If the company provides lunch, you might also have a lunch session with an employee where you can find out more about the company culture.

Pick a programming language

With your resume done, the next step of your software engineering interview journey is a simple one and won't take long — decide on a programming language. Unless you're interviewing for a specialist position like mobile or front end where there are domain-specific languages, you should be free to use any language you want for the algorithmic coding interviews.

Most of the time, you'd already have one in mind — pick the one you use the most and you're the most comfortable with. The most common programming languages used for coding interviews are Python, Java, C++, and JavaScript. I wouldn't recommend learning an entirely new language just for coding interviews as it takes a while (few weeks at least on average) to become proficient enough in a language to wield it comfortably in an interview setting, which is already stressful enough on its own. My personal programming language of choice is Python because of how terse it is and the functions/data structures the standard library provides.

Read more on programming languages for coding interviews: Picking a programming language.

Study and practice for coding interviews

The next and most important step is to practice solving algorithm questions in your chosen programming language. While *Cracking the Coding Interview* is a great resource, I prefer learning by actually solving problems.

There are many platforms that can be used for this — such as LeetCode, HackerRank and CodeForces. From my personal experience, LeetCode questions are most suitable for interview preparation whereas HackerRank and CodeForces are more for competitive programming.

However, LeetCode has thousands of questions and it can be daunting to know where to begin, or how to structure your practice. I have provided recommended preparation plans and also structured resources below.

Coding interview study plan

The recommended time period to set aside for coding interview preparation is 3 months (11 hours a week or roughly 2-3 hours a day) for a more holistic preparation. I shared my 3 month study plan here, which provides a list of coding interview topics with resources and practice questions that you should work through in order of priority every week. I will also be adding content on recommended 1 month and 1 week study plans soon.

If you have less than 3 months to prepare, you can generate your own study plans using the Grind 75 tool (built by me) which generates recommended study plans for coding interviews based on the time you have left. The algorithm behind it includes a ranking of questions by priority and also a balance between breadth and depth of topics covered.

Resources to use in your practice

In the market, there are plenty of resources vying for your attention, plenty of them just vying for your money but not providing any value. If I had to prioritize — these are the top coding interview preparation resources I would use in tandem:

- Grokking the Coding Interview: Patterns for Coding Questions
- AlgoMonster
- My (free) coding interview best practices guide

  If you have read the coding interview evaluation rubric used at top tech companies, you may be overwhelmed by the number of items evaluated and how to demonstrate hire behaviors consistently.

  This coding interview best practices guide synthesizes actionable recommendations of what to do before, during and after your coding interviews to demonstrate hire signals.

  I recommend to internalize and use the guide as an accompaniment while you practice coding interview questions — to ensure that you cultivate good habits and muscle memory with regards to interviews right from the beginning.

- My (free) coding interview techniques guide

  Is there a structured method to increase your chances of finding a good solution to the coding interview question? How about optimizing your approach's time and space complexity? My coding interview techniques guide teaches you a few techniques for handling questions that you have never encountered before — such as problem visualizing, solving by hand, breaking the problem into subproblems, etc.

- My (free) algorithms study guide

  I'm not sure if these would qualify as an in-depth guide — they are more like 1-page "study cheatsheets" of the best resources to study, best LeetCode questions to practice and the things to remember. However, they ensure you cover all the most important grounds, especially when you have no time. Because these are also the notes that helped me clinch top tech offers — they definitely work.

  For more tips on coding interview preparation, refer to my full coding interview preparation guide here.

Try out mock coding interviews

Coding right in front of your interviewer can be a nerve-wracking experience especially if you have never done it before — which is why getting hands-on experience is so important.

interviewing.io is currently the best mock technical interview resource in the market. It allows you to book mock coding interviews with real Google and Facebook engineers, albeit anonymously. You could even book interviews for specific roles like Mobile, Front End, Engineering Management. Even better — if you want to have an easier transition into real world coding interview — you could view recorded interviews and see what phone interviews are like.

Moreover, if you were to do well on your mock interviews, you will be able to unlock the "jobs page" which allows you to book interviews directly with top companies like Uber, Lyft, Quora, Asana and more. I've used interviewing.io both as an interviewer and an interviewee and found the experience to be excellent.

Prepare for the system design interview

If you are a mid or senior-level candidate, you may expect system design questions as part of your technical interview. They aren't covered adequately by LeetCode and good resources are still harder to come by.

The objective of system design interviews is to evaluate a candidate's skill at designing real-world software systems involving multiple components.

Some of the best system design interview preparation resources include:

1. ByteByteGo — This is a new System Design course by Alex Xu, author of the System Design Interview books, a bestseller on Amazon. The course covers system designs basics, then goes into deep dives of the design of over 10 famous common products (e.g. Designing YouTube, Facebook Newsfeed, etc) and multiple big data and storage systems (e.g., Designing a Chat System). For each deep dive, concepts are explained and comprehensive diagrams are used, making it very approachable for any seniority level.
2. "Grokking the System Design Interview" by Design Gurus — This is probably the most famous system design interview course on the internet and what makes it different from most other courses out there is that it is purely text-based, which is great for people who prefer reading over watching videos (such as myself!). It contains a repository of the popular system design problems along with a glossary of system design basics. I've personally completed this course and have recommended many others to use this. Highly recommended!
3. "System Design Interview Course" by Exponent — This course covers system designs basics and has a huge database of popular system design questions with videos of mock interviews. Some of the questions have text answers and a database schema and APIs for reference (which I find helpful). While the subscription

might be a little pricey for just the system design interviews content, they also offer quality technical content for Data Structures, Algorithms and Behavioral Interviews. The convenience of a one-stop platform which covers all aspects of technical interview preparation is very enticing.

4. "Grokking the Advanced System Design Interview" by Design Gurus — I haven't tried this but it's by the same people who created "Grokking the System Design Interview", so it should be good! In my opinion you probably wouldn't need this unless you're very senior or going for a specialist position.

Check out other Systems Design preparation guides and resources here.

Prepare for the behavioral interview

Every top tech company has at least one round of behavioral interviews for software engineers. Typically, behavioral interviews for software engineers include: Sharing about details of previous experiences on resume, providing examples of past situations and behavior that demonstrate certain behavioral attributes (e.g., conflict management, data driven, etc.), sharing of ambitions and career plans.

As much as these interviews seem "fluffy" or unstructured, there is actually a structured way to prepare for behavioral interviews:

STAR format

The **STAR** format helps you to organize your answers to behavioral questions. This is most applicable to questions that require you to recount past experiences or behavior.

- **Situation:** Share details about the situation that gave rise to the task
- **Task:** Explain what you needed to achieve or the problems you had to solve; focus on the
  - Scope
  - Severity
  - Specific benchmarks/outcomes required
- **Action:** Explain what you did to meet your objectives, describing options you had and how you made decisions
- **Results:** Describe the outcome of your actions and what you learnt

Read more: The STAR format for answering behavioral questions.

Practice

Practice the most common behavioral questions for software engineers. Refer to the top 30 most common behavioral questions for Software Engineers.

For more tips on behavioral interview preparation, refer to my full behavioral interview preparation guide here.

Negotiate your compensation

Finally, the last thing you absolutely need to prepare for before your interview is salary negotiation for software engineers. At any point during the interview process, conversation about salary may crop up. We also have in-depth guides about negotiation strategies and software engineer compensation.

And that is all from me — for more detail on each step of the software engineer interview preparation process, do dive into each topic within my handbook through the sidebar.

Writing your resume

Not sure why you're still not getting shortlisted at some or all of the top tech companies? Your software engineer resume could be the issue.

Having read tons of software engineering resumes as a FAANG interviewer, even some of the most qualified candidates I know fail to get shortlisted due to bad resumes. The mistake most people make is to immediately assume that they weren't qualified enough — but that could be far from the truth.

Thankfully, there are specific steps and requirements you can fulfill to write a good software engineer resume. From your resume structure, content, to free tools you can use to test your resume, I have collated a very concise summary of the best practices to prepare your resume for a FAANG software engineering job application:

1. Set up an ATS-friendly resume template
2. Fill up your template with well-framed content in a meaningful order
3. Optimize your resume with prioritization and keywords
4. Test out resume using free tools

Setting up an ATS-friendly resume

Something most candidates may not realize is that most top tech companies are using some form of Applicant Tracking Systems (ATS) to parse and screen thousands of resumes even before they reach human eyes. In many companies, the ATS may even use certain rules to reject candidates automatically.

While different companies could be using different types of ATS, it is possible to ensure that your software engineer resume is read favorably by most ATS. This section ensures your resume is at least perfectly readable by the ATS, while the next few sections improve your chances of passing ATS screenings.

> **Expert Tip**
>
> FAANG Tech Leads is currently offering resume templates and examples at 70% off.
>
> Their templates:
>
> - Are created by ex-FAANG hiring managers based on top resumes received from hundreds of candidate applications
> - Guarantee readability by FAANG ATS
> - Cater to various experience levels
>
> They also offer resume examples/references from candidates who have received multiple offers from FAANG companies — which are helpful in helping you craft content that meets the same bar.

Use MSWord or Google Docs to create and edit resume

Do's

- Submit your resume as a PDF file to preserve formatting, but always create it from Microsoft Word or Google Docs. It is important to ensure that the text in your resume is easily highlightable, which is a precondition for easy parsing.
- ATS tools are always trying to improve their readability of standard resume formats — hence the more commonplace your resume format is, the better.
- To maximize space on your resume, rather than using header or footers, reduce the margins of the page — narrow margins are 0.5 on each side.

Don'ts

- Do not use Photoshop, other graphic design tools or online resume builders to build your resume.
- Do not use the header or footer sections in a Word/Google Docs file — reduce margins instead and just write the information in the body.

Use standard fonts of readable sizes

New fonts may convert letters into special characters which are not readable by the ATS. Fonts you should use: Arial, Calibri, Garamond.

Ensure your font size remains readable for humans later on in the hiring process — use a minimum size of 10px for readability.

Add sections with standard headings and ordering

ATS readers need to identify and parse standard types of information from your resume. Using standard header titles and ordering can help them do that better.

This is the order which has worked well for me and recommended by recruiters:

Section	Heading Name
Professional summary	(Use resume headline as section title; for example: "Senior Software Engineer at Google with over 5 years of experience leading teams")
Contact information	"Contact Information
Skills — programming languages, frameworks, etc.	"Skills"
Experience	"Work Experience"
Education (Note: if you are still in school or have less than 3 years of experience, you may put Education first.)	"Education"
Projects	"Projects"
Other optional sections — certifications, awards, etc.	"Awards and Accolades" / "Certifications" / "Awards, Accolades and Certifications"

> **Caution**
>
> Never add symbols to your headers to avoid ATS readability issues.

Writing quality content

As software engineering is inherently different from other careers in terms of its required skills and experiences, the content expectations for a software engineer resume is also unique. Each of the following paragraphs will cover the content usually expected for software engineers within each resume section.

Professional summary

A good professional summary can be a game changer. Not only does it summarize your entire professional experience in a manner that individual sections cannot, it can also leave a pleasant impression on the hiring manager.

From my personal experience as a software engineering interviewer, I highly recommend professional summaries, as interviewers generally may not have the time to read into the detail — hence summaries which directly address why a candidate is a good fit for the job greatly improves their chances of capturing attention.

Here are my top tips for writing a great software engineering resume summary.

Before you start

Before you start: List down your best selling points. From your entire professional experience, list down the most important points that fulfill the job descriptions that you are applying for. This can include job experiences or skills.

Summarize selling points in your resume summary

Summarizing the selling points as much as possible, frame them into a short summary of fewer than 50 words:

Ensure you do these:

- ✅ Answer why you are a good fit for the job
- ✅ Use an active voice
- ✅ Use action words
- ✅ Start with the noun describing your job role (e.g., "Software Engineer", "Front End Engineer", etc.)

Write a headline for your resume summary

Instead of writing "Professional Summary" as the title of the section, further concise your experience into a headline with fewer than 10 words. Treat it like a slightly more elaborate version of your LinkedIn profile headline. Some examples:

Software Engineer (Full Stack)

Software Engineer with X years of full stack web development experience specializing in Ruby on Rails and PostgreSQL. Domain expert in e-commerce and payments field as a result of working at multiple e-commerce companies.

Senior Front End Engineer

Front End Engineer with X years of experience and strong fundamentals in Front End technologies. Likes building scalable web infrastructure and making websites fast. Passionate about programming languages, compilers, and developer tooling.

Software Engineering Lead

Software Engineer with X years of experience in back end, scaling complex distributed systems, and various cloud platforms. Led over 5 engineering teams with an average size of 6 members across two companies and mentored over 20 junior members.

Senior at University X

Senior Year student at University X with a focus on Artificial Intelligence and Machine Learning (ML). Interned at X companies and worked on full stack development and ML engineering roles.

> **Info**
>
> Read more about how to make your Professional Summary stand out on FAANG Tech Leads' Resume Handbook.

Contact information

Must-haves

- Name (Should be included at the very top of the resume)
- Personal phone number
  - Never include your work phone number here
- Location — City, State, Zip
  - Just enough for recruiters to determine if you are a local or international candidate
- Email address
  - Never include your work email here
  - I recommend Gmail if you are using other email services

- LinkedIn profile

Good-to-haves

- GitHub profile URL
- Personal website URL
- Stack Overflow profile URL
- Medium profile URL
- Competitive coding profile
  - CodeChef
  - HackerRank

If a divider is required between information, use "|" or tabs.

Where relevant, indicate achievements in coding platforms (e.g., max ratings, ranking, number of stars, badges, etc.).

Skills

Include programming languages and tech stacks:

Structure in the following manner:

```
[Skill summary] : [List skills separated by "|"]
```

- Programming languages — If impressive, include your familiarity by the number of lines you have written, for example "Over 10,000 lines"
- Frameworks
- Databases

Work experience

List your work experience in a familiar format and reverse chronological order. Every job listed should have the following: The company, location, title, and duration worked, following this structure:

```
[Company or Organization], [Location] | [Job Title] | [Start and end dates formatted as MM/YYYY]
```

Example:

```
Facebook, Singapore | Front End Engineering Lead | 08/2018 &amp;#8212; Present
```

List of top accomplishments, including:

- Scope of job and skills required
- Accomplishments listed following this structure

  ```
  [Accomplishment summary] : [Action] that resulted in [quantifiable outcome]
  ```

Education

Most software engineering jobs will require at least an undergraduate degree. However, unless you are a recent graduate or do not have much work experience, it should not be prioritized above your work experience.

Use the following format, eliminating information where it is not relevant:

```
[Degree Name], [Year of Graduation &amp;#8212; write expected graduation date if not graduated]
[University Name], [Location]
GPA: X.XX / 4.0 (List GPA if more than 3.50/4.00, or more than 4.3 under a 5-point system)
List key achievements, including leadership positions, skills, societies, projects, awards, etc.
```

Example:

```
BSc in Computing, Computer Science, Graduation Year 2015
National University of Singapore, Singapore
GPA: 3.82 / 4.00 (Magna cum laude)
Dean's List, Valedictorian
President of hacker society
```

Projects

Include at least 2 projects you have contributed to, outlining your key contributions. Always try to link your project name to GitHub or somewhere the hiring manager can view your project.

<details>
<summary>Example</summary>

`facebook/docusaurus`

Maintainer and lead engineer for Docusaurus v2, a static site generator which powers the documentation of many of Meta's Open Source Projects — React Native, Jest, Relay, Reason, etc. Used by 7.6k > projects on GitHub.
</details>

Awards and accolades

Only include achievements related to the job application and try to quantify your achievements. A good format to use would be

```
[Year][Quantification] [Competition]
```

Example:

```
2016 | Best All-Round Product out of 50 teams | Facebook Hackathon
```

Optimizing your resume

Less is more

Do's

- Highlighting a few of your best achievements is better than including many "average" achievements in your resume
- Use only 1 page for your resume

Don'ts

- Do not list all your achievements just to showcase a greater quantity without filtering

Keywords

Imagine you are a hiring manager or recruiter screening a resume while juggling many other tasks in your job — you simply won't have much time on each resume! When a hiring manager looks at a resume, they are in fact

quickly scanning for keywords of skills or experiences that they value, before paying any additional attention to your resume.

Recruiters and ATS do that as well, but based on the job description that the hiring manager helped to write. That is why optimizing your resume based on job descriptions is very important.

> **Info**
>
> Some ATS will determine the strength of your skills based on the frequency of a keyword in your resume, and others assign an estimated amount of experience for a skill based on its placement in your resume.
>
> For instance, if your previous job experience was 3 years long and you mentioned handling Search engine marketing (SEM), the ATS will assume 5 years of SEM experience.

Include keywords from job descriptions into your resume

You should always analyze the job description for must-have and good-to-have skills or experiences and ensure the keywords are added to your resume.

Include them under the "Skills" section and pepper the same keywords into the "Work Experience" and "Education" sections. Be sure to closely imitate the language within the job description.

Remember to include the full version of common abbreviations as well (e.g., Amazon Web Services instead of AWS, Google Cloud Platform instead of GCP, etc.).

However, do not do keyword stuffing for the sake of it — always remember that the resume will be read by a recruiter or hiring manager eventually.

Optimize keyword frequency and placement

Analyze the job description and determine how important each skill and experience is, then optimize the frequency of the keyword according to its importance.

> **Expert Tip**
>
> While optimizing for every application is ideal, you can generalize your resume to a type of position.
>
> 1. Collate 3 to 5 job descriptions for that position
> 2. Copy and paste them into a `.txt` file and upload it into a free "word and phrase" frequency tool like Online-Utility.org's Text Analyzer to identify regularly used keywords
> 3. Incorporate skills and experiences that you have into the resume

Reviewing your resume

Free resume review

Tech Interview Handbook has a resume review portal where you can upload your resume and get helpful comments and feedback from other community members and even the authors of Tech Interview Handbook!

If you are willing to spend some money we recommend FAANG Tech Leads' Resume Review service where your resume will be reviewed by ex-FAANG hiring managers and engineers, not random writers who don't understand technology.

Test readability with industry-standard ATS

Test the readability and formatting of your resume using industry-standard ATS like Resume Worded. Most big companies use such resume scanners.

The plain text file test

Simply copy the content from your resume and paste it into a plain text document! Make edits if:

- There are points missing from your original resume
- Characters are displaying incorrectly in plain text
- Sections are disorganized

Final tips

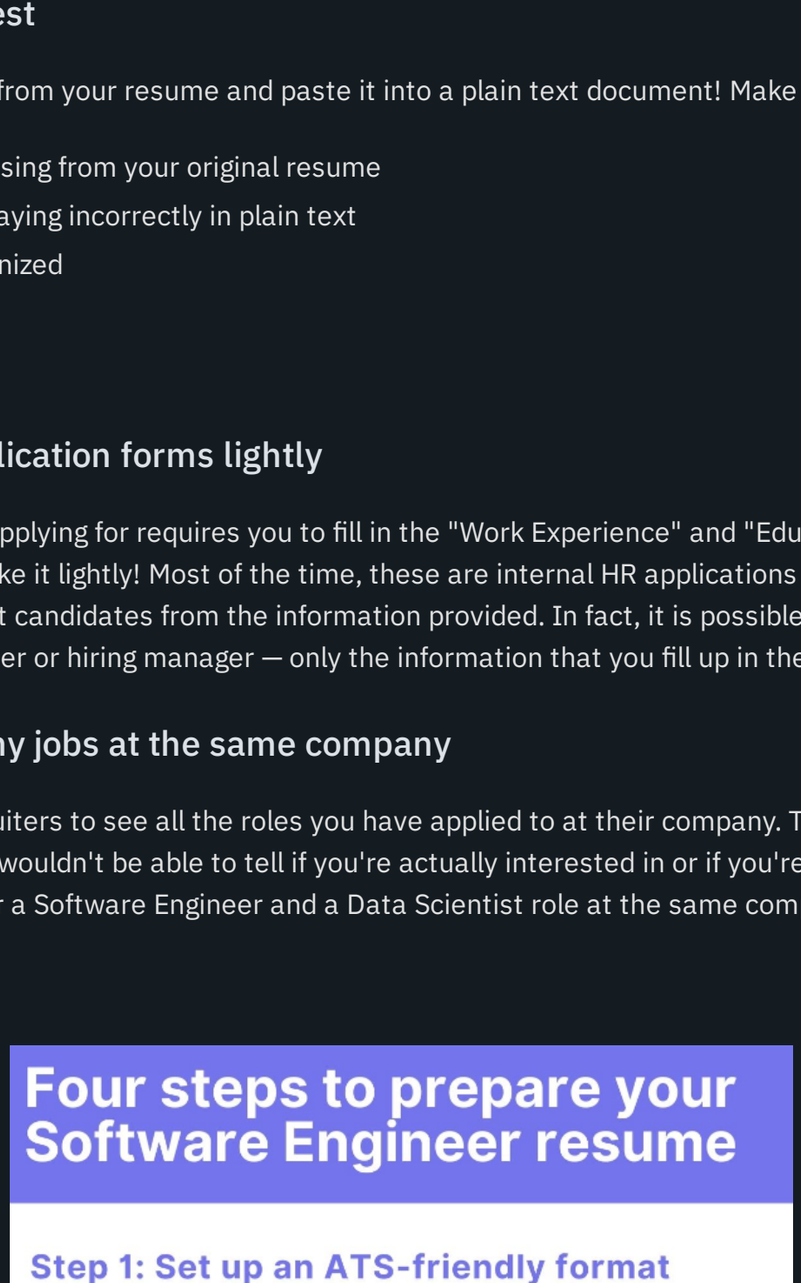
Do not take job application forms lightly

If the company you are applying for requires you to fill in the "Work Experience" and "Education" sections into their own form, do not take it lightly! Most of the time, these are internal HR applications which help parse job applications and filter out candidates from the information provided. In fact, it is possible that your resume is never seen by the recruiter or hiring manager — only the information that you fill up in their forms!

Do not apply to many jobs at the same company

The ATS also allows recruiters to see all the roles you have applied to at their company. Try not to apply to too many jobs as a recruiter wouldn't be able to tell if you're actually interested in or if you're self-aware about your abilities (e.g., applying for a Software Engineer and a Data Scientist role at the same company is not a good idea).

Summary checklist



▼ Checklist in text form

Resume preparation checklist

```
# Step 1: Set up an ATS-friendly format

[] Use MSWord or Google Docs to edit
[] Adopt a standard resume format, reducing margins (e.g., 0.5in) for more space
[] Apply standard fonts (Arial, Calibri, Garamond, etc.), min 10px
[] Add sections with standard headings and ordering

# Step 2: Write effective resume content

[] Write a professional summary of <50 words
[] Include your name, phone, location, email, and LinkedIn under "Contact Information"
[] Add your programming languages, frameworks, and databases under "Skills"
[] Work experience should be written highlighting utilized skills, frameworks, databases, and als
[] Include at least 2 open source or side projects you have worked on and link it to a site (e.g.

# Step 3: Optimize your resume

[]
[]

# Step 4: Test your resume out with free tools
```

Coding interview preparation

If you have decided to embark on the arduous process of preparing for your coding interviews and you don't know how to maximize your time, this is the only guide you need to go from zero to hero on your coding test.

Step-by-step how to prepare

What is a software engineering coding interview?

Coding interviews are a form of technical interviews used to access a potential software engineer candidate's competencies through presenting them with programming problems. Typically, coding interviews have a focus on data structures and algorithms, while other technical rounds may encompass system design (especially for middle to senior level candidates).

A coding interview round is typically 30-45 minutes. You will be given a technical question (or questions) by the interviewer, and will be expected to write code in a real-time collaborative editor such as CodePen or CoderPad (phone screen / virtual onsite) or on a whiteboard (onsite) to solve the problem within 30–45 minutes.

How will you be evaluated during a coding interview?

I have collated evaluation criteria across top tech companies and generalized them into a coding interview evaluation rubric you can use. Specific terminology or weightages may differ across companies, but top tech companies always include the following criteria in their evaluation:

1. **Communication** — Asking clarifying questions, communication of approach and tradeoffs clearly such that the interviewer has no trouble following.
2. **Problem solving** — Understanding the problem and approaching it systemically, logically and accurately, discussing multiple potential approaches and tradeoffs. Ability to accurately determine time and space complexity and optimize them.
3. **Technical competency** — Translating discussed solutions to working code with no significant struggle. Clean, correct implementation with strong knowledge of language constructs.
4. **Testing** — Ability to test code against normal and corner cases, self-correcting issues in code.

Read more about how you should behave in a coding interview to display hire signals.

How to best prepare for a coding interview?

LeetCode by itself is actually not enough to prepare you well for your coding interviews. Diving straight into LeetCode and thinking you can complete all of the thousands of questions is a bad use of your time and will never prepare you as well as a structured approach.

Given 30 min per question and an average of 3 hours practice a day, the average person will only manage to complete 160 questions within 3-4 weeks, and may not internalize the right approach or remember the questions they have practiced before.

Instead, this is how to prepare for your Software Engineer coding interview:

1. Pick a good programming language to use
2. Plan your time and tackle topics and questions in order of importance
3. Combine studying and practicing for a single topic
4. Accompany practice with coding interview cheat sheets to internalize the must-dos and must-remembers
5. Prepare a good self introduction and final questions
6. Try out mock coding interviews (with Google and Facebook engineers)
7. (If you have extra time) Internalize key tech interview question patterns

Pick a good programming language to use

A good programming language to use for coding interviews is one you are familiar with and is suitable for interviews.

What determines if a programming language should be used for interviews? Generally, we want higher level languages that have many standard library functions and data structures and are therefore "easier" to code in.

Recommended programming languages to use for coding interviews: Python, C++, Java, JavaScript

Read more about considerations for picking a programming language here.

Plan your time and tackle topics and questions in order of importance

How long does it take to prepare for a coding interview? It actually depends on how well prepared you want to be. On average, it takes about 30 hours to cover the bare minimum and ~100 hours to be well prepared.

To start preparing for your coding interviews, always begin with a plan. Calculate the amount of time you have left to realistically prepare for your interview from now till the day of the coding test, and carefully make a plan of the topics and questions you will cover per day, prioritizing the most important ones first.

But how do you know which are the most important topics and questions to practice based on the time you have left? You may use the free Grind 75 tool (built by me) which produces coding interview study plans for varying lengths of preparation time. The algorithm behind it includes a ranking of questions by priority and also a balance between breadth and depth of topics covered.

If you have the luxury of time to prepare, it is recommended to spend around 3 months (2-3 hours per day) to prepare more holistically. I came up with a personal 3-month study plan, which takes you from start to finish on which topics and questions to complete.

Combine studying and practicing for a single topic

For the sake of memory retention and efficiency, it is best to study for a single concept and then immediately do relevant practice questions for that topic.

Fortunately, there are already excellent coding interview preparation resources which enable you to do this very easily and systematically:

- AlgoMonster
- Grokking the Coding Interview: Patterns for Coding Questions

Accompany practice with coding interview cheatsheets to internalize the must-dos and must-remembers

To maximize what you get out of your practice, I recommend referring to the following coding interview cheatsheets *while* you are studying and practicing:

- **Coding interview techniques:** how to find a solution and optimize your approach
- **Coding interview best practices:** how to behave through the interview to exhibit hire signals
- **Algorithms study cheatsheets:** covers the best learning resources, must remembers (tips, corner cases) and must do practice questions for every data structure and algorithm

Coding interview techniques

Here is list of around 10 techniques to do the 2 most important things you need to do in a coding interview: finding approaches to solve the problem presented, and optimizing the time and space complexity of your approaches.

These techniques are useful to apply when you are given questions which you have never encountered before, and to get out of being stuck.

Coding interview best practices

Top tech companies evaluate candidates on 4 main criteria: communication, problem solving, technical competency and testing. To exhibit behaviors that fulfill these criteria, I have prepared a coding interview best practices cheatsheet which outlines what you should do before, during and after coding interviews. This is based on my personal experience as an interviewee as well as my observation of top candidates as an interviewer at Facebook.

Using this guide to accompany practice ensures that you cultivate good habits and muscle memory with regards to interviews right from the beginning.

Algorithms study cheatsheets for coding interviews

These are actually the notes I personally collated for my own coding interview preparation. I have organized them into 1-pagers of the best study resources, best LeetCode questions to practice, and must-remembers (tips, corner cases) for every data structure and algorithm. They ensure that you internalize the most important concepts and get the most out of your preparation. Check them out.

Prepare a good self introduction and final questions

Self introductions and final questions to ask are almost always required at the start and end of any software engineering interview. As such, you should always spend some time to craft an excellent self introduction and set of final questions to ask. When done well, these can leave a good impression with the interviewer that can turn things to your favor.

For the best software self introduction samples and tips, check out this self introduction guide for software engineers. Also check out samples of the best final questions to ask for software engineers in this final questions guide.

Try out mock coding interviews

Coding right in front of your interviewer can be a nerve-wracking experience especially if you have never done it before — which is why getting hands-on experience is so important.

interviewing.io is currently the best mock technical interview resource in the market. It allows you to book mock coding interviews with real Google and Facebook engineers, albeit anonymously. You could even book interviews for specific roles like Mobile, Front End, Engineering Management. Even better — if you want to have an easier transition into real world coding interview — you could view recorded interviews and see what phone interviews are like.

Moreover, if you were to do very well on your mock interviews, you will be able to unlock the "jobs page" which allows you to book interviews directly with top companies like Uber, Lyft, Quora, Asana and more. I've used interviewing.io both as an interviewer and an interviewee and found the experience to be excellent.

Read more about different mock coding interview platforms here.

(If you have extra time) Internalize key tech interview question patterns

Many coding interview solutions actually involve a similar set of key patterns — and learning them will help you solve any long tail problem that is outside the set of commonly asked coding interview questions.

Picking a programming language

Does the programming language you use for coding interviews matter? The answer is yes.

Most companies let you code in any language you want — the only exception I know being Google, where they only allow candidates to pick from Java, C++, JavaScript or Python for their algorithmic coding interviews.

However, the choice you make can impact your performance much more than you'd like to believe — and this is why it is important to pick a suitable programming language early on in your coding interview preparation — and use regularly in practice.

There are 3 considerations when deciding on which programming language to use:

- Suitability for interviews
- Your familiarity with the language
- Exceptions

Suitability for interviews

Some languages are just more suited for interviews — higher level languages like Python or Java provide standard library functions and data structures which allow you to translate solution to code more easily.

From my experience as an interviewer, most candidates pick Python or Java. Other commonly seen languages include JavaScript, Ruby and C++. I would absolutely avoid lower level languages like C or Go, simply because they lack many standard library functions and data structures and some may require manual memory management.

Personally, Python is my de facto choice for algorithm coding interviews because it is succinct and has a huge library of functions and data structures available. Python also uses consistent APIs that operate on different data structures, such as `len()`, `for ... in ...` and slicing notation on sequences (strings/lists/tuples). Getting the last element in a sequence is `arr[-1]` and reversing it is simply `arr[::-1]`. You can achieve a lot with minimal syntax in Python.

Java is a decent choice too but having to constantly declare types in your code means extra keystrokes which results in more typing which doesn't result in any benefit (in an interview setting). This issue will be more apparent when you have to write on a whiteboard during onsite interviews. The reasons for choosing/not choosing C++ are similar to Java. Ultimately, Python, Java and C++ are decent choices of languages.

- Recommended: Python, C++, Java, JavaScript
- Acceptable (but prefer recommended if you are familiar): Go, Ruby, PHP, C#, Swift, Kotlin
- Avoid: Haskell, Erlang, Perl, C, Matlab
- You must be mad: Brainfuck, Assembly

Your familiarity with the language

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 using in interviews.

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 revising/mastering

algorithms, there is barely spare energy 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.

If you have been using Java at work for a while now and do not have time to be comfortably familiar with another language, I would recommend just sticking to Java instead of picking up Python from scratch just for the sake of interviews. Doing so, you can avoid having to context switch between languages during work vs interviews. Most of the time, the bottleneck is in the thinking and not the writing. It takes some getting used to before one becomes fluent in a language and be able to wield it with ease.

Valid reasons to learn a new language:

- The interview requires usage of that language (domain-specific roles like mobile/front end/data science)
- You are not in a rush to start interviewing

Poor reasons to learn a new language:

- The company you are interviewing with uses that language heavily and you want to impress the interviewer/show that you fit in
- You want to show that you are trendy

Exceptions

One exception to the convention of allowing you to "pick any programming language you want" is when you are interviewing for a domain-specific position, such as Front End/iOS/Android Engineer roles, in which you would need to be familiar with coding in JavaScript, Objective-C/Swift and Java respectively. If you need to use a data structure that the language does not support, such as a Queue or Heap in JavaScript, perhaps try asking the interviewer whether you can assume that you have a data structure that implements certain methods with specified time complexities. If the implementation of that data structure is not crucial to solving the problem, the interviewer will usually allow this. In reality, being aware of existing data structures and selecting the appropriate ones to tackle the problem at hand is more important than knowing the intricate implementation details.

Study and practice plan

One of the most important questions to answer at the start of your coding interview preparation is: What study topics and practice questions should you do to most efficiently prepare for your coding interviews?

There are plenty of resources on the internet, but it can be hard to know how they fit into the time you have left to prepare. Thankfully, this article will help you with that.

I have personally gone through the dreaded Software Engineer interview process myself several times and prepared my own study plans, refining them each time.

In this article, I will share the 3 month study plan that I personally use to prepare for my coding interviews. You will find the exact topics to study (with recommended links) and exact questions to practice (with practice links).

Recommended preparation time and approach

How much time do you need to prepare for your coding interviews? Generally, 3 months (if you can dedicate 11 hours a week) is the recommended period of time for a more holistic preparation. I will be sharing recommended study plans for 3 months (recommended period), but you can generate study plans for practice questions for any time frame you need via the Grind 75 tool (built by me). More options like filtering by difficulty, topics, alternative grouping of questions can be found there.

Regardless of how long you have, if you are unfamiliar with core data structures and algorithms knowledge, you are advised to revise them before starting on the coding interview questions practice. Different people have different styles of practicing and you should do what works best for you. The various possible approaches are:

1. **Breadth-first preparation** — Revise every topic and then start practicing a variety of questions across all topics. This strategy is recommended if you have around a month to spare.
2. **Depth-first preparation** — Tackle one topic at a time, revise materials for a topic, and practice lots of questions for that topic. After ensuring mastery of a topic, move on to the next topic. Repeat for all or selected topics. If you don't have much time, this might be the best way to prepare. You can focus on the High priority topics in our recommended study plan.
3. **Depth-first-then-breadth preparation** — Tackle one topic at a time, revise materials for a topic, practice a few questions for that topic. After ensuring mastery, move on to the next topic. Repeat for all topics. At the end, practice a variety of questions across all topics. This strategy takes more time than others, so it's recommended if you have more than a month.

My personal recommendation would be the breadth-first preparation or depth-first-then-breadth preparation. It's important to have some form of breadth-level studying / practicing in your schedule so that you don't forget about the earlier topics as you move on to later topics.

3 month study plan

In each study plan, you will find a list of coding interview topics with resources and practice questions that you should work through in order of priority every week.

To best utilize it, you should create a template where you break down the dates left and hours left per day, so that you can later fill in the topics/questions to cover per day.

Keep the estimate relatively conservative so you don't end up burning out.

Weeks 1-4 (topical study and practice)

These are all the topics you should study, in order of priority. The learning resources linked are my algorithm cheatsheets — which give you an overview of must-remembers like time complexity, corner cases, and topic-specific useful techniques, as well as essential and recommended practice questions.

Don't forget to apply behaviors from coding interview best practices and methods from coding interview techniques early on while you practice!

Week 1

Topic	Priority	Time required
Array	High	2 hours
String	High	3 hours
Hash Table	Mid	3 hours
Recursion	Mid	3 hours

Week 2

Topic	Priority	Time required
Sorting and searching	High	3 hours
Matrix	High	1 hour
Linked List	Mid	3 hours
Queue	Mid	2 hours

Stack	Mid	2 hours

Week 3

Topic	Priority	Time required
Tree	High	4 hours
Graph	High	4 hours
Heap	Mid	3 hours
Trie	Mid	3 hours

Week 4

Topic	Priority	Time required
Interval	Mid	2 hours
Dynamic programming	Low	4 hours
Binary	Low	2 hours
Math	Low	1 hour
Geometry	Low	1 hour

Weeks 5-12 (in-depth practice)

Here, I listed 75 questions that you should do to be fully prepared for your coding interviews. This list of questions are generated from the Grind 75 tool (built by me), which generates recommended study plans for coding interviews based on the time you have left. More options like filtering by difficulty, topics, alternative grouping of questions can be found there.

- If you followed the Week 1 — 4 study plan, you would have done some of these questions here. Feel free to skip them or do them again.
- Feel free to skip the dynamic programming questions if you haven't studied them or feel that they won't be relevant. Many dynamic programming questions can be solved with recursion / backtracking anyway.

Don't forget to apply behaviors from coding interview best practices and methods from coding interview techniques early on while you practice!

We recommend using the Grind 75 tool which allows you to keep track of your practice progress.

Week 5

Problem	Difficulty	Duration
LC 1. Two Sum	Easy	15 mins
LC 20. Valid Parentheses	Easy	20 mins
LC 21. Merge Two Sorted Lists	Easy	20 mins
LC 121. Best Time to Buy and Sell Stock	Easy	20 mins
LC 125. Valid Palindrome	Easy	15 mins
LC 125. Valid Palindrome	Easy	15 mins
LC 226. Invert Binary Tree	Easy	15 mins
LC 242. Valid Anagram	Easy	15 mins
LC 704. Binary Search	Easy	15 mins
LC 733. Flood Fill	Easy	20 mins
LC 235. Lowest Common Ancestor of a Binary Search Tree	Easy	20 mins
LC 110. Balanced Binary Tree	Easy	15 mins
LC 141. Linked List Cycle	Easy	20 mins

Week 6

Problem	Difficulty	Duration
LC 232. Implement Queue using Stacks	Easy	20 mins
LC 278. First Bad Version	Easy	20 mins
LC 383. Ransom Note	Easy	15 mins
LC 70. Climbing Stairs	Easy	20 mins
LC 409. Longest Palindrome	Easy	20 mins
LC 206. Reverse Linked List	Easy	20 mins
LC 169. Majority Element	Easy	20 mins
LC 67. Add Binary	Easy	15 mins
LC 543. Diameter of Binary Tree	Easy	30 mins
LC 876. Middle of the Linked List	Easy	20 mins
LC 104. Maximum Depth of Binary Tree	Easy	15 mins
LC 217. Contains Duplicate	Easy	15 mins

Week 7

Problem	Difficulty	Duration
LC 155. Min Stack	Medium	20 mins
LC 53. Maximum Subarray	Medium	20 mins
LC 57. Insert Interval	Medium	25 mins

LC 542. 01 Matrix	Medium	30 mins
LC 973. K Closest Points to Origin	Medium	30 mins
LC 3. Longest Substring Without Repeating Characters	Medium	30 mins
LC 15. 3Sum	Medium	30 mins
LC 102. Binary Tree Level Order Traversal	Medium	20 mins
LC 133. Clone Graph	Medium	25 mins
LC 150. Evaluate Reverse Polish Notation	Medium	30 mins

Week 8

Problem	Difficulty	Duration
LC 207. Course Schedule	Medium	30 mins
LC 208. Implement Trie (Prefix Tree)	Medium	35 mins
LC 322. Coin Change	Medium	25 mins
LC 238. Product of Array Except Self	Medium	30 mins
LC 98. Validate Binary Search Tree	Medium	20 mins
LC 200. Number of Islands	Medium	25 mins
LC 994. Rotting Oranges	Medium	30 mins
LC 33. Search in Rotated Sorted Array	Medium	30 mins

Week 9

Problem	Difficulty	Duration
LC 39. Combination Sum	Medium	30 mins
LC 46. Permutations	Medium	30 mins
LC 56. Merge Intervals	Medium	30 mins
LC 236. Lowest Common Ancestor of a Binary Tree	Medium	25 mins
LC 981. Time Based Key-Value Store	Medium	35 mins
LC 721. Accounts Merge	Medium	30 mins
LC 75. Sort Colors	Medium	25 mins
LC 139. Word Break	Medium	30 mins

Week 10

Problem	Difficulty	Duration
LC 416. Partition Equal Subset Sum	Medium	30 mins
LC 8. String to Integer (atoi)	Medium	25 mins
LC 54. Spiral Matrix	Medium	25 mins
LC 78. Subsets	Medium	30 mins
LC 199. Binary Tree Right Side View	Medium	20 mins
LC 5. Longest Palindromic Substring	Medium	25 mins
LC 62. Unique Paths	Medium	20 mins
LC 105. Construct Binary Tree from Preorder and Inorder Traversal	Medium	25 mins
LC 11. Container With Most Water	Medium	35 mins

Week 11

Problem	Difficulty	Duration
LC 17. Letter Combinations of a Phone Number	Medium	30 mins
LC 79. Word Search	Medium	30 mins
LC 438. Find All Anagrams in a String	Medium	30 mins
LC 310. Minimum Height Trees	Medium	30 mins
LC 621. Task Scheduler	Medium	35 mins
LC 146. LRU Cache	Medium	30 mins
LC 230. Kth Smallest Element in a BST	Medium	25 mins
LC 76. Minimum Window Substring	Hard	30 mins

Week 12

Problem	Difficulty	Duration
LC 297. Serialize and Deserialize Binary Tree	Hard	40 mins
LC 42. Trapping Rain Water	Hard	35 mins
LC 295. Find Median from Data Stream	Hard	30 mins
LC 127. Word Ladder	Hard	45 mins
LC 224. Basic Calculator	Hard	40 mins

LC 1235. Maximum Profit in Job Scheduling	Hard	45 mins
LC 23. Merge k Sorted Lists	Hard	30 mins
LC 84. Largest Rectangle in Histogram	Hard	35 mins

Factor time for self introduction, final questions, and mock interviews

Besides studying and practicing for coding interviews, you should also prepare your self introduction, final questions, and try out mock coding interviews.

Prepare self introduction and final questions to ask

I would suggest around 3 hours to craft your self introduction and also prepare some final questions to ask. You may refer to this self introduction guide and final questions to ask guide which should help you complete these steps fairly quickly.

Schedule mock coding interviews

You should start scheduling for mock coding interviews when you are 60% through your coding interview studying and practicing plan. Interview slots are typically provided by interviewers, so you can view them in advance and book them. The platform I have personally used and recommend is interviewing.io. Read more about different mock coding interview platforms here.

Best practices before, during, and after

As coding interviews mature over the years, there are now firmer expectations on how candidates should behave during a coding interview. Some of these practices also help you to exhibit "hire" signals to the interviewer by displaying your ability to communicate well and deal with roadblocks.

Deriving the best practices from top candidates and also based on how you will be evaluated during a coding interview, we have summarized some of the top tips for how to behave during a coding interview in an exhaustive checklist — including top mistakes to avoid.

You should read and familiarize with this guide even before you start practicing your coding interview questions. Accompanying LeetCode grinding with this guide will allow you to ingrain these important coding interview behaviors early on.

Before your interview

- ✅ Dress comfortably.

  > Usually you do not need to wear smart clothes, casual should be fine. T-shirts and jeans are acceptable at most places.

- ✅ Ensure you have read and prepared for your self introduction and final questions to ask.

Virtual onsite

- ✅ Prepare pen and paper.

  > In case you need to jot and visualize stuff. Drawings are especially helpful for trees / graphs questions

- ✅ Use earphones or headphones and make sure you are in a quiet environment.

  > Avoid using speakers because if the echo is loud, communication is harder and participants repeating themselves will just result in loss of valuable time.

- ✅ Check that your internet connection, webcam, and audio are working.

- ✅ Familiarize yourself with and set up shortcuts for the coding environment (CoderPad / CodePen).

  > Set up the editor shortcuts, turn on autocompletion, tab spacing, etc. Interviewers are impressed if you know the shortcuts and use them well.

- ✅ Turn off the webcam if possible.

  > Most remote interviews will not require video chat and leaving it on only serves as a distraction and hogs network bandwidth.

Phone screen

- ✅ Use earphones and put the phone on the table.

  > Avoid holding a phone in one hand and only having one other hand to type

- ✅ Request for the option to use Zoom/Google Meet/Hangouts or Skype instead of a phone call.

  > It is easier to send links or text across.

Onsite whiteboarding

- ✅ Learn about whiteboard space management.

  > Leave space between lines of code in case you need to insert lines between existing code.

During interview

Make an effective self introduction

- ✅ Introduce yourself in a few sentences under a minute or 2.

  > Follow our guide on how to make a good self introduction for software engineers.

- ✅ Sound enthusiastic!

  > Speak with a smile and you will naturally sound more engaging.

- ❌ Do not spend too long on your self introduction as you will have less time left to code.

Ask clarifications for problems given

Do not jump into coding right away. Coding questions tend to be vague and underspecified on purpose to allow the interviewer to gauge the candidate's attention to detail and carefulness. Ask at least 2-3 clarifying questions.

- ✅ Paraphrase and repeat the question back at the interviewer.

  > Make sure you understand exactly what they are asking.

- ✅ Clarify assumptions (Refer to algorithms cheatsheets for common assumptions).

  > A tree-like diagram could very well be a graph that allows for cycles and a naive recursive solution would not work. Clarify if the given diagram is a tree or a graph.
  >
  > Can you modify the original array / graph / data structure in any way?
  >
  > How is the input stored?
  >
  > If you are given a dictionary of words, is it a list of strings or a Trie?
  >
  > Is the input array sorted? (e.g., for deciding between binary / linear search)

- ✅ Clarify input value range.

> Inputs: how big and what is the range?

- ✅ Clarify input value format

> Values: Negative? Floating points? Empty? Null? Duplicates? Extremely large?

- ✅ Work through a simplified example to ensure you understood the question.

> For example: You are asked to write a palindrome checker. Before coding, come up with simple test cases like "KAYAK" => true, "MOUSE" => false, then check with the interviewer if those example cases are in line with their expectations

- ❌ Do not jump into coding right away or before the interviewer gives you the green light to do so.

Work out and optimize approach WITH interviewer

The worst thing you can do next is jump straight to coding — interviewers expect there to be some time for a 2-way discussion on the correct approach to take for the question, including analysis of the time and space complexity.

This discussion can range from a few minutes to up to 5-10 minutes depending on the complexity of the question. This also gives interviewers a chance to provide you with hints to guide you towards an acceptable solution.

- ✅ If you get stuck on the approach or optimization, use this structured way to jog your memory / find a good approach.

- ✅ Explain a few approaches that you could take at a high level (don't go too much into implementation details). Discuss the tradeoffs of each approach with your interviewer as if the interviewer was your coworker and you all are collaborating on a problem.

> For algorithmic questions, space/time is a common tradeoff. Let's take the famous LC 1. Two Sum question for example. There are two common solutions:
>
> i. Use nested `for` loops. This would be $O(n^2)$ in terms of time complexity and $O(1)$ in terms of space.
> ii. In one pass of the array, you would hash a value to its index into a hash table. For subsequent values, look up the hash table to see if you can find an existing value that can sum up to the target. This approach is $O(n)$ in terms of both time and space. Discuss both solutions, mention the tradeoffs and conclude on which solution is better (typically the one with lower time complexity).

- ✅ State and explain the time and space complexity of your proposed approach(es).

> Mention the Big O complexity for time and explain why (e.g., $O(n^2)$ for time because there are nested for loops, $O(n)$ for space because an extra array is created). Master all the time and space complexity using the algorithm optimization techniques.

- ✅ Agree on the most ideal approach and optimize it. Identify repeated/duplicated/overlapping computations and reduce them via caching. Refer to the page on optimizing your solution.

- ❌ Do not jump into coding right away or before the interviewer gives you the green light to do so.

- ❌ Do not ignore any piece of information given.

- ❌ Do not appear unsure about your approach or analysis.

Code out your solution while talking through it

- ✅ Only start coding after you have explained your approach and the interviewer has given you the green light.

- ✅ Explain what you are trying to achieve as you are coding / writing. Compare different coding approaches where relevant.

> In so doing, demonstrate mastery of your chosen programming language.

- ✅ Code / write at a reasonable speed so you can talk through it — but not too slow.

> You want to type slow enough so you can explain the code, but not too slow as you may run out of time to answer all questions

- ✅ Write actual compilable, working code where possible, not pseudocode.

- ✅ Write clean, straightforward and neat code with as few syntax errors / bugs as possible.

> Always go for a clean, straightforward implementation than a complex, messy one. Ensure you adopt a neat coding style and good coding practices per language paradigms and constructs. Syntax errors and bugs should be avoided as much as possible.

- ✅ Use variable names that explain your code.

> Good variable names are important because you need to explain your code to the interviewer. It's better to use long variable names that explain themselves. Let's say you need to find the multiples of `3` in an array of numbers. Name your results array `multiplesOfThree` instead of `array`/`numbers`.

- ✅ Ask for permission to use trivial functions without having to implement them.

> For example: `reduce`, `filter`, `min`, `max` should all be ok to use.

- ✅ Write in a modular fashion, going from higher-level functions and breaking them down into smaller helper functions.

> Let's say you're asked to build a car. You can just write a few high level functions first: `gatherMaterials()`, `assemble()`. Then break down `assemble()` into smaller functions, `makeEngine()`, `polishWheels()`, `constructCarFrame()`. You could even ask the interviewer if it's ok to not code out some trivial helper functions.

- ✅ If you are cutting corners in your code, state that out loud to your interviewer and say what you would do in a non-interview setting (no time constraints).

> For example: "Under non-interview settings, I would write a regex to parse this string rather than using `split()` which may not cover certain edge cases."

- ✅ [Onsite / Whiteboarding] Practice whiteboard space management

- ❌ Do not interrupt your interviewer when they are talking. Usually if they speak, they are trying to give you hints or steer you in the right direction.

- ❌ Do not spend too much time writing comments.

- ❌ Do not repeat yourself

- ❌ Do not use bad variable names.

> Do not use extremely verbose or single-character variable names (unless they're common like `i`, `n`).

- ❌ Do not copy and paste code without checking (e.g., some variables might need to be renamed after pasting).

Check your code and add test cases after coding

Once you are done coding, do not announce that you are done. Interviewers expect you to start scanning for mistakes and adding test cases to improve on your code.

- ✅ Scan through your code for mistakes — such as off-by-one errors.

> Read through your code with a fresh pair of eyes — as if it's your first time seeing a piece of code written

> Read through your code with a fresh pair of eyes — as if it's your first time seeing a piece of code written by someone else — and talk through your process of finding mistakes.

- ✓ Brainstorm edge cases with the interviewer and add additional test cases. (Refer to algorithms cheatsheets for common corner cases.)

  > Given test cases are usually simple by design. Brainstorm on possible edge cases such as large sized inputs, empty sets, single item sets, negative numbers.

- ✓ Step through your code with those test cases.
- ✓ Look out for places where you can refactor.
- ✓ Reiterate the time and space complexity of your code.

  > This allows you to remind yourself to spot issues within your code that could deviate from the original time and space complexity.

- ✓ Explain trade-offs and how the code / approach can be improved if given more time.
- ✕ Do not immediately announce that you are done coding. Do the above first!
- ✕ Do not argue with the interviewer. They may be wrong but that is very unlikely given that they are familiar with the question.

Leave a good impression

- ✓ Ask good final questions that are tailored to the company.

  > Read tips and sample final questions to ask.

- ✓ Thank the interviewer.
- ✕ Do not end the interview without asking any questions.

After interview

- ✓ Record the interview questions and answers down as these can be useful for future reference.
- ✓ Send a follow up email or LinkedIn invitation to your interviewer(s) thanking them for their time and the opportunity to interview with them.

  > As an interviewer myself, these can leave a lasting impression on me.

Techniques to solve questions

The biggest fear most candidates will have during a coding interview is: what if I get stuck on the question and don't know how to do it? Fortunately, there are structured ways to approach coding interview questions that will increase your chances of solving them. From how to find a solution or approach, to optimizing time and space complexity, here are some of the top tips and best practices that will help you solve coding interview questions.

When given a coding interview question, candidates should start by asking clarifying questions and discussing a few possible approaches with their interviewers. However, this is where most candidates tend to get stuck. Thankfully, there are ways to do this in a structured manner.

Note that not all techniques will apply to every coding interview problem, and you can also use multiple techniques on one single problem! As you apply these techniques during your practice, you will develop the intuition for which technique will be useful for the problem at hand.

Visualize the problem by drawing it out

Ever wondered why coding interviews are traditionally done on whiteboards and videos explaining answers to coding questions tend to use diagrams? Whiteboards make it easy to draw diagrams which helps with problem solving! A huge part of coding is understanding how the internal state of a program changes and diagrams are super useful tools for representing the internal data structure's state. If you are having a hard time understanding how the solution is obtained, come up with a visual representation of the problem and, if necessary, the internal states at each step.

This technique is especially useful if the input involves trees, graphs, matrices, linked lists.

<details>
<summary>Example</summary>

How would you return all elements of a matrix in spiral order? Drawing out the matrix and the steps your iterator needs to take in each direction will help tremendously in allowing you to see the pattern.
</details>

Think about how you would solve the problem by hand

Solving the problem by hand is about solving the problem without writing any code, like how a non-programmer would. This already happens naturally most of the time when you are trying to understand the example given to you.

What some people don't realize is that sometimes a working solution is simply a code version of the manual approach. If you can come up with a concrete set of rules around the approach that works for every example, you can write the code for it. While you might not arrive at the most efficient solution by doing this, it's a start which will give you some credit.

<details>
<summary>Example</summary>

How do you validate if a tree is a valid Binary Search Tree without writing any code? You first check if the left subtree contains only values less than the root, then check that the right subtree contains only values bigger than the root, then repeat for each node. This process seems feasible. Now you just have to turn this process into code.
</details>

Come up with more examples

Coming up with more examples is something useful you can do regardless of whether you are stuck or not. It helps you to reinforce your understanding of the question, prevents you from prematurely jumping into coding, helps you to identify a pattern which can be generalized to any input, which is the solution! Lastly, the multiple examples can be used as test cases at the end when verifying your solution.

Break the question down into smaller independent parts

If the problem is large, start with a high-level function and break it down into smaller constituting functions, solving each one separately. This prevents you from getting overwhelmed with the details of doing everything at once and keeps your thinking structured.

Doing so also makes it clear to the interviewer that you have an approach, even if you don't manage to finish coding all of the smaller functions.

<details>
<summary>Example</summary>

The Group Anagrams problem can be broken down into two parts — hashing a string, grouping the strings together. Each part can be solved separately with independent implementation details. You could start off with this code:

```
def group_anagrams(strings):
```
</details>

```
def hash(string):
  # Fill in later
  pass


def group_strings(strings_hashes):
  # Fill in later
  pass

strings_hashes = [(string, hash(string)) for string in strings]
return group_strings(strings_hashes)
```

And proceed to fill in the implementation of each function. However, do note that sometimes the most efficient solutions will require you to break some abstractions and do multiple operations in one pass of the input. If your interviewer asks you to optimize based on your well-abstracted solution, that is one possible path forward.

Apply common data structures and algorithms at the problem

Unlike real-world software engineering where the problems are usually open-ended and might not have clear solutions, coding interview problems tend to be smaller in nature and are designed to be solvable within the duration of the interview. You can also expect that the knowledge required to solve the problem is not out of this world and they would have been taught during college. Thankfully, the number of common data structures and algorithms is finite and a hacky approach which works from my experience is to try going through all the common data structures and applying them to the problem.

These are the data structures to keep in mind and try, in order of frequency they appear in coding interview questions:

- **Hash Maps:** Useful for making lookup efficient. This is the most common data structure used in interviews and you are guaranteed to have to use it.
- **Graphs:** If the data is presented to you as associations between entities, you might be able to model the question as a graph and use some common graph algorithm to solve the problem.
- **Stack and Queue:** If you need to parse a string with nested properties (such as a mathematical equation), you will almost definitely need to use stacks.
- **Heap:** Question involves scheduling/ordering based on some priority. Also useful for finding the max K/min K/median elements in a set.
- **Tree/Trie:** Do you need to store strings in a space-efficient manner and look for the existence of strings (or at least part of them) very quickly?

Routines:

- Sorting
- Binary search: Useful if the input array is sorted and you need to do faster than $O(n)$ searches
- Sliding window
- Two pointers
- Union find
- BFS/DFS
- Traverse from the back
- Topological Sorting

In the future we will add tips on how to better identify the most relevant data structures and routines based on the problem.

How to optimize your approach or solution

After you've come up with an initial solution to the coding interview problem, your interviewer would most likely prompt you to optimize the solution by asking, "Can we do better". The following techniques help you further optimize the time and space complexity of your solution:

How to optimize time complexity

1. Identify the Best Theoretical Time Complexity of the solution

The Best Theoretical Time Complexity (BTTC) of a solution is a time complexity you know that you cannot beat.

Some simplified examples:

- The BTTC of finding the sum of numbers in array is $O(n)$ because you have to look at every value in the array at least once
- The BTTC of finding the number of groups of anagrams is $O(nk)$ where n is the number of words and k is the maximum number of letters in a word because you have to look at each word at least once and look at each character in each word at least once
- The BTTC of finding the number of islands in a matrix is $O(nm)$ where n is the number of rows and m is the number of columns because you have to look at each cell in the matrix at least once

Why is it important to know the BTTC? So that you don't go down the rabbit hole of trying to find a solution that is faster than the BTTC. The fastest practical solution can only ever be as fast as the BTTC, not faster than the BTTC. The BTTC is not necessarily achievable in practice (hence theoretical), it just means you can never find a real solution that is faster than it. If your initial solution is slower than the BTTC, there could be opportunities to improve such that you can attain the BTTC (but not always the case). It wouldn't hurt to mention the BTTC to your interviewer, which will be taken as a positive signal and also to remind yourself that you should not try to come up with something faster than the BTTC.

Some people might think that the BTTC is simply the total number of elements in a data structure, because you need to go through each element once. This is **not always true**. The most famous example would be finding a number in a sorted array of numbers. The sorted property changes things a whole lot:

- Finding a number would be $O(\log n)$ because you can use a binary search.
- Finding the largest number would be $O(1)$ because it is the last value in the array.

This is why it is important to pay attention to every detail given about the question. Be careful not to determine the incorrect BTTC due to lack of attention to the question details!

With the correct BTTC determined, you now know the time complexity of the optimal solution lies between your initial solution and the BTTC and can work your way towards it. If your solution already has the BTTC and the interviewer is asking you to optimize further, there are usually two things they are looking out for:

- Do even less work. Your solution could be $O(n)$ but making two passes of the array and the interviewer is looking for the solution that uses a single pass.
- Use less space. Refer to the section below on optimizing space complexity.

Identify overlapping and repeated computation

A naive/brute force solution often executes the same operation over and over again. When the code is doing an expensive operation that has been done before, take a moment to step back and consider if you can reuse results from previous computations. Dynamic programming (DP) is the most obvious type of questions you can entirely leverage past computations. There are non-DP questions that can leverage this technique too, although not as straightforward and might require a preprocessing step.

▼ Example

The Product of Array Except Self question is a good example of a problem which contains overlapping/repeated work. To get the value for an index, you need to multiply the values at all other positions. Doing this for every value in the array would take O(n2) time. However, see that:

- `result[n]`: `Product(nums[0] … nums[n-1]) * Product(nums[n + 1] … nums[N &#8212; 1])`

- `result[n + 1]`: `Product(nums[0] … nums[n]) * Product(num[n + 2] … nums[N &#8212; 1])`

There's a ton of duplicated work in computing the `result[n]` vs `result[n + 1]`! This is an opportunity to reuse earlier computations made while computing `result[n]` to compute `result[n + 1]`. Indeed, we can make use of a prefix array to help us arrive at the final solution in $O(n)$ time at the cost of more space.

Try different data structures

Choice of data structures is key to coding interviews. It can help you to reach a solution for the problem, it can also help you to optimize your existing solution. Sometimes it's worth going through the exercise of iterating through the data structures you know once again.

Is lookup time slowing your algorithm down? In general, most lookup operations should be O(1) with the help of a hash table. If the lookup operation in your solution is the bottleneck to your solution's time complexity, more often than not, you can use a hash table to optimize the lookup.

▼ Example

The K Closest Points to Origin question can be solved in a naive manner by calculating the distance of each point, sorting them and then taking the K smallest values. This takes $O(n \log n)$ time because of the sorting. However, by using a Heap data structure, the time complexity can be reduced to $O(n \log k)$ as adding/removing from the heap only takes $O(\log k)$ time when the size of the heap is capped at K elements. Changing the data structure made a whole ton of difference to the efficiency of the algorithm!

Identify redundant work

Here are a few examples of code which is doing redundant work. Although making these mistakes might not change the overall time complexity of your code, you are also evaluated on coding abilities, so it is important to write as efficient code as possible.

- **Don't check conditions unnecessarily:** These are Python examples where the second check is redundant.
  - `if not arr and len(arr) == 0`: The first check already ensures that the array is empty so there is no need for the second check.
  - `x < 5 and x < 10`: The second check is a subcondition of the first check.
- **Mind the order of checks:**
  - `if slow() or fast()`: There are two operations in this check, of varying durations. As long as one of the operations evaluates to `true`, the condition will evaluate to `true`. Most computers execute operations in order from left to right, hence it is more efficient to put the `fast()` on the left.
  - `if likely() and unlikely()`: This example uses a similar argument as above. If we execute `unlikely()` first and it is `false`, we don't have to execute `likely()`.
- **Don't invoke methods unnecessarily:** If you have to refer to a property multiple times in your function and that property has to be derived from a function call, cache the result as a variable if the value doesn't change throughout the lifetime of the function. The length of the input array is the most common example. Most of the time, the length of the input array doesn't change, declare a variable at the start called `length = len(array)` and use length in your function instead of calling `len(array)` every time you need it.
- **Early termination:** Stop after you already have the answer, return the answer immediately. Here's an example of leveraging early termination. Consider this basic question "Determine if an array of strings contain a string regardless of case sensitivity". The code for it:

```
def contains_string(search_term, strings):
  result = False
  for string in strings:
    if string.lower() == search_term.lower():
      result = True
  return result
```

Does this code work? Definitely. Is this code as efficient as it can be? Nope. We only need to know if the search term exists in the array of strings. We can stop iterating as soon as we know that there exists the value.

```
def contains_string(search_term, strings):
  for string in strings:
    if string.lower() == search_term.lower():
      return True # Stop comparing the rest of the array/list because the result won't change.
  return False
```

Most people already know this and already do this outside of an interview. However, in a stressful interview environment, people tend to forget the most obvious things. Terminate early from loops where you can.

- **Minimize work inside loops:** Let's further improve on the example above to solve the question "Determine if an array of strings contain a string regardless of case sensitivity".

```
def contains_string(search_term, strings):
  for string in strings:
    if string.lower() == search_term.lower():
      return True
  return False
```

Note that you are calling `search_term.lower()` once per loop of the `for` loop! It's a waste because the `search_term` doesn't change throughout the lifecycle of the function.

```
def contains_string(search_term, strings):
  search_term_lowercase = search_term.lower()
  for string in strings:
    if string.lower() == search_term_lowercase:
      return True
  return False
```

Minimize work inside loops and don't redo work you have already done if it doesn't change.

- **Be lazy:** Lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed. Let's use the same example as above. We could technically improve it a little bit:

```
def contains_string(search_term, strings):
  if len(strings) == 0:
    return False
  # Don't have to change the search term to lower case if there are no strings at all.
  search_term_lowercase = search_term.lower()
  for string in strings:
    if string.lower() == search_term_lowercase:
      return True
  return False
```

This is considered a micro-optimization and most of the time, strings won't be empty, but I'm using it to illustrate the example where you don't have to do certain computations if they aren't needed. This also applies to initialization of objects that you will need in your code (usually hash tables). If the input is empty, there's no need to initialize any variables!

How to optimize space complexity

Most of the time, time complexity is more important than space complexity. But when you have already reached

the optimal time complexity, the interviewer might ask you to optimize the space your solution is using (if it is using extra space). Here are some techniques you can use to improve the space complexity of your code.

Changing data in-place/overwriting input data

If your solution contains code to create new data structures to do intermediate processing/caching, memory space is being allocated and can sometimes be seen as a negative. A trick to get around this is by overwriting values in the original input array so that you are not allocating any new space in your code. However, be careful not to destroy the input data in irreversible ways if you need to use it in subsequent parts of your code.

A possible way which works (but you should never use outside of coding interviews) is to mutate the original array and use it as a hash table to store intermediate data. Refer to the example below.

Note that in Software Engineering, mutating input data is generally frowned upon and makes your code harder to read and maintain, so changing data in-place is mostly something you should do only in coding interviews.

<details>
<summary>Example</summary>

The Dutch National Flag problem could be easily solved with $O(n)$ time and $O(n)$ space by creating a new array and filling it up with the respective values in a sorted fashion. As an added challenge and space optimization, the interviewer will usually ask for an $O(n)$ time and $O(1)$ space solution which involves sorting the input array in-place.

An example of using the original array as a hash table is the First Missing Positive question. After the first `for` loop, all the values in the array are positive, and you can indicate presence of a number by negating the value at the index corresponding to the number. To indicate `4` is present, negate `nums[4]`.
</details>

Change a data structure

Data structures again!? Yes, data structures again! Data structures are so fundamental to coding interviews and mastery of it makes or breaks your interview performance. Are you using the best data structure possible for the problem?

<details>
<summary>Example</summary>

You're given a list of strings and want to find how many of these strings start with a certain prefix. What's an efficient way to store the strings so that you can compute your answer quickly? A Trie is a tree-like data structure that is very efficient for storing strings and also allows you to quickly compute how many strings start with a prefix.
</details>

Mock coding interviews

Interviewing is a skill that you can get better at. The steps mentioned above can be rehearsed over and over again until you have fully internalized them and following those steps become second nature to you. A good way to practice is to find a friend to partner with and the both of you can take turns to interview each other.

interviewing.io

A great resource for practicing mock coding interviews would be interviewing.io. interviewing.io provides anonymous practice technical interviews with Google and Facebook engineers, which can lead to real jobs and internships. By virtue of being anonymous during the interview, the inclusive interview process is de-biased and low risk. At the end of the interview, both interviewer and interviewees can provide feedback to each other for the purpose of improvement. Doing well in your mock interviews will unlock the jobs page and allow candidates to book interviews (also anonymously) with top companies like Uber, Lyft, Quora, Asana and more. You can also book mock interviews for more specific roles such as Mobile, Front End, Engineering Management. For those who are totally new to technical interviews, you can even view recorded interviews and see how phone interviews are like. Read more about them here.

I have used interviewing.io both as an interviewer and an interviewee and found the experience to be really great! Aline Lerner, the CEO and co-founder of interviewing.io and her team are passionate about revolutionizing the technical interview process and helping candidates to improve their skills at interviewing. She has also published a number of technical interview-related articles on the interviewing.io blog.

Pramp

Another platform that allows you to practice coding interviews is Pramp. Where interviewing.io matches potential job seekers with seasoned technical interviewers, Pramp takes a different approach. Pramp pairs you up with another peer who is also a job seeker and both of you take turns to assume the role of interviewer and interviewee. Pramp also prepares questions for you, along with suggested solutions and prompts to guide the interviewee.

Personally, I am not that fond of Pramp's approach because if I were to interview someone, I would rather choose a question I am familiar with. Also, many users of the platform do not have the experience of being interviewers and that can result in a horrible interview experience. There was once where my matched peer, as the interviewer, did not have the right understanding of the question and attempted to lead me down the wrong path of solving the question. However, this is more of a problem of the candidate than the platform though.

Coding interview rubrics

Ever wondered how coding interviews are evaluated at top tech companies like Google, Amazon, Apple and Netflix?

Across top tech companies, coding interview evaluation criteria do not actually differ to a great extent. While the exact terms used in the rubric could be different, the dimensions evaluated are roughly similar.

I will go into detail on the general coding interview evaluation process across big tech companies in this guide. I've also included an example rubric you can use while practicing on your own or with your peers.

Candidate scoring methedology

Generally across FAANG / MANGA companies, coding interview evaluation rubrics can be split broadly into 4 dimensions:

Dimension	Description
Communication	Does the candidate make clarifications, communicate their approach, and explain what they are doing while coding?
Problem Solving	Does the candidate show they understand the problem and are able to come up with a sound approach, conduct trade-offs analysis, and optimize their approach?
Technical Competency	How fast and accurate is the implementation? Were there syntax errors?
Testing	Was the code tested for common and corner cases? Did they self-correct bugs?

There are 2 general methods of candidate scoring in coding interviews:

- Provide a score (e.g., `1-4`) for every dimension and sum them up into an overall score
- Provide an overall score (e.g., `1-4`) based on overall performance across dimensions

Regardless of the method used, the scoring bands are generally:

- Strong hire

- Hire
- No hire

- Strong no hire

Some companies may have a middle band for indecision when the interviewer feels that the candidate requires more assessment.

How does your score impact the result?

Regardless of the scoring methodology, the final score is based on the overall performance across evaluated criteria (not purely through a certain mathematical cut-off).

For each phone screen round, there's typically only one interviewer, hence if they don't give you a "pass" equivalent to "Leaning hire" and above, you would not proceed to the full interview loop. If there were no clear signals obtained from the round, you might be asked to do a follow up phone screen round.

Most top tech companies allow candidates to go through every interview round in the full interview loop before making a decision based on the final package. If a candidate receives mixed results (some "pass" and some "fail") from different rounds, interviewers will convene for a discussion based on the signals you displayed. This is why your performance throughout the interview loop is important.

In certain cases, like follow up phone screen rounds, candidates may be invited for additional assessment rounds if:

- There were aspects that were missed out in the assessment (e.g., 2 coding round interviewers gave very similar questions)
- Candidate displayed mixed signals in particular areas and additional rounds are required to obtain more reliable signals

Generally, your scores and feedback for each round are visible to all interviewers. Sometimes, interviewers can even see feedback from your interviews at the same company in the past to avoid asking the same question again. Companies want to see that you have grown as compared to the past. So if you got rejected in the past by a company, reflect on possible reasons and address them if/when you interview with that company again.

Explanation of each evaluated criteria

Communication

Basic communication signals:

- Asks appropriate clarifying questions
- Communicates approach, rationale, and tradeoffs
- Constantly communicating, even while coding
- Well organized, succinct, clear communication

Score	Overall evaluation
Strong hire	Throughout the interview, communication was thorough, well-organized, succinct and clear in terms of thought process — including how they understand the question, their approach, trade-offs. Interviewer had no challenge following and understanding the candidate's thought process at all.
Leaning hire	Throughout the interview, communication was sufficient, clear and organized. However, the interviewer had to ask follow-up questions to understand the candidate on certain aspects such as their approach or thought process.
Leaning no hire	Throughout the interview, communication was (1 or more of the following): (1) Insufficient (e.g., jumped into coding without explaining), (2) Disorganized or unclear. Interviewer had difficulty following the candidate's thought process.
Strong no hire	Could not communicate with any clarity or stayed silent even when addressed by the interviewer. Interviewer had extreme difficulty following the candidate's thought process.

Problem solving

Basic problem solving signals:

- Understands the problem quickly by asking good clarifying questions
- Approached the problem systematically and logically
- Was able to come up with an optimized solution
- Determined time and space complexity accurately
- Did not require any major hints from the interviewer

Advanced problem solving signals:

- Came up with multiple solutions
- Explained trade-offs of each solution clearly and correctly, concluded on which of them are most suitable for the current scenario
- Had time to discuss follow up problems/extensions

Score	Overall evaluation
Strong hire	No trouble achieving all basic problem solving signals and did so with enough time to achieve most advanced problem solving signals.
Leaning hire	Managed to achieve all basic problem solving signals but did not have sufficient time to achieve advanced problem solving signals.
Leaning no hire	Showed only some basic problem solving signals, failing to achieve the rest.
Strong no hire	Unable to solve the problem or did it without much explanation of their thought process. Approach was disorganized and incorrect.

Technical competency

Basic technical competency signals:

- Translates discussed solution into working code with minimal to no bugs
- Clean and straightforward implementation with no syntax errors and unnecessary code, good coding practices (e.g., DRY, uses proper abstractions, etc.)
- Neat coding style (proper indentation, spacing, variable naming, etc.)

Advanced technical competency signals:

- Compares several coding approaches
- Demonstrates strong knowledge of language constructs and paradigms

Score	Overall evaluation
Strong hire	Demonstrated basic and advanced competency signals effortlessly.
Leaning	Demonstrated only basic technical competency signals, with some difficulty seen in translating

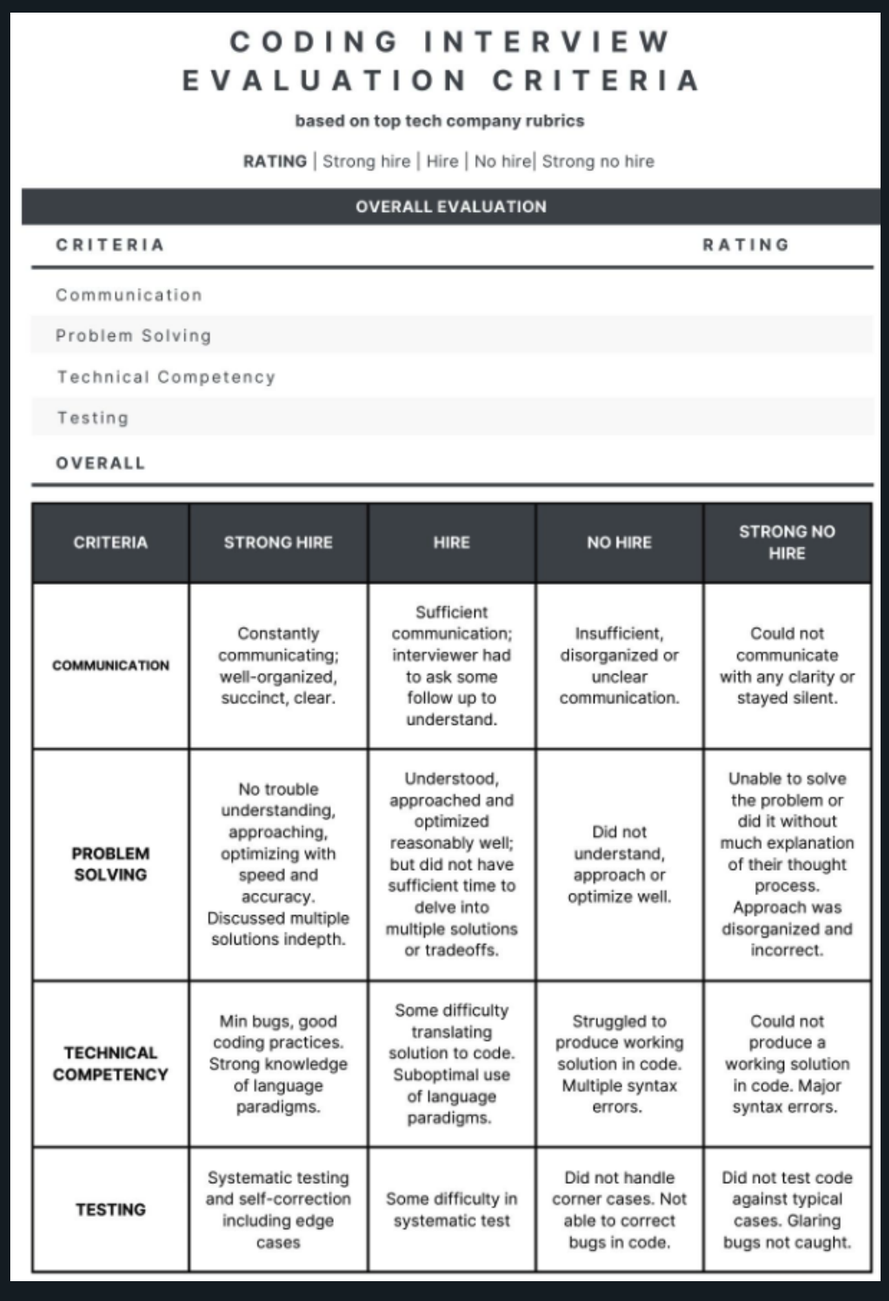
hire	approach to code. Suboptimal usage of language paradigms.
Leaning no hire	Struggled to produce a working solution in code. Multiple syntax errors and bad use of language paradigms.
Strong no hire	Could not produce a working solution in code. Major syntax errors and very bad use of language paradigms.

Testing

Testing signals:

- Came up with more typical cases and tested their code against it
- Found and handled corner cases
- Identified and self-corrected bugs in code
- Able to verify correctness of the code in a systematic manner (e.g., acting like a debugger and stepping through each line, updating the program's state at each step)

Score	Overall evaluation
Strong hire	Demonstrated testing signals effortlessly.
Leaning hire	Had some difficulty demonstrating testing signals, such as not being able to identify all the relevant corner cases.
Leaning no hire	Conducted testing but did not handle corner cases. Not able to identify or correct bugs in code.
Strong no hire	Did not even test code against typical cases. Did not spot glaring bugs in the code and announced they are done.



<details>
<summary>Coding interview evaluation criteria in text form</summary>

CRITERIA	STRONG HIRE	HIRE	NO HIRE	STRONG NO HIRE
Communication	Constantly communicating; well-organized, succinct, clear.	Sufficient communication; interviewer had to ask some follow up questions to understand.	Insufficient, disorganized, or unclear communication.	Could not communicate with any clarity or stayed silent.
Problem solving	No trouble understanding, approaching, optimizing with speed and accuracy. Discussed multiple solutions in-depth.	Understood, approached, and optimized reasonably well; but did not have sufficient time to delve into multiple solutions or tradeoffs.	Did not understand, approach, or optimize well.	Unable to solve the problem or did it without much explanation of their thought process. Approach was disorganized and incorrect.
Technical competency	Minimum bugs, good coding practices. Strong knowledge of language paradigms.	Some difficulty translating solution to code. Suboptimal use of language paradigms.	Struggled to produce working solution in code. Multiple syntax errors.	Could not produce a working solution in code. Major syntax errors.
Testing	Systematic testing and self-correction including edge cases.	Some difficulty in systematic testing.	Did not handle corner cases. Not able to correct bugs in code.	Did not test code against typical cases. Glaring bugs not caught.

</details>

System design interview preparation

The objective of system design interviews is to evaluate a candidate's skill at designing real-world software systems involving multiple components. System design questions are typically given to more senior candidates (with a few years of experience). Interns aren't typically given system design questions as it is hard to expect interns to have sufficient and relevant industry experience to answer these types of questions well.

Some common questions include:

- Design a URL shortener (e.g., Bitly)
- Design a social media website (e.g., Twitter)

- Design a video watching website (e.g., YouTube)
- Design a chatting service (e.g., Telegram, Slack, Discord)

- Design a file sharing service (e.g., Google Drive, Dropbox)
- Design a ride sharing service (e.g., Uber, Lyft)
- Design a photo sharing service (e.g., Flickr, Pinterest)
- Design an e-commerce website (e.g., Amazon, eBay)
- Design a jobs portal (e.g., LinkedIn, Indeed)
- Design a web crawler (e.g., Google)

> ⚠️ **System Design Content A Work In Progress**
>
> System design content is still a work-in-progress, but the following are some resources to help you in the meanwhile.

Quality courses

- ByteByteGo: This is a new System Design course by Alex Xu, author of the System Design Interview books, a bestseller on Amazon. The course covers system designs basics, then goes into deep dives of the design of over 10 famous common products (e.g., Designing YouTube, Facebook Newsfeed, etc.) and multiple big data and storage systems (e.g., Designing a Chat System). For each deep dive, concepts are explained and comprehensive diagrams are used, making it very approachable regardless of seniority level.
- "Grokking the System Design Interview" by Design Gurus: This is probably the most famous system design interview course on the internet and what makes it different from most other courses out there is that it is purely text-based, which is great for people who prefer reading over watching videos (such as myself!). It contains a repository of the popular system design problems along with a glossary of system design basics. I've personally completed this course and highly recommended many others to use this.
- "Grokking the Advanced System Design Interview" by Design Gurus: I haven't tried this but it's by the same people who created "Grokking the System Design Interview", so it should be good! In my opinion you probably wouldn't need this unless you're very senior or going for a specialist position.
- "Best of System Design" package by Design Gurus: This bundle allows you to purchase both System Design interview courses by Design Gurus at a discount. Best of all, it's lifetime and not subscription-based.
- "System Design Interview Course" by Exponent: This course covers system designs basics and has a huge database of popular system design questions with videos of mock interviews. Some of the questions have text answers and a database schema and APIs for reference (which I find helpful). While the subscription might be a little pricey for just the system design interviews content, they also offer quality technical content for Data Structures, Algorithms and Behavioral Interviews. The convenience of a one-stop platform which covers all aspects of technical interview preparation is very enticing.
- "Gaurav Sen Youtube - System Design Playlist": Gaurav Sen is a popular YouTuber who has gained a lot of recognition for his system design videos. His playlist on System Design provides a comprehensive guide for software engineers who are preparing for technical interviews or want to learn more about how to design scalable systems. The playlist consists of multiple videos, each video covering a different topic related to system design. The videos are designed in a way that simplifies complex topics and makes it easy to understand for beginners.

Free resources

- System Design Primer: Most comprehensive resource on system design out there. Recommended only if you have a ton of time to spare.
- System Design Interview: Contains many links to tips about system design, system design-related topics and engineering blogs of famous companies.
- System Design Cheatsheet: Brief and concise content. Serves as a good revision right before your system design interview.
- System Design Roadmap: Step-by-step guide that includes links to articles and videos on learning essential topics.

Books

- System Design Interview – An insider's guide, Second Edition: Beginner friendly resource to learn about system design, the content is easy to read and understand.

Behavioral interview prepartion

Step-by-step how to prepare

What are behavioral interviews?

Succeeding in an engineering career involves more than just technical skills. People skills become more important as an engineer becomes more senior. Senior engineers should have the ability to lead and influence, resolve conflicts, anticipate risks, plan the roadmap, and more.

Hiring a talented engineer that cannot work with others can ultimately be a net deficit for companies. Companies don't want to hire brilliant jerks. The company is better off not hiring a very talented engineer who refuses to work with others or causes an entire team to be unproductive. Companies want to hire the right person that will work well with the existing employees and help the team and company achieve greater heights, and behavioral interviews are one way of determining if someone will be good to work with from a non-technical standpoint.

Typically, behavioral interview questions can be split into several types:

1. Getting to know your career preferences, ambitions and plans
2. Discuss details of experiences or projects written in your resume
3. "Tell me about a time where you" type of questions where you describe how you demonstrated certain traits or responded to a situation

Importance of behavioral interviews

Companies value behavioral skills and do evaluate candidates on them. At the time of writing, Facebook has one round (out of four) dedicated to behavioral interviews. Airbnb has TWO rounds (out of six) dedicated to behavioral interviews/company fit. Lyft has one round (out of five) dedicated to past experience and situational questions by a hiring manager.

From the company's perspective, the interview has two purposes:

- Assess whether a candidate has a history of demonstrating the right behaviors that would make them successful at the company.
- Assess the seniority of the candidate (e.g., junior, senior, or staff).

Preparing for behavioral interviews

STAR format

The STAR format is a framework to help you organize answers to behavioral questions — especially ones requiring you to discuss previous experiences

- **Situation:** The interviewer wants you to present a recent challenge and situation which you found yourself in.
- **Task:** What were you required to achieve? The interviewer will be looking to see what you were trying to achieve from the situation. Some performance development methods use "Target" rather than "Task". Job interview candidates who describe a "Target" they set themselves instead of an externally imposed "Task" emphasize their own intrinsic motivation to perform and to develop their performance.
- **Action:** What did you do? The interviewer will be looking for information on what you did, why you did it, and what the alternatives were.

- **Results:** What was the outcome of your actions? What did you achieve through your actions and what did you learn? What steps did you take to improve after the experience?

<details>
<summary>Example of how to apply STAR to a behavioral interview question</summary>

Here's an example of how the STAR format can be used to answer the question: "Tell me about a time in which you had a conflict and needed to influence somebody else".

Situation

> "I was the team lead of a school project about building a social network mobile web app. Our designer's midterms were approaching and didn't have time to produce the mockups. Our front-end person was rushing him for the mockups so that he could proceed with his work, and that was stressing the designer out. The atmosphere in the team was tense."

Task

> "As the team lead, I had to resolve the tension between the front-end developer and the designer so that the team could work together peacefully and complete the project on time."

Action

> "I spoke to the front-end developer to ask him why he was rushing the designer for the designs. He said that he wanted the designs early because it would be a waste of time rebuilding if the designer designed something different eventually. I explained to him that the midterm dates were out of the designer's control and we had to be more understanding about each other's schedules. I spoke to the designer to get a rough idea of what he had in mind and asked him when he could commit to producing the high-fidelity designs. He replied that he could start on them as soon as his midterms were over. I explained to him why the front-end developer was pushing him for the mockups, and that the front-end developer had no ill intentions and simply wanted the project to succeed. As someone with some experience in UI/UX design, I came up with wireframe mocks, ran them by the designer for approval, then passed them to the front-end developer to start building. I encouraged the front-end developer to use placeholders and not be too concerned about the details for now. We could build the non-UI parts first (authentication, hook up with APIs) and tweak pixels and add polish later on. The front-end developer agreed and went ahead with the approach. I explained to the front-end developer that the designer will pass us the mockups after his midterm, by `<DATE>`."

Result

> "When our designer ended midterms, he came back with beautiful mockups that fit well into the wireframes. Our front-end developer implemented them with great care to detail. We ended up scoring top marks for the project and became a great team."

</details>

Preparing answers to common questions

The next natural step is to start preparing your answers for commonly asked behavioral interview questions. You may refer to my list of 30 questions which were collated across top tech companies for this.

While most people might be inclined towards memorization, it's much better to pen down bullet points to each question and practice verbalizing them near to the interviews, so that your answers will come out more naturally.

Prepare experiences to showcase company fit

Prepare experiences to showcase fit to the company's culture / core values. As mentioned previously, most top tech companies use their company values to evaluate candidates in behavioral interviews. As such, you should do your research to find out what those values are and ensure you have prepared experiences that showcase fit.

Try out mock interviews

If you would like to practice behavioral interviews with professional interviewers from top tech companies, schedule one with interviewing.io. Interviewing.io boasts a large pool of interviewers from Facebook, Amazon, Apple, Google and Microsoft. I have used interviewing.io both as an interviewer and interviewee and can guarantee a good experience with this platform.

Use structured courses

I don't really think one needs to attend a course on behavioral interviews, but your mileage may vary. I've seen candidates get rejected for failing the behavioral round even though they did super well on the coding and system design interviews. If you want to take a course on behavioral interviews, I'd recommend the following courses:

- "Behavioral Interviews" by Exponent: While Exponent also has courses on technical content, what really makes them stand out from the other interview preparation platform is their availability of content for non-software engineering roles such as Product Management and Product Marketing. Their behavioral interview course is a mix of videos (by the Exponent CEO himself!) and text, going through the most common questions and imparting you with techniques to help you ace the interview. To top it off, they also have an interview question bank for behavioral questions with responses from the platform's helpful community. While the subscription might be a little pricey for just the behavioral interviews content, they also offer quality technical content for System Design, Data Structures and Algorithms. The convenience of a one-stop platform which covers all aspects of technical interview preparation is very enticing.
- "Grokking the Behavioral Interview" on Educative: Per other courses on Educative, this course is text-based and they believe that text-based courses are the more efficient than video courses. One thing that stands out about this course is that they teach you patterns for behavioral interviews, not just about memorizing questions and preparing answers.

Behavioral interview rubrics

How behavioral interviews are evaluated

Unlike technical interviews, behavioral interviews have a lot more variance in terms of evaluation criteria. However, most top tech companies use their company values to evaluate candidates. Interviewers typically have to fill in a section evaluating how a candidate has displayed behaviors in line with company values. It is for this reason that you should search up a company's culture and values and ensure that you embody them within your answers. You'd also find it useful to speak with one of your connections currently working at the company you are applying for to find out more about which values are typically valued in the team.

Some examples of common values evaluated are:

- **Motivation:** What drives you? Ideal candidates are self-motivated, passionate about technologies and products that have a real impact.
- **Ability to be Proactive:** Are you able to take initiative? Given a difficult problem, are you able to figure out how to get it done and execute on it?
- **Ability to work in an unstructured environment:** How well are you able to take ownership in ambiguous situations? Or do you rely on others to be told what to do?
- **Perseverance:** Are you able to push through difficult problems or blockers?
- **Conflict Resolution:** How well are you able to handle and work through challenging relationships?
- **Empathy:** How well are you able to see things from the perspective of others and understand your motivations?
- **Growth:** How well do you understand your strengths, weaknesses, and growth areas? Are you making a continued effort to grow?
- **Communication:** Are you able to clearly communicate your stories during the interview?

To assess these focus areas, interviewers ask questions on your work history and dig into the details of how you handled various situations. To assess your seniority, for each focus area they determine if the scope of the situation is something they would expect for a junior, senior, or staff engineer (more details on this below).

Example questions and responses

Below are questions and answers illustrating how interviewers collect signals on the candidate for each focus area. In a typical interview, they'll ask the you five or six questions and dive deep into the details of each

situation. Each question may provide signals on more than one focus area.

Motivation

Example Questions:

- What project are you most proud of and why?
- Tell me about a recent day working that was really great and/or fun.

Example Responses:

- Junior: A story about a project they are proud of that had an impact on their team.
- Senior: A story about a project they are proud of that had a large impact on their team.
- Staff: A story about a project they are proud of that had a large impact on their org.

Ability to be proactive

Example Questions:

- Tell me about a time when you wanted to change something that was outside of your regular scope of work.
- Tell me about a time you had to make a fast decision and live with the results.

Example Responses:

- Junior: A story about a change they proactively suggested and drove that had an impact on their team's focus area. Usually only requiring the candidate themselves to work on.
- Senior: A story about a change they proactively suggested and drove that had an impact on their entire team. Usually requiring three or more people to work on.
- Staff: A story about a change they proactively suggested and drove that had an impact on their entire org. Usually requiring two or more teams to work on.

Ability to work in an unstructured environment

Example Questions:

- How do you decide what to work on next?
- Tell me about a project or task that was ambiguous or underspecified.

Example Responses:

- Junior: A story about an ambiguous task that the candidate took ownership of and was able to drive consensus from a few stakeholders in their team. Usually only requiring the candidate themselves to work on.
- Senior: A story about an ambiguous project that the candidate took ownership of and was able to drive consensus from stakeholders in their team or org. Usually requiring three or more people to work on.
- Staff: A story about an ambiguous project that the candidate took ownership of and was able to drive consensus from stakeholders in their org. Usually requiring two or more teams to work on.

Perseverance

Example Questions:

- Tell me about a time when you needed to overcome external obstacles to complete a task or project.
- Tell me about a time a project took longer as expected

Example Responses:

- Junior: A story about a task with many technical difficulties and how they overcame each blocker.
- Senior: A story about a project with many technical difficulties that were blocking their team and how they overcame each blocker.
- Staff: A story about a project with many technical difficulties that were blocking many teams and how they overcame each blocker.

Conflict resolution and empathy

Example Questions:

- Tell me about a person or team who you found the most challenging to work with.
- Tell me about a time you disagreed with a coworker.
- Tell me about a situation where two teams couldn't agree on a path forward.

Example Responses:

- Junior: A story about how they were able to work through a disagreement with a coworker on an implementation detail of a larger project.
- Senior: A story about how they were able to work through a disagreement with a few coworkers or team leads on the direction of a larger project.
- Staff: A story about how they were able to work through a disagreement with two or more teams on the direction of a large project.

Growth

Example Questions:

- Describe a situation when you made a mistake, and what you learned from it.
- Tell me about some constructive feedback you received from a manager or a peer.
- Tell me about a skill set that you observed in a peer or mentor that you want to develop in the next six months.

Example Responses:

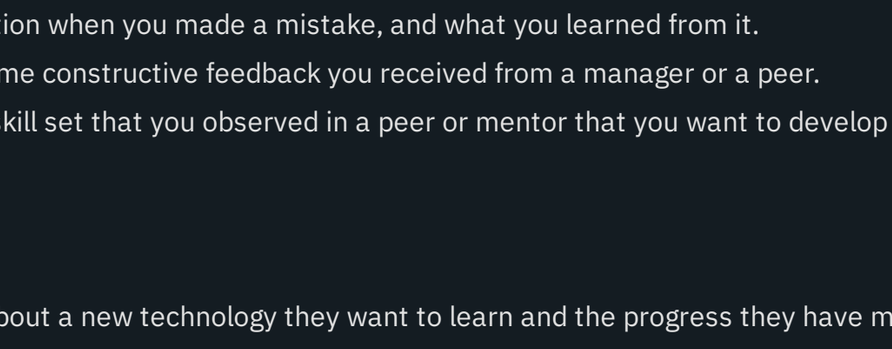
- Junior: A story about a new technology they want to learn and the progress they have made to learn it.
- Senior: A story about a soft skill or technical skill they want to develop and the progress they have made to learn it. Usually a skill that will have the potential to affect the entire team.
- Staff: A story about a soft skill or technical skill they want to develop and the progress they have made to learn it. Usually a skill that will have the potential to affect two or more teams.

Communication

Generally covered during the interview as to how clearly they are explaining the stories. There is also some overlap with Empathy and how they communicate with others.

After the interview

After the interview, the interviewer will compile their feedback and give a hire/no-hire recommendation as well as the candidate's seniority. The hire/no-hire decision is given as a spectrum, ranging from low confidence to high confidence. This is illustrated below:



The recommendation is compiled from the signals the interviewer collected on the eight focus areas. Typically, this looks like the following:

- For junior engineers, a hire recommendation is given if they showed positive signals on nearly all of the eight focus areas.
- For senior engineers, a hire recommendation is given if they showed positive signals on nearly all of the eight focus areas AND at the level we would expect for senior engineers. Otherwise, a no-hire recommendation is given.
- For staff engineers, a hire recommendation is given if they showed positive signals on nearly all of the eight focus areas AND at the level we would expect for staff engineers. Otherwise, a no-hire recommendation is given at a staff level, although they may give a hire recommendation at a senior level.

It's worth mentioning that the interviewer's decision is not final. Once all interview feedback has been collected from all interviews, the interviewers will discuss any concerns or strengths of the candidate with a "debrief committee".

Common behavioral questions to practice

In the software engineer interview process, behavioral interviews may seem so much more varied and unstructured as compared to technical interviews. However, in most cases, the interviewer is actually just trying to get to know you better and there's always a set of common questions that need to be asked to achieve that.

Most common questions across all top tech companies

Here are some of the 30 most commonly asked behavioral interview questions across top tech companies:

- Why do you want to work for X company?
- Why do you want to leave your current/last company?
- What are you looking for in your next role?
- Tell me about a time when you had a conflict with a co-worker.
- Tell me about a time in which you had a conflict and needed to influence somebody else.
- What project are you currently working on?
- What is the most challenging aspect of your current project?
- What was the most difficult bug that you fixed in the past 6 months?
- How do you tackle challenges? Name a difficult challenge you faced while working on a project, how you overcame it, and what you learned.
- What are you excited about?
- What frustrates you?
- Imagine it is your first day here at the company. What do you want to work on? What features would you improve on?
- What are the most interesting projects you have worked on and how might they be relevant to this company's environment?
- Tell me about a time you had a disagreement with your manager.
- Talk about a project you are most passionate about, or one where you did your best work.
- What does your best day of work look like?
- What is something that you had to push for in your previous projects?
- What is the most constructive feedback you have received in your career?
- What is something you had to persevere at for multiple months?
- Tell me about a time you met a tight deadline.
- If this were your first annual review with our company, what would I be telling you right now?
- Time management has become a necessary factor in productivity. Give an example of a time-management skill you've learned and applied at work.
- Tell me about a problem you've had getting along with a work associate.
- What aspects of your work are most often criticized?
- How have you handled criticism of your work?
- What strengths do you think are most important for *your job position*?
- What words would your colleagues use to describe you?
- What would you hope to achieve in the first six months after being hired?
- Tell me why you will be a good fit for the position.

Airbnb

While loving to travel or appreciating Airbnb's growth may be good answers, try to demonstrate the deep connection you have with the product.

- What does "belong anywhere" mean to you?
- What large problems in the world would you solve today?
- Why do you like Airbnb?
- If you had an unlimited budget and you could buy one gift for one person, what would you buy and who would you buy it for?
- If you had an unlimited budget and you could go somewhere, where would you go?
- Share one of your trips with us.
- What is the most challenging project in or out of school that you have worked on in the last 6 months.
- What is something that you don't want from your last internship/job?
- Give me an example of when you've been a good host.
- What's something you'd like to remove from the Airbnb experience?
- What is something new that you can teach your interviewer in a few minutes?
- Tell me about why you want to work here.
- What is the best gift you have ever given or received?
- Tell me about a time you were uncomfortable and how you dealt with it.
- Explain a project that you worked on recently.
- What do you think of Airbnb?
- Tell me something about yourself and why you'd be a good fit for the position.
- Name a situation where you were impressed by a company's customer service.
- How did you work with senior management on large projects as well as multiple internal teams?
- Tell me about a time you had to give someone terrible news.
- If you were a gerbil, which gerbil would you be?
- What excites you about the company?
- How does Airbnb impact our guests and hosts?
- What part of our mission resonates the most with you?

Amazon

- How do you deal with a failed deadline?
- Why do you want to work for Amazon?
- Tell me about a situation where you had a conflict with a teammate.
- In your professional experience have you worked on something without getting approval from your manager?
- Tell me a situation where you would have done something differently from what you actually did.
- What is the most exceedingly bad misstep you've made at any point?
- Describe what Human Resources means to you.
- How would you improve Amazon's website?

ByteDance

- What do you know about `<role>` and why?
- Take me through a product you launched from start to end
- What's the biggest achievement in your previous projects?

- Tell me about a recent failure and what you learned from the experience
- Why do you want to work at ByteDance?
- What makes you a good fit for this position?
- What excites you about the role?

Dropbox

- Talk about your favorite project.
- If you were hired here what would you do?
- State an experience about how you solved a technical problem. Be specific about the diagnosis and process.

Hired

- Tell me about yourself.
- What is your biggest strength and area of growth?
- Why are you interested in this opportunity?
- What are your salary expectations?
- Why are you looking to leave your current company?
- Tell me about a time your work responsibilities got a little overwhelming. What did you do?
- Give me an example of a time when you had a difference of opinion with a team member. How did you handle that?
- Tell me about a challenge you faced recently in your role. How did you tackle it? What was the outcome?
- Where do you want to be in five years?
- Tell me about a time you needed information from someone who wasn't responsive. What did you do?

Lyft

- Tell me about your most interesting/challenging project to date.
- Why Lyft? What are you looking for in the next role?

Palantir

- What is something 90% of people disagree with you about?
- What is broken around you?
- How do you deal with difficult coworkers? Think about specific instances where you resolved conflicts.
- How did you win over the difficult employees?
- Tell me about an analytical problem that you have worked on in the past.
- What are your three strengths and three weaknesses?
- If you were in charge of picking projects for Palantir, what problem would you try to solve?
- What are some of the best and worst things about your current company?
- What would your manager say about you?
- Describe Palantir to your grandmother.
- Teach me something you've learned.
- Tell me a time when you predicted something.
- If your supervisors were to rate you on a scale of 1-10, what would they rate you?
- What was the most fun thing you did recently?
- Tell me the story of how you became who you are today and what made you apply to Palantir.

Slack

- Tell me something about your internship.
- Why do you want to join Slack?
- Tell me about your past projects.
- Explain me your toughest project and the working architecture.
- Apart from technical knowledge, what did you learn during your internship?
- If someone has a different viewpoint to do a project like different programming language, how would handle this situation?
- What are your most interesting subjects and why?
- Did you find any bugs in Slack?
- What is your favorite feature and why?

Stack Overflow

- What have you built?
- What is the hardest technical problem you have run into? How did you solve it?
- Where do you see yourself in 5 years?
- Why do you want to work here?
- How do you handle disagreements with co-workers?

Stripe

- How do you stay up to date with the latest technologies?
- Explain a project that you worked on recently that was difficult.
- Where do you see yourself in five years?

Twitter

- What would your previous boss say your biggest strength was?

Preparing a self introduction

"Tell me about yourself" or "give me a quick introduction of your profile" is almost always the first question encountered in your software engineer interviews. This guide teaches you how to maximize this chance to impress the interviewer by crafting the perfect self introduction.

Interviewers want to work with candidates they like. Leave a good/deep impression and it will increase your chances of success. Most of us are not strangers to self introductions as we meet new people now and then and have to introduce ourselves every once in a while. However, self introductions in interviews are slightly different from real life — you need to tweak it to your advantage — tailor the self introduction to the role and company you are applying for! Your self introduction evolves as you grow and are at a different stage of your career.

When answering "tell me about yourself", you can rephrase the question into:

> "Tell me about your journey into tech. How did you get interested in coding, and why was web development (or replace with other job-specific skills) a good fit for you? How is that applicable to our role or company goals?" It is probably not a good idea to spend valuable time talking about things which aren't relevant to the job!

Make an elevator pitch

An "elevator pitch" originates from a journalist trying to pitch an idea to an editor. The only time to catch the editor was in the elevator and she had only around 30 seconds to do so. The key elements of elevator pitches include:

- **Short:** You have limited time!
- **Direct:** As you only have limited time, you should get to the point
- **Attention-grabbing:** Present your most attractive ideas

Whether you're at a job fair with hundreds of other candidates and you have limited time or you are simply explaining who you are to a potential connection or client, it is important to be able to clearly and accurately

describe your knowledge and skills quickly and succinctly. Your self introduction is an elevator pitch for yourself!

Here are some tips to develop a good elevator pitch for your software engineer self introduction:

Start with basic background information

Include who you are, who you work for (or school and major), and what you do.

- **Internships:** You should mention the following: name, school and major, focus areas, past internships and/or noteworthy projects
- **Full-time:** You should mention the following: name, past companies, noteworthy projects (best if it's a public consumer product that they would have heard of)

Does this look familiar? It should be, because it is similar to your resume! Your resume is a condensed version of your knowledge and experiences and your self introduction is essentially a condensed version of your resume. As you grow older, professional experience becomes more important and school background becomes less important. Hence your self introduction changes as you become more senior.

KISS (Keep It Simple and Sweet)

Tell them some highlights from your favorite/most impressive projects and including some numbers if they're impressive or challenges that you've overcome. Do not delve into the depths of how you reverse engineered a game and decrypted a packet to predict when to use your DKP on a drop. Tell them the executive summary: "I reverse engineered X game by decrypting Y packet to predict Z." If this catches their interest, they might ask further questions on their own.

Why do they want you?

Tell the interviewer why you would make a good hire. Is your experience relevant to the company? Have you used a similar tech stack as the company or built relevant products? What unique talent(s) do you have that may give them confidence about your ability to contribute to the company?

Practice!

Lastly, you must practice your pitch! Having a great, succinct summary of your skills only helps if you can actually deliver it rapidly! You should practice keeping a quick but easy-to-follow pace that won't overwhelm them but won't bore them. It's a precarious balance, but can be ironed out with practice.

After coming up with your self introduction, keep it somewhere where you can refer/tweak in future. Memorize them and in future you can just use it when you need to but don't sound like you're recalling it from your memory when you're actually saying it out. Sound natural!

Having an elevator pitch on hand is a great way to create a network and chance upon new job opportunities. There will often be times when you can't prepare for an interview or meeting and it is incredibly handy to have a practiced pitch.

Good examples

Front end engineer at Meta

Self introduction

> "Hi I'm XXX and I graduated from National University of Singapore in 2015 with a degree in Computer Science. My interests are in Front End Engineering and I love to create beautiful and performant products with delightful user experiences.
>
> Back in school, I designed and built a web application, NUSMods which solves a huge problem of class and timetable planning every semester. It receives over a million pageviews a month and is used by over 40,000 NUS students and even some professors. It is built using a modern web technology stack — React, Redux, Jest, Babel, Flow, webpack and is mobile-responsive."
>
> I'm interested in the Front End Engineer role at Meta because I have been using Meta Open Source Front End technologies for a while now and am inspired by Meta's mission and Open Source culture.

Breakdown

> "I love to create beautiful and performant products with delightful user experiences."

Qualities that a Front End engineer should possess.

> "It receives over a million pageviews a month and is used by over 30,000 NUS undergraduates and even some professors."

Mention something about the project which stands out.

> "It is built using a modern web technology stack — React, Redux, Jest, Babel, Flow, webpack and is mobile-responsive."

Meta tech stack! Also hints that you keep yourself updated with modern web technologies.

Front end engineer at Lyft

Self introduction

> "Hi I'm XXX and I graduated from National University of Singapore in 2015 with a degree in Computer Science. My interests are in Front End Engineering and I love to create beautiful performant products with delightful user experiences.
>
> I previously worked at Grab where I led the Grab for Work project. Grab for Work was a service for companies to make corporate transportation expenses convenient. Companies can create employee groups, set ride policies and share corporate payment methods with their employees. I built the project with another engineer over the period of 3 months on a React/Redux and Golang stack."
>
> I'm interested in the Front End Engineer role at Lyft because I like working in this ridesharing space and creating products to improve the lives of users.

Breakdown

> "I love to create beautiful and performant products with delightful user experiences."

Same as above, qualities that a Front End engineer should possess.

> "I previously worked at Grab where I led the Grab for Work project."

Lyft was Grab's sister company! In fact they even had a partnership in the past. Most Lyft engineers would have heard of Grab before and mentioning this catches their attention.

> "I built the project with another engineer over the period of 4 months on a React/Redux and Golang stack."

Acknowledge that you work with others. Building a non-trivial system with just 2 people in 3 months is quite good for a non-trivial system. Lyft also uses Golang for their high performance systems.

Preparing final questions to ask

Something you can always count on to happen at the end of your Software Engineer interview — both technical and non-technical rounds — is for the interviewer to ask you if you "have any final questions?".

This question actually isn't a real question at all — candidates are generally expected to ask questions. As an interviewer myself, candidates who don't have any questions might come off as less interested in the role.

Besides that, the questions you ask reveal what you care about. If asked well, this can be a very predictable opportunity for you to leave a good impression while also knowing more about the role (including uncovering

potential red flags).

Here, I present questions to ask at the end of your software engineer interviews, for every purpose. The ones in **bold** are the ones that tend to make the interviewer go "That's a good question" and pause and think for a bit.

Technical work questions

- **What are the engineering challenges that the company/team is facing?**
- **What has been the worst technical blunder that has happened in the recent past? How did you guys deal with it? What changes were implemented afterwards to make sure it didn't happen again?**
- **What is the most costly technical decision made early on that the company is living with now?**
- **What is the most fulfilling/exciting/technically complex project that you've worked on here so far?**
- **I do/don't have experience in domain X. How important is this for me to be able to succeed?**
- How do you evaluate new technologies? Who makes the final decisions?
- How do you know what to work on each day?
- How would you describe your engineering culture?
- How has your role changed since joining the company?
- What is your stack? What is the rationale for/story behind this specific stack?
- Do you tend to roll your own solutions more often or rely on third party tools? What's the rationale in a specific case?
- How does the engineering team balance resources between feature requests and engineering maintenance?
- What do you measure? What are your most important product metrics?
- How often have you moved teams? What made you join the team you're on right now? If you wanted to move teams, what would need to happen?
- What resources does the company have for new hires to study its product and processes? Are there specifications, requirements, documentation?
- How do you think my expertise would be relevant to this team? What unique value can I add?

Role questions

- **What qualities do you look out for when hiring for this role?**
- **What would be the most important problem you would want me to solve if I joined your team?**
- What does a typical day look like in this role?
- What are the strengths and weaknesses of the current team? What is being done to improve upon the weaknesses?
- What resources does the company have for new hires to study its product and processes? Are there specifications, requirements, documentation?
- What would I work on if I joined this team and who would I work most closely with?

Culture and welfare questions

- **What is the most frustrating part about working here?**
- **What is unique about working at this company that you have not experienced elsewhere?**
- **What is something you wish were different about your job?**
- How is individual performance measured?
- What do you like about working here?
- What is your policy on working from home/remotely?
- What does the company do to nurture and train its employees?
- Does the company culture encourage entrepreneurship and creativity? Could you give me any specific examples?

Leadership and management questions

These questions are suitable for asking Engineering Managers or senior level management, such as CEO, CTO, VPs and are especially useful for the Team Matching phase of Google interviews or post-offer calls that your recruiters set up with the various team managers.

- **How do you train/ramp up engineers who are new to the team?**
- **What does success look like for your team/project?**
- **What are the strengths and weaknesses of the current team? What is being done to improve upon the weaknesses?**
- **Can you tell me about a time you resolved an interpersonal conflict?**
- How did you become a manager?
- How do your engineers know what to work on each day?
- What is your team's biggest challenge right now?
- How do you measure individual performance?
- How often are 1:1s conducted?
- What is the current team composition like?
- What opportunities are available to switch roles? How does this work?
- Two senior team members disagree over a technical issue. How do you handle it?
- Have you managed a poor performer at some point in your career before? What did you do and how did it work?
- Where do you spend more of your time, high performers or low performers?
- Sometimes there's a trade-off between what's best for one of your team members and what's best for the team. Give an example of how you handled this and why.
- Give an example of a time you faced a difficult mentoring/coaching challenge. What did you do and why?
- What is your management philosophy?
- What is the role of data and metrics in managing a team like ours?
- What role does the manager play in making technical decisions?
- What is an example of a change you have made in the team that improved the team?
- What would be the most important problem you would want me to solve if I joined your team?
- What opportunities for growth will your team provide?
- What would I work on if I joined this team and who would I work most closely with?

Company direction questions

- **How does the company decide on what to work on next?**
- What assurance do you have that this company will be successful?
- Which companies are your main competitors and what differentiates your company?
- What are your highest priorities right now? For example, new features, new products, solidifying existing code, reducing operations overhead?

Salary and offer negotiation preparation

Understanding compensation

Compensation is one of the largest factors when it comes to deciding between job offers. This section gives you a breakdown of the common components of compensation in the tech industry.

In most companies, your compensation will consist of base salary, a performance bonus and equity/stocks. For compensation data, check out Levels.fyi.

Base salary

Base salary is a fixed amount of salary you get for showing up at work and is unaffected by how well the company is performing or the industry is doing. It is the only non-variable component of your compensation.

Fresh graduates in the Bay Area can usually expect to get a base salary of above USD 100,000 (before taxes). The salary for fresh graduates at Facebook/Google is known to be in the USD 100,000 - 150,000 range.

Startups usually offer a bit higher for fresh graduate (USD 120,000 - 130,000) to make up for the lack of liquidity of the equity grant (not yet real cash).

Base salary doesn't increase linearly as you become more senior; it will taper off eventually. As employees become increasingly senior within the company, the higher the proportion of their compensation comes from company-dependent factors such as bonus and equity. This is because senior employees are expected to influence the people around them and drive the company forward. A senior employee's performance will be based on how well the company does as a whole; the individual factor will not be as much.

There are some exceptions to the system. Companies like Netflix pay top of the industry (sometimes even more than Facebook/Google) and give employees the option to receive their entire compensation as base salary, that is, to convert the equity component into their base. For the risk-adverse, this is a great choice.

Bonus

Bonuses are usually paid on a semi-annual basis and are typically dependent on a few factors — level of seniority, individual performance in that time period, company performance in that time period.

- **Level of seniority:** This is usually a multiplier of the base salary and the multiplier increases as the employee moves up the ranks.
- **Individual performance:** This is a multiplier of how well an employee performed in that time period (e.g., meeting expectations results in a `100%` multiplier and exceeding expectations results in a `>100%` multiplier). Companies like Facebook and Apple reward their top performers handsomely, and the multiplier can go up to `300%` for the extremely high-performing employees.
- **Company performance:** How well the company is doing. This multiplier will be the same for all employees.

<details>
<summary>Examples</summary>

Bob is a Software Engineer fresh out of college. His base salary is $100,000, is a fresh grad (seniority multiplier: 10%), crushed expectations for the half (individual performance multiplier: 200%) and his company did pretty well (company performance multiplier: 120%). For that half, his bonus will be as follows:

```
Bonus:
100,000 x 50% (half a year)
        x 10% (seniority)
        x 200% (individual performance)
        x 120% (company performance)
= 12000
```

Alice is an Engineering Manager with 10 years of professional experience. Her base salary is $220,000, is an experienced engineering manager (seniority multiplier: 20%), exceeded expectations for the half (individual performance multiplier: 150%) and her company did pretty awesome (company performance multiplier: 130%). For that half, her bonus will be as follows:

```
Bonus:
220,000 x 50% (half a year)
        x 20% (seniority)
        x 150% (individual performance)
        x 130% (company performance)
= 42900
```

Hence the amount of bonus you receive can be highly variable and senior employees get a higher proportion of their total compensation from bonuses.
</details>

Equity/stocks

Equity is what differentiates a tech job from a non-tech one. Equity means a share of the company; this signifies ownership and motivates employees to work in the best interests of the company. They can be a significant portion of one's compensation, sometimes even more than the base salary, especially for senior employees.

Equity usually vests (becomes available to you) over a period of time (typically 4 years) and can vest equally every month/quarter/year. A vesting cliff means the minimum period of time before your vesting begins. For example, if you are granted 4,800 shares over a 4 year schedule with a 1 year cliff, and monthly vesting, you will get 1,200 shares at the end of your first year and 100 shares every month thereafter for the subsequent 3 years.

Until the company goes public (IPO) or gets acquired, the equity is usually not worth anything. However, there are instances where you can sell your stocks internally even though the company hasn't gone public. Be mindful of what you are getting yourself into!

Not all equity is treated equally. Depending on the company you join and which stage that company is at, you may receive one of the following types: stock options or stock grants.

Stock options

Stock options are typically given by mid-stage companies. Stock options are different from stocks, which represents the immediate ownership of a company! Stock options are the **option/right** to purchase stocks at a given strike price, hence it's not free. However, the cost of each stock is usually quite low and fixed at a strike price, which is equal to the fair market value of the stock when it was granted to you. You are guaranteed to be able to purchase the stock at that price regardless of future increases.

When you leave a company, there is an exercise window (deadline given for you to exercise your options before they are gone), so it is important to have enough liquid cash to purchase them when planning a departure.

Stock grants

A stock grant is commonly referred to as a Restricted Stock Unit (RSU) and it means you possess the stock immediately. If the company is public, you can sell them during defined trading windows.

More reading on the topic can be done on the equity compensation guide and the article "What I Wish I'd Known About Equity Before Joining A Unicorn".

Signing bonus

This is a one-time lump sum that is paid to you when you join a company. This amount is typically in the range of USD 10,000 to USD 20,000 but can even go up to USD 50,000 (Google) and USD 100,000 (Facebook).

There can be conditions attached to signing bonuses, such as having to return a pro-rated amount if an employee leaves before the one-year mark. Make sure you are aware of them before you sign the offer.

Misc bonuses/perks

While these perks are not exactly cash, they can help you save money which is almost equivalent to getting compensated more. Do find out more about these from your recruiters if you get the chance.

- **Free meals:** Food is not exactly cheap in the Bay Area and having some meals provided on weekdays can result in saving few thousand dollars a year and hours spent on travelling out to get food.
- **Relocation bonus:** Helpful if you are moving from abroad. This can partially offset costs incurred during relocation.
- **Health and dental insurance/plans:** Companies often partner with insurance companies to provide employees with health and dental plans. These can amount to a few thousand dollars worth annually and is especially useful in locations where healthcare is expensive.
- **Shuttle service:** Public transportation in the Bay Area is not that great and the most common form of commute is driving. Being able to take a shuttle service helps in saving money on gas, transport, and freeing up your mind to do other things during the commute.

Complete salary negotiation guide

Always negotiate

If you've received an offer (or even better, offers), congratulations! You may heave a huge sigh of relief and think that the toughest parts are over. Well yes, but not entirely! For most people, the reason they're finding a new job is to increase their salary, and salary negotiation is the last stretch in achieving that goal.

Here's something that recruiters don't want you to know — In most cases, there's room for negotiation on your offer and recruiters expect candidates to negotiate. **The initial offer that you are given is never the best package that the company can offer.** During my last job hunting experience, I received offers from numerous top tech companies like Facebook, Google, Airbnb, Lyft, Dropbox, and I have found this to be true. In most cases, you could always negotiate for more money, and some aspects of your salary is easier to negotiate than others. With multiple offers in hand, I was able to negotiate a better offer from every company.

Negotiation services

If you haven't been negotiating your past offers, or are new to the negotiation game, worry not! There are multiple negotiation services that can help you out. Typically, they'd be well-worth the cost. Had I know about negotiation services in the past, I'd have leveraged them!

Rora

How Rora works is that you will be guided by their experienced team of professionals throughout the entire salary negotiation process. It's also risk-free because you don't have to pay anything unless you have an increased offer. It's a **no-brainer decision** to get the help of Rora during the offer process — some increase is better than no increase. Don't leave money on the table! Check out Rora.

Things Rora can do for you:

- Help you to negotiate increases even without competing offers
- Provide tailored advice through their knowledge of compensation ranges at many companies
- Provide you with customized scripts on what and how to talk to recruiters covering a range of scenarios
- Mock negotiation services to help you practice
- Provide you with live guidance during the recruiter call through chat
- Introduce you to recruiters at other companies

Book a free consultation with Rora if you want to try it out.

Levels.fyi

Levels.fyi is most famously known for being a salary database but they also offer complementary services such as salary negotiation where you will be put in-touch with experienced recruiters to help you in the process. How Levels.fyi differs from Rora is that Levels.fyi charges a flat fee whereas Rora takes a percentage of the negotiated difference.

Negotiation courses

If you are not keen on paying for negotiation services and are a fan of courses, here's something for you — the Grokking Comp Negotiation in Tech course, a text-based course where you can get information on each aspect of negotiation. **Disclaimer: I haven't tried this course before.**

Ten rules of negotiation

Key points extracted from "Ten Rules for Negotiating a Job Offer" by Haseeb Qureshi:

- Part 1
- Part 2

Get everything in writing

Note down EVERYTHING on your phone call with the recruiters as they may be helpful later on. Even if there are things that are not directly monetary, if they relate to the job, write them down. If they tell you "we're working on porting the front-end to Angular," write that down. If they say they have 20 employees, write that down. You want as much information as you can. You'll forget a lot of this stuff, and it's going to be important in informing your final decision.

Always keep the door open

Never give up your negotiating power until you're absolutely ready to make an informed, deliberate final decision. This means your job is to traverse as many of these decision points as possible without giving up the power to continue negotiating. Very frequently, your interlocutor will try to trick you into making a decision, or tie you to a decision you didn't commit to. You must keep verbally jiu-jitsu-ing out of these antics until you're actually ready to make your final decision.

Information is power

To protect your power in the negotiation, you must protect information as much as possible. A corollary of this rule is that you should not reveal to companies what you're currently making. So given this offer, don't ask for more money or equity or anything of the sort. Don't comment on any specific details of the offer except to clarify them. Companies will ask about your current compensation at different stages in the process — some before they ever interview you, some after they decide to make you an offer. But be mindful of this, and protect information.

> "Yeah, `[COMPANY_NAME]` sounds great! I really thought this was a good fit, and I'm glad that you guys agree. Right now I'm talking with a few other companies so I can't speak to the specific details of the offer until I'm done with the process and get closer to making a decision. But I'm sure we'll be able to find a package that we're both happy with, because I really would love to be a part of the team."

Always be positive

Even if the offer is bad, it's extremely important to remain positive and excited about the company. This is because your excitement is one of your most valuable assets in a negotiation.

Despite whatever is happening in the negotiation, give the company the impression that 1) you still like the company, and that 2) you're still excited to work there, even if the numbers or the money or the timing is not working out. Generally the most convincing thing to signal this is to reiterate you love the mission, the team, or the problem they're working on, and really want to see things work out.

Don't be the decision maker

Even if you don't particularly care what your friends/family/husband/mother thinks, by mentioning them, you're no longer the only person the recruiter needs to win over. There's no point in them trying to bully and intimidate you; the "true decision-maker" is beyond their reach. This is a classic technique in customer support and remediation. It's never the person on the phone's fault, they're just some poor schmuck doing their job. It's not their decision to make. This helps to defuse tension and give them more control of the situation.

> I'll look over some of these details and discuss it with my `[FAMILY/CLOSE_FRIENDS/SIGNIFICANT_OTHER]`. I'll reach out to you if I have any questions. Thanks so much for sharing the good news with me, and I'll be in touch!

It's much harder to pressure someone if they're not the final decision-maker. So take advantage of that.

Have alternatives

If you're already in the pipeline with other companies (which you should be if you're doing it right), you should proactively reach out and let them know that you've just received an offer. Try to build a sense of urgency. Regardless of whether you know the expiration date, all offers expire at some point, so take advantage of that.

> Hello `[PERSON]`,
>
> I just wanted to update you on my own process. I've just received an offer from `[COMPANY]` which is quite strong. That said, I'm really excited about `[YOUR AMAZING COMPANY]` and really want to see if we can make it work. Since my timeline is now compressed, is there anything you can do to expedite the process?

Should you specifically mention the company that gave you an offer? Depends. If it's a well-known company or a competitor, then definitely mention it. If it's a no-name or unsexy company, you should just say you received an offer. If it's expiring soon, you should mention that as well.

Either way, send out a letter like this to every single company you're talking to. No matter how hopeless or pointless you think your application is, you want to send this signal to everyone who is considering you in the market.

Companies care that you've received other offers. They care because each company knows that their own process is noisy, and the processes of most other companies are also noisy. But a candidate having multiple offers means that they have multiple weak signals in their favor. Combined, these converge into a much stronger signal than any single interview. It's like knowing that a student has a strong SAT score, and GPA, and won various scholarships. Sure, it's still possible that they're a dunce, but it's much harder for that to be true.

This is not to say that companies respond proportionally to these signals, or that they don't overvalue credentials and brands. They do. But caring about whether you have other offers and valuing you accordingly is completely rational.

Tell other companies that you've received offers. Give them more signals so that they know you're a valued and compelling candidate. And understand why this changes their mind about whether to interview you.

Your goal should be to have as many offers overlapping at the same time as possible. This will maximize your window for negotiating.

Have a strong BATNA (Best Alternative To a Negotiated Agreement) and communicate it.

> I 've received another offer from `[OTHER CORP]` that's very compelling on salary, but I really love the mission of `[YOUR COMPANY]` and think that it would overall be a better fit for me.

> I'm also considering going back to grad school and getting a Master's degree in Postmodern Haberdashery. I'm excited about `[YOUR COMPANY]` though and would love to join the team, but the package has to make sense if I'm going to forego a life of ironic hatmaking.

Proclaim reasons for everything

It's kind of a brain-hack, both for yourself and for your negotiating partner. Just stating a reason (any reason) makes your request feel human and important. It's not you being greedy, it's you trying to fulfill your goals.

The more unobjectionable and sympathetic your reason, the better. If it's medical expenses, or paying off student loans, or taking care of family, you'll bring tears to their eyes.

Just go with it, state a reason for everything, and you'll find recruiters more willing to become your advocate.

Be motivated by more than just money

You should be motivated by money too of course, but it should be one among many dimensions you're optimizing for. How much training you get, what your first project will be, which team you join, or even who your mentor will beem these are all things you can and should negotiate.

Of course, to negotiate well you need to understand the other side's preferences. You want to make the deal better for both of you.

Understand what they value

Remember that you can always get salary raises as you continue to work at the company, but there's only one point at which you can get a signing bonus.

The easiest thing for a company to give though is stock (if the company offers stock). Companies like giving stock because it invests you in the company and aligns interests. It also shifts some of the risk from the company over to you and burns less cash.

Be winnable

This is more than just giving the company the impression that you like them (which you continually should). But more so that you must give any company you're talking to a clear path on how to win you. Don't bullshit them or play stupid games. Be clear and unequivocal with your preferences and timeline.

Don't waste their time or play games for your own purposes. Even if the company isn't your dream company, you must be able to imagine at least some package they could offer you that would make you sign. If not, politely turn them down.

Choosing between companies

If you have passed the interviews and received multiple offers, congratulations on the feat! Now you are faced with yet another problem (albeit a good one!) of choosing between companies. Here we will provide some insights into the factors you should be considering when deciding on the company to join. Every individual is different and certain factors matter more/less to others, so only you can decide what is best for you.

Do your due diligence by researching on the companies before deciding!

Compensation

First and foremost, compensation. Most technical roles at tech companies would receive compensation in similar components — Base salary, Bonuses, Stocks, and Sign On. Some companies are more flexible in their offer and allow candidates to choose between higher base salary or higher stocks. Base salary is the most straightforward to understand and you are being paid that same amount regardless of how well/poorly the company is doing. Bonuses are also usually a percentage of the base salary, so a high base salary is great for the risk-adverse.

Not all stock grants are equal as well. Some companies have linear vesting cycles (you vest the same amount every year), some companies like Amazon and Snap have backloaded schemes (you vest less in the earlier years, more later), and pay attention to cliffs as well. Stripe and Lyft recently changed their stock structure and announced that they will speed up equity payouts to the first year. This sounds good initially, but in reality there are some nuances.

Regardless of company, **always negotiate** your offer, especially if you have multiple offers to choose from! Having multiple offers in hand is the best bargaining chip you can have for negotiation and you should leverage it. Use Rora for risk-free negotiation services.

Products

What does the company work on and is the company working on a domain you are interested in? Most big tech companies have teams working on all sorts of products. For these big tech companies, common products include ads networks, chat, enterprise offerings, video watching, payments, hardware products, industry-leading AI tools, internal tools, etc. Microsoft, Amazon, Apple and ByteDance have products in some of these areas as well. For many big companies, they are large enough that they build their own infra and internal tools for most technologies they use because most existing technologies on the market can't meet their scale.

That said, each tech company is uniquely known for something and it's in their DNA. So if you are interested in specific areas and you are sure you can be able to work on them, the choice should be clearer:

- Meta/Facebook: Social networks (Facebook, Instagram), chat (WhatsApp, Messenger), Metaverse stuff (Oculus)
- Google: Search, browsers (Google Chrome), Google Maps, cloud infrastructure (Google Cloud Platform),

enterprise collaboration (Google Suite)
- Amazon: Cloud infrastructure (Amazon Web Services), e-commerce

- Microsoft: Operating systems, Microsoft Office Suite
- Apple: Hardware, operating systems, services (iCloud, Apple Music)

Personally I feel more motivated and productive if I'm working on products which I use as an external user as well, as I have a better understanding of the product from a user perspective which can potentially lead to better engineering and product decisions.

While a company can be working on various products, some products are only worked on in certain offices. For example, Google Singapore primarily works on Payments and if you're only interested in working on search, Google Singapore wouldn't be that great of a fit, although it is definitely a possibility to relocate to another country in the future while still remaining as a Google employee.

Do find out what products and teams your intended location works on so that you have a better idea of how your potential working life looks like if you were to join that company.

Company prospects

If you are awarded stocks, the company's prospects will affect your compensation! Is the company poised to be successful in the coming years or are there factors hindering the company's progress?

Career growth

How fast does the company promote its employees and does the company provide opportunities for its employees to grow? In smaller companies, you will get to wear multiple hats and build many products from 0 to 1. It's great for people who like to grow their product building skills and launch products quickly. However, technical growth might be limited due to the company's focus on product work and less on building infrastructure, which is usually only needed at scale. There's little need for a Principal Engineer in a small startup of 10 people where everyone is scrambling to build products. Career growth might be limited by the size and scope of the company and its product offerings.

Some companies promote their employees faster than the others. There's a saying that go to Google to rest and vest, go to Meta if you want to accelerate your career. I've found this to be quite accurate from what I experienced and from looking at my peers at Google. At Meta, engineers are required to get from E3 to E4 within 2 years and E4 to E5 within 3 years. On the other hand, Google has a slower promotion cycle — the average engineer at Google takes more than 2 years to get from L3 to L4, more than 3 years to get from L4 to L5 and more than 4 years to get from L5 to L6. At Google, the terminal level is L4 so there's no pressure to promote. None of my peers are L6 at Google but a few are already E6 at Meta and it is quite achievable. In my opinion, Google is a great place to learn to be excellent engineers. They require everything to be well-engineered, with design docs and high test coverage. Meta is a great place for career-driven folks who don't mind working harder to accelerate their career path.

Company culture

Company culture refers to the beliefs and behaviors of a company's leadership and its employees. The tech giants are famous for their unique cultures:

- **Google:** Candidates are evaluated for their "Googleyness" factor during interviews, which is a set of traits Google employees should embody — "Doing the right thing", "Striving for excellence", "Comfort with ambiguity". As mentioned above, Google is known for their high emphasis on engineering quality and data-driven decision making approach.
- **Meta/Facebook:** In Facebook's early days, their internal motto was "Move fast and break things", indicating their focus on shipping products fast. In 2014, Mark Zuckerberg changed it to "Move fast with stable infrastructure" as the platform has matured and stability should be a focus. I wrote more about Meta/Facebook's culture in a blog post.
- **Netflix:** Netflix's culture deck is publicly available on their website and their core philosophy is **people over process**. They like to think of themselves as professional sports teams — keeping the star players and letting adequate performers go instead of operating like a family — tolerating bad behavior and showing unconditional love.

Engineering culture

Since most readers are in tech roles, the tech culture of a company deserves special mention.

Gergely Orosz came up with 12 points to evaluate whether a tech company has a healthy software engineering culture

1. **Equity or profit sharing** — Do employees receive equity?
2. **Roadmap/backlog that engineers contribute to** — Do engineers contribute to the roadmap of their team?
3. **Engineers directly working with other ICs** — Do engineers work directly with other roles (Designers, PMs, Data Scientists, etc)?
4. **Code reviews and testing** — Do engineers peer code reviews and write tests often?
5. **CI and engineers pushing to prod** — Is there continuous integration or a way for engineers to deploy to production?
6. **Internal open source** — Can engineers access and contribute to code bases across the company?
7. **Healthy oncall as a priority** — Is the oncall load manageable?
8. **Technical managers** — Do the engineering managers possess technical background and skills?
9. **Career ladder (when above 10 engineers)** — Are career ladders and expectations for each level well-defined?
10. **Parallel IC & manager tracks (when above 30 engineers)** — Can one rise up the career ladder as an IC?
11. **Feedback culture** — Is it the practice to give feedback to each other/the company?
12. **Investing in professional growth** — Stipend for professional growth, mentorship program

It's no surprise that the FAANG companies hit all 12 points. Read more about it on his blog post.

Which company's culture resonates with you the most?

Work-life balance

While the common practice is for employees to work 40 hours a week and 8 hours a day, some places are infamous for being more stressful and asks more time from their employees. Some Chinese companies famously require employees to work 6 days a week. Most tech employees do not get paid when they work overtime, so sometimes a more accurate way of calculating your salary is dollars per hour instead of the raw annual total compensations.

Google is famous for being a company for moving slower and a place where people go to rest and vest (their stocks) while Amazon scored a C- for work-life balance in an employee survey.

Transfers and mobility

Does the company have offices around the globe where employees can possibly relocate to? This is obviously out of the question for smaller companies where there is only one headquarters, but some companies allow employees to be remote and some are even entirely remote! Meta, Google, Apple, and Stripe are examples of companies which have global presence and regional headquarters. At Meta and Google, mobility is extremely high given there are large engineering offices all across US, in London, Tel Aviv, and more recently, in Singapore. One of my managers at Meta has worked at **four** offices! Personally, I relocated from Meta/Facebook Menlo Park to Meta/Facebook Singapore right before COVID hit and a few of my Meta/Googler friends have done similar moves, some even moving from Singapore to the US.

Algorithms study cheatsheets

Introduction

What is this

This section dives deep into practical knowledge and techniques for algorithms and data structures which appear frequently in algorithm interviews. The more techniques you have in your arsenal, the higher the chances of passing the interview. They may lead you to discover corner cases you might have missed out or even lead you towards the optimal approach!

Contents of each study guide

For each topic, you can expect to find:

1. A brief overview
2. Learning resources
3. Language-specific libraries to use
4. Time complexities cheatsheet
5. Things to look out for during interviews
6. Corner cases
7. Useful techniques with recommended questions to practice

Study guides list

Here is the list of data structures and algorithms you should prepare for coding interviews and their corresponding study guides:

Topic	Priority
Array	High
String	High
Hash Table	Mid
Recursion	Mid
Sorting and searching	High
Matrix	High
Linked List	Mid
Queue	Mid
Stack	Mid
Tree	High
Graph	High
Heap	Mid
Trie	Mid
Interval	Mid
Dynamic programming	Low
Binary	Low
Math	Low
Geometry	Low

General interview tips

Clarify any assumptions you made subconsciously. Many questions are under-specified on purpose.

Always validate input first. Check for invalid/empty/negative/different type input. Never assume you are given the valid parameters. Alternatively, clarify with the interviewer whether you can assume valid input (usually yes), which can save you time from writing code that does input validation.

Are there any time/space complexity requirements/constraints?

Check for off-by-one errors.

In languages where there are no automatic type coercion, check that concatenation of values are of the same type: `int`/`str`/`list`.

After finishing your code, use a few example inputs to test your solution.

Is the algorithm meant to be run multiple times, for example in a web server? If yes, the input is likely to be preprocess-able to improve the efficiency in each call.

Use a mix of functional and imperative programming paradigms:

- Write pure functions as much as possible.
- Pure functions are easier to reason about and can help to reduce bugs in your implementation.
- Avoid mutating the parameters passed into your function especially if they are passed by reference unless you are sure of what you are doing.
- However, functional programming is usually expensive in terms of space complexity because of non-mutation and the repeated allocation of new objects. On the other hand, imperative code is faster because you operate on existing objects. Hence you will need to achieve a balance between accuracy vs efficiency, by using the right amount of functional and imperative code where appropriate.
- Avoid relying on and mutating global variables. Global variables introduce state.
- If you have to rely on global variables, make sure that you do not mutate it by accident.

Generally, to improve the speed of a program, we can either: (1) choose a more appropriate data structure/algorithm; or (2) use more memory. The latter demonstrates a classic space vs. time tradeoff, but it is not necessarily the case that you can only achieve better speed at the expense of space. Also, note that there is often a theoretical limit to how fast your program can run (in terms of time complexity). For instance, a question that requires you to find the smallest/largest element in an unsorted array cannot run faster than O(N).

Data structures are your weapons. Choosing the right weapon for the right battle is the key to victory. Be very familiar about the strengths of each data structure and the time complexities for its various operations.

Data structures can be augmented to achieve efficient time complexities across different operations. For example, a hash map can be used together with a doubly-linked list to achieve O(1) time complexity for both the `get` and `put` operation in an LRU cache.

Hash table is probably the most commonly used data structure for algorithm questions. If you are stuck on a question, your last resort can be to enumerate through the common possible data structures (thankfully there aren't that many of them) and consider whether each of them can be applied to the problem. This has worked for me sometimes.

If you are cutting corners in your code, state that out loud to your interviewer and say what you would do in a non-interview setting with no time constraints (e.g. I would write a regex to parse this string rather than using

non-interview setting with no time constraints (e.g., I would write a regex to parse this string rather than using `split()` which may not cover all cases).

Basics

Array

Introduction

Arrays hold values of the same type at contiguous memory locations. In an array, we're usually concerned about two things — the position/index of an element and the element itself. Different programming languages implement arrays under the hood differently and can affect the time complexity of operations you make to the array. In some languages like Python, JavaScript, Ruby, PHP, the array (or list in Python) size is dynamic and you do not need to have a size defined beforehand when creating the array. As a result, people usually have an easier time using these languages for interviews.

Arrays are among the most common data structures encountered during interviews. Questions which ask about other topics would likely involve arrays/sequences as well. Mastery of array is essential for interviews!

Advantages

- Store multiple elements of the same type with one single variable name
- Accessing elements is fast as long as you have the index, as opposed to linked lists where you have to traverse from the head.

Disadvantages

- Addition and removal of elements into/from the middle of an array is slow because the remaining elements need to be shifted to accommodate the new/missing element. An exception to this is if the position to be inserted/removed is at the end of the array.
- For certain languages where the array size is fixed, it cannot alter its size after initialization. If an insertion causes the total number of elements to exceed the size, a new array has to be allocated and the existing elements have to be copied over. The act of creating a new array and transferring elements over takes $O(n)$ time.

Learning resources

- Readings
  - Array in Data Structure: What is, Arrays Operations, Guru99
- Videos
  - Arrays, University of California San Diego

Common terms

Common terms you see when doing problems involving arrays:

- **Subarray:** A range of contiguous values within an array.
  - Example: given an array `[2, 3, 6, 1, 5, 4]`, `[3, 6, 1]` is a subarray while `[3, 1, 5]` is not a subarray.
- **Subsequence:** A sequence that can be derived from the given sequence by deleting some or no elements without changing the order of the remaining elements.
  - Example: given an array `[2, 3, 6, 1, 5, 4]`, `[3, 1, 5]` is a subsequence but `[3, 5, 1]` is not a subsequence.

Time complexity

Operation	Big-O	Note
Access	$O(1)$	
Search	$O(n)$	
Search (sorted array)	$O(\log n)$	
Insert	$O(n)$	Insertion would require shifting all the subsequent elements to the right by one and that takes $O(n)$
Insert (at the end)	$O(1)$	Special case of insertion where no other element needs to be shifted
Remove	$O(n)$	Removal would require shifting all the subsequent elements to the left by one and that takes $O(n)$
Remove (at the end)	$O(1)$	Special case of removal where no other element needs to be shifted

Things to look out for during interviews

- Clarify if there are duplicate values in the array. Would the presence of duplicate values affect the answer? Does it make the question simpler or harder?
- When using an index to iterate through array elements, be careful not to go out of bounds.
- Be mindful about slicing or concatenating arrays in your code. Typically, slicing and concatenating arrays would take $O(n)$ time. Use start and end indices to demarcate a subarray/range where possible.

Corner cases

- Empty sequence
- Sequence with 1 or 2 elements
- Sequence with repeated elements
- Duplicated values in the sequence

Techniques

Note that because both arrays and strings are sequences (a string is an array of characters), most of the techniques here will apply to string problems.

Sliding window

Master the sliding window technique that applies to many subarray/substring problems. In a sliding window, the two pointers usually move in the same direction will never overtake each other. This ensures that each value is only visited at most twice and the time complexity is still $O(n)$. Examples: Longest Substring Without Repeating Characters, Minimum Size Subarray Sum, Minimum Window Substring

Two pointers

Two pointers is a more general version of sliding window where the pointers can cross each other and can be on different arrays. Examples: Sort Colors, Palindromic Substrings

When you are given two arrays to process, it is common to have one index per array (pointer) to traverse/compare the both of them, incrementing one of the pointers when relevant. For example, we use this approach to merge two sorted arrays. Examples: Merge Sorted Array

Traversing from the right

Sometimes you can traverse the array starting from the right instead of the conventional approach of from the left. Examples: Daily Temperatures, Number of Visible People in a Queue

Sorting the array

Is the array sorted or partially sorted? If it is, some form of binary search should be possible. This also usually means that the interviewer is looking for a solution that is faster than $O(n)$.

Can you sort the array? Sometimes sorting the array first may significantly simplify the problem. Obviously this would not work if the order of array elements need to be preserved. Examples: Merge Intervals, Non-overlapping Intervals

Precomputation

For questions where summation or multiplication of a subarray is involved, pre-computation using hashing or a prefix/suffix sum/product might be useful. Examples: Product of Array Except Self, Minimum Size Subarray Sum, LeetCode questions tagged "prefix-sum"

Index as a hash key

If you are given a sequence and the interviewer asks for $O(1)$ space, it might be possible to use the array itself as a hash table. For example, if the array only has values from `1` to `N`, where `N` is the length of the array, negate the value at that index (minus one) to indicate presence of that number. Examples: First Missing Positive, Daily Temperatures

Traversing the array more than once

This might be obvious, but traversing the array twice/thrice (as long as fewer than n times) is still $O(n)$. Sometimes traversing the array more than once can help you solve the problem while keeping the time complexity to $O(n)$.

Essential questions

These are essential questions to practice if you're studying for this topic.

- Two Sum
- Best Time to Buy and Sell Stock
- Product of Array Except Self
- Maximum Subarray

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Contains Duplicate
- Maximum Product Subarray
- Search in Rotated Sorted Array
- 3Sum
- Container With Most Water
- Sliding Window Maximum

String

Introduction

A string is a sequence of characters. Many tips that apply to arrays also apply to strings. You're recommended to read the page on Arrays before reading this page.

Common data structures for looking up strings:

- Trie/Prefix Tree
- Suffix Tree

Common string algorithms:

- Rabin Karp for efficient searching of substring using a rolling hash
- KMP for efficient searching of substring

Time complexity

A strings is an array of characters, so the time complexities of basic string operations will closely resemble that of array operations.

Operation	Big-O
Access	$O(1)$
Search	$O(n)$
Insert	$O(n)$
Remove	$O(n)$

Operations involving another string

Here we assume the other string is of length `m`.

Operation	Big-O	Note
Find substring	$O(nm)$	This is the most naive case. There are more efficient algorithms for string searching such as the KMP algorithm
Concatenating strings	$O(n+m)$	
Slice	$O(m)$	
Split (by token)	$O(n+m)$	
Strip (remove leading and trailing whitespaces)	$O(n)$	

Things to look out for during interviews

Ask about input character set and case sensitivity. Usually the characters are limited to lowercase Latin characters, for example `a` to `z`.

Corner cases

- Empty string
- String with 1 or 2 characters
- String with repeated characters
- Strings with only distinct characters

Techniques

Many string questions fall into one of these buckets.

Counting characters

Often you will need to count the frequency of characters in a string. The most common way of doing that is by using a hash table/map in your language of choice. If your language has a built-in Counter class like Python, ask if you can use that instead.

If you need to keep a counter of characters, a common mistake is to say that the space complexity required for the counter is $O(n)$. The space required for a counter of a string of latin characters is $O(1)$ not $O(n)$. This is because the upper bound is the range of characters, which is usually a fixed constant of `26`. The input set is just lowercase Latin characters.

String of unique characters

A neat trick to count the characters in a string of unique characters is to use a 26-bit bitmask to indicate which lower case latin characters are inside the string.

```
mask = 0
for c in word:
  mask |= (1 << (ord(c) - ord('a')))
```

To determine if two strings have common characters, perform `&` on the two bitmasks. If the result is non-zero, ie. `mask_a & mask_b > 0`, then the two strings have common characters.

Anagram

An anagram is word switch or word play. It is the result of rearranging the letters of a word or phrase to produce a new word or phrase, while using all the original letters only once. In interviews, usually we are only bothered with words without spaces in them.

To determine if two strings are anagrams, there are a few approaches:

- Sorting both strings should produce the same resulting string. This takes $O(n \log n)$ time and $O(\log n)$ space.
- If we map each character to a prime number and we multiply each mapped number together, anagrams should have the same multiple (prime factor decomposition). This takes $O(n)$ time and $O(1)$ space. Examples: Group Anagram
- Frequency counting of characters will help to determine if two strings are anagrams. This also takes $O(n)$ time and $O(1)$ space.

Palindrome

A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as `madam` or `racecar`.

Here are ways to determine if a string is a palindrome:

- Reverse the string and it should be equal to itself.
- Have two pointers at the start and end of the string. Move the pointers inward till they meet. At every point in time, the characters at both pointers should match.

The order of characters within the string matters, so hash tables are usually not helpful.

When a question is about counting the number of palindromes, a common trick is to have two pointers that move outward, away from the middle. Note that palindromes can be even or odd length. For each middle pivot position, you need to check it twice — once that includes the character and once without the character. This technique is used in Longest Palindromic Substring.

- For substrings, you can terminate early once there is no match
- For subsequences, use dynamic programming as there are overlapping subproblems. Check out this question

Essential questions

These are essential questions to practice if you're studying for this topic.

- Valid Anagram
- Valid Palindrome
- Longest Substring Without Repeating Characters

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Longest Repeating Character Replacement
- Find All Anagrams in a String
- Minimum Window Substring
- Group Anagrams
- Longest Palindromic Substring
- Encode and Decode Strings (LeetCode Premium)

Hash table

Introduction

A hash table (commonly referred to as hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function on an element to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

Hashing is the most common example of a space-time tradeoff. Instead of linearly searching an array every time to determine if an element is present, which takes $O(n)$ time, we can traverse the array once and hash all the elements into a hash table. Determining if the element is present is a simple matter of hashing the element and seeing if it exists in the hash table, which is $O(1)$ on average.

In the case of hash collisions, there are a number of collision resolution techniques that can be used. You will unlikely be asked about details of collision resolution techniques in interviews:

- **Separate chaining:** A linked list is used for each value, so that it stores all the collided items.
- **Open addressing:** All entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found.

Learning resources

- Readings
  - Taking Hash Tables Off The Shelf, basecs
  - Hashing Out Hash Functions, basecs
- Videos
  - Core: Hash Tables, University of California San Diego
  - A Brief Guide to Hash Tables (slides), Samuel Albanie, University of Cambridge

Implementations

Language	API
C++	`std::unordered_map`

Java	`java.util.Map`. Use `java.util.HashMap`
Python	`dict`
JavaScript	`Object` or `Map`

Time complexity

Operation	Big-O	Note
Access	N/A	Accessing not possible as the hash code is not known
Search	$O(1)$*	
Insert	$O(1)$*	
Remove	$O(1)$*	

** This is the average case, but in interviews we only care about the average case for hash tables.*

Sample questions

- Describe an implementation of a least-used cache, and big-O notation of it.
- A question involving an API's integration with hash map where the buckets of hash map are made up of linked lists.

Essential questions

These are essential questions to practice if you're studying for this topic.

- Two Sum
- Ransom Note

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Group Anagrams
- Insert Delete GetRandom O(1)
- First Missing Positive
- LRU Cache
- All O one Data Structure

Recursion

Introduction

Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem.

All recursive functions contains two parts:

1. A base case (or cases) defined, which defines when the recursion is stopped — otherwise it will go on forever!
2. Breaking down the problem into smaller subproblems and invoking the recursive call

One of the most common example of recursion is the Fibonacci sequence.

- Base cases: `fib(0) = 0` and `fib(1) = 1`
- Recurrence relation: `fib(i) = fib(i-1) + fib(i-2)`

```
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
```

Many algorithms relevant in coding interviews make heavy use of recursion — binary search, merge sort, tree traversal, depth-first search, etc. In this article, we focus on questions which use recursion but aren't part of other well known algorithms.

Learning resources

- Readings
  - Recursion, University of Utah
- Videos
  - Tail Recursion, University of Washington

Things to look out for during interviews

- Always remember to always define a base case so that your recursion will end.
- Recursion is useful for permutation, because it generates all combinations and tree-based questions. You should know how to generate all permutations of a sequence as well as how to handle duplicates.
- Recursion implicitly uses a stack. Hence all recursive approaches can be rewritten iteratively using a stack. Beware of cases where the recursion level goes too deep and causes a stack overflow (the default limit in Python is 1000). You may get bonus points for pointing this out to the interviewer. Recursion will never be $O(1)$ space complexity because a stack is involved, unless there is tail-call optimization (TCO). Find out if your chosen language supports TCO.
- Number of base cases — In the fibonacci example above, note that one of our recursive calls invoke `fib(n - 2)`. This indicates that you should have 2 base cases defined so that your code covers all possible invocations of the function within the input range. If your recursive function only invokes `fn(n - 1)`, then only one base case is needed

Corner cases

- `n = 0`
- `n = 1`
- Make sure you have enough base cases to cover all possible invocations of the recursive function

Techniques

Memoization

In some cases, you may be computing the result for previously computed inputs. Let's look at the Fibonacci example again. `fib(5)` calls `fib(4)` and `fib(3)`, and `fib(4)` calls `fib(3)` and `fib(2)`. `fib(3)` is being called twice! If the value for `fib(3)` is memoized and used again, that greatly improves the efficiency of the algorithm and the time complexity becomes O(n).

Essential questions

These are essential questions to practice if you're studying for this topic.

- Generate Parentheses
- Combinations
- Subsets

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Letter Combinations of a Phone Number
- Subsets II
- Permutations
- Sudoku Solver
- Strobogrammatic Number II (LeetCode Premium)

Sorting and searching

Introduction

Sorting is the act of rearranging elements in a sequence in order, either in numerical or lexicographical order, and either ascending or descending.

A number of basic algorithms run in $O(n^2)$ and should not be used in interviews. In algorithm interviews, you're unlikely to need to implement any of the sorting algorithms from scratch. Instead you would need to sort the input using your language's default sorting function so that you can use binary searches on them.

On a sorted array of elements, by leveraging on its sorted property, searching can be done on them in faster than $O(n)$ time by using a binary search. Binary search compares the target value with the middle element of the array, which informs the algorithm whether the target value lies in the left half or the right half, and this comparison proceeds on the remaining half until the target is found or the remaining half is empty.

Learning resources

While you're unlikely to be asked to implement a sorting algorithm from scratch during an interview, it is good to know the various time complexities of the different sorting algorithms.

- Readings
  - Sorting Out The Basics Behind Sorting Algorithms, basecs
  - Binary Search, Khan Academy
- Additional (only if you have time)
  - Exponentially Easy Selection Sort, basecs
  - Bubbling Up With Bubble Sorts, basecs
  - Inching Towards Insertion Sort, basecs
  - Making Sense of Merge Sort (Part 1), basecs
  - Making Sense of Merge Sort (Part 2), basecs
  - Pivoting To Understand Quicksort (Part 1), basecs
  - Pivoting To Understand Quicksort (Part 2), basecs
  - Counting Linearly With Counting Sort, basecs
  - Getting To The Root Of Sorting With Radix Sort, basecs
- Videos
  - Heapsort (slides), Samuel Albanie, University of Cambridge
  - Quicksort (slides), Samuel Albanie, University of Cambridge
  - Lower bounds for comparison sorts (slides), Samuel Albanie, University of Cambridge
  - Counting sort (slides), Samuel Albanie, University of Cambridge
  - Radix sort (slides), Samuel Albanie, University of Cambridge
  - Bucket sort (slides), Samuel Albanie, University of Cambridge

Time complexity

Algorithm	Time	Space
Bubble sort	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(1)$
Quicksort	$O(n \log n)$	$O(\log n)$
Mergesort	$O(n \log n)$	$O(n)$
Heapsort	$O(n \log n)$	$O(1)$
Counting sort	$O(n + k)$	$O(k)$
Radix sort	$O(nk)$	$O(n + k)$

Algorithm	Big-O
Binary search	$O(\log n)$

Things to look out for during interviews

Make sure you know the time and space complexity of the language's default sorting algorithm! The time complexity is almost definitely $O(n \log n)$). Bonus points if you can name the sort. In Python, it's Timsort. In Java, an implementation of Timsort is used for sorting objects, and Dual-Pivot Quicksort is used for sorting primitives.

Corner cases

- Empty sequence
- Sequence with one element
- Sequence with two elements
- Sequence containing duplicate elements.

Techniques

Sorted inputs

When a given sequence is in a sorted order (be it ascending or descending), using binary search should be one of the first things that come to your mind.

Sorting an input that has limited range

Counting sort is a non-comparison-based sort you can use on numbers where you know the range of values beforehand. Examples: H-Index

Essential questions

These are essential questions to practice if you're studying for this topic.

- Binary Search
- Search in Rotated Sorted Array

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Kth Smallest Element in a Sorted Matrix
- Search a 2D Matrix
- Kth Largest Element in an Array
- Find Minimum in Rotated Sorted Array
- Median of Two Sorted Arrays

Data structures

Matrix

Introduction

A matrix is a 2-dimensional array. Questions involving matrices are usually related to dynamic programming or graph traversal.

Matrices can be used to represent graphs where each node is a cell on the matrix which has 4 neighbors (except those cells on the edge and corners). This page will focus on questions which don't use matrix as graphs. Questions which are meant to use the matrix as a graph can be found on the graph section.

Corner cases

- Empty matrix. Check that none of the arrays are 0 length
- 1 x 1 matrix
- Matrix with only one row or column

Techniques

Creating an empty N x M matrix

For questions involving traversal or dynamic programming, you almost always want to make a copy of the matrix with the same size/dimensions that is initialized to empty values to store the visited state or dynamic programming table. Be familiar with such a routine in your language of choice:

This can be done easily in Python in one line.

```
# Assumes that the matrix is non-empty
zero_matrix = [[0 for _ in range(len(matrix[0]))] for _ in range(len(matrix))]
```

Copying a matrix in Python is:

```
copied_matrix = [row[:] for row in matrix]
```

Transposing a matrix

The transpose of a matrix is found by interchanging its rows into columns or columns into rows.

Many grid-based games can be modeled as a matrix, such as Tic-Tac-Toe, Sudoku, Crossword, Connect 4, Battleship, etc. It is not uncommon to be asked to verify the winning condition of the game. For games like Tic-Tac-Toe, Connect 4 and Crosswords, where verification has to be done vertically and horizontally, one trick is to write code to verify the matrix for the horizontal cells, transpose the matrix, and reuse the logic for horizontal verification to verify originally vertical cells (which are now horizontal).

Transposing a matrix in Python is simply:

```
transposed_matrix = zip(*matrix)
```

Essential questions

These are essential questions to practice if you're studying for this topic.

- Set Matrix Zeroes
- Spiral Matrix

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Rotate Image
- Valid Sudoku

Linked list

Introduction

Like arrays, a linked list is used to represent sequential data. It is a linear collection of data elements whose order is not given by their physical placement in memory, as opposed to arrays, where data is stored in sequential blocks of memory. Instead, each element contains an address of the next element. It is a data structure consisting of a collection of nodes which together represent a sequence.

In its most basic form, each node contains: data, and a reference (in other words, a link) to the next node in the sequence.

Advantages

Insertion and deletion of a node in the list (given its location) is O(1) whereas in arrays the following elements will have to be shifted.

Disadvantages

Access time is linear because directly accessing elements by its position in the list is not possible (in arrays you can do `arr[4]` for example). You have to traverse from the start.

Learning resources

- Readings
  - What's a Linked List, Anyway? [Part 1], basecs
  - What's a Linked List, Anyway? [Part 2], basecs
- Videos
  - Singly-linked lists, University of California San Diego
  - Doubly linked lists, University of California San Diego

Types of linked lists

Singly linked list

A linked list where each node points to the next node and the last node points to `null`.

Doubly linked list

A linked list where each node has two pointers, `next` which points to the next node and `prev` which points to the previous node. The `prev` pointer of the first node and the `next` pointer of the last node point to `null`.

Circular linked list

A singly linked list where the last node points back to the first node. There is a circular doubly linked list variant where the `prev` pointer of the first node points to the last node and the `next` pointer of the last node points to the first node.

Implementations

Out of the common languages, only Java provides a linked list implementation. Thankfully it's easy to write your own linked list regardless of language.

Language	API
C++	N/A
Java	`java.util.LinkedList`
Python	N/A
JavaScript	N/A

Time complexity

Operation	Big-O	Note
Access	$O(n)$	
Search	$O(n)$	
Insert	$O(1)$	Assumes you have traversed to the insertion position
Remove	$O(1)$	Assumes you have traversed to the node to be removed

Common routines

Be familiar with the following routines because many linked list questions make use of one or more of these routines in the solution:

- Counting the number of nodes in the linked list
- Reversing a linked list in-place
- Finding the middle node of the linked list using two pointers (fast/slow)
- Merging two linked lists together

Corner cases

- Empty linked list (head is `null`)
- Single node
- Two nodes
- Linked list has cycles. **Tip:** Clarify beforehand with the interviewer whether there can be a cycle in the list. Usually the answer is no and you don't have to handle it in the code

Techniques

Sentinel/dummy nodes

Adding a sentinel/dummy node at the head and/or tail might help to handle many edge cases where operations have to be performed at the head or the tail. The presence of dummy nodes essentially ensures that operations will never have to be done on the head or the tail, thereby removing a lot of headache in writing conditional checks to dealing with null pointers. Be sure to remember to remove them at the end of the operation.

Two pointers

Two pointer approaches are also common for linked lists. This approach is used for many classic linked list problems.

- Getting the `k`th from last node — Have two pointers, where one is k nodes ahead of the other. When the node ahead reaches the end, the other node is k nodes behind
- Detecting cycles — Have two pointers, where one pointer increments twice as much as the other, if the two pointers meet, means that there is a cycle
- Getting the middle node — Have two pointers, where one pointer increments twice as much as the other. When the faster node reaches the end of the list, the slower node will be at the middle

Using space

Many linked list problems can be easily solved by creating a new linked list and adding nodes to the new linked list with the final result. However, this takes up extra space and makes the question much less challenging. The interviewer will usually request that you modify the linked list in-place and the solve the problem without additional storage. You can borrow ideas from the Reverse a Linked List problem.

Elegant modification operations

As mentioned earlier, a linked list's non-sequential nature of memory allows for efficient modification of its contents. Unlike arrays where you can only modify the value at a position, for linked lists you can also modify the `next` pointer in addition to the `value`.

Here are some common operations and how they can be achieved easily:

- Truncate a list — Set the `next` pointer to `null` at the last element
- Swapping values of nodes — Just like arrays, just swap the value of the two nodes, there's no need to swap the `next` pointer
- Combining two lists — attach the head of the second list to the tail of the first list

Essential questions

These are essential questions to practice if you're studying for this topic.

- Reverse a Linked List
- Detect Cycle in a Linked List

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Merge Two Sorted Lists
- Merge K Sorted Lists
- Remove Nth Node From End Of List
- Reorder List

Queue

Introduction

A queue is a linear collection of elements that are maintained in a sequence and can be modified by the addition of elements at one end of the sequence (**enqueue** operation) and the removal of elements from the other end (**dequeue** operation). Usually, the end of the sequence at which elements are added is called the back, tail, or rear of the queue, and the end at which elements are removed is called the head or front of the queue. As an abstract

data type, queues can be implemented using arrays or singly linked lists.

This behavior is commonly called FIFO (first in, first out). The name "queue" for this type of structure comes from the analogy to people lining up in real life to wait for goods or services.

Breadth-first search is commonly implemented using queues.

Learning resources

- Readings
  - To Queue Or Not To Queue, basecs
- Videos
  - Queues, University of California San Diego

Implementations

Language	API
C++	`std::queue`
Java	`java.util.Queue`.Use `java.util.ArrayDeque`
Python	`queue`
JavaScript	N/A

Time complexity

Operation	Big-O
Enqueue/Offer	$O(1)$
Dequeue/Poll	$O(1)$
Front	$O(1)$
Back	$O(1)$
isEmpty	$O(1)$

Things to look out for during interviews

Most languages don't have a built in Queue class which to be used, and candidates often use arrays (JavaScript) or lists (Python) as a queue. However, note that the enqueue operation in such a scenario will be $O(n)$ because it requires shifting of all other elements by one. In such cases, you can flag this to the interviewer and say that you assume that there's a queue data structure to use which has an efficient enqueue operation.

Corner cases

- Empty queue
- Queue with one item
- Queue with two items

Essential questions

These are essential questions to practice if you're studying for this topic.

- Implement Stack using Queues

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Implement Queue using Stacks
- Design Circular Queue
- Design Hit Counter (LeetCode Premium)

Stack

Introduction

A stack is an abstract data type that supports the operations **push** (insert a new element on the top of the stack) and **pop** (remove and return the most recently added element, the element at the top of the stack). As an abstract data type, stacks can be implemented using arrays or singly linked lists.

This behavior is commonly called LIFO (last in, first out). The name "stack" for this type of structure comes from the analogy to a set of physical items stacked on top of each other.

Stacks are an important way of supporting nested or recursive function calls and is used to implement depth-first search. Depth-first search can be implemented using recursion or a manual stack.

Learning resources

- Readings
  - Stacks and Overflows, basecs
- Videos
  - Stacks, University of California San Diego

Implementations

Language	API
C++	`std::stack`
Java	`java.util.Stack`
Python	Simulated using List
JavaScript	Simulated using Array

Time complexity

Operation	Big-O
Top/Peek	$O(1)$
Push	$O(1)$
Pop	$O(1)$
isEmpty	$O(1)$
Search	$O(n)$

Search	$O(n)$

Corner cases

- Empty stack. Popping from an empty stack
- Stack with one item
- Stack with two items

Essential questions

These are essential questions to practice if you're studying for this topic.

- Valid Parentheses
- Implement Queue using Stacks

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Implement Stack using Queues
- Min Stack
- Asteroid Collision
- Evaluate Reverse Polish Notation
- Basic Calculator
- Basic Calculator II
- Daily Temperatures
- Trapping Rain Water
- Largest Rectangle in Histogram

Interval

Introduction

Interval questions are a subset of array questions where you are given an array of two-element arrays (an interval) and the two values represent a start and an end value. Interval questions are considered part of the array family but they involve some common techniques hence they are extracted out to this special section of their own.

An example interval array: `[[1, 2], [4, 7]]`.

Interval questions can be tricky to those who have not tried them before because of the sheer number of cases to consider when they overlap.

Things to look out for during interviews

- Clarify with the interviewer whether `[1, 2]` and `[2, 3]` are considered overlapping intervals as it affects how you will write your equality checks.
- Clarify whether an interval of `[a, b]` will strictly follow `a < b` (`a` is smaller than `b`)

Corner cases

- No intervals
- Single interval
- Two intervals
- Non-overlapping intervals
- An interval totally consumed within another interval
- Duplicate intervals (exactly the same start and end)
- Intervals which start right where another interval ends — `[[1, 2], [2, 3]]`

Techniques

Sort the array of intervals by its starting point

A common routine for interval questions is to sort the array of intervals by each interval's starting value. This step is crucial to solving the Merge Intervals question.

Checking if two intervals overlap

Be familiar with writing code to check if two intervals overlap.

```
def is_overlap(a, b):
  return a[0] < b[1] and b[0] < a[1]
```

Merging two intervals

```
def merge_overlapping_intervals(a, b):
  return [min(a[0], b[0]), max(a[1], b[1])]
```

Essential questions

These are essential questions to practice if you're studying for this topic.

- Merge Intervals
- Insert Interval

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Non-overlapping Intervals
- Meeting Rooms (LeetCode Premium)
- Meeting Rooms II (LeetCode Premium)

Advanced data structures

Tree

Introduction

A tree is a widely used abstract data type that represents a hierarchical structure with a set of connected nodes. Each node in the tree can be connected to many children, but must be connected to exactly one parent, except for the root node, which has no parent.

A tree is an undirected and connected acyclic graph. There are no cycles or loops. Each node can be like the root node of its own subtree, making recursion a useful technique for tree traversal.

For the purpose of interviews, you will usually be asked on binary trees as opposed to ternary (3 children) or N-ary (N children) trees. In this page, we will cover binary trees and binary search trees, which is a special case of binary trees.

Trees are commonly used to represent hierarchical data, e.g. file systems, JSON, and HTML documents. Do check out the section on Trie, which is an advanced tree used for efficiently storing and searching strings.

Learning resources

- Videos
  - Trees, University of California San Diego
  - A Brief Guide to Binary Search Trees (slides), Samuel Albanie, University of Cambridge
  - A Brief Guide to Red-Black Trees (slides), Samuel Albanie, University of Cambridge
  - A Brief Guide to B-trees (slides), Samuel Albanie, University of Cambridge
- Readings
  - How To Not Be Stumped By Trees, basecs
  - Leaf It Up To Binary Trees, basecs
- Additional (only if you have time)
  - The Little AVL Tree That Could, basecs
  - Busying Oneself With B-Trees, basecs
  - Painting Nodes Black With Red-Black Trees, basecs

Common terms you need to know

- **Neighbor** - Parent or child of a node
- **Ancestor** - A node reachable by traversing its parent chain
- **Descendant** - A node in the node's subtree
- **Degree** - Number of children of a node
- **Degree** of a tree - Maximum degree of nodes in the tree
- **Distance** - Number of edges along the shortest path between two nodes
- **Level/Depth** - Number of edges along the unique path between a node and the root node
- **Width** - Number of nodes in a level

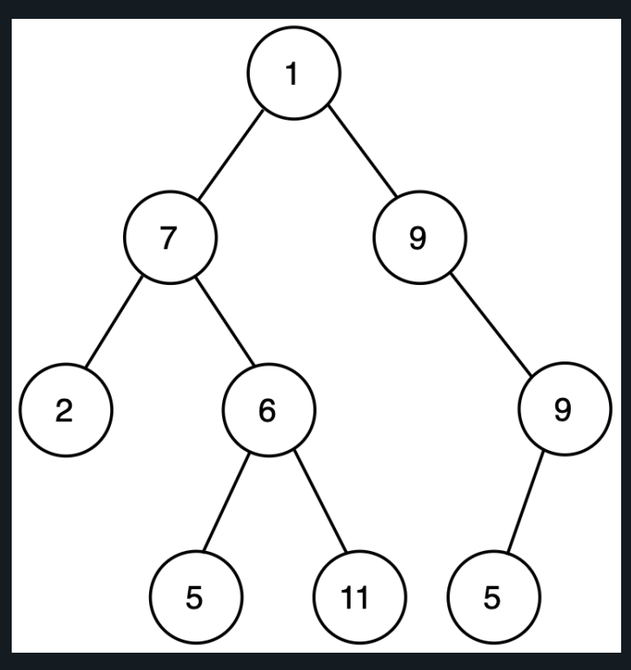
Binary tree

Binary means two, so nodes in a binary tree have a maximum of two children.

Binary tree terms

- Complete binary tree — A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- Balanced binary tree — A binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.

Traversals



Given such a tree, these are the results for the various traversals.

- **In-order traversal:** `Left -> Root -> Right`
  - Result: `2, 7, 5, 6, 11, 1, 9, 5, 9`
- **Pre-order traversal:** `Root -> Left -> Right`
  - Result: `1, 7, 2, 6, 5, 11, 9, 9, 5`
- **Post-order traversal:** `Left -> Right -> Root`
  - Result: `2, 5, 11, 6, 7, 5, 9, 9, 1`

Note that in-order traversal of a binary tree is insufficient to uniquely serialize a tree. Pre-order or post-order traversal is also required.

Binary search tree (BST)

In-order traversal of a BST will give you all elements in order.

Be very familiar with the properties of a BST and validating that a binary tree is a BST. This comes up more often than expected.

When a question involves a BST, the interviewer is usually looking for a solution which runs faster than O(n).

Time complexity

Operation	Big-O
Access	$O(\log n)$
Search	$O(\log n)$
Insert	$O(\log n)$
Remove	$O(\log n)$

Space complexity of traversing balanced trees is $O(h)$ where h is the height of the tree, while traversing very skewed trees (which is essentially a linked list) will be $O(n)$.

Things to look out for during interviews

You should be very familiar with writing pre-order, in-order, and post-order traversal recursively. As an extension, challenge yourself by writing them iteratively. Sometimes interviewers ask candidates for the iterative approach, especially if the candidate finishes writing the recursive approach too quickly.

Corner cases

- Empty tree
- Single node
- Two nodes
- Very skewed tree (like a linked list)

Common routines

Be familiar with the following routines because many tree questions make use of one or more of these routines in the solution:

- Insert value
- Delete value
- Count number of nodes in tree
- Whether a value is in the tree
- Calculate height of the tree
- Binary search tree
  - Determine if is binary search tree

- Get maximum value
- Get minimum value

Techniques

Use recursion

Recursion is the most common approach for traversing trees. When you notice that the subtree problem can be used to solve the entire problem, try using recursion.

When using recursion, always remember to check for the base case, usually where the node is `null`.

Sometimes it is possible that your recursive function needs to return two values.

Traversing by level

When you are asked to traverse a tree by level, use breadth-first search.

Summation of nodes

If the question involves summation of nodes along the way, be sure to check whether nodes can be negative.

Essential questions

These are essential questions to practice if you're studying for this topic.

- Binary Tree
  - Maximum Depth of Binary Tree
  - Invert/Flip Binary Tree
- Binary Search Tree
  - Lowest Common Ancestor of a Binary Search Tree

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Binary tree
  - Same Tree
  - Binary Tree Maximum Path Sum
  - Binary Tree Level Order Traversal
  - Lowest Common Ancestor of a Binary Tree
  - Binary Tree Right Side View
  - Subtree of Another Tree
  - Construct Binary Tree from Preorder and Inorder Traversal
  - Serialize and Deserialize Binary Tree
- Binary search tree
  - Validate Binary Search Tree
  - Kth Smallest Element in a BST

Graph

Introduction

A graph is a structure containing a set of objects (nodes or vertices) where there can be edges between these nodes/vertices. Edges can be directed or undirected and can optionally have values (a weighted graph). Trees are undirected graphs in which any two vertices are connected by exactly one edge and there can be no cycles in the graph.

Graphs are commonly used to model relationship between unordered entities, such as

- Friendship between people — Each node is a person and edges between nodes represent that these two people are friends.
- Distances between locations — Each node is a location and the edge between nodes represent that these locations are connected. The value of the edge represent the distance.

Be familiar with the various graph representations, graph search algorithms and their time and space complexities.

Learning resources

- Readings
  - From Theory To Practice: Representing Graphs, basecs
  - Deep Dive Through A Graph: DFS Traversal, basecs
  - Going Broad In A Graph: BFS Traversal, basecs
- Additional (only if you have time)
  - Finding The Shortest Path, With A Little Help From Dijkstra, basecs
  - Spinning Around In Cycles With Directed Acyclic Graphs, basecs

Graph representations

You can be given a list of edges and you have to build your own graph from the edges so that you can perform a traversal on them. The common graph representations are:

- Adjacency matrix
- Adjacency list
- Hash table of hash tables

Using a hash table of hash table would be the simplest approach during algorithm interviews. It will be rare that you have to use adjacency matrix or list for graph questions during interviews.

In algorithm interviews, graphs are commonly given in the input as 2D matrices where cells are the nodes and each cell can traverse to its adjacent cells (up/down/left/right). Hence it is important that you be familiar with traversing a 2D matrix. When traversing the matrix, always ensure that your current position is within the boundary of the matrix and has not been visited before.

Time complexity

`|V|` is the number of vertices while `|E|` is the number of edges.

Algorithm	Big-O
Depth-first search	O(|V| + |E|)
Breadth-first search	O(|V| + |E|)
Topological sort	O(|V| + |E|)

Things to look out for during interviews

- A tree-like diagram could very well be a graph that allows for cycles and a naive recursive solution would not work. In that case you will have to handle cycles and keep a set of visited nodes when traversing.
- Ensure you are correctly keeping track of visited nodes and not visiting each node more than once. Otherwise your code could end up in an infinite loop.

Corner cases

- Empty graph

- Graph with one or two nodes
- Disconnected graphs
- Graph with cycles

Graph search algorithms

- **Common:** Breadth-first Search, Depth-first Search
- **Uncommon:** Topological Sort, Dijkstra's algorithm
- **Almost never:** Bellman-Ford algorithm, Floyd-Warshall algorithm, Prim's algorithm, Kruskal's algorithm. Your interviewer likely don't know them either.

Depth-first search

Depth-first search is a graph traversal algorithm which explores as far as possible along each branch before backtracking. A stack is usually used to keep track of the nodes that are on the current search path. This can be done either by an implicit recursion stack, or an actual stack data structure.

A simple template for doing depth-first searches on a matrix goes like this:

```
def dfs(matrix):
  # Check for an empty matrix/graph.
  if not matrix:
    return []

  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

  def traverse(i, j):
    if (i, j) in visited:
      return

    visited.add((i, j))
    # Traverse neighbors.
    for direction in directions:
      next_i, next_j = i + direction[0], j + direction[1]
      if 0 <= next_i < rows and 0 <= next_j < cols:
        # Add in question-specific checks, where relevant.
        traverse(next_i, next_j)

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

Breadth-first search

Breadth-first search is a graph traversal algorithm which starts at a node and explores all nodes at the present depth, before moving on to the nodes at the next depth level. A queue is usually used to keep track of the nodes that were encountered but not yet explored.

A similar template for doing breadth-first searches on the matrix goes like this. It is important to use double-ended queues and not arrays/Python lists as dequeuing for double-ended queues is $O(1)$ but it's $O(n)$ for arrays.

```
from collections import deque

def bfs(matrix):
  # Check for an empty matrix/graph.
  if not matrix:
    return []

  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

  def traverse(i, j):
    queue = deque([(i, j)])
    while queue:
      curr_i, curr_j = queue.popleft()
      if (curr_i, curr_j) not in visited:
        visited.add((curr_i, curr_j))
        # Traverse neighbors.
        for direction in directions:
          next_i, next_j = curr_i + direction[0], curr_j + direction[1]
          if 0 <= next_i < rows and 0 <= next_j < cols:
            # Add in question-specific checks, where relevant.
            queue.append((next_i, next_j))

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

> **Info**
>
> While DFS is implemented using recursion in this sample, it could also be implemented iteratively similar to BFS. The key difference between the algorithms lies in the underlying data structure (BFS uses a queue while DFS uses a stack). The `deque` class in Python can function as both a stack and a queue.

For additional tips on BFS and DFS, you can refer to this LeetCode post.

Topological sorting

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge `uv` from vertex `u` to vertex `v`, `u` comes before `v` in the ordering. Precisely, a topological sort is a graph traversal in which each node `v` is visited only after all its dependencies are visited.

Topological sorting is most commonly used for job scheduling a sequence of jobs or tasks which has dependencies on other jobs/tasks. The jobs are represented by vertices, and there is an edge from `x` to `y` if job `x` must be completed before job `y` can be started.

Another example is taking courses in university where courses have pre-requisites.

Here's an example where the edges is an array of two-value tuples and the first value depends on the second value.

```
def graph_topo_sort(num_nodes, edges):
    from collections import deque
    nodes, order, queue = {}, [], deque()
    for node_id in range(num_nodes):
        nodes[node_id] = { 'in': 0, 'out': set() }
    for node_id, pre_id in edges:
        nodes[node_id]['in'] += 1
        nodes[pre_id]['out'].add(node_id)
    for node_id in nodes.keys():
        if nodes[node_id]['in'] == 0:
            queue.append(node_id)
    while len(queue):
        node_id = queue.pop()
        for outgoing_id in nodes[node_id]['out']:
            nodes[outgoing_id]['in'] -= 1
            if nodes[outgoing_id]['in'] == 0:
                queue.append(outgoing_id)
        order.append(node_id)
    return order if len(order) == num_nodes else None

print(graph_topo_sort(4, [[0, 1], [0, 2], [2, 1], [3, 0]]))
# [1, 2, 0, 3]
```

```
# [1, 2, 0, 3]
```

Essential questions

These are essential questions to practice if you're studying for this topic.

- Number of Islands
- Flood Fill
- 01 Matrix

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Breadth-first search
  - Rotting Oranges
  - Minimum Knight Moves (LeetCode Premium)
- Either search
  - Clone Graph
  - Pacific Atlantic Water Flow
  - Number of Connected Components in an Undirected Graph (LeetCode Premium)
  - Graph Valid Tree (LeetCode Premium)
- Topological sorting
  - Course Schedule
  - Alien Dictionary (LeetCode Premium)

Heap

Introduction

A heap is a specialized tree-based data structure which is a complete tree that satisfies the heap property.

- Max heap — In a max heap the value of a node must be greatest among the node values in its entire subtree. The same property must be recursively true for all nodes in the tree.
- Min heap — In a min heap the value of a node must be smallest among the node values in its entire subtree. The same property must be recursively true for all nodes in the tree.

In the context of algorithm interviews, heaps and priority queues can be treated as the same data structure. A heap is a useful data structure when it is necessary to repeatedly remove the object with the highest (or lowest) priority, or when insertions need to be interspersed with removals of the root node.

Learning resources

- Learning to Love Heaps, basecs
- Heapify All The Things With Heap Sort, basecs
- Heaps, James Aspnes, Yale University

Implementations

Language	API
C++	`std::priority_queue`
Java	`java.util.PriorityQueue`
Python	`heapq`
JavaScript	N/A

Time complexity

Operation	Big-O
Find max/min	$O(1)$
Insert	$O(\log n)$
Remove	$O(\log n)$
Heapify (create a heap out of given array of elements)	$O(n)$

Techniques

Mention of `k`

If you see a *top* or *lowest* `k` being mentioned in the question, it is usually a signal that a heap can be used to solve the problem, such as in Top K Frequent Elements.

If you require the top `k` elements use a Min Heap of size `k`. Iterate through each element, pushing it into the heap (for python `heapq`, invert the value before pushing to find the max). Whenever the heap size exceeds `k`, remove the minimum element, that will guarantee that you have the `k` largest elements.

Essential questions

These are essential questions to practice if you're studying for this topic.

- Merge K Sorted Lists
- K Closest Points to Origin

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Top K Frequent Elements
- Find Median from Data Stream

Trie

Introduction

Tries are special trees (prefix trees) that make searching and storing strings more efficient. Tries have many practical applications, such as conducting searches and providing autocomplete. It is helpful to know these common applications so that you can easily identify when a problem can be efficiently solved using a trie.

Be familiar with implementing from scratch, a `Trie` class and its `add`, `remove` and `search` methods.

Learning resources

- Readings
  - Trying to Understand Tries, basecs
  - Implement Trie (Prefix Tree), LeetCode
- Additional (only if you have time)
  - Compressing Radix Trees Without (Too Many) Tears, basecs