
# Technical Interviews & Assessments

***Check out these articles from our Blog – they'll be particularly useful for you in preparing for this step of the interview process*** 😎

- How to Ace Your Next Coding Interview
- How To Smash a Microsoft Interview
- What Your Future Employer Is Looking For in a Technical Interview
- What I Learned From Interviewing For A New Job

***We've also gathered some tips from another source that we think are pretty cool!*** 💻

Coding interviews are tough. But fortunately, there's a tried and proven method to get better at them. With a combination of studying, practicing questions and doing mock interviews, getting that dream job can become a reality.

- Decide on a programming language
- Study CS fundamentals
- Practice solving algorithm questions
- Internalize the Do's and Don'ts of interviews
- Know what signals and behaviors interviewers are looking out for
- Practice doing mock interviews
- Interview successfully to get the job

## Picking a Language

Before anything else, you need to pick a programming language to do your interviews in. Most companies will let you code in any language you want.

### Use a language you are familiar with

Most of the time, it is recommended that you use a language that you are extremely familiar with rather than picking up a new language just for doing interviews because the company uses that language heavily or just because you want to show that you are trendy.

If you are under time constraints, picking up a new language just for interviewing is hardly a good idea. Languages take time to master and if you are already spending most of your time and effort on mastering algorithms, there is barely spare effort left for mastering a new language. If you are familiar with using one of the mainstream languages, there isn't a strong reason to learn a new language just for interviewing.

## Study and Practice

### Recap CS fundamentals

If you have been out of college for a while, it is highly advisable to review CS fundamentals — Algorithms and Data Structures. Personally, I prefer to review as I practice, so I scan through my college notes and review the various algorithms as I work on algorithm problems from LeetCode and Cracking the Coding Interview.

This interviews repository by Kevin Naughton Jr. served as a quick refresher for me.
The Medium publication basecs by Vaidehi Joshi is also a great and light-hearted resource to recap on the various data structures and algorithms.

If you are interested in how data structures are implemented, check out Lago, a Data Structures and Algorithms library for JavaScript. It is pretty much still WIP but I intend to make it into a library that is able to be used in production and also a reference resource for revising Data Structures and Algorithms.

### Mastery through practice

Next, gain familiarity and mastery of the algorithms and data structures in your chosen programming language.

Practice coding algorithms using your chosen language. While Cracking the Coding Interview is a good resource for practice, I prefer being able to type code, run it and get instant feedback. There are various Online Judges such as LeetCode, HackerRank and CodeForces for you to

practice questions online and get used to the language. From experience, LeetCode questions are the most similar to the kind of questions being asked in interviews whereas HackerRank and CodeForces questions resemble competitive programming questions.

If you practice enough LeetCode questions, there is a good chance that you would have seen/done your actual interview question (or some variant) on LeetCode before. If you are more of a visual person, Coderust explains the common algorithm questions through step-by-step visualizations which makes understanding the solutions much easier.

## During the Coding Interview

In a coding interview, you will be given a technical question by the interviewer, write code in a real-time collaborative editor (phone screen) or on a whiteboard (on-site) to solve the problem within 30–45 minutes. ***You might also get an online technical assessment - keep in mind that for these kinds of remote technical tests, not all of the following tips will be useful.***

Your interviewer will be looking out for signals that you fit the requirements of the role and it is up to you to display those signals to them. Initially it may feel weird to be talking while you are coding as most programmers do not have the habit of explaining out loud as they are typing code. However, it is hard for the interviewer to know what you are thinking just by looking at the code that you type. If you communicate your approach to the interviewer before you start coding, you can validate your approach with them and the both of you can agree upon an acceptable approach.

### Before the interview

For phone screens/remote interviews, prepare paper and pen/pencil to jot down and visualize stuff. If you are given a question on trees and graphs, it usually helps if you draw out some examples of the data structure given in the question.

Use earphones and make sure you are in a quiet environment. You definitely do not want to be holding a phone in one hand and only be able to type with the other. Try avoiding using speakers because if the echo is bad, communication is harder and repeating of words will just result in loss of valuable time.

### Upon receiving the question

Many candidates jump into coding the moment they hear the question. That is usually a big mistake. Take a moment to repeat the question back at the interviewer and make sure that you understand exactly what they are asking. Repeating back/rephrasing the question will reduce chances of miscommunication.

Always seek clarification about the question upon hearing it even if you think it is clear to you. You might discover something that you have missed out and it also sends a signal to the interviewer that you are a careful person who pays attention to details. Some interviewers deliberately omit important details to see if you ask the right questions.

***Some common questions you can ask:***

- How big is the size of the input?
- How big is the range of values?
- What kind of values are there? Are there negative numbers? Floating points? Will there be empty inputs?
- Are there duplicates within the input?
- What are some extreme cases of the input?
- Can I destroy the original array/graph/data structure?
- How is the input stored? If you are given a dictionary of words, is it a list of strings or a Trie?

After you have sufficiently clarified the scope and intention of the problem, explain your high level approach to the interviewer even if it is a naive solution. If you are stuck, consider various approaches and explain out loud why it will/will not work. Sometimes your interviewer might drop hints and lead you towards the right path.

Start with a brute force approach, communicate it to the interviewer, explain the time and space complexity and why it is bad. It is unlikely that the brute force approach will be one that you will be coding. At this point, the interviewer will usually pop the dreaded "Can we do better?" question, meaning that they are looking for a more optimal approach. In my opinion, this is usually the hardest part of the interview. In general, look for repeated work and try to optimize them by potentially caching the calculated result somewhere and reference it later, rather than having to compute it all over again. There are some tips on tackling topic-specific questions that I dive into details below.

Only start coding after you and your interviewer have agreed on an approach and they have given you the green light.

**What to do when stuck**

Getting stuck during coding interviews is extremely common. But do not worry, that is part of the process and is a test of your problem solving abilities. Here are some tips to try out when you are stuck:

- Talk through what you initially thought might work and explain why it doesn't
- This can help guide you on the right track by avoiding the pitfalls
- Come up with more test cases and write them down
- A pattern may emerge

- Think about how you would solve it without a program
- You may spot a pattern and come up with a general algorithm for it
- Recall past questions related to the topic, what similar questions in the past have you encountered and what techniques did you use?
- Enumerate through the common data structures and whether they can be applied to the question
- Dictionaries/maps are extremely common in making algorithms more efficient
- Look out for repeated work and determine if you can cache those computations
- Trade off memory for speed

**While coding**

Write your code with good coding style. Reading code written by others is usually not an enjoyable task. Reading horribly-formatted code by others makes it worse. Your goal is to make your interviewer understand the code you have written so that they can quickly evaluate if your code does what you say it does and whether it solves the given problem. Use clear variable names, avoid single letter names unless they are for iteration. However, if you are coding on a whiteboard, you might not want to use extremely verbose variable names for the sake of reducing the amount you have to write. Abbreviations are usually fine if you explain what it means beforehand.

Always be explaining what you are currently writing/typing to the interviewer. This is not about literally reading out what you are typing to the interviewer. Talk about the section of the code you are currently implementing at a higher level, explain why it is written as such and what it is trying to achieve.

While coding, if you find yourself copying and pasting code, consider whether it is necessary. If you find yourself copying and pasting one large chunk of code spanning multiple lines, it is usually an indicator that you can refactor by extracting those lines into a function and defining parameters for the differences in them. If it is just a single line you copied, usually it is fine. Do remember to change the respective variables in your copied line of code where relevant. Copy-paste errors are a common source of bugs even in day-to-day coding!

**After coding**

After you have finished coding, do not immediately announce to the interviewer that you are done. In most cases, your code is usually not perfect and contains some bugs or syntax errors. What you need to do now is to review your code.

Firstly, look through your code from start to finish as if it is the first time you are seeing it, as if it was written by someone else and you are trying to spot bugs in it. That's exactly what your interviewer will be doing. Look through and fix any minor issues you may find.

Next, come up with small test cases and step through the code (not your algorithm!) with those sample input. What interviewers usually do after you have finished coding would be to get you to write tests. It is a huge plus if you write tests for your code even before they prompt you to do so. You should be emulating a debugger when stepping through and jot down or say out the values of the important variables as you step through the lines of code.

If there are huge duplicated chunks of code in your solution, it would be a good chance to refactor it and demonstrate to the interviewer that you are one who values code quality. Also look out for places where you can do short-circuit evaluation.

Lastly, give the time/space complexity of your code and explain why it is such. You can even annotate certain chunks of your code with the various time/space complexities to demonstrate your understanding of your code and the APIs of your chosen programming language. Explain any trade-offs in your current approach vs alternative approaches, possibly in terms of time/space.

If your interviewer is happy with the solution, the interview usually ends here. It is also not uncommon that the interviewer asks you extension questions, such as how you would handle the problem if the whole input is too large to fit into memory, or if the input arrives as a stream. This is a common follow-up question at Google where they care a lot about scale. The answer is usually a divide-and-conquer approach — perform distributed processing of the data and only read certain chunks of the input from disk into memory, write the output back to disk, and combine them later on.

---

Source:
https://yangshun.github.io/tech-interview-handbook/