treating the left and right halves as subarrays. Again, we compare x to the midpoint of this subarray and then search either its left or right side. We repeat this process until we either find x or the subarray has size 0.

Note that although the concept is fairly simple, getting all the details right is far more difficult than you might think. As you study the code below, pay attention to the plus ones and minus ones.

```
1  int binarySearch(int[] a, int x) {
2     int low = 0;
3     int high = a.length - 1;
4     int mid;
5
6     while (low <= high) {
7        mid = (low + high) / 2;
8        if (a[mid] < x) {
9           low = mid + 1;
10       } else if (a[mid] > x) {
11          high = mid - 1;
12       } else {
13          return mid;
14       }
15    }
16    return -1; // Error
17 }
18
19 int binarySearchRecursive(int[] a, int x, int low, int high) {
20    if (low > high) return -1; // Error
21
22    int mid = (low + high) / 2;
23    if (a[mid] < x) {
24       return binarySearchRecursive(a, x, mid + 1, high);
25    } else if (a[mid] > x) {
26       return binarySearchRecursive(a, x, low, mid - 1);
27    } else {
28       return mid;
29    }
30 }
```

Potential ways to search a data structure extend beyond binary search, and you would do best not to limit yourself to just this option. You might, for example, search for a node by leveraging a binary tree, or by using a hash table. Think beyond binary search!

Interview Questions

10.1 **Sorted Merge:** You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

Hints: #332

pg 396

10.2 **Group Anagrams:** Write a method to sort an array of strings so that all the anagrams are next to each other.

Hints: #177, #182, #263, #342

pg 397

10.3 ***Search in Rotated Array:*** *Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.*

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

Hints: #298, #310

pg 398

10.4 **Sorted Search, No Size:** You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

Hints: #320, #337, #348

pg 400

10.5 **Sparse Search:** Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: `ball`, {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Output: 4

Hints: #256

pg 401

10.6 **Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

Hints: #207

pg 402

10.7 **Missing Int:** Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

Hints: #235, #254, #281

pg 403

10.8 Find Duplicates: You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

Hints: #289, #315

pg 406

10.9 Sorted Matrix Search: Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Hints: #193, #211, #229, #251, #266, #279, #288, #291, #303, #317, #330

pg 407

10.10 Rank from Stream: Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

`getRankOfNumber(1) = 0`

`getRankOfNumber(3) = 1`

`getRankOfNumber(4) = 3`

Hints: #301, #376, #392

pg 412

10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

Hints: #196, #219, #231, #253, #277, #292, #316

pg 414

Additional Questions: Arrays and Strings (#1.2), Recursion (#8.3), Moderate (#16.10, #16.16, #16.21, #16.24), Hard (#17.11, #17.26).

Hints start on page 662.

16

Moderate

16.1 **Number Swapper:** Write a function to swap a number in place (that is, without temporary variables).

Hints: #492, #716, #737

pg 462

16.2 **Word Frequencies:** Design a method to find the frequency of occurrences of any given word in a book. What if we were running this algorithm multiple times?

Hints: #489, #536

pg 463

16.3 **Intersection:** Given two straight line segments (represented as a start point and an end point), compute the point of intersection, if any.

Hints: #465, #472, #497, #517, #527

pg 464

16.4 **Tic Tac Win:** Design an algorithm to figure out if someone has won a game of tic-tac-toe.

Hints: #710, #732

pg 466

16.5 **Factorial Zeros:** Write an algorithm which computes the number of trailing zeros in n factorial.

Hints: #585, #711, #729, #733, #745

pg 473

16.6 **Smallest Difference:** Given two arrays of integers, compute the pair of values (one value in each array) with the smallest (non-negative) difference. Return the difference.

EXAMPLE

Input: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Output: 3. That is, the pair (11, 8).

Hints: #632, #670, #679

pg 474

16.7 **Number Max:** Write a method that finds the maximum of two numbers. You should not use if-else or any other comparison operator.

Hints: #473, #513, #707, #728

pg 475

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

Hints: #502, #588, #688

pg 477

16.9 **Operations:** Write methods to implement the multiply, subtract, and divide operations for integers. The results of all of these are integers. Use only the add operator.

Hints: #572, #600, #613, #648

pg 478

16.10 **Living People:** Given a list of people with their birth and death years, implement a method to compute the year with the most number of people alive. You may assume that all people were born between 1900 and 2000 (inclusive). If a person was alive during any portion of that year, they should be included in that year's count. For example, Person (birth = 1908, death = 1909) is included in the counts for both 1908 and 1909.

Hints: #476, #490, #507, #514, #523, #532, #541, #549, #576

pg 482

16.11 **Diving Board**: You are building a diving board by placing a bunch of planks of wood end-to-end. There are two types of planks, one of length `shorter` and one of length `longer`. You must use exactly K planks of wood. Write a method to generate all possible lengths for the diving board.

Hints: #690, #700, #715, #722, #740, #747

pg 486

16.12 **XML Encoding:** Since XML is very verbose, you are given a way of encoding it where each tag gets mapped to a pre-defined integer value. The language/grammar is as follows:

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> some predefined mapping to int
Value     --> string value
```

For example, the following XML might be converted into the compressed string below (assuming a mapping of `family -> 1, person ->2, firstName -> 3, lastName -> 4, state -> 5`).

```
<family lastName="McDowell" state="CA">
  <person firstName="Gayle">Some Message</person>
</family>
```

Becomes:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

Write code to print the encoded version of an XML element (passed in `Element` and `Attribute` objects).

Hints: #466

pg 489

16.13 **Bisect Squares:** Given two squares on a two-dimensional plane, find a line that would cut these two squares in half. Assume that the top and the bottom sides of the square run parallel to the x-axis.

Hints: #468, #479, #528, #560

pg 490

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

Hints: #491, #520, #529, #563

pg 492

16.15 Master Mind: The Game of Master Mind is played as follows:

The computer has four slots, and each slot will contain a ball that is red (R), yellow (Y), green (G) or blue (B). For example, the computer might have RGGB (Slot #1 is red, Slots #2 and #3 are green, Slot #4 is blue).

You, the user, are trying to guess the solution. You might, for example, guess YRGB.

When you guess the correct color for the correct slot, you get a "hit." If you guess a color that exists but is in the wrong slot, you get a "pseudo-hit." Note that a slot that is a hit can never count as a pseudo-hit.

For example, if the actual solution is RGBY and you guess GGRR, you have one hit and one pseudo-hit.

Write a method that, given a guess and a solution, returns the number of hits and pseudo-hits.

Hints: #639, #730

pg 494

16.16 Sub Sort: Given an array of integers, write a method to find indices `m` and `n` such that if you sorted elements `m` through `n`, the entire array would be sorted. Minimize `n - m` (that is, find the smallest such sequence).

EXAMPLE

Input: `1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19`

Output: `(3, 9)`

Hints: #482, #553, #667, #708, #735, #746

pg 496

16.17 Contiguous Sequence: You are given an array of integers (both positive and negative). Find the contiguous sequence with the largest sum. Return the sum.

EXAMPLE

Input: `2, -8, 3, -2, 4, -10`

Output: `5 (i.e., {3, -2, 4})`

Hints: #531, #551, #567, #594, #614

pg 498

16.18 Pattern Matching: You are given two strings, `pattern` and `value`. The `pattern` string consists of just the letters a and b, describing a pattern within a string. For example, the string `catcatgocatgo` matches the pattern `aabab` (where `cat` is `a` and `go` is `b`). It also matches patterns like `a`, `ab`, and `b`. Write a method to determine if `value` matches `pattern`.

Hints: #631, #643, #653, #663, #685, #718, #727

pg 499

16.19 Pond Sizes: You have an integer matrix representing a plot of land, where the value at that location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of the pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

EXAMPLE

Input:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Output: 2, 4, 1 (in any order)

Hints: #674, #687, #706, #723

pg 503

16.20 T9: On old cell phones, users typed on a numeric keypad and the phone would provide a list of words that matched these numbers. Each digit mapped to a set of 0 - 4 letters. Implement an algorithm to return a list of matching words, given a sequence of digits. You are provided a list of valid words (provided in whatever data structure you'd like). The mapping is shown in the diagram below:

1	**2** abc	**3** def
4 ghi	**5** jkl	**6** mno
7 pqrs	**8** tuv	**9** wxyz
	0	

EXAMPLE

Input: `8733`

Output: `tree, used`

Hints: #471, #487, #654, #703, #726, #744

pg 505

16.21 Sum Swap: Given two arrays of integers, find a pair of values (one value from each array) that you can swap to give the two arrays the same sum.

EXAMPLE

Input: {4, 1, 2, 1, 1, 2} and {3, 6, 3, 3}

Output: {1, 3}

Hints: #545, #557, #564, #571, #583, #592, #602, #606, #635

pg 509

16.22 Langton's Ant: An ant is sitting on an infinite grid of white and black squares. It initially faces right. At each step, it does the following:

(1) At a white square, flip the color of the square, turn 90 degrees right (clockwise), and move forward one unit.

(2) At a black square, flip the color of the square, turn 90 degrees left (counter-clockwise), and move forward one unit.

Write a program to simulate the first K moves that the ant makes and print the final board as a grid. Note that you are not provided with the data structure to represent the grid. This is something you must design yourself. The only input to your method is K. You should print the final grid and return nothing. The method signature might be something like `void printKMoves(int K)`.

Hints: #474, #481, #533, #540, #559, #570, #599, #616, #627

pg 512

16.23 Rand7 from Rand5: Implement a method `rand7()` given `rand5()`. That is, given a method that generates a random number between 0 and 4 (inclusive), write a method that generates a random number between 0 and 6 (inclusive).

Hints: #505, #574, #637, #668, #697, #720

pg 518

16.24 Pairs with Sum: Design an algorithm to find all pairs of integers within an array which sum to a specified value.

Hints: #548, #597, #644, #673

pg 520

16.25 LRU Cache: Design and build a "least recently used" cache, which evicts the least recently used item. The cache should map from keys to values (allowing you to insert and retrieve a value associated with a particular key) and be initialized with a max size. When it is full, it should evict the least recently used item.

Hints: #524, #630, #694

pg 521

16.26 Calculator: Given an arithmetic equation consisting of positive integers, +, -, * and / (no parentheses), compute the result.

EXAMPLE

Input: `2*3+5/6*3+15`

Output: `23.5`

Hints: #521, #624, #665, #698

pg 524

17

Hard

17.1 **Add Without Plus:** Write a function that adds two numbers. You should not use + or any arithmetic operators.

Hints: #467, #544, #601, #628, #642, #664, #692, #712, #724

pg 530

17.2 **Shuffle:** Write a method to shuffle a deck of cards. It must be a perfect shuffle—in other words, each of the 52! permutations of the deck has to be equally likely. Assume that you are given a random number generator which is perfect.

Hints: #483, #579, #634

pg 531

17.3 **Random Set:** Write a method to randomly generate a set of `m` integers from an array of size n. Each element must have equal probability of being chosen.

Hints: #494, #596

pg 532

17.4 **Missing Number:** An array A contains all the integers from 0 to n, except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the `j`th bit of `A[i]`," which takes constant time. Write code to find the missing integer. Can you do it in `O(n)` time?

Hints: #610, #659, #683

pg 533

17.5 **Letters and Numbers:** Given an array filled with letters and numbers, find the longest subarray with an equal number of letters and numbers.

Hints: #485, #515, #619, #671, #713

pg 536

17.6 **Count of 2s:** Write a method to count the number of 2s that appear in all the numbers between 0 and n (inclusive).

EXAMPLE

Input: 25

Output: 9 (2, 12, 20, 21, 22, 23, 24 and 25. Note that 22 counts for two 2s.)

Hints: #573, #612, #641

pg 538

17.7 Baby Names: Each year, the government releases a list of the 10000 most common baby names and their frequencies (the number of babies with that name). The only problem with this is that some names have multiple spellings. For example, "John" and "Jon" are essentially the same name but would be listed separately in the list. Given two lists, one of names/frequencies and the other of pairs of equivalent names, write an algorithm to print a new list of the true frequency of each name. Note that if John and Jon are synonyms, and Jon and Johnny are synonyms, then John and Johnny are synonyms. (It is both transitive and symmetric.) In the final list, any name can be used as the "real" name.

EXAMPLE

Input:

Names: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Synonyms: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Output: John (27), Kris (36)

Hints: #478, #493, #512, #537, #586, #605, #655, #675, #704

pg 541

17.8 Circus Tower: A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

EXAMPLE

Input (ht, wt): `(65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)`

Output: The longest tower is length 6 and includes from top to bottom:

`(56, 90) (60,95) (65,100) (68,110) (70,150) (75,190)`

Hints: #638, #657, #666, #682, #699

pg 546

17.9 Kth Multiple: Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7. Note that 3, 5, and 7 do not have to be factors, but it should not have any other prime factors. For example, the first several multiples would be (in order) 1, 3, 5, 7, 9, 15, 21.

Hints: #488, #508, #550, #591, #622, #660, #686

pg 549

17.10 Majority Element: A majority element is an element that makes up more than half of the items in an array. Given a positive integers array, find the majority element. If there is no majority element, return -1. Do this in `O(N)` time and `O(1)` space.

EXAMPLE

Input: `1 2 5 9 5 9 5 5 5`

Output: 5

Hints: #522, #566, #604, #620, #650

pg 554

17.11 Word Distance: You have a large text file containing words. Given any two words, find the shortest distance (in terms of number of words) between them in the file. If the operation will be repeated many times for the same file (but different pairs of words), can you optimize your solution?

Hints: #486, #501, #538, #558, #633

pg 557

17.12 BiNode: Consider a simple data structure called `BiNode`, which has pointers to two other nodes.

```
public class BiNode {
   public BiNode node1, node2;
   public int data;
}
```

The data structure `BiNode` could be used to represent both a binary tree (where `node1` is the left node and `node2` is the right node) or a doubly linked list (where `node1` is the previous node and `node2` is the next node). Implement a method to convert a binary search tree (implemented with `BiNode`) into a doubly linked list. The values should be kept in order and the operation should be performed in place (that is, on the original data structure).

Hints: #509, #608, #646, #680, #701, #719

pg 560

17.13 Re-Space: Oh, no! You have accidentally removed all spaces, punctuation, and capitalization in a lengthy document. A sentence like "`I reset the computer. It still didn't boot!`" became "`iresetthecomputeritstilldidntboot`". You'll deal with the punctuation and capitalization later; right now you need to re-insert the spaces. Most of the words are in a dictionary but a few are not. Given a dictionary (a list of strings) and the document (a string), design an algorithm to unconcatenate the document in a way that minimizes the number of unrecognized characters.

EXAMPLE:

Input: `jesslookedjustliketimherbrother`

Output: `jess looked just like tim her brother` (7 unrecognized characters)

Hints: #496, #623, #656, #677, #739, #749

pg 563

17.14 Smallest K: Design an algorithm to find the smallest K numbers in an array.

Hints: #470, #530, #552, #593, #625, #647, #661, #678

pg 567

17.15 Longest Word: Given a list of words, write a program to find the longest word made of other words in the list.

EXAMPLE

Input: `cat, banana, dog, nana, walk, walker, dogwalker`

Output: `dogwalker`

Hints: #475, #499, #543, #589

pg 572

17.16 The Masseuse: A popular masseuse receives a sequence of back-to-back appointment requests and is debating which ones to accept. She needs a 15-minute break between appointments and therefore she cannot accept any adjacent requests. Given a sequence of back-to-back appointment requests (all multiples of 15 minutes, none overlap, and none can be moved), find the optimal (highest total booked minutes) set the masseuse can honor. Return the number of minutes.

EXAMPLE

Input: `{30, 15, 60, 75, 45, 15, 15, 45}`

Output: `180 minutes ({30, 60, 45, 45}).`

Hints: #495, #504, #516, #526, #542, #554, #562, #568, #578, #587, #607

pg 574

17.17 Multi Search: Given a string b and an array of smaller strings T, design a method to search b for each small string in T.

Hints: #480, #582, #617, #743

pg 578

17.18 *Shortest Supersequence:* You are given two arrays, one shorter (with all distinct elements) and one longer. Find the shortest subarray in the longer array that contains all the elements in the shorter array. The items can appear in any order.

EXAMPLE

Input: `{1, 5, 9} | {7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7}`

Output: `[7, 10]` (the underlined portion above)

Hints: #645, #652, #669, #681, #691, #725, #731, #741

pg 584

17.19 Missing Two: You are given an array with all the numbers from 1 to N appearing exactly once, except for one number that is missing. How can you find the missing number in `O(N)` time and `O(1)` space? What if there were two numbers missing?

Hints: #503, #590, #609, #626, #649, #672, #689, #696, #702, #717

pg 591

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

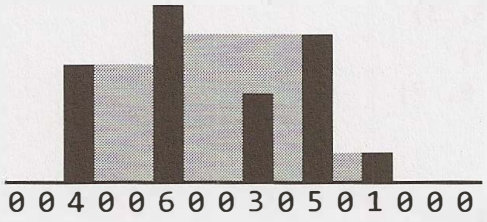
Hints: #519, #546, #575, #709

pg 595

17.21 Volume of Histogram: Imagine a histogram (bar graph). Design an algorithm to compute the volume of water it could hold if someone poured water across the top. You can assume that each histogram bar has width 1.

EXAMPLE (Black bars are the histogram. Gray is water.)

Input: `{0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 5, 0, 1, 0, 0, 0}`



Output: 26

Hints: #629, #640, #651, #658, #662, #676, #693, #734, #742

pg 596

17.22 Word Transformer: Given two words of equal length that are in a dictionary, write a method to transform one word into another word by changing only one letter at a time. The new word you get in each step must be in the dictionary.

EXAMPLE

Input: DAMP, LIKE

Output: DAMP -> LAMP -> LIMP -> LIME -> LIKE

Hints: #506, #535, #556, #580, #598, #618, #738

pg 602

17.23 Max Black Square: Imagine you have a square matrix, where each cell (pixel) is either black or white Design an algorithm to find the maximum subsquare such that all four borders are filled with black pixels.

Hints: #684, #695, #705, #714, #721, #736

pg 608

17.24 Max Submatrix: Given an NxN matrix of positive and negative integers, write code to find the submatrix with the largest possible sum.

Hints: #469, #511, #525, #539, #565, #581, #595, #615, #621

pg 611

17.25 Word Rectangle: Given a list of millions of words, design an algorithm to create the largest possible rectangle of letters such that every row forms a word (reading left to right) and every column forms a word (reading top to bottom). The words need not be chosen consecutively from the list, but all rows must be the same length and all columns must be the same height.

Hints: #477, #500, #748

pg 615

17.26 Sparse Similarity: The similarity of two documents (each with distinct words) is defined to be the size of the intersection divided by the size of the union. For example, if the documents consist of integers, the similarity of `{1, 5, 3}` and `{1, 7, 2, 3}` is `0.4`, because the intersection has size 2 and the union has size 5.

We have a long list of documents (with distinct values and each with an associated ID) where the similarity is believed to be "sparse." That is, any two arbitrarily selected documents are very likely to have similarity 0. Design an algorithm that returns a list of pairs of document IDs and the associated similarity.

Print only the pairs with similarity greater than 0. Empty documents should not be printed at all. For simplicity, you may assume each document is represented as an array of distinct integers.

EXAMPLE

Input:

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10}
```

Output:

```
ID1, ID2 : SIMILARITY
13, 19   : 0.1
13, 16   : 0.25
19, 24   : 0.14285714285714285
```

Hints: #484, #498, #510, #518, #534, #547, #555, #561, #569, #577, #584, #603, #611, #636

pg 620

Advanced Topics

XI

This section includes topics that are mostly beyond the scope of interviews but can come up on occasion. Interviewers shouldn't be surprised if you don't know these topics well. Feel free to dive into these topics if you want to. If you're pressed for time, they're low priority.

XI

Advanced Topics

When writing the 6th edition, I had a number of debates about what should and shouldn't be included. Red-black trees? Dijkstra's algorithm? Topological sort?

On one hand, I'd had a number of requests to include these topics. Some people insisted that these topics are asked "all the time" (in which case, they have a very different idea of what this phrase means!). There was clearly a desire—at least from some people—to include them. And learning more can't hurt, right?

On the other hand, I know these topics to be rarely asked. It happens, of course. Interviewers are individuals and might have their own ideas of what is "fair game" or "relevant" for an interview. But it's rare. When it does come up, if you don't know the topic, it's unlikely to be a big red flag.

> Admittedly, as an interviewer, I *have* asked candidates questions where the solution was essentially an application of one of these algorithms. On the rare occasions that a candidate already knew the algorithm, they did not benefit from this knowledge (nor were they hurt by it). I want to evaluate your ability to solve a problem you haven't seen before. So, I'll take into account whether you know the underlying algorithm in advance.

I believe in giving people a fair expectation of the interview, not scaring people into excess studying. I also have no interest in making the book more "advanced" so as to help book sales, at the expense of your time and energy. That's not fair or right to do to you.

(Additionally, I didn't want to give interviewers—who I know to be reading this—the impression that they can or should be covering these more advanced topics. Interviewers: If you ask about these topics, you're testing knowledge of algorithms. You're just going to wind up eliminating a lot of perfectly smart people.)

But there are many borderline "important" topics. They're not often asked, but sometimes they are.

Ultimately, I decided to leave the decision in your hands. After all, you know better than I do how thorough you want to be in your preparation. If you want to do an extra thorough job, read this. If you just love learning data structures and algorithms, read this. If you want to see new ways of approaching problems, read this.

But if you're pressed for time, this studying isn't a super high priority.

Useful Math

Here's some math that can be useful in some questions. There are more formal proofs that you can look up online, but we'll focus here on giving you the intuition behind them. You can think of these as informal proofs.

Sum of Integers 1 through N

What is $1 + 2 + ... + n$? Let's figure it out by pairing up low values with high values.

If n is even, we pair 1 with n, 2 with $n - 1$, and so on. We will have $\frac{n}{2}$ pairs each with sum $n + 1$.

If n is odd, we pair 0 with n, 1 with $n - 1$, and so on. We will have $\frac{n+1}{2}$ pairs with sum n.

n is even			
pair #	**a**	**b**	**a + b**
1	1	n	n + 1
2	2	n - 1	n + 1
3	3	n - 2	n + 1
4	4	n - 3	n + 1
...	...	...	...
$\frac{n}{2}$	$\frac{n}{2}$	$\frac{n}{2}+1$	n + 1
total:	$\frac{n}{2}*(n+1)$		

n is odd			
pair #	**a**	**b**	**a + b**
1	0	n	n
2	1	n - 1	n
3	2	n - 2	n
4	3	n - 3	n
...	...	...	...
$\frac{n+1}{2}$	$\frac{n-1}{2}$	$\frac{n+1}{2}$	n
total:	$\frac{n+1}{2}*n$		

In either case, the sum is $\frac{n(n+1)}{2}$.

This reasoning comes up a lot in nested loops. For example, consider the following code:

```
1  for (int i = 0; i < n; i++) {
2    for (int j = i + 1; j < n; j++) {
3      System.out.println(i + j);
4    }
5  }
```

On the first iteration of the outer for loop, the inner for loop iterates n - 1 times. On the second iteration of the outer for loop, the inner for loop iterates n-2 times. Next, $n - 3$, then $n - 4$, and so on. There are $\frac{n(n-1)}{2}$ total iterations of the inner for loop. Therefore, this code takes $O(n^2)$ time.

Sum of Powers of 2

Consider this sequence: $2^0 + 2^1 + 2^2 + ... + 2^n$. What is its result?

A nice way to see this is by looking at these values in binary.

	Power	Binary	Decimal
	2^0	00001	1
	2^1	00010	2
	2^2	00100	4
	2^3	01000	8
	2^4	10000	16
sum:	2^5-1	11111	32 - 1 = 31

Therefore, the sum of $2^0 + 2^1 + 2^2 + ... + 2^n$ would, in base 2, be a sequence of $(n + 1)$ 1s. This is $2^{n+1} - 1$.

Takeaway: The sum of a sequence of powers of two is roughly equal to the *next* value in the sequence.

Bases of Logs

Suppose we have something in $\log_2$ (log base 2). How do we convert that to $\log_{10}$? That is, what's the relationship between $\log_b k$ and $\log_x k$?

Let's do some math. Assume $c = \log_b k$ and $y = \log_x k$.

```
log_b k = c --> b^c = k          // This is the definition of log.
log_x(b^c) = log_x k             // Take log of both sides of b^c = k.
c log_x b = log_x k              // Rules of logs. You can move out the exponents.
c = log_b k = log_x k / log_x b  // Dividing above expression and substituting c.
```

Therefore, if we want to convert $\log_2 p$ to $\log_{10}$, we just do this:

$$\log_{10} p = \frac{\log_2 p}{\log_2 10}$$

Takeaway: Logs of different bases are only off by a constant factor. For this reason, we largely ignore what the base of a log within a big O expression. It doesn't matter since we drop constants anyway.

Permutations

How many ways are there of rearranging a string of n unique characters? Well, you have n options for what to put in the first characters, then n - 1 options for what to put in the second slot (one option is taken), then n - 2 options for what to put in the third slot, and so on. Therefore, the total number of strings is n!.

$$n! = \underline{n} * \underline{n - 1} * \underline{n - 2} * \underline{n - 3} * \ldots * \underline{1}$$

What if you were forming a k-length string (with all unique characters) from n total unique characters? You can follow similar logic, but you'd just stop your selection/multiplication earlier.

$$\frac{n!}{(n-k)!} = \underline{n} * \underline{n - 1} * \underline{n - 2} * \underline{n - 3} * \ldots * \underline{n - k + 1}$$

Combinations

Suppose you have a set of n distinct characters. How many ways are there of selecting k characters into a new set (where order doesn't matter)? That is, how many k-sized subsets are there out of n distinct elements? This is what the expression n-choose-k means, which is often written $\binom{n}{k}$.

Imagine we made a list of all the sets by first writing all k-length substrings and then taking out the duplicates.

From the above *Permutations* section, we'd have $n!/(n-k)!$ k-length substrings.

Since each k-sized subset can be rearranged k! unique ways into a string, each subset will be duplicated k! times in this list of substrings. Therefore, we need to divide by k! to take out these duplicates.

$$\binom{n}{k} = \frac{1}{k!} * \frac{n!}{(n-k)!} = \frac{n!}{k!(n-k)!}$$

Proof by Induction

Induction is a way of proving something to be true. It is closely related to recursion. It takes the following form.

Task: Prove statement P(k) is true for all k >= b.

- Base Case: Prove the statement is true for P(b). This is usually just a matter of plugging in numbers.
- Assumption: Assume the statement is true for P(n).
- Inductive Step: Prove that *if* the statement is true for P(n), then it's true for P(n+1).

This is like dominoes. If the first domino falls, and one domino always knocks over the next one, then all the dominoes must fall.

Let's use this to prove that there are 2^n subsets of an n-element set.

- Definitions: let $S = \{a_1, a_2, a_3, \ldots, a_n\}$ be the n-element set.

- Base case: Prove there are 2^0 subsets of {}. This is true, since the only subset of {} is {}.
- Assume that there are 2^n subsets of $\{a_1, a_2, a_3, \ldots, a_n\}$.
- Prove that there are 2^{n+1} subsets of $\{a_1, a_2, a_3, \ldots, a_{n+1}\}$.

  Consider the subsets of $\{a_1, a_2, a_3, \ldots, a_{n+1}\}$. Exactly half will contain a_{n+1} and half will not.

  The subsets that do not contain a_{n+1} are just the subsets of $\{a_1, a_2, a_3, \ldots, a_n\}$. We assumed there are 2^n of those.

  Since we have the same number of subsets with x as without x, there are 2^n subsets with a_{n+1}.

  Therefore, we have $2^n + 2^n$ subsets, which is 2^{n+1}.

Many recursive algorithms can be proved valid with induction.

Topological Sort

A topological sort of a directed graph is a way of ordering the list of nodes such that if `(a, b)` is an edge in the graph then a will appear before b in the list. If a graph has cycles or is not directed, then there is no topological sort.

There are a number of applications for this. For example, suppose the graph represents parts on an assembly line. The edge (Handle, Door) indicates that you need to assemble the handle before the door. The topological sort would offer a valid ordering for the assembly line.

We can construct a topological sort with the following approach.

1. Identify all nodes with no incoming edges and add those nodes to our topological sort.
   - » We know those nodes are safe to add first since they have nothing that needs to come before them. Might as well get them over with!
   - » We know that such a node must exist if there's no cycle. After all, if we picked an arbitrary node we could just walk edges backwards arbitrarily. We'll either stop at some point (in which case we've found a node with no incoming edges) or we'll return to a prior node (in which case there is a cycle).
2. When we do the above, remove each node's outbound edges from the graph.
   - » Those nodes have already been added to the topological sort, so they're basically irrelevant. We can't violate those edges anymore.
3. Repeat the above, adding nodes with no incoming edges and removing their outbound edges. When all the nodes have been added to the topological sort, then we are done.

More formally, the algorithm is this:

1. Create a queue `order`, which will eventually store the valid topological sort. It is currently empty.
2. Create a queue `processNext`. This queue will store the next nodes to process.
3. Count the number of incoming edges of each node and set a class variable `node.inbound`. Nodes typically only store their outgoing edges. However, you can count the inbound edges by walking through each node `n` and, for each of its outgoing edges `(n, x)`, incrementing `x.inbound`.
4. Walk through the nodes again and add to `processNext` any node where `x.inbound == 0`.
5. While `processNext` is not empty, do the following:
   - » Remove first node n from `processNext`.

- » For each edge `(n, x)`, decrement `x.inbound`. If `x.inbound == 0`, append `x` to `processNext`.
- » Append `n` to `order`.

6. If `order` contains all the nodes, then it has succeeded. Otherwise, the topological sort has failed due to a cycle.

This algorithm does sometimes come up in interview questions. Your interviewer probably wouldn't expect you to know it offhand. However, it would be reasonable to have you derive it even if you've never seen it before.

Dijkstra's Algorithm

In some graphs, we might want to have edges with weights. If the graph represented cities, each edge might represent a road and its weight might represent the travel time. In this case, we might want to ask, just as your GPS mapping system does, what's the shortest path from your current location to another point p? This is where Dijksta's algorithm comes in.

Dijkstra's algorithm is a way to find the shortest path between two points in a weighted directed graph (which might have cycles). All edges must have positive values.

Rather than just stating what Dijkstra's algorithm is, let's try to derive it. Consider the earlier described graph. We could find the shortest path from `s` to `t` by literally taking all possible routes using actual time. (Oh, and we'll need a machine to clone ourselves.)

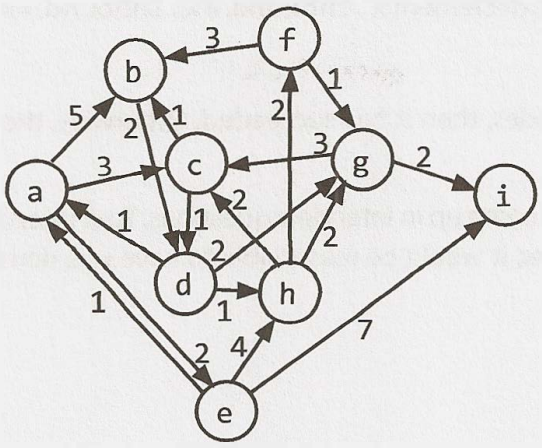
1. Start off at `s`.
2. For each of `s`'s outbound edges, clone ourselves and start walking. If the edge `(s, x)` has weight 5, we should actually take 5 minutes to get there.
3. Each time we get to a node, check if anyone's been there before. If so, then just stop. We're automatically not as fast as another path since someone beat us here from `s`. If no one has been here before, then clone ourselves and head out in all possible directions.
4. The first one to get to `t` wins.

This works just fine. But, of course, in the real algorithm we don't want to literally use a timer to find the shortest path.

Imagine that each clone could jump immediately from one node to its adjacent nodes (regardless of the edge weight), but it kept a `time_so_far` log of how long its path would have taken if it did walk at the "true" speed. Additionally, only one person moves at a time, and it's always the one with the lowest `time_so_far`. This is sort of how Dijkstra's algorithm works.

Dijkstra's algorithm finds the minimum weight path from a start node s to *every* node on the graph.

Consider the following graph.



Assume we are trying to find the shortest path from a to i. We'll use Dijkstra's algorithm to find the shortest path from a to all other nodes, from which we will clearly have the shortest path from a to i.

We first initialize several variables:

- path_weight[node]: maps from each node to the total weight of the shortest path. All values are initialized to infinity, except for path_weight[a] which is initialized to 0.
- previous[node]: maps from each node to the previous node in the (current) shortest path.
- remaining: a priority queue of all nodes in the graph, where each node's priority is defined by its path_weight.

Once we've initialized these values, we can start adjusting the values of path_weight.

> A (min) **priority queue** is an abstract data type that—at least in this case—supports insertion of an object and key, removing the object with the minimum key, and decreasing a key. (Think of it like a typical queue, except that, instead of removing the oldest item, it removes the item with the lowest or highest priority.) It is an abstract data type because it is defined by its behavior (its operations). Its underlying implementation can vary. You could implement a priority queue with an array or a min (or max) heap (or many other data structures).

We iterate through the nodes in remaining (until remaining is empty), doing the following:

1. Select the node in remaining with the lowest value in path_weight. Call this node n.
2. For each adjacent node, compare path_weight[x] (which is the weight of the current shortest path from a to x) to path_weight[n] + edge_weight[(n, x)]. That is, could we get a path from a to x with lower weight by going through n instead of our current path? If so, update path_weight and previous.
3. Remove n from remaining.

When remaining is empty, then path_weight stores the weight of the current shortest path from a to each node. We can reconstruct this path by tracing through previous.

Let's walk through this on the above graph.

1. The first value of n is a. We look at its adjacent nodes (b, c, and e), update the values of path_weight (to 5, 3, and 2) and previous (to a) and then remove a from remaining.
2. Then, we go to the next smallest node, which is e. We previously updated path_weight[e] to be 2. Its adjacent nodes are h and i, so we update path_weight (to 6 and 9) and previous for both of those.

Observe that 6 is `path_weight[e]` (which is 2) + the weight of the edge `(e, h)` (which is 4).

3. The next smallest node is `c`, which has `path_weight` 3. Its adjacent nodes are b and d. The value of `path_weight[d]` is infinity, so we update it to 4 (which is `path_weight[c] + weight(edge c, d)`. The value of `path_weight[b]` has been previously set to 5. However, since `path_weight[c] + weight(edge c, b)` (which is 3 + 1 = 4) is less than 5, we update `path_weight[b]` to 4 and `previous` to `c`. This indicates that we would improve the path from a to b by going through `c`.

We continue doing this until `remaining` is empty. The following diagram shows the changes to the `path_weight` (left) and `previous` (right) at each step. The topmost row shows the current value for n (the node we are removing from `remaining`). We black out a row after it has been removed from `remaining`.

	INITIAL		n = a		n = e		n = c		n = b		n = d		n = h		n = g		n = f		FINAL	
	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr
a	0	-	removed																0	-
b	∞	-	5	a			4	c	removed										4	c
c	∞	-	3	a			removed												3	a
d	∞	-					4	c			removed								4	c
e	∞	-	2	a	removed														2	a
f	∞	-											7	h			removed		7	h
g	∞	-									6	d			removed				6	d
h	∞	-			6	e					5	d	removed						5	d
i	∞	-	∞	-	9	e									8	g			8	g

Once we're done, we can follow this chart backwards, starting at `i` to find the actual path. In this case, the smallest weight path has weight 8 and is `a -> c -> d -> g -> i`.

Priority Queue and Runtime

As mentioned earlier, our algorithm used a priority queue, but this data structure can be implemented in different ways.

The runtime of this algorithm depends heavily on the implementation of the priority queue. Assume you have v vertices and e nodes.

- If you implemented the priority queue with an array, then you would call `remove_min` up to v times. Each operation would take $O(v)$ time, so you'd spend $O(v^2)$ time in the `remove_min` calls. Additionally, you would update the values of `path_weight` and `previous` at most once per edge, so that's $O(e)$ time doing those updates. Observe that e must be less than of equal to v^2 since you can't have more edges than there are pairs of vertices. Therefore, the total runtime is $O(v^2)$.
- If you implemented the priority queue with a min heap, then the `remove_min` calls will each take $O(\log v)$ time (as will inserting and updating a key). We will do one `remove_min` call for each vertex, so that's $O(v \log v)$ (v vertices at $O(\log v)$ time each). Additionally, on each edge, we might call one update key or insert operation, so that's $O(e \log v)$. The total runtime is $O((v + e) \log v)$.

Which one is better? Well, that depends. If the graph has a lot of edges, then v^2 will be close to e. In this case, you might be better off with the array implementation, as $O(v^2)$ is better than $O((v + v^2) \log v)$. However, if the graph is sparse, then e is much less than v^2. In this case, the min heap implementation may be better.

Hash Table Collision Resolution

Essentially any hash table can have collisions. There are a number of ways of handling this.

Chaining with Linked Lists

With this approach (which is the most common), the hash table's array maps to a linked list of items. We just add items to this linked list. As long as the number of collisions is fairly small, this will be quite efficient.

In the worst case, lookup is `O(n)`, where `n` is the number of elements in the hash table. This would only happen with either some very strange data or a very poor hash function (or both).

Chaining with Binary Search Trees

Rather than storing collisions in a linked list, we could store collisions in a binary search tree. This will bring the worst-case runtime to `O(log n)`.

In practice, we would rarely take this approach unless we expected an extremely nonuniform distribution.

Open Addressing with Linear Probing

In this approach, when a collision occurs (there is already an item stored at the designated index), we just move on to the next index in the array until we find an open spot. (Or, sometimes, some other fixed distance, like the `index + 5`.)

If the number of collisions is low, this is a very fast and space-efficient solution.

One obvious drawback of this is that the total number of entries in the hash table is limited by the size of the array. This is not the case with chaining.

There's another issue here. Consider a hash table with an underlying array of size 100 where indexes 20 through 29 are filled (and nothing else). What are the odds of the next insertion going to index 30? The odds are 10% because an item mapped to any index between 20 and 30 will wind up at index 30. This causes an issue called *clustering*.

Quadratic Probing and Double Hashing

The distance between probes does not need to be linear. You could, for example, increase the probe distance quadratically. Or, you could use a second hash function to determine the probe distance.

Rabin-Karp Substring Search

The brute force way to search for a substring S in a larger string B takes `O(s(b-s))` time, where `s` is the length of S and b is the length of B. We do this by searching through the first `b - s + 1` characters in B and, for each, checking if the next `s` characters match S.

The Rabin-Karp algorithm optimizes this with a little trick: if two strings are the same, they must have the same hash value. (The converse, however, is not true. Two different strings can have the same hash value.)

Therefore, if we efficiently precompute a hash value for each sequence of `s` characters within B, we can find the locations of S in `O(b)` time. We then just need to validate that those locations really do match S.

For example, imagine our hash function was simply the sum of each character (where space = 0, a = 1, b = 2, and so on). If S is `ear` and B = `doe are hearing me`, we'd then just be looking for sequences where the sum is 24 (e + a + r). This happens three times. For each of those locations, we'd check if the string really is `ear`.

char:	d	o	e		a	r	e		h	e	a	r	i	n	g		m	e
code:	4	15	5	0	1	18	5	0	8	5	1	18	9	14	7	0	13	5
sum of next 3:	24	20	6	19	24	23	13	13	14	24	28	41	30	21	20	18		

If we computed these sums by doing `hash('doe')`, then `hash('oe ')`, then `hash('e a')`, and so on, we would still be at `O(s(b-s))` time.

Instead, we compute the hash values by recognizing that `hash('oe ') = hash('doe') - code('d') + code(' ')`. This takes `O(b)` time to compute all the hashes.

You might argue that, still, in the worst case this will take `O(s(b-s))` time since many of the hash values could match. That's absolutely true—for this hash function.

In practice, we would use a better *rolling hash function*, such as the Rabin fingerprint. This essentially treats a string like doe as a base 128 (or however many characters are in our alphabet) number.

$$\text{hash('doe')} = \text{code('d')} * 128^2 + \text{code('o')} * 128^1 + \text{code('e')} * 128^0$$

This hash function will allow us to remove the d, shift the o and e, and then add in the space.

$$\text{hash('oe ')} = (\text{hash('doe')} - \text{code('d')} * 128^2) * 128 + \text{code(' ')}$$

This will considerably cut down on the number of false matches. Using a good hash function like this will give us expected time complexity of `O(s + b)`, although the worst case is `O(sb)`.

Usage of this algorithm comes up fairly frequently in interviews, so it's useful to know that you can identify substrings in linear time.

▸ AVL Trees

An AVL tree is one of two common ways to implement tree balancing. We will only discuss insertions here, but you can look up deletions separately if you're interested.

Properties

An AVL tree stores in each node the height of the subtrees rooted at this node. Then, for any node, we can check if it is height balanced: that the height of the left subtree and the height of the right subtree differ by no more than one. This prevents situations where the tree gets too lopsided.

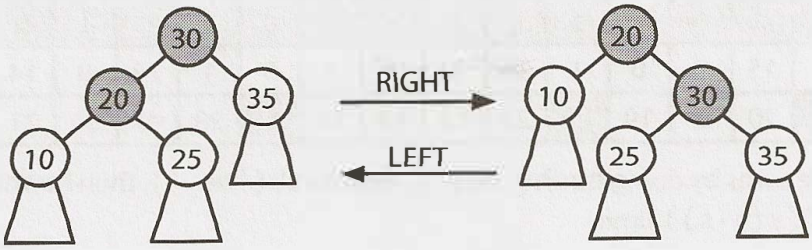
```
balance(n) = n.left.height - n.right.height

-1 <= balance(n) <= 1
```

Inserts

When you insert a node, the balance of some nodes might change to -2 or 2. Therefore, when we "unwind" the recursive stack, we check and fix the balance at each node. We do this through a series of rotations.

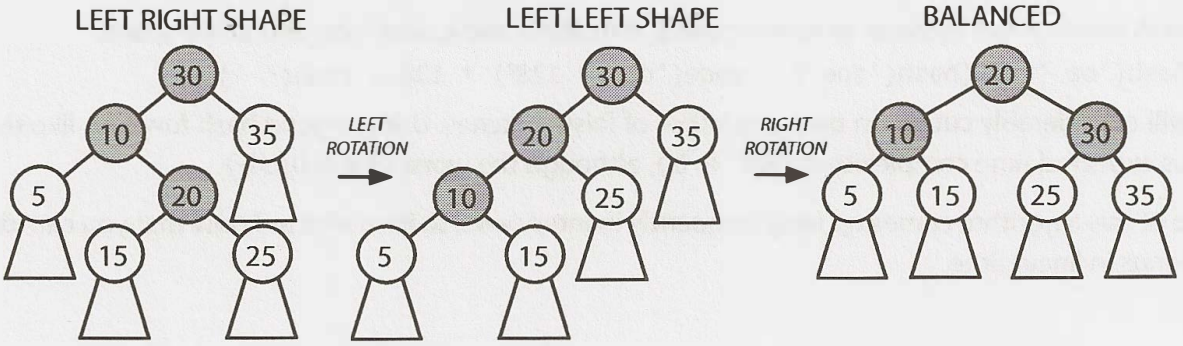
Rotations can be either left or right rotations. The right rotation is an inverse of the left rotation.



Depending on the balance and where the imbalance occurs, we fix it in a different way.

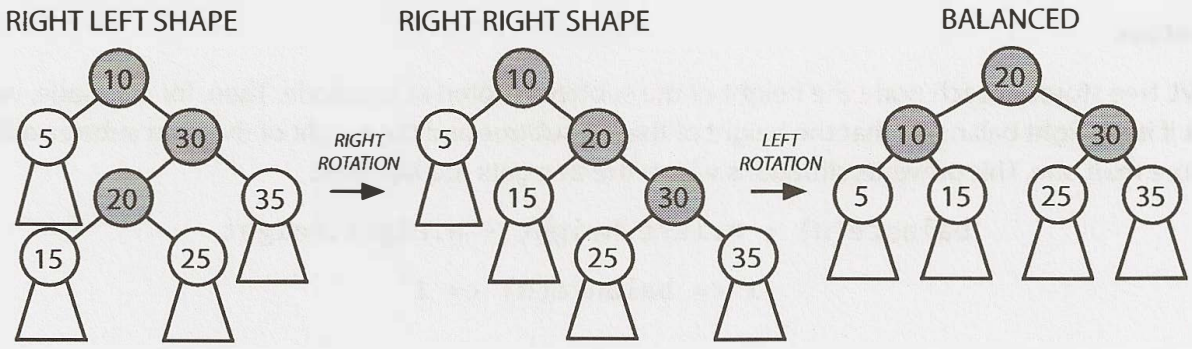
- *Case 1: Balance is 2.*

  In this case, the left's height is two bigger than the right's height. If the left side is larger, the left subtree's extra nodes must be hanging to the left (as in LEFT LEFT SHAPE) or hanging to the right (as in LEFT RIGHT SHAPE). If it looks like the LEFT RIGHT SHAPE, transform it with the rotations below into the LEFT LEFT SHAPE then into BALANCED. If it looks like the LEFT LEFT SHAPE already, just transform it into BALANCED.



- *Case 2: Balance is -2.*

  This case is the mirror image of the prior case. The tree will look like either the RIGHT LEFT SHAPE or the RIGHT RIGHT SHAPE. Perform the rotations below to transform it into BALANCED.



In both cases, "balanced" just means that the `balance` of the tree is between -1 and 1. It does not mean that the `balance` is 0.

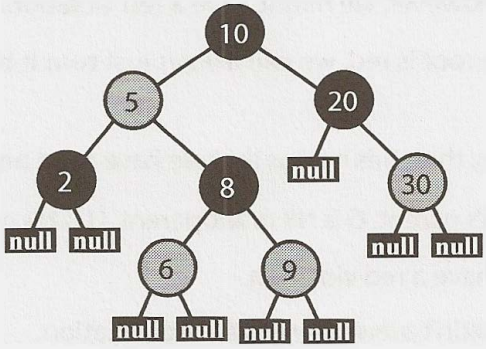
We recurse up the tree, fixing any imbalances. If we ever achieve a balance of 0 on a subtree, then we know that we have completed all the balances. This portion of the tree will not cause another, higher subtree to have a balance of -2 or 2. If we were doing this non-recursively, then we could break from the loop.

Red-Black Trees

Red-black trees (a type of self-balancing binary search tree) do not ensure quite as strict balancing, but the balancing is still good enough to ensure `O(log N)` insertions, deletions, and retrievals. They require a bit less memory and can rebalance faster (which means faster insertions and removals), so they are often used in situations where the tree will be modified frequently.

Red-black trees operate by enforcing a quasi-alternating red and black coloring (under certain rules, described below) and then requiring every path from a node to its leaves to have the same number of black nodes. Doing so leads to a reasonably balanced tree.

The tree below is a red-black tree (where the red nodes are indicated with gray):



Properties

1. Every node is either red or black.
2. The root is black.
3. The leaves, which are NULL nodes, are considered black.
4. Every red node must have two black children. That is, a red node cannot have red children (although a black node can have black children).
5. Every path from a node to its leaves must have the same number of black children.

Why It Balances

Property #4 means that two red nodes cannot be adjacent in a path (e.g., parent and child). Therefore, no more than half the nodes in a path can be red.

Consider two paths from a node (say, the root) to its leaves. The paths must have the same number of black nodes (property #5), so let's assume that their red node counts are as different as possible: one path contains the minimum number of red nodes and the other one contains the maximum number.

- Path 1 (Min Red): The minimum number of red nodes is zero. Therefore, path 1 has b nodes total.
- Path 2 (Max Red): The maximum number of red nodes is b, since red nodes must have black children and there are b black nodes. Therefore, path 2 has 2b nodes total.

Therefore, even in the most extreme case, the lengths of paths cannot differ by more than a factor of two. That's good enough to ensure an `O(log N)` find and insert runtime.

If we can maintain these properties, we'll have a (sufficiently) balanced tree—good enough to ensure `O(log N)` insert and find, anyway. The question then is how to maintain these properties efficiently. We'll only discuss insertion here, but you can look up deletion on your own.

Insertion

Inserting a new node into a red-black tree starts off with a typical binary search tree insertion.

- New nodes are inserted at a leaf, which means that they replace a black node.
- New nodes are always colored red and are given two black leaf (NULL) nodes.

Once we've done that, we fix any resulting red-black property violations. We have two possible violations:

- Red violations: A red node has a red child (or the root is red).
- Black violations: One path has more blacks than another path.

The node inserted is red. We didn't change the number of black nodes on any path to a leaf, so we know that we won't have a black violation. However, we might have a red violation.

In the special case that where the root is red, we can always just turn it black to satisfy property 2, without violating the other constraints.

Otherwise, if there's a red violation, then this means that we have a red node under another red node. Oops!

Let's call N the current node. P is N's parent. G is N's grandparent. U is N's uncle and P's sibling. We know that:

- N is red and P is red, since we have a red violation.
- G is definitely black, since we didn't *previously* have a red violation.

The unknown parts are:

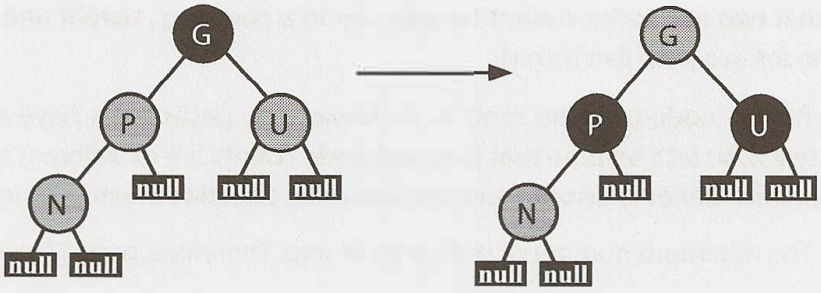
- U could be either red or black.
- U could be either a left or right child.
- N could be either a left or right child.

By simple combinatorics, that's eight cases to consider. Fortunately some of these cases will be equivalent.

- **Case 1: U is red.**

  It doesn't matter whether U is a left or right child, nor whether P is a left or right child. We can merge four of our eight cases into one.

  If U is red, we can just toggle the colors of P, U, and G. Flip G from black to red. Flip P and U from red to black. We haven't changed the number of black nodes in any path.



  However, by making G red, we might have created a red violation with G's parent. If so, we recursively apply the full logic to handle a red violation, where this G becomes the new N.

  Note that in the general recursive case, N, P, and U may also have subtrees in place of each black NULL (the leaves shown). In Case 1, these subtrees stay attached to the same parents, as the tree structure remains unchanged.

- **Case 2: U is black.**

We'll need to consider the configurations (left vs. right child) of N and U. In each case, our goal is to fix up the red violation (red on top of red) without:

» Messing up the ordering of the binary search tree.

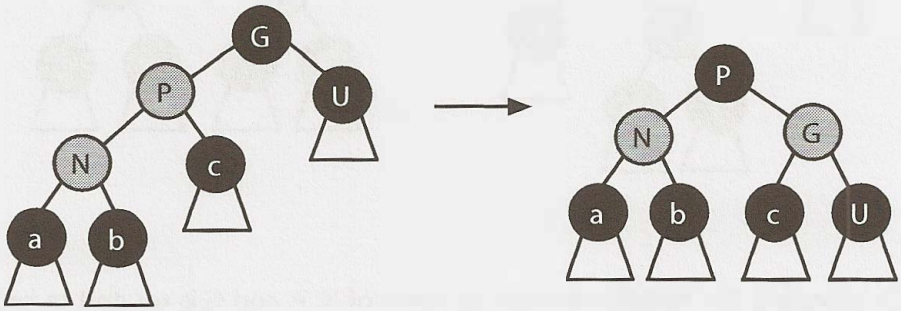
» *Introducing a black violation (more black nodes on one path than another).*

If we can do this, we're good. In each of the cases below, the red violation is fixed with rotations that maintain the node ordering.

Further, the below rotations maintain the exact number of black nodes in each path through the affected portion of the tree that were in place beforehand. The children of the rotating section are either NULL leaves or subtrees that remain internally unchanged.

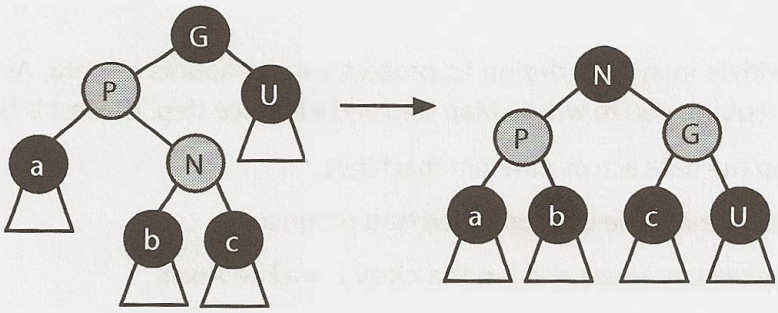
Case A: N and P are both left children.

We resolve the red violation with the rotation of N, P, and G and the associated recoloring shown below. If you picture the in-order traversal, you can see the rotation maintains the node ordering (a <= N <= b <= P <= c <= G <= U). The tree maintains the same, equal number of black nodes in the path down to each subtree a, b, c, and U (which may all be NULL) .



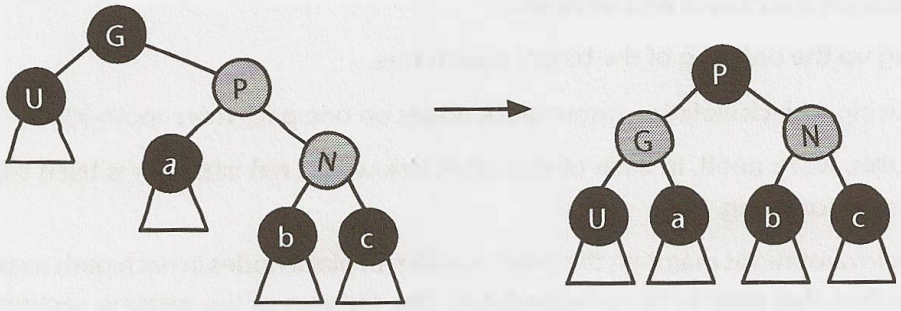
Case B: P is a left child, and N is a right child.

The rotations in Case B resolve the red violation and maintain the in-order property: a <= P <= b <= N <= c <= G <= U. Again, the count of the black nodes remains constant in each path down to the leaves (or subtrees).



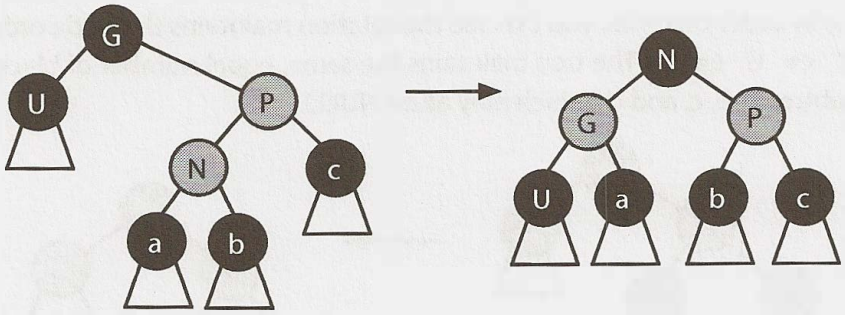
Case C: N and P are both right children.

This is a mirror image of case A.



Case D: N is a left child, and P is a right child.

This is a mirror image of case B.



In each of Case 2's subcases, the middle element by value of N, P, and G is rotated to become the root of what was G's subtree, and that element and G swap colors.

That said, do not try to just memorize these cases. Rather, study why they work. How does each one ensure no red violations, no black violations, and no violations of the binary search tree property?

▸ MapReduce

MapReduce is used widely in system design to process large amounts of data. As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.

1. The system splits up the data across different machines.
2. Each machine starts running the user-provided `Map` program.
3. The `Map` program takes some data and emits a `<key, value>` pair.
4. The system-provided `Shuffle` process reorganizes the data so that all <key, value> pairs associated with a given key go to the same machine, to be processed by `Reduce`.
5. The user-provided `Reduce` program takes a key and a set of associated values and "reduces" them in some way, emitting a new key and value. The results of this might be fed back into the `Reduce` program for more reducing.

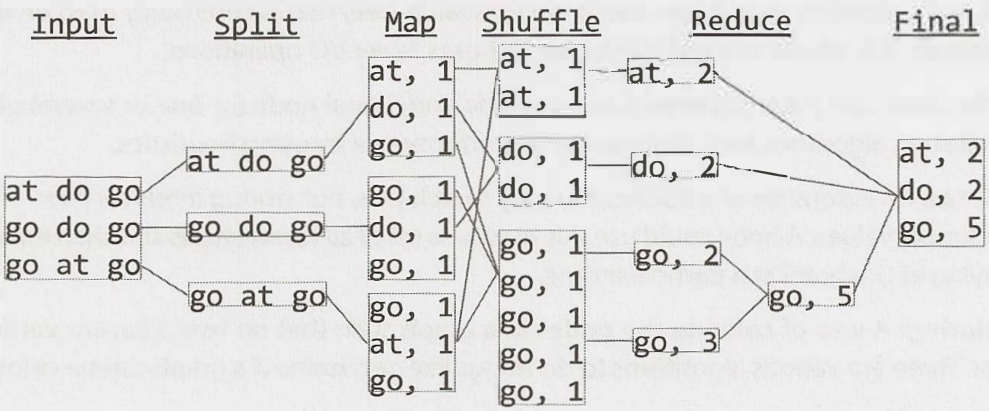
The typical example of using MapReduce—basically the "Hello World" of MapReduce—is counting the frequency of words within a set of documents.

Of course, you could write this as a single function that reads in all the data, counts the number of times each word appears via a hash table, and then outputs the result.

MapReduce allows you to process the document in parallel. The `Map` function reads in a document and emits just each individual word and the count (which is always 1). The `Reduce` function reads in keys (words) and associated values (counts). It emits the sum of the counts. This sum could possibly wind up as input for another call to `Reduce` on the same key (as shown in the diagram).

```
1  void map(String name, String document):
2    for each word w in document:
3      emit(w, 1)
4
5  void reduce(String word, Iterator partialCounts):
6    int sum = 0
7    for each count in partialCounts:
8      sum += count
9    emit(word, sum)
```

The diagram below shows how this might work on this example.



Here's another example: You have a list of data in the form {City, Temperature, Date}. Calculate the average temperature in each city every year. For example {(2012, Philadelphia, 58.2), (2011, Philadelphia, 56.6), (2012, Seattle, 45.1)}.

- **Map**: The Map step outputs a key value pair where the key is `City_Year` and the value is `(Temperature, 1)`. The '1' reflects that this is the average temperature out of one data point. This will be important for the `Reduce` step.
- **Reduce**: The Reduce step will be given a list of temperatures that correspond with a particular city and year. It must use these to compute the average temperature for this input. You cannot simply add up the temperatures and divide by the number of values.

  To see this, imagine we have five data points for a particular city and year: 25, 100, 75, 85, 50. The Reduce step might only get some of this data at once. If you averaged {75, 85} you would get 80. This might end up being input for another Reduce step with 50, and it would be a mistake to just naively average 80 and 50. The 80 has more weight.

  Therefore, our Reduce step instead takes in {(80, 2), (50, 1)}, then sums the *weighted* temperatures. So it does 80 * 2 + 50 * 1 and then divides by (2 + 1) to get an average temperature of 70. It then emits (70, 3).

  Another Reduce step might reduce {(25, 1), (100, 1)} to get (62.5, 2). If we reduce this with (70, 3) we get the final answer: (67, 5). In other words, the average temperature in this city for this year was 67 degrees.

We could do this in other ways, too. We could have just the city as the key, and the value be (Year, Temperature, Count). The Reduce step would do essentially the same thing, but would have to group by Year itself.

In many cases, it's useful to think about what the Reduce step should do first, and then design the Map step around that. What data does Reduce need to have to do its job?

Additional Studying

So, you've mastered this material and you want to learn even more? Okay. Here are some topics to get you started:

- **Bellman-Ford Algorithm**: Finds the shortest paths from a single node in a weighted directed graph with positive and negative edges.
- **Floyd-Warshall Algorithm**: Finds the shortest paths in a weighted graph with positive or negative weight edges (but no negative weight cycles).
- **Minimum Spanning Trees**: In a weighted, connected, undirected graph, a spanning tree is a tree that connects all the vertices. The minimum spanning tree is the spanning tree with minimum weight. There are various algorithms to do this.
- **B-Trees:** A self-balancing search tree (not a binary search tree) that is commonly used on disks or other storage devices. It is similar to a red-black tree, but uses fewer I/O operations.
- **A***: Find the least-cost path between a source node and a goal node (or one of several goal nodes). It extends Dijkstra's algorithm and achieves better performance by using heuristics.
- **Interval Trees**: An extension of a balanced binary search tree, but storing intervals (low -> high ranges) instead of simple values. A hotel could use this to store a list of all reservations and then efficiently detect who is staying at the hotel at a particular time.
- **Graph coloring**: A way of coloring the nodes in a graph such that no two adjacent vertices have the same color. There are various algorithms to do things like determine if a graph can be colored with only K colors.
- **P, NP, and NP-Complete**: P, NP, and NP-Complete refer to classes of problems. P problems are problems that can be quickly solved (where "quickly" means polynomial time). NP problems are those where, given a solution, the solution can be quickly verified. NP-Complete problems are a subset of NP problems that can all be reduced to each other (that is, if you found a solution to one problem, you could tweak the solution to solve other problems in the set in polynomial time).

  It is an open (and very famous) question whether P = NP, but the answer is generally believed to be no.
- **Combinatorics and Probability**: There are various things you can learn about here, such as random variables, expected value, and n-choose-k.
- **Bipartite Graph**: A bipartite graph is a graph where you can divide its nodes into two sets such that every edge stretches across the two sets (that is, there is never an edge between two nodes in the same set). There is an algorithm to check if a graph is a bipartite graph. Note that a bipartite graph is equivalent to a graph that can be colored with two colors.
- **Regular Expressions**: You should know that regular expressions exist and what they can be used for (roughly). You can also learn about how an algorithm to match regular expressions would work. Some of the basic syntax behind regular expressions could be useful as well.

There is of course a great deal more to data structures and algorithms. If you're interested in exploring these topics more deeply, I recommend picking up the hefty *Introduction to Algorithms* ("CLRS" by Cormen, Leiserson, Rivest and Stein) or *The Algorithm Design Manual* (by Steven Skiena).

Hints

XIII

Interviewers usually don't just hand you a question and expect you to solve it. Rather, they will typically offer guidance when you're stuck, especially on the harder questions. It's impossible to totally simulate the interview experience in a book, but these hints are designed to get you closer.

Try to solve the questions independently when possible. But it's okay to look for some help when you are really struggling. Again, struggling is a normal part of the process.

I've organized the hints somewhat randomly here, such that all the hints for a problem aren't adjacent. This way you won't accidentally see the second hint when you're reading the first hint.

I

Hints for Data Structures

#1. 1.2 Describe what it means for two strings to be permutations of each other. Now, look at that definition you provided. Can you check the strings against that definition?

#2. 3.1 A stack is simply a data structure in which the most recently added elements are removed first. Can you simulate a single stack using an array? Remember that there are many possible solutions, and there are tradeoffs of each.

#3. 2.4 There are many solutions to this problem, most of which are equally optimal in runtime. Some have shorter, cleaner code than others. Can you brainstorm different solutions?

#4. 4.10 If T2 is a subtree of T1, how will its in-order traversal compare to T1's? What about its pre-order and post-order traversal?

#5. 2.6 A palindrome is something which is the same when written forwards and backwards. What if you reversed the linked list?

#6. 4.12 Try simplifying the problem. What if the path had to start at the root?

#7. 2.5 Of course, you could convert the linked lists to integers, compute the sum, and then convert it back to a new linked list. If you did this in an interview, your interviewer would likely accept the answer, and then see if you could do this without converting it to a number and back.

#8. 2.2 What if you knew the linked list size? What is the difference between finding the Kth-to-last element and finding the Xth element?

#9. 2.1 Have you tried a hash table? You should be able to do this in a single pass of the linked list.

#10. 4.8 If each node has a link to its parent, we could leverage the approach from question 2.7 on page 95. However, our interviewer might not let us make this assumption.

#11. 4.10 The in-order traversals won't tell us much. After all, every binary search tree with the same values (regardless of structure) will have the same in-order traversal. This is what in-order traversal means: contents are in-order. (And if it won't work in the specific case of a binary search tree, then it certainly won't work for a general binary tree.) The pre-order traversal, however, is much more indicative.

#12. 3.1 We could simulate three stacks in an array by just allocating the first third of the array to the first stack, the second third to the second stack, and the final third to the third stack. One might actually be much bigger than the others, though. Can we be more flexible with the divisions?

#13. 2.6 Try using a stack.

#14. 4.12 Don't forget that paths could overlap. For example, if you're looking for the sum 6, the paths `1->3->2` and `1->3->2->4->-6->2` are both valid.

#15. 3.5 One way of sorting an array is to iterate through the array and insert each element into a new array in sorted order. Can you do this with a stack?

#16. 4.8 The first common ancestor is the deepest node such that p and q are both descendants. Think about how you might identify this node.

#17. 1.8 If you just cleared the rows and columns as you found 0s, you'd likely wind up clearing the whole matrix. Try finding the cells with zeros first before making any changes to the matrix.

#18. 4.10 You may have concluded that if `T2.preorderTraversal()` is a substring of `T1.preorderTraversal()`, then T2 is a subtree of T1. This is almost true, except that the trees could have duplicate values. Suppose T1 and T2 have all duplicate values but different structures. The pre-order traversals will look the same even though T2 is not a subtree of T1. How can you handle situations like this?

#19. 4.2 A minimal binary tree has about the same number of nodes on the left of each node as on the right. Let's focus on just the root for now. How would you ensure that about the same number of nodes are on the left of the root as on the right?

#20. 2.7 You can do this in `O(A+B)` time and `O(1)` additional space. That is, you do not need a hash table (although you could do it with one).

#21. 4.4 Think about the definition of a balanced tree. Can you check that condition for a single node? Can you check it for every node?

#22. 3.6 We could consider keeping a single linked list for dogs and cats, and then iterating through it to find the first dog (or cat). What is the impact of doing this?

#23. 1.5 Start with the easy thing. Can you check each of the conditions separately?

#24. 2.4 Consider that the elements don't have to stay in the same relative order. We only need to ensure that elements less than the pivot must be before elements greater than the pivot. Does that help you come up with more solutions?

#25. 2.2 If you don't know the linked list size, can you compute it? How does this impact the runtime?

#26. 4.7 Build a directed graph representing the dependencies. Each node is a project and an edge exists from A to B if B depends on A (A must be built before B). You can also build it the other way if it's easier for you.

#27. 3.2 Observe that the minimum element doesn't change very often. It only changes when a smaller element is added, or when the smallest element is popped.

#28. 4.8 How would you figure out if p is a descendent of a node n?

#29. 2.6 Assume you have the length of the linked list. Can you implement this recursively?

#30. 2.5 Try recursion. Suppose you have two lists, A = `1->5->9` (representing 951) and B = `2->3->6->7` (representing 7632), and a function that operates on the remainder of the lists (`5->9` and `3->6->7`). Could you use this to create the `sum` method? What is the relationship between `sum(1->5->9, 2->3->6->7)` and `sum(5->9, 3->6->7)`?

#31. 4.10 Although the problem seems like it stems from duplicate values, it's really deeper than that. The issue is that the pre-order traversal is the same only because there are null nodes that we skipped over (because they're null). Consider inserting a placeholder value into the pre-order traversal string whenever you reach a null node. Register the null node as a "real" node so that you can distinguish between the different structures.

#32. 3.5 Imagine your secondary stack is sorted. Can you insert elements into it in sorted order? You might need some extra storage. What could you use for extra storage?

#33. 4.4 If you've developed a brute force solution, be careful about its runtime. If you are computing the height of the subtrees for each node, you could have a pretty inefficient algorithm.

#34. 1.9 If a string is a rotation of another, then it's a rotation at a particular point. For example, a rotation of `waterbottle` at character 3 means cutting `waterbottle` at character 3 and putting the right half (`erbottle`) before the left half (`wat`).

#35. 4.5 If you traversed the tree using an in-order traversal and the elements were truly in the right order, does this indicate that the tree is actually in order? What happens for duplicate elements? If duplicate elements are allowed, they must be on a specific side (usually the left).

#36. 4.8 Start with the root. Can you identify if root is the first common ancestor? If it is not, can you identify which side of root the first common ancestor is on?

#37. 4.10 Alternatively, we can handle this problem recursively. Given a specific node within T1, can we check to see if its subtree matches T2?

#38. 3.1 If you want to allow for flexible divisions, you can shift stacks around. Can you ensure that all available capacity is used?

#39. 4.9 What is the very first value that must be in each array?

#40. 2.1 Without extra space, you'll need $O(N^2)$ time. Try using two pointers, where the second one searches ahead of the first one.

#41. 2.2 Try implementing it recursively. If you could find the (K-1)th to last element, can you find the Kth element?

#42. 4.11 Be very careful in this problem to ensure that each node is equally likely and that your solution doesn't slow down the speed of standard binary search tree algorithms (like `insert`, `find`, and `delete`). Also, remember that even if you assume that it's a balanced binary search tree, this doesn't mean that the tree is full/complete/perfect.

#43. 3.5 Keep the secondary stack in sorted order, with the biggest elements on the top. Use the primary stack for additional storage.

#44. 1.1 Try a hash table.

#45. 2.7 Examples will help you. Draw a picture of intersecting linked lists and two equivalent linked lists (by value) that do not intersect.

#46. 4.8 Try a recursive approach. Check if p and q are descendants of the left subtree and the right subtree. If they are descendants of different subtrees, then the current node is the first common ancestor. If they are descendants of the same subtree, then that subtree holds the first common ancestor. Now, how do you implement this efficiently?

#47. 4.7 Look at this graph. Is there any node you can identify that will definitely be okay to build first?

#48. 4.9 The root is the very first value that must be in every array. What can you say about the order of the values in the left subtree as compared to the values in the right subtree? Do the left subtree values need to be inserted before the right subtree?

#49. 4.4 What if you could modify the binary tree node class to allow a node to store the height of its subtree?

#50. 2.8 There are really two parts to this problem. First, detect if the linked list has a loop. Second, figure out where the loop starts.

#51. 1.7 Try thinking about it layer by layer. Can you rotate a specific layer?

#52. 4.12 If each path had to start at the root, we could traverse all possible paths starting from the root. We can track the sum as we go, incrementing `totalPaths` each time we find a path with our target sum. Now, how do we extend this to paths that can start anywhere? Remember: Just get a brute-force algorithm done. You can optimize later.

#53. 1.3 It's often easiest to modify strings by going from the end of the string to the beginning.

#54. 4.11 This is your own binary search tree class, so you can maintain any information about the tree structure or nodes that you'd like (provided it doesn't have other negative implications, like making `insert` much slower). In fact, there's probably a reason the interview question specified that it was your own class. You probably need to store some additional information in order to implement this efficiently.

#55. 2.7 Focus first on just identifying if there's an intersection.

#56. 3.6 Let's suppose we kept separate lists for dogs and cats. How would we find the oldest animal of any type? Be creative!

#57. 4.5 To be a binary search tree, it's not sufficient that the `left.value <= current.value < right.value` for each node. Every node on the left must be less than the current node, which must be less than all the nodes on the right.

#58. 3.1 Try thinking about the array as circular, such that the end of the array "wraps around" to the start of the array.

#59. 3.2 What if we kept track of extra data at each stack node? What sort of data might make it easier to solve the problem?

#60. 4.7 If you identify a node without any incoming edges, then it can definitely be built. Find this node (there could be multiple) and add it to the build order. Then, what does this mean for its outgoing edges?

#61. 2.6 In the recursive approach (we have the length of the list), the middle is the base case: `isPermutation(middle)` is true. The node `x` to the immediate left of the middle: What can that node do to check if `x->middle->y` forms a palindrome? Now suppose that checks out. What about the previous node `a`? If `x->middle->y` is a palindrome, how can it check that `a->x->middle->y->b` is a palindrome?

#62. 4.11 As a naive "brute force" algorithm, can you use a tree traversal algorithm to implement this algorithm? What is the runtime of this?

#63. 3.6 Think about how you'd do it in real life. You have a list of dogs in chronological order and a list of cats in chronological order. What data would you need to find the oldest animal? How would you maintain this data?

#64. 3.3 You will need to keep track of the size of each substack. When one stack is full, you may need to create a new stack.

#65. 2.7 Observe that two intersecting linked lists will always have the same last node. Once they intersect, all the nodes after that will be equal.

#66. 4.9 The relationship between the left subtree values and the right subtree values is, essentially, anything. The left subtree values could be inserted before the right subtree, or the reverse (right values before left), or any other ordering.

#67. 2.2 You might find it useful to return multiple values. Some languages don't directly support this, but there are workarounds in essentially any language. What are some of those workarounds?

#68. 4.12 To extend this to paths that start anywhere, we can just repeat this process for all nodes.

#69. 2.8 To identify if there's a cycle, try the "runner" approach described on page 93. Have one pointer move faster than the other.

#70. 4.8 In the more naive algorithm, we had one method that indicated if `x` is a descendent of n, and another method that would recurse to find the first common ancestor. This is repeatedly searching the same elements in a subtree. We should merge this into one `firstCommonAncestor` function. What return values would give us the information we need?

#71. 2.5 Make sure you have considered linked lists that are not the same length.

#72. 2.3 Picture the list `1->5->9->12`. Removing 9 would make it look like `1->5->12`. You only have access to the 9 node. Can you make it look like the correct answer?

#73. 4.2 You could implement this by finding the "ideal" next element to add and repeatedly calling `insertValue`. This will be a bit inefficient, as you would have to repeatedly traverse the tree. Try recursion instead. Can you divide this problem into subproblems?

#74. 1.8 Can you use `O(N)` additional space instead of $O(N^2)$? What information do you really need from the list of cells that are zero?

#75. 4.11 Alternatively, you could pick a random depth to traverse to and then randomly traverse, stopping when you get to that depth. Think this through, though. Does this work?

#76. 2.7 You can determine if two linked lists intersect by traversing to the end of each and comparing their tails.

#77. 4.12 If you've designed the algorithm as described thus far, you'll have an `O(N log N)` algorithm in a balanced tree. This is because there are `N` nodes, each of which is at depth `O(log N)` at worst. A node is touched once for each node above it. Therefore, the `N` nodes will be touched `O(log N)` time. There is an optimization that will give us an `O(N)` algorithm.

#78. 3.2 Consider having each node know the minimum of its "substack" (all the elements beneath it, including itself).

#79. 4.6 Think about how an in-order traversal works and try to "reverse engineer" it.

#80. 4.8 The `firstCommonAncestor` function could return the first common ancestor (if p and q are both contained in the tree), p if p is in the tree and not q, q if q is in the tree and not p, and `null` otherwise.

#81. 3.3 Popping an element at a specific substack will mean that some stacks aren't at full capacity. Is this an issue? There's no right answer, but you should think about how to handle this.

#82. 4.9 Break this down into subproblems. Use recursion. If you had all possible sequences for the left subtree and the right subtree, how could you create all possible sequences for the entire tree?

#83. 2.8 You can use two pointers, one moving twice as fast as the other. If there is a cycle, the two pointers will collide. They will land at the same location at the same time. Where do they land? Why there?

#84. 1.2 There is one solution that is `O(N log N)` time. Another solution uses some space, but is `O(N)` time.

#85. 4.7 Once you decide to build a node, its outgoing edge can be deleted. After you've done this, can you find other nodes that are free and clear to build?

#86. 4.5 If every node on the left must be less than or equal to the current node, then this is really the same thing as saying that the biggest node on the left must be less than or equal to the current node.

#87. 4.12 What work is duplicated in the current brute-force algorithm?

#88. 1.9 We are essentially asking if there's a way of splitting the first string into two parts, `x` and `y`, such that the first string is `xy` and the second string is `yx`. For example, `x = wat` and `y = erbottle`. The first string is `xy = waterbottle`. The second string is `yx = erbottlewat`.

#89. 4.11 Picking a random depth won't help us much. First, there's more nodes at lower depths than higher depths. Second, even if we re-balanced these probabilities, we could hit a "dead end" where we meant to pick a node at depth 5 but hit a leaf at depth 3. Re-balancing the probabilities is an interesting , though.

#90. 2.8 If you haven't identified the pattern of where the two pointers start, try this: Use the linked list 1->2->3->4->5->6->7->8->9->?, where the ? links to another node. Try making the ? the first node (that is, the 9 points to the 1 such that the entire linked list is a loop). Then make the ? the node 2. Then the node 3. Then the node 4. What is the pattern? Can you explain why this happens?

#91. 4.6 Here's one step of the logic: The successor of a specific node is the leftmost node of the right subtree. What if there is no right subtree, though?

#92. 1.6 Do the easy thing first. Compress the string, then compare the lengths.

#93. 2.7 Now, you need to find where the linked lists intersect. Suppose the linked lists were the same length. How could you do this?

#94. 4.12 Consider each path that starts from the root (there are N such paths) as an array. What our brute-force algorithm is really doing is taking each array and finding all contiguous subsequences that have a particular sum. We're doing this by computing all subarrays and their sums. It might be useful to just focus on this little subproblem. Given an array, how would you find all contiguous subsequences with a particular sum? Again, think about the duplicated work in the brute-force algorithm.

#95. 2.5 Does your algorithm work on linked lists like 9->7->8 and 6->8->5? Double check that.

#96. 4.8 Careful! Does your algorithm handle the case where only one node exists? What will happen? You might need to tweak the return values a bit.

#97. 1.5 What is the relationship between the "insert character" option and the "remove character" option? Do these need to be two separate checks?

#98. 3.4 The major difference between a queue and a stack is the order of elements. A queue removes the oldest item and a stack removes the newest item. How could you remove the oldest item from a stack if you only had access to the newest item?

#99. 4.11 A naive approach that many people come up with is to pick a random number between 1 and 3. If it's 1, return the current node. If it's 2, branch left. If it's 3, branch right. This solution doesn't work. Why not? Is there a way you can adjust it to make it work?

#100. 1.7 Rotating a specific layer would just mean swapping the values in four arrays. If you were asked to swap the values in two arrays, could you do this? Can you then extend it to four arrays?

#101. 2.6 Go back to the previous hint. Remember: There are ways to return multiple values. You can do this with a new class.

#102. 1.8 You probably need some data storage to maintain a list of the rows and columns that need to be zeroed. Can you reduce the additional space usage to `O(1)` by using the matrix itself for data storage?

#103. 4.12 We are looking for subarrays with sum `targetSum`. Observe that we can track in constant time the value of $\texttt{runningSum}_i$, where this is the sum from element 0 through element `i`. For a subarray of element `i` through element `j` to have sum `targetSum`, $\texttt{runningSum}_{i-1}$ + `targetSum` must equal $\texttt{runningSum}_j$ (try drawing a picture of an array or a number line). Given that we can track the `runningSum` as we go, how can we quickly look up the number of indices `i` where the previous equation is true?

#104. 1.9 Think about the earlier hint. Then think about what happens when you concatenate `erbottlewat` to itself. You get `erbottlewaterbottlewat`.

#105. 4.4 You don't need to modify the binary tree class to store the height of the subtree. Can your recursive function compute the height of each subtree while also checking if a node is balanced? Try having the function return multiple values.

#106. 1.4 You do not have to—and should not—generate all permutations. This would be very inefficient.

#107. 4.3 Try modifying a graph search algorithm to track the depth from the root.

#108. 4.12 Try using a hash table that maps from a `runningSum` value to the number of elements with this `runningSum`.

#109. 2.5 For the follow-up question: The issue is that when the linked lists aren't the same length, the head of one linked list might represent the 1000's place while the other represents the 10's place. What if you made them the same length? Is there a way to modify the linked list to do that, without changing the value it represents?

#110. 1.6 Be careful that you aren't repeatedly concatenating strings together. This can be very inefficient.

#111. 2.7 If the two linked lists were the same length, you could traverse forward in each until you found an element in common. Now, how do you adjust this for lists of different lengths?

#112. 4.11 The reason that the earlier solution (picking a random number between 1 and 3) doesn't work is that the probabilities for the nodes won't be equal. For example, the root will be returned with probability $1/3$, even if there are 50+ nodes in the tree. Clearly, not all the nodes have probability $1/3$, so these nodes won't have equal probability. We can resolve this one issue by picking a random number between 1 and `size_of_tree` instead. This only resolves the issue for the root, though. What about the rest of the nodes?

#113. 4.5 Rather than validating the current node's value against `leftTree.max` and `rightTree.min`, can we flip around the logic? Validate the left tree's nodes to ensure that they are smaller than `current.value`.

#114. 3.4 We can remove the oldest item from a stack by repeatedly removing the newest item (inserting those into the temporary stack) until we get down to one element. Then, after we've retrieved the newest item, putting all the elements back. The issue with this is that doing several pops in a row will require `O(N)` work each time. Can we optimize for scenarios where we might do several pops in a row?

#115. 4.12 Once you've solidified the algorithm to find all contiguous subarrays in an array with a given sum, try to apply this to a tree. Remember that as you're traversing and modifying the hash table, you may need to "reverse the damage" to the hash table as you traverse back up.

#116. 4.2 Imagine we had a `createMinimalTree` method that returns a minimal tree for a given array (but for some strange reason doesn't operate on the root of the tree). Could you use this to operate on the root of the tree? Could you write the base case for the function? Great! Then that's basically the entire function.

#117. 1.1 Could a bit vector be useful?

#118. 1.3 You might find you need to know the number of spaces. Can you just count them?

#119. 4.11 The issue with the earlier solution is that there could be more nodes on one side of a node than the other. So, we need to weight the probability of going left and right based on the number of nodes on each side. How does this work, exactly? How can we know the number of nodes?

#120. 2.7 Try using the difference between the lengths of the two linked lists.

#121. 1.4 What characteristics would a string that is a permutation of a palindrome have?

#122. 1.2 Could a hash table be useful?

#123. 4.3 A hash table or array that maps from level number to nodes at that level might also be useful.

#124. 4.4 Actually, you can just have a single `checkHeight` function that does both the height computation and the balance check. An integer return value can be used to indicate both.

#125. 4.7 As a totally different approach: Consider doing a depth-first search starting from an arbitrary node. What is the relationship between this depth-first search and a valid build order?

#126. 2.2 Can you do it iteratively? Imagine if you had two pointers pointing to adjacent nodes and they were moving at the same speed through the linked list. When one hits the end of the linked list, where will the other be?

#127. 4.1 Two well-known algorithms can do this. What are the tradeoffs between them?

#128. 4.5 Think about the `checkBST` function as a recursive function that ensures each node is within an allowable `(min, max)` range. At first, this range is infinite. When we traverse to the left, the min is negative infinity and the max is `root.value`. Can you implement this recursive function and properly adjust these ranges as you traverse the tree?

#129. 2.7 If you move a pointer in the longer linked list forward by the difference in lengths, you can then apply a similar approach to the scenario when the linked lists are equal.

#130. 1.5 Can you do all three checks in a single pass?

#131. 1.2 Two strings that are permutations should have the same characters, but in different orders. Can you make the orders the same?

#132. 1.1 Can you solve it in `O(N log N)` time? What might a solution like that look like?

#133. 4.7 Pick an arbitrary node and do a depth-first search on it. Once we get to the end of a path, we know that this node can be the last one built, since no nodes depend on it. What does this mean about the nodes right before it?

#134. 1.4 Have you tried a hash table? You should be able to get this down to `O(N)` time.

#135. 4.3 You should be able to come up with an algorithm involving both depth-first search and breadth-first search.

#136. 1.4 Can you reduce the space usage by using a bit vector?

11

Hints for Concepts and Algorithms

#137. 5.1 Break this into parts. Focus first on clearing the appropriate bits.

#138. 8.9 Try the Base Case and Build approach.

#139. 6.9 Given a specific door x, on which rounds will it be toggled (open or closed)?

#140. 11.5 What does the interviewer mean by a pen? There are a lot of different types of pens. Make a list of potential questions you would want to ask.

#141. 7.11 This is not as complicated as it sounds. Start by making a list of the key objects in the system, then think about how they interact.

#142. 9.6 First, start with making some assumptions. What do and don't you have to build?

#143. 5.2 To wrap your head around the problem, try thinking about how you'd do it for integers.

#144. 8.6 Try the Base Case and Build approach.

#145. 5.7 Swapping each pair means moving the even bits to the left and the odd bits to the right. Can you break this problem into parts?

#146. 6.10 Solution 1: Start with a simple approach. Can you just divide up the bottles into groups? Remember that you can't re-use a test strip once it is positive, but you can reuse it as long as it's negative.

#147. 5.4 Get Next: Start with a brute force solution for each.

#148. 8.14 Can we just try all possibilities? What would this look like?

#149. 6.5 Play around with the jugs of water, pouring water back and forth, and see if you can measure anything other than 3 quarts or 5 quarts. That's a start.

#150. 8.7 Approach 1: Suppose you had all permutations of abc. How can you use that to get all permutations of abcd?

#151. 5.5 Reverse engineer this, starting from the outermost layer to the innermost layer.

#152. 8.1 Approach this from the top down. What is the very last hop the child made?

#153. 7.1 Note that a "card deck" is very broad. You might want to think about a reasonable scope to the problem.

#154. 6.7 Observe that each family will have exactly one girl.

#155. 8.13 Will sorting the boxes help in any way?

#156. 6.8 This is really an algorithm problem, and you should approach it as such. Come up with a brute force, compute the worst-case number of drops, then try to optimize that.

#157. 6.4 In what cases will they not collide?

#158. 9.6 We've assumed that the rest of the eCommerce system is already handled, and we just need to deal with the analytics part of sales rank. We can get notified somehow when a purchase occurs.

#159. 5.3 Start with a brute force solution. Can you try all possibilities?

#160. 6.7 Think about writing each family as a sequence of Bs and Gs.

#161. 8.8 You could handle this by just checking to see if there are duplicates before printing them (or adding them to a list). You can do this with a hash table. In what case might this be okay? In what case might it not be a very good solution?

#162. 9.7 Will this application be write-heavy or read-heavy?

#163. 6.10 Solution 1: There is a relatively simple approach that works in 28 days, in the worst case. There are better approaches though.

#164. 11.5 Consider the scenario of a pen for children. What does this mean? What are the different use cases?

#165. 9.8 Scope the problem well. What will and won't you tackle as part of this system?

#166. 8.5 Think about multiplying 8 by 9 as counting the number of cells in a matrix with width 8 and height 9.

#167. 5.2 In a number like `.893` (in base 10), what does each digit signify? What then does each digit in `.10010` signify in base 2?

#168. 8.14 We can think about each possibility as each place where we can put parentheses. This means around each operator, such that the expression is split at the operator. What is the base case?

#169. 5.1 To clear the bits, create a "bit mask" that looks like a series of 1s, then 0s, then 1s.

#170. 8.3 Start with a brute force algorithm.

#171. 6.7 You can attempt this mathematically, although the math is pretty difficult. You might find it easier to estimate it up to families of, say, 6 children. This won't give you a good mathematical proof, but it might point you in the right direction of what the answer might be.

#172. 6.9 In which cases would a door be left open at the end of the process?

#173. 5.2 A number such as `.893` (in base 10) indicates $8 * 10^{-1} + 9 * 10^{-2} + 3 * 10^{-3}$. Translate this system into base 2.

#174. 8.9 Suppose we had all valid ways of writing two pairs of parentheses. How could we use this to get all valid ways of writing three pairs?

#175. 5.4 Get Next: Picture a binary number—something with a bunch of 1s and 0s spread out throughout the number. Suppose you flip a 1 to a 0 and a 0 to a 1. In what case will the number get bigger? In what case will it get smaller?

#176. 9.6 Think about what sort of expectations on freshness and accuracy of data is expected. Does the data always need to be 100% up to date? Is the accuracy of some products more important than others?

#177. 10.2 How do you check if two words are anagrams of each other? Think about what the definition of "anagram" is. Explain it in your own words.

#178. 8.1 If we knew the number of paths to each of the steps before step 100, could we compute the number of steps to 100?

#179. 7.8 Should white pieces and black pieces be the same class? What are the pros and cons of this?

#180. 9.7 Observe that there is a lot of data coming in, but people probably aren't reading the data very frequently.

#181. 6.2 Calculate the probability of winning the first game and winning the second game, then compare them.

#182. 10.2 Two words are anagrams if they contain the same characters but in different orders. How can you put characters in order?

#183. 6.10 Solution 2: Why do we have such a time lag between tests and results? There's a reason the question isn't phrased as just "minimize the number of rounds of testing." The time lag is there for a reason.

#184. 9.8 How evenly do you think traffic is distributed? Do all documents get roughly the same age of traffic? Or is it likely there are some very popular documents?

#185. 8.7 Approach 1: The permutations of `abc` represent all ways of ordering `abc`. Now, we want to create all orderings of `abcd`. Take a specific ordering of `abcd`, such as `bdca`. This `bdca` string represents an ordering of `abc`, too: Remove the `d` and you get `bca`. Given the string `bca`, can you create all the "related" orderings that include `d`, too?

#186. 6.1 You can only use the scale once. This means that all, or almost all, of the bottles must be used. They also must be handled in different ways or else you couldn't distinguish between them.

#187. 8.9 We could try generating the solution for three pairs by taking the list of two pairs of parentheses and adding a third pair. We'd have to add the third paren before, around, and after. That is: `()<SOLUTION>`, `(<SOLUTION>)`, `<SOLUTION>()`. Will this work?

#188. 6.7 Logic might be easier than math. Imagine we wrote every birth into a giant string of Bs and Gs. Note that the groupings of families are irrelevant for this problem. What is the probability of the next character added to the string being a B versus a G?

#189. 9.6 Purchases will occur very frequently. You probably want to limit database writes.

#190. 8.8 If you haven't solved 8.7 yet, do that one first.

#191. 6.10 Solution 2: Consider running multiple tests at once.

#192. 7.6 A common trick when solving a jigsaw puzzle is to separate edge and non-edge pieces. How will you represent this in an object-oriented manner?

#193. 10.9 Start with a naive solution. (But hopefully not too naive. You should be able to use the fact that the matrix is sorted.)

#194. 8.13 We can sort the boxes by any dimension in descending order. This will give us a partial order for the boxes, in that boxes later in the array must appear before boxes earlier in the array.

#195. 6.4 The only way they won't collide is if all three are walking in the same direction. What's the probability of all three walking clockwise?

#196. 10.11 Imagine the array were sorted in ascending order. Is there any way you could "fix it" to be sorted into alternating peaks and valleys?

#197. 8.14 The base case is when we have a single value, 1 or 0.

#198. 7.3 Scope the problem first and make a list of your assumptions. It's often okay to make reasonable assumptions, but you need to make them explicit.

#199. 9.7 The system will be write-heavy: Lots of data being imported, but it's rarely being read.

#200. 8.7 Approach 1: Given a string such as `bca`, you can create all permutations of `abcd` that have `{a, b, c}` in the order `bca` by inserting `d` into each possible location: `dbca`, `bdca`, `bcda`, `bcad`. Given all permutations of `abc`, can you then create all permutations of `abcd`?

#201. 6.7 Observe that biology hasn't changed; only the conditions under which a family stops having kids has changed. Each pregnancy has a 50% odds of being a boy and a 50% odds of being a girl.

#202. 5.5 What does it mean if `A & B == 0`?

#203. 8.5 If you wanted to count the cells in an 8x9 matrix, you could count the cells in a 4x9 matrix and then double it.

#204. 8.3 Your brute force algorithm probably ran in `O(N)` time. If you're trying to beat that runtime, what runtime do you think you will get to? What sorts of algorithms have that runtime?

#205. 6.10 Solution 2: Think about trying to figure out the bottle, digit by digit. How can you detect the first digit in the poisoned bottle? What about the second digit? The third digit?

#206. 9.8 How will you handle generating URLs?

#207. 10.6 Think about merge sort versus quick sort. Would one of them work well for this purpose?

#208. 9.6 You also want to limit joins because they can be very expensive.

#209. 8.9 The problem with the solution suggested by the earlier hint is that it might have duplicate values. We could eliminate this by using a hash table.

#210. 11.6 Be careful about your assumptions. Who are the users? Where are they using this? It might seem obvious, but the real answer might be different.

#211. 10.9 We can do a binary search in each row. How long will this take? How can we do better?

#212. 9.7 Think about things like how you're going to get the bank data (will it be pulled or pushed?), what features the system will support, etc.

#213. 7.7 As always, scope the problem. Are "friendships" mutual? Do status messages exist? Do you support group chat?

#214. 8.13 Try to break it down into subproblems.

#215. 5.1 It's easy to create a bit mask of 0s at the beginning or end. But how do you create a bit mask with a bunch of zeroes in the middle? Do it the easy way: Create a bit mask for the left side and then another one for the right side. Then you can merge those.

#216. 7.11 What is the relationship between files and directories?

#217. 8.1 We can compute the number of steps to 100 by the number of steps to 99, 98, and 97. This corresponds to the child hopping 1, 2, or 3 steps at the end. Do we add those or multiply them? That is: Is it `f(100) = f(99) + f(98) + f(97)` *or* `f(100) = f(99) * f(98) * f(97)`?

#218. 6.6 This is a logic problem, not a clever word problem. Use logic/math/algorithms to solve it.

#219. 10.11 Try walking through a sorted array. Can you just swap elements until you have fixed the array?

#220. 11.5 Have you considered both intended uses (writing, etc.) and unintended use? What about safety? You would not want a pen for children to be dangerous.

#221. 6.10 Solution 2: Be very careful about edge cases. What if the third digit in the bottle number matches the first or second digit?

#222. 8.8 Try getting the count of each character. For example, ABCAAC has 3 As, 2 Cs, and 1 B.

#223. 9.6 Don't forget that a product can be listed under multiple categories.

#224. 8.6 You can easily move the smallest disk from one tower to another. It's also pretty easy to move the smallest two disks from one tower to another. Can you move the smallest three disks?

#225. 11.6 In a real interview, you would also want to discuss what sorts of test tools we have available.

#226. 5.3 Flipping a 0 to a 1 can merge two sequences of 1s—but only if the two sequences are separated by only one 0.

#227. 8.5 Think about how you might handle this for odd numbers.

#228. 7.8 What class should maintain the score?

#229. 10.9 If you're considering a particular column, is there a way to quickly eliminate it (in some cases at least)?

#230. 6.10 Solution 2: You can run an additional day of testing to check digit 3 in a different way. But again, be very careful about edge cases here.

#231. 10.11 Note that if you ensure the peaks are in place, the valleys will be, too. Therefore, your iteration to fix the array can skip over every other element.

#232. 9.8 If you generate URLs randomly, do you need to worry about collisions (two documents with the same URL)? If so, how can you handle this?

#233. 6.8 As a first approach, you might try something like binary search. Drop it from the 50th floor, then the 75th, then the 88th, and so on. The problem is that if the first egg drops at the 50th floor, then you'll need to start dropping the second egg starting from the 1st floor and going up. This could take, at worst, 50 drops (the 50th floor drop, the 1st floor drop, the 2nd floor drop, and up through the 49th floor drop). Can you beat this?

#234. 8.5 If there's duplicated work across different recursive calls, can you cache it?

#235. 10.7 Would a bit vector help?

#236. 9.6 Where would it be appropriate to cache data or queue up tasks?

#237. 8.1 We multiply the values when it's "we do this then this." We add them when it's "we do this or this."

#238. 7.6 Think about how you might record the position of a piece when you find it. Should it be stored by row and location?

#239. 6.2 To calculate the probability of winning the second game, start with calculating the probability of making the first hoop, the second hoop, and not the third hoop.

#240. 8.3 Can you solve the problem in O(log N)?

#241. 6.10 Solution 3: Think about each test strip as being a binary indicator for poisoned vs. non-poisoned.

#242. 5.4 Get Next: If you flip a 1 to a 0 and a 0 to a 1, it will get bigger if the 0->1 bit is more significant than the 1->0 bit. How can you use this to create the next biggest number (with the same number of 1s)?

#243. 8.9 Alternatively, we could think about doing this by moving through the string and adding left and right parens at each step. Will this eliminate duplicates? How do we know if we can add a left or right paren?

#244. 9.6 Depending on what assumptions you made, you might even be able to do without a database at all. What would this mean? Would it be a good idea?

#245. 7.7 This is a good problem to think about the major system components or technologies that would be useful.

#246. 8.5 If you're doing 9*7 (both odd numbers), then you could do 4*7 and 5*7.

#247. 9.7 Try to reduce unnecessary database queries. If you don't need to permanently store the data in the database, you might not need it in the database at all.

#248. 5.7 Can you create a number that represents just the even bits? Then can you shift the even bits over by one?

#249. 6.10 Solution 3: If each test strip is a binary indicator, can we map , integer keys to a set of 10 binary indicators such that each key has a unique configuration (mapping)?

#250. 8.6 Think about moving the smallest disk from tower `X=0` to tower `Y=2` using tower `Z=1` as a temporary holding spot as having a solution for `f(1, X=0, Y=2, Z=1)`. Moving the smallest two disks is `f(2, X=0, Y=2, Z=1)`. Given that you have a solution for `f(1, X=0, Y=2, Z=1)` and `f(2, X=0, Y=2, Z=1)`, can you solve `f(3, X=0, Y=2, Z=1)`?

#251. 10.9 Since each column is sorted, you know that the value can't be in this column if it's smaller than the min value in this column. What else does this tell you?

#252. 6.1 What happens if you put one pill from each bottle on the scale? What if you put two pills from each bottle on the scale?

#253. 10.11 Do you necessarily need the arrays to be sorted? Can you do it with an unsorted array?

#254. 10.7 To do it with less memory, can you try multiple passes?

#255. 8.8 To get all permutations with 3 As, 2 Cs, and 1 B, you need to first pick a starting character: A, B, or C. If it's an A, then you need all permutations with 2 As, 2 Cs, and 1 B.

#256. 10.5 Try modifying binary search to handle this.

#257. 11.1 There are two mistakes in this code.

#258. 7.4 Does the parking lot have multiple levels? What "features" does it support? Is it paid? What types of vehicles?

#259. 9.5 You may need to make some assumptions (in part because you don't have an interviewer here). That's okay. Make those assumptions explicit.

#260. 8.13 Think about the first decision you have to make. The first decision is which box will be at the bottom.

#261. 5.5 If `A & B == 0`, then it means that A and B never have a 1 at the same spot. Apply this to the equation in the problem.

#262. 8.1 What is the runtime of this method? Think carefully. Can you optimize it?

#263. 10.2 Can you leverage a standard sorting algorithm?

#264. 6.9 Note: If an integer x is divisible by a, and `b = x / a`, then x is also divisible by b. Does this mean that all numbers have an even number of factors?

#265. 8.9 Adding a left or right paren at each step will eliminate duplicates. Each substring will be unique at each step. Therefore, the total string will be unique.

#266. 10.9 If the value x is smaller than the start of the column, then it also can't be in any columns to the right.

#267. 8.7 Approach 1: You can create all permutations of `abcd` by computing all permutations of `abc` and then inserting d into each possible location within those.

#268. 11.6 What are the different features and uses we would want to test?

#269. 5.2 How would you get the first digit in `.893`? If you multiplied by 10, you'd shift the values over to get `8.93`. What happens if you multiply by 2?

#270. 9.2 To find the connection between two nodes, would it be better to do a breadth-first search or depth-first search? Why?

#271. 7.7 How will you know if a user signs offline?

#272. 8.6 Observe that it doesn't really matter which tower is the source, destination, or buffer. You can do `f(3, X=0, Y=2, Z=1)` by first doing `f(2, X=0, Y=1, Z=2)` (moving two disks from tower 0 to tower 1, using tower 2 as a buffer), then moving disk 3 from tower 0 to tower 2, then doing `f(2, X=1, Y=2, Z=0)` (moving two disks from tower 1 to tower 2, using tower 0 as a buffer). How does this process repeat?

#273. 8.4 How can you build all subsets of `{a, b, c}` from the subsets of `{a, b}`?

#274. 9.5 Think about how you could design this for a single machine. Would you want a hash table? How would that work?

#275. 7.1 How, if at all, will you handle aces?

#276. 9.7 As much work as possible should be done asynchronously.

#277. 10.11 Suppose you had a sequence of three elements (`{0, 1, 2}`, in any order. Write out all possible sequences for those elements and how you can fix them to make `1` the peak.

#278. 8.7 Approach 2: If you had all permutations of two-character substrings, could you generate all permutations of three-character substrings?

#279. 10.9 Think about the previous hint in the context of rows.

#280. 8.5 Alternatively, if you're doing 9 * 7, you could do 4*7, double that, and then add 7.

#281. 10.7 Try using one pass to get it down to a range of values, and then a second pass to find a specific value.

#282. 6.6 Suppose there were exactly one blue-eyed person. What would that person see? When would they leave?

#283. 7.6 Which will be the easiest pieces to match first? Can you start with those? Which will be the next easiest, once you've nailed those down?

#284. 6.2 If two events are mutually exclusive (they can never occur simultaneously), you can add their probabilities together. Can you find a set of mutually exclusive events that represent making two out of three hoops?

#285. 9.2 A breadth-first search is probably better. A depth-first search can wind up going on a long path, even though the shortest path is actually very short. Is there a modification to a breadth-first search that might be even faster?

#286. 8.3 Binary search has a runtime of `O(log N)`. Can you apply a form of binary search to the problem?

#287. 7.12 In order to handle collisions, the hash table should be an array of linked lists.

#288. 10.9 What would happen if we tried to keep track of this using an array? What are the pros and cons of this?

#289. 10.8 Can you use a bit vector?

#290. 8.4 Anything that is a subset of `{a, b}` is also a subset of `{a, b, c}`. Which sets are subsets of `{a, b, c}` but not `{a, b}`?

#291. 10.9 Can we use the previous hints to move up, down, left, and right around the rows and columns?

#292. 10.11 Revisit the set of sequences for `{0, 1, 2}` that you just wrote out. Imagine there are elements before the leftmost element. Are you sure that the way you swap the elements won't invalidate the previous part of the array?

#293. 9.5 Can you combine a hash table and a linked list to get the best of both worlds?

#294. 6.8 It's actually better for the first drop to be a bit lower. For example, you could drop at the 10th floor, then the 20th floor, then the 30th floor, and so on. The worst case here will be 19 drops (10, 20, ..., 100, 91, 92, ..., 99). Can you beat that? Try not randomly guessing at different solutions. Rather, think deeper. How is the worst case defined? How does the number of drops of each egg factor into that?

#295. 8.9 We can ensure that this string is valid by counting the number of left and right parens. It is always valid to add a left paren, up until the total number of pairs of parens. We can add a right paren as long as `count(left parens) <= count(right parens)`.

#296. 6.4 You can think about this either as the probability(3 ants walking clockwise) + probability(3 ants walking counter-clockwise). Or, you can think about it as: The first ant picks a direction. What's the probability of the other ants picking the same direction?

#297. 5.2 Think about what happens for values that can't be represented accurately in binary.

#298. 10.3 Can you modify binary search for this purpose?

#299. 11.1 What will happen to the `unsigned int`?

#300. 8.11 Try breaking it down into subproblems. If you were making change, what is the first choice you would make?

#301. 10.10 The problem with using an array is that it will be slow to insert a number. What other data structures could we use?

#302. 5.5 If `(n & (n-1)) == 0`, then this means that n and `n - 1` never have a 1 in the same spot. Why would that happen?

#303. 10.9 Another way to think about this is that if you drew a rectangle around a cell extending to the bottom, right coordinate of the matrix, the cell would be bigger than all the items in this square.

#304. 9.2 Is there any way to search from both the source and destination? For what reason or in what case might this be faster?

#305. 8.14 If your code looks really lengthy, with a lot of if's (for each possible operator, "target" boolean result, and left/right side), think about the relationship between the different parts. Try to simplify your code. It should not need a ton of complicated if-statements. For example, consider expressions of the form <LEFT>OR<RIGHT> versus <LEFT>AND<RIGHT>. Both may need to know the number of ways that the <LEFT> evaluates to true. See what code you can reuse.

#306. 6.9 The number 3 has an even number of factors (1 and 3). The number 12 has an even number of factors (1, 2, 3, 4, 6, 12). What numbers do not? What does this tell you about the doors?

#307. 7.12 Think carefully about what information the linked list node needs to contain.

#308. 8.12 We know that each row must have a queen. Can you try all possibilities?

#309. 8.7 Approach 2: To generate a permutation of `abcd`, you need to pick an initial character. It can be a, b, c, or d. You can then permute the remaining characters. How can you use this approach to generate all permutations of the full string?

#310. 10.3 What is the runtime of your algorithm? What will happen if the array has duplicates?

#311. 9.5 How would you scale this to a larger system?

#312. 5.4 Get Next: Can you flip a 0 to a 1 to create the next biggest number?

#313. 11.4 Think about what load testing is designed to test. What are the factors in the load of a webpage? What criteria would be used to judge if a webpage performs satisfactorily under heavy load?

#314. 5.3 Each sequence can be lengthened by merging it with an adjacent sequence (if any) or just flipping the immediate neighboring zero. You just need to find the best choice.

#315. 10.8 Consider implementing your own bit vector class. It's a good exercise and an important part of this problem.

#316. 10.11 You should be able to design an O(n) algorithm.

#317. 10.9 A cell will be larger than all the items below it and to the right. It will be smaller than all cells above it and to the left. If we wanted to eliminate the most elements first, which element should we compare the value x to?

#318. 8.6 If you're having trouble with recursion, then try trusting the recursive process more. Once you've figured out how to move the top two disks from tower 0 to tower 2, trust that you have this working. When you need to move three disks, trust that you can move two disks from one tower to another. Now, two disks have been moved. What do you do about the third?

#319. 6.1 Imagine there were just three bottles and one had heavier pills. Suppose you put different numbers of pills from each bottle on the scale (for example, bottle 1 has 5 pills, bottle 2 has 2 pills, and bottle 3 has 9 pills). What would the scale show?

#320. 10.4 Think about how binary search works. What will be the issue with just implementing binary search?

#321. 9.2 Discuss how you might implement these algorithms and this system in the real world. What sort of optimizations might you make?

#322. 8.13 Once we pick the box on the bottom, we need to pick the second box. Then the third box.

#323. 6.2 The probability of making two out of three shots is probability(make shot 1, make shot 2, miss shot 3) + probability(make shot 1, miss shot 2, make shot 3) + probability(miss shot 1, make shot 2, make shot 3) + probability(make shot 1, make shot 2, make shot 3).

#324. 8.11 If you were making change, the first choice you might make is how many quarters you need to use.

#325. 11.2 Think about issues both within the program and outside of the program (the rest of the system).

#326. 9.4 Estimate how much space is needed for this.

#327. 8.14 Look at your recursion. Do you have repeated calls anywhere? Can you memoize it?

#328. 5.7 The value 1010 in binary is 10 in decimal or 0xA in hex. What will a sequence of 101010... be in hex? That is, how do you represent an alternating sequence of 1s and 0s with 1s in the odd places? How do you do this for the reverse (1s in the even spots)?

#329. 11.3 Consider both extreme cases and more general cases.

#330. 10.9 If we compare x to the center element in the matrix, we can eliminate roughly one quarter of the elements in the matrix.

#331. 8.2 For the robot to reach the last cell, it must find a path to the second-to-last cells. For it to find a path to the second-to-last cells, it must find a path to the third-to-last cells.

#332. 10.1 Try moving from the end of the array to the beginning.

#333. 6.8 If we drop Egg 1 at fixed intervals (e.g., every 10 floors), then the worst case is the worst case for Egg 1 + the worst case for Egg 2. The problem with our earlier solutions is that as Egg 1 does more work, Egg 2 doesn't do any less work. Ideally, we'd like to balance this a bit. As Egg 1 does more work (has survived more drops), Egg 2 should have less work to do. What might this mean?

#334. 9.3 Think about how infinite loops might occur.

#335. 8.7 Approach 2: To generate all permutations of `abcd`, pick each character (`a`, `b`, `c`, or `d`) as a starting character. Permute the remaining characters and prepend the starting character. How do you permute the remaining characters? With a recursive process that follows the same logic.

#336. 5.6 How would you figure out how many bits are different between two numbers?

#337. 10.4 Binary search requires comparing an element to the midpoint. Getting the midpoint requires knowing the length. We don't know the length. Can we find it?

#338. 8.4 Subsets that contain `c` will be subsets `{a, b, c}` but not `{a, b}`. Can you build these subsets from the subsets of `{a, b}`?

#339. 5.4 Get Next: Flipping a 0 to a 1 will create a bigger number. The farther right the index is the smaller the bigger number is. If we have a number like 1001, we want to flip the rightmost 0 (to create 1011). But if we have a number like 1010, we should not flip the rightmost 1.

#340. 8.3 Given a specific index and value, can you identify if the magic index would be before or after it?

#341. 6.6 Now suppose there were two blue-eyed people. What would they see? What would they know? When would they leave? Remember your answer from the prior hint. Assume they know the answer to the earlier hint.

#342. 10.2 Do you even need to truly "sort"? Or is just reorganizing the list sufficient?

#343. 8.11 Once you've decided to use two quarters to make change for 98 cents, you now need to figure out how many ways to make change for 48 cents using nickels, dimes, and pennies.

#344. 7.5 Think about all the different functionality a system to read books online would have to support. You don't have to do everything, but you should think about making your assumptions explicit.

#345. 11.4 Could you build your own? What might that look like?

#346. 5.5 What is the relationship between how n looks and how `n - 1` looks? Walk through a binary subtraction.

#347. 9.4 Will you need multiple passes? Multiple machines?

#348. 10.4 We can find the length by using an exponential backoff. First check index 2, then 4, then 8, then 16, and so on. What will be the runtime of this algorithm?

#349. 11.6 What can we automate?

#350. 8.12 Each row must have a queen. Start with the last row. There are eight different columns on which you can put a queen. Can you try each of these?

#351. 7.10 Should number cells, blank cells, and bomb cells be separate classes?

#352. 5.3 Try to do it in linear time, a single pass, and `O(1)` space.

#353. 9.3 How would you detect the same page? What does this mean?

#354. 8.4 You can build the remaining subsets by adding c to all the subsets of `{a, b}`.

#355. 5.7 Try masks `0xaaaaaaaa` and `0x55555555` to select the even and odd bits. Then try shifting the even and odd bits around to create the right number.

#356. 8.7 Approach 2: You can implement this approach by having the recursive function pass back the list of the strings, and then you prepend the starting character to it. Or, you can push down a prefix to the recursive calls.

#357. 6.8 Try dropping Egg 1 at bigger intervals at the beginning and then at smaller and smaller intervals. The idea is to keep the sum of Egg 1 and Egg 2's drops as constant as possible. For each additional drop that Egg 1 takes, Egg 2 takes one fewer drop. What is the right interval?

#358. 5.4 Get Next: We should flip the rightmost non-trailing 0. The number 1010 would become 1110. Once we've done that, we need to flip a 1 to a 0 to make the number as small as possible, but bigger than the original number (1010). What do we do? How can we shrink the number?

#359. 8.1 Try memoization as a way to optimize an inefficient recursive program.

#360. 8.2 Simplify this problem a bit by first figuring out if there's a path. Then, modify your algorithm to track the path.

#361. 7.10 What is the algorithm to place the bombs around the board?

#362. 11.1 Look at the parameters for `printf`.

#363. 7.2 Before coding, make a list of the objects you need and walk through the common algorithms. Picture the code. Do you have everything you need?

#364. 8.10 Think about this as a graph.

#365. 9.3 How do you define if two pages are the same? Is it the URLs? Is it the content? Both of these can be flawed. Why?

#366. 5.8 First try the naive approach. Can you set a particular "pixel"?

#367. 6.3 Picture a domino laying down on the board. How many black squares does it cover? How many white squares?

#368. 8.13 Once you have a basic recursive algorithm implemented, think about if you can optimize it. Are there any repeated subproblems?

#369. 5.6 Think about what an XOR indicates. If you do a XOR b, where does the result have 1s? Where does it have 0s?

#370. 6.6 Build up from this. What if there were three blue-eyed people? What if there were four blue-eyed people?

#371. 8.12 Break this down into smaller subproblems. The queen at row 8 must be at column 1, 2, 3, 4, 5, 6, 7, or 8. Can you print all ways of placing eight queens where a queen is at row 8 and column 3? You then need to check all the ways of placing a queen on row 7.

#372. 5.5 When you do a binary subtraction, you flip the rightmost 0s to a 1, stopping when you get to a 1 (which is also flipped). Everything (all the 1s and 0s) on the left will stay put.

#373. 8.4 You can also do this by mapping each subset to a binary number. The `i`th bit could represent a "boolean" flag for whether an element is in the set.

#374. 6.8 Let X be the first drop of Egg 1. This means that Egg 2 would do X - 1 drops if Egg 1 broke. We want to try to keep the sum of Egg 1 and Egg 2's drops as constant as possible. If Egg 1 breaks on the second drop, then we want Egg 2 to do X - 2 drops. If Egg 1 breaks on the third drop, then we want Egg 2 to do X - 3 drops. This keeps the sum of Egg 1 and Egg 2 fairly constant. What is X?

#375. 5.4 Get Next: We can shrink the number by moving all the 1s to the right of the flipped bit as far right as possible (removing a 1 in the process).

#376. 10.10 Would it work well to use a binary search tree?

#377. 7.10 To place the bombs randomly on the board: Think about the algorithm to shuffle a deck of cards. Can you apply a similar technique?

#378. 8.13 Alternatively, we can think about the repeated choices as: Does the first box go on the stack? Does the second box go on the stack? And so on.

#379. 6.5 If you fill the 5-quart jug and then use it to fill the 3-quart jug, you'll have two quarts left in the 5-quart jug. You can either keep those two quarts where they are, or you can dump the contents of the smaller jug and pour the two quarts in there.

#380. 8.11 Analyze your algorithm. Is there any repeated work? Can you optimize this?

#381. 5.8 When you're drawing a long line, you'll have entire bytes that will become a sequence of 1s. Can you set this all at once?

#382. 8.10 You can implement this using depth-first search (or breadth-first search). Each adjacent pixel of the "right" color is a connected edge.

#383. 5.5 Picture n and `n-1`. To subtract 1 from n, you flipped the rightmost 1 to a 0 and all the 0s on its right to 1s. If `n & n-1 == 0`, then there are no 1s to the left of the first 1. What does that mean about n?

#384. 5.8 What about the start and end of the line? Do you need to set those pixels individually, or can you set them all at once?

#385. 9.1 Think about this as a real-world application. What are the different factors you would need to consider?

#386. 7.10 How do you count the number of bombs neighboring a cell? Will you iterate through all cells?

#387. 6.1 You should be able to have an equation that tells you the heavy bottle based on the weight.

#388. 8.2 Think again about the efficiency of your algorithm. Can you optimize it?

#389. 7.9 The `rotate()` method should be able to run in `O(1)` time.

#390. 5.4 Get Previous: Once you've solved Get Next, try to invert the logic for Get Previous.

#391. 5.8 Does your code handle the case when x1 and x2 are in the same byte?

#392. 10.10 Consider a binary search tree where each node stores some additional data.

#393. 11.6 Have you thought about security and reliability?

#394. 8.11 Try using memoization.

#395. 6.8 I got 14 drops in the worst case. What did you get?

#396. 9.1 There's no one right answer here. Discuss several different technical implementations.

#397. 6.3 How many black squares are there on the board? How many white squares?

#398. 5.5 We know that n must have only one 1 if `n & (n-1) == 0`. What sorts of numbers have only one 1?

#399. 7.10 When you click on a blank cell, what is the algorithm to expand the neighboring cells?

#400. 6.5 Once you've developed a way to solve this problem, think about it more broadly. If you are given a jug of size X and another jug of size Y, can you always use it to measure Z?

#401. 11.3 Is it possible to test everything? How will you prioritize testing?

III

Hints for Knowledge-Based Questions

#402. 12.9 Focus on the concept firsts, then worry about the exact implementation. How should `SmartPointer` look?

#403. 15.2 A context switch is the time spent switching between two processes. This happens when you bring one process into execution and swap out the existing process.

#404. 13.1 Think about who can access private methods.

#405. 15.1 How do these differ in terms of memory?

#406. 12.11 Recall that a two dimensional array is essentially an array of arrays.

#407. 15.2 Ideally, we would like to record the timestamp when one process "stops" and the timestamp when another process "starts." But how do we know when this swapping will occur?

#408. 14.1 A GROUP BY clause might be useful.

#409. 13.2 When does a finally block get executed? Are there any cases where it won't get executed?

#410. 12.2 Can we do this in place?

#411. 14.2 It might be helpful to break the approach into two pieces. The first piece is to get each building ID and the number of open requests. Then, we can get the building names.

#412. 13.3 Consider that some of these might have different meanings depending on where they are applied.

#413. 12.10 Typically, `malloc` will just give us an arbitrary block of memory. If we can't override this behavior, can we work with it to do what we need?

#414. 15.7 First implement the single-threaded FizzBuzz problem.

#415. 15.2 Try setting up two processes and have them pass a small amount of data back and forth. This will encourage the system to stop one process and bring the other one in.

#416. 13.4 The purpose of these might be somewhat similar, but how does the implementation differ?

#417. 15.5 How can we ensure that `first()` has terminated before calling `second()`?

#418. 12.11 One approach is to call `malloc` for each array. How would we free the memory here?

#419. 15.3 A deadlock can happen when there's a "cycle" in the order of who is waiting for whom. How can we break or prevent this cycle?

#420. 13.5 Think about the underlying data structure.

#421. 12.7 Think about why we use virtual methods.

#422. 15.4 If every thread had to declare upfront what processes it might need, could we detect possible deadlocks in advance?

#423. 12.3 What is the underlying data structure behind each? What are the implications of this?

#424. 13.5 `HashMap` uses an array of linked lists. `TreeMap` uses a red-black tree. `LinkedHashMap` uses doubly-linked buckets. What is the implication of this?

#425. 13.4 Consider the usage of primitive types. How else might they differ in terms of how you can use the types?

#426. 12.11 Can we allocate this instead as a contiguous block of memory?

#427. 12.8 This data structure can be pictured as a binary tree, but it's not necessarily. What if there's a loop in the structure?

#428. 14.7 You probably need a list of students, their courses, and another table building a relationship between students and courses. Note that this is a many-to-many relationship.

#429. 15.6 The keyword `synchronized` ensures that two threads cannot execute synchronized methods on the same instance at the same time.

#430. 13.5 Consider how they might differ in terms of the order of iteration through the keys. Why might you want one option instead of the others?

#431. 14.3 First try to get a list of the IDs (just the IDs) of all the relevant apartments.

#432. 12.10 Imagine we have a sequential set of integers (3, 4, 5, ...). How big does this set need to be to ensure that one of the numbers is divisible by 16?

#433. 15.5 Why would using `boolean` flags to do this be a bad idea?

#434. 15.4 Think about the order of requests as a graph. What does a deadlock look like within this graph?

#435. 13.6 Object reflection allows you to get information about methods and fields in an object. Why might this be useful?

#436. 14.6 Be particularly careful about which relationships are one-to-one vs. one-to-many vs. many-to-many.

#437. 15.3 One idea is to just not let a philosopher hold onto a chopstick if he can't get the other one.

#438. 12.9 Think about tracking the number of references. What will this tell us?

#439. 15.7 Don't try to do anything fancy on the single-threaded problem. Just get something that is simple and easily readable.

#440. 12.10 How will we free the memory?

#441. 15.2 It's okay if your solution isn't totally perfect. That might not be possible. Discuss the tradeoffs of your approach.

#442. 14.7 Think carefully about how you handle ties when selecting the top 10%.

#443. 13.8 A naive approach is to pick a random subset size z and then iterate through the elements, putting it in the set with probability `z/list_size`. Why would this not work?

#444. 14.5 Denormalization means adding redundant data to a table. It's typically used in very large systems. Why might this be useful?

#445. 12.5 A shallow copy copies just the initial data structure. A deep copy does this, and also copies any underlying data. Given this, why might you use one versus the other?

#446. 15.5 Would semaphores be useful here?

#447. 15.7 Outline the structure for the threads without worrying about synchronizing anything.

#448. 13.7 Consider how you'd implement this first without lambda expressions.

#449. 12.1 If we already had the number of lines in the file, how would we do this?

#450. 13.8 Pick the list of all the subsets of an n-element set. For any given item x, half of the subsets contain x and half do not.

#451. 14.4 Describe INNER JOINs and OUTER JOINs. OUTER JOINs can have multiple types: left, right, and full.

#452. 12.2 Be careful about the null character.

#453. 12.9 What are all the different methods/operators we might want to override?

#454. 13.5 What would the runtime of the common operations be?

#455. 14.5 Think about the cost of joins on a large system.

#456. 12.6 The keyword `volatile` signals that a variable might be changed from outside of the program, such as by another process. Why might this be necessary?

#457. 13.8 Do not pick the length of the subset in advance. You don't need to. Instead, think about this as picking whether each element will be put into the set.

#458. 15.7 Once you get the structure of each thread done, think about what you need to synchronize.

#459. 12.1 Suppose we didn't have the number of lines in the file. Is there a way we could do this without first counting the number of lines?

#460. 12.7 What would happen if the destructor were not virtual?

#461. 13.7 Break this up into two parts: filtering the countries and then getting a sum.

#462. 12.8 Consider using a hash table.

#463. 12.4 You should discuss vtables here.

#464. 13.7 Can you do this without a `filter` operation?

IV

Hints for Additional Review Problems

#465. 16.3 Think about what you're going to design for.

#466. 16.12 Consider a recursive or tree-like approach.

#467. 17.1 Walk through binary addition by hand (slowly!) and try to really understand what is happening.

#468. 16.13 Draw a square and a bunch of lines that cut it in half. Where are those lines located?

#469. 17.24 Start with a brute force solution.

#470. 17.14 There are actually several approaches. Brainstorm these. It's okay to start off with a naive approach.

#471. 16.20 Consider recursion.

#472. 16.3 Will all lines intercept? What determines if two lines intercept?

#473. 16.7 Let k be 1 if a > b and 0 otherwise. If you were given k, could you return the max (without a comparison or if-else logic)?

#474. 16.22 The tricky bit is handling an infinite grid. What are your options?

#475. 17.15 Try simplifying this problem: What if you just needed to know the longest word made up of two other words in the list?

#476. 16.10 Solution 1: Can you count the number of people alive in each year?

#477. 17.25 Start by grouping the dictionary by the word lengths, since you know each column has to be the same length and each row has to be the same length.

#478. 17.7 Discuss the naive approach: merging names together when they are synonyms. How would you identify transitive relationships? A == B, A == C, and C == D implies A == D == B == C.

#479. 16.13 Any straight line that cuts a square in half goes through the center of the square. How then can you find a line that cuts two squares in half?

#480. 17.17 Start with a brute force solution. What is the runtime?

#481. 16.22 Option #1: Do you actually need an infinite grid? Read the problem again. Do you know the max size of the grid?

#482. 16.16 Would it help to know the longest sorted sequences at the beginning and end?

#483. 17.2 Try approaching this problem recursively.

#484. 17.26 Solution 1: Start with just a simple algorithm comparing all documents to all other documents. How would you compute the similarity of two documents as fast as possible?

#485. 17.5 It doesn't really matter which letter or number it is. You can simplify this problem to just having an array of As and Bs. You would then be looking for the longest subarray with an equal number of As and Bs.

#486. 17.11 Consider first the algorithm for finding the closest distance if you will run the algorithm only once. You should be able to do this in O(N) time, where N is the number of words in the document.

#487. 16.20 Can you recursively try all possibilities?

#488. 17.9 Be clear about what this problem is asking for. It's asking for the kth smallest number in the form $3^a * 5^b * 7^c$.

#489. 16.2 Think about what the best conceivable runtime is for this problem. If your solution matches the best conceivable runtime, then you probably can't do any better.

#490. 16.10 Solution 1: Try using a hash table, or an array that maps from a birth year to how many people are alive in that year.

#491. 16.14 Sometimes, a brute force is a pretty good solution. Can you try all possible lines?

#492. 16.1 Try picturing the two numbers, a and b, on a number line.

#493. 17.7 The core part of the problem is to group names into the various spellings. From there, figuring out the frequencies is relatively easy.

#494. 17.3 If you haven't already, solve 17.2 on page 186.

#495. 17.16 There are recursive and iterative solutions to this problem, but it's probably easier to start with the recursive solution.

#496. 17.13 Try a recursive approach.

#497. 16.3 Infinite lines will almost always intersect—unless they're parallel. Parallel lines might still "intercept"—if they're the same lines. What does this mean for line segments?

#498. 17.26 Solution 1: To compute the similarity of two documents, try reorganizing the data in some way. Sorting? Using another data structure?

#499. 17.15 If we wanted to know just the longest word made up of other words in the list, then we could iterate over all words, from longest to shortest, checking if each could be made up of other words. To check this, we split the string in all possible locations.

#500. 17.25 Can you find a word rectangle of a specific length and width? What if you just tried all options?

#501. 17.11 Adapt your algorithm for one execution of the algorithm for repeated executions. What is the slow part? Can you optimize it?

#502. 16.8 Try thinking about the number in terms of chunks of three digits.

#503. 17.19 Start with the first part: Finding the missing number if only one number is missing.

#504. 17.16 Recursive solution: You have two choices at each appointment (take the appointment or reject the appointment). As a brute force approach, you can recurse through all possibilities. Note, though, that if you take request `i`, your recursive algorithm should skip request `i + 1`.

#505. 16.23 Be very careful that your solution actually returns each value from 0 through 6 with equal probability.

#506. 17.22 Start with a brute force, recursive solution. Just create all words that are one edit away, check if they are in the dictionary, and then attempt that path.

#507. 16.10 Solution 2: What if you sorted the years? What would you sort by?

#508. 17.9 What does a brute force solution to get the kth smallest value for $3^a * 5^b * 7^c$ look like?

#509. 17.12 Try a recursive approach.

#510. 17.26 Solution 1: You should be able to get an O(A+B) algorithm to compute the similarity of two documents.

#511. 17.24 The brute force solution requires us to continuously compute the sums of each matrix. Can we optimize this?

#512. 17.7 One thing to try is maintaining a mapping of each name to its "true" spelling. You would also need to map from a true spelling to all the synonyms. Sometimes, you might need to merge two different groups of names. Play around with this algorithm to see if you can get it to work. Then see if you can simplify/optimize it.

#513. 16.7 If k were 1 when `a > b` and 0 otherwise, then you could return `a*k + b*(not k)`. But how do you create k?

#514. 16.10 Solution 2: Do you actually need to match the birth years and death years? Does it matter when a specific person died, or do you just need a list of the years of deaths?

#515. 17.5 Start with a brute force solution.

#516. 17.16 Recursive solution: You can optimize this approach through memoization. What is the runtime of this approach?

#517. 16.3 How can we find the intersection between two lines? If two line segments intercept, then this must be at the same point as their "infinite" extensions. Is this intersection point within both lines?

#518. 17.26 Solution 1: What is the relationship between the intersection and the union? Can you compute one from the other?

#519. 17.20 Recall that the median means the number for which half the numbers are larger and half the numbers are smaller.

#520. 16.14 You can't truly try all possible lines in the world—that's infinite. But you know that a "best" line must intersect at least two points. Can you connect each pair of points? Can you check if each line is indeed the best line?

#521. 16.26 Can we just process the expression from left to right? Why might this fail?

#522. 17.10 Start with a brute force solution. Can you just check each value to see if it's the majority element?

#523. 16.10 Solution 2: Observe that people are "fungible." It doesn't matter who was born and when they died. All you need is a list of birth years and death years. This might make the question of how you sort the list of people easier.

#524. 16.25 First scope the problem. What are the features you would want?

#525. 17.24 Can you do any sort of precomputation to make computing the sum of a submatrix `O(1)`?

#526. 17.16 Recursive solution: The runtime of your memoization approach should be `O(N)`, with `O(N)` space.

#527. 16.3 Think carefully about how to handle the case of line segments that have the same slope and y-intercept.

#528. 16.13 To cut two squares in half, a line must go through the middle of both squares.

#529. 16.14 You should be able to get to an $O(N^2)$ solution.

#530. 17.14 Consider thinking about reorganizing the data in some way or using additional data structures.

#531. 16.17 Picture the array as alternating sequences of positive and negative numbers. Observe that we would never include just part of a positive sequence or part of a negative sequence.

#532. 16.10 Solution 2: Try creating a sorted list of births and a sorted list of deaths. Can you iterate through both, tracking the number of people alive at any one time?

#533. 16.22 Option #2: Think about how an `ArrayList` works. Can you use an `ArrayList` for this?

#534. 17.26 Solution 1: To understand the relationship between the union and the intersection of two sets, consider a Venn diagram (a diagram where one circle overlaps another circle).

#535. 17.22 Once you have a brute force solution, try to find a faster way of getting all valid words that are one edit away. You don't want to create all strings that are one edit away when the vast majority of them are not valid dictionary words.

#536. 16.2 Can you use a hash table to optimize the repeated case?

#537. 17.7 An easier way of taking the above approach is to have each name map to a list of alternate spellings. What should happen when a name in one group is set equal to a name in another group?

#538. 17.11 You could build a lookup table that maps from a word to a list of the locations where each word appears. How then could you find the closest two locations?

#539. 17.24 What if you precomputed the sum of the submatrix starting at the top left corner and continuing to each cell? How long would it take you to compute this? If you did this, could you then get the sum of an arbitrary submatrix in `O(1)` time?

#540. 16.22 Option #2: It's not impossible to use an `ArrayList`, but it would be tedious. Perhaps it would be easier to build your own, but specialized for matrices.

#541. 16.10 Solution 3: Each birth adds one person and each death removes a person. Try writing an example of a list of people (with birth and death years) and then re-formatting this into a list of each year and a +1 for a birth and a -1 for a death.

#542. 17.16 Iterative solution: Take the recursive solution and investigate it more. Can you implement a similar strategy iteratively?

#543. 17.15 Extend the earlier idea to multiple words. Can we just break each word up in all possible ways?

#544. 17.1 You can think about binary addition as iterating through the number, bit by bit, adding two bits, and then carrying over the one if necessary. You could also think about it as grouping the operations. What if you first added each of the bits (without carrying any overflow)? After that, you can handle the overflow.

#545. 16.21 Do some math here or play around with some examples. What does this pair need to look like? What can you say about their values?

#546. 17.20 Note that you have to store all the elements you've seen. Even the smallest of the first 100 elements could become the median. You can't just toss very low or very high elements.

#547. 17.26 Solution 2: It's tempting to try to think of minor optimizations—for example, keeping track of the min and max elements in each array. You could then figure out quickly, in specific cases, if two arrays don't overlap. The problem with that (and other optimizations along these lines) is that you still need to compare all documents to all other documents. It doesn't leverage the fact that the similarity is sparse. Given that we have a lot of documents, we really need to not compare all documents to all other documents (even if that comparison is very fast). All such solutions will be $O(D^2)$, where D is the number of documents. We shouldn't compare all documents to all other documents.

#548. 16.24 Start with a brute force solution. What is the runtime? What is the best conceivable runtime for this problem?

#549. 16.10 Solution 3: What if you created an array of years and how the population changed in each year? Could you then find the year with the highest population?

#550. 17.9 In looking for the kth smallest value of $3^a * 5^b * 7^c$, we know that a, b, and c will be less than or equal to k. Can you generate all such numbers?

#551. 16.17 Observe that if you have a sequence of values which have a negative sum, those will never start or end a sequence. (They could be present in a sequence if they connected two other sequences.)

#552. 17.14 Can you sort the numbers?

#553. 16.16 We can think about the array as divided into three subarrays: `LEFT`, `MIDDLE`, `RIGHT`. `LEFT` and `RIGHT` are both sorted. The `MIDDLE` elements are in an arbitrary order. We need to expand `MIDDLE` until we could sort those elements and then have the entire array sorted.

#554. 17.16 Iterative solution: It's probably easiest to start with the end of the array and work backwards.

#555. 17.26 Solution 2: If we can't compare all documents to all other documents, then we need to dive down and start looking at things at the element level. Consider a naive solution and see if you can extend that to multiple documents.

#556. 17.22 To quickly get the valid words that are one edit away, try to group the words in the dictionary in a useful way. Observe that all words in the form `b_ll` (such as `bill`, `ball`, `bell`, and `bull`) will be one edit away. However, those aren't the only words that are one edit away from `bill`.

#557. 16.21 When you move a value a from array A to array B, then A's sum decreases by a and B's sum increases by a. What happens when you swap two values? What would be needed to swap two values and get the same sum?

#558. 17.11 If you had a list of the occurrences of each word, then you are really looking for a pair of values within two arrays (one value for each array) with the smallest difference. This could be a fairly similar algorithm to your initial algorithm.

#559. 16.22 Option #2: One approach is to just double the size of the array when the ant wanders to an edge. How will you handle the ant wandering into negative coordinates, though? Arrays can't have negative indices.

#560. 16.13 Given a line (slope and y-intercept), can you find where it intersects another line?

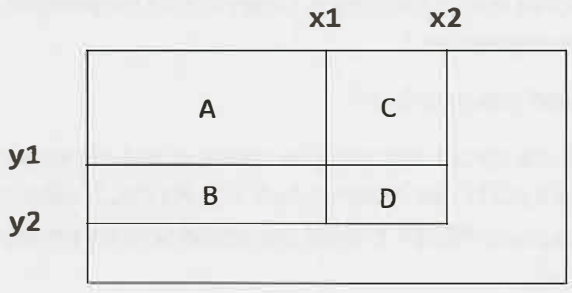
#561. 17.26 Solution 2: One way to think about this is that we need to be able to very quickly pull a list of all documents with some similarity to a specific document. (Again, we should not do this by saying "look at all documents and quickly eliminate the dissimilar documents." That will be at least $O(D^2)$.)

#562. 17.16 Iterative solution: Observe that you would never skip three appointments in a row. Why would you? You would always be able to take the middle booking.

#563. 16.14 Have you tried using a hash table?

#564. 16.21 If you swap two values, a and b, then the sum of A becomes `sumA - a + b` and the sum of B becomes `sumB - b + a`. These sums need to be equal.

#565. 17.24 If you can precompute the sum from the top left corner to each cell, you can use this to compute the sum of an arbitrary submatrix in $O(1)$ time. Picture a particular submatrix. The full, precomputed sum will include this submatrix, an array immediately above it (C), and array to the left (B), and an area to the top and left (A). How can you compute the sum of just D?



#566. 17.10 Consider the brute force solution. We pick an element and then validate if it's the majority element by counting the number of matching and non-matching elements. Suppose, for the first element, the first few checks reveal seven non-matching elements and three matching elements. Is it necessary to keep checking this element?

#567. 16.17 Start from the beginning of the array. As that subsequence gets larger, it stays as the best subsequence. Once it becomes negative, though, it's useless.

#568. 17.16 Iterative solution: If you take appointment `i`, you will never take appointment `i + 1`, but you will always take appointment `i + 2` or `i + 3`.

#569. 17.26 Solution 2: Building off the earlier hint, we can ask what defines the list of documents with some similarity to a document like {13, 16, 21, 3}. What attributes does that list have? How would we gather all documents like that?

#570. 16.22 Option #2: Observe that nothing in the problem stipulates that the label for the coordinates must remain the same. Can you move the ant and all cells into positive coordinates? In other words, what would happen if, whenever you needed to grow the array in a negative direction, you relabeled all the indices such that they were still positive?

#571. 16.21 You are looking for values a and b where `sumA - a + b = sumB - b + a`. Do the math to work out what this means for a and b's values.

#572. 16.9 Approach these one by one, starting with subtraction. Once you've completed one function, you can use it to implement the others.

#573. 17.6 Start with a brute force solution.

#574. 16.23 Start with a brute force solution. How many times does it call `rand5()` in the worst case?

#575. 17.20 Another way to think about this is: Can you maintain the bottom half of elements and the top half of elements?

#576. 16.10 Solution 3: Be careful with the little details in this problem. Does your algorithm/code handle a person who dies in the same year that they are born? This person should be counted as one person in the population count.

#577. 17.26 Solution 2: The list of documents similar to {13, 16, 21, 3} includes all documents with a 13, 16, 21, and 3. How can we efficiently find this list? Remember that we'll be doing this for many documents, so some precomputing can make sense.

#578. 17.16 Iterative solution: Use an example and work backwards. You can easily find the optimal solution for the subarrays $\{r_n\}$, $\{r_{n-1}, r_n\}$, $\{r_{n-2}, \ldots, r_n\}$. How would you use those to quickly find the optimal solution for $\{r_{n-3}, \ldots, r_n\}$?

#579. 17.2 Suppose you had a method `shuffle` that worked on decks up to `n - 1` elements. Could you use this method to implement a new `shuffle` method that works on decks up to n elements?

#580. 17.22 Create a mapping from a wildcard form (like `b_ll`) to all words in that form. Then, when you want to find all words that are one edit away from `bill`, you can look up `_ill`, `b_ll`, `bi_l`, and `bil_` in the mapping.

#581. 17.24 The sum of just D will be `sum(A&B&C&D) - sum(A&B) - sum(A&C) + sum(A)`.

#582. 17.17 Can you use a trie?

#583. 16.21 If we do the math, we are looking for a pair of values such that `a - b = (sumA - sumB) / 2`. The problem then reduces to looking for a pair of values with a particular difference.

#584. 17.26 Solution 2: Try building a hash table from each word to the documents that contain this word. This will allow us to easily find all documents with some similarity to {13, 16, 21, 3}.

#585. 16.5 How does a zero get into the result of `n!`? What does it mean?

#586. 17.7 If each name maps to a list of its alternate spellings, you might have to update a lot of lists when you set X and Y as synonyms. If X is a synonym of {A, B, C}, and Y is a synonym of {D, E, F} then you would need to add {Y, D, E, F} to A's synonym list, B's synonym list, C's synonym list, and X's synonym list. Ditto for {Y, D, E, F}. Can we make this faster?

#587. 17.16 Iterative solution: If you take an appointment, you can't take the next appointment, but you can take anything after that. Therefore, optimal(r_i, ..., r_n) = max(r_i + optimal(r_{i+2}, ..., r_n), optimal(r_{i+1}, ..., r_n)). You can solve this iteratively by working backwards.

#588. 16.8 Have you considered negative numbers? Does your solution work for values like 100,030,000?

#589. 17.15 When you get recursive algorithms that are very inefficient, try looking for repeated subproblems.

#590. 17.19 Part 1: If you have to find the missing number in O(1) space and O(N) time, then you can do a only constant number of passes through the array and can store only a few variables.

#591. 17.9 Look at the list of all values for $3^a * 5^b * 7^c$. Observe that each value in the list will be 3*(some previous value), 5*(some previous value), or 7*(some previous value).

#592. 16.21 A brute force solution is to just look through all pairs of values to find one with the right difference. This will probably look like an outer loop through A with an inner loop through B. For each value, compute the difference and compare it to what we're looking for. Can we be more specific here, though? Given a value in A and a target difference, do we know the exact value of the element within B we're looking for?

#593. 17.14 What about using a heap or tree of some sort?

#594. 16.17 If we tracked the running sum, we should reset it as soon as the subsequence becomes negative. We would never add a negative sequence to the beginning or end of another subsequence.

#595. 17.24 With precomputation, you should be able to get a runtime of $O(N^4)$. Can you make this even faster?

#596. 17.3 Try this recursively. Suppose you had an algorithm to get a subset of size m from n - 1 elements. Could you develop an algorithm to get a subset of size m from n elements?

#597. 16.24 Can we make this faster with a hash table?

#598. 17.22 Your previous algorithm probably resembles a depth-first search. Can you make this faster?

#599. 16.22 Option #3: Another thing to think about is whether you even need a grid to implement this. What information do you actually need in the problem?

#600. 16.9 Subtraction: Would a negate function (which converts a positive integer to negative) help? Can you implement this using the add operator?

#601. 17.1 Focus on just one of the steps above. If you "forgot" to carry the ones, what would the add operation look like?

#602. 16.21 What the brute force really does is look for a value within B which equals `a - target`. How can you more quickly find this element? What approaches help us quickly find out if an element exists within an array?

#603. 17.26 Solution 2: Once you have a way of easily finding the documents similar to a particular document, you can go through and just compute the similarity to those documents using a simple algorithm. Can you make this faster? Specifically, can you compute the similarity directly from the hash table?

#604. 17.10 The majority element will not necessarily look like the majority element at first. It is possible, for example, to have the majority element appear in the first element of the array and then not appear again for the next eight elements. However, in those cases, the majority element will appear later in the array (in fact, many times later on in the array). It's not necessarily critical to continue checking a specific instance of an element for majority status once it's already looking "unlikely."

#605. 17.7 Instead, X, A, B, and C should map to the same instance of the set `{X, A, B, C}`. Y, D, E, and F should map to the same instance of `{Y, D, E, F}`. When we set X and Y as synonyms, we can then just copy one of the sets into the other (e.g., add `{Y, D, E, F}` to `{X, A, B, C}`). How else do we change the hash table?

#606. 16.21 We can use a hash table here. We can also try sorting. Both help us locate elements more quickly.

#607. 17.16 Iterative solution: If you're careful about what data you really need, you should be able to solve this in `O(n)` time and `O(1)` additional space.

#608. 17.12 Think about it this way: If you had methods called `convertLeft` and `convertRight` (which would convert left and right subtrees to doubly linked lists), could you put those together to convert the whole tree to a doubly linked list?

#609. 17.19 Part 1: What if you added up all the values in the array? Could you then figure out the missing number?

#610. 17.4 How long would it take you to figure out the least significant bit of the missing number?

#611. 17.26 Solution 2: Imagine you are looking up the documents similar to {1, 4, 6} by using a hash table that maps from a word to documents. The same document ID appears multiple times when doing this lookup. What does that indicate?

#612. 17.6 Rather than counting the number of twos in each number, think about digit by digit. That is, count the number of twos in the first digit (for each number), then the number of twos in the second digit (for each number), then the number of twos in the third digit (for each number), and so on.

#613. 16.9 Multiply: it's easy enough to implement `multiply` using add. But how do you handle negative numbers?

#614. 16.17 You can solve this in `O(N)` time and `O(1)` space.

#615. 17.24 Suppose this was just a single array. How could we compute the subarray with the largest sum? See 16.17 for a solution to this.

#616. 16.22 Option #3: All you actually need is some way of looking up if a cell is white or black (and of course the position of the ant). Can you just keep a list of all the white cells?

#617. 17.17 One solution is to insert every suffix of the larger string into the trie. For example, if the word is dogs, the suffixes would be dogs, ogs, gs, and s. How would this help you solve the problem? What is the runtime here?

#618. 17.22 A breadth-first search will often be faster than a depth-first search—not necessarily in the worst case, but in many cases. Why? Can you do something even faster than this?

#619. 17.5 What if you just started from the beginning, counting the number of As and the number of Bs you've seen so far? (Try making a table of the array and the number of As and Bs thus far.)

#620. 17.10 Note also that the majority element must be the majority element for some subarray and that no subarray can have multiple majority elements.

#621. 17.24 Suppose I just wanted you to find the maximum submatrix starting at row `r1` and ending at row `r2`, how could you most efficiently do this? (See the prior hint.) If I now wanted you find the maximum subarray from `r1` to `(r2+2)`, could you do this efficiently?

#622. 17.9 Since each number is 3, 5, or 7 times a previous value in the list, we could just check all possible values and pick the next one that hasn't been seen yet. This will result in a lot of duplicated work. How can we avoid this?

#623. 17.13 Can you just try all possibilities? What might that look like?

#624. 16.26 Multiplication and division are higher priority operations. In an expression like `3*4 + 5*9/2 + 3`, the multiplication and division parts need to be grouped together.

#625. 17.14 If you picked an arbitrary element, how long would it take you to figure out the rank of this element (the number of elements bigger or smaller than it)?

#626. 17.19 Part 2: We're now looking for two missing numbers, which we will call a and b. The approach from part 1 will tell us the sum of a and b, but it won't actually tell us a and b. What other calculations could we do?

#627. 16.22 Option #3: You could consider keeping a hash set of all the white cells. How will you be able to print the whole grid, though?

#628. 17.1 The adding step alone would convert 1 + 1 -> 0, 1 + 0 -> 1, 0 + 1 -> 1, 0 + 0 -> 0. How do you do this without the + sign?

#629. 17.21 What role does the tallest bar in the histogram play?

#630. 16.25 What data structure would be most useful for the lookups? What data structure would be most useful to know and maintain the order of items?

#631. 16.18 Start with a brute force approach. Can you try all possibilities for a and b?

#632. 16.6 What if you sorted the arrays?

#633. 17.11 Can you just iterate through both arrays with two pointers? You should be able to do it in `O(A+B)` time, where A and B are the sizes of the two arrays.

#634. 17.2 You could build this algorithm recursively by swapping the nth element for any of the elements before it. What would this look like iteratively?

#635. 16.21 What if the sum of A is 11 and the sum of B is 8? Can there be a pair with the right difference? Check that your solution handles this situation appropriately.

#636. 17.26 Solution 3: There's an alternative solution. Consider taking all of the words from all of the documents, throwing them into one giant list, and sorting this list. Assume you could still know which document each word came from. How could you track the similar pairs?

#637. 16.23 Make a table indicating how each possible sequence of calls to `rand5()` would map to the result of `rand7()`. For example, if you were implementing `rand3()` with `(rand2() + rand2()) % 3`, then the table would look like the below. Analyze this table. What can it tell you?

```
1st     2nd     Result
0       0       0
0       1       1
1       0       1
1       1       2
```

#638. 17.8 This problem asks us to find the longest sequence of pairs you can build such that both sides of the pair are constantly increasing. What if you needed only one side of the pair to increase?

#639. 16.15 Try first creating an array with the frequency that each item occurs.

#640. 17.21 Picture the tallest bar, and then the next tallest bar on the left and the next tallest bar on the right. The water will fill the area between those. Can you calculate that area? What do you do about the rest?

#641. 17.6 Is there a faster way of calculating how many twos are in a particular digit across a range of numbers? Observe that roughly $1/10$ th of any digit should be a 2—but only roughly. How do you make that more exact?

#642. 17.1 You can do the add step with an XOR.

#643. 16.18 Observe that one of the substrings, either a or b, must start at the beginning of the string. That cuts down the number of possibilities.

#644. 16.24 What if the array were sorted?

#645. 17.18 Start with a brute force solution.

#646. 17.12 Once you have a basic idea for a recursive algorithm, you might get stuck on this: sometimes your recursive algorithm needs to return the start of the linked list, and sometimes it needs to return the end. There are multiple ways of solving this issue. Brainstorm some of them.

#647. 17.14 If you picked an arbitrary element, you would, on average, wind up with an element around the 50th percentile mark (half the elements above it and half the elements below). What if you did this repeatedly?

#648. 16.9 Divide: If you're trying to compute, where $x = a/b$, remember that $a = bx$. Can you find the closest value for x? Remember that this is integer division and x should be an integer.

#649. 17.19 Part 2: There are a lot of different calculations we could try. For example, we could multiply all the numbers, but that will only lead us to the product of a and b.

#650. 17.10 Try this: Given an element, start checking if this is the start of a subarray for which it's the majority element. Once it's become "unlikely" (appears less than half the time), start checking at the next element (the element after the subarray).

#651. 17.21 You can calculate the area between the tallest bar overall and the tallest bar on the left by just iterating through the histogram and subtracting out any bars in between. You can do the same thing with the right side. How do you handle the remainder of the graph?

#652. 17.18 One brute force solution is to take each starting position and move forward until you've found a subsequence which contains all the target characters.

#653. 16.18 Don't forget to handle the possibility that the first character in the `pattern` is `b`.

#654. 16.20 In the real world, we should know that some prefixes/substrings won't work. For example, consider the number 33835676368. Although 3383 does correspond to `fftf`, there are no words that start with `fftf`. Is there a way we can short-circuit in cases like this?

#655. 17.7 An alternative approach is to think of this as a graph. How would this work?

#656. 17.13 You can think about the choices the recursive algorithm makes in one of two ways: (1) At each character, should I put a space here? (2) Where should I put the next space? You can solve both of these recursively.

#657. 17.8 If you needed only one side of the pair to increase, then you would just sort all the values on that side. Your longest sequence would in fact be all of the pairs (other than any duplicates, since the longest sequence needs to strictly increase). What does this tell you about the original problem?

#658. 17.21 You can handle the remainder of the graph by just repeating this process: find the tallest bar and the second tallest bar, and subtract out the bars in between.

#659. 17.4 To find the least significant bit of the missing number, note that you know how many `0`s and `1`s to expect. For example, if you see three `0`s and three `1`s in the least significant bit, then the missing number's least significant bit must be a `1`. Think about it: in any sequence of `0`s and `1`s, you'd get a `0`, then a `1`, then a `0`, then a `1`, and so on.

#660. 17.9 Rather than checking all values in the list for the next value (by multiplying each by 3, 5, and 7), think about it this way: when you insert a value `x` into the list, you can "create" the values `3x`, `5x`, and `7x` to be used later.

#661. 17.14 Think about the previous hint some more, particularly in the context of quicksort.

#662. 17.21 How can you make the process of finding the next tallest bar on each side faster?

#663. 16.18 Be careful with how you analyze the runtime. If you iterate through $O(n^2)$ substrings and each one does an $O(n)$ string comparison, then the total runtime is $O(n^3)$.

#664. 17.1 Now focus on the carrying. In what cases will values carry? How do you apply the carry to the number?

#665. 16.26 Consider thinking about it as, when you get to a multiplication or division sign, jumping to a separate "process" to compute the result of this chunk.

#666. 17.8 If you sort the values based on height, then this will tell you the ordering of the final pairs. The longest sequence must be in this relative order (but not necessarily containing all of the pairs). You now just need to find the longest increasing subsequence on weight while keeping the items in the same relative order. This is essentially the same problem as having an array of integers and trying to find the longest sequence you can build (without reordering those items).

#667. 16.16 Consider the three subarrays: `LEFT`, `MIDDLE`, `RIGHT`. Focus on just this question: Can you sort middle such that the entire array becomes sorted? How would you check this?

#668. 16.23 Looking at this table again, note that the number of rows will be 5^k, where k is the max number of calls to `rand5()`. In order to make each value between 0 and 6 have equal probability, $1/7$ th of the rows must map to 0, $1/7$ th to 1, and so on. Is this possible?

#669. 17.18 Another way of thinking about the brute force is that we take each starting index and *find the next instance of each element in the target string*. The maximum of *all* these next instances marks the end of a subsequence which contains all the target characters. What is the runtime of this? How can we make it faster?

#670. 16.6 Think about how you would merge two sorted arrays.

#671. 17.5 When the above tables have equal values for the number of As and Bs, the entire subarray (starting from index 0) has an equal number of As and Bs. How could you use this table to find qualifying subarrays that don't start at index 0?

#672. 17.19 Part 2: Adding the numbers together will tell us the result of `a + b`. Multiplying the numbers together will tell us the result of `a * b`. How can we get the exact values for `a` and `b`?

#673. 16.24 If we sorted the array, we could do repeated binary searches for the complement of a number. What if, instead, the array is given to us sorted? Could we then solve the problem in `O(N)` time and `O(1)` space?

#674. 16.19 If you were given the row and column of a water cell, how can you find all connected spaces?

#675. 17.7 We can treat adding X, Y as synonyms as adding an edge between the X node and the Y node. How then do we figure out the groups of synonyms?

#676. 17.21 Can you do precomputation to compute the next tallest bar on each side?

#677. 17.13 Will the recursive algorithm hit the same subproblems repeatedly? Can you optimize with a hash table?

#678. 17.14 What if, when you picked an element, you swapped elements around (as you do in quicksort) so that the elements below it would be located before the elements above it? If you did this repeatedly, could you find the smallest one million numbers?

#679. 16.6 Imagine you had the two arrays sorted and you were walking through them. If the pointer in the first array points to 3 and the pointer in the second array points to 9, what effect will moving the second pointer have on the difference of the pair?

#680. 17.12 To handle whether your recursive algorithm should return the start or the end of the linked list, you could try to pass a parameter down that acts as a flag. This won't work very well, though. The problem is that when you call `convert(current.left)`, you want to get the end of `left`'s linked list. This way you can join the end of the linked list to `current`. But, if `current` is someone else's right subtree, `convert(current)` needs to pass back the start of the linked list (which is actually the start of `current.left`'s linked list). Really, you need both the start and end of the linked list.

#681. 17.18 Consider the previously explained brute force solution. A bottleneck is repeatedly asking for the next instance of a particular character. Is there a way you can optimize this? You should be able to do this in `O(1)` time.

#682. 17.8 Try a recursive approach that just evaluates all possibilities.

#683. 17.4 Once you've identified that the least significant bit is a 0 (or a 1), you can rule out all the numbers without 0 as the least significant bit. How is this problem different from the earlier part?

#684. 17.23 Start with a brute force solution. Can you try the biggest possible square first?

#685. 16.18 Suppose you decide on a specific value for the "a" part of a pattern. How many possibilities are there for b?

#686. 17.9 When you add x to the list of the first k values, you can add 3x, 5x, and 7x to some new list. How do you make this as optimal as possible? Would it make sense to keep multiple queues of values? Do you always need to insert 3x, 5x, and 7x? Or, perhaps sometimes you need to insert only 7x? You want to avoid seeing the same number twice.

#687. 16.19 Try recursion to count the number of water cells.

#688. 16.8 Consider dividing up a number into sequences of three digits.

#689. 17.19 Part 2: We could do both. If we know that `a + b = 87` and `a * b = 962`, then we can solve for a and b: `a = 13` and `b = 74`. But this will also result in having to multiply really large numbers. The product of all the numbers could be larger than 10^{157}. Is there a simpler calculation you can make?

#690. 16.11 Consider building a diving board. What are the choices you make?

#691. 17.18 Can you precompute the next instance of a particular character from each index? Try using a multi-dimensional array.

#692. 17.1 The carry will happen when you are doing 1 + 1. How do you apply the carry to the number?

#693. 17.21 As an alternative solution, think about it from the perspective of each bar. Each bar will have water on top of it. How much water will be on top of each bar?

#694. 16.25 Both a hash table and a doubly linked list would be useful. Can you combine the two?

#695. 17.23 The biggest possible square is NxN. So if you try that square first and it works, then you know that you've found the best square. Otherwise, you can try the next smallest square.

#696. 17.19 Part 2: Almost any "equation" we can come up with will work here (as long as it's not equivalent to a linear sum). It's just a matter of keeping this sum small.

#697. 16.23 It is not possible to divide 5^k evenly by 7. Does this mean that you can't implement `rand7()` with `rand5()`?

#698. 16.26 You can also maintain two stacks, one for the operators and one for the numbers. You push a number onto the stack every time you see it. What about the operators? When do you pop operators from the stack and apply them to the numbers?

#699. 17.8 Another way to think about the problem is this: if you had the longest sequence ending at each element `A[0]` through `A[n-1]`, could you use that to find the longest sequence ending at element `A[n-1]`?

#700. 16.11 Consider a recursive solution.

#701. 17.12 Many people get stuck at this point and aren't sure what to do. Sometimes they need the start of the linked list, and sometimes they need the end. A given node doesn't necessarily know what to return on its `convert` call. Sometimes the simple solution is easiest: always return both. What are some ways you could do this?

#702. 17.19 Part 2: Try a sum of squares of the values.

#703. 16.20 A trie might help us short-circuit. What if you stored the whole list of words in the trie?

#704. 17.7 Each connected subgraph represents a group of synonyms. To find each group, we can do repeated breadth-first (or depth-first) searches.

#705. 17.23 Describe the runtime of the brute force solution.

#706. 16.19 How can you make sure that you're not revisiting the same cells? Think about how breadth-first search or depth-first search on a graph works.

#707. 16.7 When `a > b`, then `a - b > 0`. Can you get the sign bit of `a - b`?

#708. 16.16 In order to be able to sort `MIDDLE` and have the whole array become sorted, you need `MAX(LEFT) <= MIN(MIDDLE and RIGHT)` and `MAX(LEFT and MIDDLE) <= MIN(RIGHT)`.

#709. 17.20 What if you used a heap? Or two heaps?

#710. 16.4 If you were calling `hasWon` multiple times, how might your solution change?

#711. 16.5 Each zero in `n!` corresponds to `n` being divisible by a factor of 10. What does that mean?

#712. 17.1 You can use an AND operation to compute the carry. What do you do with it?

#713. 17.5 Suppose, in this table, index `i` has `count(A, 0->i) = 3` and `count(B, 0->i) = 7`. This means that there are four more Bs than As. If you find a later spot `j` with the same difference (`count(B, 0->j) - count(A, 0->j)`), then this indicates a subarray with an equal number of As and Bs.

#714. 17.23 Can you do preprocessing to optimize this solution?

#715. 16.11 Once you have a recursive algorithm, think about the runtime. Can you make this faster? How?

#716. 16.1 Let `diff` be the difference between `a` and `b`. Can you use `diff` in some way? Then can you get rid of this temporary variable?

#717. 17.19 Part 2: You might need the quadratic formula. It's not a big deal if you don't remember it. Most people won't. Remember that there is such a thing as good enough.

#718. 16.18 Since the value of `a` determines the value of `b` (and vice versa) and either `a` or `b` must start at the beginning of the value, you should have only `O(n)` possibilities for how to split up the pattern.

#719. 17.12 You could return both the start and end of a linked list in multiple ways. You could return a two-element array. You could define a new data structure to hold the start and end. You could re-use the `BiNode` data structure. If you're working in a language that supports this (like Python), you could just return multiple values. You could solve the problem as a circular linked list, with the start's previous pointer pointing to the end (and then break the circular list in a wrapper method). Explore these solutions. Which one do you like most and why?

#720. 16.23 You can implement `rand7()` with `rand5()`, you just can't do it deterministically (such that you know it will definitely terminate after a certain number of calls). Given this, write a solution that works.

#721. 17.23 You should be able to do this in $O(N^3)$ time, where N is the length of one dimension of the square.

#722. 16.11 Consider memoization to optimize the runtime. Think carefully about what exactly you cache. What is the runtime? The runtime is closely related to the max size of the table.

#723. 16.19 You should have an algorithm that's $O(N^2)$ on an NxN matrix. If your algorithm isn't, consider if you've miscomputed the runtime or if your algorithm is suboptimal.

#724. 17.1 You might need to do the add/carry operation more than once. Adding `carry` to `sum` might cause new values to carry.

#725. 17.18 Once you have the precomputation solution figured out, think about how you can reduce the space complexity. You should be able to get it down to $O(SB)$ time and $O(B)$ space (where B is the size of the larger array and S is the size of the smaller array).

#726. 16.20 We're probably going to run this algorithm many times. If we did more preprocessing, is there a way we could optimize this?

#727. 16.18 You should be able to have an $O(n^2)$ algorithm.

#728. 16.7 Have you considered how to handle integer overflow in `a - b`?

#729. 16.5 Each factor of 10 in `n!` means `n!` is divisible by 5 and 2.

#730. 16.15 For ease and clarity in implementation, you might want to use other methods and classes.

#731. 17.18 Another way to think about it is this: Imagine you had a list of the indices where each item appeared. Could you find the first possible subsequence with all the elements? Could you find the second?

#732. 16.4 If you were designing this for an NxN board, how might your solution change?

#733. 16.5 Can you count the number of factors of 5 and 2? Do you need to count both?

#734. 17.21 Each bar will have water on top of it that matches the minimum of the tallest bar on the left and the tallest bar on the right. That is, `water_on_top[i] = min(tallest_bar(0->i), tallest_bar(i, n))`.

#735. 16.16 Can you expand the middle until the earlier condition is met?

#736. 17.23 When you're checking to see if a particular square is valid (all black borders), you check how many black pixels are above (or below) a coordinate and to the left (or right) of this coordinate. Can you precompute the number of black pixels above and to the left of a given cell?

#737. 16.1 You could also try using XOR.

#738. 17.22 What if you did a breadth-first search starting from both the source word and the destination word?

#739. 17.13 In real life, we would know that some paths will not lead to a word. For example, there are no words that start with `hellothisism`. Can we terminate early when going down a path that we know won't work?

#740. 16.11 There's an alternate, clever (and very fast) solution. You can actually do this in linear time without recursion. How?

#741. 17.18 Consider using a heap.

#742. 17.21 You should be able to solve this in `O(N)` time and `O(N)` space.

#743. 17.17 *Alternatively, you could insert each of the smaller strings into the trie.* How would this help you solve the problem? What is the runtime?

#744. 16.20 With preprocessing, we can actually get the lookup time down to `O(1)`.

#745. 16.5 Have you considered that 25 actually accounts for two factors of 5?

#746. 16.16 You should be able to solve this in `O(N)` time.

#747. 16.11 Think about it this way. You are picking K planks and there are two different types. All choices with 10 of the first type and 4 of the second type will have the same sum. Can you just iterate through all possible choices?

#748. 17.25 Can you use a trie to terminate early when a rectangle looks invalid?

#749. 17.13 For early termination, try a trie.