DATA STRUCTURES

1. Array

Concept: Arrays are contiguous blocks of memory used to store fixed-size data collections of the same type.

Declaration:

int arr[10]; // Declares an integer array with 10 elements.

Initialization:

int arr[5] = {1, 2, 3, 4, 5}; // Initializes with specific values.

Basic Operations:

- Access: arr[0] (access first element).
- Update: arr[2] = 10; (assign new value to the 3rd element).
- Traversal: Use loops to access all elements.

```
for (int i = 0; i < 5; ++i) {
   cout << arr[i];
}
```

In-built Functions:

- sizeof(arr) (returns size in bytes).
- std::begin(arr) and std::end(arr) (get iterators to start and end).

2. Vector

Concept: Dynamic arrays provided by STL, allowing flexible resizing.

Declaration:

std::vector<int> vec; // Empty vector.

Initialization:

std::vector<int> vec = {1, 2, 3};

std::vector<int> vec(5, 0); // 5 elements initialized to 0.

Basic Operations:

Add: vec.push_back(4); (adds element at the end).

Remove: vec.pop_back(); (removes the last element).

Access: vec[2]; (direct indexing).

Traversal:

for (auto val : vec) cout << val;

In-built Functions:

vec.size();, vec.empty();, vec.clear();

Sorting: std::sort(vec.begin(), vec.end());

3. Linked List

Concept: A linear data structure consisting of nodes, each containing data and a pointer to the next node.

Declaration:

```
struct Node {
   int data;
   Node* next;
};
```

Initialization:

```
Node* head = new Node{10, nullptr}; // Creates a node with data 10.
```

Basic Operations:

Insertion at the head:

```
Node* newNode = new Node{20, head};
head = newNode;
```

Traversal:

```
Node* temp = head;
while (temp != nullptr) {
   cout << temp->data;
   temp = temp->next;
}
```

In-built Functions: Use std::list for built-in linked list:

```
std::list<int> lst = {1, 2, 3};
lst.push_front(0);
lst.push_back(4);
```

4. Stack

Concept: A container following the Last In First Out (LIFO) principle.

Declaration:

std::stack<int> stk;

Basic Operations:

Push: stk.push(10);

Pop: stk.pop();

Peek: stk.top();

In-built Functions:

stk.empty();, stk.size();

5. Queue

Concept: A container following the First In First Out (FIFO) principle.

Declaration:

std::queue<int> q;

Basic Operations:

- Enqueue: q.push(10);
- Dequeue: q.pop();
- Access Front: q.front();, q.back();

In-built Functions:

q.empty();, q.size();

6. Deque (Double-Ended Queue)

Concept: A dynamic container that allows insertion and deletion at both ends.

Declaration:

std::deque<int> dq;

Basic Operations:

- Insert: dq.push_front(10); dq.push_back(20);
- Remove: dq.pop_front(); dq.pop_back();

In-built Functions:

dq.front(); dq.back(); dq.size(); dq.empty();

7. Priority Queue

Concept: A queue where elements are prioritized. By default, implements a max-heap.

Declaration:

std::priority_queue<int> pq;

Basic Operations:

- Add: pq.push(10);
- Remove: pq.pop();
- Peek: pq.top();

8. Set

Concept: Stores unique, ordered elements.

Declaration:

std::set<int> s;

Basic Operations:

Add: s.insert(10);

Remove: s.erase(10);

Search: s.find(10);

In-built Functions:

s.count(10);, s.empty();, s.size();

9. Map

Concept: A container storing key-value pairs in sorted order.

Declaration:

std::map<int, std::string> mp;

Basic Operations:

Add/Update: mp[1] = "Value";

Remove: mp.erase(1);

Search: mp.find(1);

In-built Functions:

mp.size();, mp.empty();

10. Unordered Map

Concept: Similar to std::map, but no guaranteed order (uses hash tables).

Declaration:

std::unordered_map<int, std::string> ump;

Basic Operations: Same as std::map.

11. Graph

Concept: Represented using adjacency list/matrix.

Adjacency List:

std::vector<int> adj[N];

adj[0].push_back(1); // Adds edge 0 -> 1.

Basic Operations: BFS, DFS using recursion or iterative methods.

12. Heap

Concept: A complete binary tree used for efficient min/max element retrieval.

Implementation:

std::make_heap(vec.begin(), vec.end());

Basic Operations:

- Insert: std::push_heap();
- Remove: std::pop_heap();

13. Trie

Concept: A tree-like data structure used for efficient string searching.

Implementation:

```
struct TrieNode {
   std::map<char, TrieNode*> children;
   bool isEndOfWord = false;
};
```

Basic Operations:

Insert, search, delete by traversing through characters.

INBUILT FUNCTIONS:

1. Array (C-style)

Although C-style arrays have limited built-in functions, the following utilities can be used:

1. std::begin(arr) - Returns a pointer to the beginning of the array.
2. std::end(arr) - Returns a pointer to the end of the array.
3. sizeof(arr) - Returns the size of the array in bytes.
4. std::sort(begin(arr), end(arr)) - Sorts the array.
5. std::reverse(begin(arr), end(arr)) - Reverses the array elements.
6. std::find(begin(arr), end(arr), value) - Finds an element.
7. std::accumulate(begin(arr), end(arr), initial) - Sums up elements.
8. std::fill(begin(arr), end(arr), value) - Fills the array with a value.
9. std::copy(begin(arr), end(arr), destination) - Copies elements to another array.
10. std::count(begin(arr), end(arr), value) - Counts occurrences of a value.

2. Vector

1. vec.push_back(value) - Adds an element to the end.
2. vec.pop_back() - Removes the last element.
3. vec.size() - Returns the number of elements.
4. vec.capacity() - Returns the storage capacity.
5. vec.clear() - Removes all elements.
6. vec.empty() - Checks if the vector is empty.
7. vec.front() - Accesses the first element.
8. vec.back() - Accesses the last element.
9. vec.insert(position, value) - Inserts at a position.
10. vec.erase(position) - Removes an element by position.

3. Linked List (using std::list)

1. lst.push_front(value) - Adds an element to the front.
2. lst.push_back(value) - Adds an element to the back.
3. lst.pop_front() - Removes the first element.
4. lst.pop_back() - Removes the last element.
5. lst.size() - Returns the size of the list.
6. lst.empty() - Checks if the list is empty.
7. lst.front() - Accesses the first element.
8. lst.back() - Accesses the last element.
9. lst.remove(value) - Removes all elements matching the value.
10. lst.reverse() - Reverses the list.

4. Stack

1. stk.push(value) - Pushes an element onto the stack.
2. stk.pop() - Removes the top element.
3. stk.top() - Returns the top element.
4. stk.size() - Returns the size of the stack.
5. stk.empty() - Checks if the stack is empty.
6. std::swap(stk1, stk2) - Swaps two stacks.
7. stk.emplace(value) - Inserts a new element.
8. stk.clear() (custom, for emptying stack via loop).
9. std::stack<int> stk2 = stk; - Copies a stack.

No built-in iterators, but can use temporary containers.

5. Queue

1. q.push(value) - Adds an element to the back.
2. q.pop() - Removes the front element.
3. q.front() - Accesses the front element.
4. q.back() - Accesses the back element.
5. q.size() - Returns the size of the queue.
6. q.empty() - Checks if the queue is empty.
7. std::swap(q1, q2) - Swaps two queues.
8. q.emplace(value) - Adds an element efficiently.
9. No direct iterators; use custom code.
10. Copy via std::queue<int> q2 = q;.

6. Deque

1. dq.push_front(value) - Adds an element to the front.
2. dq.push_back(value) - Adds an element to the back.
3. dq.pop_front() - Removes the front element.
4. dq.pop_back() - Removes the back element.
5. dq.front() - Accesses the first element.
6. dq.back() - Accesses the last element.
7. dq.size() - Returns the size of the deque.
8. dq.empty() - Checks if the deque is empty.
9. dq.insert(position, value) - Inserts at a position.
10. dq.erase(position) - Removes at a position.

7. Priority Queue

1. pq.push(value) - Adds an element.
2. pq.pop() - Removes the top element.
3. pq.top() - Accesses the highest-priority element.
4. pq.empty() - Checks if the priority queue is empty.
5. pq.size() - Returns the size of the priority queue.
6. std::priority_queue<int> pq2(pq); - Copies the priority queue.

Min-Heap: Use std::greater<>:

1. std::priority_queue<int, std::vector<int>, std::greater<int>> pq;

2. std::swap(pq1, pq2) - Swaps two priority queues.

Cannot iterate directly; use intermediate structures.

Custom comparison: Pass comparator function.

8. Set

1. s.insert(value) - Adds an element.
2. s.erase(value) - Removes an element.
3. s.find(value) - Finds an element.
4. s.count(value) - Checks if an element exists.
5. s.empty() - Checks if the set is empty.
6. s.size() - Returns the size of the set.
7. s.clear() - Removes all elements.
8. s.lower_bound(value) - First element >= value.
9. s.upper_bound(value) - First element > value.
10. Iterators: s.begin(), s.end(), etc.

9. Map

1. mp[key] = value; - Adds or updates a key-value pair.
2. mp.erase(key) - Removes a key-value pair.
3. mp.find(key) - Finds a key.
4. mp.count(key) - Checks if a key exists.
5. mp.size() - Returns the size.
6. mp.empty() - Checks if the map is empty.
7. mp.clear() - Removes all elements.
8. mp.begin(), mp.end() - Iterators.
9. mp.lower_bound(key) - First key >= given key.
10. mp.upper_bound(key) - First key > given key.

10. Unordered Map

1. ump[key] = value; - Adds or updates a key-value pair.
2. ump.erase(key) - Removes a key-value pair.
3. ump.find(key) - Finds a key.
4. ump.count(key) - Checks if a key exists.
5. ump.size() - Returns the size.
6. ump.empty() - Checks if the map is empty.
7. ump.clear() - Removes all elements.
8. Iterators: ump.begin(), ump.end().
9. std::swap(ump1, ump2) - Swaps two unordered maps.
10. ump.bucket_count() - Returns the number of buckets.

11. Graph

1. For adjacency list (using std::vector):
2. Add edges: adj[u].push_back(v);
3. Remove edges: std::remove(adj[u].begin(), adj[u].end(), v);
4. Traversal: Use DFS or BFS.
5. adj[i].size() - Degree of vertex.

6. std::find(adj[u].begin(), adj[u].end(), v) - Checks edge existence.
7. Clear graph: adj.clear();
8. adj[i].empty() - Checks if vertex has neighbors.

12. Heap

1. Use std::make_heap:
2. std::make_heap(vec.begin(), vec.end());
3. std::push_heap(vec.begin(), vec.end());
4. std::pop_heap(vec.begin(), vec.end());
5. std::sort_heap(vec.begin(), vec.end());
6. std::is_heap(vec.begin(), vec.end());
7. std::is_heap_until(vec.begin(), vec.end());

13. Trie

Custom implementations:

insert(), search(), delete() - Add/remove/search strings.

No direct in-built utilities; rely on map-based traversal.

TIME AND SPACE COMPLEXITY

DATA STRUCTURE	TYPES	SPACE COMPLEXITY	TIME COMPLEXITY (OPERATIONS)
ARRAY	1D Array, 2D Array, Dynamic Array (e.g., std::vector)	Static: O(n) Dynamic: O(n)	Access: O(1) Search: O(n) Insert/Delete: O(n)
STRING	Static String (fixed length), Dynamic String (resizable)	Static: O(n) Dynamic: O(n)	Access: O(1) Search: O(n) Insert/Delete: O(n)
LINKED LIST	Singly Linked List, Doubly Linked List, Circular Linked List	O(n)	Access: O(n) Search: O(n) Insert/Delete: O(1) (if node is given)
STACK	Array-based Stack, Linked List-based Stack	O(n)	Push: O(1) Pop: O(1) Top: O(1) Search: O(n)
QUEUE	Array-based Queue, Linked List-based Queue, Circular Queue, Priority Queue	O(n)	Enqueue: O(1) Dequeue: O(1) Front: O(1) Search: O(n)

TREE	Binary Tree, Binary Search Tree (BST), AVL Tree, Red-Black Tree	O(n)	Access/Search: O(log n) (balanced BST) Insert/Delete: O(log n)
HEAP	Min-Heap, Max-Heap, Binary Heap, Fibonacci Heap	O(n)	Insert: O(log n) Extract Min/Max: O(log n) Search: O(n)
HASH TABLE	Hash Map, Open Addressing, Chaining	O(n)	Insert/Search/Delete: O(1) (avg.) Worst: O(n) (collision resolution)
GRAPH	Adjacency List, Adjacency Matrix, Edge List	Adjacency List: O(V + E) Adjacency Matrix: $O(V^2)$	Add Edge: O(1) Add Vertex: O(1) DFS/BFS: O(V + E)
SET	Hash Set, Tree Set	O(n)	Insert/Delete/Search: O(1) (avg.) O(log n) (tree-based)
MAP	Hash Map, Tree Map	O(n)	Insert/Search/Delete: O(log n) (tree-based) O(1) (hash map avg.)
TRIE	Prefix Tree, Compressed Trie	O(m × n) (m: max string length, n: count)	Insert/Search/Delete: O(m) (m: length of the string)

DIFFERENT TYPES OF THE ABOVE MENTIONED DATA STRCUTURES

1. Arrays

- 1D Array: A simple linear array where elements are stored sequentially (e.g., int arr[10];).
- 2D Array: An array of arrays, used to represent matrices or grids (e.g., int arr[3][3];).
- Dynamic Array: An array that can resize itself when elements are added or removed (e.g., std::vector<int> in C++).
- Jagged Array: An array of arrays where each element can have different sizes (e.g., int arr[][3]).

2. Strings

- Static String: Fixed-length string, usually represented as a character array (e.g., char str[10];).
- Dynamic String: A string that can grow or shrink in size dynamically, often implemented as a sequence of characters with memory management (e.g., std::string in C++).
- StringBuffer (mutable): A string-like data structure that allows modifications without creating new strings (e.g., StringBuffer in Java, std::ostringstream in C++).

3. Linked List

- Singly Linked List: Each node has a pointer to the next node, but not to the previous one (e.g., struct Node { int data; Node* next; };).
- Doubly Linked List: Each node has pointers to both the next and previous nodes, allowing traversal in both directions (e.g., struct Node { int data; Node* next; Node* prev; };).
- Circular Linked List: A linked list where the last node points back to the first node (it could be singly or doubly linked).

4. Stack

- Array-based Stack: A stack implemented using a fixed-size or dynamic array (e.g., std::vector in C++).
- Linked List-based Stack: A stack implemented using a linked list where each node holds one element of the stack.

5. Queue

- Array-based Queue: A queue implemented using a fixed-size or dynamic array (e.g., std::deque in C++).
- Linked List-based Queue: A queue implemented using a linked list where each node represents one element of the queue.
- Circular Queue: A queue that connects the last element to the first to avoid wasted space (e.g., implemented with a fixed-size array).

6. Tree

- Binary Tree: Each node has at most two children (left and right).
- Binary Search Tree (BST): A binary tree where each node's left child is smaller, and the right child is larger.
- Balanced Tree: A tree that maintains a balanced structure, such as AVL or Red-Black Tree, to ensure logarithmic time complexity for operations.
- N-ary Tree: A tree where each node can have more than two children (e.g., ternary tree, quaternary tree).
- Heap: A special kind of binary tree used for implementing priority queues (min-heap or max-heap).

7. Heap

- Min-Heap: A binary heap where the parent node is smaller than or equal to its children.
- Max-Heap: A binary heap where the parent node is greater than or equal to its children.
- Binomial Heap: A more advanced type of heap with better time complexity for certain operations.
- Fibonacci Heap: A heap with improved asymptotic time complexity for several operations.

8. Hash Table

- Open Addressing: A hash table where collisions are resolved by probing (linear probing, quadratic probing, double hashing).
- Chaining: A hash table where each bucket is a linked list, and all elements that hash to the same index are stored in that list.
- Cuckoo Hashing: A method where two hash functions are used, and elements may "kick out" other elements from their spots when collisions occur.

9. Graph

- Adjacency Matrix: A 2D array representing the edges in a graph, where the cell at (i, j) indicates the presence or weight of an edge between vertex i and vertex j.
- Adjacency List: A list where each vertex points to a list of vertices it is connected to, which is more space-efficient for sparse graphs.
- Edge List: A list of all edges in the graph, typically as pairs (u, v) for each edge connecting vertices u and v.
- Incidence Matrix: A matrix representing the relationship between vertices and edges (less commonly used).

10. Set

- Unordered Set: A set with no particular ordering of elements, typically implemented using hash tables (e.g., std::unordered_set in C++).
- Ordered Set: A set where elements are stored in a sorted order, typically implemented using a balanced tree structure (e.g., std::set in C++).
- Multiset: A set that allows duplicate elements (e.g., std::multiset in C++).

11. Map

- Unordered Map: A map with no particular ordering of elements, typically implemented using hash tables (e.g., std::unordered_map in C++).
- Ordered Map: A map where elements are sorted based on keys, usually implemented using balanced trees (e.g., std::map in C++).
- Multimap: A map that allows multiple values for a single key (e.g., std::multimap in C++).

12. Trie

- Standard Trie: A tree-like data structure used to store strings, where each node represents a single character in the string.
- Compressed Trie: A more memory-efficient version of the Trie that compresses chains of nodes that share a common path.
- Radix Tree: A type of Trie that combines nodes with shared prefixes, often used to implement associative arrays.

13. Priority Queue

- Min-Priority Queue: A queue where the smallest element is given the highest priority (often implemented with a min-heap).
- Max-Priority Queue: A queue where the largest element is given the highest priority (often implemented with a max-heap).
- D-ary Heap: A generalization of binary heaps where each node can have D children, which is used to optimize certain operations.
- Space Complexity Considerations:
- Static Structures (like 1D arrays, static strings, or fixed-size heaps) typically have constant space usage.

Dynamic Structures (like dynamic arrays, linked lists, hash tables) require more space as elements are added, usually proportional to the number of elements stored.

Time Complexity Considerations:

1. Arrays are fast for direct access (O(1)) but inefficient for dynamic operations like insertion and deletion (O(n)).
2. Linked lists provide O(1) insertion and deletion at known positions but slower access (O(n)).
3. Trees (balanced) maintain log(n) time complexity for insertion, search, and deletion, but unbalanced trees may degrade to linear time (O(n)).
4. Hash tables are efficient for most operations (O(1) on average), but performance can degrade due to collisions.

80-20 RULE:

The 80/20 DSA Learning Strategy Understanding the 80/20 Rule
The 80/20 rule suggests that roughly 80% of the effects come from 20% of the causes.

Core Focus Areas (The 20% that yields 80% results)

1. Data Structures

Focus on these fundamental data structures:

- Arrays and Strings
- Hash Tables (HashMaps/HashSets)
- Linked Lists
- Stacks and Queues
- Trees (Binary Trees, Binary Search Trees)
- Graphs
- Heaps

2. Algorithms and Techniques

Master these problem-solving approaches:

- Binary Search
- Two Pointers
- Sliding Window
- BFS and DFS (for trees and graphs)
- Dynamic Programming (focus on common patterns)
- Greedy Algorithms
- Sorting algorithms (especially QuickSort, MergeSort)
- Recursion and Backtracking

3. Problem Patterns

Learn to identify these common patterns:

- Traversal problems (trees, graphs)
- Searching problems
- Interval problems
- Two-sum and its variations
- Matrix traversal
- Topological sort
- Subarray/substring problems
- Tree balancing and validation

Practical Implementation Plan

Week 1-2: Foundations

- Learn the basic data structures and their operations
- Understand time and space complexity (Big O notation)
- Solve 2-3 easy problems daily on platforms like LeetCode

Week 3-6: Pattern Recognition

- Group similar problems together
- Study one pattern per week
- Solve 20-30 problems that follow these patterns
- Create a personal cheat sheet of patterns and their applications

Week 7-12: Targeted Practice

- Focus on medium-difficulty problems
- Dedicate more time to your weaker areas
- Review company-specific question patterns

Final 4 Weeks: Refinement

- Mock interviews with peers or platforms like Pramp
- Time-constrained practice (mimic real interview conditions)
- Review and refine your solutions for common problems
- Practice explaining your thought process out loud

The 20% of Problems to Focus On

1. **Arrays/Strings:**
    - Two Sum and Three Sum
    - Container With Most Water
    - Sliding Window problems (Maximum Subarray, Longest Substring Without Repeating Characters)
    - Rotate Array/Matrix
2. **Linked Lists:**
    - Reverse a Linked List

- Detect Cycle
- Merge Two Sorted Lists
- Find Middle Element

3. **Trees:**
   - Binary Tree Traversals (Inorder, Preorder, Postorder)
   - Level Order Traversal
   - Validate Binary Search Tree
   - Lowest Common Ancestor
4. **Graphs:**
   - BFS/DFS implementations
   - Number of Islands
   - Course Schedule (Topological Sort)
   - Graph Connectivity
5. **Dynamic Programming:**
   - Climb Stairs
   - Coin Change
   - Longest Increasing Subsequence
   - Edit Distance
6. **Heap/Priority Queue:**
   - Kth Largest Element
   - Merge K Sorted Lists
   - Top K Frequent Elements

Quality Over Quantity

Rather than solving hundreds of problems, focus on:

- Solving problems multiple times to internalize patterns
- Implementing different approaches to the same problem
- Analyzing and optimizing your solutions
- Understanding the trade-offs between different solutions

Weekly Routine (20% of your time that produces 80% results)

1. **Learning (2-3 hours):**
   - Study one new concept or pattern
   - Watch explanatory videos or read articles
2. **Problem Solving (6-8 hours):**
   - Solve 10-15 carefully selected problems
   - Revisit and optimize previous solutions
3. **Review (2-3 hours):**
   - Review solutions from others
   - Analyze edge cases and optimizations
4. **Mock Interviews (2 hours):**
   - Practice explaining solutions
   - Time yourself and work under pressure

Tracking Progress

Keep a log of:

- Problems you've solved
- Patterns you've mastered
- Areas that need improvement
- Success rate for different problem types

Final Tips

1. **Consistency over intensity:** Regular practice is more effective than cramming
2. **Focus on understanding, not memorization:** Learn why solutions work
3. **Practice communication:** Being able to explain your solution is crucial
4. **Learn from failures:** Analyze where you went wrong
5. **Build a support system:** Join DSA study groups or forums

80/20 Deep Dive: Arrays

Key Concepts (20% that gives you 80% results)

1. **Basic Operations**
    - Insertion, deletion, and traversal
    - Time complexity of these operations
    - In-place vs. creating new arrays
2. **Common Patterns**
    - Two pointers technique

- Sliding window
- Prefix sums
- Binary search on sorted arrays
- Kadane's algorithm

3. **Memory Management**
   - Contiguous memory allocation
   - Understanding index-based access
   - Cache locality benefits

Problem-Solving Techniques (20% of techniques for 80% of array problems)

1. Two Pointers Technique

- **When to use**: Problems involving searching, sorting, or finding pairs
- **Implementation**: Initialize two pointers (often at opposite ends or same start) and move them based on conditions
- **Time complexity**: Usually O(n)
- **Space complexity**: Usually O(1)

2. Sliding Window

- **When to use**: Finding subarrays/substrings with certain properties
- **Implementation**: Maintain a window that expands/contracts as you iterate
- **Time complexity**: Usually O(n)
- **Space complexity**: Usually O(1) or O(k) where k is window size

3. Prefix Sums

- **When to use**: Range sum queries, finding subarrays with target sum
- **Implementation**: Precompute cumulative sums, then use them for quick range calculations
- **Time complexity**: O(n) preprocessing, O(1) per query
- **Space complexity**: O(n)

4. Binary Search on Arrays

- **When to use**: Searching in sorted arrays or finding insertion positions
- **Implementation**: Divide and conquer by comparing middle element
- **Time complexity**: O(log n)
- **Space complexity**: O(1) iterative, O(log n) recursive

Must-Know Problems (20% of problems that cover 80% of patterns)

Easy Problems (Foundation)

1. **Two Sum**
   - Find two numbers that add up to a target
   - Teaches hash table usage with arrays
   - Time: O(n), Space: O(n)
2. **Best Time to Buy and Sell Stock**
   - Find maximum profit from one transaction
   - Teaches tracking minimum values and calculating differences
   - Time: O(n), Space: O(1)
3. **Maximum Subarray Sum (Kadane's Algorithm)**
   - Find contiguous subarray with largest sum
   - Teaches dynamic programming fundamentals
   - Time: O(n), Space: O(1)

Medium Problems (Core Competency)

1. **Container With Most Water**
   - Find two lines that contain the most water
   - Teaches two-pointer technique optimization
   - Time: O(n), Space: O(1)
2. **Subarray Sum Equals K**
   - Find number of continuous subarrays that sum to target
   - Teaches prefix sum with hash map
   - Time: O(n), Space: O(n)
3. **Product of Array Except Self**
   - Compute product of all elements except current one without division
   - Teaches prefix/suffix product techniques
   - Time: O(n), Space: O(1)
4. **Next Permutation**
   - Rearrange numbers to the lexicographically next greater permutation
   - Teaches in-place array manipulation and pattern recognition
   - Time: O(n), Space: O(1)

Hard Problems (Mastery)

1. **Trapping Rain Water**
   - Calculate how much water can be trapped between elevations
   - Teaches two-pointer technique with dynamic programming concepts

- Time: O(n), Space: O(1)

2. **Median of Two Sorted Arrays**
    - Find median of two sorted arrays with logarithmic complexity
    - Teaches binary search on arrays and partition theory
    - Time: O(log(min(m,n))), Space: O(1)

Implementation Strategy (4-Week Plan)

Week 1: Foundations

- Learn basic array operations and time complexity
- Master the two-pointer technique
- Solve: Two Sum, Best Time to Buy and Sell Stock
- Practice: 5 easy array problems focusing on iteration and basic manipulation

Week 2: Core Patterns

- Learn sliding window technique
- Understand prefix sums
- Solve: Maximum Subarray, Container With Most Water
- Practice: 5 medium problems applying these patterns

Week 3: Advanced Patterns

- Learn binary search variations on arrays
- Understand in-place array manipulation
- Solve: Product of Array Except Self, Next Permutation
- Practice: 3-4 medium/hard problems that combine multiple techniques

Week 4: Mastery and Edge Cases

- Focus on edge cases and optimizations
- Master the hardest array problems
- Solve: Trapping Rain Water, attempt Median of Two Sorted Arrays
- Review all patterns and consolidate understanding

Implementation Tips

1. **When solving array problems:**
    - Always clarify input constraints (size, value range, duplicates)
    - Consider edge cases: empty array, single element, all identical elements
    - Think about in-place solutions to optimize space complexity
2. **Common pitfalls to avoid:**
    - Off-by-one errors in indexing
    - Not handling edge cases
    - Unnecessary traversals when one pass is sufficient
3. **Optimization techniques:**
    - Look for opportunities to use constant extra space
    - Consider sorted vs. unsorted input
    - Use early termination conditions when possible

Practice Methodology

1. Solve each core problem twice:
    - First attempt: Try to solve independently
    - Second attempt: Optimize and refine your solution
2. After solving a problem:
    - Analyze time and space complexity
    - Look for alternative approaches
    - Identify which core pattern it demonstrates

Signs of Mastery

- Identify the appropriate technique for a given array problem within minutes
- Implement two-pointer and sliding window solutions without hesitation
- Handle edge cases automatically in your solutions
- Explain the trade-offs between different approaches

80/20 Deep Dive: Hash Tables (HashMaps/HashSets)

Key Concepts (20% that gives you 80% results)

1. **Fundamentals**
    - Hash function purpose and properties

    - Collision resolution (chaining vs. open addressing)
    - Load factor and rehashing
    - Average vs. worst-case time complexity
2. **Operations to Master**
    - Insertion, deletion, and lookup
    - Iterating through keys, values, and entries
    - Handling duplicates and null values
3. **Common Use Cases**
    - Counting frequencies
    - Caching/memoization
    - Finding duplicates or unique elements
    - Two-sum type problems

Problem-Solving Techniques

1. Frequency Counting

- **When to use**: Problems involving counting occurrences or finding duplicates
- **Implementation**: Use elements as keys, counts as values
- **Time complexity**: Usually O(n)
- **Space complexity**: O(n) or O(k) where k is the number of unique elements

2. Lookup Optimization

- **When to use**: When you need O(1) average-case lookups
- **Implementation**: Pre-process data into a hash table for quick access
- **Time complexity**: O(n) preprocessing, O(1) per lookup
- **Space complexity**: O(n)

3. Caching Results

- **When to use**: Problems with repeated computations or subproblems
- **Implementation**: Store computed results with relevant keys
- **Benefits**: Converts exponential algorithms to linear or polynomial time

Must-Know Problems (20% of problems that cover 80% of patterns)

Easy Problems

1. **Two Sum**
    - Find pair that sums to target value
    - Teaches basic hash table lookup
    - Time: O(n), Space: O(n)
2. **Contains Duplicate**
    - Determine if array contains any duplicates
    - Teaches set operations and lookups
    - Time: O(n), Space: O(n)

Medium Problems

1. **Group Anagrams**
    - Group strings that are anagrams of each other
    - Teaches using hash tables with custom keys
    - Time: O(n * k log k) where k is max string length, Space: O(n * k)
2. **LRU Cache**
    - Implement Least Recently Used cache with O(1) operations
    - Teaches hash table with doubly linked list
    - Operations: O(1) time complexity
3. **Longest Consecutive Sequence**
    - Find length of longest consecutive elements sequence
    - Teaches hash set optimization for lookups
    - Time: O(n), Space: O(n)

Hard Problems

1. **Minimum Window Substring**
    - Find smallest substring containing all characters from target
    - Teaches hash table with sliding window
    - Time: O(n), Space: O(k) where k is alphabet size

Implementation Strategy (3-Week Plan)

Week 1: Fundamentals

- Learn hash function concepts and collision handling
- Implement a basic hash map from scratch
- Solve: Two Sum, Contains Duplicate
- Practice: 3-4 easy hash table problems

Week 2: Intermediate Applications

- Master frequency counting pattern
- Learn to combine hash tables with other data structures
- Solve: Group Anagrams, Valid Sudoku
- Practice: 4-5 medium problems

Week 3: Advanced Applications

- Implement LRU Cache (combining hash table and linked list)
- Solve complex problems using hash table optimizations
- Solve: Longest Consecutive Sequence, attempt Minimum Window Substring
- Practice: 3-4 hard problems that use hash tables as part of the solution

80/20 Deep Dive: Linked Lists

Key Concepts (20% that gives you 80% results)

1. **Node Structure**
    - Singly linked vs. doubly linked
    - Memory allocation and pointers
    - Head, tail, and sentinel nodes
2. **Basic Operations**
    - Traversal: linear time access
    - Insertion and deletion: constant time (with proper reference)
    - Finding, reversing, and merging
3. **Common Patterns**
    - Runner (fast/slow pointers)
    - Reversing segments
    - Detecting cycles
    - Recursive vs. iterative approaches

Problem-Solving Techniques

1. Fast and Slow Pointers

- **When to use**: Finding middle elements, detecting cycles, finding intersections
- **Implementation**: One pointer moves twice as fast as the other
- **Time complexity**: O(n)
- **Space complexity**: O(1)

2. Reversing Technique

- **When to use**: Reversing entire list or segments, palindrome verification
- **Implementation**: Carefully manipulate next pointers while traversing
- **Time complexity**: O(n)
- **Space complexity**: O(1) for iterative approach

3. Dummy Head Node

- **When to use**: When head may change or to simplify edge cases
- **Implementation**: Create a dummy node pointing to head
- **Benefits**: Eliminates special cases for empty lists or head modifications

Must-Know Problems

Easy Problems

1. **Reverse Linked List**
    - Reverse a singly linked list iteratively and recursively
    - Fundamental technique for many other problems
    - Time: O(n), Space: O(1) iterative, O(n) recursive
2. **Middle of the Linked List**
    - Find the middle node using fast/slow pointers
    - Teaches runner technique
    - Time: O(n), Space: O(1)

Medium Problems

1. **Remove Nth Node From End**
    - Remove node at specified position from end
    - Teaches two-pointer technique with offset
    - Time: O(n), Space: O(1)
2. **Detect Cycle**
    - Determine if linked list has a cycle
    - Teaches Floyd's Tortoise and Hare algorithm
    - Time: O(n), Space: O(1)
3. **Intersection of Two Linked Lists**
    - Find the node where two lists intersect
    - Teaches pointer manipulation and list properties
    - Time: O(n+m), Space: O(1)

Hard Problems

1. **Merge K Sorted Lists**
    - Merge k sorted linked lists into one sorted list
    - Teaches priority queue usage with lists
    - Time: O(n log k), Space: O(k)
2. **Reverse Nodes in k-Group**
    - Reverse list nodes in groups of k while maintaining overall structure
    - Teaches advanced reversal and segment manipulation

- Time: O(n), Space: O(1)

Implementation Strategy (3-Week Plan)

Week 1: Basics

- Implement singly and doubly linked list from scratch
- Master traversal and basic operations
- Solve: Reverse Linked List, Middle of Linked List
- Practice: 3-4 easy linked list manipulations

Week 2: Intermediate Techniques

- Master fast/slow pointer technique
- Learn cycle detection
- Solve: Detect Cycle, Intersection of Two Linked Lists
- Practice: 4-5 medium problems focusing on pointer manipulation

Week 3: Advanced Manipulations

- Implement complex operations (reversal in groups, merging)
- Solve: Merge K Sorted Lists, attempt Reverse Nodes in k-Group
- Practice: 2-3 hard problems that combine multiple techniques
- Review all patterns and special cases

80/20 Deep Dive: Stacks and Queues

Key Concepts (20% that gives you 80% results)

1. **Stack Fundamentals**
   - LIFO (Last-In-First-Out) principle
   - Push, pop, peek, and isEmpty operations
   - Implementation using arrays vs. linked lists
   - Applications: function call stack, expression evaluation, backtracking
2. **Queue Fundamentals**
   - FIFO (First-In-First-Out) principle
   - Enqueue, dequeue, peek, and isEmpty operations
   - Implementation using arrays (circular) vs. linked lists
   - Applications: BFS, scheduling, buffering
3. **Variations**
   - Deque (double-ended queue)
   - Priority Queue
   - Monotonic Stack/Queue

Problem-Solving Techniques

1. Parentheses Matching

- **When to use**: Validating expressions with brackets, parentheses
- **Implementation**: Push opening brackets, pop and match on closing
- **Time complexity**: O(n)
- **Space complexity**: O(n)

2. Monotonic Stack

- **When to use**: Finding next greater/smaller element, histogram areas
- **Implementation**: Maintain stack in increasing/decreasing order
- **Time complexity**: O(n) (each element pushed/popped at most once)
- **Space complexity**: O(n)

3. BFS with Queue

- **When to use**: Level-order traversals, shortest path in unweighted graphs
- **Implementation**: Process nodes level by level using a queue
- **Time complexity**: O(V + E) for graphs, O(n) for trees
- **Space complexity**: O(V) or O(width) for maximum queue size

Must-Know Problems

Easy Problems

1. **Valid Parentheses**
   - Check if string has valid bracket pairings
   - Teaches basic stack usage
   - Time: O(n), Space: O(n)
2. **Implement Stack using Queues**
   - Implement stack operations using only queue operations
   - Teaches understanding of data structure properties
   - Push: O(n) or O(1), Pop: O(1) or O(n) depending on approach

Medium Problems

1. **Daily Temperatures**
   - Find days until warmer temperature
   - Teaches monotonic stack pattern
   - Time: O(n), Space: O(n)
2. **Implement Queue using Stacks**

    - Implement queue operations using only stack operations
    - Teaches amortized analysis
    - Enqueue: O(1), Dequeue: Amortized O(1)
3. **Min Stack**
    - Design stack that supports push, pop, top, and retrieving minimum
    - Teaches stack with auxiliary data
    - All operations: O(1)

Hard Problems

1. **Largest Rectangle in Histogram**
    - Find largest rectangular area in histogram
    - Teaches advanced monotonic stack usage
    - Time: O(n), Space: O(n)
2. **Sliding Window Maximum**
    - Find maximum in each sliding window
    - Teaches monotonic queue (deque) usage
    - Time: O(n), Space: O(k)

Implementation Strategy (2-Week Plan)

Week 1: Stack Mastery

- Implement stack using array and linked list
- Learn expression evaluation techniques
- Solve: Valid Parentheses, Min Stack
- Practice: 3-4 stack-based problems
- Master monotonic stack pattern with 2-3 practice problems

Week 2: Queue and Combined Applications

- Implement queue (regular and circular)
- Understand BFS applications
- Solve: Implement Queue using Stacks, Sliding Window Maximum
- Practice: 2-3 queue-based problems
- Attempt Largest Rectangle in Histogram
- Review both data structures and their applications

Signs of Mastery

For each data structure, you should be able to:

1. Choose the appropriate data structure for a given problem instantly
2. Implement the basic operations without reference
3. Recognize common patterns where these structures excel
4. Handle edge cases (empty structures, overflows, etc.)
5. Analyze time and space complexity for your implementations

80/20 Deep Dive: Trees

Key Concepts (20% that gives you 80% results)

1. **Tree Structure**
    - Nodes, edges, root, leaves, and height
    - Binary trees vs. n-ary trees
    - Binary Search Trees (BST) properties
    - Balanced vs. unbalanced trees
    - Complete, full, and perfect trees
2. **Tree Traversals**
    - Depth-First: Preorder, Inorder, Postorder
    - Breadth-First: Level Order
    - Time complexity: O(n)
    - Space complexity: O(h) for DFS, O(w) for BFS (h = height, w = max width)
3. **Common Operations**
    - Insertion and deletion in BST
    - Search in BST
    - Tree balancing concepts (AVL, Red-Black tree principles)
    - Lowest Common Ancestor (LCA)

Problem-Solving Techniques

1. Recursive DFS

- **When to use**: Tree traversal, validation, path finding
- **Implementation**: Base case + recursive calls on left/right children
- **Time complexity**: O(n)

- **Space complexity**: O(h) where h is tree height (could be O(n) worst case)

2. Iterative BFS

- **When to use**: Level-based operations, shortest path in trees
- **Implementation**: Queue to process nodes level by level
- **Time complexity**: O(n)
- **Space complexity**: O(w) where w is maximum width of the tree

3. Global State vs. Return Values

- **When to use**: Accumulating results vs. computing and returning
- **Tradeoffs**: Clean code vs. potential need for class variables
- **Applications**: Path sums, tree properties

Must-Know Problems

Easy Problems

1. **Maximum Depth of Binary Tree**
    - Find the deepest level of the tree
    - Teaches basic recursive DFS
    - Time: O(n), Space: O(h)
2. **Symmetric Tree**
    - Check if tree is a mirror of itself
    - Teaches recursive comparison technique
    - Time: O(n), Space: O(h)

Medium Problems

1. **Validate Binary Search Tree**
    - Check if tree satisfies BST properties
    - Teaches range validation technique
    - Time: O(n), Space: O(h)
2. **Binary Tree Level Order Traversal**
    - Return nodes grouped by level
    - Teaches queue-based BFS
    - Time: O(n), Space: O(n)
3. **Lowest Common Ancestor of a Binary Tree**
    - Find the lowest common ancestor of two nodes
    - Teaches bottom-up recursion technique
    - Time: O(n), Space: O(h)

Hard Problems

1. **Binary Tree Maximum Path Sum**
    - Find path with maximum sum between any two nodes
    - Teaches complex recursive state tracking
    - Time: O(n), Space: O(h)
2. **Serialize and Deserialize Binary Tree**
    - Convert tree to string and back
    - Teaches encoding/decoding techniques
    - Time: O(n), Space: O(n)

Implementation Strategy (3-Week Plan)

Week 1: Fundamentals

- Implement binary tree and BST
- Master all traversal methods (recursive and iterative)
- Solve: Maximum Depth of Binary Tree, Symmetric Tree
- Practice: 3-4 easy tree traversal problems

Week 2: BST and Common Patterns

- Implement BST operations (insert, delete, search)
- Understand tree balancing concepts
- Solve: Validate BST, Level Order Traversal
- Practice: 4-5 medium tree problems

Week 3: Advanced Tree Problems

- Focus on complex recursion and state tracking
- Solve: LCA, Binary Tree Maximum Path Sum
- Attempt: Serialize and Deserialize Binary Tree
- Practice: 2-3 hard tree problems
- Review all tree patterns and optimizations

80/20 Deep Dive: Graphs

Key Concepts (20% that gives you 80% results)

1. **Graph Representation**
    - Adjacency matrix vs. adjacency list
    - Directed vs. undirected
    - Weighted vs. unweighted
    - Connected components
    - Sparse vs. dense graphs
2. **Graph Traversals**
    - Depth-First Search (DFS)
    - Breadth-First Search (BFS)
    - Time complexity: O(V + E)
    - Space complexity: O(V)

3. **Key Algorithms**
    - Topological Sort
    - Shortest Path (BFS for unweighted, Dijkstra's for weighted)
    - Minimum Spanning Tree concept
    - Cycle detection

Problem-Solving Techniques

1. Graph Coloring/Marking

- **When to use**: Visited node tracking, bipartite checking, cycle detection
- **Implementation**: Use boolean array, set, or node attributes
- **Time complexity**: O(V + E)
- **Space complexity**: O(V)

2. BFS for Shortest Paths

- **When to use**: Finding shortest path in unweighted graphs
- **Implementation**: Queue-based traversal with distance tracking
- **Time complexity**: O(V + E)
- **Space complexity**: O(V)

3. DFS for Exploration

- **When to use**: Finding all paths, cycle detection, connected components
- **Implementation**: Recursive or stack-based traversal
- **Time complexity**: O(V + E)
- **Space complexity**: O(V)

Must-Know Problems

Easy/Medium Problems

1. **Number of Islands**
    - Count connected land cells in a grid
    - Teaches DFS/BFS on 2D grid
    - Time: O(m * n), Space: O(m * n)
2. **Course Schedule (Topological Sort)**
    - Determine if courses can be finished given prerequisites
    - Teaches cycle detection and topological ordering
    - Time: O(V + E), Space: O(V + E)

Medium Problems

1. **Clone Graph**
    - Deep copy a connected graph
    - Teaches graph traversal with mapping
    - Time: O(V + E), Space: O(V)
2. **Pacific Atlantic Water Flow**
    - Find cells where water can flow to both oceans
    - Teaches multi-source BFS/DFS
    - Time: O(m * n), Space: O(m * n)

Hard Problems

1. **Word Ladder**
    - Find shortest transformation sequence
    - Teaches BFS on implicit graph
    - Time: O(n * m^2) where n is the number of words, m is word length
    - Space: O(n * m)
2. **Alien Dictionary**
    - Derive lexicographical order from sorted words
    - Teaches graph construction and topological sort
    - Time: O(C) where C is total length of all words
    - Space: O(1) or O(U) where U is unique letters

Implementation Strategy (3-Week Plan)

Week 1: Graph Basics

- Implement graph using adjacency list
- Master DFS and BFS implementations
- Solve: Number of Islands
- Practice: 2-3 grid-based DFS/BFS problems

Week 2: Graph Algorithms

- Implement topological sort
- Learn cycle detection
- Solve: Course Schedule, Clone Graph
- Practice: 3-4 medium graph traversal problems

Week 3: Advanced Graph Problems

- Focus on complex graph construction
- Solve: Pacific Atlantic Water Flow
- Attempt: Word Ladder, Alien Dictionary
- Practice: 2-3 hard graph problems
- Review all graph patterns and optimizations

80/20 Deep Dive: Heaps (Priority Queues)

Key Concepts (20% that gives you 80% results)

1. **Heap Properties**
    - Complete binary tree structure
    - Min-heap vs. max-heap property
    - Binary heap implementation using arrays
    - Zero-based vs. one-based indexing
2. **Basic Operations**
    - Insertion: O(log n)
    - Extraction (min/max): O(log n)
    - Peek (min/max): O(1)
    - Heapify: O(n)
3. **Applications**
    - Priority scheduling
    - Top-K problems
    - Heap sort
    - Graph algorithms (Dijkstra's)

Problem-Solving Techniques

1. K-Element Operations

- **When to use**: Finding top/smallest K elements, median finding
- **Implementation**: Maintain heap of size K
- **Time complexity**: O(n log k)
- **Space complexity**: O(k)

2. Heap as Implicit Data Structure

- **When to use**: When you need continual access to min/max while processing data
- **Implementation**: Use language's built-in implementation (e.g., PriorityQueue in Java, heapq in Python)
- **Benefits**: Simplifies code while maintaining performance

3. Dual-Heap Technique

- **When to use**: Finding median in streaming data, balancing elements
- **Implementation**: Maintain a min-heap and max-heap
- **Applications**: Median finding, interval processing

Must-Know Problems

Easy/Medium Problems

1. **Kth Largest Element in an Array**
    - Find the kth largest element
    - Teaches min-heap usage for top-K
    - Time: O(n log k), Space: O(k)
2. **Merge K Sorted Lists**
    - Merge k sorted linked lists
    - Teaches heap for multi-way merging
    - Time: O(n log k), Space: O(k)

Medium Problems

1. **Top K Frequent Elements**
    - Find k most frequent elements
    - Teaches counting + heap usage
    - Time: O(n log k), Space: O(n)
2. **Find Median from Data Stream**
    - Design structure to find median of streaming data
    - Teaches dual-heap technique
    - Operations: O(log n) insertion, O(1) find median

Hard Problems

1. **Sliding Window Median**
    - Find median in each sliding window
    - Teaches dynamic dual-heap maintenance
    - Time: O(n log k), Space: O(k)
2. **The Skyline Problem**
    - Compute skyline from building heights
    - Teaches heap with sweepline algorithm
    - Time: O(n log n), Space: O(n)

Implementation Strategy (2-Week Plan)

Week 1: Heap Basics

- Implement min-heap and max-heap from scratch
- Master insertion, deletion, and heapify
- Solve: Kth Largest Element, Top K Frequent Elements
- Practice: 2-3 basic heap usage problems

Week 2: Advanced Applications

- Learn dual-heap technique
- Understand heap usage in graph algorithms
- Solve: Find Median from Data Stream
- Attempt: Sliding Window Median
- Practice: 2-3 hard heap problems
- Review all heap patterns and optimization techniques

Signs of Mastery for All Data Structures:

1. Identify the right data structure for a given problem instantly
2. Implement common operations without hesitation
3. Analyze time and space complexity accurately
4. Choose optimal traversal/access methods for specific scenarios
5. Handle edge cases gracefully
6. Combine multiple data structures when needed

Final 80/20 Tips for Interview Success

1. **Focus on patterns, not problems**
    - The same 20% of patterns solve 80% of problems
    - Practice recognizing these patterns quickly
2. **Optimize your practice**
    - Revisit and redo problems to internalize patterns
    - Focus more time on your weak areas
    - Simulate real interview conditions
3. **Communication is key**
    - Practice explaining your thought process
    - Learn to break down complex problems step by step
    - Be able to analyze tradeoffs between different approaches
4. **Build a problem-solving framework**
    - Develop a consistent approach to tackling new problems
    - Learn to identify constraints and edge cases quickly
    - Practice working through examples by hand before coding

Fundamental Concepts

- Algorithmic Complexity
- Big-O Cheat Sheet
- Bit Manipulation Techniques
- Sorting Algorithms
- Linked List
    - Dummy Node Technique
- Queues
- Stacks
- Hash Tables
- Heaps
- Recursion
- Backtracking
- Trees
- Tries
- Binary Search
- Greedy Algorithm
- Dynamic Programming
- Graph Theory
- Important Graph Algorithms
- DFS Traversal
- BFS Traversal
- Union-Find
- Dijkstra Algorithm
- Minimum Spanning Tree

Patterns

- 15 Leetcode Patterns
- 20 DP Patterns
- Two Pointers Pattern
- Sliding Window Pattern
- Prefix Sum Pattern
- Fast and Slow Pointers Pattern
- Top 'K' Elements Pattern
- Kadane's Algorithm
- Sliding Window Pattern

- Linked List In-place Reversal Pattern
- Monotonic Stack Pattern
- Overlapping Intervals Pattern
- Backtracking Pattern
- Modified Binary Search Pattern
- Tree Patterns
    - Tree Iterative Traversal
    - Tree Question Pattern
- Graph Patterns
- DFS + BFS Patterns (1)
- DFS + BFS Patterns (2)

Comprehensive Problem-Solving Techniques for Technical Interviews

Fundamental Techniques Brute Force

- **Approach**: Try all possible solutions exhaustively
- **When to use**: When problem size is small or as a first step to understand constraints
- **Complexity**: Often O(n!) or O(2^n)
- **Example problems**: String permutations, subset generation

Optimiation Techniques Divide and Conquer

- **Approach**: Break problem into smaller subproblems, solve independently, and combine results
- **When to use**: Problems that can be naturally divided into similar subproblems
- **Complexity**: Often O(n log n)
- **Example problems**: Merge Sort, Quick Sort, Closest Pair of Points

Greedy Algorithms

- **Approach**: Make locally optimal choice at each step
- **When to use**: When local optimality leads to global optimality
- **Complexity**: Usually O(n log n) or better
- **Example problems**: Huffman coding, Dijkstra's algorithm, Activity Selection

Dynamic Programming

- **Approach**: Solve overlapping subproblems and store results to avoid redundant computation
- **When to use**: Problems with optimal substructure and overlapping subproblems
- **Complexity**: Typically $O(n^2)$ to $O(n^3)$
- **Variants**:
  - Top-down (memoization)
  - Bottom-up (tabulation)
- **Example problems**: Knapsack, Longest Common Subsequence, Edit Distance

Data Structure-Specific Techniques Array and String Manipulation Two Pointers

- **Approach**: Use two pointers to traverse data, often moving in tandem or opposite directions
- **When to use**: Searching pairs, partitioning, merging sorted arrays

- **Complexity**: Usually O(n)
- **Example problems**: Two Sum, Remove Duplicates, Container With Most Water

Sliding Window

- **Approach**: Maintain a window of elements and slide it through data
- **When to use**: Substring problems, finding subarrays with certain properties
- **Complexity**: Usually O(n)
- **Variants**:
    - Fixed-size window
    - Variable-size window
- **Example problems**: Maximum Sum Subarray of Size K, Longest Substring Without Repeating Characters

Prefix Sums

- **Approach**: Precompute cumulative sums to enable O(1) range queries
- **When to use**: Range sum queries, finding subarrays with target sum
- **Complexity**: O(n) preprocessing, O(1) per query
- **Example problems**: Subarray Sum Equals K, Range Sum Query

Tree and Graph Algorithms Depth-First Search (DFS)

- **Approach**: Explore as far as possible along each branch before backtracking
- **When to use**: Path finding, cycle detection, topological sorting
- **Complexity**: O(V + E)
- **Implementation**: Recursive or stack-based
- **Example problems**: Path exists between nodes, Cycle detection

Breadth-First Search (BFS)

- **Approach**: Explore all neighbors at current depth before moving to next depth
- **When to use**: Shortest path (unweighted), level order traversal
- **Complexity**: O(V + E)
- **Implementation**: Queue-based
- **Example problems**: Shortest path, Connected components

Union-Find (Disjoint Set)

- **Approach**: Track connected components with efficient union and find operations
- **When to use**: Dynamic connectivity, cycle detection in undirected graphs
- **Complexity**: Nearly O(1) per operation with path compression and union by rank
- **Example problems**: Kruskal's algorithm, Redundant Connection

Advanced Techniques Backtracking

- **Approach**: Build solutions incrementally and abandon paths that fail constraints
- **When to use**: Combinatorial problems, constraint satisfaction
- **Complexity**: Often exponential, but prunes search space
- **Example problems**: N-Queens, Sudoku Solver, Word Search

Binary Search

- **Approach**: Repeatedly divide search interval in half
- **When to use**: Finding elements in sorted arrays or when search space can be monotonically partitioned
- **Complexity**: O(log n)
- **Example problems**: Search in Rotated Sorted Array, Find First and Last Position

Monotonic Stack/Queue

- **Approach**: Maintain elements in increasing/decreasing order
- **When to use**: Next greater/smaller element problems, histogram areas
- **Complexity**: O(n) (each element pushed/popped once)
- **Example problems**: Daily Temperatures, Largest Rectangle in Histogram

Bit Manipulation

- **Approach**: Leverage bitwise operations for optimization
- **When to use**: Space optimization, certain math problems
- **Complexity**: Usually O(1) or O(log n)
- **Example problems**: Single Number, Counting Bits

Hybrid/Combined Techniques Graph + Dynamic Programming

- **Approach**: Apply DP principles to graph traversal

- **When to use**: Shortest paths with constraints, counting paths
- **Example problems**: Minimum Path Sum, Unique Paths

BFS/DFS + State Representation

- **Approach**: Define state space and explore using search algorithms
- **When to use**: Complex state transitions, puzzles
- **Example problems**: Word Ladder, Sliding Puzzle

Sliding Window + Hash Map

- **Approach**: Track window elements with a hash map for efficient lookups
- **When to use**: Substring problems with complex constraints
- **Example problems**: Longest Substring with K Distinct Characters

Two Heaps

- **Approach**: Maintain two heaps (typically min and max) to track median or partitioning
- **When to use**: Stream processing, median finding
- **Example problems**: Find Median from Data Stream

Advanced Problem-Solving Strategies Pattern Recognition

- **Approach**: Identify common patterns in problems to apply known solutions
- **When to use**: All problems - develops with experience
- **Example patterns**: Runner technique, merge intervals, topological sort

Space-Time Tradeoff

- **Approach**: Use additional space to reduce time complexity
- **When to use**: When memory constraints allow
- **Example**: Hash tables for O(1) lookup

Amortized Analysis

- **Approach**: Average operation cost over a sequence of operations
- **When to use**: When individual operations vary in cost but pattern is predictable
- **Example**: Dynamic array resizing, Sliding window

Simulation

- **Approach**: Simulate the process described in the problem

- **When to use**: When direct mathematical solution isn't obvious
- **Example problems**: Game of Life, Robot movement

Mathematical Insights

- **Approach**: Leverage mathematical properties to simplify solution
- **When to use**: Problems with mathematical patterns
- **Example**: Geometric problems, number theory

Effective Problem-Solving Process

1. **Understand the problem completely**
    - Clarify inputs, outputs, constraints
    - Work through examples manually
2. **Develop a brute force solution first**
    - Establish correctness before optimization
    - Identify bottlenecks
3. **Optimize incrementally**
    - Apply appropriate technique from above
    - Consider time and space complexity tradeoffs
4. **Test with edge cases**
    - Empty inputs, single elements
    - Maximum/minimum values
    - Duplicates and special cases
5. **Refine and communicate**
    - Explain your approach clearly
    - Discuss complexity analysis
    - Consider further optimizations

📌 Data Structure Implementation Techniques

Category	Data Structure	Implementation Techniques
Linear DS	**Array**	- Fixed-size variables with index notation - Loop iterations for traversal (for/while loops) - Index variables for random access - Nested loops for multi-dimensional operations
	Linked List	- Node structures with data + pointer variables - Pointer manipulation for insertion/deletion - Traversal via while loops with pointer advancement - Head/tail pointer variables for quick access - Temporary pointer variables for operations
	Stack	- Array or linked list as underlying structure - Top/size variables to track elements - Push/pop with pointer/index manipulation - Overflow/underflow checks
	Queue	- Array (circular) or linked list - Front/rear pointer variables - Enqueue/dequeue via pointer manipulation - Size tracking variable - Modulo arithmetic for circular queue
	Deque	- Doubly linked list or dynamic array - Front/rear pointers for both ends - Bidirectional insertion/deletion - Size tracking with increment/decrement
Non-Linear DS	**Tree**	- Node structures with data + child pointers - Recursive traversal/operations - Optional parent pointers - Level tracking for BFS - Balance factors/height variables
	Graph	- Adjacency matrix (2D array) - Adjacency list (array of lists) - Edge list (pairs) - Visited flags/arrays - Queue/stack for BFS/DFS - Priority queue for weighted graphs
	Heap	- Array-based with index formulas - Heapify (recursive/iterative) - Swap operations for reordering - Size variable for heap boundary - Index calcs: 2i+1, 2i+2, (i-1)/2
Hashing & Advanced	**Hash Table**	- Array of buckets/slots - Hash function for key mapping - Collision handling (chaining, open addressing) - Load factor tracking - Rehashing loops - Probing sequences
	Bloom Filter	- Bit array - Multiple hash functions - Bitwise operations for set/check - Track false positive probability
	Skip List	- Multi-level linked lists - Probabilistic level assignment - Pointer arrays per level - Random number generation for balancing
	Disjoint Set (Union-Find)	- Parent array representation - Rank/size arrays - Path compression (rec/iterative) - Root finding loops
Specialized	**Segment Tree**	- Array-based with index calculations

Category	Data Structure	Implementation Techniques
		- Recursive build/query/update - Binary splitting for ranges - Lazy propagation variables
	Fenwick Tree (BIT)	- Array + bit manipulation - lowbit (i & -i) - Cumulative update/query - Index manipulation via bitwise ops
	Suffix Array	- Sorting & ranking arrays - LCP (Longest Common Prefix) array - Doubling algorithm - Nested loops for suffix comparison
	Trie	- Node structures with char maps/arrays - End-of-word flags - Char-by-char traversal - Prefix tracking vars - Nested loops for pattern matching
	LRU Cache	- Doubly linked list + hash map - Key-value pair structures - MRU tracking via pointers - Capacity checks - Get/Put with list reorg
Hybrid	**Indexed Priority Queue**	- Heap array + Position map - Double indexing - Position updates in modifications - Key-index mapping
	Range Tree	- Multi-dimensional tree - Nested BSTs - Fractional cascading - Range vars for queries
	Concurrent DS	- Atomic operations - Locks (mutex, semaphores) - Compare-and-swap ops - Thread-safe modifications

---

📌 Data Structures & Basic Operations (Simplified Pseudo-Code)

Data Structure	Basic Operations (Pseudo-code)
Array	**Init:** array = new Array(size) **Access:** value = array[i] **Traverse:** for i=0 to size-1: print(array[i])
Linked List	**Node:** Node(data) **Insert:** newNode.next = head; head = newNode **Traverse:** curr=head; while(curr) { print(curr.data); curr=curr.next }
Stack	**Push:** stack.append(x) **Pop:** if !empty: val=stack.pop() **Traverse:** for item in stack: print(item)
Queue	**Enqueue:** queue.append(x) **Dequeue:** if !empty: val=queue.pop(0) **Traverse:** for item in queue: print(item)
Deque	**Add Front:** deque.insert(0, x) **Add Rear:** deque.append(x) **Remove Front:** x=deque.pop(0) **Remove Rear:** x=deque.pop()
Binary Tree	**Insert:** if root=null: root=new TreeNode(x) **Inorder:** inorder(node): if node { inorder(node.left); print(node.val); inorder(node.right) }
Graph	**Add Edge:** graph[u].append(v) **Traverse:** for v in graph: for n in graph[v]: print(n)
Min Heap	**Insert:** heap.append(x)

Data Structure	Basic Operations (Pseudo-code)
	Bubble Up: while idx>0: p=(idx-1)//2; if heap[idx]<heap[p]: swap(...)
Hash Table	**Insert:** idx=hash(key) if !table[idx]: table[idx]=[] table[idx].append((k,v)) **Traverse:** for bucket in table: print(bucket)
Bloom Filter	**Add:** for h in hashFns: bits[h(x)] = 1 **Check:** for h in hashFns: if !bits[h(x)]: return false
Skip List	**Insert:** curr=head; for level=maxLevel→0: while curr.fwd[level] && curr.fwd[level].val<x: curr=curr.fwd[level]
Disjoint Set	**Find:** if parent[x]!=x: parent[x]=find(parent[x]) **Union:** rx=find(x); ry=find(y); if rx!=ry: parent[ry]=rx
Segment Tree	**Build:** if start==end: tree[node]=arr[start] else: mid=(s+e)//2; build(left); build(right)
Fenwick Tree	**Update:** while i<=n: bit[i]+=x; i+=i&-i **Query:** sum=0; while i>0: sum+=bit[i]; i-=i&-i
Suffix Array	**Sort Suffixes:** for i=0→n-1: suffixes[i]=s[i:] sort(suffixes)
Trie	**Insert:** curr=root; for c in word: if c not in curr: curr[c]=TrieNode(); curr=curr[c]
LRU Cache	**Get:** if k in cache: order.remove(k); order.append(k); return cache[k] **Put:** if len(cache)==cap: old=order.pop(0); del cache[old]; cache[k]=v; order.append(k)

Concepts related to Data Structures and Algorithms (DSA)

1. Mathematical Concepts

- **Basic Arithmetic**: Understanding of addition, subtraction, multiplication, and division.
- **Algebra**: Solving equations, working with variables, and understanding functions.
- **Combinatorics**: Counting techniques, permutations, combinations, and the principles of counting.
- **Probability**: Basic probability concepts, including independent and dependent events, and calculating probabilities of events.
- **Number Theory**: Prime numbers, greatest common divisor (GCD), least common multiple (LCM), and modular arithmetic.

2. Logical Reasoning

- **Pattern Recognition**: Identifying patterns in sequences or data sets.
- **Analytical Thinking**: Breaking down complex problems into simpler components.
- **Deductive Reasoning**: Drawing logical conclusions from given premises or facts.

3. Geometric Concepts

- **Coordinate Geometry**: Understanding points, lines, and shapes in a coordinate system.
- **Area and Volume Calculations**: Calculating the area and volume of geometric shapes.

4. Graph Theory

- **Basic Graph Concepts**: Understanding vertices, edges, paths, cycles, and connectivity.

- **Traversal Algorithms**: Depth-First Search (DFS) and Breadth-First Search (BFS).
- **Shortest Path Algorithms**: Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm.

5. Complexity Analysis

- **Time Complexity**: Understanding Big O notation, best-case, average-case, and worst-case scenarios.

- **Space Complexity**: Analyzing the amount of memory used by an algorithm.

6. Data Structure Operations

- **Array Manipulations**: Understanding indexing, searching, and sorting.
- **Linked List Operations**: Insertion, deletion, and traversal.
- **Stack and Queue Operations**: Push, pop, enqueue, and dequeue.
- **Tree Traversals**: Inorder, preorder, and postorder traversals.

7. Recursion and Backtracking

- **Recursive Thinking**: Understanding how to break problems into smaller subproblems.
- **Backtracking Techniques**: Solving problems by exploring all possible solutions and abandoning paths that fail to meet criteria.

8. Dynamic Programming

- **Optimal Substructure**: Recognizing problems that can be broken down into overlapping subproblems.
- **Memoization and Tabulation**: Techniques for optimizing recursive solutions.

9. Bit Manipulation

- **Understanding Bits**: Operations like AND, OR, XOR, NOT, and bit shifts.
- **Applications**: Using bit manipulation for optimization and solving specific problems.

10. String Manipulation

- **String Algorithms**: Understanding substring search algorithms (e.g., KMP, Rabin-Karp).
- **Pattern Matching**: Techniques for finding patterns within strings.

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Expanded Data Structures Catalog

Data Structure Variants and Hybrids

Base Data Structure	Variants	Hybrid Structures
Array	Sparse Array, Jagged Array, Dynamic Array, Matrix	Array of Linked Lists, Array-Based Stack, Array-Based Queue
Linked List	Singly, Doubly, Circular, Skip List, XOR Linked List, Unrolled Linked List	LinkedList-based Hashtable, LinkedList-based Queue, LRU Cache
Stack	Min Stack, Max Stack, Double-ended Stack	Stack of Queues, Array-Based Stack, List-Based Stack
Queue	Priority Queue, Circular Queue, Double-ended Queue (Deque)	Queue of Stacks, Array-Based Queue, List-Based Queue
Tree	Binary Tree, N-ary Tree, Binary Search Tree, AVL Tree, Red-Black Tree, B-Tree, B+ Tree, 2-3 Tree, 2-4 Tree, Splay Tree, Treap	TreeMap (Tree+HashMap), Trie-BST Hybrid, Interval Tree
Heap	Binary Heap, Binomial Heap, Fibonacci Heap, Leftist Heap, Skew Heap	Heapified BST, Heap-ordered Tree
Graph	Directed, Undirected, Weighted, Unweighted, DAG, Multigraph, Hypergraph, Bipartite Graph	Graph with Priority Queue (for Dijkstra's), Graph with Disjoint Sets (for MST)
Hash Table	Open Addressing, Separate Chaining, Perfect Hashing, Cuckoo Hashing, Robin Hood Hashing	LinkedHashMap (HashMap+LinkedList), HashSet with Bloom Filter
Trie	Compressed Trie (Radix Tree), Suffix Trie, Ternary Search Trie, PATRICIA Trie	Trie-BST Hybrid, Hash-Array Mapped Trie (HAMT)

Specialized Combined Data Structures

Combined Structure	Component Structures	Description	Applications
LRU Cache	HashMap + Doubly Linked List	Fast lookup with ordered access for cache replacement policy	Browser caching, DB query caching

Suffix Tree	Tree + Array	Tree structure with suffix array for string operations	Pattern matching, genomic research
Combined Structure	**Component Structures**	**Description**	**Applications**
Augmented BST	BST + Additional Data	BST with extra data at nodes to support more operations	Range queries, order statistics
Segment Tree with Lazy Propagation	Segment Tree + Queue	Segment tree optimized for range updates	Range queries with updates
Treap	BST + Heap	Tree with BST ordering and heap priority	Balancing without explicit rotations
Skip Graph	Skip List + Graph	Distributed data structure combining skip list efficiency with graph connectivity	P2P networks, distributed systems
Indexed Priority Queue	Priority Queue + Hash Map	Priority queue with fast element modification	Graph algorithms (Dijkstra's)
Bloom Filter with Counting	Bloom Filter + Counter Array	Bloom filter that supports deletions	Streaming algorithms, databases
Disjoint Set with Path Compression	Union-Find + Tree	Optimized union-find structure with flattened trees	Dynamic connectivity, Kruskal's algorithm
Hash Array Mapped Trie (HAMT)	Trie + Hash Table	Compressed trie using hash functions	Immutable collections, functional data structures
R-Tree	Tree + Spatial Indexing	Tree structure for spatial data	Geographic information systems, spatial databases
Persistent Data Structures	Various + Version History	Data structures that preserve previous versions	Functional programming, undo functionality
Concurrent Skip List	Skip List + Locks/CAS	Thread-safe skip list implementation	Concurrent sorted sets, parallel processing
Rope	Binary Tree + String/Array	Tree-based data structure for string operations	Text editors, large string manipulation

Sparse Matrix Representations	Array + Hash Table/List	Efficient storage of matrices with many zeros	Scientific computing, large-scale simulations
QuadTree/OctTree	Tree + Spatial Partitioning	Hierarchical spatial partitioning structure	Computer graphics, collision detection
Combined Structure	**Component Structures**	**Description**	**Applications**
Dancing Links	Circular Linked Lists + Matrix	Specialized structure for backtracking algorithms	Exact cover problems, Sudoku solving
Wavelet Tree	Tree + Bit Vector	Tree structure for array operations	Range queries on sequences, text indexing
Concurrent Hash Map	Hash Table + Concurrency Control	Thread-safe hash map with lock striping	Parallel computing, web servers
B+ Tree	B-Tree + Linked List	B-tree with linked leaves for range queries	Database indexing, file systems

Problem-Solving Techniques for Technical Interviews

Category	Technique	Core Approach	When to Use	Time Complexity	Space Complexity	Example Problems
Fundamental	Brute Force	Try all possible solutions exhaustively	Small inputs, understand problem	O(n!) or O(2^n)	O(n)	Permutations, Combinations
Optimization	Divide and Conquer	Break into smaller subproblems, solve independently	Natural partitioning exists	O(n log n)	O(log n) to O(n)	Merge Sort, Quick Sort
Optimization	Greedy	Make locally optimal choice at each step	Local optimality leads to global optimality	O(n log n) or better	O(1) to O(n)	Huffman coding, Dijkstra's

Optimization	Dynamic Programming	Solve and store overlapping subproblems	Optimal substructure + overlapping subproblems	$O(n^2)$ to $O(n^3)$	O(n) to $O(n^2)$	Knapsack, Edit Distance
Array/String	Two Pointers	Use two indices to traverse data	Searching pairs, sorted array operations	O(n)	O(1)	Two Sum, Remove Duplicates
Category	**Technique**	**Core Approach**	**When to Use**	**Time Complexity**	**Space Complexity**	**Example Problems**
Array/String	Sliding Window	Maintain a window sliding through data	Substring/subarray problems	O(n)	O(1) to O(k)	Max Sum Subarray, Longest Substring
Array/String	Prefix Sums	Precompute cumulative sums	Range queries, target sum subarrays	O(n) preprocessing, O(1) queries	O(n)	Subarray Sum Equals K
Tree/Graph	DFS	Explore branches deeply before backtracking	Path finding, connectivity	O(V + E)	O(H) or O(V)	Path Exists, Cycle Detection
Tree/Graph	BFS	Explore by levels	Shortest path (unweighted), level order	O(V + E)	O(W) or O(V)	Shortest Path, Connected Components
Tree/Graph	Union-Find	Track connected components efficiently	Dynamic connectivity	Almost O(1) per operation	O(n)	Kruskal's Algorithm, Redundant Connection
Advanced	Backtracking	Build solutions incrementally, abandon invalid paths	Combinatorial problems	Often exponential	O(n) to $O(n^2)$	N-Queens, Sudoku Solver
Advanced	Binary Search	Repeatedly divide search interval	Sorted arrays, monotonic search spaces	O(log n)	O(1)	Search in Rotated Array

Advanced	Monotonic Stack/Queue	Maintain elements in increasing/decreasing order	Next greater/smaller element	O(n)	O(n)	Daily Temperatures, Largest Rectangle
Advanced	Bit Manipulation	Use bitwise operations	Space optimization, math problems	O(1) to O(log n)	O(1)	Single Number, Counting Bits

Category	**Technique**	**Core Approach**	**When to Use**	**Time Complexity**	**Space Complexity**	**Example Problems**
Hybrid	Graph + DP	Apply DP to graph traversal	Path problems with constraints	O(V + E) to O(V²)	O(V) to O(V²)	Minimum Path Sum
Hybrid	Sliding Window + Hash Map	Track window elements with hash map	Substring with complex constraints	O(n)	O(k)	Longest Substring with K Distinct Chars
Hybrid	Two Heaps	Maintain min and max heaps	Median finding, partitioning	O(log n) per operation	O(n)	Find Median from Data Stream
Strategy	Pattern Recognition	Identify common patterns	All problems	Varies	Varies	Depends on pattern
Strategy	Space-Time Tradeoff	Use space to optimize time	When memory allows	Reduced time	Increased space	Memoization
Strategy	Amortized Analysis	Average operation cost over sequence	Operations with varying costs	Varies	Varies	Dynamic array resizing
Strategy	Mathematical Insights	Use math properties to simplify	Problems with mathematical patterns	Often O(1) to O(log n)	O(1)	Number theory problems

Problem-Solving Process

1. **Understand** - Clarify inputs, outputs, constraints, examples
2. **Brute Force** - Establish correctness before optimization
3. **Optimize** - Apply appropriate technique from above

4. **Test** - Check edge cases and typical inputs
5. **Refine** - Improve solution and communication

H = height of tree, W = maximum width of tree/graph, V = vertices, E = edges, k = window size or distinct elements

Retry

SI No	Problem Name	Difficulty	Concept	Try1	Try2	Try3	…	
1	Merge Sorted Array	Easy	Array / String					
2	Remove Element	Easy	Array / String					
3	Remove Duplicates from Sorted Array	Easy	Array / String					
4	Remove Duplicates from Sorted Array II	Medium	Array / String					
5	Majority Element	Easy	Array / String					
6	Rotate Array	Medium	Array / String					
7	Best Time to Buy and Sell Stock	Easy	Array / String					
8	Best Time to Buy and Sell Stock II	Medium	Array / String					
9	Jump Game	Medium	Array / String					
10	Jump Game II	Medium	Array / String					
11	H-Index	Medium	Array / String					
12	Insert Delete GetRandom O(1)	Medium	Array / String					
13	Product of Array Except Self	Medium	Array / String					
14	Gas Station	Medium	Array / String					
15	Candy	Hard	Array / String					
16	Trapping Rain Water	Hard	Array / String					
17	Roman to Integer	Easy	Array / String					
18	Integer to Roman	Medium	Array / String					
19	Length of Last Word	Easy	Array / String					
20	Longest Common Prefix	Easy	Array / String					
21	Reverse Words in a String	Medium	Array / String					
22	Zigzag Conversion	Medium	Array / String					
23	Find the Index of the First Occurrence in a String	Easy	Array / String					
24	Text Justification	Hard	Array / String					
25	Valid Palindrome	Easy	Two Pointers					
26	Is Subsequence	Easy	Two Pointers					

SI No	Problem Name	Difficulty	Concept					
27	Two Sum II - Input Array Is Sorted	Medium	Two Pointers					
28	Container With Most Water	Medium	Two Pointers					
29	3Sum	Medium	Two Pointers					
30	Minimum Size Subarray Sum	Medium	Sliding Window					
31	Longest Substring Without Repeating Characters	Medium	Sliding Window					
32	Substring with Concatenation of All Words	Hard	Sliding Window					
33	Minimum Window Substring	Hard	Sliding Window					
34	Valid Sudoku	Medium	Matrix					
35	Spiral Matrix	Medium	Matrix					
36	Rotate Image	Medium	Matrix					
37	Set Matrix Zeroes	Medium	Matrix					
38	Game of Life	Medium	Matrix					
SI No	Problem Name	Difficulty	Concept					
1	Ransom Note	Easy	Hashmap					
2	Isomorphic Strings	Easy	Hashmap					
3	Word Pattern	Easy	Hashmap					
4	Valid Anagram	Easy	Hashmap					
5	Group Anagrams	Medium	Hashmap					
6	Two Sum	Easy	Hashmap					
7	Happy Number	Easy	Hashmap					
8	Contains Duplicate II	Easy	Hashmap					
9	Longest Consecutive Sequence	Medium	Hashmap					
10	Summary Ranges	Easy	Intervals					
11	Merge Intervals	Medium	Intervals					
12	Insert Interval	Medium	Intervals					

SI No	Problem Name	Difficulty	Concept					
13	Minimum Number of Arrows to Burst Balloons	Medium	Intervals					
14	Valid Parentheses	Easy	Stack					
15	Simplify Path	Medium	Stack					
16	Min Stack	Medium	Stack					
17	Evaluate Reverse Polish Notation	Medium	Stack					
18	Basic Calculator	Hard	Stack					
19	Linked List		Linked List					
20	Linked List Cycle	Easy	Linked List					
21	Add Two Numbers	Medium	Linked List					
22	Merge Two Sorted Lists	Easy	Linked List					
23	Copy List with Random Pointer	Medium	Linked List					
24	Reverse Linked List II	Medium	Linked List					
25	Reverse Nodes in k-Group	Hard	Linked List					
26	Remove Nth Node From End of List	Medium	Linked List					
27	Remove Duplicates from Sorted List II	Medium	Linked List					
28	Rotate List	Medium	Linked List					
29	Partition List	Medium	Linked List					
30	LRU Cache	Medium	Linked List					
31	Maximum Depth of Binary Tree	Easy	Binary Tree					
32	Same Tree	Easy	Binary Tree					
33	Invert Binary Tree	Easy	Binary Tree					
34	Symmetric Tree	Easy	Binary Tree					
35	Construct Binary Tree from Preorder and Inorder Traversal	Medium	Binary Tree					
36	Construct Binary Tree from Inorder and Postorder Traversal	Medium	Binary Tree					

SI No	Problem Name	Difficulty	Concept					
37	Populating Next Right Pointers in Each Node II	Medium	Binary Tree					
38	Flatten Binary Tree to Linked List	Medium	Binary Tree					
39	Path Sum	Easy	Binary Tree					
40	Sum Root to Leaf Numbers	Medium	Binary Tree					
41	Binary Tree Maximum Path Sum	Hard	Binary Tree					
42	Binary Search Tree Iterator	Medium	Binary Tree					
43	Count Complete Tree Nodes	Easy	Binary Tree					
44	Lowest Common Ancestor of a Binary Tree	Medium	Binary Tree					
45	Binary Tree Right Side View	Medium	Binary Tree					
46	Average of Levels in Binary Tree	Easy	Binary Tree					
47	Binary Tree Level Order Traversal	Medium	Binary Tree					
48	Binary Tree Zigzag Level Order Traversal	Medium	Binary Tree					
49	Minimum Absolute Difference in BST	Easy	Binary Search Tree					
50	Kth Smallest Element in a BST	Medium	Binary Search Tree					
51	Validate Binary Search Tree	Medium	Binary Search Tree					
52	Number of Islands	Medium	Graph					
53	Surrounded Regions	Medium	Graph					

SI No	Problem Name	Difficulty	Concept						
1	Snakes and Ladders	Medium	Graph BFS						
2	Minimum Genetic Mutation	Medium	Graph BFS						
3	Word Ladder	Hard	Graph BFS						
4	Implement Trie (Prefix Tree)	Medium	Trie						

SI No	Problem Name	Difficulty	Concept						
5	Design Add and Search Words Data Structure	Medium	Trie						
6	Word Search II	Hard	Trie						
7	Letter Combinations of a Phone Number	Medium	Backtracking						
8	Combinations	Medium	Backtracking						
9	Permutations	Medium	Backtracking						
10	Combination Sum	Medium	Backtracking						
11	N-Queens II	Hard	Backtracking						
12	Generate Parentheses	Medium	Backtracking						
13	Word Search	Medium	Backtracking						
14	Convert Sorted Array to Binary Search Tree	Easy	Divide & Conquer						
15	Sort List	Medium	Divide & Conquer						
16	Construct Quad Tree	Medium	Divide & Conquer						
17	Merge k Sorted Lists	Hard	Divide & Conquer						
18	Maximum Subarray	Medium	Kadane's Algorithm						
19	Maximum Sum Circular Subarray	Medium	Kadane's Algorithm						
20	Search Insert Position	Easy	Binary Search						
21	Search a 2D Matrix	Medium	Binary Search						
22	Find Peak Element	Medium	Binary Search						
23	Search in Rotated Sorted Array	Medium	Binary Search						
24	Find First and Last Position of Element in Sorted Array	Medium	Binary Search						
25	Find Minimum in Rotated Sorted Array	Medium	Binary Search						
26	Median of Two Sorted Arrays	Hard	Binary Search						
27	Kth Largest Element in an Array	Medium	Heap						
28	IPO	Hard	Heap						
29	Find K Pairs with Smallest Sums	Medium	Heap						
30	Find Median from Data Stream	Hard	Heap						

SI No	Problem Name	Difficulty	Concept						
31	Add Binary	Easy	Bit Manipulation						
32	Reverse Bits	Easy	Bit Manipulation						
33	Number of 1 Bits	Easy	Bit Manipulation						
34	Single Number	Easy	Bit Manipulation						
35	Single Number II	Medium	Bit Manipulation						
36	Bitwise AND of Numbers Range	Medium	Bit Manipulation						
37	Palindrome Number	Easy	Math						
38	Plus One	Easy	Math						
39	Factorial Trailing Zeroes	Medium	Math						
40	Sqrt(x)	Easy	Math						
41	Pow(x, n)	Medium	Math						
42	Max Points on a Line	Hard	Math						
43	Climbing Stairs	Easy	1D DP						
44	House Robber	Medium	1D DP						
45	Word Break	Medium	1D DP						
46	Coin Change	Medium	1D DP						
47	Longest Increasing Subsequence	Medium	1D DP						
48	Triangle	Medium	Multidimensional DP						
49	Minimum Path Sum	Medium	Multidimensional DP						
50	Unique Paths II								

S.No	Category	Problem
1	Array / String	Merge Strings Alternately
2	Array / String	Greatest Common Divisor of Strings
3	Array / String	Kids With the Greatest Number of Candies
4	Array / String	Can Place Flowers

S.No	Category	Problem
5	Array / String	Reverse Vowels of a String
6	Array / String	Reverse Words in a String
7	Array / String	Product of Array Except Self
8	Array / String	Increasing Triplet Subsequence
9	Array / String	String Compression
10	Two Pointers	Move Zeroes
11	Two Pointers	Is Subsequence
12	Two Pointers	Container With Most Water
13	Two Pointers	Max Number of K-Sum Pairs
14	Sliding Window	Maximum Average Subarray I
15	Sliding Window	Maximum Number of Vowels in a Substring of Given Length
16	Sliding Window	Max Consecutive Ones III
17	Sliding Window	Longest Subarray of 1's After Deleting One Element
18	Prefix Sum	Find the Highest Altitude
19	Prefix Sum	Find Pivot Index
20	Hash Map / Set	Find the Difference of Two Arrays
21	Hash Map / Set	Unique Number of Occurrences
22	Hash Map / Set	Determine if Two Strings Are Close
23	Hash Map / Set	Equal Row and Column Pairs
24	Stack	Removing Stars From a String
25	Stack	Asteroid Collision
26	Stack	Decode String
27	Queue	Number of Recent Calls
28	Queue	Dota2 Senate
29	Linked List	Delete the Middle Node of a Linked List
30	Linked List	Odd Even Linked List
31	Linked List	Reverse Linked List

S.No	Category	Problem
32	Linked List	Maximum Twin Sum of a Linked List
33	Binary Tree - DFS	Maximum Depth of Binary Tree
34	Binary Tree - DFS	Leaf-Similar Trees
35	Binary Tree - DFS	Count Good Nodes in Binary Tree
36	Binary Tree - DFS	Path Sum III
37	Binary Tree - DFS	Longest ZigZag Path in a Binary Tree
38	Binary Tree - DFS	Lowest Common Ancestor of a Binary Tree
39	Binary Tree - BFS	Binary Tree Right Side View
40	Binary Tree - BFS	Maximum Level Sum of a Binary Tree
41	Binary Search Tree	Search in a Binary Search Tree
42	Binary Search Tree	Delete Node in a BST
43	Graphs - DFS	Keys and Rooms
44	Graphs - DFS	Number of Provinces
45	Graphs - DFS	Reorder Routes to Make All Paths Lead to the City Zero
46	Graphs - DFS	Evaluate Division
47	Graphs - BFS	Nearest Exit from Entrance in Maze
48	Graphs - BFS	Rotting Oranges
49	Heap / Priority Queue	Kth Largest Element in an Array
50	Heap / Priority Queue	Smallest Number in Infinite Set
51	Heap / Priority Queue	Maximum Subsequence Score
52	Heap / Priority Queue	Total Cost to Hire K Workers
53	Binary Search	Guess Number Higher or Lower
54	Binary Search	Successful Pairs of Spells and Potions
55	Binary Search	Find Peak Element
56	Binary Search	Koko Eating Bananas
57	Backtracking	Letter Combinations of a Phone Number
58	Backtracking	Combination Sum III

S.No	Category	Problem
59	DP - 1D	N-th Tribonacci Number
60	DP - 1D	Min Cost Climbing Stairs
61	DP - 1D	House Robber
62	DP - 1D	Domino and Tromino Tiling
63	DP - Multidimensional	Unique Paths
64	DP - Multidimensional	Longest Common Subsequence
65	DP - Multidimensional	Best Time to Buy and Sell Stock with Transaction Fee
66	DP - Multidimensional	Edit Distance
67	Bit Manipulation	Counting Bits
68	Bit Manipulation	Single Number
69	Bit Manipulation	Minimum Flips to Make a OR b Equal to c
70	Trie	Implement Trie (Prefix Tree)
71	Trie	Search Suggestions System
72	Intervals	Non-overlapping Intervals
73	Intervals	Minimum Number of Arrows to Burst Balloons
74	Monotonic Stack	Daily Temperatures
75	Monotonic Stack	Online Stock Span

Techniques that we are going implement these 150 problems **General Problem-Solving Techniques:**

1. **Divide and Conquer** – Break the problem into smaller subproblems, solve them independently, and combine results. (Example: Merge Sort, Quick Sort)
2. **Brute Force** – Try all possible solutions and pick the best one. (Example: Checking all possible combinations in password cracking)
3. **Greedy Algorithm** – Make the best choice at each step to reach an optimal solution. (Example: Dijkstra's Algorithm for shortest paths)
4. **Backtracking** – Explore all possible solutions recursively and discard invalid ones. (Example: N-Queens Problem, Sudoku Solver)
5. **Dynamic Programming** – Solve overlapping subproblems and store results to avoid redundant computation. (Example: Fibonacci Series, Knapsack Problem)

6. **Recursion** – Solve a problem by solving smaller instances of the same problem. (Example: Factorial Calculation)
7. **Graph-Based Approach** – Represent problems as graphs and solve using traversal techniques. (Example: BFS and DFS for shortest paths)
8. **Heuristic Methods** – Approximate solutions for complex problems where exact solutions are infeasible. (Example: Genetic Algorithms)
9. **Pattern Recognition** – Identify patterns to simplify problem-solving. (Example: Detecting cycles in a graph)
10. **Constraint Satisfaction** – Solve problems with specific conditions. (Example: Scheduling problems, Cryptarithm puzzles)

Expanded Data Structures Catalog

Data Structure Variants and Hybrids

Base Data Structure	**Variants**	**Hybrid Structures**
Array	Sparse Array, Jagged Array, Dynamic Array, Matrix	Array of Linked Lists, Array-Based Stack, Array-Based Queue
Linked List	Singly, Doubly, Circular, Skip List, XOR Linked List, Unrolled Linked List	LinkedList-based Hashtable, LinkedList- based Queue, LRU Cache
Stack	Min Stack, Max Stack, Double-ended Stack	Stack of Queues, Array-Based Stack, List-Based Stack
Queue	Priority Queue, Circular Queue, Double-ended Queue (Deque)	Queue of Stacks, Array-Based Queue, List-Based Queue
Tree	Binary Tree, N-ary Tree, Binary Search Tree, AVL Tree, Red-Black Tree, B-Tree, B+ Tree, 2-3 Tree, 2-4 Tree, Splay Tree, Treap	TreeMap (Tree+HashMap), Trie-BST Hybrid, Interval Tree
Heap	Binary Heap, Binomial Heap, Fibonacci Heap, Leftist Heap, Skew Heap	Heapified BST, Heap-ordered Tree
Graph	Directed, Undirected, Weighted, Unweighted, DAG, Multigraph, Hypergraph, Bipartite Graph	Graph with Priority Queue (for Dijkstra's), Graph with Disjoint Sets (for MST)
Base Data Structure	**Variants**	**Hybrid Structures**

Hash Table	Open Addressing, Separate Chaining, Perfect Hashing, Cuckoo Hashing, Robin Hood Hashing	LinkedHashMap (HashMap+LinkedList), HashSet with Bloom Filter
Trie	Compressed Trie (Radix Tree), Suffix Trie, Ternary Search Trie, PATRICIA Trie	Trie-BST Hybrid, Hash-Array Mapped Trie (HAMT)

Specialized Combined Data Structures

Combined Structure	Component Structures	Description	Applications
LRU Cache	HashMap + Doubly Linked List	Fast lookup with ordered access for cache replacement policy	Browser caching, DB query caching
Suffix Tree	Tree + Array	Tree structure with suffix array for string operations	Pattern matching, genomic research
Augmented BST	BST + Additional Data	BST with extra data at nodes to support more operations	Range queries, order statistics
Segment Tree with Lazy Propagation	Segment Tree + Queue	Segment tree optimized for range updates	Range queries with updates
Treap	BST + Heap	Tree with BST ordering and heap priority	Balancing without explicit rotations
Skip Graph	Skip List + Graph	Distributed data structure combining skip list efficiency with graph connectivity	P2P networks, distributed systems
Indexed Priority Queue	Priority Queue + Hash Map	Priority queue with fast element modification	Graph algorithms (Dijkstra's)
Bloom Filter with Counting	Bloom Filter + Counter Array	Bloom filter that supports deletions	Streaming algorithms, databases
Disjoint Set with Path Compression	Union-Find + Tree	Optimized union-find structure with flattened trees	Dynamic connectivity, Kruskal's algorithm
Hash Array Mapped Trie (HAMT)	Trie + Hash Table	Compressed trie using hash functions	Immutable collections, functional data structures
R-Tree	Tree + Spatial Indexing	Tree structure for spatial data	Geographic information systems, spatial databases

Combined Structure	Component Structures	Description	Applications
Persistent Data Structures	Various + Version History	Data structures that preserve previous versions	Functional programming, undo functionality
Concurrent Skip List	Skip List + Locks/CAS	Thread-safe skip list implementation	Concurrent sorted sets, parallel processing
Rope	Binary Tree + String/Array	Tree-based data structure for string operations	Text editors, large string manipulation
Sparse Matrix Representations	Array + Hash Table/List	Efficient storage of matrices with many zeros	Scientific computing, large-scale simulations
QuadTree/OctTree	Tree + Spatial Partitioning	Hierarchical spatial partitioning structure	Computer graphics, collision detection
Dancing Links	Circular Linked Lists + Matrix	Specialized structure for backtracking algorithms	Exact cover problems, Sudoku solving
Wavelet Tree	Tree + Bit Vector	Tree structure for array operations	Range queries on sequences, text indexing
Concurrent Hash Map	Hash Table + Concurrency Control	Thread-safe hash map with lock striping	Parallel computing, web servers
B+ Tree	B-Tree + Linked List	B-tree with linked leaves for range queries	Database indexing, file systems

Problem-Solving Techniques for Technical Interviews

Category	Technique	Core Approach	When to Use	Time Complexity	Space Complexity	Example Problems
Fundamental	Brute Force	Try all possible solutions exhaustively	Small inputs, understand problem	O(n!) or O(2^n)	O(n)	Permutations, Combinations

Category	Technique	Core Approach	When to Use	Time Complexity	Space Complexity	Example Problems
Optimization	Divide and Conquer	Break into smaller subproblems, solve independently	Natural partitioning exists	O(n log n)	O(log n) to O(n)	Merge Sort, Quick Sort
Optimization	Greedy	Make locally optimal choice at each step	Local optimality leads to global optimality	O(n log n) or better	O(1) to O(n)	Huffman coding, Dijkstra's
Optimization	Dynamic Programming	Solve and store overlapping subproblems	Optimal substructure + overlapping subproblems	$O(n^2)$ to $O(n^3)$	O(n) to $O(n^2)$	Knapsack, Edit Distance
Array/String	Two Pointers	Use two indices to traverse data	Searching pairs, sorted array operations	O(n)	O(1)	Two Sum, Remove Duplicates
Array/String	Sliding Window	Maintain a window sliding through data	Substring/subarray problems	O(n)	O(1) to O(k)	Max Sum Subarray, Longest Substring
Array/String	Prefix Sums	Precompute cumulative sums	Range queries, target sum subarrays	O(n) preprocessing, O(1) queries	O(n)	Subarray Sum Equals K
Tree/Graph	DFS	Explore branches deeply before backtracking	Path finding, connectivity	O(V + E)	O(H) or O(V)	Path Exists, Cycle Detection
Tree/Graph	BFS	Explore by levels	Shortest path (unweighted), level order	O(V + E)	O(W) or O(V)	Shortest Path, Connected Components
Tree/Graph	Union-Find	Track connected components efficiently	Dynamic connectivity	Almost O(1) per operation	O(n)	Kruskal's Algorithm, Redundant Connection
Advanced	Backtracking	Build solutions incrementally,	Combinatorial problems	Often exponential	O(n) to $O(n^2)$	N-Queens, Sudoku Solver

Category	Technique	Core Approach	When to Use	Time Complexity	Space Complexity	Example Problems
		abandon invalid paths				
Advanced	Binary Search	Repeatedly divide search interval	Sorted arrays, monotonic search spaces	O(log n)	O(1)	Search in Rotated Array
Advanced	Monotonic Stack/Queue	Maintain elements in increasing/decreasing order	Next greater/smaller element	O(n)	O(n)	Daily Temperatures, Largest Rectangle
Advanced	Bit Manipulation	Use bitwise operations	Space optimization, math problems	O(1) to O(log n)	O(1)	Single Number, Counting Bits
Hybrid	Graph + DP	Apply DP to graph traversal	Path problems with constraints	O(V + E) to $O(V^2)$	O(V) to $O(V^2)$	Minimum Path Sum
Hybrid	Sliding Window + Hash Map	Track window elements with hash map	Substring with complex constraints	O(n)	O(k)	Longest Substring with K Distinct Chars
Hybrid	Two Heaps	Maintain min and max heaps	Median finding, partitioning	O(log n) per operation	O(n)	Find Median from Data Stream
Strategy	Pattern Recognition	Identify common patterns	All problems	Varies	Varies	Depends on pattern
Strategy	Space-Time Tradeoff	Use space to optimize time	When memory allows	Reduced time	Increased space	Memoization
Strategy	Amortized Analysis	Average operation cost over sequence	Operations with varying costs	Varies	Varies	Dynamic array resizing
Strategy	Mathematical Insights	Use math properties to simplify	Problems with mathematical patterns	Often O(1) to O(log n)	O(1)	Number theory problems

Problem-Solving Process

1. **Understand** - Clarify inputs, outputs, constraints, examples

2. **Brute Force** - Establish correctness before optimization
3. **Optimize** - Apply appropriate technique from above
4. **Test** - Check edge cases and typical inputs
5. **Refine** - Improve solution and communication

H = height of tree, W = maximum width of tree/graph, V = vertices, E = edges, k = window size or distinct elements.