21 SEPTEMBER 2017 / #WEB DEVELOPMENT #TECHNOLOGY #SELF IMPROVEMENT

The 30-minute guide to rocking your next coding interview

by Yangshun Tay

Despite scoring decent grades in both my CS101 Algorithm class and my Data Structures class in university, I shudder at the thought of going through a coding interview that focuses on algorithms.

Hence I spent the last three months figuring out how to improve my coding interview skills and eventually received offers from the big tech companies. In this post, I'll be sharing the insights and tips I gained along the way. Experienced candidates can also expect system design questions, but that is out of the scope of this post.

Many of the algorithmic concepts tested in coding interviews are not what I usually use at work, where I am a front-end web engineer. Naturally, I have forgotten quite a bit about these algorithms and data structures, which I learned mostly during my freshmen and sophomore years of college.

It's stressful to have to produce (working) code in an interview, while someone scrutinizes every keystroke that you make. What's worse is that as an interviewee, you're encouraged to communicate your thought process out loud to the interviewer.

I used to think that being able to think, code, and communicate simultaneously was an impossible feat, until I realized that most people are just not good at coding interviews when they first start out. Interviewing is a skill that you can get better at by studying, preparing, and practicing for it.

My recent job search has led me on a journey to improve my coding interview skills. Front-end engineers like to rant about how the current hiring process is broken because technical interviews can include skills not related to front-end development. For example, writing a maze solving algorithm and merging two sorted lists of numbers. As a front-end engineer myself, I can empathize with them.

Front end is a specialized domain where engineers have to care about many issues related to browser compatibilities, the Document Object Model, JavaScript performance, CSS layouts, and so on. It is uncommon for front-end engineers to implement some of the complex algorithms tested in interviews.

> **At companies like Facebook and Google, the people are software engineers first, domain experts second.**

Unfortunately, the rules are set by the companies, not by the candidates. There is a high emphasis on general computer science concepts like algorithms, design patterns, data structures; core skills that a good software engineer should possess. If you want the job, you

coding interview skills!

This post is structured into the following two sections. Feel free to skip ahead to the section that interests you.

- The breakdown of coding interviews, and how to prepare for them.
- Helpful tips and hints for each algorithm topic (arrays, trees, dynamic programming, etc.), along with recommended LeetCode practice questions to review core concepts and to improve on those topics.

The content for this post can be found in my Tech Interview Handbook repo on GitHub. Updates will be made there. Pull requests for suggestions and corrections are welcome!

yangshun/tech-interview-handbook

*tech-interview-handbook - 💯 Algorithms and behavioral content for rocking your coding interview.*github.com

If you are interested in Front End content, check out the answers to

yangshun/front-end-interview-handbook
*front-end-interview-handbook - 🕸 Almost complete answers to "Front-end Job Interview Questions"g*ithub.com

Picking a programming language

Before anything else, you need to pick a programming language for your algorithmic coding interview. Most companies will allow you to code in the language of your choice. The only exception I know is Google. They allow their candidates to pick from only Java, C++, Python, Go or JavaScript. For the most part, I recommend using a language that you are extremely familiar with, rather than one that is new to you but that the company uses widely.

There are some languages that are more suitable than others for coding interviews. Then there are some that you absolutely want to avoid. From my experience as an interviewer, most candidates pick Python or Java. Other languages commonly selected include JavaScript, Ruby, and C++. I would absolutely avoid lower-level languages like C or Go, simply because they lack standard library functions and data structures.

Personally, Python is my de facto choice for coding algorithms during interviews. It is succinct and has a huge library of functions and data structures. One of the top reasons I recommend Python is that it uses consistent APIs that operate on different data structures, such as `len()`, `for ... in ...` and slicing notation on sequences (strings, lists, and 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.

declare types in your code, it means entering extra keystrokes. This will slow down the speed at which you code and type. This issue will be more apparent when you have to write on a whiteboard during on-site interviews.

The reasons for choosing or not choosing C++ are similar to Java. Ultimately, Python, Java, and C++ are decent choices. If you have been using Java for a while, and do not have time to become familiar with another language, I recommend sticking to Java instead of picking up Python from scratch. This helps you to avoid having to use one language for work and another one for interviews. Most of the time, the bottleneck is in the thinking and not the writing.

One exception to the convention of allowing the candidate to "pick any programming language they want" is when the interview is for a domain-specific position, such as front-end, iOS, or Android engineer roles. You need to be familiar with coding algorithms 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, ask the interviewer if 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 it. 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.

Review your CS101

review the CS fundamentals. I prefer to review it as I practice. I scan through my notes from college and revise the various algorithms as I work on the algorithm problems from LeetCode and Cracking the Coding Interview.

This interviews repository by Kevin Naughton Jr. served as a quick refresher for me.

The Medium publication basecs by Vaidehi Joshi is also a great and light-hearted resource to recap on the various data structures and algorithms.

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

yangshun/lago
*lago - Data Structures and Algorithms library for JavaScript.*github.com

Mastery through practice

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

Practice and solve algorithm questions in your chosen language. While Cracking the Coding Interview is a good resource, I prefer solving problems by typing code, letting it run, and getting instant feedback. There are various Online Judges, such as LeetCode, HackerRank, and CodeForces for you to practice questions online and to get used to the language. From my experience, LeetCode questions are most similar

questions are more similar to questions in competitive programming. If you practice enough LeetCode questions, there is a good chance that you will either see or complete one of your actual interview questions (or some variant of it).

Learn and understand the time and space complexities of the common operations in your chosen language. For Python, this page will come in handy. Also, learn about the underlying sorting algorithm being used in the language's `sort()` function and its time and space complexities (in Python it's Timsort, which is a hybrid). After completing a question on LeetCode, I usually add the time and space complexities of the written code as comments above the function body. I use the comments to remind myself to communicate the analysis of the algorithm after I have completed the implementation.

Read up on the recommended coding style for your language and stick to it. If you choose Python, refer to the PEP 8 Style Guide. If you choose Java, refer to Google's Java Style Guide.

Learn about and be familiar with the common pitfalls and caveats of the language. If you point them out during the interview and avoid falling into them, you will earn bonus points and impress the interviewer, regardless of whether the interviewer is familiar with the language or not.

Gain a broad exposure to questions from various topics. In the second half of the article, I mention algorithm topics and the useful questions for each topic to practice. Do around 100 to 200 LeetCode questions, and you should be good.

Practice, practice, and more practice!

Phases of a coding interview

Congratulations, you are ready to put your skills to practice! In a coding interview, you will be given a technical question by the interviewer. You will write the code in a real-time, collaborative editor (phone screen) or on a whiteboard (on-site), and have 30 to 45 minutes to solve the problem. This is where the real fun begins!

Your interviewer will be looking to see that you meet the requirements of the role. It is up to you to show them that you have the skills. Initially, it may feel weird to talk while you code, as most programmers do not make a habit of explaining out loud their thoughts while they are typing code.

However, it is hard for the interviewer to know what you are thinking by just looking at your code. If you communicate your approach to the interviewer even before you start to code, you can validate your approach with them. This way, the two of you can agree on an acceptable approach.

Preparing for a remote interview

For phone screens and remote interviews, have a paper and pen or pencil to jot down any notes or diagrams. If you are given a question about trees and graphs, it usually helps if you draw examples of the data structure.

Use earphones. Make sure you are in a quiet environment. You do not want to be holding a phone in one hand and typing with the other. Try to avoid using speakers. If the feedback is bad, communication is made harder. Having to repeat yourself will just result in the loss of valuable time.

Many candidates start coding as soon as they hear the question. That is usually a big mistake. First, take a moment and repeat the question back to the interviewer to make sure that you understand the question. If you misunderstand the question, then the interviewer can clarify.

Always seek clarification about the question upon hearing it, even if you think it is clear. You might discover that you have missed something. It also lets the interviewer know that you are attentive to details.

Consider asking the following questions.

- How big is the size of the input?
- How big is the range of values?
- What kind of values are there? Are there negative numbers? Floating points? Will there be empty inputs?
- Are there duplicates within the input?
- What are some extreme cases of the input?
- How is the input stored? If you are given a dictionary of words, is it a list of strings or a trie?

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

Start with a brute-force approach. Communicate it to the interviewer.

unlikely that the brute-force approach will be the one that you will be coding. At this point, the interviewer will usually pop the dreaded, “Can we do better?” question. This means they arelooking for a more optimal approach.

This is usually the hardest part of the interview. In general, look for repeated work and try to optimize them by potentially caching the calculated result somewhere. Reference it later, rather than computing it all over again. I provide some tips on tackling topic-specific questions in detail below.

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

Starting to code

Use a good style to write your code. Reading code written by others is usually not an enjoyable task. Reading horribly formatted code written by others is even worse. Your goal is to make your interviewer understand your code so that they can quickly evaluate if your code does what it is suppose to and if it solves a given problem. Use clear variable names and avoid names that are single letters, unless they are for iteration. However, if you are coding on a whiteboard, avoid using verbose variable names. This reduces the amount of writing you will have to do.

Always explain to the interviewer what you are writing or typing. This is not about reading, verbatim, to the interviewer the code you are producing. Talk about the section of the code you are currently implementing at a higher level. Explain why it is written as such, and what it is trying to achieve.

When you copy and paste in code, consider whether it is necessary. Sometimes it is, sometimes it is not. If you find yourself copying and pasting a large chunk of code spanning multiple lines, it is probably an indicator that you can restructure the code by extracting those lines into a function. If it is just a single line you copied, usually it is fine. However, remember to change the respective variables in your copied line of code where relevant. Copying and pasting errors are a common source of bugs, even in day-to-day coding!

After coding

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

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

Next, come up with small test cases and step through the code (not your algorithm) with those sample input. Interviewers like it when you read their minds. What they usually do after you have finished coding is get you to write tests. It is a huge plus if you write tests for your code even before they prompt you to do so. You should be emulating a debugger when stepping through your code. Jot down or tell them the values of certain variables as you walk the interviewer through the lines of code.

If there are large duplicated chunks of code in your solution, restructure the code to show the interviewer that you value quality

evaluation.

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

If your interviewer is happy with the solution, the interview usually ends here. It is also common that the interviewer asks you extension questions, such as how you would handle the problem if the whole input is too large to fit into memory, or if the input arrives as a stream. This is a common follow-up question at Google, where they care a lot about scale. The answer is usually a divide-and-conquer approach — perform distributed processing of the data and only read certain chunks of the input from disk into memory, write the output back to disk and combine them later.

Practice with mock interviews

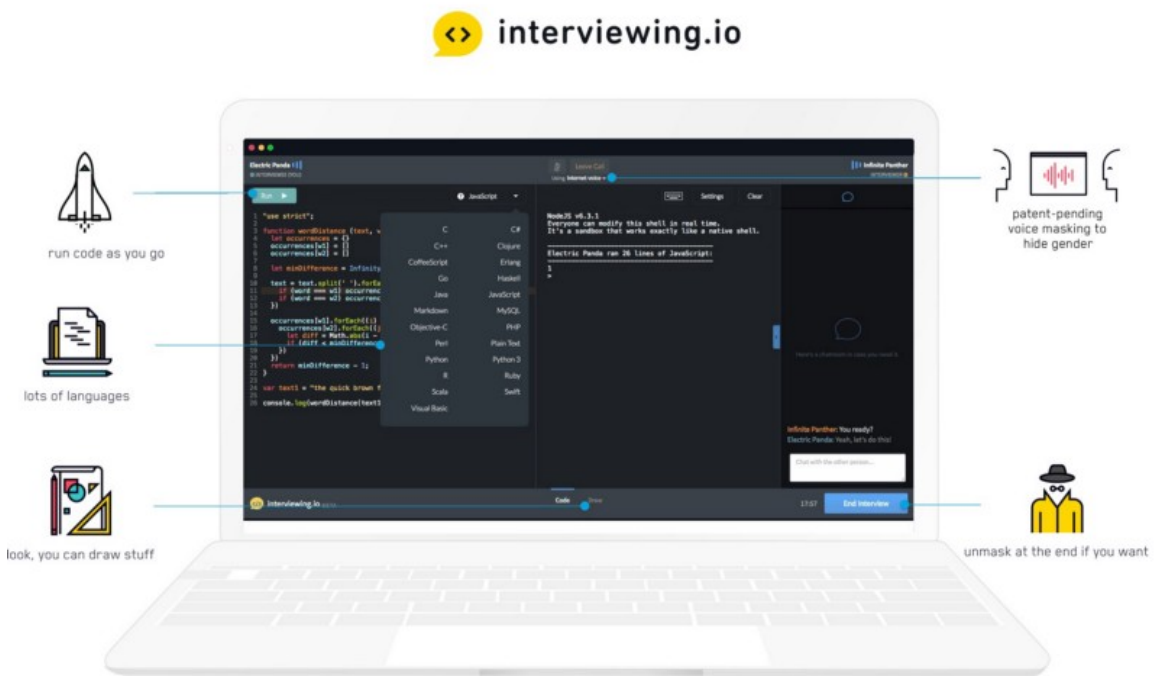
The steps mentioned above can be rehearsed over and over again until you have fully internalized them and they become second nature to you. A good way to practice is by partnering with a friend and taking turns to interview each other.

A great resource for preparing for coding interviews is interviewing.io. This platform provides free and anonymous practice 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 unbiased and low risk. At the end of the

to each other for the purpose of helping one another improve.

Doing well in mock interviews will unlock the jobs page for candidates, and allow them to book interviews (also anonymously) with top companies like Uber, Lyft, Quora, Asana, and more. For those who are new to coding interviews, a demo interview can be viewed on this site. Note that this site requires users to sign in.

I have used interviewing.io, both as an interviewer and an interviewee. The experience was great. Aline Lerner, the CEO and co-founder of interviewing.io, and her team are passionate about revolutionizing the process for coding interviews and helping candidates improve their interview skills. She has also published a number of coding interview-related articles on the interviewing.io blog. I recommend signing up as early as possible with interviewing.io, even though its in beta, to increase the likelihood of receiving an invite.



Practice interviewing anonymously with engineers from top companies!

Another platform that allows you to practice coding interviews is Pramp. Where interviewing.io matches potential job seekers with seasoned coding interviewers, Pramp takes a different approach. Pramp pairs you up with another peer who is also a job seeker. The two of you take turns assuming the roles of interviewer and interviewee. Pramp also prepares questions, and provides solutions and prompts to guide the interviewee.

Personally, I am not that fond of Pramp's approach. Because when I do interviews, I ask questions that are familiar to me. Also, many users do not have the experience of being an interviewer, and that can result in a horrible interview experience. In one instance, my matched peer took on the role of the interviewer but he did not have the correct understanding of the question and attempted to lead me down the wrong path to solve the question. However, this is more of a problem of the candidate than the platform though.

Go forth and conquer

After doing a fair amount of questions on LeetCode and having enough practice doing mock interviews, go forth and put your new-found interviewing skills to the test. Apply to your favorite companies or, better still, get referrals from your friends working for those companies. Referrals tend to get noticed earlier and have a faster response rate than applying without a referral. Good luck!

Practical tips for coding questions

This section dives deep into practical tips for specific topics of algorithms and data structures, which appear frequently in coding questions. Many algorithm questions involve techniques that can be applied to questions of a similar nature.

The more techniques you have in your arsenal, the greater your chances of passing the interview. For each topic, there is also a list of recommended questions, which is valuable for mastering the core concepts. Some of the questions are only available with a paid subscription to LeetCode, which in my opinion is absolutely worth the money if it lands you a job.

General tips

Always validate input first. Check for inputs that are invalid, empty, negative, or different. 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 and space complexities requirements or 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`, and `list`.

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

Is the algorithm supposed to run multiple times, perhaps on a web server? If yes, the input can likely be pre-processed to improve the efficiency in each API call.

Use a mix of functional and imperative programming paradigms:

- Use pure functions because they are easier to reason with and can help 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.
- Achieve a balance between accuracy and efficiency. Use the right amount of functional and imperative code where appropriate. 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.
- Avoid relying on mutating global variables. Global variables introduce state.
- Make sure that you do not accidentally mutate global variables, especially if you have to rely on them.

Generally, to improve the speed of a program, we can choose to either use an appropriate data structure or algorithm, or to use more memory. It's a classic space and time trade off.

Data structures are your weapons. Choosing the right weapon for the right battle is the key to victory. Know the strengths of each data structure and the time complexity for its various operations.

Data structures can be augmented to achieve efficient time complexity across different operations. For example, a HashMap 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.

algorithm questions. If you are stuck on a question, your last resort can be to enumerate through the possible data structures (thankfully there aren't that many) and consider whether each of them can be applied to the problem. This has worked for me at times.

If you are cutting corners in your code, state that out loud to your interviewer, and explain to them what you would do outside of an interview setting (no time constraints). For example, explain that you would write a regex to parse a string rather than using `split`, which does not cover all cases.

Sequence

Notes

Arrays and strings are considered sequences (a string is a sequence of characters). There are tips for dealing with both arrays and strings, which will be covered here.

Are there duplicate values in the sequence? Would they affect the answer?

Check for sequence out of bounds.

Be mindful about slicing or concatenating sequences in your code. Typically, slicing and concatenating sequences require O(n) time. Use start and end indices to demarcate a subarray or substring where possible.

Sometimes you traverse the sequence from the right side rather than from the left.

subarray problems.

When you are given two sequences to process, it is common to have one index per sequence to traverse. For example, we use the same approach to merge two sorted arrays.

Corner Cases

- Empty sequence
- Sequence with 1 or 2 elements
- Sequence with repeated elements

Array

Notes

Is the array sorted or partially sorted? If it is either, some form of binary search should be possible. This 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. Make sure that the order of array elements do not need to be preserved before attempting to sort it.

For questions where summation or multiplication of a subarray is involved, pre-computation using hashing or a prefix, suffix sum, or product might be useful.

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 has values only from 1 to N, where N is the length of the

presence of that number.

Practice Questions

- Two Sum
- Best Time to Buy and Sell Stock
- Contains Duplicate
- Product of Array Except Self
- Maximum Subarray
- Maximum Product Subarray
- Find Minimum in Rotated Sorted Array
- Search in Rotated Sorted Array
- 3Sum
- Container With Most Water

Binary

Study Links

- Bits, Bytes, Building With Binary

Notes

Questions involving binary representations and bitwise operations are asked sometimes. You must know how to convert a number from decimal form into binary form, and vice versa, in your chosen programming language.

Some helpful utility snippets:

- Set kth bit: `num |= (1 << k)`
- Turn off kth bit: `num &= ~(1 << k)`
- Toggle the kth bit: `num ^= (1 << k)`
- To check if a number is a power of 2: `num & num - 1 == 0`.

Corner Cases

- Check for overflow/underflow
- Negative numbers

Practice Questions

- Sum of Two Integers
- Number of 1 Bits
- Counting Bits
- Missing Number
- Reverse Bits

Dynamic programming

Study Links

- Demystifying Dynamic Programming

Notes

Dynamic Programming (DP) is usually used to solve optimization problems. Alaina Kafkes has written an awesome post on tackling DP problems. You should read it.

The only way to get better at DP is with practice. It takes lots of practice to recognize that a problem can be solved by DP.

To optimize space, sometimes you do not have to store the entire DP table in memory. The last two values or the last two rows of the matrix will suffice.

Practice Questions

- 0/1 Knapsack
- Climbing Stairs
- Coin Change
- Longest Increasing Subsequence
- Longest Common Subsequence
- Word Break Problem
- Combination Sum
- House Robber and House Robber II
- Decode Ways
- Unique Paths
- Jump Game

Geometry

Notes

When comparing Euclidean distance between two pairs of points, using $dx^2 + dy^2$ is sufficient. It is unnecessary to square root the value.

To find out if two circles overlap, check that the distance between the

Graph

Study Links

- From Theory To Practice: Representing Graphs
- Deep Dive Through A Graph: DFS Traversal
- Going Broad In A Graph: BFS Traversal

Notes

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

You can be given a list of edges and tasked to build your own graph from the edges to perform a traversal on. The common graph representations are

- Adjacency matrix
- Adjacency list
- HashMap of HashMaps

Some inputs look like they are trees, but they are actually graphs. Clarify this with your interviewer. In that case, you will have to handle cycles and keep a set of visited nodes when traversing.

Graph search algorithms

- Common: Breadth first search (BFS), Depth first search (DFS)
- Uncommon: Topological sort, Dijkstra's algorithm

Prim's algorithm, and Kruskal's algorithm

In coding interviews, graphs are commonly represented as 2-D matrices, where cells are the nodes and each cell can traverse to its adjacent cells (up, down, left, and right). Hence it is important to be familiar with traversing a 2-D matrix. When recursively traversing the matrix, always ensure that your next position is within the boundary of the matrix. More tips for doing DFS on a matrix can be found here. A simple template for doing DFS on a matrix appears something like this:

```
def traverse(matrix):
  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))
  def dfs(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: # Check boundar
        # Add any other checking here ^
        dfs(next_i, next_j)
  for i in range(rows):
    for j in range(cols):
      dfs(i, j)
```

Corner Cases

- Empty graph
- Graph with one or two nodes
- Disjoint graphs

Practice Questions

- Clone Graph
- Course Schedule
- Alien Dictionary
- Pacific Atlantic Water Flow
- Number of Islands
- Graph Valid Tree
- Number of Connected Components in an Undirected Graph
- Longest Consecutive Sequence

Interval

Notes

Interval questions are questions that give an array of two-element arrays (an interval). The two values represent a start and an end value. Interval questions are considered to be part of the array family, but they involve some common techniques. Hence, they have their own special section.

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

Interval questions can be tricky for those who do not have experience with them. This is because of the sheer number of cases to consider when interval arrays overlap.

Clarify with the interviewer whether `[1, 2]` and `[2, 3]` are

your equality checks.

A common routine for interval questions is to sort the array of intervals by the start value of each interval.

Be familiar with writing code to check if two intervals overlap and to merge two overlapping intervals:

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

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

Corner Cases

- Single interval
- Non-overlapping intervals
- An interval totally consumed within another interval
- Duplicate intervals

Practice Questions

- Insert Interval
- Merge Intervals
- Meeting Rooms and Meeting Rooms II
- Non-overlapping Intervals

Notes

Like arrays, linked lists are used to represent sequential data. The benefit of linked lists is that insertion and deletion of code from anywhere in the list is O(1), whereas in arrays, the elements have to be shifted.

Adding a 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 ensures that operations will never have be executed on the head or the tail. Dummy nodes remove the headache of writing conditional checks to deal with null pointers. Be sure to remove them at the end of the operation.

Sometimes linked lists problem can be solved without additional storage. Try to borrow ideas from the for reverse a linked list problem.

For deletion in linked lists, you can either modify the node values or change the node pointers. You might need to keep a reference to the previous element.

For partitioning linked lists, create two separate linked lists and join them back together.

Linked lists problems share similarities with array problems. Think about how you would solve an array problem and apply it to a linked list.

Two pointer approaches are also common for linked lists:

- Getting the kth from the last node: Have two pointers, where one is *k* nodes ahead of the other. When the node ahead

- Detecting cycles: Have two pointers, where one pointer increments twice as much as the other. If the two pointers meet, it means that there is a cycle.
- Getting the middle node: Have two pointers. 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.

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

- Count the number of nodes in the linked list
- Reverse a linked list in place
- Find the middle node of the linked list using fast or slow pointers
- Merge two lists together

Corner Cases

- Single node
- Two nodes
- Linked list has cycle. Clarify with the interviewer whether there can be a cycle in the list. Usually the answer is no.

Practice Questions

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

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

Math

Notes

If the code involves division or modulo, remember to check for division or modulo by 0 case.

When a question involves “a multiple of a number”, modulo might be useful.

Check for and handle overflow and underflow if you are using a typed language like Java and C++. At the very least, mention that overflow or underflow is possible and ask whether you need to handle it.

Consider negative numbers and floating point numbers. This may sound obvious, but when you are under pressure in an interview, many obvious points go unnoticed.

If the question asks to implement an operator such as power, squareroot, or division, and it is to be faster than O(n), binary search is usually the approach.

Some common formulas

- Sum of 1 to N = (n+1) * n/2
- Sum of GP = $2^0 + 2^1 + 2^2 + 2^3 + \ldots$ 2^n = 2^(n+1)-1
- Permutations of N = N! / (N-K)!

Corner Cases

- Division by 0
- Integer overflow and underflow

Practice Questions

- Pow(x, n)
- Sqrt(x)
- Integer to English Words

Matrix

Notes

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

For questions involving traversal or dynamic programming, make a copy of the matrix with the same dimensions that are initialized to empty values. Use these values to store the visited state or dynamic programming table. Be familiar with this routine:

```
rows, cols = len(matrix), len(matrix[0])
copy = [[0 for _ in range(cols)] for _ in range(rows)
```

- Many grid-based games can be modeled as a matrix. For example, Tic-Tac-Toe, Sudoku, Crossword, Connect 4, and

winning condition of the game. For games like Tic-Tac-Toe, Connect 4, and Crosswords, verification has to be done vertically and horizontally. One trick is to write code to verify the matrix for the horizontal cells. Then transpose the matrix, reusing the logic used for horizontal verification to verify originally vertical cells (which are now horizontal).

- Transposing a matrix in Python is simply:

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

Corner Cases

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

Practice Questions

- Set Matrix Zeroes
- Spiral Matrix
- Rotate Image
- Word Search

Recursion

Notes

combinations and tree-based questions. You should know how to generate all permutations of a sequence as well as how to handle duplicates.

Remember to always define a base case so that your recursion will end.

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.

Practice Questions

- Subsets and Subsets II
- Strobogrammatic Number II

String

Notes

Please read the above tips on sequence. They apply to strings too.

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

When you need to compare strings where the order isn't important

language has a built-in `Counter` class like Python, ask to 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 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.

Common data structures for looking up strings efficiently are

- Trie/Prefix Tree
- Suffix Tree

Common string algorithms are

- Rabin Karp, which conducts efficient searches of substrings, using a rolling hash
- KMP, which conducts efficient searches of substrings

Non-repeating characters

Use a 26-bit bitmask to indicate which lower case Latin characters are inside the string.

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

the two bitmasks. If the result is non-zero, `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 re-arranging 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 plausible approaches:

- Sorting both strings should produce the same resulting string. This takes O(nlgn) time and O(lgn) 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.
- 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 that reads the same backward and forward, such as *madam* or *racecar*.

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

- Have two pointers at the start and end of the string. Move the pointers inward till they meet. At any point in time, the characters at both pointers should match.

The order of characters within the string matters, so HashMaps 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.

- 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.

Corner Cases

- Empty string
- Single-character string
- Strings with only one distinct character

Practice Questions

- Longest Substring Without Repeating Characters
- Longest Repeating Character Replacement
- Minimum Window Substring
- Encode and Decode Strings

- Group Anagrams
- Valid Parentheses
- Valid Palindrome
- Longest Palindromic Substring
- Palindromic Substrings

Tree

Study Links

- Leaf It Up To Binary Trees

Notes

A tree is an undirected and connected acyclic graph.

Recursion is a common approach for 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`.

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

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

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

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.

Binary tree

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. Validate 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).

Corner Cases

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

Practice Questions

- Maximum Depth of Binary Tree

- Invert or Flip Binary Tree
- Binary Tree Maximum Path Sum
- Binary Tree Level Order Traversal
- Serialize and Deserialize Binary Tree
- Subtree of Another Tree
- Construct Binary Tree from Preorder and Inorder Traversal
- Validate Binary Search Tree
- Kth Smallest Element in a BST
- Lowest Common Ancestor of BST

Tries

Study Links

- Trying to Understand Tries
- Implement Trie (Prefix Tree)

Notes

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.

Sometimes preprocessing a dictionary of words (given in a list) into a trie, will improve the efficiency of searching for a word of length *k*, among *n* words. Searching becomes O(k) instead of O(n).

`remove`, and `search` methods.

Practice Questions

- Implement Trie (Prefix Tree)
- Add and Search Word
- Word Search II

Heap

Study Links

- Learning to Love Heaps

Notes

If you see a top or lowest *k* mentioned in the question, it is usually a sign 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. Whenever the heap size exceeds *k*, remove the minimum element. That will guarantee that you have the *k* largest elements.

Practice Questions

- Merge K Sorted Lists
- Top K Frequent Elements
- Find Median from Data Stream

Coding interviews are tough. But fortunately, you can get better at them by studying and practicing for them, and doing mock interviews.

To recap, to do well in coding interviews:

1. Decide on a programming language
2. Study CS fundamentals
3. Practice solving algorithm questions
4. Internalize the Do's and Don'ts of interviews
5. Practice by doing mock technical interviews
6. Interview successfully to get the job

By following these steps, you will improve your coding interview skills, and be one step closer (or probably more) to landing your dream job.

All the best!

The content for this post can be found in my Tech Interview Handbook repo on GitHub. Future updates will be posted there. Pull requests for suggestions and corrections are welcome!

yangshun/tech-interview-handbook
*tech-interview-handbook - 💯 Algorithms, front end and behavioral content for rocking your coding interview.*github.com

If you enjoyed this article, please don't forget to leave a 👏 . (Do you know that you can clap more than once? Try it and see for yourself!)

You can also follow me on GitHub and Twitter.



Show comments

Countinue reading about

Web Development

How to Kill Your Procrastination and Absolutely Crush It With Your Ideas

The Story of requesting twice - CORS

Soft skills every developer should have

See all 1597 posts →

#PROGRAMMING #WEB DEVELOPMENT #TECH

I learned to touch type at the ripe old age of 29. Was it worth it?

2 YEARS AGO

#PROGRAMMING #LEARNING TO CODE #SELF IMPROVEMENT

What programming language should I pick? Should I focus on front-end? Back-end? Machine learning?

2 YEARS AGO

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff. You can make a tax-deductible donation here.

Our Nonprofit

About
Donate
Shop
Sponsors
Email Us

Our Community

News
Alumni Network
Study Groups
Forum
Gitter
GitHub
Support
Academic Honesty
Code of Conduct
Privacy Policy
Terms of Service

Our Learning Resources

Learn
Guide
Youtube
Podcast
Twitter
Instagram