Interview Cake

Coding Interview Tips

How to get better at technical interviews without practicing

Chitchat like a pro.

Before diving into code, most interviewers like to chitchat about your background. They're looking for:

- **Metacognition about coding**. Do you think about how to code well?
- **Ownership/leadership**. Do you see your work through to completion? Do you fix things that aren't quite right, even if you don't have to?
- **Communication**. Would chatting with you about a technical problem be useful or painful?

You should have at least one:

- example of an interesting technical problem you solved
- example of an interpersonal conflict you overcame
- example of leadership or ownership
- story about what you should have done differently in a past project
- piece of trivia about your favorite language, and something you do and don't like about said language
- question about the company's product/business
- question about the company's engineering strategy (testing, Scrum, etc)

Nerd out about stuff. Show you're proud of what you've done, you're amped about what they're doing, and you have opinions about languages and workflows.

Communicate.

Once you get into the coding questions, communication is key. A candidate who needed some help along the way but communicated clearly can be even better than a candidate who breezed through the question.

Understand what kind of problem it is. There are two types of problems:

1. **Coding**. The interviewer wants to see you write clean, efficient code for a problem.
2. **Chitchat**. The interviewer just wants you to talk about something. These questions are often either (1) high-level system design ("How would you build a Twitter clone?") or (2) trivia ("What is hoisting in Javascript?"). Sometimes the trivia is a lead-in for a "real" question e.g., "How quickly can we sort a list of integers? Good, now suppose instead of integers we had . . ."

If you start writing code and the interviewer just wanted a quick chitchat answer before moving on to the "real" question, they'll get frustrated. Just ask, "Should we write code for this?"

Make it feel like you're on a team. The interviewer wants to know what it feels like to work through a problem with you, so make the interview feel collaborative. Use "we" instead of "I," as in, "If we did a breadth-first search we'd get an answer in $O(n)$ time." If you get to choose between coding on paper and coding on a whiteboard, always choose the whiteboard. That way you'll be situated next to the interviewer, facing the problem (rather than across from her at a table).

Think out loud. Seriously. Say, "Let's try doing it this way—not sure yet if it'll work." If you're stuck, just say what you're thinking. Say what might work. Say what you thought could work and why it doesn't work. This also goes for trivial chitchat questions. When asked to explain Javascript closures, "It's something to do with scope and putting stuff in a function" will probably get you 90% credit.

Say you don't know. If you're touching on a *fact* (e.g., language-specific trivia, a hairy bit of runtime analysis), don't try to appear to know something you don't. Instead, say "*I'm not sure, but I'd guess $thing, because...*". The *because* can involve ruling out other options by showing they have nonsensical implications, or pulling examples from other languages or other problems.

Slow the eff down. Don't confidently blurt out an answer right away. If it's right you'll still have to explain it, and if it's wrong you'll seem reckless. You don't win anything for speed and you're more likely to annoy your interviewer by cutting her off or appearing to jump to conclusions.

Get unstuck.

Sometimes you'll get stuck. Relax. It doesn't mean you've failed. Keep in mind that the interviewer usually cares more about your ability to cleverly poke the problem from a few different angles than your ability to stumble into the correct answer. When hope seems lost, keep poking.

Draw pictures. Don't waste time trying to think in your head—think on the board. Draw a couple different test inputs. Draw how you would get the desired output by hand. Then think about translating your approach into code.

Solve a simpler version of the problem. Not sure how to find the 4th largest item in the set? Think about how to find the 1st largest item and see if you can adapt that approach.

Write a naive, inefficient solution and optimize it later. Use brute force. Do whatever it takes to get *some kind* of answer.

Think out loud more. Say what you know. Say what you thought might work and why it won't work. You might realize it actually does work, or a modified version does. Or you might get a hint.

Wait for a hint. Don't stare at your interviewer expectantly, but do take a *brief* second to "think"—your interviewer might have already decided to give you a hint and is just waiting to avoid interrupting.

Think about the bounds on space and runtime. If you're not sure if you can optimize your solution, think about it out loud. For example:

- "I have to at least look at all of the items, so I can't do better than $O(n)$."
- "The brute force approach is to test all possibilities, which is $O(n^2)$."
- "The answer will contain n^2 items, so I must at least spend that amount of time."

Get your thoughts down.

It's easy to trip over yourself. Focus on getting your thoughts down first and worry about the details at the end.

Call a helper function and keep moving. If you can't immediately think of how to implement some part of your algorithm, big or small, just skip over it. Write a call to a reasonably-named helper function, say "this will do X" and keep going. If the helper function is trivial, you might even get away with never implementing it.

Don't worry about syntax. Just breeze through it. Revert to English if you have to. Just say you'll get back to it.

Leave yourself plenty of room. You may need to add code or notes in between lines later. Start at the top of the board and leave a blank line between each line.

Save off-by-one checking for the end. Don't worry about whether your for loop should have "<" or "<=." Write a checkmark to remind yourself to check it at the end. Just get the general algorithm down.

Use descriptive variable names. This will take time, but it will prevent you from losing track of what your code is doing. Use `names_to_phone_numbers` instead of `nums`. Imply the type in the name. Functions returning booleans should start with "`is_*`". Vars that hold a list should end with "`s`." Choose standards that make sense to you and stick with them.

Clean up when you're done.

Walk through your solution by hand, out loud, with an example input. Actually *write down* what values the variables hold as the program is running—you don't win any brownie points for doing it in your head. This'll help you find bugs and clear up confusion your interviewer might have about what you're doing.

Look for off-by-one errors. Should your for loop use a "<=" instead of a "<"?

Test edge cases. These might include empty sets, single-item sets, or negative numbers. Bonus: mention unit tests!

Don't be boring. Some interviewers won't care about these cleanup steps. If you're unsure, say something like, "Then I'd usually check the code against some edge cases—should we do that next?"

Practice.

In the end, there's no substitute for running practice questions.

Actually write code with pen and paper. Be honest with yourself. It'll probably feel awkward at first. Good. You want to get over that awkwardness now so you're not fumbling when it's time for the real interview.

Our practice questions mirror the interview process by offering hints when you're stuck and nudges if your algorithm could be improved.

Try some questions now ➜
