Get that Job at Google

Mental Prep

So! You're a hotshot programmer with a long list of accomplishments. Time to forget about all that and focus on interview survival.

You should go in humble, open-minded, and focused.

If you come across as arrogant, then people will question whether they want to work with you. The best way to appear arrogant is to question the validity of the interviewer's question – it really ticks them off, as I pointed out earlier on. Remember how I said you can't tell an interviewer how to interview? Well, that's *especially* true if you're a candidate.

So don't ask: "gosh, are algorithms really all that important? do you ever need to do that kind of thing in real life? I've never had to do that kind of stuff." You'll just get rejected, so don't say that kind of thing. Treat every question as legitimate, even if you are frustrated that you don't know the answer.

Feel free to ask for help or hints if you're stuck. Some interviewers take points off for that, but occasionally it will get you past some hurdle and give you a good performance on what would have otherwise been a horrible stony half-hour silence.

Don't say "choo choo choo" when you're "thinking".

Don't try to change the subject and answer a different question. Don't try to divert the interviewer from asking you a question by telling war stories. Don't try to bluff your interviewer. You should *focus* on each problem they're giving you and make your best effort to answer it fully.

Some interviewers will not ask you to write code, but they will *expect* you to start writing code on the whiteboard at some point during your answer. They will give you hints but won't necessarily come right out and say: "I want you to write some code on the board now." If in doubt, you should ask them if they would like to see code.

Interviewers have vastly different expectations about code. I personally don't care about syntax (unless you write something that could obviously never work in any programming language, at which point I will dive in and verify that you are not, in fact, a circus clown and that it was an honest mistake). But some interviewers are really picky about syntax, and some will even silently mark you down for missing a semicolon or a curly brace,*without telling you*. I think of these interviewers as – well, it's a technical term that rhymes with "bass soles", but they think of themselves as brilliant technical evaluators, and there's no way to tell them otherwise.

So ask. Ask if they care about syntax, and if they do, try to get it right. Look over your code carefully from different angles and distances. Pretend it's someone else's code and you're tasked with finding bugs in it. You'd be amazed at what you can miss when you're standing 2 feet from a whiteboard with an interviewer staring at your shoulder blades.

It's OK (and highly encouraged) to ask a few clarifying questions, and occasionally verify with the interviewer that you're on the track they want you to be on. Some interviewers will mark you down if you just jump up and start coding, *even if you get the code right*. They'll say you didn't think carefully first, and you're one of those "let's not do any design" type cowboys. So even if you think you know the answer to the problem, ask some questions and talk about the approach you'll take a little before diving in.

On the flip side, don't take too long before actually solving the problem, or some interviewers will give you a delay-of-game penalty. Try to move (and write) quickly, since often interviewers want to get through more than one question during the interview, and if you solve the first one too slowly then they'll be out of time. They'll mark you down because they couldn't get a full picture of your skills. The benefit of the doubt is rarely given in interviewing.

One last non-technical tip: bring your own whiteboard dry-erase markers. They sell pencil-thin ones at office supply stores, whereas most companies (including Google) tend to stock the fat kind. The thin ones turn your whiteboard from a 480i standard-definition tube into a 58-inch 1080p HD plasma screen. You need all the help you can get, and free whiteboard space is a real blessing.

You should also practice whiteboard space-management skills, such as not starting on the right and coding down into the lower-right corner in Teeny Unreadable Font. Your interviewer will not be impressed. Amusingly, although it always irks me when people do this, I did it during my interviews, too. Just be aware of it!

Oh, and don't let the marker dry out while you're standing there waving it. I'm tellin' ya: you want minimal distractions during the interview, and that one is surprisingly common.

OK, that should be good for non-tech tips. On to X, for some value of X! Don't stab me!

Tech Prep Tips

The best tip is: go get a computer science degree. The more computer science you have, the better. You don't have to have a CS degree, but it helps. It doesn't have to be an advanced degree, but that helps too.

However, you're probably thinking of applying to Google a little sooner than 2 to 8 years from now, so here are some shorter-term tips for you.

Algorithm Complexity: you need to know Big-O. It's a must. If you struggle with basic big-O complexity analysis, then you are almost guaranteed not to get hired. It's, like, one chapter in the beginning of one theory of computation book, so just go read it. You can do it.

Sorting: know how to sort. Don't do bubble-sort. You should know the details of at least one n*log(n) sorting algorithm, preferably two (say, quicksort and merge sort). Merge sort can be highly useful in situations where quicksort is impractical, so take a look at it.

For God's sake, don't try sorting a linked list during the interview.

Hashtables: hashtables are arguably the single most important data structure known to mankind. You *absolutely have to know how they work*. Again, it's like one chapter in one data structures book, so just go read about them. You should be able to implement one using only arrays in your favorite language, in about the space of one interview.

Trees: you should know about trees. I'm tellin' ya: this is basic stuff, and it's embarrassing to bring it up, but some of you out there don't know basic tree construction, traversal and manipulation algorithms. You should be familiar with binary trees, n-ary trees, and trie-trees at the very *very* least. Trees are probably the best source of practice problems for your long-term warmup exercises.

You should be familiar with at least one flavor of balanced binary tree, whether it's a red/black tree, a splay tree or an AVL tree. You should actually know how it's implemented.

You should know about tree traversal algorithms: BFS and DFS, and know the difference between inorder, postorder and preorder.

You might not use trees much day-to-day, but if so, it's because you're avoiding tree problems. You won't need to do that anymore once you know how they work. Study up!

Graphs

Graphs are, like, really *really* important. More than you think. Even if you already think they're important, it's probably more than you think.

There are three basic ways to represent a graph in memory (objects and pointers, matrix, and adjacency list), and you should familiarize yourself with each representation and its pros and cons.

You should know the basic graph traversal algorithms: breadth-first search and depth-first search. You should know their computational complexity, their tradeoffs, and how to implement them in real code.

You should try to study up on fancier algorithms, such as Dijkstra and A*, if you get a chance. They're really great for just about anything, from game programming to distributed computing to you name it. You should know them.

Whenever someone gives you a problem, *think graphs*. They are the most fundamental and flexible way of representing any kind of a relationship, so it's about a 50-50 shot that any interesting design problem has a graph involved in it. Make absolutely sure you can't think of a way to solve it using graphs before moving on to other solution types. This tip is important!

Other data structures

You should study up on as many other data structures and algorithms as you can fit in that big noggin of yours. You should especially know about the most famous classes of NP-complete problems, such as traveling salesman and the knapsack problem, and be able to recognize them when an interviewer asks you them in disguise.

You should find out what NP-complete means.

Basically, hit that data structures book hard, and try to retain as much of it as you can, and you can't go wrong.

Math

Some interviewers ask basic discrete math questions. This is more prevalent at Google than at other places I've been, and I consider it a Good Thing, even though I'm not particularly good at discrete math. We're surrounded by counting problems, probability problems, and other Discrete Math 101 situations, and those innumerate among us blithely hack around them without knowing what we're doing.

Don't get mad if the interviewer asks math questions. Do your best. Your best will be a heck of a lot better if you spend some time before the interview refreshing your memory on (or teaching yourself) the essentials of combinatorics and probability. You should be familiar with n-choose-k problems and their ilk – the more the better.

I know, I know, you're short on time. But this tip can really help make the difference between a "we're not sure" and a "let's hire her". And it's actually not all that bad – discrete math doesn't use much of the high-school math you studied and forgot. It starts back with elementary-school math and builds up from there, so you can probably pick up what you need for interviews in a couple of days of intense study.

Sadly, I don't have a good recommendation for a Discrete Math book, so if you do, please mention it in the comments. Thanks.

Operating Systems

This is just a plug, from me, for you to know about processes, threads and concurrency issues. A lot of interviewers ask about that stuff, and it's pretty fundamental, so you should know it. Know about locks and mutexes and semaphores and monitors and how they work. Know about deadlock and livelock and how to avoid them. Know what resources a processes needs, and a thread needs, and how context switching works, and how it's initiated by the operating system and underlying hardware. Know a little about scheduling. The world is rapidly moving towards multi-core, and you'll be a dinosaur in a real hurry if you don't understand the fundamentals of "modern" (which is to say, "kinda broken") concurrency constructs.

The best, most practical book I've ever personally read on the subject is Doug Lea'sConcurrent Programming in Java. It got me the most bang per page. There are obviously lots of other books on concurrency. I'd avoid the academic ones and focus on the practical stuff, since it's most likely to get asked in interviews.

Coding

You should know at least one programming language really well, and it should*preferably* be C++ or Java. C# is OK too, since it's pretty similar to Java. You will be expected to write some code in at least some of your interviews. You will be expected to know a fair amount of detail about your favorite programming language.

Other Stuff

Because of the rules I outlined above, it's still possible that you'll get Interviewer A, and none of the stuff you've studied from these tips will be directly useful (except being warmed up.) If so, just do your best. Worst case, you can always come back in 6-12 months, right? Might seem like a long time, but I assure you it will go by in a flash.

The stuff I've covered is actually mostly red-flags: stuff that really worries people if you don't know it. The discrete math is potentially optional, but somewhat risky if you don't know the first thing about it. Everything else I've mentioned you should know cold, and then you'll at least be prepped for the baseline interview level. It could be a lot harder than that, depending on the interviewer, or it could be easy.

It just depends on how lucky you are. Are you feeling lucky? Then give it a try!