Tricks For Getting Unstuck During a Coding Interview

Getting stuck during a coding interview is rough.

If you weren't in an interview, you might take a break or ask Google for help. But the clock is ticking, and you don't have Google.

You just have an empty whiteboard, a smelly marker, and an interviewer who's looking at you expectantly. And all you can think about is how stuck you are.

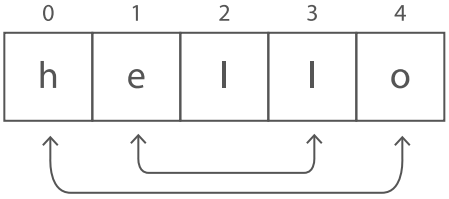
You need a lifeline for these moments—like a little box that says "In Case of Emergency, Break Glass."

Inside that glass box? A list of tricks for getting unstuck. Here's that list of tricks.

When you're stuck on getting started

1) Write a sample input on the whiteboard and turn it into the correct output "by hand." Notice the *process* you use. Look for patterns, and think about how to implement your process in code.

Trying to reverse a string? Write “hello” on the board. Reverse it “by hand”—draw arrows from each character’s current position to its desired position.



Notice the pattern: it looks like we’re *swapping* pairs of characters, starting from the outside and moving in. Now we’re halfway to an algorithm.

2) Solve a simpler version of the problem. Remove or simplify one of the requirements of the problem. Once you have a solution, see if you can adapt that approach for the original question.

Trying to find the k-largest element in a set? Walk through finding the *largest* element, then the *second largest*, then the *third largest*. Generalizing from there to find the k-largest isn’t so bad.

3) Start with an inefficient solution. Even if it feels *stupidly inefficient*, it’s often helpful to start with *something* that’ll return the right answer. From there, you just have to optimize your solution. Explain to your interviewer that this is only your first idea, and that you suspect there are faster solutions.

Suppose you were given two lists of sorted numbers and asked to find the median of both lists combined. It’s messy, but you could simply:

1. Concatenate the arrays together into a new array.
2. Sort the new array.
3. Return the value at the middle index.

> Notice that you could’ve *also* arrived at this algorithm by using trick (2): Solve a simpler version of the problem. “How would I find the median of *one* sorted list of numbers? Just grab the item at the middle index. Now, can I adapt that approach for getting the median of *two* sorted lists?”

When you’re stuck on finding optimizations

1) Look for repeat work. If your current solution goes through the same data multiple times, you're doing unnecessary repeat work. See if you can save time by looking through the data just once.

Say that inside one of your loops, there's a brute-force operation to find an element in an array. You're repeatedly looking through items that you don't have to. Instead, you could convert the array to a lookup table (/concept/hash-map) to dramatically improve your runtime.

2) Look for hints in the specifics of the problem. Is the input array sorted? Is the binary tree (/concept/binary-tree) balanced? Details like this can carry huge hints about the solution. If it didn't matter, your interviewer wouldn't have brought it up. It's a strong sign that the best solution to the problem exploits it.

Suppose you're asked to find the first occurrence of a number in a sorted array. The fact that the array is *sorted* is a strong hint—take advantage of that fact by using a binary search (/concept/binary-search).

> Sometimes interviewers leave the question deliberately vague because they want you to *ask questions* to unearth these important tidbits of context. So ask some questions at the beginning of the problem.

3) Throw some data structures (/article/data-structures-computer-science) at the problem. Can you save time by using the fast lookups of a hash table (/concept/hash-map)? Can you express the relationships between data points as a graph (/concept/graph)? Look at the requirements of the problem and ask yourself if there's a data structure that has those properties.

4) Establish bounds on space and runtime. Think *out loud* about the parameters of the problem. Try to get a sense for how fast your algorithm *could possibly* be:

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

When All Else Fails

1) Make it clear where you are. State what you know, what you're trying to do, and highlight the gap between the two. The clearer you are in expressing *exactly* where you're stuck, the easier it is for your interviewer to help you.

2) Pay attention to your interviewer. If she asks a question about something you just said, there's probably a hint buried in there. Don't worry about losing your train of thought—drop what you're doing and dig into her question.

Relax. You're *supposed* to get stuck.

Interviewers choose hard problems on purpose. They want to see how you poke at a problem you don't immediately know how to solve.

Seriously. If you *don't* get stuck and just breeze through the problem, your interviewer's evaluation might just say "Didn't get a good read on candidate's problem-solving process—maybe she'd already seen this interview question before?"

On the other hand, if you *do* get stuck, use one of these tricks to get unstuck, and communicate clearly with your interviewer throughout...*that's* how you get an evaluation like, "Great problem-solving skills. Hire."

← course home (/table-of-contents)

Next up: The 24 Hours Before Your Interview → (/24-hours-before-onsite-whiteboard-coding-interview?course=fc1&section=interview-tips)

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.