Mastering Problem Solving

A Comprehensive Roadmap to C++ & DSA

The Secret Weapon for Landing Your Dream Tech Job

www.gurucodes.dev

Table of Contents

About me

> *I'm Vasanta Kumar, a software developer, educator. I followed a similar path as many of you. I cracked the GATE exam, landed a high-paying job at KLA Tencor, and have been working there for the past 2 years. But my journey started with a desire to excel in the tech field, just like you.* ***I too was clueless when I first got started. There's a lot of figuring things out on your own. And that's exactly what I did.*** *Now, with over 100k followers on my Instagram page (gurucodes.dev) appreciating my teaching, because I've been through the entire journey myself, I understand the challenges students face while learning to code. This experience allows me to anticipate and address every single aspect they need to know about problem-solving, ensuring they have a comprehensive foundation. I'm excited to giveaway this comprehensive programming and DSA roadmap to help you achieve your coding dreams!*

Follow me on Instagram: https://instagram.com/gurucodes.dev

What is Programming and DSA?

Programming is essentially giving instructions to a computer in a way it can understand. It's like creating a recipe for the computer to follow, but instead of ingredients and cooking steps, you use code to tell it what data to work with and what actions to take.

Data structures are like specialized containers you use to organize data in a computer's memory. They define how the data is arranged and accessed, which plays a crucial role in how efficiently programs can work.

Why Learn Programming and DSA?

By mastering both Programming and DSA, you'll gain the ability to:

- **Think logically and solve problems efficiently**
- **Write clean, maintainable, and powerful code**
- **Prepare for technical interviews and coding challenges**
- **Build a strong foundation for a career in programming or software development**

DSA (Data Structures and Algorithms) and **problem-solving skills** are **very important for securing a high-paying job** in tech, and here's why:

High Demand & High-Paying Roles:

- **Technical Interviews:** Given the vast number of applicants, companies need a standardized method to evaluate core programming skills, rather than a resource-intensive, individual skills assessment for each candidate. Therefore, **companies often rely on coding exams to assess a candidate's problem-solving abilities**, which can be significantly enhanced by learning Data Structures and Algorithms (DSA)**.**
- **Technical Expertise:** Strong DSA and problem-solving skills are a hallmark of skilled developers and engineers. Companies are willing to pay a premium for these skillsets, as they directly translate to the ability to build complex and performant systems.
- **Career Growth:** Strong DSA proficiency opens doors to senior developer positions, system design roles, and even leadership opportunities in tech.

How to Develop these Skills:

- **Practice with Coding Platforms:** Platforms like LeetCode, HackerRank, and GeeksForGeeks offer a variety of coding problems to practice and hone your DSA skills.

Effective Learning Techniques

Learning programming, data structures & algorithms (DSA), and problem-solving effectively involves a combination of understanding concepts, practicing consistently, and utilizing various resources. Here's a breakdown:

Building a Strong Foundation:

1. **Pick a Programming Language:** Start with a language like C++, Java, Python. They have a lot of community support.
2. **Master the Basics:** Focus on core programming concepts like variables, data types, operators, control flow statements (if/else, loops), and functions.
3. **Learn About Complexity Analysis:** Understand how to analyze the time and space complexity of algorithms. This helps you choose the most efficient approach for solving problems.

Sharpening Your DSA Skills:

1. **Grasp Data Structures:** Start with fundamental data structures like arrays, linked lists, stacks, queues, trees, and graphs. Understand their operations, use cases, and trade-offs.
2. **Explore Algorithms:** Dive into common algorithms like searching, sorting, dynamic programming, recursion, and backtracking. Practice implementing them in code.
3. **Practice, Practice, Practice:** The key to mastering DSA is solving problems. Utilize online platforms like HackerRank, LeetCode, or Codeforces. Start with easy problems and gradually progress to more challenging ones.

Developing Problem-Solving Skills:

1. **Break Down Problems:** When faced with a problem, break it down into smaller, more manageable sub-problems. This will make the solution clearer and easier to implement.
2. **Identify Patterns:** Look for patterns in problem statements and existing solutions. This can help you determine the appropriate data structure or algorithm to apply. (Two-pointer, sliding window, prefix-sum, counting sort, greedy etc.)
3. **Test and Debug line-by-line in case of any error.**

Practicing for Interviews:

- **Find a Learning Community:** Join online forums, communities, or attend coding meetups to connect with other learners and get help when stuck.

- **Participate in Coding Challenges:** Take part in online coding competitions to test your skills and knowledge against others.
- **Be Patient and Persistent:** Learning to code and master DSA takes time and effort. Don't get discouraged by setbacks; keep practicing and learning from your mistakes.

Choosing your programming language

Here's a comparison table of C++, Java, and Python for competitive programming and problem-solving:

Feature	C++	Java	Python
Speed	Fastest	Medium	Slowest
Memory Usage	Manages memory manually	Automatic garbage collection	Automatic garbage collection
Syntax	More complex	Medium	Simplest
Readability	Can be less readable	More readable	Most readable
Development Time	Longer	Medium	Shorter
Libraries	Extensive (STL)	Large and diverse	Extensive (SciPy, NumPy)
Learning Curve	Steeper	Moderate	Gentlest
Competitive Programming	Best for efficiency-critical problems	Good all-rounder	Not ideal for tight time/memory constraints

Advantages:

- **C++:** Unmatched speed and memory control for complex algorithms. Large community and resources for competitive programming.
- **Java:** Platform-independent, good balance of speed and readability with extensive libraries. Large developer community.
- **Python:** Short development time, easy to learn and read. Extensive libraries for data science and machine learning problems.

Disadvantages:

- **C++:** Complex syntax and manual memory management can lead to errors. Steeper learning curve.
- **Java:** Can be verbose compared to Python. Slower execution speed compared to C++.
- **Python:** Not ideal for problems with tight time or memory constraints due to slower execution speed. While convenient, its built-in functions might hurt long-term problem-solving skills.

Choosing the right language:

- For competitive programming with a focus on efficiency, C++ is the go-to choice.
- If you value readability, ease of development, and a good balance of speed, Java is a solid option.
- Python is a great choice for problems with a strong emphasis on data manipulation and prototyping but might not be ideal for highly optimized solutions.

Problem-Solving and its Platforms

DSA problem-solving involves applying your knowledge of data structures (arrays, linked lists, trees, etc.) and algorithms (sorting, searching, dynamic programming, etc.) to devise efficient solutions for real-world or coding challenge scenarios. It's a crucial skill for programmers, as it helps building:

- **Logical Thinking, Algorithm Selection, Code Implementation, Efficiency Analysis.**

Problem-Solving Platforms (LeetCode, HackerRank, etc.):

- **LeetCode** and **HackerRank** are popular online platforms that offer a vast collection of coding challenges categorized by difficulty level, topic (arrays, strings, trees, graphs, etc.), and company interview questions (for targeted preparation).
- **Features:**
  - **Interactive Coding Environment:** Code, compile, and test your solutions directly on the platform.
  - **Test Cases:** Verify your code's correctness with provided test cases.
  - **Discussions:** Learn from other users' approaches and insights.
  - **Contests:** Participate in timed coding competitions to improve your skills under pressure.
  - **Skill Tracking:** Monitor your progress and identify areas for improvement.

How to Use These Platforms:

1. **Choose Your Level:** Start with easier problems to build your foundation and gradually increase the difficulty as you progress.
2. **Read the Problem Statement Carefully:** Understand the input format, expected output, and any constraints (time/memory limits).
3. **Plan Your Approach:** Think about the data structures and algorithms that might be suitable for solving the problem. Consider edge cases and potential optimizations.
4. **Code Implementation:** Write your code, ensuring clarity, efficiency, and correct handling of inputs.
5. **Test and Debug:** Use the provided test cases and write your own to catch errors.

6. **Analyze Time and Space Complexity:** Understand how your solution performs for different input sizes.
7. **Compare Solutions:** See how other users approached the problem and learn from different techniques. (Very very crucial)

Additional Tips:

- **Focus on Understanding, Not Just Getting the Answer.**
- **Practice Regularly.**
- **Don't Be Afraid to Ask for Help:** Utilize the platform's discussion forums or online communities to seek guidance if you're stuck.

Let's begin learning!

Start Here⬇

Module 1: Command Prompt

- ○ History
  - ○ Types of OS
  - ○ Why Servers prefer CLI over GUI?
- ○ Why should you learn Command Prompt?
- ○ How CMD is same as using a full OS?
- ○ Absolute and Relative Paths
- ○ A Command: Command + Arguments
- ○ Getting started with basic commands
- ○ Working with Folders: Creating, moving, deleting etc.
  - ○ Moving around folders
  - ○ Listing files from folders
- ○ Working with files: Creating, updating, deleting files
- ○ Revision

Module 2: VS Code

- ○ What are code editors/text editors?
- ○ Visual Studio Code
- ○ Installing Visual Studio Code
- ○ Opening VS Code from explorer/command prompt
- ○ Opening Files & Folders
- ○ Creating Files & Folders
- ○ Adding Plugins
- ○ Opening Command Prompt from Code Editor
- ○ Revision

Module 3: Introduction to Programming

- ○ Programming and Benefits
- ○ CPP Introduction and Benefits
- ○ Compilers and Lifecycle of a Program
- ○ Install GCC Compilers
- ○ Compiler and run your first program

Module 4: Data Types and Variables

- ○ Data Types
- ○ Variables
- ○ Variables of Different Data Types
- ○ Problems on Variables
- ○ ASCII Table

- ○ First Program
- ○ Comments
- ○ Revision & basic problem solving

- ○ Type Conversions
- ○ Macros & Type Range Macros
- ○ Revision & basic problem solving

Module 5: Input_Output

- ○ Input
- ○ Output

Module 6: Maths required for Problem Solving

- ○ Number Systems
- ○ Binary Number System
- ○ Converting one number system to another
- ○ Division, Modulus
- ○ Factors, multiples of a number
- ○ LCM, HCF
- ○ Prime Number, Prime Factorization
- ○ AP Series, Factorial
- ○ Matrices (Basics)
- ○ Graph Theory (Can also be learnt later)
- ○ Revision

Module 7: Operators:

- ○ Introduction to Operators
- ○ Arithmetic Operators
- ○ Relational Operators
- ○ Bitwise Operators
- ○ Logical Operators
- ○ Assignment Operators
- ○ Increment Operators
- ○ Miscellaneous
- ○ Operator Precedence
- ○ Basic problems on all the above topics
- ○ Bit Manipulation(Should have been a whole new module)
  - ○ Bit Manipulation Concepts
- ○ Revision

Module 8: Problem Solving

- ○ Problem Solving Introduction
- ○ Problem Format
- ○ Understanding Constraints
- ○ Reading an integer variable and printing the same in console.

Module 9: Conditional Statements

- ○ If else conditions
- ○ Ternary Operator
- ○ Nested If else
- ○ Determine if a person is eligible for voting

- ○ Reading different data type variables and printing the same in console.
- ○ Sum of 2 numbers
- ○ Swap 2 numbers
- ○ Swap 2 numbers without using third variable
- ○ Write a function that converts a temperature from Fahrenheit to Celsius.
- ○ Write a program that calculates the simple interest.
- ○ Cube of a number
- ○ Inbuilt functions: abs, power, sqrt
- ○ Debugging
- ○ Watch variables
- ○ Revision of above problems

- ○ If a number is even or odd?
- ○ Check if a number is divisible by 6?
- ○ Minimum of 2 numbers
- ○ Maximum of 2 Numbers
- ○ Minimum of 3 numbers
- ○ Maximum of 3 numbers
- ○ Whether a number is positive, negative or zero?
- ○ Leap year or not?
- ○ Switch-Case
- ○ Revision of above problems

Module 10: Loops

- ○ For Loop
  - ○ Tricky question - what is for(::) {}
  - ○ Tricky question - dummy loops for();
- ○ While Loop
- ○ Converting for loop to while loop
- ○ Converting while loop to for loop
- ○ Different ways of solving the same problem with minor tweaks in the for loop
- ○ Read input elements till EOL (when no size is given) .

Module 11: Variables & Scope

- ○ Default/garbage values of Variables
- ○ Local Variables
- ○ Global Variables

Module 12: Functions

- ○ Functions Introduction
- ○ return
- ○ Passing Parameters
- ○ Call by value
- ○ Call by reference

○ Problems on Loops(solve with both while loops and for loops)
  - ○ Print 1-100 numbers
  - ○ Print numbers from 100-1
  - ○ Print only the odd numbers
  - ○ Sum of 1-100 numbers
  - ○ Sum of first N numbers
  - ○ Revision of above problems
  - ○ Print digits of a number
  - ○ Sum of digits of a number
  - ○ Reverse a number
  - ○ Finding 2 power x
  - ○ Finding x power y
  - ○ Multiplication table
  - ○ Palindrome
  - ○ Check Prime Number
  - ○ Generate the Fibonacci series
  - ○ Find maximum element among the given inputs
  - ○ Find the minimum element among the given inputs
  - ○ Revision of above problems
  - ○ Sum of numbers in a given range
  - ○ Prime number within a given range
  - ○ Armstrong number
  - ○ Determine if a number is perfect square
  - ○ Adding factorials (For Example: 1!+ 2!+ 3!+ 4!+ 5!)

○ Function Scope

○ Practice functions by solving the previous questions(Very important to understand the working of functions)

○ Revision

Module 13: Strings

○ String Fundamentals

○ String Input and Output

○ String Manipulation

○ Substrings

○ Finding substrings to locate a substring's starting position

○ String conversion(uppercase to lowercase and lowercase to uppercase)

○ String Comparison

○ String Tokenization

○ Practice

○ Linear Search

○ Write a function that reverses a given string

○ Write a function that checks if a given string is a palindrome (reads the same backward as forward) regardless of case

○ Write a function that counts the number of vowels (a, e, i, o, u) in a given string, handling both uppercase and lowercase vowels

○ Write a function that removes all punctuation characters from a given string

  - ○ Maximum number consecutive same numbers among the given input
  - ○ Factorial of a number
  - ○ Factors of a number
  - ○ LCM, HCF
- ○ Do-While loop
- ○ Nested Loops
- ○ Different Pattern related questions(Google and have a look at them)(Very very important)
- ○ Revision of above problems

- ○ Write a function that counts the number of words in a given string
- ○ Write a function that replaces all occurrences of a specific character or substring with another character or substring in a string
- ○ Write a function that checks if two strings are anagrams of each other (contain the same letters with the same frequency)
- ○ Write a function that rotates a string by a given number of characters (e.g., rotate "Hello, world!" by 2 becomes "!lo, worldHel")
- ○ Revision

Module 14: Data Structures & Algorithms

- ○ Introduction to Data Structures
- ○ Why Data Structures
- ○ Time & Space Complexity
- ○ Understand Logarithm, Power, and Root Functions
- ○ Try answering time and space complexities of previously solved questions.
- ○ Algorithms
- ○ Revision

Module 15: Searching

- ○ Linear Search vs Binary Search
- ○ Binary Search : Understanding time/space complexity
- ○ Modify the binary search function to find the first or last occurrence of a target element in a sorted array that may contain duplicates.
- ○ Overflow case = (low+high) /2 alternative low + (high-low)/2
- ○ Revision

Module 16: Arrays

- ○ Array Fundamentals
- ○ Array Operations
- ○ Write a function to find the largest or smallest element in an array

Module 17: Sorting Algorithms

- ○ Bubble Sort
- ○ Selection Sort
- ○ Insertion Sort
- ○ Merge Sort

○ Write a function to calculate the sum of all elements in an array

○ Implement linear search to find a specific element in an array

○ Implement binary search to find a specific element in an array

○ Write a function to reverse the order of elements in an array

○ Given an array containing consecutive numbers with one missing number, find the missing number (assuming no duplicates)

○ Write a function to check if an array contains duplicate elements. Start with simpler cases like sorted arrays or arrays with a limited range of values

○ Rearrange array alternatively

○ Sort an array of 0s, 1s and 2s

○ Write a function to move all zeroes in an array to the end while maintaining the relative order of other elements

○ Given an array of numbers and a target sum, find two numbers that add up to the target sum (assuming there's one unique pair)

○ Merge 2 sorted arrays.

○ Trapping Rain Water

○ Chocolate Distribution Problem

○ Stock buy and sell

○ Spirally traversing a matrix

○ Revision

○ Quick Sort

○ Need for different Sorting Algorithms

○ Revision

Module 18: Stacks

○ Stack Fundamentals

○ Stack Operations Practice

○ Write a function that uses a stack to check if parentheses (round, square, curly) in a string are balanced (e.g., `((({})))` is balanced)

○ Write code to implement a stack using arrays

○ Write a function that uses a stack to convert an infix expression (e.g., `a + b * c`) to a postfix expression (e.g., `a b c * +`)

○ Write a function that uses a stack to evaluate a postfix expression (see above question) and return the result.

○ Revision

Module 19: Queues

○ Queue Fundamentals

○ Queue Operations Practice

○ Write code to implement a queue using either arrays

○ Given a queue containing characters, write a function to check if the queue is a palindrome

○ Explore how you can implement queue-like behaviour using two stacks

- ○ Revision

Module 20: Linked Lists

- ○ Linked Lists Fundamentals
- ○ Linked Lists Practice
- ○ Singly Linked List
- ○ Doubly Linked List
- ○ Circular Linked List
- ○ Write code to create a basic singly linked list with functionalities like adding nodes, printing the list, and finding the length
- ○ Write a function to reverse the order of nodes in a linked list (e.g., 1 -> 2 -> 3 becomes 3 -> 2 -> 1)
- ○ Write a function to determine if a linked list contains a cycle (a loop where a node points back to an earlier node)
- ○ Write a function to merge two sorted linked lists into a new sorted linked list
- ○ Write a function to remove duplicate nodes from a sorted linked list. Start with a simpler case where duplicates are consecutive
- ○ Write a function to find the middle node in a linked list (efficiently handle even and odd lengths)
- ○ Write a function to find the Nth node from the end of the linked list (consider cases where N is greater than the list length)
- ○ Write a function to calculate the sum of all elements in a linked list

Module 21: Trees

- ○ Trees Fundamentals
- ○ Trees Operations & Practice
  - ○ Trees Traversal
  - ○ Trees Searching
  - ○ Trees Insertion
  - ○ Trees Deletion
- ○ Binary Search Tree
- ○ Write code to create a basic binary tree with functionalities like adding nodes, printing the tree (pre-order, in-order, post-order), and finding the height
- ○ Write a function to verify if a given binary tree is a binary search tree (BST)
- ○ Search for a specific value in a BST
- ○ Find the minimum or maximum element from BST efficiently
- ○ Write a function to find the depth of a specific node in a tree (the number of edges from the root node to that node)
- ○ Write a function to calculate the sum of all node values in a tree using a chosen traversal method.
- ○ Implement a function to check if a binary tree is balanced (all leaves have roughly the same depth)
- ○ Write a function to create a mirror image of a binary tree (left subtree becomes right subtree and vice versa)
- ○ Revision

○ Write a function to create a deep copy of a linked list, ensuring a new list with independent nodes

○ Revision

Module 22: Heaps

○ Heaps Introduction

○ Min-Heap

○ Max-Heap

○ Remove and return the root node (min/max element) while maintaining the heap property

○ Add a new element to the heap and re-arrange nodes to maintain the heap property

○ Change the value of an existing node in the heap

○ Write code to create a min-heap using an array, with functionalities like insert, extract minimum, and printing the heap in level order

○ Similar to above question, but implement a max-heap

○ Given an array, write a function to determine if it represents a valid min-heap or max-heap based on the heap property.

○ Find the kth largest element in an array efficiently using a min-heap. Add elements to the heap, ensuring it only contains the k largest elements, and then return the root (minimum) which will be the kth largest element.

Module 23: Graphs

○ Graphs Introduction

○ Nodes, Vertices

○ Directed Graphs vs Undirected Graphs

○ Weighted vs Unweighted Graphs

○ Adjacency List

○ BFS, DFS

○ Implement BFS to traverse a graph and print the nodes visited in the order they are explored.

○ Implement DFS to traverse a graph

○ Write a function using DFS to determine if an undirected graph contains a cycle (a loop where a node connects back to itself or an ancestor).

○ Check for Connected Components: In an undirected graph, connected components are groups of nodes reachable from each other

○ Given a weighted undirected graph, find a subset of edges that connects all nodes with the minimum total weight, forming a tree structure. Start with simpler cases like Kruskal's algorithm for dense graphs.

○ Find the number of islands

○ Find whether path exist

- ○ Given an array of k sorted linked lists, write a function using a min-heap to merge them into a single sorted linked list.
- ○ Understand the basic concept of Huffman coding for data compression, which uses a min-heap to assign codes based on symbol frequencies.
- ○ Revision

- ○ Minimum Cost Path
- ○ Dijkstra's algorithm
- ○ Bellman-Ford algorithm
- ○ Prim's algorithm
- ○ Kruskal's algorithm
- ○ Revision

Module 24: Hashing

- ○ Hash Table
- ○ Collision Resolution Techniques
- ○ STL: Set, Map
- ○ Time & Space Complexity of Set & Map
- ○ Use a hash table to count the occurrences of each word in a given text string.
- ○ Given two strings, write a function using a hash table to check if they are anagrams (have the same letters but possibly in a different order).
- ○ Relative Sorting
- ○ Sorting Elements of an Array by Frequency
- ○ Largest subarray with 0 sum
- ○ Common elements
- ○ Count distinct elements in every window
- ○ Array Subset of another array
- ○ First element to occur k times
- ○ Revision

Module 25: Recursion

- ○ Recursion Introduction
- ○ Benefits and Drawbacks
- ○ Write a function that calculates the factorial of a non-negative number (n!) using recursion
- ○ Implement a function that generates the Fibonacci sequence using recursion
- ○ Write a recursive function to find the greatest common divisor (GCD) of two positive integers using the Euclidean algorithm
- ○ Implement a function that reverses a string using recursion
- ○ Write a recursive function to perform binary search on a sorted array
- ○ Implement a function to calculate the sum of all elements in an array using recursion
- ○ Write a function to check if a string is a palindrome using recursion
- ○ Given a binary tree structure, implement a recursive function to perform an in-order traversal

○ Revision

Module 26: Greedy Algorithms

○ Sorting

○ Activity Selection

○ N meetings in one room

○ Coin Piles

○ Maximize Toys

○ Largest number possible

○ Minimize the heights

○ Minimize the sum of product

○ Geek collects the balls

○ Revision

Module 27: Divide and Conquer

○ Divide and Conquer Concept

○ Find the element that appears once in sorted array

○ Search in a Rotated Array

○ Binary Search

○ Sum of Middle Elements of two sorted arrays

○ Quick Sort

○ Merge Sort

○ K-th element of two sorted Arrays

○ Revision

Module 28: Backtracking

○ Backtracking Concept

Module 29: Dynamic Programming

○ Dynamic Programming Concept

○ Bottom-Up vs. Top-Down Approach

○ Fibonacci Series using DP

○ Minimum Operations

○ Max length chain

○ Minimum number of Coins

○ Longest Common Substring

○ Longest Increasing Subsequence

○ Longest Common Subsequence

○ 0 – 1 Knapsack Problem

○ Maximum sum increasing subsequence

○ Minimum number of jumps

○ Edit Distance

○ Coin Change Problem

○ Subset Sum Problem

○ Box Stacking

○ Rod Cutting

○ Path in Matrix

○ Minimum sum partition

○ Count number of ways to cover a distance

○ Egg Dropping Puzzle

○ Optimal Strategy for a Game

○ Shortest Common Supersequence

○ Revision

- ○ N-Queen Problem
- ○ Solve the Sudoku
- ○ Rat in a Maze Problem
- ○ Word Boggle
- ○ Generate IP Addresses
- ○ Implement Permutation of an Array
- ○ Revision

Module 30: Trie

- ○ Trie Concept
- ○ Trie Implementation
- ○ Given an array of strings, find the longest common prefix shared by all strings. Use a trie to efficiently traverse the shared prefix path.
- ○ Revision

Module 31 - Suffix Trees

- ○ Suffix trees and arrays
- ○ Suffix Tree Operations
- ○ Practical Implementation
- ○ Applications
- ○ Longest Common Substring Problem
- ○ Longest Repeated Substring Problem
- ○ Longest Palindromic Substring Problem
- ○ Revision

Module 32 - Advanced Data Structures

- ○ Bloom Filters
- ○ Self-Balancing Trees
- ○ Red-Black Trees
- ○ Segment Trees
- ○ Disjoint Sets
- ○ LRU Cache
- ○ Skip List
- ○ Revision

Essential Coding Patterns

1. Two Pointers:

- **Pros:** Brings time complexity of O(n^2) to O(n)
- **Challenges:** Might take some time to get used to the pointer's movement.

Example Problems:

1. Pair with Target Sum
2. Find Non-Duplicate Number Instances
3. Squaring a Sorted Array
4. Triplet Sum to Zero

2. Sliding Window:

- **Pros:** Mostly used in problems involving subarrays. Brings time complexity of O(n^2) to O(n)
- **Challenges:** Takes time to understand how to adjust the window size based on the problem.

Example Problems:

1. Maximum Sum Subarray of Size K
2. Fruits Into Baskets
3. Longest Substring with K Distinct Characters
4. Longest Substring with Same Letters after Replacement

3. Island (Matrix Traversal) Pattern

- **Challenges:** Complex and occupies more space.

Example Problems:

1. Number of Islands

2. Biggest Island
3. Flood Fill

4. Slow and Fast Pointers

- Used mainly for:
  - Cycle Detection
  - Finding Middle Elements

Example Problems:

1. LinkedList Cycle
2. Middle of the LinkedList
3. Palindrome LinkedList

5. Counting Sort

- **Usage:** to sort elements when the range of elements is small.

Example Problems:

1. Height Checker - LeetCode
2. Array Partition - LeetCode

6. Merge Intervals

- Used mainly for:
  - Overlapping Intervals
  - Interval Scheduling

Example Problems:

1. Merge Intervals
2. Insert Interval
3. Intervals Intersection

7. Cyclic Sort

- Used mainly for:
  - Consecutive Numbers
  - In-Place Sorting

Example Problems:

1. Find the Missing Number
2. Find all Duplicates
3. Duplicates In Array

8. In-place Reversal of a Linked List

Usage: Used for reversing a Sub-Linked List or Sub-list/array.

Example Problems:

1. Reverse a LinkedList
2. Reverse a Sub-list
3. Reverse Every K-element Sub-list

9. Subsets

- Used mainly for:
  - Combinatorial Problems
  - Exhaustive Search

Example Problems:

1. Subsets
2. Subsets With Duplicates
3. Permutations

10. Modified Binary Search

Example Problems:

1. Order-agnostic Binary Search

2. Ceiling of a Number
3. Next Letter

11. Bitwise XOR

- Used mainly for:
  - Finding Missing or Duplicate Numbers
  - Bit Manipulation

Example Problems:

1. Single Number
2. Two Single Numbers
3. Complement of Base 10 Number

12. Top 'K' Elements

- Used mainly for:
  - Priority Queue
  - Streaming Data

Example Problems:

1. Top 'K' Numbers
2. Kth Smallest Number
3. ‘K’ Closest Points to the Origin

13. K-way Merge

- Used mainly for:
  - Multiple Sorted Arrays
  - External Sorting

Example Problems:

1. Merge K Sorted Lists

2. Kth Smallest Number in M Sorted Lists
3. Find the Smallest Range Covering Elements from K Lists

14. Topological Sort

- Used mainly for:
  - Task Scheduling

Example Problems:

1. Topological Sort
2. Tasks Scheduling
3. Tasks Scheduling Order

15. Trie

- Used mainly for:
  - Autocomplete
  - Spell Checker
  - IP Routing

Example Problems:

1. Insert into and Search in a Trie
2. Longest Common Prefix
3. Word Search

16. Monotonic Stack

- Used mainly for:
  - Next Greater or Smaller Element
  - Maximum Area Histogram

Example Problems:

1. Next Greater Element (NGE) for every element in given Array

2. Next Smaller Element
3. Largest Rectangular Area in a Histogram using Stack
4. The Stock Span Problem

17. 0/1 Knapsack

- Used mainly for:
  - Resource Allocation
  - Budgeting

Example Problems:

1. 0/1 Knapsack
2. Equal Subset Sum Partition
3. Subset Sum

18. Prefix Sum

Example Problems:

1. Equilibrium index of an array
2. Find if there is a subarray with 0 sums
3. Maximum subarray sum modulo m
4. Maximum occurred integer in n ranges

Resources:

1. book.pdf (cses.fi)
2. Main Page - Algorithms for Competitive Programming (cp-algorithms.com)

Additional Challenges

○ Creating Realistic Goals, Timeframes and Study Schedule

○ Staying Motivated

○ Overcoming Hurdles

○ Finding a mentor/guide

By following this comprehensive roadmap, you'll develop the capabilities to confidently pursue jobs with salaries ranging 10, 20 LPA and even more.

You'll also have to learn :

○ OOPS Concepts,

○ Operating Systems Concepts,

○ Database Concepts,

○ Networking Concepts

While some of the above concepts can be learnt quickly, mastering Data Structures & Algorithms (DSA) and problem-solving skills takes dedicated practice, typically ranging from 6 months to 2 years.

You can solve problems from Leetcode to get more expertise on each topic.
Feel free to take a printout of the roadmap and use it as a checklist to track your progress.

> Remember, consistency is key! Dedicate some time daily or weekly to practice, even if it's just solving a few problems.