Triplebyte interview prep resources 2

Here's a collection of interview preparation resources that we've collated at Triplebyte.

This is long, and much of it is only relevant to some people. You should read the sections that are relevant to you and not worry about the other sections.

general resources:

- Our blog post. I strongly recommend reading this.

Introductory notes on how to learn things

- For most people, it's worth trying to study things in the most enjoyable way possible. For example, if you find coding on a computer much more fun than writing code on paper, you should consider using LeetCode instead of doing problems in CtCI.
- If you want to memorize things, I recommend using Anki. Especially if you have longer to study, eg if you have a month.

Practicing coding problems

Pretty much everyone should try to get faster at solving small-scale programming problems of the type you're asked in interviews. Here's some advice on how to do this.

PRACTICE

The most important thing to do is to practice solving programming problems. This is obvious, but it's kind of annoying to do, so people don't normally do enough of it. So my main goal here is presenting enough different types of useful programming practice that at least one of them will appeal to you. I strongly recommend trying whichever of these looks like the most fun, and then spending an hour on it.

- LeetCode. I personally find this one super satisfying.
- Cracking the Coding Interview. CtCI has lots of questions in it, and it's good if you want to practice on paper.
- InterviewCake has the best walkthroughs of its problems.
- exercism.io has lots of reasonably easy exercises.
- GeeksForGeeks is my favorite source of harder algorithms problems.

Here's one slightly unusual part of my philosophy of practicing for interviews: I think it's fine for you to “cheat” and read the answers to questions you're studying. I sometimes talk to engineers who tell me that they spent four hours thinking about how to solve a LeetCode question. I almost always think they would have been better giving up after twenty minutes and reading the answer until they understood it well enough to implement it, then implementing it and moving on to the next question, and then doing this three more times. So if you're trying to solve a problem and can't come up with a solution, I recommend reading the solution and then implementing it.

THINGS THAT AREN'T JUST PRACTICING MORE

What else is important? Well, you might find it useful to learn some more about algorithms and data structures. I give some advice about that under the “Algorithms” heading below.

Even in interviews that don't cover the classic algorithms and data structures, interviewers sometimes want you to understand some key algorithmic concepts

Here's my main other idea: I think you might find it valuable to study up on the powerful things your programming language can do that you don't really know how to use. Often I find myself doing things in a way that I know isn't really optimal because I can't be bothered learning the nicer way to do it. I think that it might make sense to try to identify a few language features that you don't use because you're not comfortable with them, and then learn how to use those.

One idea: go read the docs for the Array class in your favorite programming language and see what methods it has that you didn't know about.

Some examples of fun language features that might be useful to know about in interviews, as well as in your day-to-day work:

- Python: nested list comprehensions, generator functions
- Ruby: tap, group_by, the collect method on Hashes
- Javascript: Object.keys, map, find, Array(10).fill(4)
- Java: Arrays.binarySearch

Core algorithms to know

You should definitely know how to write breadth first search and depth first search, because these come up in a ridiculous proportion of interviews and they're a little bit fiddly. A few notes:

- Here are some different problems where the answer is to write a breadth first search (or a different graph search):
    - Implement the Microsoft Paint "fill" button—given a bitmap and a starting position and a target color, find all the cells which are currently the color of the starting cell and change their color to the target color.
    - Given a maze encoded in ASCII, find the shortest path from the start to the end and print out the path.
    - Word chains
    - Writing a web crawler.
- BFS involves knowing how to use a FIFO queue in your language of choice. You should figure out now what the queue implementation is in your language and how to use it. Remember that using an array as a FIFO queue means that pop_front() takes linear time.
- I think that my notes on graph search here are somewhat useful.

Here are the data structures you should know:

- Dynamic arrays (aka ArrayLists)
    - You should know how resizing works.
- Hash tables
    - You should know how long it takes to insert and update and retrieve values.
    - You might enjoy learning how the hash map implementation in your favorite programming language actually works. You can probably find that information in this blog post of mine.
- Binary search trees
    - You should know how long insertion, deletion, and searching take.
    - You don't need to know how self-balancing binary search trees work. If you want to learn it anyway, I recommend reading these slides. If you want to just get a rough idea of it, I recommend these notes of mine.

    - Advanced material: BST augmentation.
- Binary heaps
    - You should know how long insert, popMin, and getMin take, and you should probably know how they work.
    - Here's an interesting point that a lot of people don't know: Binary heaps aren't actually asymptotically faster than BSTs for most of their operations. But they consume a lot less memory and they're faster in practice.
- Linked lists
    - You should know about singly vs doubly linked lists. You should know how long insertion, deletion, and retrieval by index take.

My favorite algorithms textbook is Skiena's Algorithm Design Manual. A PDF of it is freely available online. **I particularly recommend reading chapter 3.** Chapter 3 is a great refresher on basic data structures. If you're more experienced and want to get a better understanding of how to think about algorithms problems, I recommend chapters 3-6. Either way, I also recommend you take a look at chapter 12. It's fantastic—it's just a long list of data structures and advice on how to choose the right data structure for a given task.

You should know what data structures are used to implement the most commonly used collections in your programming language.

Interesting advanced readings in algorithms

If you're already good at algorithms but want to learn some cool advanced stuff, here's some stuff I like.

- The isometry between red-black trees and BTrees. BTrees are funny because most engineers have heard of them but very few engineers actually know how they work. Also, if you read this and understand it, you'll understand red-black trees.
- Augmented binary search trees (slide 8 and onwards) are an extremely useful data structure. There are a lot of hard algorithm problems which you can solve to within a O(log(n)) factor of the optimal solution just by throwing an augmented BST at the problem.
- Bloom filters. These are a fun randomized data structure.
    - If you liked bloom filters, you'll love count-min sketches and HyperLogLog. These are some even cooler randomized data structures.
- Treaps are an interestingly simple balanced binary search tree implementation.
- A Fenwick tree is like an augmented binary search tree without the BST part.
- I really loved the slides to the Stanford course CS166. I feel like reading these slides helped me to significantly level up my data structures knowledge.
    - The range-minimum query problem has some surprisingly beautiful solutions. I got a lot better at designing novel data structures from reading these slides.
    - The list of suggested topics for final projects is a good resource if you just want to find some interesting data structures to read about.

Web systems knowledge

Specific concepts that are useful to know about:

- Databases
    - Normalization. This is a really core concept. If you don't know what a foreign key is, read this.

        - Companies **usually don't seem to want you to know the meanings of the different normal forms (1NF, 2NF, etc)**, but they do want you to be able to design a normalized schema. You should also know about why denormalizing your schema might sometimes be a good idea (it can speed up your reads, and it also lets you enforce some constraints you wouldn't be able to enforce otherwise).
    - Database indexes. It is very important to know about these! They speed up reads for the column that they are on, but they make writes slower and increase memory requirements. They are usually implemented with B-trees.
        - You can have composite indexes, which are indexes on multiple columns.
    - SQL explain.
    - Query planning in Postgres.
    - NoSQL—It's worth knowing a little bit about it. Mongo is probably the most famous NoSQL database. Read this.
    - ORMs. There's kind of a cultural association with ORMs which makes it important to know about them: lots of old-school programmers in PHP or C# or whatever don't use them, and startups are wary of hiring people like that. It's useful to remember that ORMs protect you from SQL injection attacks.
    - If you want to learn much more about databases, I recommend reading lecture notes from the Hellerstein course.
- HTTP
    - Read all about the protocol here. Some fun facts to know: How does chunked transfer encoding work?
    - Suppose I'm browsing Reddit at work, over HTTPS. Does HTTPS prevent my boss from seeing that I'm browsing Reddit? Does it prevent my boss from seeing what page I'm on?
    - How do cookies work?
    - How does a web application control the caching behavior of its clients? check this out
    - WebSockets
- TCP/UDP: what's the difference

System design

BASICS

This InterviewCake question is a great explanation of answering a common system design problem.

PAPER Magazine's Kim Kardashian feature.

This is a good overview of common server setups. Definitely read this if you aren't sure of a good answer to "What are the first couple of routine things that you should do when your web application is approaching its limits?"

This gist has a bunch of bunch of topics which are likely to come up, and it also suggests a good process for answering these questions.

Other links:

- Another system design primer. This is longer.

INTERMEDIATE

Reading and understanding blog posts about how real-world systems are designed can help a lot when answering systems design questions. Here are some of my favorite blog posts about scaling specific websites: Imgur, Reddit, Pinterest (1, 2), StackOverflow.

Articles on scaling:

- Amazon Web Services in Plain English. This is short and super worthwhile to read.
- Scaling Your Web App 101: Lessons in Architecture Under Load.
- A beginner's guide to scaling to 11 million users on Amazon's AWS.

Specific terms to understand:

- Horizontal vs vertical scaling
- Sharding, read replicas
- Load balancer vs reverse proxy
- High availability
- I think it's pretty interesting to know how CDNs can use Anycast

ADVANCED

- Aphyr's distributed systems class
- Jeff Hodges, "Distributed Systems for Young Bloods"

Concurrency

Key concepts you will benefit from knowing about:

- Mutexes, semaphores. I think this three part series does a pretty good job of explaining the differences. Maybe read this article too.
- Locks and atomic operations.
- Monitors.
- Readers-writer lock.
- Race conditions and deadlocks
- Condition variables

If you want to get a better idea of how to use concurrency primitives to solve problems, I strongly recommend "The Little Book Of Semaphores". Most of the book is just a long series of exercises with semaphores, in which you develop a good intuition about how to solve problems with them. You can download it for free. I recommend doing some of these if you're going to be doing interviews about concurrency stuff.

When I was learning this I found that it was really useful to see the concrete details of how you actually use concurrency primitives in practice. So you might find it useful to read the APIs for the concurrency and threading primitives in your favorite programming language.

- Pthreads resource
- c++ threads resource
- One person told me: "I was having a hell of a time figuring out what each of the concurrency primitives does, until I just read: https://docs.python.org/2/library/threading.html and used each one in a toy program. So easy."

Low-level systems

If you want to learn a little about computer architecture, I like this fun video.

Here are some concepts that you should know about if you're applying for jobs which require you to know about low-level systems.

- The stack and the heap.
- Implementation of Malloc.
- Implementation of system calls.
- CPU caches
- If you're interested, this long article can tell you about a lot of modern CPU features.
- Threads vs processes.
- Green threads are cooperative multitasking implemented by the programming language runtime, instead of being preemptive multitasking from the OS. see also this article

If you have more time and want a more complete course on low level systems, I agree with the recommendations made here.

If you want to read a book with much more detail on all of this, I really enjoyed Andrew Tanembaum's books "Structured Computer Organization" and "Modern Operating Systems" (the latter of which was Linus Torvald's inspiration to make Linux!).

A classic book on Unix is "Advanced Programming in the Unix Environment"—this is worth looking at if you have a while and you're going to be doing some unusually hardcore low level systems interviews.

Front end specific stuff

I mostly think that you should read the links on this page.

If you're preparing for a front end interview, you might also want to read up on these sections of this guide: "Getting better at coding problems", "Core algorithms to know", and the HTTP part of "Web systems knowledge".

Here are some mistakes that we often see front-end candidates making in interviews:

- **Using technologies they're not familiar with**. If you've heard that MobX is really cool (it is, btw!) and you want to try it out, you should install and use it in the comfort of your own home, rather than `npm install`ing it during a coding exercise and wasting thirty minutes trying to figure out why Webpack doesn't understand decorators.
    - It's probably worth practicing starting up a new project, if you're going to be doing it in interviews. If you like React, `create-react-app` is probably all you need. Otherwise it might be worth building a skeleton project that you can use as the basis of your code.
    - If you know Backbone well and you know React badly, you should probably use Backbone in your interviews, even though it's less fashionable.
- **Getting hung up on nested state**. If you're reading this, you probably managed to handle the nested state in the Kanban problem in your interview. But a lot of people get stuck on this stuff, so I'm mentioning it anyway. Nested state is a common source of complexity in front-end apps, so it's worth making sure you're comfortable with dealing with it. ImmutableJS, MobX and Redux are all good ways of dealing with it in React. I'm not good in any other framework so can't comment on how I advise dealing with complex, nested state in them.

Mobile

It is natural to expect an interview for mobile developers to focus strongly on technologies around building mobile apps, and that is indeed the case most of the times. However, you're also likely to get coding questions that's more focused on algorithm or pure logic. Similar to front-end interviews, you might also want to read up these sections: “Getting better at coding problems”, “Core algorithms to know”, and the HTTP part of “Web systems knowledge”.

For mobile specific stuff, please be prepared to answer these type of questions (these are common ones but other types of question do occur):

- Coding up a small mobile app. Most importantly, be proficient with using common UI components and APIs.
- Know common technologies used in iOS/Android apps. e.g. What are the app components in an Android app? What are the ways to store data on the phone?
- Know your programming language well, and be comfortable using it.
- Be comfortable talking about how to implement specific features or effects. e.g. Some special type of view, infinite scroll, offline mode
- Be comfortable talking about product design.
- Expect questions about dealing with common technical challenges that often arises in mobile app development. e.g. Multi-threading issues, API design and making network API calls.