

Core Data Structures

1. Arrays & ArrayLists

অ্যারের একটি নির্দিষ্ট আকার থাকে, যখন অ্যারেলিস্টগুলি গতিশীল। নমনীয়তার জন্য আপনাকে অ্যারেলিস্টগুলি আরও বেশি ব্যবহার করতে হবে।

- কখন ব্যবহার করবেন: যখন আপনাকে উপাদানগুলির একটি সংগ্রহ সংরক্ষণ করতে হবে এবং সূচক অনুসারে সেগুলি অ্যাক্সেস করতে হবে। যখন আপনি আগে থেকে সংগ্রহের আকার জানেন না তখন অ্যারেলিস্ট দুর্দান্ত।

```
// Array (fixed size)
int[] numbers = new int[5]; // An array that can hold 5 integers.
numbers[0] = 10;
int first = numbers[0];
System.out.println("Array length: " + numbers.length);

// ArrayList (dynamic size)
import java.util.ArrayList;
import java.util.List;

List<Integer> numberList = new ArrayList<>();
numberList.add(10); // Add an element
numberList.add(20);
numberList.remove(0); // Remove element at index 0
int value = numberList.get(0); // Get element at index 0
System.out.println("ArrayList size: " + numberList.size());
```

2. Hash Map (or Dictionary)

একটি হ্যাশম্যাপ কী-মান জোড়া সংরক্ষণ করে। এটি অবিশ্বাস্যভাবে দ্রুত লুকআপ, সন্নিবেশ এবং মুছে ফেলার সুবিধা প্রদান করে, সাধারণত গড়ে O(1) সময়ে।

কখন ব্যবহার করবেন: ফ্রিকোয়েল্সি গণনা, লুকআপ, অথবা একটি মানকে অন্য মান ম্যাপ করার সমস্যাগুলির জন্য উপযুক্ত (e.g., "Two Sum," "Valid Anagram")

```
import java.util.HashMap;
import java.util.Map;

// Maps a String key to an Integer value
Map<String, Integer> studentScores = new HashMap<>();

studentScores.put("Alice", 95);
studentScores.put("Bob", 88);

// Get a value by key
int alicesScore = studentScores.get("Alice");
System.out.println("Alice's score: " + alicesScore);

// Check if a key exists
if (studentScores.containsKey("Charlie")) {
    System.out.println("Charlie's score found.");
} else {
    System.out.println("Charlie's score not found.");
}

// Iterate over the map
for (Map.Entry<String, Integer> entry : studentScores.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

3. Linked List

একটি `LinkedList`-এ এমন নোড থাকে যেখানে প্রতিটি নোডে ডেটা থাকে এবং পরবর্তী নোডের দিকে একটি পয়েন্টার থাকে।

কখন ব্যবহার করবেন: ক্রমের মাঝখান থেকে ঘন ঘন সন্ধিবেশ বা মুছে ফেলার প্রয়োজন হয় এমন সমস্যার জন্য আদর্শ। অ্যারের তুলনায় এলোমেলো অ্যাক্সেসের জন্য কম দক্ষ।

```
// Definition for a singly-linked list node.
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

// Basic operations
// Creating a simple list: 1 -> 2 -> 3
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);

// Traversing the list
ListNode current = head;
while (current != null) {
    System.out.print(current.val + " -> ");
    current = current.next;
}
System.out.println("null");
```

4. Stack

একটি স্ট্যাক হল একটি লাস্ট-ইন, ফাস্ট-আউট (LIFO) ডেটা স্ট্রাকচার। এটিকে প্লেটের স্ট্যাক হিসাবে ভাবুন। 🍔

কখন ব্যবহার করবেন: বন্ধনী মেলানো, ক্রম বিপরীত করা, অথবা গভীরতা-প্রথম অনুসন্ধান (DFS) ট্র্যাভার্সাল সম্পর্কিত সমস্যা।

```
import java.util.Stack;

Stack<Integer> stack = new Stack<>();

// Push items onto the stack
stack.push(1);
stack.push(2);
stack.push(3); // Top of stack is now 3

// See the top item without removing it
int topItem = stack.peek(); // topItem is 3

// Remove and get the top item
int removedItem = stack.pop(); // removedItem is 3

System.out.println("Is stack empty? " + stack.isEmpty()); // false
```

5. Queue

সারি হলো একটি ফার্স্ট-ইন, ফার্স্ট-আউট (FIFO) ডেটা স্ট্রাকচার। এটিকে অপেক্ষারত মানুষের একটি লাইন হিসেবে ভাবুন। 

কখন ব্যবহার করবেন: ব্রেডথ-ফার্স্ট সার্চ (BFS) অ্যালগরিদম, প্রাপ্ত ক্রমানুসারে আইটেম প্রক্রিয়াকরণ, অথবা কাজ পরিচালনা।

```
import java.util.LinkedList;
import java.util.Queue;

// LinkedList is often used to implement the Queue interface
Queue<Integer> queue = new LinkedList<>();

// Add items to the end of the queue
queue.add(1);
queue.add(2);
queue.add(3);

// See the front item without removing it
int frontItem = queue.peek(); // frontItem is 1

// Remove and get the front item
int removedItem = queue.poll(); // removedItem is 1

System.out.println("Queue size: " + queue.size()); // 2
```

6. Binary Tree

একটি ডেটা স্ট্রাকচার যেখানে প্রতিটি নোডে সর্বাধিক দুটি শিল্প থাকে, যাকে বাম শিল্প এবং ডান শিল্প বলা হয়।

কখন ব্যবহার করবেন: হায়ারার্কিকাল ডেটা, অনুসন্ধান (বাইনারি অনুসন্ধান গাছের মতো), এবং সমস্যাগুলি যা ছোট, অনুরূপ উপ-সমস্যাগুলিতে বিভক্ত করা যেতে পারে।

```

// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

// In-order Traversal (Left, Root, Right) - very common!
public void inorderTraversal(TreeNode root) {
    if (root == null) {
        return;
    }
    inorderTraversal(root.left);
    System.out.print(root.val + " ");
    inorderTraversal(root.right);
}

```

Key Algorithms

এগুলি হল মৌলিক অ্যালগরিদম যা আপনার জানা আবশ্যিক।

1. Binary Search

একটি সাজানো অ্যারে থেকে একটি আইটেম খুঁজে বের করার জন্য একটি দক্ষ অ্যালগরিদম। এটি অনুসন্ধান ব্যবধানকে বারবার অর্ধেক ভাগ করে কাজ করে। এর সময়ের জটিলতা হল $O(\log n)$ ।

```

// Finds the index of 'target' in a sorted array 'nums'.
// Returns -1 if target is not found.
public int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoids overflow

        if (nums[mid] == target) {
            return mid; // Target found
        } else if (nums[mid] < target) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }
    return -1; // Target not found
}

```

2. Two Pointers

একটি খুবই সাধারণ কৌশল যেখানে দুটি পয়েন্টার একটি ডেটা স্ট্রাকচারের মধ্য দিয়ে পুনরাবৃত্তি করে যতক্ষণ না একটি বা উভয়ই মিলিত হয়।

কখন ব্যবহার করবেন: একটি সাজানো অ্যারেতে জোড়া খুঁজে বের করা, একটি অ্যারে উল্টানো, অথবা প্যালিনড্রোম সনাক্ত করা।

```
// Example: Check if a string is a palindrome
public boolean isPalindrome(String s) {
    int left = 0;
    int right = s.length() - 1;

    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

3. Recursion (Factorial Example)

একটি ফাংশন যা নিজেকে কল করে। এটি ট্রি/গ্রাফ ট্র্যাভার্সাল এবং ডিভাইড-এন্ড-কনকার অ্যালগরিদমের জন্য অত্যন্ত গুরুত্বপূর্ণ। রিকার্সন বন্ধ করার জন্য প্রতিটি রিকার্সিভ ফাংশনের একটি বেস কেস প্রয়োজন।

```
public int factorial(int n) {
    // Base case: The condition to stop the recursion.
    if (n <= 1) {
        return 1;
    }
    // Recursive step: The function calls itself with a modified input.
    return n * factorial(n - 1);
}
```

"Medium" category.

1. Set (HashSet)

সেট হলো এমন একটি সংগ্রহ যা শুধুমাত্র অনন্য উপাদান সংরক্ষণ করে। HashSet অভ্যন্তরীণভাবে একটি হ্যাশ ম্যাপ ব্যবহার করে, যার ফলে একটি উপাদানের অস্তিত্ব যোগ করা, অপসারণ করা এবং পরীক্ষা করা অত্যন্ত দ্রুত হয়, সাধারণত গড়ে O(1)।

কখন ব্যবহার করবেন: যেকোনো সময় আপনাকে অনন্য আইটেমগুলির ট্র্যাক রাখতে হবে অথবা দ্রুত পরীক্ষা করতে হবে যে কোনও আইটেম আগে দেখা হয়েছে কিনা। "Contains Duplicate" বা সংগ্রহে অনন্য উপাদান খুঁজে বের করার মতো সমস্যার জন্য উপযুক্ত।

```
import java.util.HashSet;
import java.util.Set;

Set<Integer> uniqueNumbers = new HashSet<>();

uniqueNumbers.add(5);
uniqueNumbers.add(10);
uniqueNumbers.add(5); // This will be ignored, as 5 is already in the set.

// Check for existence
if (uniqueNumbers.contains(10)) {
    System.out.println("Set contains 10."); // This will print
}

uniqueNumbers.remove(10); // Remove an element

System.out.println("Set size: " + uniqueNumbers.size()); // Prints 1

// Example: Find if an array has duplicates
int[] nums = {1, 2, 3, 1};
Set<Integer> seen = new HashSet<>();
for (int num : nums) {
    if (seen.contains(num)) {
        System.out.println("Found a duplicate: " + num);
        break;
    }
    seen.add(num);
}
```

2. Priority Queue (Heap)

PriorityQueue হল একটি বিশেষ ধরণের কিউ যেখানে উপাদানগুলিকে তাদের প্রাকৃতিক ক্রম বা একটি কাস্টম তুলনাকারীর উপর ভিত্তি করে সাজানো হয়। জোড়াতে, এটি ডিফল্টভাবে একটি মিন-হিপ হিসাবে প্রয়োগ করা হয়, যার অর্থ সবচেয়ে ছোট উপাদানটি সর্বদা সামনে থাকে (peek())।

কখন ব্যবহার করবেন: "Top K" উপাদান, সবচেয়ে ছোট/বৃহত্তম উপাদান খুঁজে বের করা, অথবা অগ্রাধিকারের ক্রম অনুসারে ইভেন্ট পরিচালনা করা সম্পর্কিত সমস্যা (যেমন, Dijkstra's বা A* অনুসন্ধান অ্যালগরিদম)।

```
import java.util.PriorityQueue;
import java.util.Collections;

// Min-Heap (default behavior) - smallest element first
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
minHeap.add(30);
minHeap.add(10);
minHeap.add(20);

System.out.println("Top of min-heap: " + minHeap.peek()); // Prints 10
minHeap.poll(); // Removes 10
System.out.println("Top of min-heap now: " + minHeap.peek()); // Prints 20

// Max-Heap - largest element first
// We achieve this by providing a reverse order comparator.
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
maxHeap.add(30);
maxHeap.add(10);
maxHeap.add(20);

System.out.println("Top of max-heap: " + maxHeap.peek()); // Prints 30
maxHeap.poll(); // Removes 30
System.out.println("Top of max-heap now: " + maxHeap.peek()); // Prints 20
```

3. Graph (Adjacency List Representation)

গ্রাফগুলি প্রাণ্ত দ্বারা সংযুক্ত নোডের (শীর্ষস্থান) নেটওয়ার্কগুলিকে প্রতিনিধিত্ব করে।
গেডিং সাক্ষাৎকারে গ্রাফ উপস্থাপনের সবচেয়ে সাধারণ উপায় হল একটি সংলগ্ন তালিকা,
যা একটি মানচিত্র যেখানে প্রতিটি কী একটি নোড এবং এর মান তার প্রতিবেশীদের একটি
তালিকা।

কখন ব্যবহার করবেন: কোনও নেটওয়ার্কের সাথে সম্পর্কিত যেকোনো সমস্যা, যেমন
সামাজিক নেটওয়ার্ক, ফ্লাইট পাথ, নির্ভরতা, অথবা একটি গোলকধাঁধা অতিক্রম করা।

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

// Graph representation using an Adjacency List
class Graph {
    private Map<Integer, List<Integer>> adjList = new HashMap<>();

    // Add a new node to the graph
    public void addNode(int node) {
        adjList.putIfAbsent(node, new ArrayList<>());
    }

    // Add an edge between two nodes (for an undirected graph)
    public void addEdge(int source, int destination) {
        // Ensure both nodes exist in the graph
        addNode(source);
        addNode(destination);

        // Add edge from source to destination
        adjList.get(source).add(destination);
        // Add edge from destination to source
        adjList.get(destination).add(source);
    }

    // Get the neighbors of a node
    public List<Integer> getNeighbors(int node) {
        return adjList.getOrDefault(node, new ArrayList<>());
    }

    @Override
    public String toString() {
        return adjList.toString();
    }
}

// Example usage
Graph g = new Graph();
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 3);

System.out.println(g); // Prints {0=[1, 2], 1=[0, 2], 2=[0, 1, 3], 3=[2]}
```

Algorithms & Techniques

1. Breadth-First Search (BFS)

An algorithm for traversing a tree or graph. It explores neighbor nodes first before moving to the next level of neighbors. It uses a **Queue**.

- **Key Idea:** Level-by-level traversal.
- **Use Cases:** Finding the shortest path in an unweighted graph, level-order traversal of a tree, web crawling.

```
// BFS for Tree Level-Order Traversal
// Uses the TreeNode class from the previous response
public void bfs(TreeNode root) {
    if (root == null) return;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root); // Start with the root node

    while (!queue.isEmpty()) {
        TreeNode current = queue.poll(); // Get and remove the node at the front

        System.out.print(current.val + " "); // Process the current node

        // Add its children to the queue
        if (current.left != null) {
            queue.add(current.left);
        }
        if (current.right != null) {
            queue.add(current.right);
        }
    }
}
```

2. Depth-First Search (DFS)

Another algorithm for traversing a tree or graph. It explores as far as possible along each branch before backtracking. It can be implemented with Recursion (using the call stack) or iteratively with a Stack.

- Key Idea: Go deep first, then backtrack.
- Use Cases: Finding a path between two nodes, detecting cycles in a graph, solving maze puzzles, tree traversals (pre-order, in-order, post-order are all types of DFS).

```
// Recursive DFS for a Tree (Pre-order Traversal: Root, Left, Right)
// Uses the TreeNode class from the previous response
public void dfs(TreeNode node) {
    if (node == null) {
        return; // Base case
    }

    // Process the current node
    System.out.print(node.val + " ");

    // Recurse on the left child
    dfs(node.left);

    // Recurse on the right child
    dfs(node.right);
}
```

3. Sliding Window

A technique used on arrays or strings. A "window" of a certain size or property slides over the data. It's highly efficient because you avoid re-computing values for overlapping parts of the window. The time complexity is often $O(N)$.

- Key Idea: Maintain a subarray (window) and slide it through the collection, updating it one element at a time.

- Use Cases: Finding the maximum/minimum sum of a fixed-size subarray, finding the longest substring with certain properties.

```
// Example: Find the max sum of any subarray of size 'k'
public int findMaxSumSubarray(int[] nums, int k) {
    int maxSum = 0;
    int windowSum = 0;
    int windowStart = 0;

    for (int windowEnd = 0; windowEnd < nums.length; windowEnd++) {
        windowSum += nums[windowEnd]; // Add the next element to the window

        // Slide the window when it's of size k
        if (windowEnd >= k - 1) {
            maxSum = Math.max(maxSum, windowSum); // Update the max sum
            windowSum -= nums[windowStart]; // Subtract the element leaving the window
            windowStart++; // Slide the window forward
        }
    }
    return maxSum;
}
```

Example Problem Walkthrough: "Valid Parentheses" (LeetCode #20)

Let's see how to combine these ideas.

Problem: Given a string s containing just the characters $($, $)$, $\{$, $\}$, $[$ and $]$, determine if the input string is valid. An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Thought Process:

1. We need to process the brackets in order.
2. When we see an opening bracket like $($, we need to remember it.

3. When we see a closing bracket like), it must match the *most recently seen* opening bracket.
4. This "most recently seen" suggests a Last-In, First-Out (LIFO) behavior.
5. **Conclusion:** A **Stack** is the perfect data structure for this job.

```

import java.util.Stack;
import java.util.HashMap;

class ValidParenthesesSolution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        // Use a map for easy lookup of matching pairs
        HashMap<Character, Character> map = new HashMap<>();
        map.put(')', '(');
        map.put('}', '{');
        map.put(']', '[');

        for (char c : s.toCharArray()) {
            // If it's a closing bracket
            if (map.containsKey(c)) {
                // If stack is empty or top doesn't match, it's invalid
                if (stack.isEmpty() || stack.pop() != map.get(c)) {
                    return false;
                }
            } else {
                // If it's an opening bracket, push it onto the stack
                stack.push(c);
            }
        }

        // If the stack is empty at the end, all brackets were matched.
        return stack.isEmpty();
    }
}

```

"Hard" level problems

1. Dynamic Programming (DP)

Dynamic Programming is a powerful technique for solving optimization problems by breaking them down into simpler, overlapping subproblems. The key is to solve each subproblem only once and store its result.

- **When to use:** When you see problems asking for the "optimal," "maximum," or "minimum" value of something, or the "number of ways" to do something, and you can see that the main problem can be built up from solutions to smaller versions of the same problem.
- **Two Main Approaches:**
 1. **Memoization (Top-Down):** You write a standard recursive function and use a cache (like a HashMap or an array) to store the results of function calls. Before computing, you check the cache. If the result is there, you return it; otherwise, you compute it, store it, and then return it.
 2. **Tabulation (Bottom-Up):** You build a table (usually an array or 2D array) from the ground up, starting with the base cases. You fill the table iteratively until you reach the answer for the main problem.

Classic Example: Fibonacci Sequence

```

// Approach 1: Memoization (Top-Down)
class DPMemoization {
    // A map to store results of solved subproblems
    private Map<Integer, Integer> cache = new HashMap<>();

    public int fib(int n) {
        if (n <= 1) {
            return n;
        }

        // If we've computed this before, return the cached value
        if (cache.containsKey(n)) {
            return cache.get(n);
        }

        // Otherwise, compute it, cache it, and return it
        int result = fib(n - 1) + fib(n - 2);
        cache.put(n, result);
        return result;
    }
}

// Approach 2: Tabulation (Bottom-Up)
class DPTabulation {
    public int fib(int n) {
        if (n <= 1) {
            return n;
        }

        // A table to store results from the bottom up
        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;

        // Fill the table iteratively
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}

```

2. Backtracking

ব্যাকট্র্যাকিং হল একটি অ্যালগরিদমিক কৌশল যা ক্রমবর্ধমানভাবে একটি সমাধান তৈরি করার চেষ্টা করে পুনরাবৃত্তিমূলকভাবে সমস্যা সমাধানের জন্য। আপনি একটি পথ অন্বেষণ করেন, এবং যদি আপনি একটি "অচলাবস্থায়" পৌঁছান বা বুঝতে

পারেন যে পথটি একটি বৈধ সমাধানের দিকে নিয়ে যাবে না, তাহলে আপনি "ব্যাকট্র্যাক" (আপনার শেষ পছন্দটি পূর্বাবস্থায় ফিরিয়ে আনুন) এবং অন্য পথ চেষ্টা করুন।

When to use: Problems that ask for "all possible" solutions, such as generating all permutations, combinations, subsets, or solving puzzles like Sudoku or N-Queens.

```
import java.util.ArrayList;
import java.util.List;

class BacktrackingPermutations {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), nums);
        return result;
    }

    private void backtrack(List<List<Integer>> result, List<Integer> tempList, int[] nums) {
        // Base case: If the temporary list is the same size as the input array
        // we have found a complete permutation.
        if (tempList.size() == nums.length) {
            result.add(new ArrayList<>(tempList));
        } else {
            for (int i = 0; i < nums.length; i++) {
                // Skip if the element is already in our current permutation
                if (tempList.contains(nums[i])) {
                    continue;
                }

                // 1. Make a choice
                tempList.add(nums[i]);

                // 2. Explore
                backtrack(result, tempList, nums);

                // 3. Un-choice (Backtrack)
                tempList.remove(tempList.size() - 1);
            }
        }
    }
}
```

3. Trie (Prefix Tree)

Trie হলো একটি বিশেষ ট্রি-ভিত্তিক ডেটা স্ট্রাকচার যা দক্ষতার সাথে স্ট্রিং সংরক্ষণ এবং পুনরুদ্ধারের জন্য ব্যবহৃত হয়। প্রতিটি নোড একটি অক্ষরকে প্রতিনিধিত্ব করে এবং রুট থেকে নোডে ঘাওয়ার পথ একটি উপসর্গকে প্রতিনিধিত্ব করে। এটি উপসর্গ-ভিত্তিক অনুসন্ধানের জন্য অবিশ্বাস্যভাবে দ্রুত।

When to use: Autocomplete systems, spell checkers, dictionary lookups, or any problem involving string prefixes.

```
// Node definition for a Trie
class TrieNode {
    // Each node can have up to 26 children (for 'a' through 'z')
    public TrieNode[] children = new TrieNode[26];
    public boolean isEndOfWord = false;
}

class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie
    public void insert(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }
}
```

```

// Returns if there is any word in the trie that starts with the given prefix
public boolean startsWith(String prefix) {
    TrieNode current = root;
    for (char c : prefix.toCharArray()) {
        int index = c - 'a';
        if (current.children[index] == null) {
            return false; // Prefix not found
        }
        current = current.children[index];
    }
    return true; // Prefix found
}

// Returns if the word is in the trie
public boolean search(String word) {
    TrieNode current = root;
    for (char c : word.toCharArray()) {
        int index = c - 'a';
        if (current.children[index] == null) {
            return false;
        }
        current = current.children[index];
    }
    return current.isEndOfWord;
}

```

4. Divide and Conquer

This is a core algorithmic paradigm where a problem is solved by:

1. **Divide:** Breaking the problem into smaller, independent subproblems of the same type.
2. **Conquer:** Solving the subproblems recursively.
3. **Combine:** Combining the solutions of the subproblems to solve the original problem.

- **When to use:** Problems that can be naturally split into smaller, non-overlapping parts. **Merge Sort** and **Quick Sort** are classic examples. Binary Search is also a form of Divide and Conquer.

Example: Merge Sort

```

class MergeSort {
    public void sort(int[] arr, int l, int r) {
        if (l < r) {
            // 1. Divide: Find the middle point
            int m = l + (r - l) / 2;

            // 2. Conquer: Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);

            // 3. Combine: Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    // Merges two subarrays of arr[]
    void merge(int[] arr, int l, int m, int r) {
        // Create temp arrays
        int n1 = m - l + 1;
        int n2 = r - m;
        int[] L = new int[n1];
        int[] R = new int[n2];

        // Copy data to temp arrays
        for (int i = 0; i < n1; ++i) L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j) R[j] = arr[m + 1 + j];

        // Merge the temp arrays
        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k++] = L[i++];
            } else {
                arr[k++] = R[j++];
            }
        }

        // Copy remaining elements
        while (i < n1) arr[k++] = L[i++];
        while (j < n2) arr[k++] = R[j++];
    }
}

```

Here are some mental shortcuts to help you recognize which pattern might apply:

- If the problem asks for the **shortest path** in an unweighted grid/graph
👉 Think **BFS**.
- If the problem asks for the **max/min value** or **number of ways** to do something by making optimal choices 👕 Think **Dynamic Programming**.
- If the problem asks to generate **all possible combinations/permuations/subsets** 👕 Think **Backtracking**.
- If the problem involves **string prefixes**, like in an autocomplete system
👉 Think **Trie**.
- If the problem can be cleanly split into **smaller, identical, independent subproblems** 👕 Think **Divide and Conquer**.
- If you need the **top 'K'** largest/smallest items 👕 Think **Priority Queue (Heap)**.

1. Union-Find (Disjoint Set Union - DSU)

এই ডেটা স্ট্রাকচারটি পার্টিশন বা সংযুক্ত উপাদানগুলির সমস্যাগুলির জন্য অবিশ্বাস্যভাবে কার্যকর। এটি বেশ কয়েকটি বিচ্ছিন্ন (অ-ওভারল্যাপিং) সেটে বিভক্ত উপাদানগুলির একটি সংগ্রহ ট্র্যাক করে। এর দুটি প্রধান, অত্যন্ত অপ্টিমাইজড অপারেশন রয়েছে:

`find(i)`: যে উপাদান i এর সাথে সম্পর্কিত তার "প্রতিনিধিত্ব মূলক" (বা রুট) নির্ধারণ করে।

`union(i, j):` i এবং j উপাদান ধারণকারী সেটগুলিকে একত্রিত করে।

কখন ব্যবহার করবেন: একটি অনির্দেশিত গ্রাফে চক্র সনাক্ত করা, রিয়েল-টাইম গ্রাফ সংযোগ পরীক্ষা করা (যেমন, প্রান্তগুলি যুক্ত হওয়ার সাথে সাথে), অথবা যে কোনও সমস্যা যেখানে আপনাকে উপাদানগুলিকে গোষ্ঠীভুক্ত করতে হবে এবং তারা কোন গোষ্ঠীর অন্তর্ভুক্ত তা জিজ্ঞাসা করতে হবে। এটিকে "বন্ধু বৃক্ত" পরিচালনা করার মতো ভাবুন। 

গতির জন্য মূল অপ্টিমাইজেশন:

পাথ কম্প্রেশন: একটি ফাইন্ড অপারেশনের সময়, আমরা পাথের প্রতিটি নোডকে সরাসরি রুটের দিকে নির্দেশ করি। এটি ভবিষ্যতের অনুসন্ধানের জন্য ট্রিকে সমতল করে।

র্যাঙ্ক/আকার অনুসারে মিলন: দুটি সেট একত্রিত করার সময়, আমরা ছোট গাছটিকে বড় গাছের গোড়ার সাথে সংযুক্ত করি। এটি গাছগুলিকে খুব বেশি গভীর হওয়া থেকে বিরত রাখে।

```
// A highly optimized Union-Find / DSU implementation
class UnionFind {
    private int[] root; // root[i] is the parent of i
    private int[] rank; // rank[i] is the height of the tree rooted at i
    private int count; // Number of disjoint sets

    public UnionFind(int size) {
        root = new int[size];
        rank = new int[size];
        count = size;
        for (int i = 0; i < size; i++) {
            root[i] = i; // Initially, each node is its own root
            rank[i] = 1; // Initially, rank is 1
        }
    }

    public int find(int i) {
        if (root[i] != i) {
            // Path compression: set the parent of i to be its grandparent
            root[i] = find(root[i]);
        }
        return root[i];
    }

    public void union(int i, int j) {
        int ri = find(i);
        int rj = find(j);
        if (ri == rj) {
            return;
        }
        if (rank[ri] > rank[rj]) {
            root[rj] = ri;
        } else if (rank[rj] > rank[ri]) {
            root[ri] = rj;
        } else {
            root[rj] = ri;
            rank[ri]++;
        }
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

```

// Find with Path Compression
public int find(int i) {
    if (root[i] == i) {
        return i;
    }
    // Compress the path by setting the root directly
    root[i] = find(root[i]);
    return root[i];
}

// Union by Rank
public void union(int i, int j) {
    int rootI = find(i);
    int rootJ = find(j);
    if (rootI != rootJ) {
        if (rank[rootI] > rank[rootJ]) {
            root[rootJ] = rootI;
        } else if (rank[rootI] < rank[rootJ]) {
            root[rootI] = rootJ;
        } else { // Ranks are the same, merge one into the other and increment
            root[rootJ] = rootI;
            rank[rootI] += 1;
        }
        count--; // A merge reduces the number of disjoint sets
    }
}

public int getCount() {
    return count;
}

```

2. Topological Sort

This algorithm produces a linear ordering of nodes in a **Directed Acyclic Graph (DAG)**. For every directed edge from node u to node v, u must come before v in the ordering.

- **When to use:** Problems involving dependencies or prerequisites, like course scheduling, build system task ordering, or determining if a graph has a cycle (if a topological sort is not possible, the graph has a cycle). 

Kahn's Algorithm (The standard approach):

1. Compute the "in-degree" (number of incoming edges) for every node.
2. Initialize a queue with all nodes that have an in-degree of 0.
3. While the queue is not empty: a. Dequeue a node. Add it to your result list. b. For each of its neighbors, decrement their in-degree. c. If a neighbor's in-degree becomes 0, enqueue it.

```
// Topological Sort using Kahn's Algorithm
class TopologicalSort {
    public List<Integer> findOrder(int numNodes, int[][] prerequisites) {
        // Step 1: Build graph and in-degree map
        Map<Integer, List<Integer>> adjList = new HashMap<>();
        int[] inDegree = new int[numNodes];
        for (int[] prereq : prerequisites) {
            int dest = prereq[0];
            int src = prereq[1];
            adjList.computeIfAbsent(src, k -> new ArrayList<>()).add(dest);
            inDegree[dest]++;
        }

        // Step 2: Initialize queue with zero-in-degree nodes
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numNodes; i++) {
            if (inDegree[i] == 0) {
                queue.offer(i);
            }
        }

        List<Integer> result = new ArrayList<>();
        // Step 3: Process the queue
        while (!queue.isEmpty()) {
            int node = queue.poll();
            result.add(node);
            for (int dest : adjList.getOrDefault(node, new ArrayList<>())) {
                inDegree[dest]--;
                if (inDegree[dest] == 0) {
                    queue.offer(dest);
                }
            }
        }
        return result;
    }
}
```

```

        if (adjList.containsKey(node)) {
            for (int neighbor : adjList.get(node)) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    queue.offer(neighbor);
                }
            }
        }

        // If result size is not equal to numNodes, there is a cycle.
        if (result.size() != numNodes) {
            return new ArrayList<>(); // Or throw an error
        }

        return result;
    }
}

```

3. Dijkstra's Algorithm

নন-নেগেটিভ এজ ওয়েট সহ একটি ওয়েটেড গ্রাফে একটি একক উৎস নোড থেকে অন্য সকল নোডে সংক্ষিপ্ততম পথ খুঁজে বের করার জন্য ক্লাসিক অ্যালগরিদম।

- **When to use:** Any shortest path problem on a weighted graph where edge weights represent distance, time, or cost, and they are all positive. Think Google Maps routing. 
- **Core Idea:** It uses a **Priority Queue** to greedily and efficiently explore the node that is currently closest to the source.

```
// Dijkstra's Algorithm for Single-Source Shortest Path
class Dijkstra {    no usages
    public Map<Integer, Integer> findShortestPaths(int startNode, Map<Integer, List<int[]>> adjList) {
        // Priority queue to store {distance, node}. Ordered by distance.
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);
        // Map to store the shortest distance found so far for each node
        Map<Integer, Integer> distances = new HashMap<>();

        // Start with the source node
        distances.put(startNode, 0);
        pq.offer(new int[]{0, startNode});

        while (!pq.isEmpty()) {
            int[] current = pq.poll();
            int currentDist = current[0];
            int currentNode = current[1];

            // If we've found a shorter path already, skip
            if (currentDist > distances.getOrDefault(currentNode, Integer.MAX_VALUE)) {
                continue;
            }

            // Explore neighbors
            if (adjList.containsKey(currentNode)) {
                for (int[] edge : adjList.get(currentNode)) {
                    int neighbor = edge[0];
                    int weight = edge[1];
                    int newDist = currentDist + weight;

                    // If we found a shorter path to the neighbor, update and add to PQ
                    if (newDist < distances.getOrDefault(neighbor, Integer.MAX_VALUE)) {
                        distances.put(neighbor, newDist);
                        pq.offer(new int[]{newDist, neighbor});
                    }
                }
            }
        }
        return distances;
    }
}
```

4. Segment Tree

A versatile tree-based data structure that allows for very fast **range queries** and **point updates** on an array. For an array of size N, it can perform range queries (like sum, min, or max) and update elements in O(logN) time.

- **When to use:** Problems like "Range Sum Query," "Range Minimum Query," or any scenario where you need to repeatedly query properties of a subarray and potentially update elements within it.

```
// Segment Tree for Range Sum Query
class SegmentTree {
    private int[] tree;
    private int[] nums;
    private int n;

    public SegmentTree(int[] nums) {
        this.n = nums.length;
        this.nums = nums;
        // The size of the tree array is roughly 4*n
        this.tree = new int[4 * n];
        build(0, 0, n - 1);
    }

    // Recursively build the tree
    private void build(int node, int start, int end) {
        if (start == end) {
            tree[node] = nums[start];
        } else {
            int mid = start + (end - start) / 2;
            int leftChild = 2 * node + 1;
            int rightChild = 2 * node + 2;
            build(leftChild, start, mid);
            build(rightChild, mid + 1, end);
            tree[node] = tree[leftChild] + tree[rightChild];
        }
    }

    // Public method to query a range sum
    public int query(int l, int r) {
        return query(0, 0, n - 1, l, r);
    }
}
```

```

// Private helper for range sum query
private int query(int node, int start, int end, int l, int r) {
    if (r < start || end < l) {
        return 0; // Range is completely outside
    }
    if (l <= start && end <= r) {
        return tree[node]; // Range is completely inside
    }
    int mid = start + (end - start) / 2;
    int p1 = query(2 * node + 1, start, mid, l, r);
    int p2 = query(2 * node + 2, mid + 1, end, l, r);
    return p1 + p2;
}

// Public method to update a value
public void update(int idx, int val) {
    update(0, 0, n - 1, idx, val);
}

// Private helper for update
private void update(int node, int start, int end, int idx, int val) {
    if (start == end) {
        nums[idx] = val;
        tree[node] = val;
    } else {
        int mid = start + (end - start) / 2;
        int leftChild = 2 * node + 1;
        int rightChild = 2 * node + 2;
        if (start <= idx && idx <= mid) {
            update(leftChild, start, mid, idx, val);
        } else {
            update(rightChild, mid + 1, end, idx, val);
        }
        tree[node] = tree[leftChild] + tree[rightChild];
    }
}

```