How to Break Into the Tech Industry—a Guide to Job Hunting and Tech Interviews

I recently completed a job search for my first role as a software engineer.

Despite having first learned how to code almost a year before, having a background as an English major and former professional poker player, I was able to land a total of 8 offers including Google, Uber, Yelp, and Airbnb (where I ultimately joined). In this three-part blog post I'm going to describe my advice to a job-seeker trying to break into the tech industry.

If you haven't read the story of my job search, you can read about it here. It provides some of the backdrop for this post.

First, several caveats.

I have a weird background, but make no mistake—I both worked my ass off and got very lucky. When I got my job offers, I was working as an instructor and Director of Product of App Academy, and had been studying (and teaching) this stuff for a little under a year.

In light of that, some people have interpreted this as a “get rich quick” sort of thing. It’s not. It’s more of a a “go into a cave for six months and hopefully get a job when you emerge” sort of thing. Subtle difference, but an important one.

Note that my advice here draws not just from my own experience, but also from the advice of many others. It also comes from observing the job searches of 200+ App Academy students breaking into the tech industry.

That said, take all of my advice with a grain of salt. There are people out there who are much more successful than I, and who know much more about this domain. So if you find there’s some aspect of my advice that seems flat-out wrong or contradicted by others, defer to your own judgment.

I entered into tech so that I could earn-to-give. My hope is that in supplying this advice, I will encourage and enable others to do the same.

I’ve broken this up into three parts: getting interviews, the interview process, and negotiating. This post will cover the first two parts, and I’ll conclude with a lot of advice about the negotiation process in a subsequent post.

Note: this advice will be primarily geared toward the software engineer market in the Bay Area, but a lot of the advice will be generally applicable. When I mention “engineering,” just substitute your own field. Though the details might be different, the underlying dynamics are largely the same.

I hope you find this post helpful in your own job search.

Part 1: Job Hunting

First and foremost, job hunting is a numbers game. You must have faith in the numbers.

If I told you that you have a 4% chance of getting an offer from any application, that might seem dishearteningly low. But mathematically that means that 50 such applications will give you an 87% chance of getting at least one job offer.

And yet, any given application is almost certainly not going to be the one that ends up turning into an offer.

That's okay. Expect those dynamics. It's weird and counterintuitive, but you should probably expect the company that ultimately gives you an offer to seem like a long shot. This is because almost every application will seem like a long shot.

Be comfortable with that. Trust in the law of large numbers.

This applies to networking, raw applications, even to phone screens and on-sites.

Networking

Networking is probably the highest leverage activity in getting job offers.

Networking sounds intimidating. But I promise you, at its heart, it's pretty simple.

Just buy everyone coffee.

And when I say everyone, I mean everyone who works at a tech company.

Every time you meet someone in tech, and I mean anyone at all, you should tell them, “Hey, that’s really cool. I’m actually trying to get a job as a software engineer. I don’t know very much about [the thing you do]. I’d love to buy you coffee and pick your brain about it. Are you free sometime this week around [time]?”

Buy tons and tons of coffee for people. Do it religiously and have no shame about it. I don’t care if you can’t afford it. Go broke inviting people to coffee.

In fact, if you go broke, let me know and I’ll fund your coffee dates until you land an awesome job and can pay me back.

How do you meet tech people in the first place? Go to meetups. Go to local events. Go to developer conferences. Go to house parties. (Techies are normal people, and people are everywhere.)

Get creative. @Message people on Twitter. Cold-email them (specific people, not companies). Don’t be creepy, but it’s totally okay to reach out to people out of the blue, learn about them online, and ask directly if you can buy them coffee.

Most importantly, *plumb your network*. Chances are, you have friends or friends of friends who already work at tech companies. You're just not paying attention.

Okay.

So now you've got a bunch of coffee dates lined up with all these random people who work for tech companies. Some of them might be engineers, or managers, operations, designers, customer support, salespeople. Doesn't matter. Meet anyone and everyone.

Once you've sat someone down for coffee, it's time for an informational interview.

Informational Interviewing

Here's what you do in an informational interview.

First, learn everything and anything you can about this person. Learn everything you can about their company. Learn everything you can about the industry or technology they work with.

Notice I didn't say "ask for a job." Don't ask for a job. This is not about you asking for a job. This is about everything other than you asking for a job.

Get to know them! Get to know their company! Learn more about what they do and why it's important. Ask things you really want to know.

They should be doing most of the talking, otherwise you're doing it wrong. Shut up and ask more questions.

Chances are, if you shut up for long enough, they'll be curious about you and want to know what you're doing. Tell them your story. Tell them about who you are, what brought you here, and why you want a job as a software engineer.

At the end of the conversation, go for the close—which, most of the time, is not to ask for a job referral. Do not ask for a referral unless they bring it up. Rather, ask them: "who else do you know who'd be a good person for me to talk to?" Tell them you want to learn more about front-end development, or cryptography, or iOS, or whatever you were asking them questions about.

You *were* asking them questions, right?

If they can't think of anyone to pass you along to, ask them if they know any other engineers anywhere you could talk to. Or anyone at all who works at another tech company you're interested in. Whatever it is, get contacts for more people to get introduced to. Every conversation should lead to at least one more contact.

Then, just keep following this chain. Schedule more coffee dates, ask more questions, get more contacts. Go through all of it without asking for any unsolicited referrals.

This might seem weird. But studies have shown that this is the most effective way to get a job. And ultimately, it's the most effective way to generate referrals outside of your network.

Why does this work?

It works because people hate it when you ask them for a job.

Give you a job? Why? They don't know you. Why would they give you a job? Why would they even waste their time on you?

The power of informational interviewing is that instead of making it about you, *you make it about them*. People like to talk about themselves. They like to teach others. They want to help. But they don't want to be pestered by strangers for favors.

If you keep doing this, people will see your curiosity and your genuineness. They'll believe in your story, and *they will want to refer* *you*. By not pressuring people to recommend you, you will make them want to do so—or, at the very least, become an ally in your journey.

The reality is, most people *can't* refer you. This is because their company might be too small to hire junior devs, or they might not need people with your skillset, or they might have overhired for the quarter, or, believe it or not, their company could just be secretly failing and not have any money.

That's okay.

If you do this enough times, someone will eventually float the idea: "I could refer you if you want."

If they say this, graciously ask them if they'd be comfortable doing so. They'll then say yes. Now say "golly gee, that would be swell." Then follow up with them over e-mail, thank them, and send them your resume. Congrats, you've just landed yourself a referral.

See? That wasn't so hard.

Now keep them posted on your progress through their company's pipeline. Don't hesitate to also ask them for further advice.

Oh, and, before you leave that coffee date, still ask them for more contacts. I mean it. The chain stops for no one.

Getting the Chain Started

But how do you start the chain?

Chances are, you already have contacts in the tech industry. Even if you think you don’t, you probably do. Go scour your LinkedIn, your Facebook, ask your family, talk to people you kinda knew in high school. Nothing is off limits.

Make a spreadsheet of every single person you know who works for a tech company. Mark down what company they work for, and what their role is at that company.

Then sort all of your contacts by company, and then by role. You should reach out to your most relevant contact at every single company on your list (an engineer is more relevant than someone in operations, etc).

Reach out to every single one.

Invite them to coffee, lunch, dinner, whatever. Always buy. Farm them for referrals and further contacts. If you have a strong network, this is likely to be the most powerful source of leads for you.

This is great, because unless you have a stellar resume (which I assume you don't because you're bothering to read this guide), referrals are going to be an order of magnitude more impactful than raw applications. This is especially true at top companies.

So keep your funnel healthy. Keep scheduling coffee dates. Keep networking, even if you're mid-way through your job search. I didn't stop meeting with people until I had about 4 offers, and even then I still reached out to people at new companies I was interested in.

Resumes and Cover Letter

I'm not going to offer a lot of specific advice here, because there's so much to say and it's so individual to your background and experience. Scour the internet for resources. I'd strongly recommend getting your resume reviewed by someone who knows what they're doing.

Resume short version: keep your resume to one page, always. Highlight anything that sounds kinda math-y or technical. Measure everything in terms of impact, don't just describe your responsibilities. Make your resume pretty, especially if you're claiming to be a front-end or full-stack developer. Don't use Arial or Times New Roman. A little bit of design effort will go a long way. Never, ever have typos.

Some people read cover letters, some people don't. You usually need to write one, and you'll likely end up coming up with a script that you adapt. Always personalize them—try to find some reason why you like the company's mission or find their work interesting. Keep it short, and minimize the cruft. If you need more advice, there's no shortage out there on the internet for cover letter advice.

Job Platforms

Get on all the job hiring platforms. The first job offers I got were from TripleByte, which I cannot recommend highly enough for software engineers.

Also set up a profile on AngelList, WhiteTruffle, SmartHires, and all the rest. Apply to Hired. Clean up your LinkedIn and Github profiles. Get your projects up and README'd and pretty.

Even if some of these platforms reject you or you don't get any bites, they're enormous return on investment for the time you'll spend.

Mass Applying

Referrals and job platforms are unequivocally higher-yield strategies, but you shouldn't ignore raw applications. This is especially true if you have a strong resume, as you're likely to have a higher reply rate. Raw applications are low investment, so you might as well pump them out in your free time and maximize the number of potential bites. They are also the primary way I’ve seen most App Academy students get jobs, so they certainly work. But they require more volume.

I recommend most people mix in some mass applying into their strategy.

You should pre-commit to a number of companies to apply to per day.

Mass applying to companies is really mind-numbing. It’s easy to lose focus. So structure your time and set goals for yourself. Apply to batches of companies in 45-minute sprints. Use Pomodoros to keep yourself on point and accountable.

Company Size

From my observations, small-medium size companies are more likely to take a chance on a raw application.

Large and prestigious companies tend to have rigid resume review processes, and are more likely to trash your resume if it’s not studded with exciting phrases like “Stanford PhD” or “literally a superhero.”

Tiny companies are hit-or-miss.

Shoot for unsexy companies. You’re much more likely to get an interview, and at this stage, getting interview experience under your belt is ultimately what matters.

Understand: just having your first offer matters much more than where the offer is from. Even if the offer is from a company you’re not excited about, you can easily leverage it up to a company you want more. More on this later when we talk about negotiation.

Study, study, study

This is perhaps the most important piece, and where a lot of the hustle comes in. You need to study your ass off. Your study habits are one of the chief determinants of your performance in interviews (not to mention your performance in your subsequent job).

You’re trying to break into this field, so break in!

Below, I’m going to give my general advice to someone aiming for a role as a full-stack or back-end developer. If you want to get a job doing front-end development, game development, embedded systems, or anything sufficiently specialized, you should look up more specific advice for those fields.

This is the preparation that I personally used. My approach to studying has been informed by my knowledge of performance psychology and by the science of learning. But your mileage may vary.

I’m going to assume you already know how to program. If you don’t at all, read this instead as a first step.

I should also note that the following advice is strongly tailored toward interviews at larger, more prestigious companies. At smaller companies, your interview is less likely to consist of abstract algorithms problems. You may have to do raw coding exercises, fix a bug in production code, or build a small semi-practical system. It’s also more likely you’ll be asked questions about specific languages or frameworks. I ran into a few of these interviews, but only at smaller companies. Adjust your study plan accordingly.

Disclaimer: a common refrain in tech is that “programming interviews are broken.” I agree. I think standard tech interviews are poor predictors of software engineering ability. But like any good hacker, I designed this advice to be a reverse-engineering of the standard programming interview. Take it for what it is, and no more.

General Study Strategy

First, a high-level roadmap.

Watch the Princeton Coursera course on algorithms (by Robert Sedgewick), mostly up until Dijkstra’s algorithm and not much more than that. Try to follow along and code as you go. You don’t need to implement things in Java unless you already know Java.

Read The Algorithm Design Manual for general algorithms and data structures background. Go through most of it. (You can skip the giant appendix about specific algorithmic topics).

Buy Cracking the Coding Interview. We’ll be using this as a repository of coding problems and solutions.

Listen regularly to podcasts about software like Software Engineering Daily or The Bike Shed. Read Hacker News. There’s a lot of stuff you won’t understand, but that’s okay. Just soak it in. Over time, this will give you some familiarity with modern web stacks, current trends, and how developers talk.

For system design and architecture, I recommend reading through HiredInTech and reading a bunch of the top blog posts on High Scalability (appears to be down at the time of posting). There’s also a massive Github repo of resources you can check out specifically for systems design interviews.

If you already know a programming language, do your practicing in whatever language you know best. Interviewers won't care. If you have a choice among languages, prefer expressive scripting languages like Ruby, Python, JavaScript, etc. More on this later when we talk about interview advice.

Object oriented design and code organization are very important concepts, and they'll come up in interviews. Unfortunately, there's pretty much no way to learn them other than by just writing lots of well-structured programs, and getting critiques and code reviews.

If you're struggling to understand a concept in a book or on Wikipedia, look up Youtube videos of people explaining it in different ways until you get it. I've found this strategy amazingly effective.

Ask human beings for help when you're struggling. If you don't know anyone, get on StackOverflow or find a good IRC channel for the language you're working in. Be a burden on others. Ask quickly and unashamedly. You have my permission.

If you're working on an algorithms problem (such as out of CTCI) and can't figure it out, spend no more than 20-30 minutes without making progress. Just go look up the answer. Contrary to popular belief, most struggling past 30 minutes is pointless.

Some people will disagree with this.

Screw those people.

That said, if you find up having to look up the answer to more than 2/3rds of problems, your problem set is too hard and you need to downgrade.

But banging your head against a wall or staring blankly at code is a waste of time. For code/algorithms problems, you very much want to maximize the density of learning per hour.

Any time you look up the answer to a coding problem, try to understand the solution. Walk through it with a debugger if it's particularly mysterious. Read multiple explanations of why it works.

Then—and this is extremely important—if you didn't solve the problem completely on your own, treat the problem as though you never solved it at all. Put it back into your rotation of problems. Mark it as something you need to try again from scratch. Wait at least a day, and try to solve it fresh. If you fail, rinse and repeat ad infinitum.

And finally, structure the hell out of your learning. Make a schedule. Know exactly when you're going to be working on this stuff, when you're going to take breaks, when you're going to go to lunch, etc. Build flexibility into your schedule, but have clear goals for how many hours a day you'll spend studying.

That's most of my general studying advice. Now here's the specific study guide I'd recommend.

Programming Interview Study Guide

First and foremost, work through the Princeton algorithms course. You should follow along and implement as much as you can. Along the way, also implement all these major data structures. (Many of these are not explicitly covered in the course—implement them on your own).

- Linked List
- Dynamic array, implemented with a ring buffer (use a statically sized array underneath the hood)
- Hash set
- Hash map (with chaining)
- Binary heap (without decrease-key; know that Fibonacci heaps exist and know their guarantees)
- Binary search tree (doesn't need to be self-balancing; know that self-balancing trees exist and know their guarantees)
- Prefix tree (a.k.a. trie)
- Suffix tree (don't worry about compression, just build a dumb version; know that Ukkonen's algorithm exists and learn its guarantees)
- An object-oriented adjacency list for graphs

Know all of the time complexities for their basic operations. It helps to visually intuit them. Look at images. Draw them out on paper/whiteboard.

Once you've gotten all of those out of the way, implement this subset of the most common algorithms:

- Binary search (implement it both iteratively and recursively)
- Randomized quicksort (pay extra attention to the partition subroutine, as it's useful in a lot of places)
- Mergesort
- Breadth-first search in a graph
- Depth-first search in a graph (augment it to detect cycles)
- Tree traversals (pre-order, in-order, post-order)
- Topological sort (using Tarjan's algorithm)
- Dijkstra's algorithm (without decrease-key)
- Longest common subsequence (using dynamic programming with matrices)
- Knapsack problem (also dynamic programming)

Know the time complexities and space complexities of all of these algorithms (cheat-sheet). Know the difference between amortized, average, and worst-case time guarantees.

Do at least one or two problems where you BFS or DFS through a matrix.

Once you finish these you should have a decent grounding in algorithms and data structures. Read through the entirety of *Algorithm Design*

Manual to solidify your understanding. Implement anything that you find sufficiently interesting.

A few points of interest: learn about heap sort, but don't bother coding it. Know that it's O(1) space but practically very slow due to cache-inefficiency. Learn about quickselect and median-of-medians. Code them if you want.

Learn the basics of bit manipulation. Be able to explain what AND, OR, and XOR do. Know what a signed integer is. Know that there are 8 bits in a byte. Know what assembly is (from a high level), and get a sense of the basic operations that hardware exposes. This awesome video about the inner workings of the original Game Boy covers just about all you need to know.

Learn about SQL databases. Learn how to design a SQL database schema; it comes up in interviews often. Read about ACID, the CAP theorem, and BASE (you don't need to memorize the terms, just understand the concepts at a high level.) Understand why joins can become unscalable. Learn the basics of NoSQL databases.

Read about caches and cache efficiency. Know what a cache miss is. Know that reading from registers is lightning-fast, reading from the various caches is pretty fast, reading from memory is slow, and that reading from the hard disk is abysmally slow.

Read how to implement an LRU cache, and then actually write one that gets and sets in worst-case O(1) time. This is a weirdly common interview problem.

Learn what happens when you type a URL into your browser and press enter. Be comfortable talking about DNS lookups, the request-response cycle, HTTP verbs, TCP vs UDP, and how cookies work. Shows up all the time in interviews.

Learn the standard ways to speed up a slow website. This includes a lot of things like adding database indices to optimize common queries, better caching, loading front-end assets from a CDN, cleaning up zombie listeners, etc. Also an interview classic, pretty rabbit-holey.

(These last two are obviously web development specific.)

Once you've covered those, you now have most of the grounding you need. You should be ready to start interview circuit training.

(If you already have a background in CS, I'd probably start here.)

Interview Circuit Training

First, work through the first 3 problems on every relevant section of *Cracking the Coding Interview*. (That is, skip sections like Java or C++ if you don't know those languages, skip testing, skip concurrency unless you have experience writing concurrent programs. Do read about it though.)

Write out your solutions in code. Make sure they work. Check edge cases.

After you've written a good 30+ solutions, stop coding every single solution. By now you should have some solid coding practice, and you should be developing your high-level intuition how to determine the optimal approach to a problem. The rest is pattern recognition.

Continue working through the rest of *Cracking the Coding Interview*. But from here on out, instead of actually writing code, just write a high-level description of the algorithm you'd use to solve the problem.

Here's an example of what I mean: "First sort the array, then repeatedly binary search through it on each subsequent query. This should take nlog(n) pre-processing and log(n) for each query."

That's enough detail. Once you've written your high-level solution, check it against the solution in the book. If your solution is suboptimal, write a description of the optimal solution. Make sure it makes sense to you. If it even *slightly* doesn't make sense and you're not confident you could write the code, then code it up, test for edge cases, etc.

With this workflow, you should be coding only a small percentage of the remaining problems. You should be able to work through many more problems a day now.

Make sure you have your solutions collected into files so that you can review them easily. In your solutions files, you should also have the problem statement along with any solution you have written/described.

Whenever you review them (which you should at weekly intervals), here's how you should review: read the problem statement only, decide how you'd solve the problem in your head, and then check it against the algorithm you have written down. Again, reaffirm to yourself why the correct answer is correct.

Once you make it through the entire book, spend the remainder of your study time grinding problems on LeetCode.

LeetCode is an online coding platform like HackerRank or CodeWars that gives you a problem statement and lets you write code against test cases.

I recommend LeetCode particularly because it has the most accurate problem set I've seen for what most algorithms interviews are like. It also has an awesome feature that it tags programming problems by companies where the problem was asked. (If the company has fewer than 2000 employees, then it's unlikely to be tagged anywhere.)

So any time you have an interview at a big company, do every single problem that company is tagged with on LeetCode. You have to pay for a subscription to see the tags, but it's worth it.

GlassDoor is also a good source of company-specific problems to grind on. (You can also try CareerCup, but their accuracy tends to be a bit more dubious.)

After all that, there's one last piece that's missing—actual interview practice. One of the chief principles of learning is *practice like performance*. That is, make the way you practice as close to the actual performance as possible. In this case, the actual performance will be standing in front of a whiteboard and getting verbally interviewed by someone.

Practice for it.

Find someone to mock-interview you in front of a whiteboard (and reciprocate for them afterward). Have them ask you questions you've never seen before. Practice all of your interviewing skills, including introducing yourself, clarifying the problem, making jokes, everything that you plan to do in a real interview. Treat it as though it's the real thing. Be willing to struggle and look stupid. Don't let yourself break character.

This is also really the only way to practice architecture-oriented interviews. Have your interviewer ask you about the request-response cycle or how you'd architect Twitter, or other standard interview favorites.

A great resource is interviewing.io, which allows you to anonymously interview and get interviewed by other people and practice programming problems. They're in private beta right now, but check them out.

If you actually implement every part of this workflow, you should become an algorithms beast within a few months. Your performance on a coding interview will be comparable if not superior to the average CS graduate.

There's nothing magical about any of this. It's all just practice. All of the information you need to pass a coding interview is out there, freely available. It's up to you to put the time in.

Okay.

You’re clearly awesome at algorithms now, so let’s keep going.

The moral of this story is: hang in there. Keep grinding. Remember that it’s a numbers game. Keep putting yourself out there, and most importantly, keep taking care of yourself.

The simple stuff is important. It’ll sound stupid, but focus on self-care before anything else in the job search.

Do things that de-stress you, whether that be meditation, taking baths, playing video games, whatever. Spend time with friends. Eat good, real food.

Make sure you’re sleeping. The marginal hour spent sleeping is almost always more useful than the marginal hour spent working.

Get regular exercise, even if you don’t want to and hate everything about it. Countless studies have shown that exercise is basically a panacea. It boosts personal well-being more than any other psychological intervention, including anti-depressants.

Depression will creep up on you.

If it does, notice that it’s there. Talk to people about it. Don’t be ashamed. Fight it actively. Exercise and Vitamin D are your allies. Spend time outside, in the sun. See a professional if you need it.

But your best defense is structuring your time. Religiously schedule yourself. Know what you should be doing everyday.

In many ways, freedom is your enemy. Let structure be your shelter.

Mindset

I used to work as a mental coach for professional poker players, as well as other folks in high-stakes professions. A lot of that work has informed my approach to job searching.

Here’s my advice on mindset.

First and foremost, treat everything as a learning experience. By that I mean, don’t ever go into a phone screen or an onsite interview hoping for a job. Don’t hope for anything at all. Just go there to learn, to mess up, to try things out and see what this whole interviewing thing is all about.

I’d almost say—imagine someone else was invited to this interview, and you’re merely their doppelganger. You get to be a fly on the wall, try things out, but you don’t really care whether they get hired or not.

You’re just doing it because it’s interesting. And why not get more experience interviewing?

Treat every interview like this. Stop caring about the outcome. Don’t hope to get hired at all, just go in there and learn.

This approach will have tremendously beneficial effects for you as a candidate. You won’t be so damn nervous or terrified. You’ll be less attached to the outcome and won’t be crushed if you don’t get the job. (Almost never does a candidate pass their first couple interviews, so you should expect that anyway.) And it makes the entire process more fun and easier to actually learn from.

The second big piece of advice is: continually set goals for yourself. But only set goals that are completely in your control.

For example, a goal like “I want to have a job offer by January 20th” is a terrible goal. It’s not up to you. How would you even know if you’re making incremental progress toward this? You’ll either have a job before then or you won’t.

Instead, choose goals about your own behavior. A goal like “I want to apply to 4 jobs every weekday,” or “I want to ask 4 people for coffee dates every week” are completely in your control and easy to measure.

The third and final piece of advice is on how to frame your job search. Here’s a little parable I used to tell my students.

Your Number

Imagine that somewhere, in God’s Database, is a number. That number is the number of days you’re going to have to work your ass off until you find a job.

Everyone has their own number in God’s database. Even you.

Let’s call your number N.

Every day you spend seriously job searching decrements that N by 1. You don’t know what that N is of course. And you’ll never know what N is until you finally get your first offer. (Assume God sanitizes his inputs against SQL injections.)

You don’t get to choose your N. Many things affect the size of your N—your experience, your education, your network, systemic biases in the industry, and some randomness. A shitload of randomness, actually.

Other people will have smaller Ns than you, even though they’re not as skilled as you are. That’s okay.

Ignore other people. They have it easier, and that’s fine. But your N is already decided, and your only job is to whittle that N down.

So treat every day as progress. Whether you get rejected from a company, whether you screw up an interview, whether you get inexplicably lowballed, it doesn’t matter. That N is getting lower and lower, and your interviewing skills are getting better and better. Be confident. You’re getting closer.

Every single day is progress.

And, lastly…

Keep Having Fun

This is really important.

Keep making things. Keep working on projects. Find things that excite you. Learn new technologies, frameworks, algorithms, languages. When someone asks you “what are you working on lately?” you should have a good answer—one that excites you.

Work on things that are actually fun. It’ll keep you sharp, motivated, and happy.

You’re breaking into this industry because you want to program, right?

So keep programming!

Part 2: The Interview Process

Okay. You did all of part 1 and you’ve got interviews lined up. Great. Now what?

I’m going to reiterate this again: approach every interview as a learning experience. Stop caring whether you’ll get the job or not. The first few times, you’ll almost certainly bomb. It’s okay. Expect that and roll with it.

Interviewing is a skill, and not a particularly mysterious one. Trust that the more times you do it and the more deliberately you analyze your mistakes, the better you’ll get at it.

Don’t worry about getting a job. Each interview probably won’t be the one that leads to a job. That’s okay. Just learn and have fun. You’ll perform much better that way.

The Basics

So your interview is coming up.

If you are uncertain about anything in particular, know that you can ask your recruiter or point of contact at the company. You can ask them things like: What sort of problems will be on the interview? Anything specific I should review? Can I use Python? Will there be any pair programming? Should I bring my own laptop for coding? Will I be able to Google documentation during the interview?

These are all totally fine questions. Nobody will care at all if you ask. I’ve seen many people agonize over questions like these. Just ask.

Some basics:

At a tech startup in the Bay, dress nicely, but casual. Something like a sweater or button-up and jeans works well. In more conservative markets, wear a dress shirt and slacks. If you’re not sure, ask your recruiter. This is in no way a taboo question.

They’ll often offer you something to drink. Take it if you want it. Use their bathroom. Do whatever you can to make yourself relaxed.

When you meet your interviewer, say hello, introduce yourself. Relax. Try to remember their name. (No big deal if you don’t.)

There’s always some ritual to beginning an interview. It’s not too complicated.

Your interviewer will walk in the room. Smile. Tell her your name. Shake hands confidently. Look her in the eye.

Adopt relaxed body language. Lean back in your chair. Try not to fidget—just clasp your hands together if you can't help it.

You're completely fine. This interview is just for practice anyway. Don't analyze yourself. No, it's not weird that your arms are dangling like that.

Ask your interviewer what team she's on and what she's been working on lately. This will get the conversation flowing.

If you're nervous, it's okay to tell them that. If you're eager to get started, it's also okay to tell them that.

Talking About Yourself

When it comes to talking about yourself, most tech interviews are incredibly formulaic.

Almost every question you'll be asked will be a permutation of one of these four:

- What's your story / walk me through your resume / why'd you leave your last job? (these are essentially the same question)
- Why us?
- Tell me about a challenging bug you faced and how you solved it.
- Tell me about an interesting project you worked on.

The first question is particularly important. Essentially, they want to hear your personal narrative. Your answer will strongly influence their perception of you.

This really just comes down to storytelling. Consider yourself a character in a story, and structure the story with a beginning, middle, and end. There should be inflection points, characterization, and easy to understand motivations. Keep it as short as possible, while preserving color and what makes you interesting. Try not to be negative. Frame your story around seeking challenge and wanting to better yourself, rather than rejecting or disliking things.

You will tell this narrative again and again. If you interview enough, eventually it will congeal into a script. The best way to improve at it is to literally practice it out loud and listen to a recording of it. Also try to get feedback from someone whose judgment you trust, and ask them to be as ruthlessly critical as possible.

For the remaining three questions, you should have have pre-crafted answers. If you’re at a loss for stories, it may help to sit down and just brainstorm every relevant story you can remember (for example, every bug you remember working on), and then narrow it down from a larger list.

It’s hard to practice this effectively in a vacuum, so this is a good thing to work with someone else on.

Once you’ve answered all your interviewer’s soft questions, they may ask you if you have questions for them. Tell them you’ll save your questions until the end of the interview. This will give you more time to for the actual programming part, which is the majority of where you’ll be evaluated.

Now let’s look at advice for the actual programming interview. This is going to be very whiteboarding-specific, but much of the advice applies to coding on a machine as well.

How to Approach an Interview Problem

Before you answer any (non-trivial) question, always, always, always repeat back the question in your own words. Make sure you understand exactly what they’re asking.

Expect that 10% of the time, you will misunderstand what they’re actually asking. If that’s the case, when you repeat back the question, they’ll clarify. But if you misunderstand and jump straight to working on the wrong problem, they won’t correct you for a while.

This is because if you start doing things that make no sense, your interviewer is usually just going to assume that you’re incompetent.

This is okay. Your interviewer has a hard job, because most people are incompetent, and it's their job to interview incompetent people.

Thus, if they see you initially do something stupid, they will not charitably assume that you're a smart person who is just having a bad day. That's fine; you'll gradually gain credibility over the course of the interview. But it's important to be mindful of this initial dynamic.

Thus, you should be as explicit as possible in all of your communication. Spell things out in nauseating detail. This will both eliminate any potential miscommunication between you and your interviewer, and will also quickly convince them that you are not incompetent. This is surprisingly important.

Okay, let's move on.

For any non-trivial question, I almost always start by working through a small-medium sized example on the whiteboard. This again confirms that I understand the problem and develops my intuition for how to solve it. It's also worth throwing in some edge cases here to give yourself a sense for why the problem is tricky.

Don't ever jump straight to coding. Even if you think you know the answer, don't jump straight to coding. First, explain your approach in the abstract. If there are problems with your approach, your interviewer will often point them out here; by jumping straight in, your interviewer is more likely to let you hang yourself.

Once you begin mentally working through the problem, try to keep talking. Don’t stop moving your mouth. Do your best to let the interviewer know everything that you’re thinking at all times. No matter how small or stupid or uncertain the step you’re taking in your head, just say it out loud. If you need time to think, say “give me a moment to think about this.” Then recount the results of your thinking, even if it was unfruitful.

Engineering interviews are as much about communication and thought process as they are about results. You can absolutely pass an interview on communication alone, even if you don’t get the optimal answer.

Look at your interviewer. Engage with them. Ask them questions. Make a joke. Don’t ask for help, ever. And don’t give up.

Don’t ever, ever give up. As long as they don’t give up on you, you don’t give up on yourself. Keep pushing, trying different things, and if you can’t think of anywhere productive to go, just explain why everything you can think of won’t work.

You’ll get way more help, way more leeway, and way more empathy from your interviewer if you follow this advice.

If an efficient solution doesn't immediately jump out at you, go ahead and describe the brute force solution to the problem. You'll almost always get credit for something if you at least enumerate a correct but inefficient solution. You don't have to necessarily code it (depends if they ask you to). It's usually enough to just describe how it works. Then say, "So that solves the problem in exponential time. But it's definitely not optimal. Let me see if I can find a more efficient approach." This will also help you grapple with the problem, and may give you a flash of insight on how to solve it efficiently.

A good heuristic for finding a brute-force solution is asking yourself: what's the algorithm that my brain uses to try to solve this problem if I wanted to solve it by hand?

Don't be afraid of looking stupid. It's okay to be wrong. It's okay to ask questions if you don't know what something means. Take it in stride and laugh about it. At almost all of the interviews where I received an offer, I botched at least one question. Don't let it ruffle you.

That said, if you think you know what something means but you're not 100% sure, trust your intuition. You'll usually be right. If you're not, it's okay to be corrected.

Correcting your Interviewer

Sometimes you might think your interviewer is wrong about something.

Your interviewer will definitely be sometimes wrong. But it’s much, much, much more likely that you’ll be the one who’s wrong. Think about it: you’re stressed out of your mind and coding for dear life, while your interviewer is sitting in a chair and watching the hundredth person this year struggle to solve their favorite pet problem.

As a default, always defer to your interviewer and assume that you are mistaken. Never get defensive. Figure out why you’re wrong. If you are in fact right, you should be able to prove it methodically.

Okay, so you’ve talked everything to death, now you’re ready to write code. First, say: “Would you like me to code this up?”

If your interviewer has any final misgivings about your approach, this gives them an opportunity to stop you. They may also sometimes stop you if your algorithm is acceptable but they want you to find a more optimal approach.

But usually they’ll say yes. So let’s start writing code.

Writing Code

Just kidding. You’re not ready to code yet.

Before you write anything, first verbally verify your assumptions about the input. May I assume I never get negative inputs? May I assume these are all 64-bit integers? May I assume everything in this input string is ASCII-encoded? May I assume that my input is a hash map, formatted like so? Should I raise an error for an invalid input? How should this function behave if the input is a float?

Some of the time, your interviewer will be satisfied with this attention to detail and will allow you to hand-wave some error handling for invalid inputs. But if you don't mention it at all, they might later ask you "ah, but what if you get a negative integer?" Then you'll have to erase something and squeeze in a line for error handling and waste more time.

You're almost ready to code now. Announce the language you're going to write this in. You should strongly prefer a dynamically typed language with a rich standard library like Ruby or Python. Choosing a language like Java or C++ will only handicap you in how long it takes to express the same algorithm. You will not be graded on how hardcore of a programmer you are, only on whether you solved the problem in the allotted time.

Now is also the time to mention if there are any special data structures or libraries you need. It's totally cool to say "May I assume that I have a prefix tree?" even if you don't know of a prefix tree library in your language. It’s okay to just declare the API as you go: “I’m going to assume this heap has a .pop_min method.”

And now it’s time to code.

Always start at the top left corner of the board. Board management is surprisingly important in whiteboarding interviews. Erase stuff that’s in your way and give yourself as much room as possible for the code.

More board management advice: don’t close blocks (whether curly braces or *if* blocks) until you actually get to them. Closing them early will make things messy if you have to then squeeze in unexpected lines of code or do a lot of erasing.

As a general habit, add plenty of white space around lines. It’s very, very common to have to add extra lines somewhere in a function. The more breathing room you leave, the easier that will be.

If you don’t remember the exact name of a method, your interviewer won’t care most of the time. Don’t stress about it. Just come up with a reasonably close-enough name for it. If they ask what it does, just say: “I could be mistaken about the name here, but this method creates an inverted version of a hash map so the unique values now point to their keys.” Most interviewers will be satisfied with that.

Name your variables descriptively, but keep them short. It’s going to suck to write long variable names repeatedly without autocomplete.

As you’re writing, abstract away as much logic as you can into helper functions. This is a tremendously useful whiteboarding optimization.

Just invoke the helper functions as you go and don’t feel the need to write their declarations immediately. Just say: “I’ll write this function afterward, but this just returns a hash table mapping each character in the string to its frequency.”

Go ahead and keep working through your main function, and worry about the helpers later. This will keep you in your flow and will make it easier to read and reason about your code.

After you finish and establish the characteristics of your primary function’s logic, oftentimes your interviewer will trust that you can write some of the helpers and won’t want to watch you code them.

Just say: “so this helper function does [blah blah blah]. Do you want me to write this as well?” If they say yes, go ahead and write all your helper functions out. But this at least gives them the chance to say no. This optimization alone ended up saving me a lot of time across my interviews.

Finally, once you’ve finished all of the code, review everything. Just say “okay, let me review this now.” Read your code from top to bottom and ensure it makes sense. Follow it as though you had a debugger, actually tracing the path and transformations of an input. Fix any obvious mistakes. Then start inspecting for edge cases.

Most common edge cases: the input is null/empty, the input is 0 or size 0, the input is 1 or size 1, or the input contains duplicates. Try to suss out the assumptions you’re making and see how they could be violated.

Once you’re satisfied that your solution accounts for edge cases, then and only then turn to your interviewer and announce “I believe this solves the problem.” Make it clear that you believe you’re done.

Now go ahead and analyze the algorithm. Announce the time complexity and space complexity of your solution. Talk about any tradeoffs you might be able to make. “If you wanted to take O(n^3) time, then you could just memoize all the answers and return them in O(1) time for subsequent queries. But that comes at a big up-front time and space cost.”

Assorted Heuristics

This is kind of dumb, but I thought I’d include this because people seem to find it helpful.

If you’re blanking during an interview, try these rules of thumb, roughly in order of usefulness:

1. Try hash maps.
2. If your problem fundamentally breaks down to searching for something, consider binary search.
3. Try sorting your input.
4. Can you break the problem down into sub-problems? Does solving the problem for size (N – 1) make solving it for size N any easier? If so, try to solve recursively and/or with dynamic programming.
5. If you have a lot of strings, try putting them in a prefix tree.
6. Any time you repeatedly have to take the min or max of a dynamic collection, think heaps. (If you don’t need to insert random elements, prefer a sorted array.)

Okay, so the interview is finally over. Now it’s time to ask your interviewer questions. This part, too, is surprisingly important.

Grill Your Interviewers

A common trope people throw around is “you’re interviewing them as much as they’re interviewing you.” I say if that’s true, then put them through the goddamn wringer.

Ask them why they decided to join the company. Ask them what they think the company could improve at. Ask them about a time that they messed up and how it was handled. Ask them how they see themselves growing at this company in the next few years. Ask them what they wish someone would have told them before they joined. Ask them if they ever think about leaving, and if they were to leave, where they would go.

To be clear, these are not necessarily questions about the company. They are questions *about your interviewer*.

I advocate grilling your interviewer for a few reasons:

- Your interviewer is likely representative of the people you'll be working with if you join. You should learn as much as possible about the kind of people who work here.
- It's going to give you a lot more insight into the company than any generic question will.
- It gives you some momentum in the interview. Instead of constantly being on the defensive, by asking tough questions, you get some time to breathe while your interviewer reacts to your questions.
- It's also just more interesting!

That said, you will probably have some burning questions you actually want to ask your specific interviewer, like "How did you get into data engineering?” or “What are your feelings on the company’s transition toward service-oriented architecture?”

That leads me to my second piece of advice when it comes to grilling interviewers.

Most of the time you won’t have snappy, specific questions to ask. So memorize a set of questions and ask them again and again. Not just for different companies—literally ask successive interviewers at the same company the exact same thing.

Why?

Because for one, interviewers usually solicit questions at the end of an interview. By then there’s a good chance you’ll be psychologically drained and won’t feel like coming up with novel, incisive questions.

(Never, ever, ever say “I have no questions.” Be interested!)

Second, interviewers don’t compare notes on which questions a candidate asked them. So if you have a good set of questions, nobody at the company will know you asked the same questions.

Third, even if they did, they wouldn’t care. Good questions are still good questions, and it’s worthwhile to hear different people’s answers to the same good question. I’ve learned a lot about different companies’ dynamics this way.

Be Excited

This is the final point I want to make, and it applies to your entire interview process.

Be excited about the company. It’s trite, but it makes a huge difference. Be excited to interview. Be excited to learn about what your interviewer does and the prospect of getting to work with them. Tell them how much you like the people you met and that you dig the culture.

Interviewing is surprisingly like dating. Your interviewer is disproportionately more likely to be excited about you if you’re excited about them. That positivity will make them more want to work with you, and even go to bat for you when it comes decision time.

Not only that, but excitement is a signal that you actually want to do the job they’re hiring you for, and that you’re going to work your ass off to do it well. If your interview performance was anywhere near the edge, your excitement may end up pushing you over the finish line.

You’ll likely be interviewing at different companies, but you should always talk about what makes this company unique and captivating. If you’re not sure, just ask your interviewers—what makes them love working here? You can then reflect back that answer.

At the same time, you don’t want to appear desperate. I liked using the phrase “looking for good mutual fit.” You want to be discriminating, but you also want to seem winnable if they make you the right offer. If you seem outright disinterested in the company, then they probably won’t want to make you an offer even if you pass their technical bar.

Companies do vary significantly in culture. But a lot of “culture fit” really just comes down to a simple question: would your interviewers actually want to be your colleague?

It’s much easier for them to say yes if you sound excited. So make it easy for them.

That concludes parts 1 and 2 of this guide. I’ll be following up with the final part of this guide, focusing on post-offer negotiation.