CRACKING *the* CODING INTERVIEW

189 PROGRAMMING QUESTIONS & SOLUTIONS

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

6TH EDITION

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.

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.

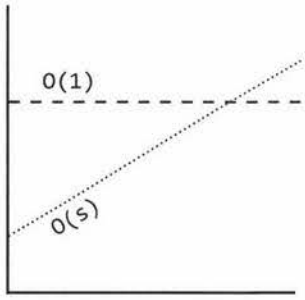
What if there were no flights, and instead you had to drive across the country? Even then, for a really huge file, it would be faster to drive.

▸ 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:

```
int pairSumSequence(int n) { /* Ex 2.*/
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += pairSum(i, i + 1);
  }
  return sum;
}

int pairSum(int a, int b) {
  return a + b;
}
```

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

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
  if (x < min) min = x;
  if (x > max) max = x;
}
```

Min and Max 2

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
  if (x < min) min = x;
}
for (int x : array) {
  if (x > max) max = x;
}
```

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.

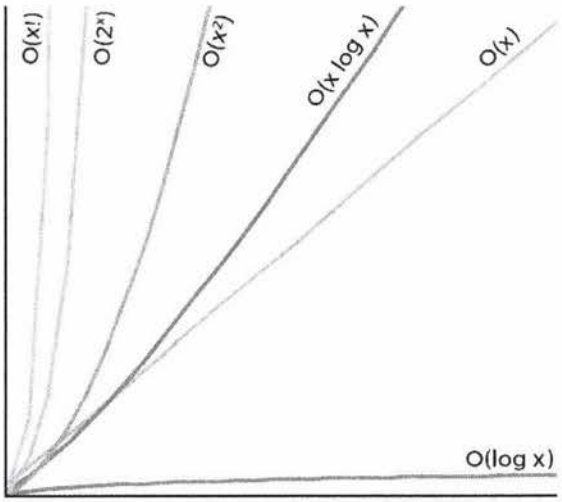
We already said that we drop constants. Therefore, $O(N^2 + N^2)$ would be $O(N^2)$. If we don't care about that latter N^2 term, why would we care about N? We don't.

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

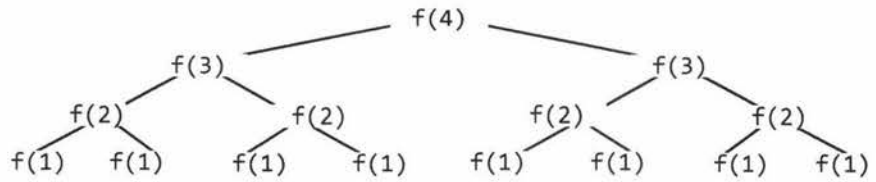
Here's a tricky one. What's the runtime of this code?

```
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 $x = \sqrt{n}$ (when x equals the square root of n).

This for loop is really something like this:

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

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

Example 11

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

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

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

Example 12

This code counts all permutations of a string.

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

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

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

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

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

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

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

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

How long does each function call take?

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

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

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

What is the total runtime?

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

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

Example 13

The following code computes the Nth Fibonacci number.

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

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

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

Listen

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

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?

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!

BUD Optimization

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

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!

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.

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.

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.

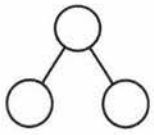
2. Draw an Example

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).

```
n = 1000
for c from 1 to n
  for d from 1 to n
    result = c³ + d³
    append (c, d) to list at value map[result]
for a from 1 to n
  for b from 1 to n
    result = a³ + b³
    list = map.get(result)
    for each pair in list
      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.

```
n = 1000
for c from 1 to n
  for d from 1 to n
    result = c³ + d³
    append (c, d) to list at value map[result]

for each result, list in map
  for each pair1 in list
    for each pair2 in list
      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(N²)
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:

```
dict = {"left": 1, "right": 2, "top": 3, "bottom": 4};
```

Java:

```
HashMap<String, Integer> dict = new HashMap<String, Integer>().
dict.put("left", 1);
dict.put("right", 2);
dict.put("top", 3);
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:

```
HM<S, I> dict = new HM<S, I>().
dict.put("left", 1);
...      "right", 2
...      "top", 3
...      "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]` * $x^{\texttt{exponents[i]}}$.

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.

IX Interview Questions

*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

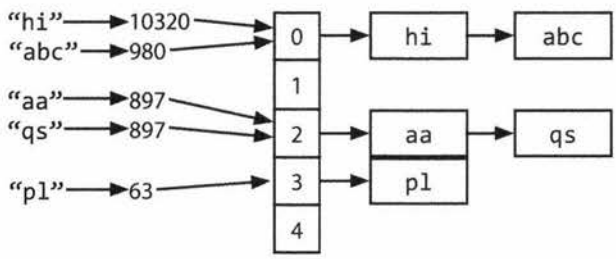
A hash table is a data structure that maps keys to values for highly efficient lookup. There are a number of ways of implementing this. Here, we will describe a simple but common implementation.

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).

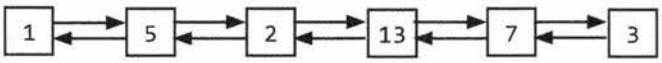
Hints start on page 653.

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  }
```

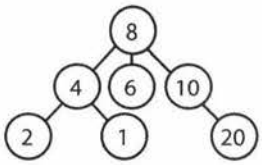
You might also have a `Tree` class to wrap this node. For the purposes of interview questions, we typically do not use a `Tree` class. You can if you feel it makes your code simpler or better, but it rarely does.

```
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.

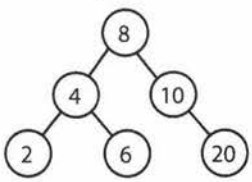
Binary Tree vs. Binary Search Tree

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`.

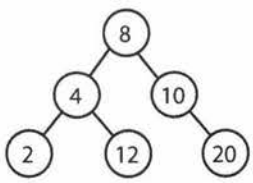
> The definition of a binary search tree can vary slightly with respect to equality. Under some definitions, the tree cannot have duplicate values. In others, the duplicate values will be on the right or can be on either side. All are valid definitions, but you should clarify this with your interviewer.

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).

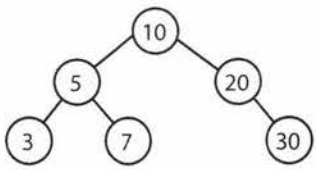
One way to think about it is that a "balanced" tree really means something more like "not terribly imbalanced." It's balanced enough to ensure `O(log n)` times for `insert` and `find`, but it's not necessarily as balanced as it could be.

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.

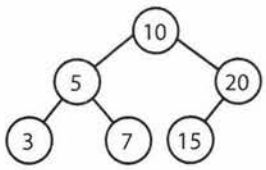
Complete Binary Trees

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



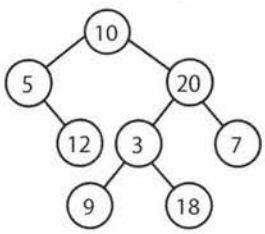
a complete binary tree



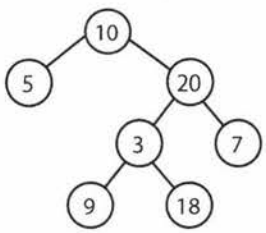
Full Binary Trees

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

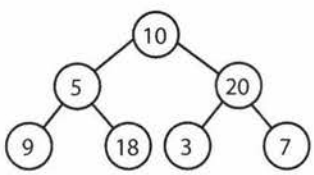


a full binary tree



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").

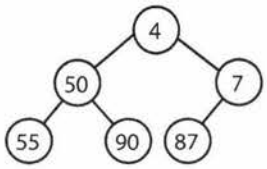
```
1  void postOrderTraversal(TreeNode node) {
2    if (node != null) {
3      postOrderTraversal(node.left);
4      postOrderTraversal(node.right);
5      visit(node);
6    }
7  }
```

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.

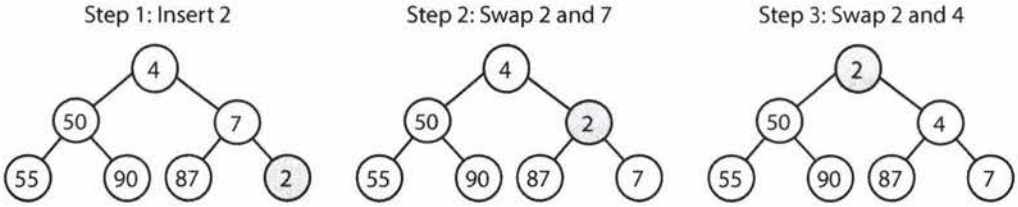


We have two key operations on a min-heap: `insert` and `extract_min`.

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.



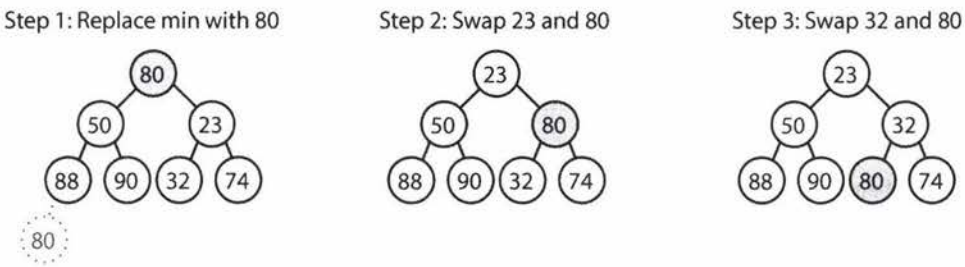
This takes `O(log n)` time, where n is the number of nodes in the heap.

Extract Minimum Element

Finding the minimum element of a min-heap is easy: it's always at the top. The trickier part is how to remove it. (In fact, this isn't that tricky.)

First, we remove the minimum element and swap it with the last element in the heap (the bottommost, rightmost element). Then, we bubble down this element, swapping it with one of its children until the min-heap property is restored.

Do we swap it with the left child or the right child? That depends on their values. There's no inherent ordering between the left and right element, but you'll need to take the smaller one in order to maintain the min-heap ordering.



This algorithm will also take `O(log n)` time.

▸ Tries (Prefix Trees)

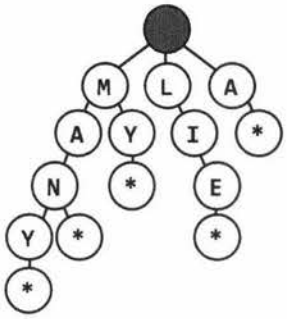
A trie (sometimes called a prefix tree) is a funny data structure. It comes up a lot in interview questions, but algorithm textbooks don't spend much time on this data structure.

A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word.

The * nodes (sometimes called "null nodes") are often used to indicate complete words. For example, the fact that there is a * node under MANY indicates that MANY is a complete word. The existence of the MA path indicates there are words that start with MA.

The actual implementation of these * nodes might be a special type of child (such as a `TerminatingTrieNode`, which inherits from `TrieNode`). Or, we could use just a boolean flag `terminates` within the "parent" node.

A node in a trie could have anywhere from 1 through `ALPHABET_SIZE + 1` children (or, 0 through `ALPHABET_SIZE` if a boolean flag is used instead of a * node).



Very commonly, a trie is used to store the entire (English) language for quick prefix lookups. While a hash table can quickly look up whether a string is a valid word, it cannot tell us if a string is a prefix of any valid words. A trie can do this very quickly.

> How quickly? A trie can check if a string is a valid prefix in `O(K)` time, where K is the length of the string. This is actually the same runtime as a hash table will take. Although we often refer to hash table lookups as being `O(1)` time, this isn't entirely true. A hash table must read through all the characters in the input, which takes `O(K)` time in the case of a word lookup.

Many problems involving lists of valid words leverage a trie as an optimization. In situations when we search through the tree on related prefixes repeatedly (e.g., looking up M, then MA, then MAN, then MANY), we might pass around a reference to the current node in the tree. This will allow us to just check if Y is a child of MAN, rather than starting from the root each time.

▸ Graphs

A tree is actually a type of graph, but not all graphs are trees. Simply put, a tree is a connected graph without cycles.

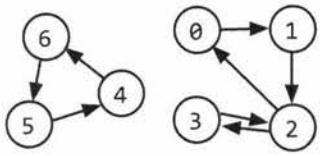
A graph is simply a collection of nodes with edges between (some of) them.

- Graphs can be either directed (like the following graph) or undirected. While directed edges are like a

one-way street, undirected edges are like a two-way street.

- The graph might consist of multiple isolated subgraphs. If there is a path between every pair of vertices, it is called a "connected graph."
- The graph can also have cycles (or not). An "acyclic graph" is one without cycles.

Visually, you could draw a graph like this:



In terms of programming, there are two common ways to represent a graph.

Adjacency List

This is the most common way to represent a graph. Every vertex (or node) stores a list of adjacent vertices. In an undirected graph, an edge like `(a, b)` would be stored twice: once in a's adjacent vertices and once in b's adjacent vertices.

A simple class definition for a graph node could look essentially the same as a tree node.

```
1  class Graph {
2    public Node[] nodes;
3  }
4
5  class Node {
6    public String name;
7    public Node[] children;
8  }
```

The Graph class is used because, unlike in a tree, you can't necessarily reach all the nodes from a single node.

You don't necessarily need any additional classes to represent a graph. An array (or a hash table) of lists (arrays, arraylists, linked lists, etc.) can store the adjacency list. The graph above could be represented as:

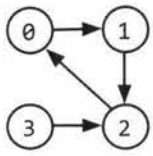
```
0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5
```

This is a bit more compact, but it isn't quite as clean. We tend to use node classes unless there's a compelling reason not to.

Adjacency Matrices

An adjacency matrix is an NxN boolean matrix (where N is the number of nodes), where a `true` value at `matrix[i][j]` indicates an edge from node `i` to node `j`. (You can also use an integer matrix with 0s and 1s.)

In an undirected graph, an adjacency matrix will be symmetric. In a directed graph, it will not (necessarily) be.



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

The same graph algorithms that are used on adjacency lists (breadth-first search, etc.) can be performed with adjacency matrices, but they may be somewhat less efficient. In the adjacency list representation, you can easily iterate through the neighbors of a node. In the adjacency matrix representation, you will need to iterate through all the nodes to identify a node's neighbors.

Graph Search

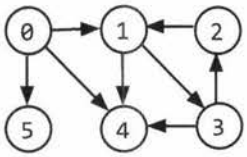
The two most common ways to search a graph are depth-first search and breadth-first search.

In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first (hence the name *depth*-first search) before we go wide.

In breadth-first search (BFS), we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide (hence *breadth*-first search) before we go deep.

See the below depiction of a graph and its depth-first and breadth-first search (assuming neighbors are iterated in numerical order).

Graph



Depth-First Search

```
1  Node 0
2    Node 1
3      Node 3
4        Node 2
5        Node 4
6  Node 5
```

Breadth-First Search

```
1  Node 0
2  Node 1
3  Node 4
4  Node 5
5  Node 3
6  Node 2
```

Breadth-first search and depth-first search tend to be used in different scenarios. DFS is often preferred if we want to visit every node in the graph. Both will work just fine, but depth-first search is a bit simpler.

However, if we want to find the shortest path (or just any path) between two nodes, BFS is generally better. Consider representing all the friendships in the entire world in a graph and trying to find a path of friendships between Ash and Vanessa.

In depth-first search, we could take a path like `Ash -> Brian -> Carleton -> Davis -> Eric -> Farah -> Gayle -> Harry -> Isabella -> John -> Kari`... and then find ourselves very far away. We could go through most of the world without realizing that, in fact, Vanessa is Ash's friend. We will still eventually find the path, but it may take a long time. It also won't find us the shortest path.

In breadth-first search, we would stay close to Ash for as long as possible. We might iterate through many of Ash's friends, but we wouldn't go to his more distant connections until absolutely necessary. If Vanessa is Ash's friend, or his friend-of-a-friend, we'll find this out relatively quickly.

Depth-First Search (DFS)

In DFS, we visit a node a and then iterate through each of a's neighbors. When visiting a node b that is a neighbor of a, we visit all of b's neighbors before going on to a's other neighbors. That is, a exhaustively searches b's branch before any of its other neighbors.

Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited. If we don't, we risk getting stuck in an infinite loop.

The pseudocode below implements DFS.

```
1  void search(Node root) {
2    if (root == null) return;
3    visit(root);
4    root.visited = true;
5    for each (Node n in root.adjacent) {
6      if (n.visited == false) {
7        search(n);
8      }
9    }
10 }
```

Breadth-First Search (BFS)

BFS is a bit less intuitive, and many interviewees struggle with the implementation unless they are already familiar with it. The main tripping point is the (false) assumption that BFS is recursive. It's not. Instead, it uses a queue.

In BFS, node a visits each of a's neighbors before visiting any of *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 }
```

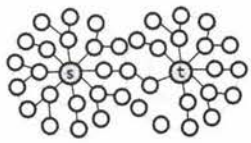
If you are asked to implement BFS, the key thing to remember is the use of the queue. The rest of the algorithm flows from this fact.

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.

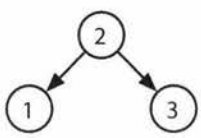
Hints: #10, #16, #28, #36, #46, #70, #80, #96

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.

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.

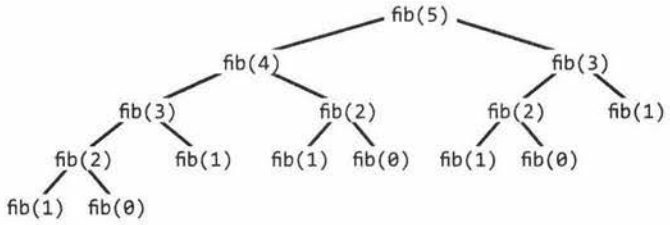
Recursive

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.

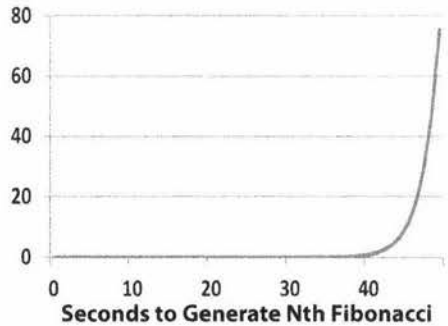
> Tip: Remember this for future problems. Drawing the recursive calls as a tree is a great way to figure out the runtime of a recursive algorithm.

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.

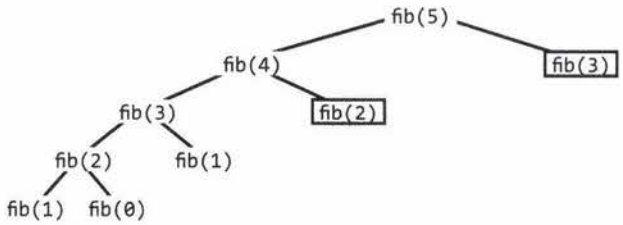
This is exactly what memoization is.

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 }
```

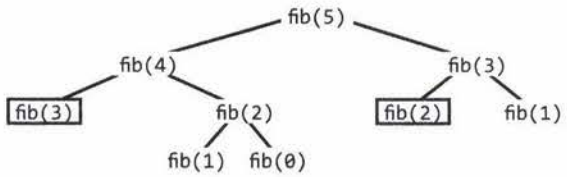
While the first recursive function may take over a minute to generate the 50th Fibonacci number on a typical computer, the dynamic programming method can generate the 10,000th Fibonacci number in just fractions of a millisecond. (Of course, with this exact code, the `int` would have overflowed very early 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.

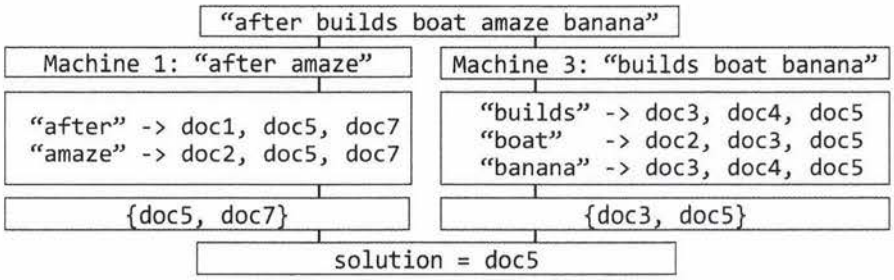
The advantage of this approach is that the lookup table is small and simple (since it must only specify a range of values), and each machine can store a copy of the lookup table. However, the disadvantage is that if new documents or words are added, we may need to perform an expensive shift of keywords.

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.

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