MASTERING DSA ROADMAP by HIMANSHU GUPTA

CONTENTS

ROADMAP INTERACTIVE GRAPH…OPEN THE LINK

https://roadmap.sh/datastructures-and-algorithms

1. Understand the Basics

a. Learn a Programming Language:

- **Choose a language**: C++, Java, or Python (C++ is highly recommended for DSA due to its Standard Template Library (STL)).
- **Resources**:
    - C++: "C++ Primer" by Stanley B. Lippman, or online tutorials like CodeChef or GeeksforGeeks.
    - Java: "Head First Java" by Kathy Sierra.
    - Python: "Automate the Boring Stuff with Python" by Al Sweigart.
- **Practice syntax**: Start with simple programs like Fibonacci series, palindrome check, etc.

b. Master Basic Concepts:

- Variables, data types, loops, conditional statements, and recursion.
- Practice questions from beginner-friendly platforms like HackerRank or from any good courses like from coding ninjas or coding blocks.

---

2. Learn Core Data Structures

a. Arrays:

- Understand the basics of arrays, indexing, and operations (insertion, deletion, traversal).
- Solve problems like:
    - Two Sum
    - Kadane's Algorithm (Maximum Subarray Sum)
    - Rotate Array

b. Strings:

- Learn string manipulation techniques, pattern matching, and palindrome problems.
- Problems to solve:
    - Longest Palindromic Substring
    - String Rotation Check

c. Linked Lists:

- Master singly and doubly linked lists.
- Implement operations (insertion, deletion, reverse).
- Problems to solve:
    - Detect Cycle in a Linked List
    - Merge Two Sorted Lists

d. Stacks and Queues:

- Learn stack and queue operations using arrays and linked lists.
- Understand real-world applications (parsing expressions, undo mechanisms).
- Problems to solve:
    - Next Greater Element
    - Implement Queue using Stacks

e. Hashing:

- Master hash tables and maps for efficient lookups.
- Problems to solve:
    - Longest Consecutive Sequence

- Subarray with Given Sum

f. Trees:

- Learn binary trees, binary search trees, and tree traversal techniques (in-order, pre-order, post-order, level-order).
- Problems to solve:
  - Lowest Common Ancestor (LCA)
  - Serialize and Deserialize Binary Tree

g. Graphs:

- Understand graph representations (adjacency list, adjacency matrix).
- Learn traversal algorithms (BFS, DFS) and shortest path algorithms (Dijkstra, Floyd-Warshall).
- Problems to solve:
  - Number of Islands
  - Minimum Spanning Tree

---

3. Algorithms

a. Sorting and Searching:

- Master common sorting algorithms (bubble sort, merge sort, quick sort).
- Learn binary search and its applications.
- Problems to solve:
  - Median of Two Sorted Arrays
  - Search in Rotated Sorted Array

b. Recursion and Backtracking:

- Master the art of solving problems using recursion and backtracking.
- Problems to solve:
  - N-Queens Problem
  - Rat in a Maze

c. Dynamic Programming (DP):

- Understand the principles of overlapping subproblems and optimal substructure.
- Problems to solve:
    - Longest Increasing Subsequence
    - 0/1 Knapsack Problem

d. Greedy Algorithms:

- Learn greedy principles and how to apply them to optimization problems.
- Problems to solve:
    - Activity Selection
    - Huffman Encoding

e. Divide and Conquer:

- Understand divide-and-conquer techniques.
- Problems to solve:
    - Merge Sort
    - Quick Sort

---

4. Build a Structured Study Plan

- Focus on learning a programming language and basic data structures.
- Practice beginner-level problems on LeetCode and start giving contests.
- Dive deeper into advanced data structures (trees, graphs, trie) and algorithms.
- Start solving medium-level problems on platforms like LeetCode, InterviewBit.
- Start doing CP and start participating in contests on Codechef, Codeforces.
- Start upsolving those Questions you were not able to solve during the contests.
- Start learning Fenwick Tree, Segment Tree to solve range based problems.
- Contests are important to improve your thinking ability and build a mapping for each type of problems.

---

5. Practice, Practice, Practice

- Allocate at least 1–2 hours daily for problem-solving.
- Set goals to solve a specific number of problems weekly.
- Use these platforms:
    - Beginner: HackerRank.
    - Intermediate: LeetCode, GeeksforGeeks.
    - Advanced: Codeforces, CodeChef, AtCoder.

---

6. Participate in Competitive Programming

- Join coding contests regularly to improve speed and accuracy.
- Platforms: Codeforces, CodeChef, AtCoder, LeetCode Contests.

---

7. Build Projects

- Apply DSA knowledge in real-world projects, such as:
    - Building a file compression tool using Huffman encoding.
    - Creating a mini search engine using Trie data structures.

---

8. Resources for Guidance

- MIT lectures on DP or any other topics are goldmine.
- Books:
    - “Introduction to Algorithms” by Cormen, Leiserson, Rivest, and Stein.
    - “The Algorithm Design Manual” by Steven S. Skiena.
- Online Courses:
    - "DSA Specialization" on Coursera by UC San Diego.
    - "Data Structures and Algorithms" on Udemy.

---

9. Consistency is Key

- DSA is not mastered overnight. Practice consistently.
- Track your progress with spreadsheets and set achievable milestones.

---

10. Connect with Peers and Mentors

- Join college coding clubs or communities.
- Participate in hackathons to gain collaborative problem-solving experience.
- Make a healthy circle with your coder friends to do better.

By following this roadmap, you can systematically master DSA during college and excel in placements, competitive programming, and beyond. Stay consistent and keep challenging yourself!

HOW TO APPROACH A SPECIFIC DSA PROBLEM EFFICIENTLY!

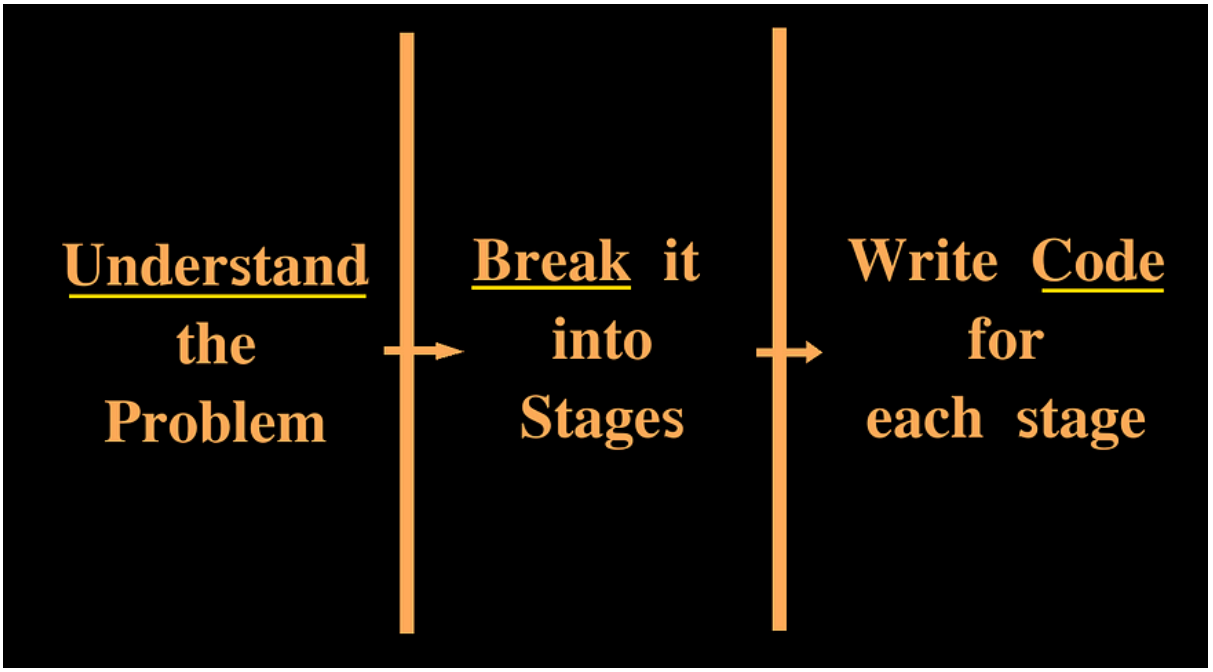
Few notes about DS & Algos, before we start

1. These highly optimised DS & Algos you see everyday, took years to develop by some of the best minds working on that problem. So, don't feel depressed if you can't get it the first time you read it, cause the person who wrote it, spent years refining it.
2. DS & Algos comes easily to those inquisitive minds who enjoy puzzles and problem-solving. In my case, I loved problem solving, still these problems would scare the life out of me. I couldn't even visualise the algorithm iterations and it was giving me brain freeze. To fix it, I started with a lot of super simple problems for many weeks and worked my way up. This fixed my initial brain freeze issues.
3. No matter how difficult the problem, we don't cheat. We start with the **simplest dumbest** code to solve that problem. We only take reference once we have taken a stab at the problem from different angles.

My philosophy on DSA is that — *"You shouldn't be focused on DSA too much. Fall in love with programming first and eventually there will be this hunger to make your code faster & memory-optimised. That's the stage where you organically absorb these DSA concepts. So, instead of jumping into competitive coding, try to build something first. Find your reason to code everyday. "*

Now, Let's get to business

Every Data Structure and Algorithm problem can be solved if we follow these 3 steps correctly. Take one step lightly and you'll suffer!



Three steps of solving DSA problems.

Let's unpack these 3 steps.

Step 1: Understand the Problem

When we come across a DSA problem, we read through it, understand the given input and expected output. We try it out in pen and paper.

We do not move to the next step unless given some random input we can find the expected output in pen & paper ourselves. This shows that we have a clear understanding of the problem statement.

Therefore, to validate if Step one is complete, we take random inputs and find out the expected outputs. If it is correct every time we can move to step two.

Step 2: Break it into stages

If we solve the problem in pen & paper multiple times during step one, we start seeing the stages appear. Note down these stages and they will become your building blocks for the final solution.

The trick here is to keep breaking the stages unless you feel, you can write a function to replace that stage in Step three. Also don't worry about the loops-recursions here, just write the bare minimum stupid stages to break the problem. It doesn't matter at all if our stages are stupid and slow, we'll keep on refining it in the future.

Follow @codeprime.io on Instagram

Always remember that there are dozens of ways to solve that problem. If we do not try to solve it "*our way*", then Google will show us the most optimised but difficult way of doing it. We'll never be able to solve the same problem ourselves and these problems will keep coming back in different flavours to bully us. If we don't try the stupid ways first then there will never be any respect for the smart ways.

Not able to identify all the stages? Go back to step one and solve it again & again with different inputs. Move to step three only if you feel confident about writing code for all the stages you have just discovered.

Step 3: Write Code for those stages

This is the easiest and most rewarding stage of all three!

If you have been coding daily for past few weeks then this stage will tickle your brain and your solution will start emerging as code smoothly. This is so satisfying that sometimes we stay lost in this bliss for days.

If you are constantly struggling to convert those stages to code then **QUIT that problem right away** or you will hurt yourself or even worse you'll cheat or stay depressed for days. It's not worth it to break our daily coding rhythm for some problem. After quitting that problem, come back to basics and work on your personal projects to stay motivated. Meanwhile, find online resources and stories related to that problem you quit. Join some online programming community chat rooms or talk to your mentors about that problem. Basically, you're trying to understand your enemy here and slowly but surely weaponizing yourself with missing concepts to beat that problem someday. No need to rush ourselves here, if you are coding daily on something, I personally guarantee that someday in the next few weeks you WILL feel a strong urge to come back to that problem and solve it.

Conclusion

These three steps are simple but has the power to make or break you. I followed it religiously and you know the trick lies in Step 3, to ignore the problem and keep coding regardless. That's how I was able to develop this strong and confident mental model, where I am never scared of jumping into a problem solving session. I hope this guide helps you get out of your DSA fear and reverse bully the problems.

Always remember, you're struggling because either one of three is happening.

1. **You haven't been coding everyday.**
2. **You didn't start with super simple problems and skipped to tough ones.**
3. **You didn't do your homework on your enemy and might be fighting it alone.**

HOW TO USE LEETCODE EFFICIENTLY

The goal of this article is not to teach you how to scramble your way into a top tech company, but to help you learn how to acquire algorithmic problem solving skills, which in turn will help you get the jobs you want. We'll start with some general tips that apply pretty much to anything that you want to get better at:

- Consistency is key. You want to start as much in advance as you can, and do it daily. Doing it for one hour a day is better than doing it for seven hours on Sunday only.
- Focus on active improvement. This means that if you're on auto-pilot or if you're not focusing, then it's OK to just stop and come back later.
- Remember that everybody is different and there is no one-size-fits-all. So feel free to deviate from this (or any) guide.

For generalist SWE roles, you should focus on the Algorithms type of problems. Years ago, these were the only problems, but since then, Database, Shell, and Concurrency have been added. So you should view only Algorithms problems in the problem set:

https://leetcode.com/problemset/algorithms/

This is going to be your "home base", so let's customize it.

- You can click the "Difficulty" header twice to sort it from easiest to hardest. Interesting enough, this also sorts them within each Easy/Medium/Hard category in reverse order by acceptance rate.
- Click the empty header in the first column to sort with the unsolved problems on top.

Now you're ready to go. Obviously the best thing you can do it just solve every single problem on LeetCode. If you do that, you're really good to go. But, unfortunately, nobody has unlimited time on their hands, so we're going to have to optimize a little.

- Start from easiest to hardest. You can start skipping questions if you feel like the problems are getting too easy for you.
- Start with problems that have an editorial already written. These are the ones with a little "document page" icon in the "Solution" column of the problem set. As an exercise to the reader, you could re-purpose the script above to filter only for problems that have a per-written editorial.
- Start with problems that have good reviews. While LeetCode is a pretty great platform, not all problems are created equal. If you open a problem, you can see how many people upvoted or downvoted a problem. I'd initially stay away from problems

that have a worse than 2:1 ratio of upvotes to downvotes, and problems that have 4:1 or above are usually of fairly high quality. It's much easier to learn from the higher-rated problems.

Eventually, you're going to find a problem that's too hard, and you'll get stuck. That's totally fine. In fact, it's absolutely fine (and perhaps even efficient) to give up on some problems. It's possible that it requires an algorithm or data structure that you haven't seen before, and there's no need for you to pull your hair out trying to re-discover an algorithm or data structure. Here are some tips for how to get un-stuck.

- First, if there is an editorial already written for the problem, start with that. These tend to be fairly high-quality and often include a well-written code solution.
- Next, open the "Discuss" tab for the problem, and read some of the posts. These vary wildly in quality: some people just post a solution; others go into great detail. Your mileage may vary here, but this is also often a helpful resource if you get stuck.
- Finally, you can Google the problem itself, and oftentimes other people will have posted solutions or written about it on third-party sites, like their personal blog or Github repo.

Once you've gotten into the groove, and you feel mostly confident solving the problems on LeetCode, it's time to refine your skills.

- If you feel weaker in a certain algorithm, you can filter the problem set by "Tag". This is NOT recommended for general practice, since much of the actual problem-solving skill you want to have is the ability to identify the type of algorithm to solve a problem. So, if you have filtered by "Binary Search", you know the solution to the problem is probably going to be binary search.
- Go through some older problems, and make sure that you have found the optimal solution. Oftentimes, the LeetCode online judge will actually accept suboptimal solutions. If an O(N) solution exists, but you submitted an O(N log N) solution, most likely it's going to still pass. The percentiles for runtime/memory are actually a bit misleading so don't worry too much about that. The only time that it kind of helps if there are two very different runtimes, like O(N) vs O(N^2), and the runtime distribution will look bimodal.
- The weekly contests are a great way to see where you stack up against the rest of the community. Plus, they apply some time pressure, and usually give new problems that you haven't seen before.
- This is new and not available several years ago, but the new Mock Interview is actually really good for adding some time pressure. Unfortunately, there is no way to filter out previously solved problems, so you might get repeats of ones you've done

before--but you can view this as an opportunity to revisit some of the older problems. Apparently, this can also help you identify areas of weakness.

I've never used it, but hearing from others, I'd consider it to almost be "cheating" but LeetCode premium is actually pretty good if you're really in a crunch and want to see what questions companies tend to ask. I do not recommend studying for a specific company, since your goal should be to build up algorithmic problem solving skills, rather than memorizing the answers to a few known problems. However, if you are really in a time crunch, then this could be your best bet. (I'm recommending paying for LeetCode premium, and I'm not even sponsored by LeetCode, believe it or not! That's how good it is.)

Finally, if you've got some money to spare, or have a nice friend or library who can lend you the book, I also strongly recommend "Cracking the Coding Interview". Many of those problems are also available on LeetCode (see this list), so you can actually use LeetCode as a complementary tool alongside the book. I would consider the book to be fairly entry-level, so if you're new to technical interviews, I'd recommend starting by going through the book, even before doing LeetCode full-time.

Follow @codeprime.io on Instagram