DSA Leetcode

Arrays

What is an Array?

- An **array** is a **linear data structure** that stores elements in **contiguous memory locations**.
- It allows you to store multiple values of the **same data type** in a single structure.

Key Characteristics

1. **Indexing**: Arrays use **0-based indexing**, so the first element is at index 0, the second at 1, and so on.
2. **Fixed Size**: In most programming languages (e.g., Java), arrays have a **fixed size** after being initialized.
3. **Homogeneous Data**: All elements in an array must be of the **same type** (e.g., integers, strings, etc.).

Common Operations and Their Time Complexities

Detailed Explanation of Operations:

1. **Accessing Elements**:
    - To access the element at index i, the memory address is computed as:

$$\text{address} = \text{base\_address} + i \times \text{size\_of\_element}$$

2. **Insertion and Deletion**:
    - **At the End**: Constant time if no resizing is needed ($O(1)$).
    - **At the Beginning or Middle**: Requires shifting elements to maintain order ($O(n)$).
3. **Search**:
    - **Unsorted Array**: Linear search is required, iterating through the array ($O(n)$).
    - **Sorted Array**: Binary search can be used ($O(\log n)$).

Advantages of Arrays

1. **Fast Access**: O(1) access for random indices.
2. **Contiguity**: Efficient in terms of memory locality (helps with caching).

Disadvantages of Arrays

1. **Fixed Size**: Arrays are not dynamic (fixed size in most languages like Java).
2. **Costly Insert/Delete**: Requires shifting elements.

Strings

What is a String?

- A **string** is a sequence of characters, typically stored as an array of characters.
- In Java, the `String` class is **immutable**, meaning its value cannot be changed once created.

Key Characteristics

1. **Immutable (in Java)**:
    - Modifying a string creates a **new string object** instead of altering the original.
2. **Indexing**:
    - Strings use 0-based indexing like arrays.

Common Operations and Their Time Complexities

Detailed Explanation of Operations:

1. **Accessing Characters**:
    - Like arrays, accessing a specific character (e.g., `s.charAt(i)`) is O(1).
2. **Concatenation**:
    - Concatenating two strings of lengths `n` and `m` involves creating a new string of size `n + m` and copying the characters (`O(n + m)`).
3. **Substring**:
    - Extracting a substring of length `k` is O(k) because characters need to be copied into the new substring.
4. **Search**:
    - Searching for a character or substring in `s` using methods like `indexOf()` or `contains()` has a time complexity of O(n).

Memory Usage in Strings

1. Strings are stored in the **string pool** in Java to reduce memory usage.
2. When you modify a string, Java creates a **new string object** to ensure immutability.

Advantages of Strings

1. **Immutability**:
    - Useful for security (e.g., passwords in memory cannot be altered accidentally).
2. **Easy Comparison**:
    - Strings can be compared using `equals()` or `compareTo()`.

Disadvantages of Strings

1. **Immutable in Java**:
    - Modifying strings is costly due to frequent creation of new objects.
2. **High Memory Overhead**:
    - Storing a large number of strings requires significant memory.

Problem Patterns

ARRAYS

1. Two Pointers

Two pointers is one of the most versatile and powerful techniques for solving array problems. The idea is to have two pointers traverse the array simultaneously, often in opposite directions or at different speeds, to achieve efficient solutions.

Why Two Pointers?

- Reduces time complexity by avoiding nested loops (e.g., from $O(n^2)$ to $O(n)$).
- Ideal for sorted arrays, as relative order is leveraged to guide the pointers.

Example: Remove Duplicates from Sorted Array

Problem: Given a sorted array `nums`, remove duplicates in-place and return the length of the unique elements.
Constraints: You must do this in O(1) extra space.

Approach:

1. Use one pointer (`j`) to keep track of the "unique" position in the array.
2. Use the second pointer (`i`) to traverse the array.
3. Whenever `nums[i] != nums[j]`, update `nums[j + 1] = nums[i]` and increment `j`.

Step-by-Step:

- Input: `nums = [1, 1, 2, 3, 3]`
- **Initialization**: `j = 0`
- Traverse with `i`:
  - When `i = 1`, skip (duplicate).
  - When `i = 2`, overwrite `nums[1]` with `2`.
  - When `i = 3`, overwrite `nums[2]` with `3`.
- Result: `[1, 2, 3, _, _]`.

Code:

```
public int removeDuplicates(int[] nums) {
    if (nums.length == 0) return 0;

    int j = 0; // Pointer for unique elements
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] != nums[j]) {
            j++;
            nums[j] = nums[i];
        }
    }
    return j + 1;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

---

2. Sliding Window

The sliding window pattern is used to solve problems involving contiguous subarrays or substrings. A "window" is essentially a subarray defined by two pointers (usually `start` and `end`), which dynamically adjust as you traverse the array.

Why Sliding Window?

- Efficiently calculates results for problems involving subsets of data, like finding max/min values, sums, or unique elements within a range.
- Optimizes brute force approaches by avoiding repetitive computations.

Example: Maximum Sum Subarray of Size K

Problem: Find the maximum sum of a subarray of size `k`.
Approach:

1. Use a fixed-size sliding window of length `k`.
2. Start with the first `k` elements to initialize the window.
3. Slide the window by adding the next element and removing the first.

Step-by-Step:

- Input: `nums = [2, 3, 4, 1, 5], k = 3`
- Start with `windowSum = 2 + 3 + 4 = 9`.
- Slide the window:
  - Remove `2`, add `1`: `windowSum = 9 - 2 + 1 = 8`.
  - Remove `3`, add `5`: `windowSum = 8 - 3 + 5 = 10`.
- Maximum sum is `10`.

Code:

```
public int maxSumSubarray(int[] nums, int k) {
    int maxSum = 0, windowSum = 0;

    for (int i = 0; i < nums.length; i++) {
        windowSum += nums[i];
        if (i >= k - 1) { // Window has reached size k
            maxSum = Math.max(maxSum, windowSum);
            windowSum -= nums[i - (k - 1)]; // Slide window
        }
    }
    return maxSum;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Example: Longest Subarray with Sum ≤ K

Problem: Find the longest subarray whose sum is ≤ k.
Approach:

1. Maintain a variable windowSum and adjust the window size dynamically.
2. Expand the window (move end pointer) while the condition is satisfied.
3. Shrink the window (move start pointer) if the condition is violated.

Code:

```
public int longestSubarraySum(int[] nums, int k) {
    int start = 0, windowSum = 0, maxLength = 0;

    for (int end = 0; end < nums.length; end++) {
        windowSum += nums[end];
        while (windowSum > k) { // Shrink window
            windowSum -= nums[start];
            start++;
        }
        maxLength = Math.max(maxLength, end - start + 1);
    }
    return maxLength;
}
```

---

3. Prefix Sum

The prefix sum technique is used to compute range-based operations efficiently by precomputing cumulative sums. This avoids recalculating sums for overlapping subarrays.

Why Prefix Sum?

- Quickly calculates subarray sums: sum[i:j] = prefix[j] - prefix[i-1].
- Works well with hash maps for cumulative frequency counts.

Example: Subarray Sum Equals K

Problem: Find the number of subarrays that sum to k.
Approach:

1. Compute the cumulative sum (prefixSum) as you traverse.
2. Use a HashMap to store counts of prefixSum.
3. For every prefixSum, check if (prefixSum - k) exists in the map.

Code:

```
public int subarraySum(int[] nums, int k) {
    int count = 0, sum = 0;
    Map<Integer, Integer> map = new HashMap<>();
    map.put(0, 1); // Base case for subarrays starting at index 0

    for (int num : nums) {
        sum += num;
        count += map.getOrDefault(sum - k, 0); // Check for complement
        map.put(sum, map.getOrDefault(sum, 0) + 1); // Store prefixSum
    }
    return count;
}
```

Time Complexity: O(n)
Space Complexity: O(n)

STRINGS

1. Sliding Window for Strings

Sliding window applies similarly to strings, especially for substring problems like "longest substring with X properties."

Example: Longest Substring Without Repeating Characters

Problem: Find the length of the longest substring without repeating characters.
Approach:

1. Use a sliding window and a `Set` to store unique characters.
2. Expand the window by adding characters to the set.
3. If a duplicate is found, shrink the window from the left until it's valid.

Code:

```
public int lengthOfLongestSubstring(String s) {
    Set<Character> set = new HashSet<>();
    int start = 0, maxLength = 0;

    for (int end = 0; end < s.length(); end++) {
        while (set.contains(s.charAt(end))) { // Shrink window
            set.remove(s.charAt(start));
            start++;
        }
        set.add(s.charAt(end)); // Expand window
        maxLength = Math.max(maxLength, end - start + 1);
    }
```

```
    return maxLength;
}
```

---

2. Hash Maps for Character Frequencies

Hash maps are commonly used for counting characters, especially in problems involving anagrams or frequency checks.

Example: Valid Anagram

Problem: Check if two strings are anagrams of each other.
Approach:

1. Count character frequencies for both strings.
2. Compare the frequency arrays.

Code:

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;

    int[] count = new int[26];
    for (char c : s.toCharArray()) count[c - 'a']++;
    for (char c : t.toCharArray()) {
        if (--count[c - 'a'] < 0) return false;
    }
    return true;
}
```

What's Next?

Would you like to explore **advanced patterns** like **Matrix Traversals** or dive into **specific problems**?

Sure! Here's the **Two Pointer Approach** explained with **Java examples** for different scenarios.

Two Pointer Approach in DSA

The **Two Pointer Technique** is an efficient approach for solving array and string problems. It involves using two indices (pointers) that traverse the data structure in different ways to optimize time complexity, often reducing it from **O(n²) to O(n).**

Types of Two Pointer Techniques

1. Opposite Direction (Two Ends)

- Used in **sorted arrays or strings**.
- One pointer starts from the **left** (low), and the other starts from the **right** (high).
- Commonly used in problems like **pair sum, palindrome checking, and two-sum (sorted array).**

Example 1: Finding if a pair sums to a target in a sorted array

```
class TwoSumSorted {
    public static boolean hasPairWithSum(int[] arr, int target) {
        int left = 0, right = arr.length - 1;

        while (left < right) {
            int sum = arr[left] + arr[right];

            if (sum == target) {
                System.out.println("Pair found: (" + arr[left] + ", " + arr[right] + ")");
                return true;
            } else if (sum < target) {
                left++;  // Move the left pointer right
            } else {
                right--; // Move the right pointer left
            }
        }

        return false; // No pair found
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 8, 9};
        int target = 10;
        hasPairWithSum(arr, target);
    }
}
```

Time Complexity: O(n) (since we traverse the array once)

Example 2: Checking if a String is a Palindrome

```
class PalindromeCheck {
    public static boolean isPalindrome(String s) {
        int left = 0, right = s.length() - 1;

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

```
            left++;
            right--;
        }

        return true;
    }

    public static void main(String[] args) {
        String str = "racecar";
        System.out.println("Is palindrome? " + isPalindrome(str));
    }
}
```

Time Complexity: O(n) (each character is checked once)

---

2. Same Direction (Sliding Window)

- One pointer **moves ahead of the other** to maintain a condition.
- Common in **subarray problems, string problems, and longest/shortest subarrays.**

Example 3: Finding the longest subarray with a sum ≤ target

```
class LongestSubarray {
    public static int longestSubarray(int[] arr, int target) {
        int left = 0, sum = 0, maxLength = 0;

        for (int right = 0; right < arr.length; right++) {
            sum += arr[right];

            while (sum > target) {
                sum -= arr[left];  // Shrink the window
                left++;
            }

            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        int target = 10;
        System.out.println("Longest subarray length: " + longestSubarray(arr, target));
    }
}
```

Time Complexity: O(n) (each element is processed at most twice)

---

3. Merging Two Sorted Arrays (Like Merge Sort)

```
import java.util.Arrays;

class MergeSortedArrays {
    public static int[] mergeSortedArrays(int[] arr1, int[] arr2) {
        int n1 = arr1.length, n2 = arr2.length;
        int[] result = new int[n1 + n2];
        int i = 0, j = 0, k = 0;

        while (i < n1 && j < n2) {
            if (arr1[i] < arr2[j]) {
                result[k++] = arr1[i++];
            } else {
                result[k++] = arr2[j++];
            }
        }

        while (i < n1) result[k++] = arr1[i++];
        while (j < n2) result[k++] = arr2[j++];

        return result;
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5, 7};
        int[] arr2 = {2, 4, 6, 8};
        System.out.println("Merged array: " + Arrays.toString(mergeSortedArrays(arr1, arr2)));
    }
}
```

Time Complexity: O(n + m) (since we traverse both arrays once)

---

4. Removing Duplicates from a Sorted Array (In-place)

```
class RemoveDuplicates {
    public static int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        int i = 0; // Slow pointer
        for (int j = 1; j < nums.length; j++) {
            if (nums[j] != nums[i]) {
                i++;
                nums[i] = nums[j]; // Overwrite duplicate
            }
        }

        return i + 1; // New length of the array
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 2, 2, 3, 4, 4, 5};
        int newLength = removeDuplicates(nums);
```

```
        System.out.println("New length: " + newLength);
    }
}
```

Time Complexity: O(n) (each element is checked once)

Common Problems Solved Using Two Pointers

1. **Pair Sum / Two Sum (Sorted Array)**
2. **Checking for Palindromes**
3. **Merging Two Sorted Arrays**
4. **Finding Triplets (Three Sum)**
5. **Removing Duplicates from a Sorted Array**
6. **Max/Min Subarrays using the Sliding Window technique**
7. **Reversing an Array or String in-place**

Key Takeaways

- The **Two Pointer Technique** is highly effective in reducing nested loops from **O(n²) to O(n)**.
- It is used for **array and string-based problems**.
- The two main strategies are:
    - **Opposite Direction:** When pointers start from both ends and move toward each other.
    - **Same Direction (Sliding Window):** When one pointer advances while the other maintains a window.

Would you like help with a specific **two-pointer problem**?

3 Sum Problem Explanation

The **3Sum problem** is a classic **two-pointer problem** where the goal is to find all unique triplets (a, b, c) in a given array such that:

$$a + b + c = 0$$

Example

Input:

nums = [-1, 0, 1, 2, -1, -4]

Output:

[[-1, -1, 2], [-1, 0, 1]]

These are the unique triplets where the sum is **zero**.

Optimal Approach: Sorting + Two Pointers

A **brute force** solution (using three nested loops) takes **O(n³)** time, which is inefficient. A better approach uses **sorting + two pointers**, reducing the time complexity to **O(n²)**.

Algorithm

1. **Sort the array** (O(n log n))
2. **Fix one element (nums[i])** and use a **two-pointer approach** to find the other two elements that sum to -nums[i]
3. **Avoid duplicates** by skipping identical elements.

Java Solution

```
import java.util.*;

class ThreeSum {
    public static List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums); // Step 1: Sort the array

        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue; // Skip duplicates

            int left = i + 1, right = nums.length - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                if (sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));

                    // Move pointers and avoid duplicates
                    while (left < right && nums[left] == nums[left + 1]) left++;
                    while (left < right && nums[right] == nums[right - 1]) right--;

                    left++;
                    right--;
```

```
            } else if (sum < 0) {
                left++; // Increase sum by moving left pointer
            } else {
                right--; // Decrease sum by moving right pointer
            }
        }
    }
    return result;
  }

  public static void main(String[] args) {
    int[] nums = {-1, 0, 1, 2, -1, -4};
    System.out.println(threeSum(nums));
  }
}
```

Time Complexity: O(n²)

- Sorting takes **O(n log n)**
- The two-pointer approach inside the loop takes **O(n)**
- Since we iterate `n` times, the overall complexity is **O(n²)**

Space Complexity: O(1)

- No extra space is used apart from the output list.

Key Takeaways

✓ Sorting helps eliminate duplicates easily
✓ Two-pointer approach optimizes the solution to O(n²)
✓ Avoiding duplicate triplets is crucial

Would you like further explanation or another example?

Prefix Sum Explanation

Prefix Sum is a powerful technique used in **array-based problems** to efficiently calculate cumulative sums. It helps optimize **range sum queries** and subarray problems by reducing time complexity from **O(n) per query** to **O(1) per query** after preprocessing.

How Prefix Sum Works

For an array **arr[]** of size **n**, the **prefix sum array** is constructed as:

$$\text{prefix}[i] = \text{prefix}[i-1] + \text{arr}[i]$$

Where:

- **prefix[i]** represents the sum of elements from index **0** to **i**.

Once the prefix sum array is built, we can **quickly find the sum of any subarray (l, r)** using:

$$\text{Sum}(l, r) = \text{prefix}[r] - \text{prefix}[l-1]$$

Example

1. Finding the Sum of a Subarray in O(1)

Given Array:

arr = {2, 4, 6, 8, 10}

Prefix Sum Array:

prefix = {2, 6, 12, 20, 30}

To find the sum of subarray **arr[1] to arr[3]** (4 + 6 + 8), use:

$$\text{Sum}(1, 3) = \text{prefix}[3] - \text{prefix}[0] = 20 - 2 = 18$$

Java Implementation

```
class PrefixSumExample {
    public static int[] buildPrefixSum(int[] arr) {
        int n = arr.length;
        int[] prefix = new int[n];
        prefix[0] = arr[0];

        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i - 1] + arr[i];
        }
        return prefix;
    }

    public static int getRangeSum(int[] prefix, int left, int right) {
        if (left == 0) return prefix[right];
```

```
        return prefix[right] - prefix[left - 1];
    }

    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10};
        int[] prefix = buildPrefixSum(arr);

        System.out.println("Sum of range (1, 3): " + getRangeSum(prefix, 1, 3)); // Output: 18
    }
}
```

Time Complexity

- **Prefix Sum Construction: O(n)**
- **Querying a Subarray Sum: O(1)**
- **Overall Complexity (Preprocessing + Queries): O(n) + O(1) per query**

Use Cases of Prefix Sum

1. **Efficient Range Sum Queries** (as shown above)
2. **Finding the Equilibrium Index** (where left sum = right sum)
3. **Subarray Sum Problems**
4. **Finding the Maximum Sum of a Fixed-Length Subarray**
5. **Checking if an Array Can Be Split into Two Equal Sum Parts**

Would you like a more complex example, such as **handling negative numbers or 2D prefix sums**?

Use Cases of Prefix Sum in DSA (with Java Implementations & Detailed Comments)

Prefix Sum is a powerful technique used to optimize range sum queries and subarray problems. Below are some common use cases with detailed Java implementations.

---

1. Efficient Range Sum Queries (O(1) Query Time)

Problem:

Given an array, efficiently compute the sum of elements from index `L` to `R` multiple times.

Solution:

1. **Precompute the prefix sum array** where prefix[i] stores the sum of elements from index 0 to i.
2. For any range (L, R), use:

$$\text{Sum}(L, R) = \text{prefix}[R] - \text{prefix}[L - 1]$$

Java Implementation

```
class RangeSumQuery {
    // Function to build the prefix sum array
    public static int[] buildPrefixSum(int[] arr) {
        int n = arr.length;
        int[] prefix = new int[n];
        prefix[0] = arr[0]; // First element remains the same

        // Compute prefix sums iteratively
        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i - 1] + arr[i];
        }
        return prefix;
    }

    // Function to get the sum of elements from index L to R in O(1)
    public static int getRangeSum(int[] prefix, int L, int R) {
        if (L == 0) return prefix[R]; // If starting from index 0, return directly
        return prefix[R] - prefix[L - 1]; // Use prefix sum formula
    }

    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10}; // Example array
        int[] prefix = buildPrefixSum(arr);

        System.out.println("Sum of range (1, 3): " + getRangeSum(prefix, 1, 3)); // Output: 18
    }
}
```

✅ Time Complexity:

- **Preprocessing:** O(n)
- **Query:** O(1)

---

2. Count Subarrays with a Given Sum

Problem:

Find the number of **subarrays** whose sum equals target.

Solution:

- Use **prefix sum + HashMap** to track occurrences of prefix[i] - target.
- If prefix[i] - target exists in the map, it means there is a subarray that sums to target.

Java Implementation

```
import java.util.*;

class SubarraySumEqualsTarget {
    public static int countSubarrays(int[] arr, int target) {
        Map<Integer, Integer> prefixMap = new HashMap<>(); // Stores prefix sum occurrences
        int prefixSum = 0, count = 0;

        prefixMap.put(0, 1); // To handle cases where prefixSum itself equals target

        for (int num : arr) {
            prefixSum += num; // Compute current prefix sum

            // Check if (prefixSum - target) exists in the map
            if (prefixMap.containsKey(prefixSum - target)) {
                count += prefixMap.get(prefixSum - target); // Add the occurrences
            }

            // Store the current prefix sum in the map
            prefixMap.put(prefixSum, prefixMap.getOrDefault(prefixSum, 0) + 1);
        }
        return count;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 1};
        int target = 2;
        System.out.println("Count of subarrays with sum " + target + ": " + countSubarrays(arr, target)); //
Output: 2
    }
}
```

✅ **Time Complexity:** O(n) (Optimized using HashMap)

---

3. Finding Equilibrium Index

Problem:

Find an **index i** where the sum of elements to the left equals the sum to the right.

Solution:

1. Compute **total sum** of the array.
2. Maintain a running **left sum**.
3. If `leftSum * 2 == totalSum - arr[i]`, then `i` is the equilibrium index.

Java Implementation

```
class EquilibriumIndex {
    public static int findEquilibrium(int[] arr) {
        int totalSum = 0, leftSum = 0;

        // Compute total sum of the array
        for (int num : arr) totalSum += num;

        for (int i = 0; i < arr.length; i++) {
            // Check if left sum equals right sum
            if (leftSum == totalSum - leftSum - arr[i]) return i;
            leftSum += arr[i]; // Update left sum
        }
        return -1; // No equilibrium index found
    }

    public static void main(String[] args) {
        int[] arr = {1, 7, 3, 6, 5, 6};
        System.out.println("Equilibrium Index: " + findEquilibrium(arr)); // Output: 3
    }
}
```

✅ **Time Complexity:** O(n)

---

4. Maximum Sum of Fixed-Length Subarray

Problem:

Find the **maximum sum of any subarray of length `k`**.

Solution:

- Use **prefix sum** to compute subarray sums in **O(1)**.
- Instead of recalculating the sum every time, use:

$$\text{Sum}(i, i + k - 1) = \text{prefix}[i + k - 1] - \text{prefix}[i - 1]$$

Java Implementation

```
class MaxSumSubarray {
    public static int maxSumSubarray(int[] arr, int k) {
        int n = arr.length;
        int[] prefix = new int[n];
        prefix[0] = arr[0];

        // Compute prefix sum
        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i - 1] + arr[i];
        }

        int maxSum = Integer.MIN_VALUE;

        // Iterate over possible subarrays of size k
        for (int i = 0; i <= n - k; i++) {
            int sum = i == 0 ? prefix[i + k - 1] : prefix[i + k - 1] - prefix[i - 1];
            maxSum = Math.max(maxSum, sum);
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        int k = 3;
        System.out.println("Max sum of subarray of size " + k + ": " + maxSumSubarray(arr, k)); // Output: 18
    }
}
```

✅ **Time Complexity:** O(n)

---

5. 2D Prefix Sum (Matrix Sum Queries)

Problem:

Given a 2D matrix, efficiently find the sum of any submatrix (r1, c1) to (r2, c2).

Java Implementation

```
class MatrixPrefixSum {
    public static int[][] buildPrefixSum(int[][] mat) {
        int rows = mat.length, cols = mat[0].length;
        int[][] prefix = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int sum = mat[i][j];
                if (i > 0) sum += prefix[i - 1][j];
                if (j > 0) sum += prefix[i][j - 1];
                if (i > 0 && j > 0) sum -= prefix[i - 1][j - 1];
                prefix[i][j] = sum;
```

```
            }
        }
        return prefix;
    }

    public static int getSubmatrixSum(int[][] prefix, int r1, int c1, int r2, int c2) {
        int sum = prefix[r2][c2];
        if (r1 > 0) sum -= prefix[r1 - 1][c2];
        if (c1 > 0) sum -= prefix[r2][c1 - 1];
        if (r1 > 0 && c1 > 0) sum += prefix[r1 - 1][c1 - 1];
        return sum;
    }
}
```

✅ **Time Complexity:**

- **Preprocessing:** O(n × m)
- **Query:** O(1)

Would you like more variations or a specific problem explained?

Let's go through **Problem 2 (Count Subarrays with a Given Sum)** and **Problem 5 (2D Prefix Sum - Matrix Sum Queries)** with detailed explanations and step-by-step examples.

Problem 2: Count Subarrays with a Given Sum

Problem Statement

Given an array `arr[]` and an integer `target`, find the number of **subarrays** whose sum equals `target`.

Example

Input:

```
arr = [1, 1, 1]
target = 2
```

Output:

```
2
```

Explanation:

The valid subarrays are:

1. `[1, 1]` from index `0` to `1`

2. [1, 1] from index 1 to 2

Brute Force Approach (O(n²))

We check all possible subarrays and count those with a sum equal to target.

```
class BruteForceSolution {
    public static int countSubarrays(int[] arr, int target) {
        int count = 0;

        for (int i = 0; i < arr.length; i++) {
            int sum = 0;
            for (int j = i; j < arr.length; j++) {
                sum += arr[j];
                if (sum == target) count++;
            }
        }

        return count;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 1};
        int target = 2;
        System.out.println("Count of subarrays: " + countSubarrays(arr, target)); // Output: 2
    }
}
```

✅ **Time Complexity:** O(n²) (Nested loop for all subarrays)

Optimized Approach (O(n) using Prefix Sum + HashMap)

1. **Use a HashMap** to store occurrences of prefixSum[i].
2. For each element, check if prefixSum[i] - target exists in the HashMap.
    - This means there exists a subarray that sums to target.

Java Implementation (Optimized)

```
import java.util.HashMap;

class SubarraySumOptimized {
    public static int countSubarrays(int[] arr, int target) {
        HashMap<Integer, Integer> prefixMap = new HashMap<>();
        int prefixSum = 0, count = 0;
```

```
        // Initialize map with prefix sum 0 to handle cases where prefixSum itself equals target
        prefixMap.put(0, 1);

        for (int num : arr) {
            prefixSum += num; // Compute current prefix sum

            // Check if (prefixSum - target) exists in the map
            if (prefixMap.containsKey(prefixSum - target)) {
                count += prefixMap.get(prefixSum - target); // Add occurrences
            }

            // Store the current prefix sum in the map
            prefixMap.put(prefixSum, prefixMap.getOrDefault(prefixSum, 0) + 1);
        }
        return count;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 1};
        int target = 2;
        System.out.println("Count of subarrays: " + countSubarrays(arr, target)); // Output: 2
    }
}
```

✅ **Time Complexity:** O(n)
✅ **Space Complexity:** O(n) (for HashMap)

Problem 5: 2D Prefix Sum (Matrix Sum Queries)

Problem Statement

Given a **2D matrix**, efficiently compute the **sum of any submatrix** (r1, c1) to (r2, c2).

Example

Input:

Matrix:
1 2 3
4 5 6
7 8 9

Submatrix Query: (1,1) to (2,2)

Output:

28

Explanation:

Submatrix (1,1) to (2,2):

5 6
8 9

Sum = **5 + 6 + 8 + 9 = 28**

Step 1: Build the Prefix Sum Matrix

A **prefix sum matrix** stores the sum of all elements from (0,0) to (i,j).
Formula:

$$\text{prefix}[i][j] = \text{matrix}[i][j] + \text{prefix}[i-1][j] + \text{prefix}[i][j-1] - \text{prefix}[i-1][j-1]$$

- prefix[i-1][j] removes the **top** extra sum.
- prefix[i][j-1] removes the **left** extra sum.
- prefix[i-1][j-1] is **added back** to avoid double subtraction.

Step 2: Query a Submatrix Sum Efficiently

Once the prefix sum is built, any submatrix sum (r1, c1) to (r2, c2) can be found in **O(1)** using:

$$\text{Sum}(r1, c1, r2, c2) = \text{prefix}[r2][c2] - \text{prefix}[r1-1][c2] - \text{prefix}[r2][c1-1] + \text{prefix}[r1-1][c1-1]$$

Java Implementation

```
class MatrixPrefixSum {
    // Function to build prefix sum matrix
    public static int[][] buildPrefixSum(int[][] mat) {
        int rows = mat.length, cols = mat[0].length;
        int[][] prefix = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int sum = mat[i][j]; // Start with original matrix value

                if (i > 0) sum += prefix[i - 1][j]; // Add top prefix sum
                if (j > 0) sum += prefix[i][j - 1]; // Add left prefix sum
                if (i > 0 && j > 0) sum -= prefix[i - 1][j - 1]; // Remove double counted area

                prefix[i][j] = sum; // Store in prefix sum matrix
            }
        }
        return prefix;
```

```
    }

    // Function to get sum of submatrix (r1, c1) to (r2, c2)
    public static int getSubmatrixSum(int[][] prefix, int r1, int c1, int r2, int c2) {
        int sum = prefix[r2][c2]; // Start with total sum up to (r2, c2)

        if (r1 > 0) sum -= prefix[r1 - 1][c2]; // Subtract top portion
        if (c1 > 0) sum -= prefix[r2][c1 - 1]; // Subtract left portion
        if (r1 > 0 && c1 > 0) sum += prefix[r1 - 1][c1 - 1]; // Add back double-subtracted area

        return sum;
    }

    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int[][] prefix = buildPrefixSum(matrix);
        System.out.println("Submatrix sum (1,1) to (2,2): " + getSubmatrixSum(prefix, 1, 1, 2, 2)); // Output:
28
    }
}
```

✅ **Time Complexity:**

- **Preprocessing (buildPrefixSum):** O(n × m)
- **Query (getSubmatrixSum):** O(1)

Summary

1. **Problem 2 (Count Subarrays with Given Sum)** → Use **Prefix Sum + HashMap** to optimize to **O(n)**.
2. **Problem 5 (2D Matrix Sum Queries)** → Build a **2D Prefix Sum Matrix** for **O(1)** queries.

Let me know if you need further clarification!

Let's go through **Problem 2 (Count Subarrays with a Given Sum)** and **Problem 5 (2D Prefix Sum - Matrix Sum Queries)** with detailed explanations and step-by-step examples.

Problem 2: Count Subarrays with a Given Sum

Problem Statement

Given an array arr[] and an integer target, find the number of **subarrays** whose sum equals target.

Example

Input:

arr = [1, 1, 1]
target = 2

Output:

2

Explanation:

The valid subarrays are:

1. [1, 1] from index 0 to 1
2. [1, 1] from index 1 to 2

Brute Force Approach ($O(n^2)$)

We check all possible subarrays and count those with a sum equal to target.

```
class BruteForceSolution {
    public static int countSubarrays(int[] arr, int target) {
        int count = 0;

        for (int i = 0; i < arr.length; i++) {
            int sum = 0;
            for (int j = i; j < arr.length; j++) {
                sum += arr[j];
                if (sum == target) count++;
            }
        }

        return count;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 1};
        int target = 2;
        System.out.println("Count of subarrays: " + countSubarrays(arr, target)); // Output: 2
    }
}
```

✅ **Time Complexity:** O(n²) (Nested loop for all subarrays)

Optimized Approach (O(n) using Prefix Sum + HashMap)

1. **Use a HashMap** to store occurrences of prefixSum[i].
2. For each element, check if prefixSum[i] - target exists in the HashMap.
    - This means there exists a subarray that sums to target.

Java Implementation (Optimized)

```
import java.util.HashMap;

class SubarraySumOptimized {
    public static int countSubarrays(int[] arr, int target) {
        HashMap<Integer, Integer> prefixMap = new HashMap<>();
        int prefixSum = 0, count = 0;

        // Initialize map with prefix sum 0 to handle cases where prefixSum itself equals target
        prefixMap.put(0, 1);

        for (int num : arr) {
            prefixSum += num; // Compute current prefix sum

            // Check if (prefixSum - target) exists in the map
            if (prefixMap.containsKey(prefixSum - target)) {
                count += prefixMap.get(prefixSum - target); // Add occurrences
            }

            // Store the current prefix sum in the map
            prefixMap.put(prefixSum, prefixMap.getOrDefault(prefixSum, 0) + 1);
        }
        return count;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 1};
        int target = 2;
        System.out.println("Count of subarrays: " + countSubarrays(arr, target)); // Output: 2
    }
}
```

✅ **Time Complexity:** O(n)
✅ **Space Complexity:** O(n) (for HashMap)

Problem 5: 2D Prefix Sum (Matrix Sum Queries)

Problem Statement

Given a **2D matrix**, efficiently compute the **sum of any submatrix** (r1, c1) to (r2, c2).

Example

Input:

Matrix:
1 2 3
4 5 6
7 8 9

Submatrix Query: (1,1) to (2,2)

Output:

28

Explanation:

Submatrix (1,1) to (2,2):

5 6
8 9

Sum = **5 + 6 + 8 + 9 = 28**

Step 1: Build the Prefix Sum Matrix

A **prefix sum matrix** stores the sum of all elements from (0,0) to (i,j).
Formula:

$$\text{prefix}[i][j] = \text{matrix}[i][j] + \text{prefix}[i-1][j] + \text{prefix}[i][j-1] - \text{prefix}[i-1][j-1]$$

- prefix[i-1][j] removes the **top** extra sum.
- prefix[i][j-1] removes the **left** extra sum.
- prefix[i-1][j-1] is **added back** to avoid double subtraction.

Step 2: Query a Submatrix Sum Efficiently

Once the prefix sum is built, any submatrix sum (r1, c1) to (r2, c2) can be found in **O(1)** using:

$$\text{Sum}(r1, c1, r2, c2) = \text{prefix}[r2][c2] - \text{prefix}[r1-1][c2] - \text{prefix}[r2][c1-1] + \text{prefix}[r1-1][c1-1]$$

Java Implementation

```
class MatrixPrefixSum {
    // Function to build prefix sum matrix
    public static int[][] buildPrefixSum(int[][] mat) {
        int rows = mat.length, cols = mat[0].length;
        int[][] prefix = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int sum = mat[i][j]; // Start with original matrix value

                if (i > 0) sum += prefix[i - 1][j]; // Add top prefix sum
                if (j > 0) sum += prefix[i][j - 1]; // Add left prefix sum
                if (i > 0 && j > 0) sum -= prefix[i - 1][j - 1]; // Remove double counted area

                prefix[i][j] = sum; // Store in prefix sum matrix
            }
        }
        return prefix;
    }

    // Function to get sum of submatrix (r1, c1) to (r2, c2)
    public static int getSubmatrixSum(int[][] prefix, int r1, int c1, int r2, int c2) {
        int sum = prefix[r2][c2]; // Start with total sum up to (r2, c2)

        if (r1 > 0) sum -= prefix[r1 - 1][c2]; // Subtract top portion
        if (c1 > 0) sum -= prefix[r2][c1 - 1]; // Subtract left portion
        if (r1 > 0 && c1 > 0) sum += prefix[r1 - 1][c1 - 1]; // Add back double-subtracted area

        return sum;
    }

    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int[][] prefix = buildPrefixSum(matrix);
        System.out.println("Submatrix sum (1,1) to (2,2): " + getSubmatrixSum(prefix, 1, 1, 2, 2)); // Output:
28
    }
}
```

✅ **Time Complexity:**

- **Preprocessing (buildPrefixSum):** O(n × m)
- **Query (getSubmatrixSum):** O(1)

Summary

1. **Problem 2 (Count Subarrays with Given Sum)** → Use **Prefix Sum + HashMap** to optimize to **O(n)**.
2. **Problem 5 (2D Matrix Sum Queries)** → Build a **2D Prefix Sum Matrix** for **O(1)** queries.

Let me know if you need further clarification!

Sliding Window Technique

The **sliding window** technique is an optimization method used to **reduce time complexity** in problems involving **contiguous subarrays or substrings**. Instead of using nested loops (brute force), we use a **dynamic window** that moves efficiently through the data.

Types of Sliding Window

1. **Fixed-size Sliding Window** → Used when the window size is fixed.
2. **Variable-size Sliding Window** → Used when the window size depends on conditions.

1️⃣ Fixed-Size Sliding Window

Problem:

Find the **maximum sum of a subarray of size** `k`.

Brute Force Approach (O(n²))

We check all possible subarrays of size `k` using nested loops.

```
class BruteForce {
    public static int maxSumSubarray(int[] arr, int k) {
        int maxSum = Integer.MIN_VALUE;
        for (int i = 0; i <= arr.length - k; i++) {
            int sum = 0;
            for (int j = i; j < i + k; j++) {
                sum += arr[j];
            }
            maxSum = Math.max(maxSum, sum);
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        int k = 3;
        System.out.println("Max sum: " + maxSumSubarray(arr, k)); // Output: 18
    }
}
```

✅ **Time Complexity:** O(n²) → **Inefficient for large inputs**

Optimized Sliding Window Approach (O(n))

Instead of recalculating sums from scratch, we:

- **Initialize** the sum of the first `k` elements.
- **Slide the window** by adding the new element and removing the old one.

Java Implementation

```
class SlidingWindowFixed {
    public static int maxSumSubarray(int[] arr, int k) {
        int n = arr.length;
        int maxSum = 0, windowSum = 0;

        // Compute sum of first 'k' elements
        for (int i = 0; i < k; i++) {
            windowSum += arr[i];
        }
        maxSum = windowSum;

        // Slide the window
        for (int i = k; i < n; i++) {
            windowSum += arr[i] - arr[i - k]; // Add new element, remove old one
            maxSum = Math.max(maxSum, windowSum);
```

```
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        int k = 3;
        System.out.println("Max sum: " + maxSumSubarray(arr, k)); // Output: 18
    }
}
```

✅ **Time Complexity:** O(n) → **Optimized!**

2️⃣ Variable-Size Sliding Window

Problem:

Find the **smallest subarray with a sum ≥ target**.

Brute Force Approach (O(n²))

Check all subarrays and return the minimum length.

```
class BruteForceVariable {
    public static int minSubarrayLen(int[] arr, int target) {
        int n = arr.length, minLen = Integer.MAX_VALUE;

        for (int i = 0; i < n; i++) {
            int sum = 0;
            for (int j = i; j < n; j++) {
                sum += arr[j];
                if (sum >= target) {
                    minLen = Math.min(minLen, j - i + 1);
                    break; // No need to check longer subarrays
                }
            }
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 1, 2, 4, 3};
        int target = 7;
        System.out.println("Min subarray length: " + minSubarrayLen(arr, target)); // Output: 2
    }
}
```

✅ **Time Complexity:** O(n²) → **Too slow for large inputs**

Optimized Sliding Window Approach (O(n))

- **Expand** the window by adding elements until sum ≥ target.
- **Shrink** the window from the left to find the **minimum length**.

Java Implementation

```
class SlidingWindowVariable {
    public static int minSubarrayLen(int[] arr, int target) {
        int minLen = Integer.MAX_VALUE, sum = 0, left = 0;

        for (int right = 0; right < arr.length; right++) {
            sum += arr[right]; // Expand window by adding element

            while (sum >= target) { // Shrink window when condition is met
                minLen = Math.min(minLen, right - left + 1);
                sum -= arr[left]; // Remove element from left
                left++; // Move left pointer
            }
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 1, 2, 4, 3};
        int target = 7;
        System.out.println("Min subarray length: " + minSubarrayLen(arr, target)); // Output: 2
    }
}
```

✅ **Time Complexity:** O(n)
✅ **Space Complexity:** O(1) (No extra space used)

---

3 Sliding Window in Strings

Problem:

Find the **longest substring without repeating characters**.

Approach:

- Expand window while characters are unique.
- If a duplicate appears, shrink the window from the left.

Java Implementation

```
import java.util.HashSet;

class LongestSubstring {
    public static int longestUniqueSubstring(String s) {
        HashSet<Character> set = new HashSet<>();
        int left = 0, maxLength = 0;

        for (int right = 0; right < s.length(); right++) {
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left)); // Shrink window
                left++;
            }
            set.add(s.charAt(right)); // Expand window
            maxLength = Math.max(maxLength, right - left + 1);
        }
        return maxLength;
    }

    public static void main(String[] args) {
        String s = "abcabcbb";
        System.out.println("Longest substring length: " + longestUniqueSubstring(s)); // Output: 3
    }
}
```

✅ **Time Complexity:** O(n)
✅ **Space Complexity:** O(min(n, 26)) (Alphabet size limit)

Summary

When to Use Sliding Window?

✅ **Works best when dealing with contiguous subarrays or substrings.**
✅ **Helps reduce nested loops (O(n²)) to O(n)**.
✅ **Commonly used for sum, max, min, or unique character problems.**

Let me know if you need more explanations or examples!

Matrix in DSA (Data Structures & Algorithms)

A **matrix** is a **2D array** (grid-like structure) commonly used in **graphs, dynamic programming, image processing, and pathfinding algorithms**. Understanding how to traverse and manipulate matrices efficiently is essential for solving many coding problems.

---

1 Matrix Representation and Traversal

Declaration & Initialization

A matrix is represented as a **2D array** in Java.

```
// 3x3 Matrix (2D Array)
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Accessing elements
System.out.println(matrix[0][1]); // Output: 2
```

Accessing any element takes **O(1) time**, but most operations require **O(n × m) complexity**.

Row-wise Traversal

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[0].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

Column-wise Traversal

```
for (int j = 0; j < matrix[0].length; j++) {
    for (int i = 0; i < matrix.length; i++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

Both traversals have **O(n × m) complexity**.

2️⃣ Transpose of a Matrix

Problem: Convert **rows into columns**.

Example Input:

```
1 2 3
4 5 6
7 8 9
```

Example Output (Transpose):

```
1 4 7
2 5 8
3 6 9
```

Java Code:

```
class TransposeMatrix {
    public static int[][] transpose(int[][] matrix) {
        int rows = matrix.length, cols = matrix[0].length;
        int[][] transposed = new int[cols][rows];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                transposed[j][i] = matrix[i][j]; // Swap rows and columns
            }
        }
        return transposed;
    }
}
```

This approach runs in **O(n × m) time** and uses extra **O(n × m) space**.

3️⃣ Rotate a Matrix by 90 Degrees

Problem: Rotate a square matrix **clockwise by 90°**.

Example Input:

```
1 2 3
4 5 6
7 8 9
```

Example Output:

```
7 4 1
```

```
8 5 2
9 6 3
```

Approach:

1. **Transpose the matrix** (swap matrix[i][j] with matrix[j][i]).
2. **Reverse each row** to get a 90° rotation.

Java Code:

```
class RotateMatrix {
    public static void rotate90(int[][] matrix) {
        int n = matrix.length;

        // Step 1: Transpose the matrix
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }

        // Step 2: Reverse each row
        for (int i = 0; i < n; i++) {
            for (int j = 0, k = n - 1; j < k; j++, k--) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[i][k];
                matrix[i][k] = temp;
            }
        }
    }
}
```

This method runs in **O(n²) time** and uses **O(1) extra space**.

---

4 Spiral Matrix Traversal

Problem: Print the elements in **spiral order**.

Example Input:

```
1 2 3
4 5 6
7 8 9
```

Example Output:

```
1 2 3 6 9 8 7 4 5
```

Approach:

1. Move **left to right**.
2. Move **top to bottom**.
3. Move **right to left**.
4. Move **bottom to top**, and repeat.

Java Code:

```
class SpiralMatrix {
    public static void spiralOrder(int[][] matrix) {
        int top = 0, bottom = matrix.length - 1;
        int left = 0, right = matrix[0].length - 1;

        while (top <= bottom && left <= right) {
            // Left to Right
            for (int i = left; i <= right; i++) System.out.print(matrix[top][i] + " ");
            top++;

            // Top to Bottom
            for (int i = top; i <= bottom; i++) System.out.print(matrix[i][right] + " ");
            right--;

            // Right to Left
            if (top <= bottom) {
                for (int i = right; i >= left; i--) System.out.print(matrix[bottom][i] + " ");
                bottom--;
            }

            // Bottom to Top
            if (left <= right) {
                for (int i = bottom; i >= top; i--) System.out.print(matrix[i][left] + " ");
                left++;
            }
        }
    }
}
```

This algorithm runs in **O(n × m) time** and requires **O(1) space**.

---

5 When to Use Matrices in DSA?

- **Graphs** → Matrices are used in **adjacency matrices** for graph representation.
- **Dynamic Programming** → Many **grid-based DP problems** use matrices (e.g., `dp[i][j]`).
- **Image Processing** → Used for **image transformations, convolution operations**.
- **Game Development** → Used in **pathfinding algorithms (BFS, DFS) on grids**.

Final Thoughts

- Matrix problems are common in **coding interviews**, especially for **graph-based, dynamic programming, and traversal problems**.
- Many problems can be solved **brute-force ($O(n^2)$)**, but can often be optimized using **better traversal techniques**.
- Understanding **matrix properties (diagonals, rows, columns)** helps in solving problems efficiently.

Let me know if you need **more explanations or additional problems**!

HashMap in DSA (Data Structures & Algorithms)

A **HashMap** is a **key-value pair** data structure that allows for **fast lookups, insertions, and deletions**. It is widely used in **problem-solving, caching, frequency counting, and graph algorithms**.

---

1 How Does a HashMap Work?

A **HashMap** uses a **hash function** to map keys to an **index (bucket)** in an internal array.

1. The **hash function** generates an index based on the key.
2. If multiple keys generate the same index (**collision**), they are handled using **chaining (LinkedList)** or **open addressing**.
3. Operations like **get(), put(), and remove()** work in **O(1) average time complexity**.

---

2 Declaring and Using a HashMap in Java

Creating a HashMap

```
import java.util.*;
```

```
class HashMapExample {
    public static void main(String[] args) {
        // Declare a HashMap
        HashMap<String, Integer> map = new HashMap<>();

        // Insert key-value pairs
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Get a value
        System.out.println("Alice's age: " + map.get("Alice")); // Output: 25

        // Check if a key exists
        System.out.println("Contains Bob? " + map.containsKey("Bob")); // Output: true

        // Remove a key
        map.remove("Charlie");

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

Key Points:

- put(key, value) → Inserts an element.
- get(key) → Retrieves the value for the key.
- remove(key) → Deletes a key.
- containsKey(key) → Checks if a key exists.
- size() → Returns the number of elements.

---

3 Common Use Cases of HashMap in DSA

1. Frequency Counting (Character or Word Frequency)

Problem: Count the frequency of characters in a string.

Example Input: "hello"

Example Output: {h=1, e=1, l=2, o=1}

Java Code:

```
import java.util.*;

class CharacterFrequency {
    public static void main(String[] args) {
        String str = "hello";
        HashMap<Character, Integer> freqMap = new HashMap<>();
```

```
        for (char c : str.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        System.out.println(freqMap);
    }
}
```

This solution runs in **O(n) time** and is useful in **anagram problems**.

---

2. Two Sum Problem (Pair Sum using HashMap)

Problem: Find two numbers in an array that sum to a target value.

Example Input: arr = [2, 7, 11, 15], target = 9

Example Output: [0,1] (Indices of 2 and 7)

Java Code:

```
import java.util.*;

class TwoSum {
    public static int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];

            if (map.containsKey(complement)) {
                return new int[]{map.get(complement), i};
            }

            map.put(nums[i], i);
        }
        return new int[]{-1, -1}; // No solution found
    }

    public static void main(String[] args) {
        int[] nums = {2, 7, 11, 15};
        int target = 9;
        int[] result = twoSum(nums, target);
        System.out.println(Arrays.toString(result));
    }
}
```

✅ **O(n) time complexity** (instead of O(n²) for brute force).
✅ **Efficient for large input sizes**.

---

3. Finding Duplicates in an Array

Problem: Check if an array contains duplicates.

Example Input: [1, 2, 3, 4, 2]

Example Output: true

Java Code:

```
import java.util.*;

class ContainsDuplicate {
    public static boolean hasDuplicate(int[] nums) {
        HashSet<Integer> set = new HashSet<>();

        for (int num : nums) {
            if (set.contains(num)) {
                return true; // Duplicate found
            }
            set.add(num);
        }
        return false; // No duplicates
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3, 4, 2};
        System.out.println(hasDuplicate(nums)); // Output: true
    }
}
```

✅ **O(n) time complexity**.
✅ Uses **HashSet** instead of HashMap (faster for checking duplicates).

---

4. Longest Substring Without Repeating Characters

Problem: Find the length of the longest substring without repeating characters.

Example Input: "abcabcbb"

Example Output: 3 (Substring: "abc")

Java Code:

```
import java.util.*;

class LongestUniqueSubstring {
    public static int lengthOfLongestSubstring(String s) {
        HashMap<Character, Integer> map = new HashMap<>();
        int maxLength = 0, left = 0;

        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);

            if (map.containsKey(c)) {
```

```
                left = Math.max(map.get(c) + 1, left);
            }

            map.put(c, right);
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }

    public static void main(String[] args) {
        System.out.println(lengthOfLongestSubstring("abcabcbb")); // Output: 3
    }
}
```

✅ Uses **Sliding Window + HashMap**.
✅ Runs in **O(n) time complexity**.

4️⃣ HashMap Time Complexity Analysis

- **Insertion (`put()`)** → **O(1)** on average, **O(n)** in worst case (if many collisions occur).
- **Deletion (`remove()`)** → **O(1)** on average.
- **Lookup (`get()`)** → **O(1)** on average, **O(n)** in worst case.
- **Iteration over all elements** → **O(n)**.

HashMap is **not sorted**, but **TreeMap** (O(log n) complexity) maintains order.

5️⃣ When to Use HashMap in DSA?

- **Frequency counting** (words, characters, numbers).
- **Fast lookups** in problems like **Two Sum, Anagrams, Duplicates detection**.
- **Dynamic programming optimizations** (memoization with HashMap).
- **Graph algorithms** (adjacency list representation using `HashMap<Integer, List<Integer>>`).

Final Thoughts

- **HashMaps** are extremely useful for solving **many coding problems in O(1) time**.
- **Handling collisions** properly is key to **efficient performance**.

- Always **check if a key exists** before calling `get()` to avoid `NullPointerException`.

Let me know if you need **more explanations or additional problems**!

Intervals in DSA (Data Structures & Algorithms)

Intervals are commonly used in **range-based problems**, such as **merging overlapping intervals, scheduling, and segment intersections**. Efficient handling of intervals is crucial for solving **sorting, greedy algorithms, and sweep-line technique** problems.

---

1 What is an Interval?

An **interval** is a range represented as **[start, end]**, where:

- `start` → Beginning of the interval
- `end` → End of the interval

For example, the intervals **[1, 5]** and **[3, 7]** overlap, while **[1, 3]** and **[4, 6]** do not.

---

2 Common Operations on Intervals

1. Sorting Intervals

Sorting intervals is **often the first step** before merging or processing them.

Sorting by Start Time (Ascending Order)

```
import java.util.*;

class IntervalSorting {
    public static void main(String[] args) {
        int[][] intervals = { {5, 10}, {1, 3}, {8, 12} };

        Arrays.sort(intervals, (a, b) -> a[0] - b[0]); // Sort by start time
```

```
        for (int[] interval : intervals) {
            System.out.println(Arrays.toString(interval));
        }
    }
}
```

✅ **Time Complexity: O(n log n)** due to sorting.

---

2. Merging Overlapping Intervals

Problem: Given overlapping intervals, merge them into one.

Example:

Input: [[1,3], [2,6], [8,10], [15,18]]
Output: [[1,6], [8,10], [15,18]]

Approach:

1. **Sort intervals** by start time.
2. **Compare overlapping intervals** and merge them.

Java Code:

```
import java.util.*;

class MergeIntervals {
    public static int[][] merge(int[][] intervals) {
        if (intervals.length <= 1) return intervals;

        Arrays.sort(intervals, (a, b) -> a[0] - b[0]); // Sort by start time
        List<int[]> merged = new ArrayList<>();

        int[] current = intervals[0];

        for (int i = 1; i < intervals.length; i++) {
            if (current[1] >= intervals[i][0]) { // Overlap detected
                current[1] = Math.max(current[1], intervals[i][1]);
            } else {
                merged.add(current);
                current = intervals[i];
            }
        }

        merged.add(current); // Add the last interval
        return merged.toArray(new int[merged.size()][]);
    }

    public static void main(String[] args) {
        int[][] intervals = {{1,3}, {2,6}, {8,10}, {15,18}};
```

```
        int[][] merged = merge(intervals);

        for (int[] interval : merged) {
            System.out.println(Arrays.toString(interval));
        }
    }
}
```

✅ **Time Complexity: O(n log n)** (due to sorting)
✅ **Space Complexity: O(n)** (for storing merged intervals)

---

3. Insert a New Interval

Problem: Given non-overlapping sorted intervals, insert a new interval and merge if necessary.

Example:

Input: intervals = [[1,3], [6,9]], newInterval = [2,5]
Output: [[1,5], [6,9]]

Java Code:

```
import java.util.*;

class InsertInterval {
    public static int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> result = new ArrayList<>();
        int i = 0, n = intervals.length;

        // Add all intervals before the new interval
        while (i < n && intervals[i][1] < newInterval[0]) {
            result.add(intervals[i]);
            i++;
        }

        // Merge overlapping intervals
        while (i < n && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
            newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
            i++;
        }
        result.add(newInterval);

        // Add remaining intervals
        while (i < n) {
            result.add(intervals[i]);
            i++;
        }
```

```
        return result.toArray(new int[result.size()][]);
    }

    public static void main(String[] args) {
        int[][] intervals = {{1,3}, {6,9}};
        int[] newInterval = {2,5};
        int[][] result = insert(intervals, newInterval);

        for (int[] interval : result) {
            System.out.println(Arrays.toString(interval));
        }
    }
}
```

✅ **Time Complexity: O(n)**
✅ **Space Complexity: O(n)**

---

4. Find the Intersection of Two Interval Lists

Problem: Given two lists of intervals, find their intersection.

Example:

Input:

```
A = [[1,3],[5,6],[7,9]]
B = [[2,5],[6,8]]
```

Output:

```
[[2,3], [5,5], [6,6], [7,8]]
```

Java Code:

```
import java.util.*;

class IntervalIntersection {
    public static int[][] intervalIntersection(int[][] A, int[][] B) {
        List<int[]> result = new ArrayList<>();
        int i = 0, j = 0;

        while (i < A.length && j < B.length) {
            int start = Math.max(A[i][0], B[j][0]);
            int end = Math.min(A[i][1], B[j][1]);

            if (start <= end) {
                result.add(new int[]{start, end});
            }

            if (A[i][1] < B[j][1]) {
                i++;
```

```
            } else {
                j++;
            }
        }

        return result.toArray(new int[result.size()][]);
    }

    public static void main(String[] args) {
        int[][] A = {{1,3},{5,6},{7,9}};
        int[][] B = {{2,5},{6,8}};
        int[][] result = intervalIntersection(A, B);

        for (int[] interval : result) {
            System.out.println(Arrays.toString(interval));
        }
    }
}
```

✅ **Time Complexity: O(n + m)** (two-pointer technique)
✅ **Space Complexity: O(n + m)**

3️⃣ When to Use Intervals in DSA?

- **Meeting room scheduling** (e.g., check if a person can attend all meetings).
- **Merging time slots** (e.g., merging overlapping bookings).
- **Finding free time slots** in schedules.
- **Finding intersections** in sorted data.

4️⃣ Final Thoughts

- **Sorting intervals** is often a prerequisite for interval problems.
- **Two-pointer or Greedy approach** helps in merging and intersection problems.
- **Use sorting + iteration for optimal solutions** instead of brute force approaches.

Let me know if you need **more explanations or additional problems**!

Stack in DSA (Data Structures & Algorithms)

A **Stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. It is widely used in **recursive algorithms, expression evaluation, backtracking, and depth-first search (DFS)**.

---

1 How Does a Stack Work?

A stack operates with two main operations:

- **Push(x)** → Inserts x at the top of the stack.
- **Pop()** → Removes the top element of the stack.

Other useful operations include:

- **Peek()** → Returns the top element without removing it.
- **isEmpty()** → Checks if the stack is empty.
- **size()** → Returns the number of elements.

Stack Example

Stack: [10, 20, 30] (Top: 30)
Push(40) → [10, 20, 30, 40]
Pop() → [10, 20, 30] (Returns 40)

---

2 Implementing Stack in Java

1. Using Java's Built-in Stack Class

Java provides a built-in Stack class in java.util package.

```
import java.util.*;

class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Stack: " + stack); // Output: [10, 20, 30]

        System.out.println("Top Element: " + stack.peek()); // Output: 30

        stack.pop();
```

```
        System.out.println("Stack after pop: " + stack); // Output: [10, 20]

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

✅ **Time Complexity:** push(), pop(), and peek() → **O(1)**

---

2. Implementing Stack Using Array

```
class StackUsingArray {
    private int[] stack;
    private int top;
    private int capacity;

    public StackUsingArray(int size) {
        stack = new int[size];
        capacity = size;
        top = -1;
    }

    public void push(int x) {
        if (top == capacity - 1) {
            System.out.println("Stack Overflow");
            return;
        }
        stack[++top] = x;
    }

    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return -1;
        }
        return stack[top--];
    }

    public int peek() {
        if (top == -1) return -1;
        return stack[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public static void main(String[] args) {
        StackUsingArray stack = new StackUsingArray(5);
        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Top Element: " + stack.peek()); // Output: 30
```

```
        stack.pop();
        System.out.println("Stack Empty? " + stack.isEmpty()); // Output: false
    }
}
```

✅ **Time Complexity:** push(), pop() → **O(1)**

---

3. Implementing Stack Using Linked List

```
class StackNode {
    int data;
    StackNode next;

    StackNode(int data) {
        this.data = data;
        this.next = null;
    }
}

class StackUsingLinkedList {
    private StackNode top;

    public void push(int x) {
        StackNode newNode = new StackNode(x);
        newNode.next = top;
        top = newNode;
    }

    public int pop() {
        if (top == null) {
            System.out.println("Stack Underflow");
            return -1;
        }
        int popped = top.data;
        top = top.next;
        return popped;
    }

    public int peek() {
        return (top != null) ? top.data : -1;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public static void main(String[] args) {
        StackUsingLinkedList stack = new StackUsingLinkedList();
        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Top Element: " + stack.peek()); // Output: 30
```

```
        stack.pop();
        System.out.println("Stack Empty? " + stack.isEmpty()); // Output: false
    }
}
```

✅ **Time Complexity:** push(), pop() → **O(1)**
✅ **Space Complexity: O(n)** (for storing linked list nodes)

3️⃣ Applications of Stack in DSA

1. Reverse a String Using Stack

Example: Input "hello", Output "olleh"

```
import java.util.*;

class ReverseStringUsingStack {
    public static String reverseString(String str) {
        Stack<Character> stack = new Stack<>();
        for (char c : str.toCharArray()) {
            stack.push(c);
        }

        StringBuilder reversed = new StringBuilder();
        while (!stack.isEmpty()) {
            reversed.append(stack.pop());
        }

        return reversed.toString();
    }

    public static void main(String[] args) {
        System.out.println(reverseString("hello")); // Output: olleh
    }
}
```

✅ **Time Complexity: O(n)**

---

2. Check for Balanced Parentheses

Example: Input "({[()]})" → **Balanced** ✅

```
import java.util.*;

class BalancedParentheses {
    public static boolean isValid(String s) {
```

```
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else {
                if (stack.isEmpty()) return false;
                char top = stack.pop();
                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) return false;
            }
        }
        return stack.isEmpty();
    }

    public static void main(String[] args) {
        System.out.println(isValid("({[()]})")); // Output: true
    }
}
```

✅ **Time Complexity: O(n)**

---

3. Next Greater Element

Example: Input [4,5,2,10] → Output [5,10,10,-1]

```
import java.util.*;

class NextGreaterElement {
    public static int[] nextGreater(int[] arr) {
        Stack<Integer> stack = new Stack<>();
        int[] result = new int[arr.length];

        for (int i = arr.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= arr[i]) {
                stack.pop();
            }
            result[i] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(arr[i]);
        }
        return result;
    }

    public static void main(String[] args) {
        int[] arr = {4, 5, 2, 10};
        System.out.println(Arrays.toString(nextGreater(arr))); // Output: [5, 10, 10, -1]
    }
}
```

✅ **Time Complexity: O(n)** (each element is pushed/popped once)

4️⃣ When to Use Stack in DSA?

- **Undo/Redo operations** in text editors.
- **Function call stack** (recursion).
- **Expression evaluation** (infix to postfix, postfix evaluation).
- **Depth-First Search (DFS)** in graphs.
- **Backtracking problems** (e.g., solving mazes, N-Queens).

5️⃣ Final Thoughts

- **Stacks use LIFO (Last In, First Out)** principle.
- **Efficient for solving problems with nested structures** (e.g., parentheses, HTML tags).
- **Can be implemented using arrays or linked lists** based on requirements.

Let me know if you need **more explanations or additional problems**!

Linked List in DSA (Data Structures & Algorithms)

A **Linked List** is a linear data structure where elements are stored in nodes, and each node points to the next node. Unlike arrays, linked lists **do not require contiguous memory allocation** and can dynamically grow or shrink.

1️⃣ Types of Linked Lists

1. **Singly Linked List** → Each node points to the next node.
2. **Doubly Linked List** → Each node has pointers to both the next and previous nodes.
3. **Circular Linked List** → The last node connects back to the first node.

2️⃣ Basic Structure of a Node

A linked list consists of nodes, where each node contains:

- **Data** → The value stored in the node.
- **Pointer** → A reference to the next node in the list.

Java Implementation of a Node

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null; // Initially, the next node is null
    }
}
```

3️⃣ Singly Linked List Implementation

A **Singly Linked List (SLL)** has nodes linked in one direction.

Java Code for Basic Operations

```
class SinglyLinkedList {
    Node head; // Head (first node) of the linked list

    // Insert at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }

    // Delete a node
    public void delete(int key) {
        if (head == null) return;

        if (head.data == key) {
            head = head.next;
            return;
        }
```

```
        Node temp = head;
        while (temp.next != null && temp.next.data != key) {
            temp = temp.next;
        }

        if (temp.next == null) return; // Key not found
        temp.next = temp.next.next;
    }

    // Print the linked list
    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList(); // Output: 10 -> 20 -> 30 -> null

        list.delete(20);
        list.printList(); // Output: 10 -> 30 -> null
    }
}
```

✅ **Time Complexity:**

- Insert: **O(n)** (at the end) or **O(1)** (at the beginning)
- Delete: **O(n)** (traversal required)

4️⃣ Doubly Linked List (DLL)

A **Doubly Linked List (DLL)** has **two pointers**:

- next → Points to the next node.
- prev → Points to the previous node.

Java Code for Doubly Linked List

```
class DoublyNode {
    int data;
    DoublyNode next, prev;

    public DoublyNode(int data) {
```

```
        this.data = data;
        this.next = this.prev = null;
    }
}

class DoublyLinkedList {
    DoublyNode head;

    // Insert at the end
    public void insert(int data) {
        DoublyNode newNode = new DoublyNode(data);
        if (head == null) {
            head = newNode;
            return;
        }
        DoublyNode temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
        newNode.prev = temp;
    }

    // Print the list
    public void printList() {
        DoublyNode temp = head;
        while (temp != null) {
            System.out.print(temp.data + " <-> ");
            temp = temp.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList(); // Output: 10 <-> 20 <-> 30 <-> null
    }
}
```

✅ **Advantages over SLL:**

- Faster deletion (**O(1)** if the node reference is known).
- Can be traversed **both forward and backward**.

5️⃣ Circular Linked List (CLL)

A **Circular Linked List** has its last node pointing to the first node, forming a loop.

Java Code for Circular Linked List

```
class CircularNode {
    int data;
    CircularNode next;

    public CircularNode(int data) {
        this.data = data;
        this.next = null;
    }
}

class CircularLinkedList {
    CircularNode head, tail;

    public void insert(int data) {
        CircularNode newNode = new CircularNode(data);
        if (head == null) {
            head = newNode;
            tail = newNode;
            newNode.next = head; // Circular link
        } else {
            tail.next = newNode;
            tail = newNode;
            tail.next = head; // Circular link
        }
    }

    public void printList() {
        if (head == null) return;
        CircularNode temp = head;
        do {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        } while (temp != head);
        System.out.println("(Back to Head)");
    }

    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList(); // Output: 10 -> 20 -> 30 -> (Back to Head)
    }
}
```

✅ Use Cases:

- **Memory-efficient queues**
- **Round-robin scheduling**

6️⃣ Applications of Linked Lists

- **Dynamic memory allocation** (better than arrays).
- **Undo/Redo** operations in text editors.
- **Implementation of stacks and queues**.
- **Graph adjacency lists** (used in DFS & BFS).
- **Hashing (Chaining for Collision Handling)**.

7️⃣ When to Use Linked List vs. Array?

✅ **Use Linked List if:** You need **frequent insertions/deletions**.
✅ **Use Array if:** You need **fast random access**.

8️⃣ Final Thoughts

- **Singly Linked List** → Best for **simple dynamic storage**.
- **Doubly Linked List** → Faster **deletions** and **bidirectional traversal**.
- **Circular Linked List** → Useful for **queues and scheduling**.

Let me know if you need **more examples, problems, or explanations!**

Binary Tree in DSA (Data Structures & Algorithms)

A **Binary Tree** is a hierarchical data structure in which each node has at most **two children**, referred to as the **left** and **right** child.

It is a foundational structure used in many algorithms and applications like expression trees, heaps, and binary search trees (BSTs).

1️⃣ Structure of a Binary Tree Node

Each node typically contains:

- A **value** or **data**
- A **left child**
- A **right child**

Java Code: Binary Tree Node

```
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        this.left = this.right = null;
    }
}
```

---

2 Types of Binary Trees

1. **Full Binary Tree**: Every node has 0 or 2 children.
2. **Perfect Binary Tree**: All levels are fully filled.
3. **Complete Binary Tree**: All levels except possibly the last are completely filled; nodes are as left as possible.
4. **Balanced Binary Tree**: The height difference between left and right subtrees is no more than 1.
5. **Degenerate Tree**: Each parent has only one child — behaves like a linked list.

---

3 Basic Tree Traversals

1. Inorder (Left, Root, Right)

```
void inorder(TreeNode root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.data + " ");
    inorder(root.right);
}
```

2. Preorder (Root, Left, Right)

```
void preorder(TreeNode root) {
    if (root == null) return;
    System.out.print(root.data + " ");
    preorder(root.left);
    preorder(root.right);
}
```

3. Postorder (Left, Right, Root)

```
void postorder(TreeNode root) {
```

```
    if (root == null) return;
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.data + " ");
}
```

4. Level Order (Breadth-First Search)

```
void levelOrder(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
        TreeNode current = queue.poll();
        System.out.print(current.data + " ");

        if (current.left != null) queue.add(current.left);
        if (current.right != null) queue.add(current.right);
    }
}
```

---

4 Example Tree and Traversal Output

Example:

```
      1
     / \
    2   3
   / \
  4   5
```

- **Inorder**: 4 2 5 1 3
- **Preorder**: 1 2 4 5 3
- **Postorder**: 4 5 2 3 1
- **Level Order**: 1 2 3 4 5

---

5 Common Binary Tree Problems

1. Height of Tree

```
int height(TreeNode root) {
    if (root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}
```

2. Count Total Nodes

```
int countNodes(TreeNode root) {
    if (root == null) return 0;
```

```
    return 1 + countNodes(root.left) + countNodes(root.right);
}
```

3. Check if Tree is Balanced

```
boolean isBalanced(TreeNode root) {
    return checkHeight(root) != -1;
}

int checkHeight(TreeNode node) {
    if (node == null) return 0;
    int left = checkHeight(node.left);
    int right = checkHeight(node.right);

    if (left == -1 || right == -1 || Math.abs(left - right) > 1)
        return -1;

    return 1 + Math.max(left, right);
}
```

---

6 Use Cases of Binary Trees

- **Hierarchical data** (e.g., file systems)
- **Expression parsing**
- **Binary Search Trees (BSTs)**
- **Heaps and Priority Queues**
- **Huffman Encoding Trees**

---

7 Advantages

- Dynamic size (unlike arrays)
- Efficient insertion/deletion (especially in BSTs)
- Fast searching (for balanced trees)

---

8 Summary

- Binary trees are **recursive** by nature.
- Traversals are key to solving problems.
- Know how to build, traverse, and manipulate trees for interviews.

Want to dive into **Binary Search Trees**, **Heaps**, or solve tree problems next?

Binary Tree BFS in DSA (Data Structures & Algorithms)

1️⃣ What is BFS in a Binary Tree?

- **BFS (Breadth-First Search)** is a way of traversing the tree **level-by-level**.
- You first visit all nodes at **depth 0 (root)**, then all nodes at **depth 1**, and so on.
- **Key Point**: We use a **Queue** to perform BFS traversal.

2️⃣ Why BFS?

- Needed when:
  - You want to **process all nodes at the same level first**.
  - Finding the **shortest path** (in unweighted graphs/trees).
  - **Level order operations** (like finding the sum of nodes at a particular level).

3️⃣ Java Code for Basic BFS (Level Order Traversal)

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

class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        this.left = this.right = null;
    }
}

public class BinaryTreeBFS {

    // Function to perform BFS traversal (Level Order Traversal)
    public static void levelOrderTraversal(TreeNode root) {
        if (root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
```

```
        while (!queue.isEmpty()) {
            TreeNode current = queue.poll();
            System.out.print(current.data + " ");

            if (current.left != null) queue.add(current.left); // add left child to queue
            if (current.right != null) queue.add(current.right); // add right child to queue
        }
    }

    public static void main(String[] args) {
        // Build a sample tree
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);

        // Call BFS
        levelOrderTraversal(root);
    }
}
```

Output:

1 2 3 4 5

✅ Traverses nodes **level-by-level** from top to bottom and left to right.

4️⃣ BFS with Levels Separated

Sometimes we want to **print nodes level-wise**, one level per line.

```
public static void bfsWithLevels(TreeNode root) {
    if (root == null) return;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
        int size = queue.size(); // Number of nodes at current level

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            System.out.print(current.data + " ");

            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }
        System.out.println(); // Move to next line after printing one level
    }
```

}

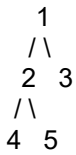
5️⃣ Important Variations of BFS Problems

Here are some very popular DSA problems based on BFS on binary trees:

6️⃣ BFS vs DFS in Trees

7️⃣ Quick Example

Tree:



BFS Traversal Output:

1 2 3 4 5

Summary

- BFS = **Level-order traversal** using a **Queue**.
- Useful for **level-based** problems.
- Common variations include **Zigzag traversal**, **Bottom-up traversal**, **Finding level with max sum**, etc.

Would you also like me to show **some real interview problems** based on **BFS on trees**?
(Like LeetCode 102, 103, 107)
They are really good for practice!

Of course!
Here's a **list of important variations** of **BFS on Binary Trees** you should definitely know for DSA and interviews:

Important BFS Variations in Trees

1. **Level Order Traversal**
   - Traverse all nodes level by level.
2. **Level Order Traversal with Level Separation**
   - Print each level on a new line.
3. **Level Order Traversal in Reverse (Bottom-Up)**
   - Start traversal from the bottom level to the top.
4. **Zigzag (Spiral) Level Order Traversal**
   - Alternate levels: left-to-right, then right-to-left.
5. **Find Maximum Width of Binary Tree**
   - Maximum number of nodes at any level.
6. **Minimum Depth of Binary Tree**
   - Find the shortest path from root to any leaf node.
7. **Right View of Binary Tree**
   - Nodes visible when tree is viewed from the right side.
8. **Left View of Binary Tree**
   - Nodes visible when tree is viewed from the left side.
9. **Top View of Binary Tree**
   - Nodes visible when viewed from the top.
10. **Bottom View of Binary Tree**
    - Nodes visible when viewed from the bottom.
11. **Sum of Nodes at K-th Level**
    - Find the sum of all nodes at a given level K.
12. **Cousins in Binary Tree**
    - Find if two nodes are cousins (same level, different parents).
13. **Check Completeness of a Binary Tree**
    - Check if a tree is a complete binary tree using BFS.
14. **Serialize and Deserialize a Binary Tree**
    - Convert tree to string and back (commonly using BFS).
15. **Distance K in Binary Tree**
    - Find all nodes at distance K from a target node.
16. **Connecting Nodes at Same Level (Next Pointer)**
    - Populate each node's next pointer to its right neighbor node.

Bonus

Some **graphs** problems also extend BFS ideas like:

- Shortest path in an unweighted graph.

- Number of connected components.

Would you like me to also give **short Java templates** for these common variations? (Helps in fast revision during interviews!)
Let me know!

Below is a more complete and clear explanation of a Binary Search Tree (BST) along with well-commented Java code.

Binary Search Tree (BST) Overview

A **Binary Search Tree (BST)** is a binary tree with an ordering property:

- For each node, all values in its **left subtree** are **less than** the node's value.
- All values in its **right subtree** are **greater than** the node's value.

This property makes BSTs efficient for operations like searching, insertion, and deletion. For balanced trees, these operations can run in O(log n) on average.

Key Operations in a BST

1. **Insertion:**
   Insert a new value by comparing it to existing nodes and placing it in the correct position.
2. **Search:**
   Look for a value by comparing the value with the current node and traversing left (if the value is smaller) or right (if larger).
3. **Deletion:**
   Remove a node from the BST.
   - If the node is a leaf, simply remove it.
   - If the node has one child, replace it with its child.
   - If the node has two children, replace it with its inorder successor (smallest in the right subtree) or predecessor.
4. **Traversals:**
   Inorder traversal of a BST returns the values in **sorted order**.

Complete Java Code Example

Below is a complete Java program that defines a BST with methods for insertion, search, deletion, and inorder traversal. Each method includes clear comments.

```
// Define the structure of each node in the Binary Search Tree.
class TreeNode {
    int data;
    TreeNode left, right;

    // Constructor for TreeNode
    TreeNode(int data) {
        this.data = data;
        this.left = this.right = null;
    }
}

// Define the BST class with various operations.
public class BinarySearchTree {

    // Root of the BST.
    TreeNode root;

    // Constructor to initialize BST.
    public BinarySearchTree() {
        root = null;
    }

    // Insert a new value into the BST.
    public TreeNode insert(TreeNode root, int val) {
        // Base case: found the correct spot for insertion.
        if (root == null) {
            return new TreeNode(val);
        }
        // If val is less than current node, insert in the left subtree.
        if (val < root.data) {
            root.left = insert(root.left, val);
        }
        // Otherwise, insert in the right subtree.
        else {
            root.right = insert(root.right, val);
        }
        return root;
    }

    // Wrapper for insertion that updates the BST root.
    public void insert(int val) {
        root = insert(root, val);
    }

    // Search for a value in the BST. Returns true if found, false otherwise.
    public boolean search(TreeNode root, int key) {
```

```
    if (root == null) {
        return false;
    }
    if (root.data == key) {
        return true;
    }
    // Check left or right subtree based on key comparison.
    return key < root.data
            ? search(root.left, key)
            : search(root.right, key);
}

// Delete a node with the given key from the BST.
public TreeNode delete(TreeNode root, int key) {
    if (root == null) {
        return null;
    }

    if (key < root.data) {
        // Key lies in the left subtree.
        root.left = delete(root.left, key);
    } else if (key > root.data) {
        // Key lies in the right subtree.
        root.right = delete(root.right, key);
    } else {
        // Found the node to be deleted.

        // Case 1: Node with only one child or no child.
        if (root.left == null) {
            return root.right;
        } else if (root.right == null) {
            return root.left;
        }

        // Case 2: Node with two children.
        // Find the inorder successor (smallest in the right subtree).
        root.data = findMin(root.right);
        // Delete the inorder successor.
        root.right = delete(root.right, root.data);
    }
    return root;
}

// Helper method to find the minimum value in a subtree.
public int findMin(TreeNode root) {
    while (root.left != null) {
        root = root.left;
    }
    return root.data;
}

// Inorder traversal: Left, Root, Right.
// For a BST, this returns values in sorted order.
public void inorder(TreeNode root) {
    if (root != null) {
        inorder(root.left);
```

```
            System.out.print(root.data + " ");
            inorder(root.right);
        }
    }

    // Main method to run and test the BST.
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        // Insertion of values.
        bst.insert(8);
        bst.insert(3);
        bst.insert(10);
        bst.insert(1);
        bst.insert(6);
        bst.insert(14);
        bst.insert(4);
        bst.insert(7);

        System.out.println("Inorder Traversal (Sorted Order):");
        bst.inorder(bst.root);
        // Expected Output: 1 3 4 6 7 8 10 14

        // Searching for a value.
        int key = 6;
        if (bst.search(bst.root, key)) {
            System.out.println("\nValue " + key + " found in BST.");
        } else {
            System.out.println("\nValue " + key + " NOT found in BST.");
        }

        // Deleting a value.
        bst.root = bst.delete(bst.root, 3);
        System.out.println("Inorder Traversal after deleting 3:");
        bst.inorder(bst.root);
        // Expected Output: The tree without the node containing value 3,
        // and the remaining values should be in sorted order.
    }
}
```

Explanation of the Code

1. **TreeNode Class:**
   Each node stores an integer and has pointers to left and right children.
2. **Insertion:**
   The `insert` method recursively finds the correct location to place the new node while maintaining the BST property.
3. **Search:**
   The `search` method recursively checks whether a given key exists in the BST.
4. **Deletion:**
   The `delete` method handles three cases:
   - Node with no child (leaf).

    - Node with one child.
    - Node with two children (using the inorder successor to replace the node).
5. **Traversal:**
   The `inorder` method recursively traverses the tree. For a BST, the inorder traversal yields a sorted sequence.
6. **Main Method:**
   Demonstrates insertion, search, and deletion while printing the inorder traversal before and after deletion.

This complete example should help clarify how to build and manipulate a BST. Let me know if you need further details or explanations on specific aspects like balanced BSTs, more traversal methods, or advanced BST problems!

Graph in DSA (Data Structures and Algorithms)

---

1 What is a Graph?

A **graph** is a data structure that consists of:

- A set of **nodes** (also called **vertices**).
- A set of **edges** that connect pairs of nodes.

Graphs are used to model real-world problems like:

- Social networks
- Maps and navigation
- Network routing
- Web crawling
- Scheduling and planning

---

2 Types of Graphs

- **Directed vs Undirected**
    - Directed: Edges have direction (A → B)
    - Undirected: Edges go both ways (A — B)
- **Weighted vs Unweighted**
    - Weighted: Each edge has a weight/cost
    - Unweighted: All edges are equal
- **Cyclic vs Acyclic**

    - Cyclic: Graph contains at least one cycle
    - Acyclic: No cycles (like trees or DAGs)
- **Connected vs Disconnected**
    - Connected: Every node is reachable
    - Disconnected: Some nodes are isolated

---

3 Graph Representation in Code

Adjacency List (Efficient & Preferred)

```
Map<Integer, List<Integer>> graph = new HashMap<>();

// Adding an edge (u — v)
graph.computeIfAbsent(u, k -> new ArrayList<>()).add(v);
graph.computeIfAbsent(v, k -> new ArrayList<>()).add(u); // Only for undirected
```

Adjacency Matrix (2D Array)

```
int[][] matrix = new int[n][n]; // For n nodes
matrix[u][v] = 1; // Edge from u to v
matrix[v][u] = 1; // Only if undirected
```

---

4 Basic Graph Traversal Algorithms

Breadth-First Search (BFS) – Level Order (Queue)

```
void bfs(int start, Map<Integer, List<Integer>> graph) {
    Queue<Integer> queue = new LinkedList<>();
    Set<Integer> visited = new HashSet<>();

    queue.add(start);
    visited.add(start);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");

        for (int neighbor : graph.get(node)) {
            if (!visited.contains(neighbor)) {
                queue.add(neighbor);
                visited.add(neighbor);
            }
        }
    }
}
```

Depth-First Search (DFS) – Recursive (Stack)

```
void dfs(int node, Map<Integer, List<Integer>> graph, Set<Integer> visited) {
    if (visited.contains(node)) return;
```

```
    visited.add(node);
    System.out.print(node + " ");

    for (int neighbor : graph.get(node)) {
        dfs(neighbor, graph, visited);
    }
}
```

---

5 Common Graph Problems in DSA

1. **Detect Cycle (Directed/Undirected)**
2. **Find Connected Components**
3. **Topological Sort (for DAG)**
4. **Shortest Path (BFS for Unweighted, Dijkstra for Weighted)**
5. **Minimum Spanning Tree (Prim's, Kruskal's)**
6. **Flood Fill (Matrix as Graph)**
7. **Clone Graph**
8. **Course Schedule (Cycle Detection in Directed Graph)**
9. **Word Ladder (Shortest Path in Word Graph)**
10. **Island Count (DFS/BFS on matrix)**

---

6 When to Use Graphs in DSA

Use graphs when you deal with:

- Networks
- Dependencies
- Relationships between entities
- Shortest paths or connectivity
- Scheduling tasks or detecting cycles

---

7 Time Complexity (Adjacency List)

Where V is the number of vertices and E is the number of edges.

Let me know if you'd like implementations of specific graph problems like **Dijkstra**, **Kruskal**, or **Topological Sort** next!

Finding a Cycle in a Graph (DSA)

Detecting cycles in a graph is a common problem, especially when dealing with:

- **Dependency graphs** (like task scheduling)
- **Network routing**
- **Social networks**

The approach to cycle detection varies depending on whether the graph is **directed** or **undirected**.

---

1 Cycle Detection in Undirected Graph

Approach: Using DFS (Depth-First Search)

- An undirected graph contains a cycle if **DFS finds a back edge** (an edge that points to an ancestor in the current path).
- Maintain a **visited set** and **parent pointer** to keep track of the previous node.

Java Code: Cycle Detection in an Undirected Graph

```
import java.util.*;

public class UndirectedCycleDetection {

    // Check if there is a cycle in the undirected graph
    public static boolean isCyclic(Map<Integer, List<Integer>> graph, int start, Set<Integer> visited, int
parent) {
        visited.add(start); // Mark the current node as visited

        for (int neighbor : graph.getOrDefault(start, new ArrayList<>())) {
            if (!visited.contains(neighbor)) {
                // If not visited, perform DFS recursively
                if (isCyclic(graph, neighbor, visited, start)) {
                    return true;
                }
            } else if (neighbor != parent) {
                // A visited neighbor which is not the parent indicates a cycle
                return true;
            }
        }
        return false;
    }
```

```
    // Utility to detect cycle in the whole graph
    public static boolean hasCycle(Map<Integer, List<Integer>> graph, int vertices) {
        Set<Integer> visited = new HashSet<>();

        // Check for cycle in every connected component
        for (int vertex = 0; vertex < vertices; vertex++) {
            if (!visited.contains(vertex)) {
                if (isCyclic(graph, vertex, visited, -1)) {
                    return true;
                }
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();

        // Adding edges (Undirected Graph)
        graph.computeIfAbsent(0, k -> new ArrayList<>()).add(1);
        graph.computeIfAbsent(1, k -> new ArrayList<>()).add(0);
        graph.computeIfAbsent(1, k -> new ArrayList<>()).add(2);
        graph.computeIfAbsent(2, k -> new ArrayList<>()).add(1);
        graph.computeIfAbsent(2, k -> new ArrayList<>()).add(3);
        graph.computeIfAbsent(3, k -> new ArrayList<>()).add(2);
        graph.computeIfAbsent(3, k -> new ArrayList<>()).add(0);
        graph.computeIfAbsent(0, k -> new ArrayList<>()).add(3);

        int vertices = 4;
        System.out.println("Cycle detected: " + hasCycle(graph, vertices)); // Output: true
    }
}
```

Explanation:

- Uses **DFS** to traverse the graph.
- Tracks the **visited nodes** and their **parent** to detect cycles.
- If a **visited node** is encountered which is **not the parent**, it means a cycle exists.

---

2 Cycle Detection in Directed Graph

Approach: Using DFS with Recursion Stack

- A directed graph contains a cycle if there is a **back edge** (an edge pointing to a node that is currently in the recursion stack).
- Use two sets:
    - **visited**: Tracks visited nodes.
    - **recStack**: Tracks nodes in the current recursive call stack.

Java Code: Cycle Detection in a Directed Graph

```
import java.util.*;

public class DirectedCycleDetection {

    // Check if a cycle exists using DFS and recursion stack
    public static boolean isCyclic(int node, Map<Integer, List<Integer>> graph, Set<Integer> visited,
Set<Integer> recStack) {
        if (recStack.contains(node)) return true; // Cycle found
        if (visited.contains(node)) return false;

        visited.add(node);
        recStack.add(node);

        // Check all neighbors
        for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {
            if (isCyclic(neighbor, graph, visited, recStack)) {
                return true;
            }
        }

        recStack.remove(node); // Remove from recursion stack
        return false;
    }

    // Utility to check for cycles in the entire graph
    public static boolean hasCycle(Map<Integer, List<Integer>> graph, int vertices) {
        Set<Integer> visited = new HashSet<>();
        Set<Integer> recStack = new HashSet<>();

        for (int vertex = 0; vertex < vertices; vertex++) {
            if (isCyclic(vertex, graph, visited, recStack)) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();

        // Directed edges
        graph.computeIfAbsent(0, k -> new ArrayList<>()).add(1);
        graph.computeIfAbsent(1, k -> new ArrayList<>()).add(2);
        graph.computeIfAbsent(2, k -> new ArrayList<>()).add(3);
        graph.computeIfAbsent(3, k -> new ArrayList<>()).add(1); // Back edge creating a cycle

        int vertices = 4;
        System.out.println("Cycle detected: " + hasCycle(graph, vertices)); // Output: true
    }
}
```

Explanation:

- Uses **DFS** to explore nodes and maintain a **recursion stack**.
- If a node is found in the **recursion stack** during traversal, a **cycle exists**.

---

3 Comparison: Directed vs Undirected Cycle Detection

Feature	Undirected Graph	Directed Graph
Algorithm	DFS with parent tracking	DFS with recursion stack
Complexity	O(V + E)	O(V + E)
Detects which cycles	Any cycle	Cycles with back edges only
Use Case	Bi-directional relationships	One-way dependencies

---

4 Real-Life Applications

- **Dependency Management:** Finding circular dependencies (e.g., software modules).
- **Deadlock Detection:** In operating systems to detect circular waits.
- **Network Routing:** Detecting loops in data transmission.
- **Course Prerequisites:** Finding cycles in course schedules.

Summary:

- Use **DFS** for cycle detection in both **directed** and **undirected** graphs.
- Handle directed and undirected graphs differently due to their structural differences.
- Cycle detection is crucial for identifying **loops, deadlocks, and circular dependencies**.

Would you like examples of real-world problems using these cycle detection methods?

Graph BFS (Breadth-First Search) in DSA

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores all the nodes at the **present depth** before moving on to nodes at the **next depth level**. It is especially useful for **finding the shortest path** in an unweighted graph.

Key Characteristics of BFS:

1. **Level-Order Traversal:** Visits nodes level by level.
2. **Queue-Based:** Uses a queue to maintain the current level nodes.
3. **Undirected/Directed Graphs:** Works with both.
4. **Time Complexity:** O(V + E), where V = vertices and E = edges.
5. **Space Complexity:** O(V) due to the queue.

---

1 How BFS Works:

1. Start from a **source node**.
2. Mark it as **visited**.
3. Push it into a **queue**.
4. While the queue is not empty:
    - Pop the front node.
    - For every **unvisited neighbor** of this node:
        - Mark as visited.
        - Push into the queue.
5. Repeat until the queue is empty.

---

2 Applications of BFS:

- **Shortest Path in Unweighted Graphs**
- **Level Order Traversal of Trees**
- **Cycle Detection in Undirected Graphs**
- **Finding Connected Components**
- **Web Crawlers**
- **Social Network Analysis**

3️⃣ BFS Implementation in Java (using Adjacency List)

Example:

- Graph Structure:
- 0 -- 1 -- 3
- | |
- 2 -- 4
- **Adjacency List:**
- 0 -> [1, 2]
- 1 -> [0, 3, 4]
- 2 -> [0, 4]
- 3 -> [1]
- 4 -> [1, 2]

Java Code:

```
import java.util.*;

public class GraphBFS {

    // BFS function to traverse the graph
    public static void bfs(int start, List<List<Integer>> graph, boolean[] visited) {
        Queue<Integer> queue = new LinkedList<>();  // Queue for BFS
        visited[start] = true;                      // Mark the start node as visited
        queue.add(start);                           // Enqueue the start node

        System.out.println("BFS Traversal starting from node " + start + ":");

        while (!queue.isEmpty()) {
            int currentNode = queue.poll();         // Dequeue a node
            System.out.print(currentNode + " ");    // Process the current node

            // Visit all adjacent nodes
            for (int neighbor : graph.get(currentNode)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;       // Mark as visited
                    queue.add(neighbor);            // Enqueue the unvisited neighbor
                }
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int vertices = 5;  // Number of vertices

        // Create an adjacency list
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < vertices; i++) {
            graph.add(new ArrayList<>());  // Initialize each vertex's adjacency list
        }
```

```
        // Adding edges to the undirected graph
        graph.get(0).add(1);
        graph.get(0).add(2);
        graph.get(1).add(0);
        graph.get(1).add(3);
        graph.get(1).add(4);
        graph.get(2).add(0);
        graph.get(2).add(4);
        graph.get(3).add(1);
        graph.get(4).add(1);
        graph.get(4).add(2);

        // Visited array to keep track of visited nodes
        boolean[] visited = new boolean[vertices];

        // Start BFS from vertex 0
        bfs(0, graph, visited);
    }
}
```

Output:

```
BFS Traversal starting from node 0:
0 1 2 3 4
```

Explanation:

1. **Initialization:**
   - Creates an adjacency list for the graph.
   - Marks all nodes as **unvisited**.
2. **Breadth-First Search:**
   - Starts from a **given node**.
   - Uses a **queue** to maintain the order of exploration.
   - Marks nodes as **visited** as they are enqueued.
3. **Traversal:**
   - Dequeues a node and processes it.
   - Enqueues all **unvisited neighbors**.
   - Stops when the **queue becomes empty**.

---

4 BFS with Adjacency Matrix (Alternative Representation)

If the graph is represented as a **2D matrix**, the logic remains the same, but accessing neighbors changes.
Replace the adjacency list with:

```
if (graph[currentNode][neighbor] == 1 && !visited[neighbor]) {
    visited[neighbor] = true;
    queue.add(neighbor);
}
```

5️⃣ BFS Variations:

1. **Multi-Source BFS:**
   - Start BFS from **multiple sources** simultaneously.
   - Useful for problems like **finding the shortest distance from multiple points**.
2. **Level Order Traversal (Binary Tree):**
   - Uses BFS to traverse the tree level by level.
3. **Cycle Detection:**
   - Use BFS to detect cycles in **undirected graphs** by checking visited neighbors.
4. **Shortest Path (Unweighted Graph):**
   - Uses BFS to find the **minimum distance** between nodes.
5. **Bipartite Graph Check:**
   - Color the graph using BFS to see if it is **2-colorable**.

6️⃣ Time and Space Complexity:

Time Complexity:

- O(V + E) for adjacency list, where:
  - V = number of vertices.
  - E = number of edges.

Space Complexity:

- O(V) for storing the **visited array** and **queue**.
- O(V + E) for the **adjacency list** representation.

7️⃣ Real-World Applications:

- **Social Networking:** Finding **shortest connection paths** between users.
- **Web Crawling:** Exploring all links on a webpage level by level.
- **GPS Navigation:** Finding **shortest routes** in road networks.
- **Network Broadcasting:** Sending messages to **all connected nodes**.
- **Finding Connected Components:** In a graph with multiple disjoint subgraphs.

Summary:

- BFS is an essential graph traversal technique that processes nodes level by level.
- It uses a **queue** to maintain the order of exploration.
- Efficient for finding **shortest paths** in unweighted graphs.
- Works well with **both adjacency list and matrix** representations.

Would you like to see BFS variations or specific problem-solving examples?

Trie (Prefix Tree) in DSA

A **Trie** (pronounced "try") is a special tree-like data structure used to store a dynamic set of strings, where the keys are usually **strings**. Tries are efficient for **searching, inserting, and deleting strings**, especially in applications related to dictionaries and word searches.

---

1 Key Characteristics of Trie:

1. **Tree Structure:** Each node represents a **character** of the string.
2. **Efficient Prefix Search:** Enables fast prefix-based searches.
3. **Space Complexity:** Uses more space compared to hash tables, especially when storing many short strings.
4. **Common Operations:**
    - **Insertion:** Add a word to the trie.
    - **Search:** Check if a word exists.
    - **Prefix Search:** Check if a prefix exists.
    - **Deletion:** Remove a word from the trie.

---

2 Applications of Tries:

- **Autocomplete Systems:** Suggest words based on prefixes.
- **Spell Checkers:** Validate words.
- **IP Routing:** Store routing tables.
- **DNA Sequence Matching:** Efficient string matching.
- **Word Games:** Quickly find words with given prefixes.

---

3 Trie Structure:

- **Node Structure:**
  - An array or map of children (one for each character).
  - A **boolean flag** indicating the end of a word.

Example: Storing Words: ["cat", "cap", "bat", "ball"]

- **Words Stored:** cat, cap, bat, ball

---

4 Trie Implementation in Java

Node Structure:

```
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    // Constructor
    public TrieNode() {
        children = new TrieNode[26]; // Assuming only lowercase a-z
        isEndOfWord = false;
    }
}
```

Trie Class:

```
class Trie {
    private TrieNode root;

    // Constructor to initialize the root
    public Trie() {
        root = new TrieNode();
    }

    // Inserting a word into the trie
    public void insert(String word) {
        TrieNode current = root;
        for (char ch : word.toCharArray()) {
```

```
            int index = ch - 'a'; // Map 'a' to 0, 'b' to 1, ...
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    // Searching for a word in the trie
    public boolean search(String word) {
        TrieNode current = root;
        for (char ch : word.toCharArray()) {
            int index = ch - 'a';
            if (current.children[index] == null) {
                return false; // Not found
            }
            current = current.children[index];
        }
        return current.isEndOfWord;
    }

    // Checking if a prefix exists in the trie
    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char ch : prefix.toCharArray()) {
            int index = ch - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return true;
    }

    // Deleting a word from the trie
    public boolean delete(String word) {
        return delete(root, word, 0);
    }

    private boolean delete(TrieNode current, String word, int depth) {
        if (current == null) {
            return false;
        }

        // Base case: end of the word
        if (depth == word.length()) {
            if (!current.isEndOfWord) {
                return false;
            }
            current.isEndOfWord = false;
            return current.children.length == 0;
        }

        int index = word.charAt(depth) - 'a';
        if (delete(current.children[index], word, depth + 1)) {
```

```
            current.children[index] = null;
            return !current.isEndOfWord && isEmpty(current);
        }

        return false;
    }

    // Helper function to check if a node has no children
    private boolean isEmpty(TrieNode node) {
        for (TrieNode child : node.children) {
            if (child != null) {
                return false;
            }
        }
        return true;
    }
}
```

5️⃣ Usage Example:

```
public class Main {
    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("cat");
        trie.insert("cap");
        trie.insert("bat");
        trie.insert("ball");

        System.out.println("Search 'cat': " + trie.search("cat")); // true
        System.out.println("Search 'cap': " + trie.search("cap")); // true
        System.out.println("Search 'bat': " + trie.search("bat")); // true
        System.out.println("Search 'batman': " + trie.search("batman")); // false

        System.out.println("Prefix 'ca': " + trie.startsWith("ca")); // true
        System.out.println("Prefix 'ba': " + trie.startsWith("ba")); // true
        System.out.println("Prefix 'bat': " + trie.startsWith("bat")); // true
        System.out.println("Prefix 'bal': " + trie.startsWith("bal")); // true

        trie.delete("bat");
        System.out.println("Search 'bat' after delete: " + trie.search("bat")); // false
    }
}
```

6️⃣ Output:

```
Search 'cat': true
Search 'cap': true
Search 'bat': true
Search 'batman': false
Prefix 'ca': true
Prefix 'ba': true
Prefix 'bat': true
Prefix 'bal': true
```

Search 'bat' after delete: false

7️⃣ Complexity Analysis:

Time Complexity:

- **Insertion/Search:** O(L) per word, where **L** is the length of the word.
- **Prefix Search:** O(L) for checking if a prefix exists.
- **Deletion:** O(L) for removing a word.
- Efficient compared to hash maps for prefix-based operations.

Space Complexity:

- O(26 * L * N) for storing N words of length L, assuming lowercase English letters.
- Can be optimized using **hash maps** instead of fixed-size arrays.

8️⃣ Advantages of Tries:

- Efficient for **prefix searching** and **autocomplete**.
- Supports operations in **O(L)** time.
- Preserves the **lexicographical order** of words.

Disadvantages:

- **High Space Complexity:** Uses more memory compared to hash maps, especially with sparse nodes.
- **Complex Implementation:** More challenging to implement and manage compared to hash tables.

9️⃣ Real-World Applications:

1. **Search Engines:** Implementing **autocomplete** and **spell checking**.
2. **IP Routing:** Matching IP addresses efficiently.
3. **Word Games:** Validating or suggesting words.
4. **Genome Sequencing:** Storing and searching DNA sequences.
5. **Data Compression:** Implementing **Huffman coding**.

Summary:

A **Trie** is a highly efficient tree structure for managing dynamic sets of strings, allowing fast **insert**, **search**, **prefix matching**, and **deletion**. It is particularly useful when working with **dictionaries**, **autocomplete systems**, and **longest prefix matching**.

Would you like examples of more advanced Trie operations or applications?

Backtracking in DSA

Backtracking is a problem-solving technique used in algorithms where you build a solution incrementally, **one step at a time**, and remove those solutions that fail to satisfy the constraints. It is widely used for problems involving **combinatorial optimization** and **decision-making**.

---

1 Key Concepts of Backtracking:

1. **Incremental Solution Building:** Start with an empty solution and add components step by step.
2. **Feasibility Check:** After adding a component, check if the current solution is still valid.
3. **Backtrack:** If the current solution is invalid, **remove the last component** and try a new one.
4. **Recursion:** Use recursive function calls to explore all possibilities.

---

2 When to Use Backtracking:

- **Combinatorial Problems:** Finding all permutations, combinations, and subsets.
- **Constraint Satisfaction Problems:** Solving Sudoku, N-Queens problem.
- **Pathfinding Problems:** Finding paths in a maze.
- **Optimization Problems:** Finding the optimal way to arrange or select items.

---

3 Template for Backtracking:

```
void backtrack(PartialSolution solution) {
    if (isSolution(solution)) {
        processSolution(solution); // Valid solution found
        return;
```

```
    }
    for (Choice c : getChoices(solution)) {
        makeChoice(c);          // Add choice to the solution
        backtrack(solution);    // Recur with the new choice
        undoChoice(c);          // Backtrack (remove choice)
    }
}
```

---

4 Example Problems with Backtracking:

1. N-Queens Problem:

Place N queens on an N×N chessboard such that no two queens attack each other.

2. Subset Sum Problem:

Find subsets that sum up to a given number.

3. Permutations of a String:

Generate all permutations of a given string.

Example: N-Queens Problem (Java Implementation)

Problem Statement:

Place N queens on an $N \times N$ chessboard such that no two queens threaten each other.

Java Code:

```
import java.util.*;

public class NQueens {

    // Function to print the chessboard
    private static void printBoard(char[][] board) {
        for (char[] row : board) {
            for (char ch : row) {
                System.out.print(ch + " ");
            }
            System.out.println();
        }
        System.out.println();
    }
```

```
    // Function to check if a queen can be placed at board[row][col]
    private static boolean isSafe(char[][] board, int row, int col, int n) {
        // Check the column
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') return false;
        }

        // Check the upper-left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') return false;
        }

        // Check the upper-right diagonal
        for (int i = row, j = col; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q') return false;
        }

        return true;
    }

    // Backtracking function to place queens
    private static void solveNQueens(char[][] board, int row, int n) {
        if (row == n) { // All queens are placed
            printBoard(board);
            return;
        }

        // Try placing the queen in all columns one by one
        for (int col = 0; col < n; col++) {
            if (isSafe(board, row, col, n)) {
                board[row][col] = 'Q'; // Place queen
                solveNQueens(board, row + 1, n); // Recurse to next row
                board[row][col] = '.'; // Backtrack
            }
        }
    }

    public static void main(String[] args) {
        int n = 4; // Size of the chessboard (4x4)
        char[][] board = new char[n][n];

        // Initialize the board with '.'
        for (char[] row : board) {
            Arrays.fill(row, '.');
        }

        // Solve the N-Queens problem
        solveNQueens(board, 0, n);
    }
}
```

Output:

```
. Q . .
. . . Q
```

```
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

Explanation:

1. **Board Initialization:**
   - Initialize an `N x N` chessboard with `.` representing empty cells.
2. **Safety Check (isSafe):**
   - Check if placing a queen at `board[row][col]` will not lead to a clash with other queens.
3. **Backtracking (solveNQueens):**
   - Place a queen and recursively attempt to place the next queen.
   - If placing the next queen fails, **backtrack** by removing the previously placed queen.
4. **Print Solution (printBoard):**
   - Display the board configuration when a valid arrangement is found.

---

5 Complexity Analysis:

Time Complexity: O(N!)

- For each row, you have `N` possibilities, and the problem size decreases by 1 with each recursive call.
- The worst-case scenario is O(N!) for finding all solutions.

Space Complexity: O(N^2)

- Due to storing the board configuration.

---

6 Other Backtracking Problems:

1. **Subset Sum Problem:** Finding subsets whose elements sum to a given value.
2. **String Permutations:** Generating all possible permutations of a given string.
3. **Sudoku Solver:** Filling the board with numbers from 1 to 9 following Sudoku rules.
4. **Hamiltonian Path:** Finding a path through a graph that visits each vertex exactly once.

5. **Knight's Tour Problem:** Finding a path where a knight visits every square exactly once.

7️⃣ Advantages of Backtracking:

- **Versatile:** Solves a wide range of combinatorial problems.
- **Elegant and Simple:** Easy to understand through recursion.
- **Efficient Pruning:** Stops exploring invalid paths early.

Disadvantages:

- **Exponential Time Complexity:** Inefficient for large input sizes.
- **Stack Overflow:** Deep recursion can cause stack overflow.

8️⃣ Optimization Techniques:

- **Memoization:** Store intermediate results to avoid redundant calculations.
- **Branch and Bound:** Cut off paths that are guaranteed to fail.
- **Bit Masking:** Efficiently represent state in problems like subset sum.

Summary:

Backtracking is a powerful technique for solving problems that involve exploring **all possible combinations**. It relies on recursion to build solutions step by step and backtracks when a solution path is invalid. The N-Queens problem is a classic example demonstrating how to use backtracking effectively.

Would you like to see more backtracking examples or solve a specific problem?

📚 Divide and Conquer in DSA

Divide and Conquer is a classic algorithmic paradigm where a problem is divided into **smaller subproblems**, each subproblem is **solved recursively**, and then their results are **combined** to solve the original problem.

🔧 Key Steps:

1. **Divide** – Split the problem into smaller subproblems (usually of the same type).
2. **Conquer** – Solve each subproblem recursively.
3. **Combine** – Merge the solutions of the subproblems to get the final answer.

✅ Common Characteristics:

- Uses **recursion**
- Typically has a **base case** where recursion stops
- Often leads to **O(n log n)** or **logarithmic** time complexity

---

🚀 Real-World Applications:

- **Sorting algorithms**: Merge Sort, Quick Sort
- **Searching algorithms**: Binary Search
- **Matrix multiplication**: Strassen's algorithm
- **Finding closest pair of points** in a 2D plane
- **Computational geometry**
- **Divide & Conquer DP (Advanced)**

💡 Examples with Java Code

1️⃣ Binary Search (O(log n))

Problem: Find the index of a target element in a sorted array.

```
public class BinarySearch {

    public static int binarySearch(int[] arr, int target, int low, int high) {
        if (low > high) return -1;

        int mid = low + (high - low) / 2;

        if (arr[mid] == target) return mid;
        else if (arr[mid] > target) return binarySearch(arr, target, low, mid - 1);
        else return binarySearch(arr, target, mid + 1, high);
    }
```

```
    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 10, 15, 20};
        int target = 10;
        int index = binarySearch(arr, target, 0, arr.length - 1);
        System.out.println("Index of " + target + " is: " + index); // Output: 3
    }
}
```

---

2 Merge Sort (O(n log n))

Problem: Sort an array using divide and conquer.

```
import java.util.Arrays;

public class MergeSort {

    public static void mergeSort(int[] arr, int low, int high) {
        if (low < high) {
            int mid = (low + high) / 2;

            mergeSort(arr, low, mid);     // Sort left half
            mergeSort(arr, mid + 1, high); // Sort right half

            merge(arr, low, mid, high);    // Merge sorted halves
        }
    }

    private static void merge(int[] arr, int low, int mid, int high) {
        int[] temp = new int[high - low + 1];
        int i = low, j = mid + 1, k = 0;

        // Merge both halves into temp[]
        while (i <= mid && j <= high) {
            temp[k++] = (arr[i] <= arr[j]) ? arr[i++] : arr[j++];
        }

        while (i <= mid) temp[k++] = arr[i++];
        while (j <= high) temp[k++] = arr[j++];

        // Copy temp[] to original array
        for (i = low, k = 0; i <= high; i++, k++) {
            arr[i] = temp[k];
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 6, 3, 7};
        mergeSort(arr, 0, arr.length - 1);
        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

---

3 Maximum Subarray (Kadane's + Divide & Conquer)

Problem: Find the subarray with the maximum sum.

```
public class MaxSubarray {

    public static int maxCrossingSum(int[] arr, int l, int m, int h) {
        int sum = 0;
        int leftSum = Integer.MIN_VALUE;
        for (int i = m; i >= l; i--) {
            sum += arr[i];
            if (sum > leftSum) leftSum = sum;
        }

        sum = 0;
        int rightSum = Integer.MIN_VALUE;
        for (int i = m + 1; i <= h; i++) {
            sum += arr[i];
            if (sum > rightSum) rightSum = sum;
        }

        return leftSum + rightSum;
    }

    public static int maxSubArray(int[] arr, int l, int h) {
        if (l == h) return arr[l];

        int m = (l + h) / 2;

        int leftMax = maxSubArray(arr, l, m);
        int rightMax = maxSubArray(arr, m + 1, h);
        int crossMax = maxCrossingSum(arr, l, m, h);

        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }

    public static void main(String[] args) {
        int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        int maxSum = maxSubArray(arr, 0, arr.length - 1);
        System.out.println("Maximum Subarray Sum: " + maxSum); // Output: 6
    }
}
```

Time Complexities Comparison

Algorithm	Time Complexity	Space Complexity
Binary Search	O(log n)	O(1) or O(log n)
Merge Sort	O(n log n)	O(n)

Algorithm	Time Complexity	Space Complexity
Quick Sort	O(n log n) avg	O(log n)
Max Subarray (D&C)	O(n log n)	O(log n)

✅ Advantages

- Breaks complex problems into smaller, manageable parts
- Reduces time complexity (often from $O(n^2)$ to $O(n \log n)$)
- Highly recursive-friendly and easy to reason about

⚠️ Disadvantages

- May use extra space (e.g., merge sort)
- Recursion may lead to stack overflow for very deep calls

🧠 Summary

Step	What it does
Divide	Break problem into smaller parts
Conquer	Solve each part recursively
Combine	Merge the results of subproblems

Divide and Conquer is **fundamental** to DSA and forms the basis for many efficient algorithms.

Would you like help applying this to a specific problem (e.g., closest pair of points, matrix multiplication)?

🔍 Binary Search in DSA

Binary Search is a powerful algorithm used to **efficiently search** for an element in a **sorted array**. Instead of checking every element, it **divides** the search space in half at each step.

✅ Key Characteristics

- **Input must be sorted** (ascending or descending).
- Works in **O(log n)** time.
- Can be applied to arrays, trees, and even used for answer-based problems (e.g., "min/max possible value").

🧠 Intuition

At each step:

1. Compare the `mid` element with the `target`.
2. If equal → return index.
3. If `target < mid` → search left half.
4. If `target > mid` → search right half.

✅ Java Code – Iterative Version

```
public class BinarySearch {

    public static int binarySearch(int[] nums, int target) {
        int low = 0;
        int high = nums.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] == target) {
                return mid; // Found target
            } else if (nums[mid] < target) {
                low = mid + 1; // Search right half
            } else {
                high = mid - 1; // Search left half
            }
        }
```

```
        return -1; // Target not found
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 5, 7, 9, 11, 13};
        int target = 9;
        int result = binarySearch(nums, target);
        System.out.println("Index of " + target + ": " + result);  // Output: 4
    }
}
```

🌀 Recursive Version

```
public static int binarySearchRecursive(int[] nums, int target, int low, int high) {
    if (low > high) return -1;

    int mid = low + (high - low) / 2;

    if (nums[mid] == target) return mid;
    else if (nums[mid] < target)
        return binarySearchRecursive(nums, target, mid + 1, high);
    else
        return binarySearchRecursive(nums, target, low, mid - 1);
}
```

⏱️ Time & Space Complexity

Metric	Value
Time	O(log n)
Space (iter.)	O(1)
Space (recur.)	O(log n)

🔧 Applications of Binary Search

1. **Search in sorted array**
2. **Lower/Upper bound** (first/last occurrence)
3. **Peak Element in Mountain Array**
4. **Search in Rotated Sorted Array**
5. **Binary Search on Answer** – used in optimization problems:
    - Allocate minimum pages
    - Capacity to ship packages
    - Koko Eating Bananas
    - Aggressive cows

□ Binary Search Variations

Variation	Description
Standard Binary Search	Find if element exists
Lower Bound (First ≥ x)	Smallest index where value ≥ target
Upper Bound (First > x)	Smallest index where value > target
Find First/Last Occurrence	Use modified binary search on boundaries
Binary Search on Answer	Search space is not an array but a value range

---

🔄 Lower Bound Example (Java)

```
public static int lowerBound(int[] nums, int target) {
    int low = 0, high = nums.length;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] < target) low = mid + 1;
        else high = mid;
    }
    return low;
}
```

✅ Summary

- Binary Search is **fast and efficient**, ideal for sorted data.
- Time complexity: **O(log n)**
- Use cases go beyond just arrays – think **searching in ranges**, **matrices**, **bit manipulation**, and **optimization problems**.

Would you like to explore **Binary Search on Rotated Arrays** or **Binary Search on Answer** problems next?

⛏️ Heap in DSA (Data Structures & Algorithms)

A **Heap** is a special **binary tree** data structure used for efficiently retrieving the **minimum or maximum element**. It's often used to implement **priority queues**, **scheduling**, and **Top-K problems**.

✅ Types of Heaps

1. **Min Heap**: Root is the **smallest** element.
2. **Max Heap**: Root is the **largest** element.

Properties:

- Complete binary tree (filled from left to right).
- Every parent node obeys the heap condition (min or max).
- Usually implemented using **arrays** for efficiency.

🧠 Heap as Array (1-based index idea):

For node at index `i`:

- Left child: `2*i`
- Right child: `2*i + 1`
- Parent: `i/2`

For 0-based arrays (Java):

- Left child: `2*i + 1`
- Right child: `2*i + 2`
- Parent: `(i - 1) / 2`

🔨 Java Code: Min Heap using PriorityQueue

Java provides a built-in **PriorityQueue** which behaves like a **Min Heap** by default.

```
import java.util.PriorityQueue;

public class MinHeapExample {
    public static void main(String[] args) {
```

```
        // Min Heap (default)
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        minHeap.add(10);
        minHeap.add(5);
        minHeap.add(20);
        minHeap.add(2);

        System.out.println("Min element: " + minHeap.peek()); // 2

        while (!minHeap.isEmpty()) {
            System.out.print(minHeap.poll() + " "); // 2 5 10 20
        }
    }
}
```

🔨 Max Heap in Java

To make a Max Heap using PriorityQueue, use a **comparator**:

```
import java.util.*;

public class MaxHeapExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        maxHeap.add(10);
        maxHeap.add(5);
        maxHeap.add(20);
        maxHeap.add(2);

        System.out.println("Max element: " + maxHeap.peek()); // 20

        while (!maxHeap.isEmpty()) {
            System.out.print(maxHeap.poll() + " "); // 20 10 5 2
        }
    }
}
```

🔧 Common Use Cases

Problem Type	Example
Top-K Elements	Top K frequent elements, scores
Priority Queues	OS job scheduling, event handling
Dijkstra’s Shortest Path	Graphs

Problem Type	Example
Merge K Sorted Lists	Leetcode 23
Kth Largest/Smallest Element	Heap of size K
Median of Data Stream	Two heaps

💡 Kth Largest Element (Min Heap)

```
public static int findKthLargest(int[] nums, int k) {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for (int num : nums) {
        minHeap.add(num);
        if (minHeap.size() > k) {
            minHeap.poll();  // remove smallest
        }
    }
    return minHeap.peek();  // kth largest
}
```

□ Time & Space Complexity

Operation	Time (Array Heap)
Insert	O(log n)
Delete (poll)	O(log n)
Peek (min/max)	O(1)
Heapify	O(n)

🛠□ Build Heap from Array (Heapify)

```
public void heapify(int[] arr) {
    int n = arr.length;
    for (int i = n / 2 - 1; i >= 0; i--) {
        siftDown(arr, i, n);
    }
}

private void siftDown(int[] arr, int i, int n) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
```

```
    if (left < n && arr[left] < arr[smallest]) smallest = left;
    if (right < n && arr[right] < arr[smallest]) smallest = right;

    if (smallest != i) {
        int temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
        siftDown(arr, smallest, n);
    }
}
```

✅ Summary

- **Heap** is a binary tree used for efficiently accessing min/max.
- **Min Heap**: root is smallest; **Max Heap**: root is largest.
- Java's `PriorityQueue` simplifies heap usage.
- Extremely useful for problems involving **order**, **rank**, or **priority**.

Would you like to explore how to **implement a heap from scratch** or apply it to a real problem like **Median of Data Stream**?

Absolutely! Let's visually walk through the `heapify` code and understand **how it transforms an array into a Min-Heap** step by step.

🔍 What Is Heapify?

`heapify()` builds a valid **Min-Heap** from an **unsorted array**.
It works by calling `siftDown()` (also called `heapifyDown` or `bubbleDown`) from the **bottom non-leaf nodes upward**.

🎁 Input Array

Let's say we have:

```
arr = [5, 3, 8, 4, 1, 2]
Indexes:  0  1  2  3  4  5
```

This array represents a **complete binary tree** like this:

```
        5
```

```
     /    \
    3      8
  /   \    /
 4     1  2
```

We want to convert this into a **Min-Heap**, where every parent is ≤ children.

🔄 Step-by-Step Heapify

We loop from the **last parent node** down to the root:

```
for (int i = n/2 - 1; i >= 0; i--) {
    siftDown(arr, i, n);
}
```

In our case, n = 6 ⟶ n/2 - 1 = 2

We will call siftDown() in this order:

- i = 2
- i = 1
- i = 0

🔎 Step 1: siftDown(arr, 2)

Node: arr[2] = 8
Children: left = 5, right = out of bounds
Compare 8 with 2 → Swap!

```
Before: [5, 3, 8, 4, 1, 2]
            ^
After:  [5, 3, 2, 4, 1, 8]
                  ^
```

🔎 Step 2: siftDown(arr, 1)

Node: arr[1] = 3
Children: arr[3] = 4, arr[4] = 1
Smallest child = 1, so swap 3 and 1

```
Before: [5, 3, 2, 4, 1, 8]
               ^
After:  [5, 1, 2, 4, 3, 8]
```

^

Now arr[4] is a leaf, done.

🔎 Step 3: siftDown(arr, 0)

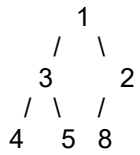
Node: arr[0] = 5
Children: arr[1] = 1, arr[2] = 2
Smallest child = 1 → Swap 5 and 1

Before: [5, 1, 2, 4, 3, 8]
After: [1, 5, 2, 4, 3, 8]

Now check arr[1] = 5
Children: arr[3] = 4, arr[4] = 3
Smallest = 3 → Swap 5 and 3

Final: [1, 3, 2, 4, 5, 8]

✅ Final Heapified Array (Min-Heap)



Each parent is ≤ its children. ✓

🔄 Visualization Summary

Each siftDown(i) step compares a node to its children:

- If the node is **larger**, it **swaps** with the **smallest child**.
- This ensures the heap property percolates **down the tree**.

The loop runs from bottom-up: n/2 - 1 to 0, fixing one subtree at a time.

Would you like a visual **animation-style drawing** of the process, or help implementing a **Max-Heap version**?

Great follow-up! Finding the **Kth largest or smallest element** is a classic use case of **Heaps**. Let me explain both scenarios clearly using **Min Heap** and **Max Heap**, along with **code and logic**.

🔢 1. Kth Largest Element Using Min Heap

🧠 Idea:

- Maintain a **Min Heap** of the **K largest elements** seen so far.
- If the heap size exceeds K, remove the **smallest** element.
- At the end, the root of the Min Heap is the **Kth largest**.

✅ Java Code:

```
import java.util.PriorityQueue;

public class KthLargest {
    public static int findKthLargest(int[] nums, int k) {
        // Min Heap to store k largest elements
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.add(num);  // Add current number
            if (minHeap.size() > k) {
                minHeap.poll(); // Remove smallest of the k+1 numbers
            }
        }

        return minHeap.peek(); // Kth largest element
    }

    public static void main(String[] args) {
        int[] nums = {3, 2, 1, 5, 6, 4};
        int k = 2;
        System.out.println(findKthLargest(nums, k)); // Output: 5
    }
}
```

🔢 2. Kth Smallest Element Using Max Heap

🧠 Idea:

- Maintain a **Max Heap** of the **K smallest elements**.
- If the heap size exceeds K, remove the **largest** element.

- The root of the Max Heap is the **Kth smallest**.

✅ Java Code:

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

public class KthSmallest {
    public static int findKthSmallest(int[] nums, int k) {
        // Max Heap to store k smallest elements
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        for (int num : nums) {
            maxHeap.add(num);
            if (maxHeap.size() > k) {
                maxHeap.poll(); // Remove the largest
            }
        }

        return maxHeap.peek(); // Kth smallest element
    }

    public static void main(String[] args) {
        int[] nums = {7, 10, 4, 3, 20, 15};
        int k = 3;
        System.out.println(findKthSmallest(nums, k)); // Output: 7
    }
}
```

□ Time & Space Complexity

Operation	Complexity
Insertion in heap	O(log k)
Total for n elements	O(n log k)
Space (heap size)	O(k)

🔁 Alternate: Sorting (Less Efficient)

You could sort the array and pick the element:

```
Arrays.sort(nums);
return nums[nums.length - k]; // kth largest
```

But this is **O(n log n)** time and not efficient for large input.

□ When to Use Heap

- When you only need the **top K smallest/largest**.
- Much faster than full sorting for partial results.

Would you like to see this applied to **streams (online data)** or how this works for **2D arrays or** matrice.

Bit Manipulation

Bit Manipulation is a powerful technique in DSA that involves working directly with the binary representation of numbers using **bitwise operators**.

🔧 Why Use Bit Manipulation?

- Super fast operations: O(1) time
- Efficient memory use
- Useful in low-level optimizations, toggling flags, solving problems like:
  - Finding the **only non-repeating number**
  - Checking **even/odd**
  - Counting set bits
  - Working with **subsets**, **masks**, **power-of-two checks**, etc.

🔢 Common Bitwise Operators

Operator	Symbol	Description	Example (a = 5, b = 3)
AND	&	1 only if both bits are 1	5 & 3 = 1 (101 & 011)
OR	`	`	1 if any bit is 1
XOR	^	1 if bits are different	5 ^ 3 = 6

Operator	Symbol	Description	Example (a = 5, b = 3)
NOT	~	Inverts bits	~5 = -6
Left Shift	<<	Multiplies by 2^k	5 << 1 = 10
Right Shift	>>	Divides by 2^k	5 >> 1 = 2

□ Key Tricks and Use Cases

1. Check if Number is Even or Odd

```
if ((n & 1) == 0) System.out.println("Even");
else System.out.println("Odd");
```

2. Count Set Bits (1s)

```
int count = 0;
while (n > 0) {
    count += (n & 1);  // add 1 if last bit is 1
    n >>= 1;
}
```

Or use **Brian Kernighan's Algorithm**:

```
while (n > 0) {
    n = n & (n - 1);
    count++;
}
```

3. Check if Number is Power of 2

```
if (n > 0 && (n & (n - 1)) == 0)
    System.out.println("Power of 2");
```

4. Swap Two Numbers Without Temp

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

5. Find Unique Number (All Others Appear Twice)

```
int res = 0;
for (int num : arr) {
    res ^= num;
}
System.out.println("Unique number: " + res);
```

6. Set / Clear / Toggle ith Bit

```
// Set bit
n = n | (1 << i);
```

```
// Clear bit
n = n & ~(1 << i);

// Toggle bit
n = n ^ (1 << i);
```

□ Real Problem Example

Problem: All numbers appear twice except one

```
int[] arr = {2, 3, 5, 4, 5, 3, 4};
int result = 0;

for (int num : arr)
    result ^= num;

System.out.println(result);  // Output: 2
```

Why this works?
Because `x ^ x = 0` and `x ^ 0 = x`.

✓ When to Use Bit Manipulation

- Counting problems with powers of 2
- Subsets / combinations (with bitmasking)
- Fast toggling / setting of flags
- Game theory problems (e.g., Grundy numbers)

Would you like to dive into **bitmasking + DP**, **subset generation**, or **specific interview problems** next?

1D DP

Let’s dive into 1D Dynamic Programming (DP) — a fundamental concept in DSA used to optimize recursive problems by storing intermediate results in a 1D array.

✅ What is 1D DP?

1D DP uses a 1-dimensional array to store results of subproblems and build the final solution bottom-up.

- It's typically used when the problem depends only on a fixed number of previous states — usually just one dimension (like index i).

 General Approach

1. Define the state:

Dp[i] = solution to subproblem ending at index i.

2. Initialize base cases:

Define dp[0], dp[1], etc., depending on the problem.

3. Write the recurrence relation:

How dp[i] depends on dp[i-1], dp[i-2], etc.

4. Iterate and fill the array.

📌 Examples

---

 1. Fibonacci Numbers

```
Int fib(int n) {
    If (n <= 1) return n;
    Int[] dp = new int[n + 1];
    Dp[0] = 0;
    Dp[1] = 1;

    For (int I = 2; I <= n; i++) {
        Dp[i] = dp[I – 1] + dp[I – 2];
    }

    Return dp[n];
}
```

Dp[i] = dp[I – 1] + dp[I – 2]

Can be optimized to O(1) space using 2 variables.

---

 2. Climbing Stairs

- You can climb 1 or 2 steps. How many distinct ways to climb to the top (n steps)?

```
Int climbStairs(int n) {
    If (n <= 1) return 1;
    Int[] dp = new int[n + 1];
    Dp[0] = 1;  // 1 way to stay at ground
    Dp[1] = 1;

    For (int I = 2; I <= n; i++) {
        Dp[i] = dp[I – 1] + dp[I – 2];
    }

    Return dp[n];
}
```

Same recurrence as Fibonacci.

---

 3. House Robber

- Rob non-adjacent houses to maximize the amount.

```
Int rob(int[] nums) {
    Int n = nums.length;
    If (n == 0) return 0;
    If (n == 1) return nums[0];

    Int[] dp = new int[n];
    Dp[0] = nums[0];
    Dp[1] = Math.max(nums[0], nums[1]);

    For (int I = 2; I < n; i++) {
        Dp[i] = Math.max(dp[I – 1], nums[i] + dp[I – 2]);
    }

    Return dp[n – 1];
}
```

Choose between robbing current + i-2, or skipping to i-1.

---

 4. Maximum Subarray (Kadane’s Algorithm)

```
Int maxSubArray(int[] nums) {
    Int n = nums.length;
    Int[] dp = new int[n];
    Dp[0] = nums[0];
    Int max = dp[0];

    For (int I = 1; I < n; i++) {
        Dp[i] = Math.max(nums[i], dp[I – 1] + nums[i]);
        Max = Math.max(max, dp[i]);
    }

    Return max;
}
```

You can reduce space to a single variable.

🛠 When to Use 1D DP?

Subproblems depend only on 1 previous dimension (like I – 1, I – 2)

Problems with “maximum”, “minimum”, “count ways”, etc.

Efficient for problems involving linear sequences.

 Practice Problems

1. Leetcode 70: Climbing Stairs

2. Leetcode 198: House Robber

3. Leetcode 53: Maximum Subarray

4. Leetcode 746: Min Cost Climbing Stairs

5. Leetcode 121: Best Time to Buy and Sell Stock

Would you like to explore 2D DP, space optimizations, or real interview variations next?

MultiDimensional DP

Great! Let's break down **multidimensional (2D/3D/etc.) Dynamic Programming** in DSA — this is crucial for problems with multiple changing parameters, like grids, substrings, or knapsacks.

✅ What is Multidimensional DP?

Multidimensional DP uses a **2D or higher array (`dp[][]`, `dp[][][]`, etc.)** to store subproblem results when **more than one variable/state** changes.

It's used when a problem can't be solved by a single state like `dp[i]` — you need combinations like `dp[i][j]`, `dp[i][j][k]`, etc.

□ General Template for 2D DP

```
// 2D DP array
int[][] dp = new int[n + 1][m + 1];

// Base cases (often row 0 or col 0)
dp[0][0] = some_base_case;

// Recurrence
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        dp[i][j] = compute_from(dp[i-1][j], dp[i][j-1], ...);
    }
}
```

📌 Common Examples

🥢 1. Longest Common Subsequence (LCS)

Find the length of the longest subsequence common to two strings.

```
int longestCommonSubsequence(String text1, String text2) {
    int m = text1.length(), n = text2.length();
```

```
    int[][] dp = new int[m + 1][n + 1];

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1.charAt(i - 1) == text2.charAt(j - 1))
                dp[i][j] = 1 + dp[i - 1][j - 1];
            else
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[m][n];
}
```

- State: `dp[i][j]` = LCS length of `text1[0..i-1]` and `text2[0..j-1]`

---

2. 0/1 Knapsack

Given weights and values, maximize value within a weight limit.

```
int knapsack(int[] wt, int[] val, int W) {
    int n = wt.length;
    int[][] dp = new int[n + 1][W + 1];

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (wt[i - 1] <= w)
                dp[i][w] = Math.max(dp[i - 1][w], val[i - 1] + dp[i - 1][w - wt[i - 1]]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }
    return dp[n][W];
}
```

- State: `dp[i][w]` = max value with first `i` items and weight limit `w`

---

3. Grid Unique Paths

Count number of paths from top-left to bottom-right in `m x n` grid.

```
int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];

    // First row and col = 1 way
    for (int i = 0; i < m; i++) dp[i][0] = 1;
    for (int j = 0; j < n; j++) dp[0][j] = 1;
```

```
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }

    return dp[m - 1][n - 1];
}
```

- `dp[i][j]` = number of ways to reach cell `(i,j)`

---

4. Edit Distance

> Minimum operations to convert one string to another.

```
int minDistance(String word1, String word2) {
    int m = word1.length(), n = word2.length();
    int[][] dp = new int[m + 1][n + 1];

    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1.charAt(i - 1) == word2.charAt(j - 1))
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = 1 + Math.min(dp[i - 1][j - 1],
                               Math.min(dp[i - 1][j], dp[i][j - 1]));
        }
    }
    return dp[m][n];
}
```

When to Use 2D or Higher DP?

Use when:

- You have **two inputs changing independently** (like two strings, or item + weight)
- You need to track states with multiple variables
- Grids or matrices are involved

□ Popular Leetcode Problems to Practice

1. 1143. Longest Common Subsequence
2. 416. Partition Equal Subset Sum
3. 322. Coin Change
4. 72. Edit Distance
5. 62. Unique Paths
6. 518. Coin Change 2

Would you like a **visual diagram** or explanation of how any of these DP tables evolve step-by-step?