

Data Structure specific algorithms

1. Arrays

- Sorting:
 - QuickSort: Efficient average-case time complexity ($O(n \log n)$)
 - MergeSort: Stable sort, useful when order matters ($O(n \log n)$)
- Searching:
 - Binary Search: Fast search in sorted arrays ($O(\log n)$)
- Two Pointers:
 - In-place manipulation, often for sorted arrays (e.g., removing duplicates)
- Sliding Window:
 - Subarray problems, finding maximum/minimum within a window

2. Linked Lists

- Traversal:
 - Iterate through the list, understand the node structure
- Insertion/Deletion:
 - At beginning, end, or at a specific position
- Reversal:
 - In-place reversal, recursive and iterative approaches
- Cycle Detection:
 - Floyd's Tortoise and Hare algorithm

3. Hash Tables (Hash Maps/Sets)

- Implementation not needed. Just understand following:
 - Understand how hash functions work
 - Insertion/Deletion/Lookup

- Collision Handling

4. Trees (Binary Trees, Binary Search Trees, etc.)

- Traversal:
 - Inorder, Preorder, Postorder (recursive and iterative)
- Searching:
 - Find a node with a given value (especially in BSTs)

5. Stacks

- Implementation not needed. Just understand following:
 - Push/Pop/Peek Operations

6. Queues

- Implementation not needed. Just understand following:
 - Enqueue/Dequeue Operations

7. Heaps (Priority Queues)

- Implementation not needed. Just understand following:
 - Insertion/Deletion (extract-min/max)
 - Building a Heap
- Top K Elements:
 - Using a heap to find k largest/smallest elements

8. Graphs

- Traversal:
 - Breadth-First Search (BFS)
 - Depth-First Search (DFS)
- Shortest Path:
 - Dijkstra's Algorithm
- Cycle Detection:

- DFS

9. Tries

- Implement Trie from scratch
- Insertion/Searching:
 - For words/prefixes
- Autocompletion:
 - Using a trie for word suggestions

10. Union-Find (Disjoint Set)

- Implement Union-Find from scratch
 - Find/Union Operations
 - Cycle Detection in undirected graphs
-

General algorithms/techniques

1. Recursion

- Defining a problem in terms of itself, often leading to elegant and concise solutions.
- Solve: Factorial calculation, tree traversals, depth-first search.

2. Dynamic Programming

- Breaking down a problem into overlapping subproblems and storing solutions to avoid recomputation.
- Solve: Fibonacci sequence, Knapsack problem, Longest Common Subsequence.

3. Greedy Algorithms

- Making locally optimal choices at each step with the hope of finding a global optimum.

- Implement: Kruskal's algorithm for minimum spanning trees.

4. Backtracking

- Incrementally building solutions, exploring all possible paths, and abandoning invalid ones.
- Solve: Sudoku solver, N-Queens problem, generating permutations.

WHAT'S NEXT?

Once you have implemented the above algorithms, solve [*Interview Master 100*](#), which contains top 100 interview problems.

Each problem builds upon previous problems so that you can gradually expand your knowledge as you progress.