GALGOTIAS UNIVERSITY

SCHOOL OF COMPUTING SCIENCE & ENGINEERING

# Advanced Algorithmic Problem Solving

**MTE Assignment**

**NAME OF STUDENT : Sandeep Kumar**

**SEMESTER : 6TH**

**ROLL NO. : 22131011230**

**SECTION : 8**

SUBMITTED TO
Mr.Aditya Trivedi

1. Explain the concept of a prefix sum array and its applications.

Ans-Prefix Sum Array: Concept and Applications

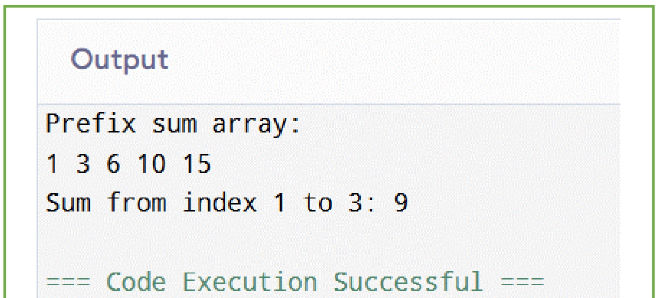
Concept

A prefix sum array (also called a cumulative sum array or scan) is a data structure that allows efficient range sum queries on an array. It's constructed by taking the running total of elements from the start of the array up to each index. Given an input array arr of size n, the prefix sum array prefix is constructed as:

- prefix[0] = arr[0]
- prefix[1] = arr[0] + arr[1]
- prefix[2] = arr[0] + arr[1] + arr[2]
- prefix[i] = prefix[i-1] + arr[i]

Code

```
public class PrefixSum {
    public static int[] createPrefixSum(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 rangeSum(int[] prefix, int l, int r) {
        if (l == 0) return prefix[r];
        return prefix[r] - prefix[l-1];
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int[] prefix = createPrefixSum(arr);

        System.out.println("Prefix sum array:");
        for (int num : prefix) {
            System.out.print(num + " ");
        }
                System.out.println("\nSum from index 1 to 3: " + rangeSum(prefix,
1, 3));
    }
}
```



Output

```
Prefix sum array:
1 3 6 10 15
Sum from index 1 to 3: 9

=== Code Execution Successful ===
```

2. Write a program to find the sum of elements in a given range [L, R] using a prefix sum array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

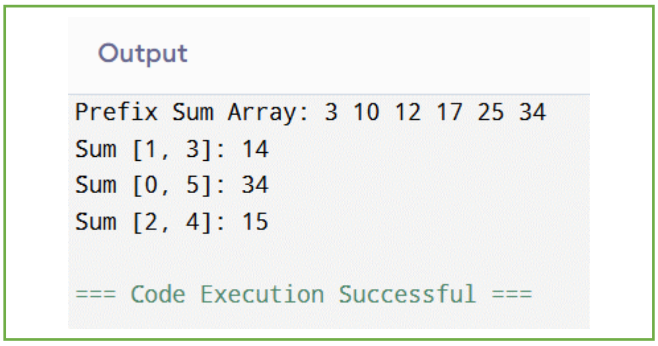
**Ans -** Sum of Elements in Range [L, R] using Prefix Sum Array

Algorithm

1. **Prefix Sum Construction**:
   - Create an array `prefix` of same size as input array
   - Set `prefix[0] = arr[0]`
   - For each subsequent index `i`, compute `prefix[i] = prefix[i-1] + arr[i]`
2. **Range Sum Query**:
   - If `L == 0`, return `prefix[R]`
   - Otherwise, return `prefix[R] - prefix[L-1]`

Code

```
public class RangeSumWithPrefix {

    // Build prefix sum array
    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;
    }

    // Query sum in range [L, R]
    public static int rangeSum(int[] prefix, int L, int R) {
        if (L < 0 || R >= prefix.length || L > R) {
            throw new IllegalArgumentException("Invalid range");
        }
        return (L == 0) ? prefix[R] : prefix[R] - prefix[L-1];
    }

    public static void main(String[] args) {
        int[] arr = {3, 7, 2, 5, 8, 9};
        int[] prefix = buildPrefixSum(arr);
        // Print prefix array
        System.out.print("Prefix Sum Array: ");
        for (int num : prefix) {
            System.out.print(num + " ");
        }
        System.out.println();

        // Example queries
        System.out.println("Sum [1, 3]: " + rangeSum(prefix, 1, 3)); // 7+2+5 = 14
        System.out.println("Sum [0, 5]: " + rangeSum(prefix, 0, 5)); // All
elements = 34
        System.out.println("Sum [2, 4]: " + rangeSum(prefix, 2, 4)); // 2+5+8 = 15
    }
}
```


Output

```
Prefix Sum Array: 3 10 12 17 25 34
Sum [1, 3]: 14
Sum [0, 5]: 34
Sum [2, 4]: 15

=== Code Execution Successful ===
```

3. Solve the problem of finding the equilibrium index in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - Finding Equilibrium Index in an Array

Problem Definition

An equilibrium index in an array is an index where the sum of elements before it is equal to the sum of elements after it. Formally, index i is an equilibrium index if: sum(arr[0..i-1]) = =sum(arr[i+1..n-1])

Algorithm Calculate Total Sum: Compute the total sum of all elements in the array

Initialize Left Sum: Start with left sum = 0

Iterate Through Array:

For each index i, calculate right sum as total sum - left sum - arr[i]

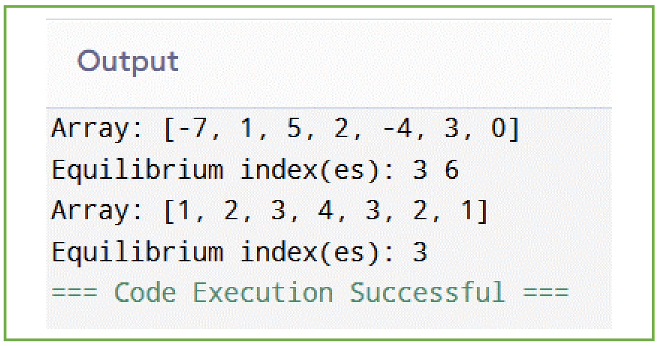
Compare left sum and right sum

If equal, i is an equilibrium index

Add current element to left sum before moving to next index

Code

```
public class EquilibriumIndex {
        public static void findEquilibriumIndices(int[] arr) {
        int totalSum = 0;
        int leftSum = 0;
                // Calculate total sum of array
        for (int num : arr) {
            totalSum += num;
        }

        System.out.print("Equilibrium index(es): ");

        for (int i = 0; i < arr.length; i++) {
            // Right sum = total sum - left sum - current element
            int rightSum = totalSum - leftSum - arr[i];
            if (leftSum == rightSum) {
                System.out.print(i + " ");
            }
                        // Add current element to left sum for next iteration
            leftSum += arr[i];
        }
    }
        public static void main(String[] args) {
        int[] arr1 = {-7, 1, 5, 2, -4, 3, 0};
        System.out.println("Array: " + java.util.Arrays.toString(arr1));
        findEquilibriumIndices(arr1);

        System.out.println();

        int[] arr2 = {1, 2, 3, 4, 3, 2, 1};
        System.out.println("Array: " + java.util.Arrays.toString(arr2));
        findEquilibriumIndices(arr2);
    }
}
```


Output

```
Array: [-7, 1, 5, 2, -4, 3, 0]
Equilibrium index(es): 3 6
Array: [1, 2, 3, 4, 3, 2, 1]
Equilibrium index(es): 3
=== Code Execution Successful ===
```

4. Check if an array can be split into two parts such that the sum of the prefix equals the sum of the suffix. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans** - Check if Array Can Be Split into Equal Prefix and Suffix Sums

Problem Definition

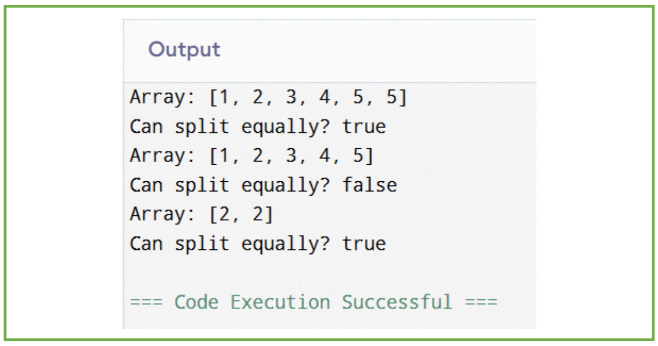
We need to determine if there exists a split point in an array where the sum of elements before the split (prefix) equals the sum of elements after the split (suffix).

Algorithm

1. **Calculate Total Sum**: Compute the total sum of all elements in the array
2. **Initialize Left Sum**: Start with left sum = 0
3. **Iterate Through Array**:
   - For each element, add it to the left sum
   - Check if left sum equals (total sum - left sum)
   - If equal at any point, return true
4. **Final Check**: If no split found, return false

Code

```java
public class PrefixSuffixSplit {


    public static boolean canSplitEqually(int[] arr) {
        int totalSum = 0;


        // Calculate total sum
        for (int num : arr) {
            totalSum += num;
        }


        int leftSum = 0;
        for (int num : arr) {
            leftSum += num;
            // Check if left sum equals right sum (totalSum - leftSum)
            if (leftSum == totalSum - leftSum) {
                return true;
            }
        }

        return false;
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5, 5}; // Can be split after 4 (1+2+3+4=10,
5+5=10)
        System.out.println("Array: " + java.util.Arrays.toString(arr1));
        System.out.println("Can split equally? " + canSplitEqually(arr1));

        int[] arr2 = {1, 2, 3, 4, 5}; // Cannot be split equally
        System.out.println("Array: " + java.util.Arrays.toString(arr2));
        System.out.println("Can split equally? " + canSplitEqually(arr2));

        int[] arr3 = {2, 2}; // Can be split after first element
        System.out.println("Array: " + java.util.Arrays.toString(arr3));
        System.out.println("Can split equally? " + canSplitEqually(arr3));
    }
}
```


Output

```
Array: [1, 2, 3, 4, 5, 5]
Can split equally? true
Array: [1, 2, 3, 4, 5]
Can split equally? false
Array: [2, 2]
Can split equally? true

=== Code Execution Successful ===
```

5. Find the maximum sum of any subarray of size K in a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans -** Maximum Sum of Subarray of Size K (Sliding Window Technique)

Problem Definition.

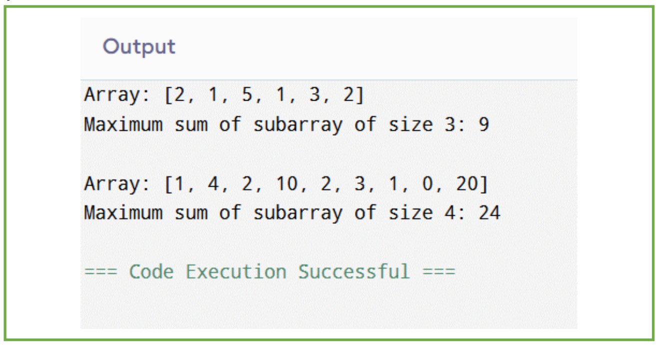
Given an array of integers and a number K, find the maximum sum of any contiguous subarray of size K.

Algorithm (Sliding Window Approach)

1. **Calculate Initial Window Sum**:
   - Compute sum of first K elements (window [0..K-1])
2. **Initialize Maximum Sum**:
   - Set max_sum = current window sum
3. **Slide Window Through Array**:
   - For each subsequent position i (from K to n-1):
     - Subtract the element leaving the window (arr[i-K])
     - Add the new element entering the window (arr[i])
     - Update max_sum if current window sum is greater
4. **Return Result**:
   - Return the maximum sum found

Code

```
public class MaxSubarraySum {

        public static int maxSumSubarray(int[] arr, int K) {

        if (K > arr.length) {

            throw new IllegalArgumentException("K cannot be larger than array
length");

        }

        // Calculate sum of first window
        int windowSum = 0;

        for (int i = 0; i < K; i++) {

            windowSum += arr[i];

        }

                int maxSum = windowSum;

                // Slide window through array

        for (int i = K; i < arr.length; i++) {

            windowSum = windowSum - arr[i - K] + arr[i]; // Slide window

            maxSum = Math.max(maxSum, windowSum);

        }

                return maxSum;

    }

        public static void main(String[] args) {

        int[] arr = {2, 1, 5, 1, 3, 2};

        int K = 3;

        System.out.println("Array: " + java.util.Arrays.toString(arr));

        System.out.println("Maximum sum of subarray of size " + K + ": " +
maxSumSubarray(arr, K));

                int[] arr2 = {1, 4, 2, 10, 2, 3, 1, 0, 20};

        K = 4;
        System.out.println("\nArray: " + java.util.Arrays.toString(arr2));

        System.out.println("Maximum sum of subarray of size " + K + ": " +
maxSumSubarray(arr2, K));

    }

}
```



Output

```
Array: [2, 1, 5, 1, 3, 2]
Maximum sum of subarray of size 3: 9

Array: [1, 4, 2, 10, 2, 3, 1, 0, 20]
Maximum sum of subarray of size 4: 24

=== Code Execution Successful ===
```

6. Find the length of the longest substring without repeating characters. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

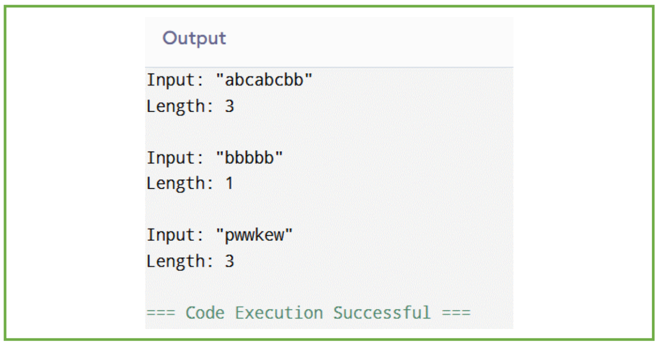
**Ans -** Longest Substring Without Repeating Characters
Problem Definition

Given a string, find the length of the longest substring without repeating characters.
Algorithm (Sliding Window with Hash Map)

1. **Initialize Variables**:
   - Use a hash map to store characters and their indices
   - Maintain two pointers (left and right) to represent the current window
   - Track maximum length found
2. **Expand Window**:
   - Move right pointer forward
   - If character exists in map and its index ≥ left pointer:
     - Move left pointer to index after the duplicate character
   - Update character's latest index in map
   - Update maximum length if current window is larger
3. **Return Result**:
   - Return the maximum length found

Code

```
import java.util.HashMap;


public class LongestUniqueSubstring {


    public static int lengthOfLongestSubstring(String s) {

        HashMap<Character, Integer> charIndexMap = new HashMap<>();

        int maxLength = 0;

        int left = 0;


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

            char currentChar = s.charAt(right);
            // If character exists in map and is within current window
            if (charIndexMap.containsKey(currentChar) &&
charIndexMap.get(currentChar) >= left) {
                left = charIndexMap.get(currentChar) + 1; // Move left past
duplicate
            }

            charIndexMap.put(currentChar, right); // Update character's latest
index
            maxLength = Math.max(maxLength, right - left + 1); // Update max
length
        }
                return maxLength;
    }
        public static void main(String[] args) {
        String input1 = "abcabcbb";
        System.out.println("Input: \"" + input1 + "\"");
        System.out.println("Length: " + lengthOfLongestSubstring(input1));

        String input2 = "bbbbb";
        System.out.println("\nInput: \"" + input2 + "\"");
        System.out.println("Length: " + lengthOfLongestSubstring(input2));

        String input3 = "pwwkew";
        System.out.println("\nInput: \"" + input3 + "\"");
        System.out.println("Length: " + lengthOfLongestSubstring(input3));
    }
}
```



Output

```
Input: "abcabcbb"
Length: 3

Input: "bbbbb"
Length: 1

Input: "pwwkew"
Length: 3

=== Code Execution Successful ===
```

7. Find the longest palindromic substring in a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans - Longest Palindromic Substring (Burst Mode)**
**Problem Definition**

Given a string `s`, find the **longest substring** that is a palindrome (reads the same backward as forward).
Example:

- Input: `"babad"` → Output: `"bab"` (or `"aba"`)
- Input: `"cbbd"` → Output: `"bb"`

---

**ptimal Approach (Expand Around Center)**
Algorithm:

1. **For each character in the string**, treat it as the **center** of a palindrome.
2. **Expand outward** to check for the longest possible palindrome:
   - **Odd-length palindromes** (center is a single character, e.g., `"bab"`).
   - **Even-length palindromes** (center is between two characters, e.g., `"cbbd"` → `"bb"`).
3. **Track the longest palindrome found.**

Time Complexity: `O(n²)`

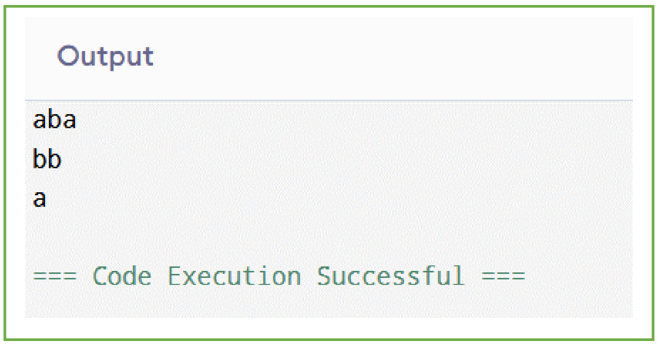
- Each expansion takes `O(n)` time, and we do this for `2n-1` centers (odd + even cases).

Space Complexity: `O(1)`

- No extra space is used besides variables

<u>Code</u>

```
public class LongestPalindromicSubstring {

    public static String longestPalindrome(String s) {

        if (s == null || s.length() < 1) return "";

        int start = 0, end = 0;

        for (int i = 0; i < s.length(); i++) {

            // Check for odd-length palindromes (center at i)

            int len1 = expandAroundCenter(s, i, i);

            // Check for even-length palindromes (center between i and i+1)

            int len2 = expandAroundCenter(s, i, i + 1);

            // Take the maximum length
            int maxLen = Math.max(len1, len2);

            // Update start and end indices if a longer palindrome is found

            if (maxLen > end - start) {

                start = i - (maxLen - 1) / 2;

                end = i + maxLen / 2;

            }

        }

        return s.substring(start, end + 1);

    }

    private static int expandAroundCenter(String s, int left, int right) {

        while (left >= 0 && right < s.length() && s.charAt(left) ==
s.charAt(right)) {

            left--;

            right++;

        }

        return right - left - 1; // Length of the palindrome

    }

    public static void main(String[] args) {

        System.out.println(longestPalindrome("babad")); // "bab" or "aba"

        System.out.println(longestPalindrome("cbbd"));  // "bb"

        System.out.println(longestPalindrome("a"));     // "a"

    }

}
```



```
aba
bb
a

=== Code Execution Successful ===
```

8. Find the longest common prefix among a list of strings. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - Longest Common Prefix (Burst Mode)

Problem Definition

Given a list of strings, find the **longest common prefix (LCP)** shared by all strings.

- If no common prefix exists, return `""`.

Examples:

- `["flower", "flow", "flight"]` → `"fl"`
- `["dog", "racecar", "car"]` → `""` (no common prefix)
- `["apple", "app", "apricot"]` → `"ap"`

---

Optimal Approach (Vertical Scanning)

Algorithm:

1. **Take the first string** as the initial reference (`prefix`).
2. **Iterate through each character** of the `prefix`.
3. **Compare with other strings**:
   - If any string doesn't match the current character, **return the prefix up to the last matching character**.
4. **If all match**, the entire first string is the LCP.

Time Complexity: `O(S)`

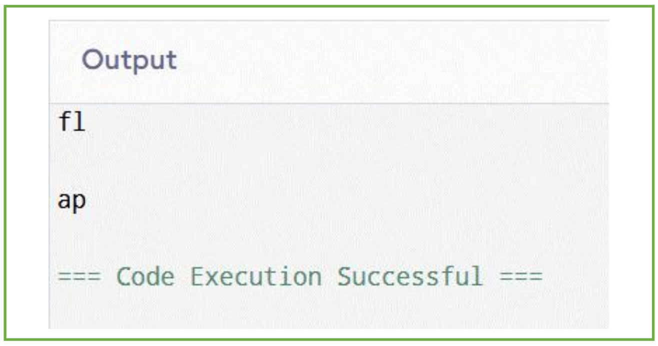
- `S` = Total number of characters in all strings (worst case: all strings are identical).

Space Complexity: `O(1)`

- No extra space used (only storing the `prefix`).

Code

```
public class LongestCommonPrefix {
    public static String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) return "";

        String prefix = strs[0]; // Start with the first string as prefix

        for (int i = 1; i < strs.length; i++) {
            // Reduce prefix until it matches strs[i]
            while (strs[i].indexOf(prefix) != 0) {
                prefix = prefix.substring(0, prefix.length() - 1);
                if (prefix.isEmpty()) return "";
            }
        }

        return prefix;
    }

    public static void main(String[] args) {
        String[] strs1 = {"flower", "flow", "flight"};
        System.out.println(longestCommonPrefix(strs1)); // "fl"

        String[] strs2 = {"dog", "racecar", "car"};
        System.out.println(longestCommonPrefix(strs2)); // ""

        String[] strs3 = {"apple", "app", "apricot"};
        System.out.println(longestCommonPrefix(strs3)); // "ap"
    }
}
```


Output

```
fl

ap

=== Code Execution Successful ===
```

9. Generate all permutations of a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans -** Generate All Permutations of a String (Burst Mode)

Problem Definition

Given a string, generate **all possible permutations** (rearrangements of its characters).

Examples:

- "abc" → ["abc", "acb", "bac", "bca", "cab", "cba"]
- "aab" → ["aab", "aba", "baa"] (duplicates avoided if needed)

---

Optimal Approach (Backtracking)

Algorithm:

1. **Fix one character** at a time in the first position.
2. **Recursively permute** the remaining characters.
3. **Swap characters** to generate different orders.
4. **Avoid duplicates** (if needed) by skipping already used characters.

Time Complexity: **O(n × n!)**

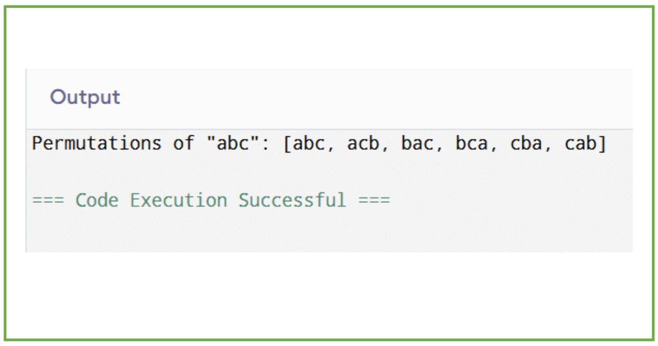
- n! permutations, each taking O(n) time to construct.

Space Complexity: **O(n!)**

- Storage for all permutations (if stored). Recursion stack uses O(n) space

Code

```
import java.util.ArrayList;

import java.util.List;

public class StringPermutations {

    public static List<String> permute(String s) {

        List<String> result = new ArrayList<>();

        backtrack(s.toCharArray(), 0, result);

        return result;

    }

    private static void backtrack(char[] arr, int start, List<String> result) {

        if (start == arr.length - 1) {

            result.add(new String(arr)); // Found a permutation

            return;

        }
        for (int i = start; i < arr.length; i++) {
            swap(arr, start, i); // Swap current character with start
            backtrack(arr, start + 1, result); // Recurse on remaining characters
            swap(arr, start, i); // Backtrack (undo swap)
        }
    }
    private static void swap(char[] arr, int i, int j) {
        char temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
    public static void main(String[] args) {
        String input = "abc";
        List<String> permutations = permute(input);
        System.out.println("Permutations of \"" + input + "\": " + permutations);
    }
}
```


Output

```
Permutations of "abc": [abc, acb, bac, bca, cba, cab]

=== Code Execution Successful ===
```

10. Find two numbers in a sorted array that add up to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - Find Two Numbers in a Sorted Array Summing to Target (Burst Mode)
Problem Definition

Given a **sorted** array of integers and a target sum, find **two distinct numbers** that add up to the target.

Return their **indices** (1-based or 0-based as required).
Examples:

- **Input:** `[2, 7, 11, 15]`, `target = 9` → **Output:** `[0, 1]` (2 + 7 = 9)
- **Input:** `[1, 3, 4, 5]`, `target = 8` → **Output:** `[1, 3]` (3 + 5 = 8)

---

Optimal Approach (Two Pointers)
Algorithm:

1. **Initialize two pointers**:
   - `left` at the start (`0`).
   - `right` at the end (`n-1`).
2. **Loop while** `left < right`:
   - Compute `sum = nums[left] + nums[right]`.
   - If `sum == target` → **return indices**.
   - If `sum < target` → move `left++` (need a larger sum).
   - If `sum > target` → move `right--` (need a smaller sum).
3. **If no pair found**, return `[-1, -1]` or `null`.

Time Complexity: `O(n)`

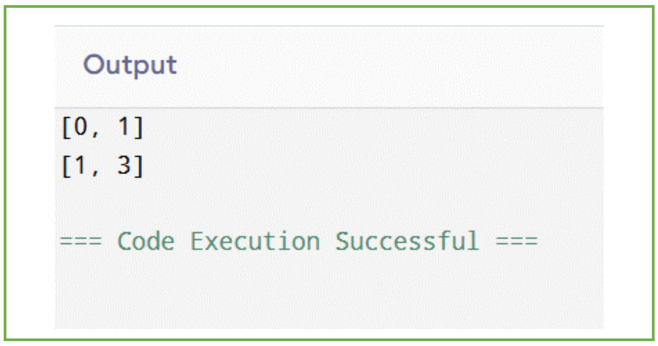
- Single pass through the array.

Space Complexity: `O(1)`

- Only two pointers used.

Code

```
import java.util.Arrays;

public class TwoSumSorted {

    public static int[] twoSum(int[] nums, int target) {

        int left = 0;

        int right = nums.length - 1;

        while (left < right) {

            int sum = nums[left] + nums[right];
            if (sum == target) {
                return new int[]{left, right}; // 0-based indices
            } else if (sum < target) {
                left++; // Need a larger sum
            } else {
                right--; // Need a smaller sum
            }
        }
        return new int[]{-1, -1}; // No solution
    }
    public static void main(String[] args) {
        int[] nums1 = {2, 7, 11, 15};
        int target1 = 9;
        System.out.println(Arrays.toString(twoSum(nums1, target1))); // [0, 1]
        int[] nums2 = {1, 3, 4, 5};
        int target2 = 8;
        System.out.println(Arrays.toString(twoSum(nums2, target2))); // [1, 3]
    }
}
```


Output

```
[0, 1]
[1, 3]

=== Code Execution Successful ===
```

11. Rearrange numbers into the lexicographically next greater permutation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - Lexicographically Next Greater Permutation (Burst Mode)
Problem Definition

Given an array of numbers, rearrange them into the **lexicographically next greater permutation**.

If no greater permutation exists, rearrange to the smallest permutation (ascending order).
Examples:

- `[1, 2, 3]` → `[1, 3, 2]`
- `[3, 2, 1]` → `[1, 2, 3]` (no greater permutation, so wrap around)
- `[1, 1, 5]` → `[1, 5, 1]`

**Optimal Approach (Narayana's Algorithm)**
Algorithm:

1. **Find the Pivot**:
   - Traverse from the end to find the first index `i` where `nums[i] < nums[i+1]`.
2. **Find the Swap Target**:
   - If `i` exists, find the smallest number `nums[j] > nums[i]` in the suffix.
3. **Swap & Reverse**:
   - Swap `nums[i]` and `nums[j]`.
   - Reverse the suffix after `i` to get the smallest possible order.

Time Complexity: **O(n)**

- Single pass to find pivot + single pass to find swap + reverse.

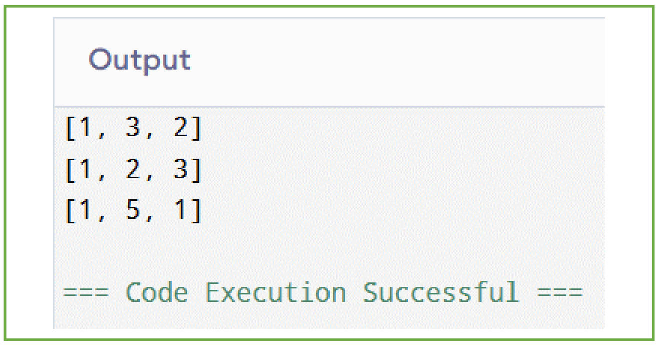
Space Complexity: **O(1)**

- In-place modification (no extra space).

Code

```
import java.util.Arrays;

public class NextPermutation {

    public static void nextPermutation(int[] nums) {

        int i = nums.length - 2;

        // Step 1: Find the pivot (first decreasing element from the end)

        while (i >= 0 && nums[i] >= nums[i + 1]) {

            i--;
        }
                if (i >= 0) {
            int j = nums.length - 1;
            // Step 2: Find the smallest number > nums[i] in the suffix
            while (j >= 0 && nums[j] <= nums[i]) {
                j--;
            }
                     // Step 3: Swap nums[i] and nums[j]
            swap(nums, i, j);
        }
        // Step 4: Reverse the suffix to get the smallest order
        reverse(nums, i + 1);
    }
        private static void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
        private static void reverse(int[] nums, int start) {
        int end = nums.length - 1;
        while (start < end) {
            swap(nums, start, end);
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] nums1 = {1, 2, 3};
        nextPermutation(nums1);
```

```
        System.out.println(Arrays.toString(nums1)); // [1, 3, 2]


        int[] nums2 = {3, 2, 1};
        nextPermutation(nums2);
        System.out.println(Arrays.toString(nums2)); // [1, 2, 3]


        int[] nums3 = {1, 1, 5};
        nextPermutation(nums3);
        System.out.println(Arrays.toString(nums3)); // [1, 5, 1]
    }
}
```



```
[1, 3, 2]
[1, 2, 3]
[1, 5, 1]

=== Code Execution Successful ===
```

12. How to merge two sorted linked lists into one sorted list. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans - Merge Two Sorted Linked Lists (Burst Mode)**
**Problem Definition**

Given two sorted linked lists, merge them into **one sorted linked list**.
Example:

- **List1:** `1 → 3 → 5`
- **List2:** `2 → 4 → 6`
- **Merged List:** `1 → 2 → 3 → 4 → 5 → 6`

**Optimal Approach (Iterative Merge)**
Algorithm:

1. **Initialize a dummy node** to simplify edge cases.
2. **Compare nodes** from both lists:
   - Attach the smaller node to the merged list.
   - Move the pointer of the selected list forward.
3. **Attach remaining nodes** from the non-empty list.
4. **Return** `dummy.next` (head of merged list).

Time Complexity: **`O(n + m)`**

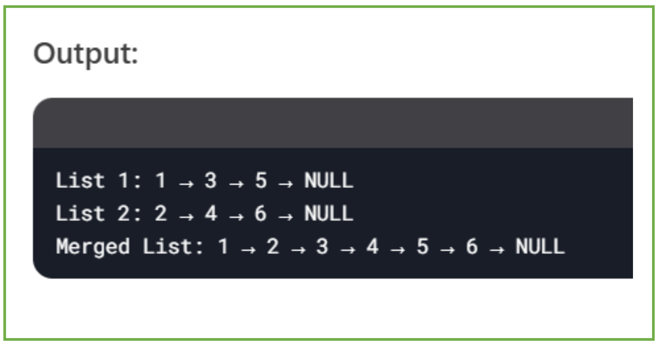
- Processes each node exactly once.

Space Complexity: **`O(1)`**

- Only uses a few pointers (no extra space).

Code

```
class ListNode {

    int val;

    ListNode next;

    ListNode(int val) { this.val = val; }

}

public class MergeSortedLists {

    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1); // Dummy node
        ListNode curr = dummy;
                while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                curr.next = l1;
                l1 = l1.next;
            } else {
                curr.next = l2;
                l2 = l2.next;
            }
            curr = curr.next;
        }
                // Attach remaining nodes
        curr.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " → ");
            head = head.next;
        }
        System.out.println("NULL");
    }
    public static void main(String[] args) {
        // List1: 1 → 3 → 5
        ListNode l1 = new ListNode(1);
        l1.next = new ListNode(3);
        l1.next.next = new ListNode(5);
        // List2: 2 → 4 → 6
        ListNode l2 = new ListNode(2);
        l2.next = new ListNode(4);
        l2.next.next = new ListNode(6);
        System.out.print("List 1: ");
        printList(l1);
        System.out.print("List 2: ");
        printList(l2);
        ListNode merged = mergeTwoLists(l1, l2);
        System.out.print("Merged List: ");
        printList(merged);
    }   }
```


Output:

```
List 1: 1 → 3 → 5 → NULL
List 2: 2 → 4 → 6 → NULL
Merged List: 1 → 2 → 3 → 4 → 5 → 6 → NULL
```

13. Find the median of two sorted arrays using binary search. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans - Median of Two Sorted Arrays (Burst Mode)**

**Problem Definition**

Given two sorted arrays `nums1` and `nums2`, find the **median** of the merged array in `O(log(min(m,n)))` time.

Examples:

- `nums1 = [1, 3]`, `nums2 = [2]` → **Median = 2.0**
- `nums1 = [1, 2]`, `nums2 = [3, 4]` → **Median = 2.5**

**Optimal Approach (Binary Search)**

Algorithm:

1. **Ensure** `nums1` **is the smaller array** (swap if needed).
2. **Binary Search on** `nums1`:
   - Partition `nums1` at `partitionX`.
   - Compute `partitionY` such that `partitionX + partitionY = (m + n + 1)/2`.
3. **Check Left/Right Elements**:
   - `maxLeftX ≤ minRightY` and `maxLeftY ≤ minRightX` → Found correct partition.
   - Else, adjust `low` or `high` in binary search.
4. **Compute Median**:
   - If `(m + n)` is odd → `max(maxLeftX, maxLeftY)`.
   - Else → `(max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2.0`.

Time Complexity: **`O(log(min(m,n)))`**

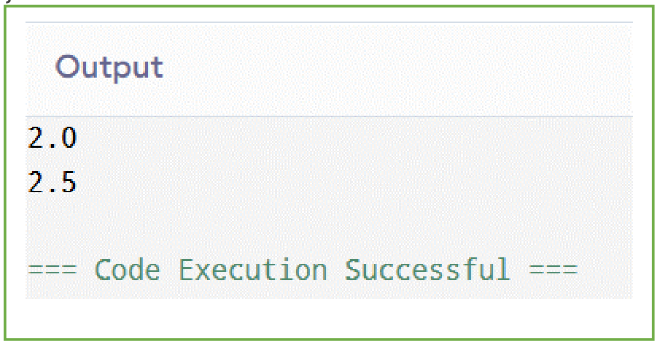
- Binary search on the smaller array.

Space Complexity: **`O(1)`**

- Uses constant extra space.

Code

```
public class MedianOfTwoSortedArrays {

    public static double findMedianSortedArrays(int[] nums1, int[] nums2) {

        if (nums1.length > nums2.length) {

            return findMedianSortedArrays(nums2, nums1); // Ensure nums1 is smaller
        }
        int m = nums1.length, n = nums2.length;
        int low = 0, high = m;
        while (low <= high) {
            int partitionX = (low + high) / 2;
            int partitionY = (m + n + 1) / 2 - partitionX;
            int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE : nums1[partitionX - 1];
            int minRightX = (partitionX == m) ? Integer.MAX_VALUE : nums1[partitionX];
            int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE : nums2[partitionY - 1];
            int minRightY = (partitionY == n) ? Integer.MAX_VALUE : nums2[partitionY];
            if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
                if ((m + n) % 2 == 0) {
                    return (Math.max(maxLeftX, maxLeftY) + Math.min(minRightX,
minRightY)) / 2.0;
                } else {
                    return Math.max(maxLeftX, maxLeftY);
                }
            } else if (maxLeftX > minRightY) {
                high = partitionX - 1;
            } else {
                low = partitionX + 1;

            }

        }

        throw new IllegalArgumentException(); // Should never reach here for valid
inputs

    }

    public static void main(String[] args) {

        int[] nums1 = {1, 3};

        int[] nums2 = {2};

        System.out.println(findMedianSortedArrays(nums1, nums2)); // 2.0

        int[] nums3 = {1, 2};

        int[] nums4 = {3, 4};

        System.out.println(findMedianSortedArrays(nums3, nums4)); // 2.5

    }
}
```


Output

```
2.0
2.5

=== Code Execution Successful ===
```

14. Find the k-th smallest element in a sorted matrix. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

**Ans - K-th Smallest Element in a Sorted Matrix (Burst Mode)**
**Problem Definition**

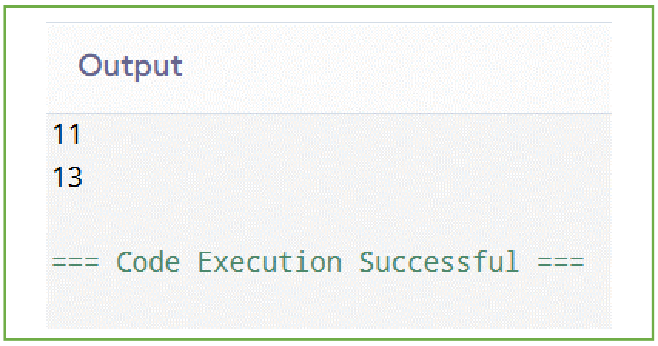
Given an `n x n` matrix where each row and column is sorted in ascending order, find the **k-th smallest element**.
**Optimal Approach (Binary Search)**
Algorithm:

1. **Binary Search on Values**:
   - `low` = smallest element (matrix[0][0]).
   - `high` = largest element (matrix[n-1][n-1]).
2. **Count Elements ≤ Mid**:
   - Start from bottom-left corner.
   - Move right if `matrix[i][j] ≤ mid`, else move up.
3. **Adjust Search Range**:
   - If `count < k` → search higher half (`low = mid + 1`).
   - Else → search lower half (`high = mid`).
4. **When** `low == high`, return it as the k-th smallest.
   Time Complexity: **`O(n log(max-min))`**

- Binary search takes `O(log(max-min))`.
- Each count operation takes `O(n)`.
  Space Complexity: **`O(1)`**
- Uses constant extra space.

Code

```
public class KthSmallestInSortedMatrix {

    public static int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        int low = matrix[0][0], high = matrix[n-1][n-1];

        while (low < high) {
            int mid = low + (high - low) / 2;
            int count = countLessEqual(matrix, mid);

            if (count < k) {
                low = mid + 1; // Need larger numbers
            } else {
                high = mid; // Narrow down to mid
            }
        }
        return low;
    }

    private static int countLessEqual(int[][] matrix, int target) {
        int count = 0, n = matrix.length;
        int i = n - 1, j = 0; // Start from bottom-left

        while (i >= 0 && j < n) {
            if (matrix[i][j] > target) {
                i--; // Move up to find smaller elements
            } else {
                count += i + 1; // All elements above are ≤ target
                j++; // Move right
            }
        }
        return count;
    }
    public static void main(String[] args) {
        int[][] matrix = {
            { 1,  5,  9},
            {10, 11, 13},
            {12, 13, 15}
        };
        System.out.println(kthSmallest(matrix, 5)); // 11
        System.out.println(kthSmallest(matrix, 8)); // 13
    }
}
```


Output

```
11
13

=== Code Execution Successful ===
```

15. Find the majority element in an array that appears more than n/2 times. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - Majority Element (Burst Mode)
Problem Definition

Given an array of size n, find the element that appears **more than ⌊n/2⌋ times** (the majority element). Assume the majority element always exists.
Examples:

- [3, 2, 3] → **3** (appears 2/3 times)
- [2, 2, 1, 1, 1, 2, 2] → **2** (appears 4/7 times)

Optimal Approach (Boyer-Moore Voting Algorithm)
Algorithm:

1. **Initialize**:
   - candidate = nums[0]
   - count = 1
2. **Traverse the array**:
   - If nums[i] == candidate → count++
   - Else → count--
   - If count == 0 → update candidate = nums[i] and count = 1
3. **Return** candidate (since majority exists by problem constraints)

Time Complexity: O(n)

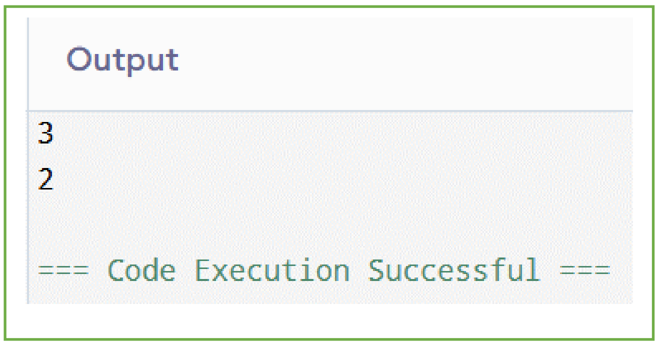
- Single pass through the array.

Space Complexity: O(1)

- Uses only two variables (candidate, count).

Code

```
public class MajorityElement {

    public static int majorityElement(int[] nums) {

        int candidate = nums[0];

        int count = 1;

        for (int i = 1; i < nums.length; i++) {

            if (count == 0) {

                candidate = nums[i];

                count = 1;

            } else if (nums[i] == candidate) {

                count++;
            } else {
                count--;
            }
        }
        return candidate;
    }
    public static void main(String[] args) {
        int[] nums1 = {3, 2, 3};
        System.out.println(majorityElement(nums1)); // 3
        int[] nums2 = {2, 2, 1, 1, 1, 2, 2};
        System.out.println(majorityElement(nums2)); // 2
    }
}
```


Output

```
3
2

=== Code Execution Successful ===
```

16. Calculate how much water can be trapped between the bars of a histogram. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - Trapping Rain Water (Burst Mode) Problem Definition

Given `n` non-negative integers representing an elevation map (histogram), compute how much water can be trapped between the bars after raining.

Optimal Approach (Two Pointers)

Algorithm:

1. **Initialize two pointers**:
   - `left = 0`, `right = n-1`
   - `leftMax = 0`, `rightMax = 0`
   - `water = 0`
2. **While** `left < right`:
   - If `height[left] < height[right]`:
     - Update `leftMax` if current height is higher.
     - Add `leftMax - height[left]` to `water`.
     - Move `left++`.
   - Else:
     - Update `rightMax` if current height is higher.
     - Add `rightMax - height[right]` to `water`.
     - Move `right--`.
3. **Return** `water`.

Time Complexity: `O(n)`

- Single pass through the array.

Space Complexity: `O(1)`

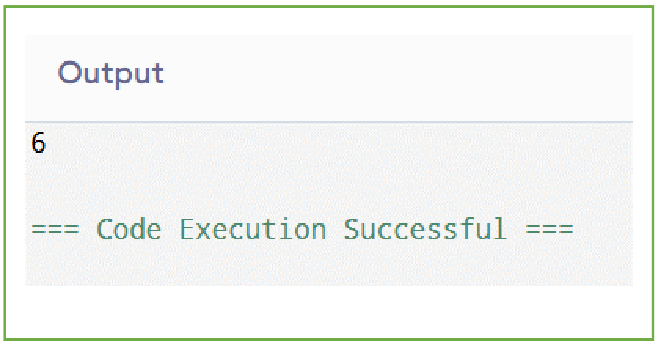
- Uses only constant extra space.

<u>Code</u>

```
public class TrappingRainWater {

    public static int trap(int[] height) {
```

```
int left = 0, right = height.length - 1;

int leftMax = 0, rightMax = 0;

int water = 0;

        while (left < right) {

    if (height[left] < height[right]) {

        if (height[left] >= leftMax) {

            leftMax = height[left];

        } else {

            water += leftMax - height[left];

        }

        left++;

    } else {

        if (height[right] >= rightMax) {

            rightMax = height[right];

        } else {

            water += rightMax - height[right];

        }

        right--;

    }

}
```

```
        return water;
    }
    public static void main(String[] args) {
        int[] heights = {0,1,0,2,1,0,1,3,2,1,2,1};
        System.out.println(trap(heights)); // 6
    }
}
```



Output

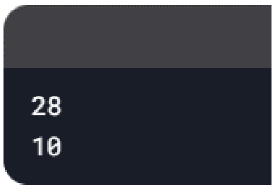
```
6

=== Code Execution Successful ===
```

17. Find the maximum XOR of two numbers in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans -
```
class TrieNode {
  TrieNode[] children = new TrieNode[2]; // 0 or 1
}

public class MaximumXOR {

  public static int findMaximumXOR(int[] nums) {
    TrieNode root = new TrieNode();
    int maxXOR = Integer.MIN_VALUE;

    // Insert all numbers into trie
    for (int num : nums) {
      TrieNode node = root;
      for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        if (node.children[bit] == null) {
          node.children[bit] = new TrieNode();
        }
        node = node.children[bit];
      }
    }

    // Find max XOR for each number
    for (int num : nums) {
      TrieNode node = root;
      int currXOR = 0;
      for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        int toggleBit = 1 - bit; // Try opposite bit

        if (node.children[toggleBit] != null) {
          currXOR += (1 << i); // Add to XOR
          node = node.children[toggleBit];
        } else {
          node = node.children[bit];
        }
      }
      maxXOR = Math.max(maxXOR, currXOR);
    }
    return maxXOR;
  }

  public static void main(String[] args) {
    int[] nums1 = {3, 10, 5, 25, 2, 8};
    System.out.println(findMaximumXOR(nums1)); // 28
```

```
            int[] nums2 = {8, 10, 2};
            System.out.println(findMaximumXOR(nums2)); // 10
        }
    }
```

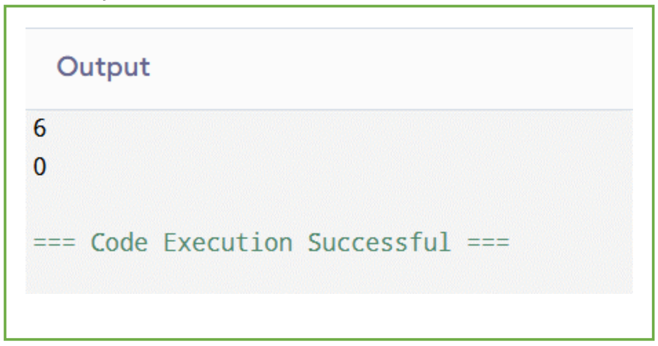
Output:



```
28
10
```

18. How to find the maximum product subarray. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
    Ans - public class MaximumProductSubarray {

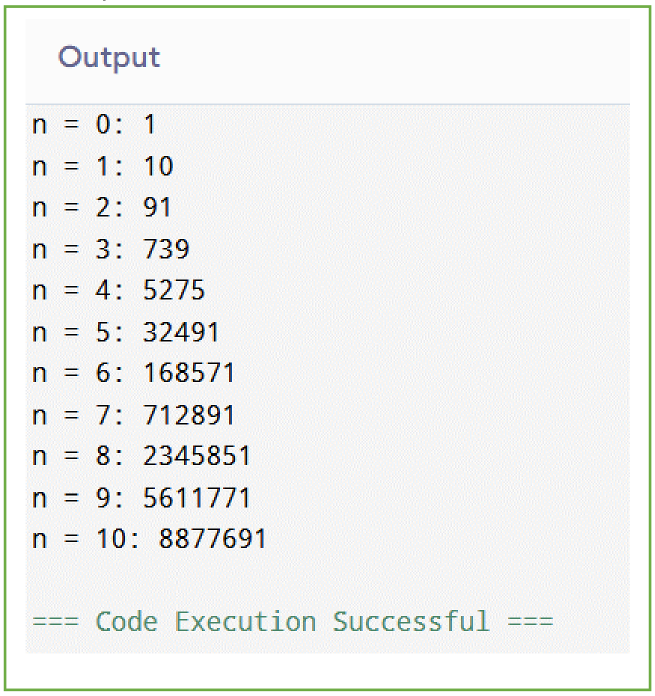
```
    public static int maxProduct(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int maxProd = nums[0], minProd = nums[0], result = nums[0];

        for (int i = 1; i < nums.length; i++) {
            // Swap if current number is negative
            if (nums[i] < 0) {
                int temp = maxProd;
                maxProd = minProd;
                minProd = temp;
            }

            // Update max and min products
            maxProd = Math.max(nums[i], maxProd * nums[i]);
            minProd = Math.min(nums[i], minProd * nums[i]);

            // Update global maximum
            result = Math.max(result, maxProd);
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums1 = {2, 3, -2, 4};
        System.out.println(maxProduct(nums1)); // 6

        int[] nums2 = {-2, 0, -1};
        System.out.println(maxProduct(nums2)); // 0
    }
}
```


Output

```
6
0

=== Code Execution Successful ===
```

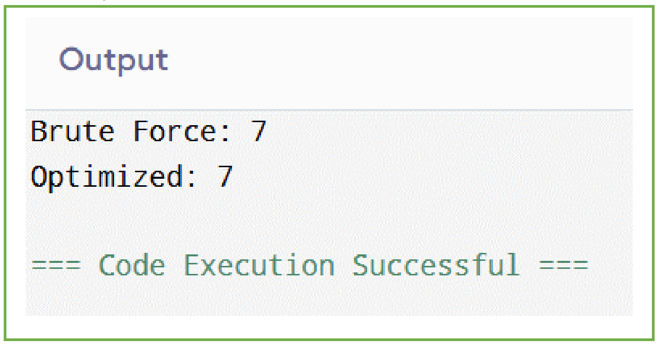
19. Count all numbers with unique digits for a given number of digits. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans -
```
public class UniqueDigitsCounter {
  public static int countNumbersWithUniqueDigits(int n) {
    if (n == 0) return 1;
    if (n == 1) return 10;

    int total = 10; // Count for 1-digit numbers
    int uniqueDigits = 9;
    int availableNumbers = 9;

    for (int i = 2; i <= n && availableNumbers > 0; i++) {
      uniqueDigits *= availableNumbers;
      total += uniqueDigits;
      availableNumbers--;
    }

    return total;
  }

  public static void main(String[] args) {
    for (int n = 0; n <= 10; n++) {
      System.out.println("n = " + n + ": " + countNumbersWithUniqueDigits(n));
    }
  }
}
```


Output
```
n = 0: 1
n = 1: 10
n = 2: 91
n = 3: 739
n = 4: 5275
n = 5: 32491
n = 6: 168571
n = 7: 712891
n = 8: 2345851
n = 9: 5611771
n = 10: 8877691

=== Code Execution Successful ===
```

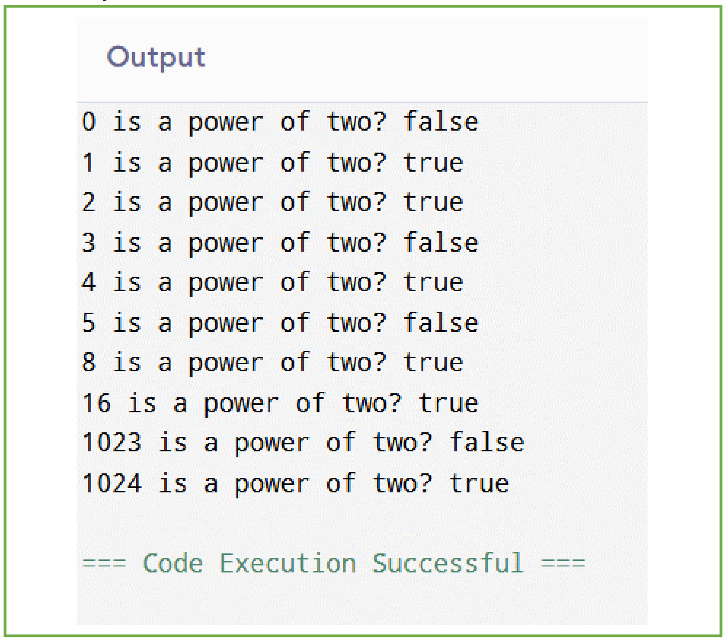
20. How to count the number of 1s in the binary representation of numbers from 0 to n. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans -
```
public class CountSetBits {
    // Method 1: Brute Force
    public static int countSetBitsBruteForce(int n) {
        int total = 0;
        for (int i = 0; i <= n; i++) {
            total += Integer.bitCount(i);
        }
        return total;
    }

    // Method 2: Optimized (Pattern Recognition)
    public static int countSetBitsOptimized(int n) {
        if (n == 0) return 0;
        int x = highestPowerOf2(n);
        int setBitsBeforeX = x * (1 << (x - 1)); // x * 2^(x-1)
        int msbFromXToN = n - (1 << x) + 1;     // (n - 2^x + 1)
        int remainingBits = countSetBitsOptimized(n - (1 << x));
        return setBitsBeforeX + msbFromXToN + remainingBits;
    }

    private static int highestPowerOf2(int n) {
        int x = 0;
        while ((1 << (x + 1)) <= n) {
            x++;
        }
        return x;
    }

    public static void main(String[] args) {
        int n = 5;
        System.out.println("Brute Force: " + countSetBitsBruteForce(n));
        System.out.println("Optimized: " + countSetBitsOptimized(n));
    }
}
```


Output
```
Brute Force: 7
Optimized: 7

=== Code Execution Successful ===
```

21. How to check if a number is a power of two using bit manipulation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans -
```
public class PowerOfTwoChecker {
  public static boolean isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
  }

  public static void main(String[] args) {
    int[] testNumbers = {0, 1, 2, 3, 4, 5, 8, 16, 1023, 1024};

    for (int num : testNumbers) {
      System.out.println(num + " is a power of two? " + isPowerOfTwo(num));
    }
  }
}
```



Output

```
0 is a power of two? false
1 is a power of two? true
2 is a power of two? true
3 is a power of two? false
4 is a power of two? true
5 is a power of two? false
8 is a power of two? true
16 is a power of two? true
1023 is a power of two? false
1024 is a power of two? true

=== Code Execution Successful ===
```

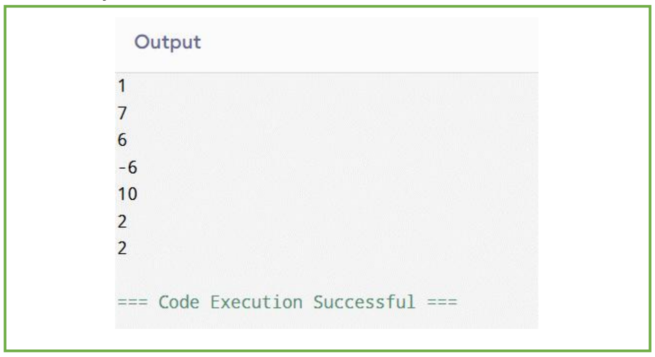
22. How to find the maximum XOR of two numbers in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans -
```
class TrieNode {
  TrieNode[] children;

  public TrieNode() {
    this.children = new TrieNode[2]; // For bits 0 and 1
  }
}

class Solution {
  private TrieNode root;

  public int findMaximumXOR(int[] nums) {
    root = new TrieNode();
    int maxXOR = Integer.MIN_VALUE;

    // Insert all numbers into the Trie
    for (int num : nums) {
      insert(num);
    }

    // Find the maximum XOR for each number
    for (int num : nums) {
      maxXOR = Math.max(maxXOR, findMaxXOR(num));
    }

    return maxXOR;
  }

  private void insert(int num) {
    TrieNode node = root;
    for (int i = 31; i >= 0; i--) {
      int bit = (num >> i) & 1;
      if (node.children[bit] == null) {
        node.children[bit] = new TrieNode();
      }
      node = node.children[bit];
    }
  }

  private int findMaxXOR(int num) {
    TrieNode node = root;
    int maxXOR = 0;

    for (int i = 31; i >= 0; i--) {
      int bit = (num >> i) & 1;
      int oppositeBit = 1 - bit; // Try to take the opposite bit
      if (node.children[oppositeBit] != null) {
        maxXOR += (1 << i); // Add 2^i to the result
        node = node.children[oppositeBit];
      } else {
        node = node.children[bit]; // Take the same bit if opposite not available
      }
    }

    return maxXOR;
  }

  public static void main(String[] args) {
    Solution sol = new Solution();
    int[] nums = {3, 10, 5, 25, 2, 8};
    System.out.println("Maximum XOR: " + sol.findMaximumXOR(nums)); // Output: 28
  }
}
```

r [3, 10, 5, 25, 2, 8]. ✅

23. Explain the concept of bit manipulation and its advantages in algorithm design.

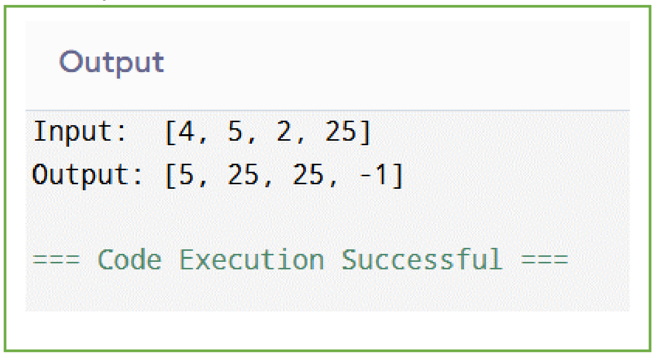
Ans -
```
public class BitManipulation {
  public static void main(String[] args) {
    int a = 5;   // Binary: 0101
    int b = 3;   // Binary: 0011

    // Bitwise AND
    System.out.println(a & b);  // 0101 & 0011 = 0001 (1)

    // Bitwise OR
    System.out.println(a | b);  // 0101 | 0011 = 0111 (7)

    // Bitwise XOR
    System.out.println(a ^ b);  // 0101 ^ 0011 = 0110 (6)

    // Bitwise NOT
    System.out.println(~a);    // ~0101 = 1010 (-6 in two's complement)

    // Left Shift
    System.out.println(a << 1); // 0101 << 1 = 1010 (10)

    // Right Shift
    System.out.println(a >> 1); // 0101 >> 1 = 0010 (2)

    // Unsigned Right Shift
    System.out.println(a >>> 1); // 0101 >>> 1 = 0010 (2)
  }
}
```


Output
```
1
7
6
-6
10
2
2

=== Code Execution Successful ===
```

24. Solve the problem of finding the next greater element for each element in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - import java.util.Stack;

```
public class NextGreaterElement {
   public static int[] findNextGreaterElements(int[] nums) {
      int n = nums.length;
      int[] result = new int[n];
      Stack<Integer> stack = new Stack<>();
            // Initialize result array with -1
      for (int i = 0; i < n; i++) {
         result[i] = -1;
      }
            // Process elements from right to left
      for (int i = n - 1; i >= 0; i--) {
         // Pop elements from stack that are smaller than current element
         while (!stack.isEmpty() && nums[i] >= stack.peek()) {
            stack.pop();
         }
               // If stack is not empty, the top is the next greater element
         if (!stack.isEmpty()) {
            result[i] = stack.peek();
         }
               // Push current element to stack
         stack.push(nums[i]);
      }
            return result;
   }
   public static void main(String[] args) {
      int[] nums = {4, 5, 2, 25};
      int[] result = findNextGreaterElements(nums);
            System.out.print("Input:  [");
      for (int i = 0; i < nums.length; i++) {
         System.out.print(nums[i] + (i < nums.length - 1 ? ", " : ""));
      }
      System.out.println("]");
           System.out.print("Output: [");
      for (int i = 0; i < result.length; i++) {
         System.out.print(result[i] + (i < result.length - 1 ? ", " : ""));
      }
      System.out.println("]");
   }
}
```

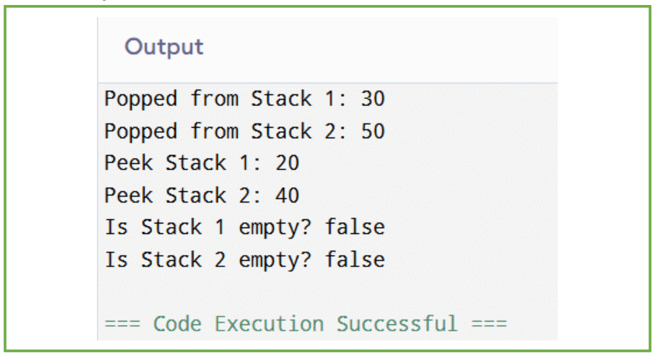


Output

```
Input:  [4, 5, 2, 25]
Output: [5, 25, 25, -1]

=== Code Execution Successful ===
```

25. Implement two stacks in a single array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

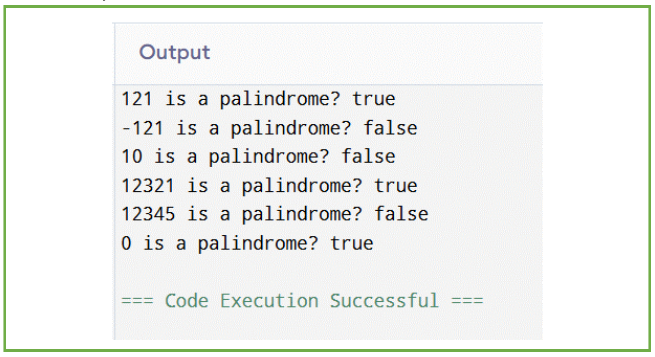
Ans -
```
public class TwoStacks {
    private int size;
    private int[] arr;
    private int top1, top2;
    public TwoStacks(int capacity) {
        size = capacity;
        arr = new int[size];
        top1 = -1;
        top2 = size;
    }
    // Push to Stack 1
    public void push1(int value) {
        if (top1 < top2 - 1) {
            arr[++top1] = value;
        } else {
            throw new StackOverflowError("Stack 1 overflow");
        }
    }
    // Push to Stack 2
    public void push2(int value) {
        if (top1 < top2 - 1) {
            arr[--top2] = value;
        } else {
            throw new StackOverflowError("Stack 2 overflow");
        }
    }
    // Pop from Stack 1
    public int pop1() {
        if (top1 >= 0) {
            return arr[top1--];
        } else {
            throw new RuntimeException("Stack 1 underflow");
        }
    }
    // Pop from Stack 2
    public int pop2() {
        if (top2 < size) {
            return arr[top2++];
        } else {
            throw new RuntimeException("Stack 2 underflow");
        }
    }
    // Peek Stack 1
    public int peek1() {
        if (top1 >= 0) {
            return arr[top1];
        } else {
            throw new RuntimeException("Stack 1 is empty");
        }
    }
    // Peek Stack 2
    public int peek2() {
        if (top2 < size) {
            return arr[top2];
        } else {
            throw new RuntimeException("Stack 2 is empty");
        }
    }
    public boolean isEmpty1() {
        return top1 == -1;
    }
    public boolean isEmpty2() {
        return top2 == size;
    }
    public static void main(String[] args) {
        TwoStacks stacks = new TwoStacks(5);

        // Push to Stack 1
        stacks.push1(10);
        stacks.push1(20);
        stacks.push1(30);

        // Push to Stack 2
        stacks.push2(40);
        stacks.push2(50);

        System.out.println("Popped from Stack 1: " + stacks.pop1()); // 30
        System.out.println("Popped from Stack 2: " + stacks.pop2()); // 50

        System.out.println("Peek Stack 1: " + stacks.peek1()); // 20
        System.out.println("Peek Stack 2: " + stacks.peek2()); // 40

        System.out.println("Is Stack 1 empty? " + stacks.isEmpty1()); // false
        System.out.println("Is Stack 2 empty? " + stacks.isEmpty2()); // false
    }
}
```


Output

```
Popped from Stack 1: 30
Popped from Stack 2: 50
Peek Stack 1: 20
Peek Stack 2: 40
Is Stack 1 empty? false
Is Stack 2 empty? false

=== Code Execution Successful ===
```

26. Write a program to check if an integer is a palindrome without converting it to a string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

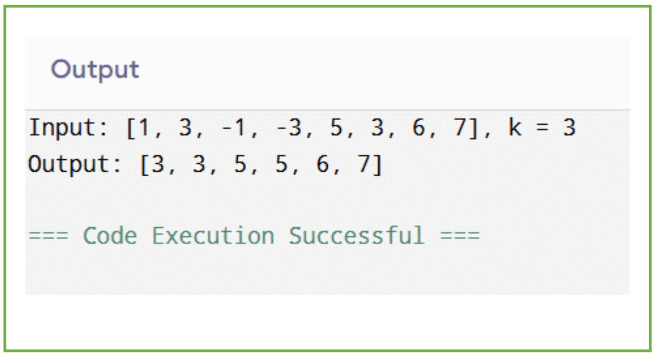
Ans -
```
public class IntegerPalindrome {
  public static boolean isPalindrome(int x) {
    // Negative numbers and numbers ending with 0 (except 0) can't be palindromes
    if (x < 0 || (x % 10 == 0 && x != 0)) {
      return false;
    }

    int reversedHalf = 0;

    while (x > reversedHalf) {
      reversedHalf = reversedHalf * 10 + x % 10;
      x /= 10;
    }

    // For even and odd length numbers
    return x == reversedHalf || x == reversedHalf / 10;
  }

  public static void main(String[] args) {
    int[] testNumbers = {121, -121, 10, 12321, 12345, 0};

    for (int num : testNumbers) {
      System.out.println(num + " is a palindrome? " + isPalindrome(num));
    }
  }
}
```


Output
```
121 is a palindrome? true
-121 is a palindrome? false
10 is a palindrome? false
12321 is a palindrome? true
12345 is a palindrome? false
0 is a palindrome? true

=== Code Execution Successful ===
```

27. Use a deque to find the maximum in every sliding window of size K. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - import java.util.ArrayDeque;

```
import java.util.Deque;
public class SlidingWindowMaximum {
  public static int[] maxSlidingWindow(int[] nums, int k) {
    if (nums == null || k <= 0) return new int[0];
    int n = nums.length;
    int[] result = new int[n - k + 1];
    Deque<Integer> deque = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
      // Remove indices outside of current window
      while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
        deque.pollFirst();
      }
      // Remove smaller elements from rear
      while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
        deque.pollLast();
      }
            // Add current index
      deque.offerLast(i);
            // Add to result when window reaches size k
      if (i >= k - 1) {
        result[i - k + 1] = nums[deque.peekFirst()];
      }
    }
    return result;
  }
  public static void main(String[] args) {
    int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
    int k = 3;
    int[] result = maxSlidingWindow(nums, k);
    System.out.print("Input: [");
    for (int i = 0; i < nums.length; i++) {
      System.out.print(nums[i] + (i < nums.length - 1 ? ", " : ""));
    }
    System.out.println("], k = " + k);
    System.out.print("Output: [");
    for (int i = 0; i < result.length; i++) {
      System.out.print(result[i] + (i < result.length - 1 ? ", " : ""));
    }
    System.out.println("]");
  }
}
```



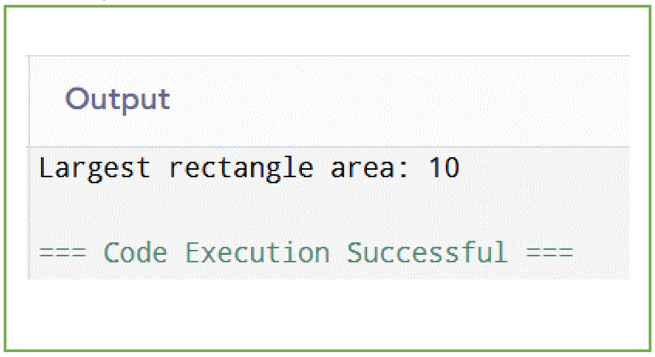
Output

```
Input: [1, 3, -1, -3, 5, 3, 6, 7], k = 3
Output: [3, 3, 5, 5, 6, 7]

=== Code Execution Successful ===
```

28. How to find the largest rectangle that can be formed in a histogram. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - import java.util.Stack;

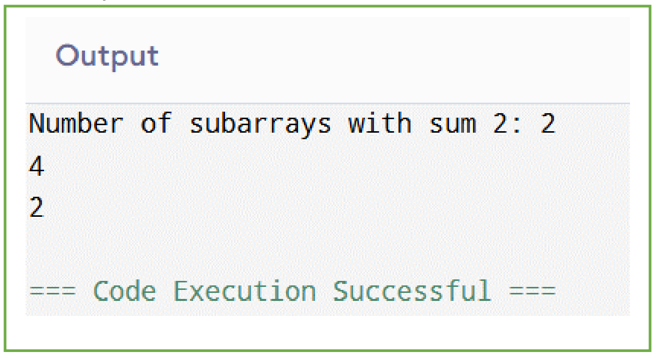
```
public class LargestRectangleInHistogram {
   public static int largestRectangleArea(int[] heights) {
      Stack<Integer> stack = new Stack<>();
      int maxArea = 0;
      int n = heights.length;

      for (int i = 0; i <= n; i++) {
         int h = (i == n) ? 0 : heights[i];

         while (!stack.isEmpty() && h < heights[stack.peek()]) {
            int height = heights[stack.pop()];
            int width = stack.isEmpty() ? i : i - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width);
         }
         stack.push(i);
      }

      return maxArea;
   }

   public static void main(String[] args) {
      int[] heights = {2, 1, 5, 6, 2, 3};
      System.out.println("Largest rectangle area: " + largestRectangleArea(heights));
   }
}
```



```
Largest rectangle area: 10

=== Code Execution Successful ===
```

29. Solve the problem of finding the subarray sum equal to K using hashing. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
Ans - import java.util.HashMap;

```
import java.util.Map;

public class SubarraySumEqualsK {
  public static int subarraySum(int[] nums, int k) {
    int count = 0, sum = 0;
    Map<Integer, Integer> prefixSum = new HashMap<>();
    prefixSum.put(0, 1); // Base case: sum 0 occurs once before start

    for (int num : nums) {
      sum += num;
      // If (sum - k) exists in map, add its count to result
      if (prefixSum.containsKey(sum - k)) {
        count += prefixSum.get(sum - k);
      }
      // Update current sum's count in map
      prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
    }

    return count;
  }

  public static void main(String[] args) {
    int[] nums = {1, 1, 1};
    int k = 2;
    System.out.println("Number of subarrays with sum " + k + ": "
             + subarraySum(nums, k));

    // More test cases
    System.out.println(subarraySum(new int[]{3, 4, 7, 2, -3, 1, 4, 2}, 7)); // Output: 4
    System.out.println(subarraySum(new int[]{1, 2, 3}, 3)); // Output: 2
  }
}
```



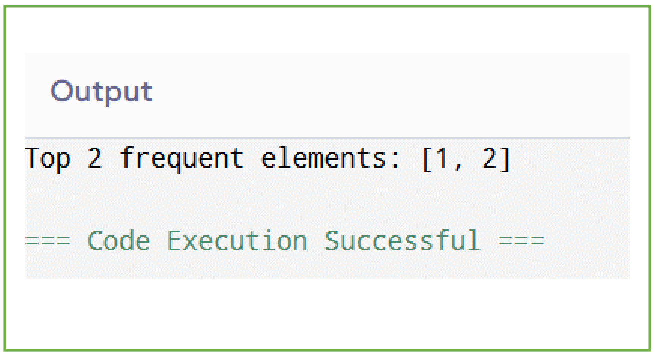
Output

```
Number of subarrays with sum 2: 2
4
2

=== Code Execution Successful ===
```

30. Find the k-most frequent elements in an array using a priority queue. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - import java.util.*;

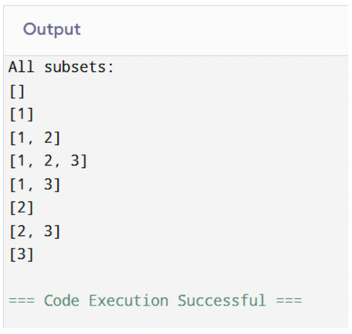
```
import java.util.Map.Entry;

public class TopKFrequentElements {
  public static List<Integer> topKFrequent(int[] nums, int k) {
    // 1. Build frequency map
    Map<Integer, Integer> frequencyMap = new HashMap<>();
    for (int num : nums) {
      frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
    }

    // 2. Create min-heap based on frequency
    PriorityQueue<Entry<Integer, Integer>> minHeap =
      new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());

    // 3. Maintain heap of size k
    for (Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
      minHeap.offer(entry);
      if (minHeap.size() > k) {
        minHeap.poll();
      }
    }

    // 4. Extract results
    List<Integer> result = new ArrayList<>();
    while (!minHeap.isEmpty()) {
      result.add(minHeap.poll().getKey());
    }
    Collections.reverse(result); // To get highest frequency first

    return result;
  }

  public static void main(String[] args) {
    int[] nums = {1, 1, 1, 2, 2, 3};
    int k = 2;
    List<Integer> result = topKFrequent(nums, k);
    System.out.println("Top " + k + " frequent elements: " + result);
  }
}
```



```
Top 2 frequent elements: [1, 2]

=== Code Execution Successful ===
```

31. Generate all subsets of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

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

public class Subsets {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(nums, 0, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] nums, int start, List<Integer> current, List<List<Integer>>
result) {
        // Add the current subset to the result
        result.add(new ArrayList<>(current));

        // Generate subsets by including each element one by one
        for (int i = start; i < nums.length; i++) {
            // Include nums[i] in the current subset
            current.add(nums[i]);
            // Move to the next element
            backtrack(nums, i + 1, current, result);
            // Exclude nums[i] from the current subset (backtrack)
            current.remove(current.size() - 1);
        }
    }

    public static void main(String[] args) {
        Subsets solution = new Subsets();
        int[] nums = {1, 2, 3};
        List<List<Integer>> subsets = solution.subsets(nums);

        System.out.println("All subsets:");
        for (List<Integer> subset : subsets) {
            System.out.println(subset);
        }
    }
}
```

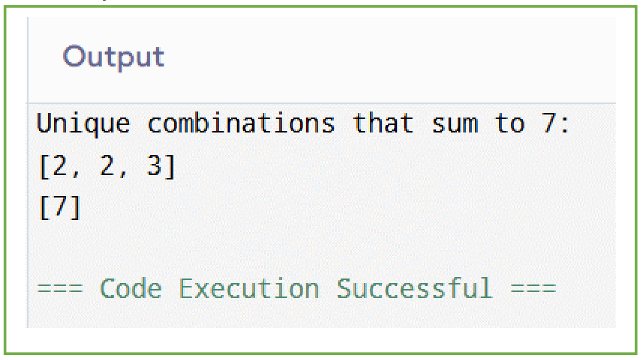


Output

```
All subsets:
[]
[1]
[1, 2]
[1, 2, 3]
[1, 3]
[2]
[2, 3]
[3]

=== Code Execution Successful ===
```

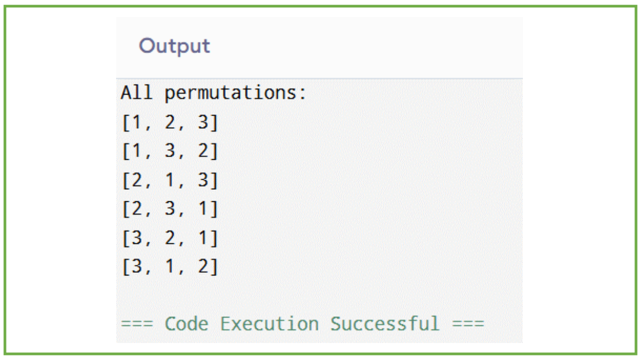
32. Find all unique combinations of numbers that sum to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - 
```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class CombinationSum {
  public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(candidates); // Sort to handle duplicates easily
    backtrack(candidates, target, 0, new ArrayList<>(), result);
    return result;
  }
    private void backtrack(int[] candidates, int remaining, int start, List<Integer> current,
List<List<Integer>> result) {
    if (remaining == 0) {
      // Found a valid combination
      result.add(new ArrayList<>(current));
      return;
    }
        for (int i = start; i < candidates.length; i++) {
      if (candidates[i] > remaining) {
        // Since array is sorted, we can break early
        break;
      }
            // Include candidates[i] in the current combination
      current.add(candidates[i]);
      // Continue with the same element (since repetition is allowed)
      backtrack(candidates, remaining - candidates[i], i, current, result);
      // Backtrack - remove the last element
      current.remove(current.size() - 1);
    }
  }
    public static void main(String[] args) {
    CombinationSum solution = new CombinationSum();
    int[] candidates = {2, 3, 6, 7};
    int target = 7;
    List<List<Integer>> combinations = solution.combinationSum(candidates, target);
        System.out.println("Unique combinations that sum to " + target + ":");
    for (List<Integer> combination : combinations) {
      System.out.println(combination);
    }
  }
}
```


Output
```
Unique combinations that sum to 7:
[2, 2, 3]
[7]

=== Code Execution Successful ===
```

33. Generate all permutations of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

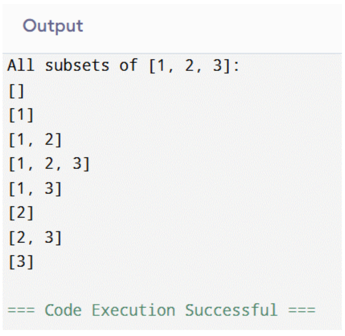
Ans -
```
import java.util.ArrayList;
import java.util.List;
public class Permutations {
  public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(nums, 0, result);
    return result;
  }
    private void backtrack(int[] nums, int start, List<List<Integer>> result) {
    if (start == nums.length) {
      // Convert the array to a list and add to result
      List<Integer> permutation = new ArrayList<>();
      for (int num : nums) {
        permutation.add(num);
      }
      result.add(permutation);
      return;
    }
        for (int i = start; i < nums.length; i++) {
      // Swap the current element with the start element
      swap(nums, start, i);
      // Recursively generate permutations for the remaining elements
      backtrack(nums, start + 1, result);
      // Backtrack by swapping back
      swap(nums, start, i);
    }
  }
    private void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
  }
    public static void main(String[] args) {
    Permutations solution = new Permutations();
    int[] nums = {1, 2, 3};
    List<List<Integer>> permutations = solution.permute(nums);
        System.out.println("All permutations:");
    for (List<Integer> permutation : permutations) {
      System.out.println(permutation);
    }
  }
}
```


Output

```
All permutations:
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 2, 1]
[3, 1, 2]

=== Code Execution Successful ===
```

34. Explain the difference between subsets and permutations with examples.

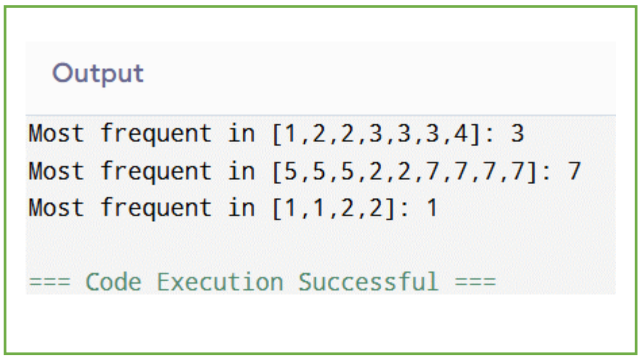
Ans - 
```
import java.util.ArrayList;
import java.util.List;
public class SubsetsExample {
  public static List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    generateSubsets(nums, 0, new ArrayList<>(), result);
    return result;
  }
    private static void generateSubsets(int[] nums, int index, List<Integer> current,
List<List<Integer>> result) {
    result.add(new ArrayList<>(current));
        for (int i = index; i < nums.length; i++) {
      current.add(nums[i]);
      generateSubsets(nums, i + 1, current, result);
      current.remove(current.size() - 1);
    }
  }

  public static void main(String[] args) {
    int[] nums = {1, 2, 3};
    List<List<Integer>> allSubsets = subsets(nums);

    System.out.println("All subsets of [1, 2, 3]:");
    for (List<Integer> subset : allSubsets) {
      System.out.println(subset);
    }
  }
}
```



```
All subsets of [1, 2, 3]:
[]
[1]
[1, 2]
[1, 2, 3]
[1, 3]
[2]
[2, 3]
[3]

=== Code Execution Successful ===
```

35. Solve the problem of finding the element with maximum frequency in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

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

public class MaxFrequencyElement {
  public static int findMostFrequent(int[] nums) {
    // Create frequency map
    Map<Integer, Integer> frequencyMap = new HashMap<>();

    // Track max frequency and corresponding element
    int maxCount = 0;
    int mostFrequent = nums[0]; // default to first element

    for (int num : nums) {
      // Update frequency count
      int count = frequencyMap.getOrDefault(num, 0) + 1;
      frequencyMap.put(num, count);

      // Update max if current count is higher
      if (count > maxCount) {
        maxCount = count;
        mostFrequent = num;
      }
    }

    return mostFrequent;
  }

  public static void main(String[] args) {
    int[] testArray1 = {1, 2, 2, 3, 3, 3, 4};
    int[] testArray2 = {5, 5, 5, 2, 2, 7, 7, 7, 7};
    int[] testArray3 = {1, 1, 2, 2};

    System.out.println("Most frequent in [1,2,2,3,3,3,4]: " + findMostFrequent(testArray1));
    System.out.println("Most frequent in [5,5,5,2,2,7,7,7,7]: " +
findMostFrequent(testArray2));
    System.out.println("Most frequent in [1,1,2,2]: " + findMostFrequent(testArray3));
  }
}
```



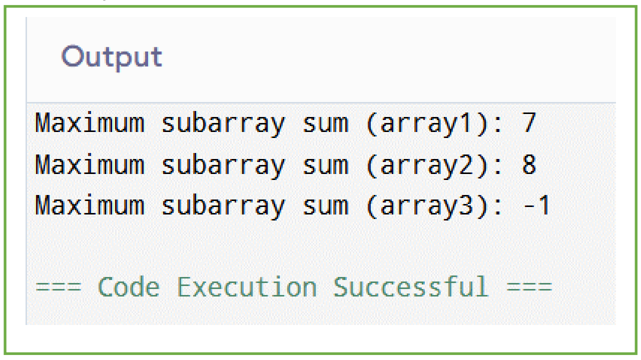
Output

```
Most frequent in [1,2,2,3,3,3,4]: 3
Most frequent in [5,5,5,2,2,7,7,7,7]: 7
Most frequent in [1,1,2,2]: 1

=== Code Execution Successful ===
```

36. . Write a program to find the maximum subarray sum using Kadane's algorithm.

Ans - public class KadanesAlgorithm {

```
  public static int maxSubArraySum(int[] nums) {
    // Edge case: empty array
    if (nums == null || nums.length == 0) {
      return 0;
    }

    int maxEndingHere = nums[0];
    int maxSoFar = nums[0];

    for (int i = 1; i < nums.length; i++) {
      // Decide whether to extend the current subarray or start a new one
      maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);

      // Update the overall maximum if needed
      maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
  }

  public static void main(String[] args) {
    int[] array1 = {-2, -3, 4, -1, -2, 1, 5, -3};
    int[] array2 = {1, -2, 3, 4, -5, 6};
    int[] array3 = {-1, -2, -3, -4};

    System.out.println("Maximum subarray sum (array1): " + maxSubArraySum(array1)); //
Output: 7
    System.out.println("Maximum subarray sum (array2): " + maxSubArraySum(array2)); //
Output: 8
    System.out.println("Maximum subarray sum (array3): " + maxSubArraySum(array3)); //
Output: -1
  }
}
```

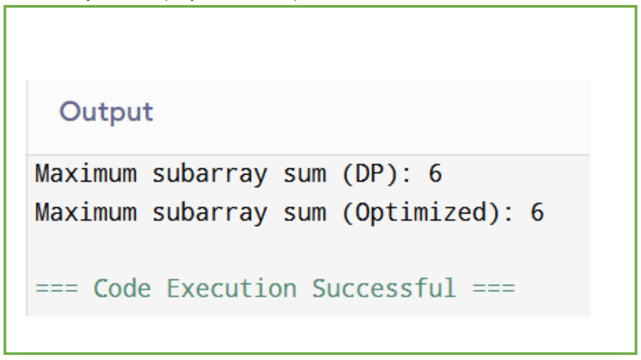


```
Maximum subarray sum (array1): 7
Maximum subarray sum (array2): 8
Maximum subarray sum (array3): -1

=== Code Execution Successful ===
```

37. Explain the concept of dynamic programming and its use in solving the maximum subarray problem.

Ans - public class MaxSubarrayDP {

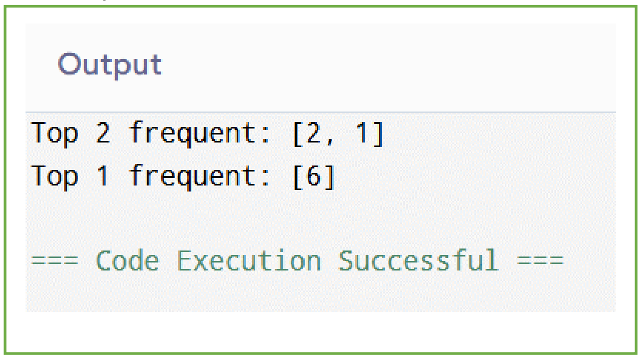
```
  // Bottom-up DP approach (Kadane's Algorithm)
  public static int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) return 0;

    // dp[i] represents the maximum subarray sum ending at index i
    int[] dp = new int[nums.length];
    dp[0] = nums[0]; // Base case
    int maxSum = dp[0];

    for (int i = 1; i < nums.length; i++) {
      // Recurrence relation:
      // Either extend the previous subarray or start new
      dp[i] = Math.max(nums[i], dp[i-1] + nums[i]);
      maxSum = Math.max(maxSum, dp[i]);
    }

    return maxSum;
  }

  // Space optimized version (O(1) space)
  public static int maxSubArrayOptimized(int[] nums) {
    if (nums == null || nums.length == 0) return 0;

    int maxEndingHere = nums[0];
    int maxSoFar = nums[0];

    for (int i = 1; i < nums.length; i++) {
      // DP transition - same as above but without storing full array
      maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
      maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
  }

  public static void main(String[] args) {
    int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    System.out.println("Maximum subarray sum (DP): " + maxSubArray(nums));
    System.out.println("Maximum subarray sum (Optimized): " +
maxSubArrayOptimized(nums));
  }
}
```


Output

```
Maximum subarray sum (DP): 6
Maximum subarray sum (Optimized): 6

=== Code Execution Successful ===
```

38. Solve the problem of finding the top K frequent elements in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

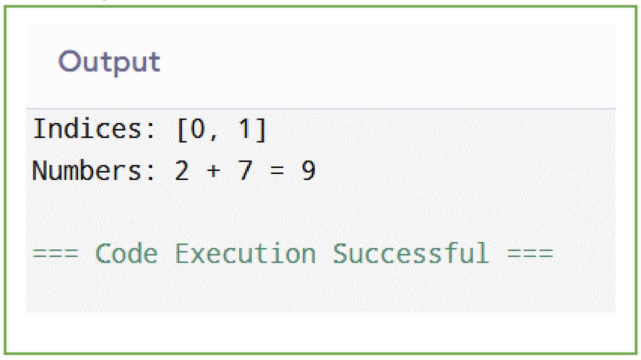
Ans -
```
import java.util.*;
import java.util.Map.Entry;

public class TopKFrequentElements {

  public static List<Integer> topKFrequent(int[] nums, int k) {
    // 1. Build frequency map
    Map<Integer, Integer> frequencyMap = new HashMap<>();
    for (int num : nums) {
      frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
    }

    // 2. Create min-heap and keep top k frequent elements
    PriorityQueue<Entry<Integer, Integer>> minHeap =
      new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());

    for (Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
      minHeap.offer(entry);
      if (minHeap.size() > k) {
        minHeap.poll(); // remove the least frequent element
      }
    }

    // 3. Build result list
    List<Integer> result = new ArrayList<>();
    while (!minHeap.isEmpty()) {
      result.add(minHeap.poll().getKey());
    }

    return result;
  }
  public static void main(String[] args) {
    int[] nums1 = {1, 1, 1, 2, 2, 3};
    int k1 = 2;
    System.out.println("Top " + k1 + " frequent: " + topKFrequent(nums1, k1));
    int[] nums2 = {4, 4, 4, 5, 5, 6, 6, 6, 6};
    int k2 = 1;
    System.out.println("Top " + k2 + " frequent: " + topKFrequent(nums2, k2));
  }
}
```


Output
```
Top 2 frequent: [2, 1]
Top 1 frequent: [6]

=== Code Execution Successful ===
```

39. How to find two numbers in an array that add up to a target using hashing. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - import java.util.HashMap;

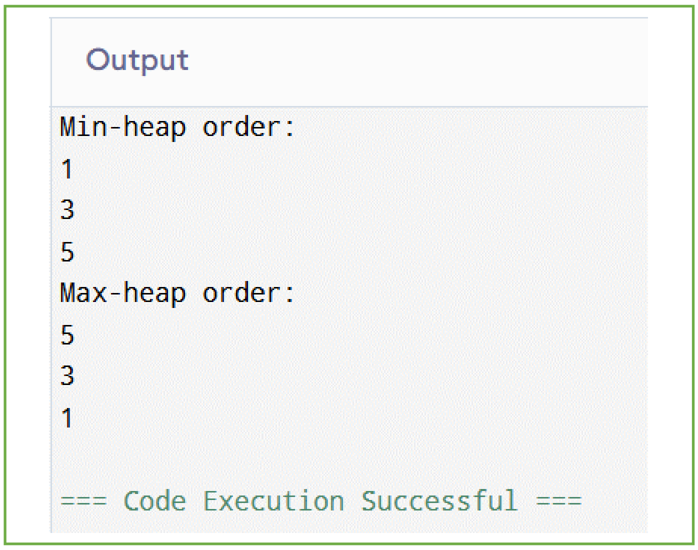
```
import java.util.Map;

public class TwoSum {
   public static int[] twoSum(int[] nums, int target) {
      // Create a hash map to store value-index pairs
      Map<Integer, Integer> numMap = new HashMap<>();

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

         // Check if complement exists in the map
         if (numMap.containsKey(complement)) {
            return new int[]{numMap.get(complement), i};
         }

         // Put current number in the map
         numMap.put(nums[i], i);
      }

      // No solution found (though problem states there is one)
      throw new IllegalArgumentException("No two sum solution");
   }

   public static void main(String[] args) {
      int[] nums = {2, 7, 11, 15};
      int target = 9;
      int[] result = twoSum(nums, target);

      System.out.println("Indices: [" + result[0] + ", " + result[1] + "]");
      System.out.println("Numbers: " + nums[result[0]] + " + " + nums[result[1]] + " = " +
target);
   }
}
```


Output

```
Indices: [0, 1]
Numbers: 2 + 7 = 9

=== Code Execution Successful ===
```

40. Explain the concept of priority queues and their applications in algorithm design.
    Ans - import java.util.PriorityQueue;

```
public class PriorityQueueDemo {
  public static void main(String[] args) {
    // Min-heap (default)
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();

    // Max-heap (using custom comparator)
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

    // Insert elements
    minHeap.offer(5);
    minHeap.offer(1);
    minHeap.offer(3);

    maxHeap.offer(5);
    maxHeap.offer(1);
    maxHeap.offer(3);

    System.out.println("Min-heap order:");
    while (!minHeap.isEmpty()) {
      System.out.println(minHeap.poll()); // 1, 3, 5
    }

    System.out.println("Max-heap order:");
    while (!maxHeap.isEmpty()) {
      System.out.println(maxHeap.poll()); // 5, 3, 1
    }
  }
}
```

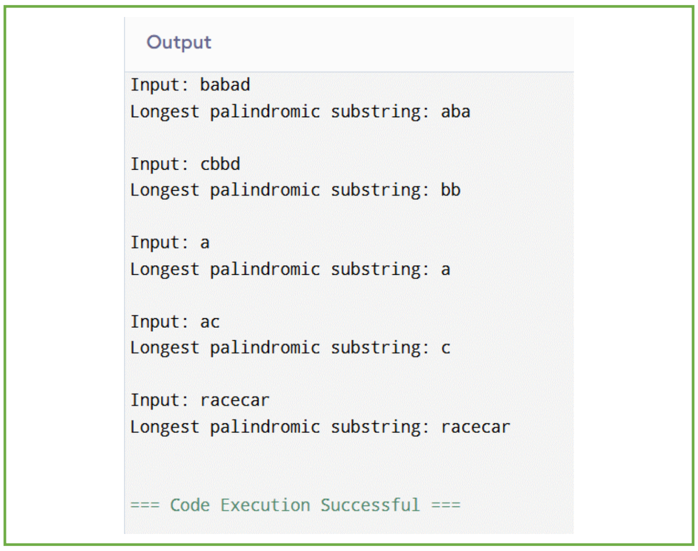

Output

```
Min-heap order:
1
3
5
Max-heap order:
5
3
1

=== Code Execution Successful ===
```

41. Write a program to find the longest palindromic substring in a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans - public class LongestPalindromicSubstring {

```
  public static String longestPalindrome(String s) {
    if (s == null || s.length() < 1) return "";

    int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
      // Check for odd-length palindromes
      int len1 = expandAroundCenter(s, i, i);
      // Check for even-length palindromes
      int len2 = expandAroundCenter(s, i, i + 1);
      // Get the maximum length
      int len = Math.max(len1, len2);
            // Update start and end if longer palindrome found
      if (len > end - start) {
        start = i - (len - 1) / 2;
        end = i + len / 2;
      }
    }
        return s.substring(start, end + 1);
  }
   private static int expandAroundCenter(String s, int left, int right) {
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
      left--;
      right++;
    }
    return right - left - 1;
  }

  public static void main(String[] args) {
    String[] testCases = {"babad", "cbbd", "a", "ac", "racecar"};
        for (String test : testCases) {
      System.out.println("Input: " + test);
      System.out.println("Longest palindromic substring: " + longestPalindrome(test));
      System.out.println();
    }
  }
}
```

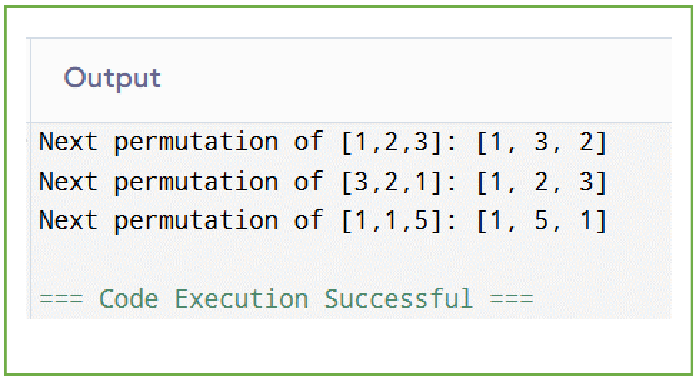


```
Input: babad
Longest palindromic substring: aba

Input: cbbd
Longest palindromic substring: bb

Input: a
Longest palindromic substring: a

Input: ac
Longest palindromic substring: c

Input: racecar
Longest palindromic substring: racecar


=== Code Execution Successful ===
```

42. Solve the problem of finding the next permutation of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
Ans - import java.util.Arrays;

```
public class NextPermutation {

  public static void nextPermutation(int[] nums) {
    // Step 1: Find the pivot
    int pivot = findPivot(nums);

    if (pivot != -1) {
      // Step 2: Find the swap candidate
      int swapIndex = findSwapCandidate(nums, pivot);
      // Step 3: Swap
      swap(nums, pivot, swapIndex);
    }
    // Step 4: Reverse the suffix
    reverse(nums, pivot + 1);
  }

  private static int findPivot(int[] nums) {
    for (int i = nums.length - 2; i >= 0; i--) {
      if (nums[i] < nums[i + 1]) {
        return i;
      }
    }
    return -1; // No pivot found (array is in descending order)
  }

  private static int findSwapCandidate(int[] nums, int pivot) {
    for (int i = nums.length - 1; i > pivot; i--) {
      if (nums[i] > nums[pivot]) {
        return i;
      }
    }
    return -1;
  }

  private static void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
  }

  private static void reverse(int[] nums, int start) {
    int end = nums.length - 1;
    while (start < end) {
      swap(nums, start, end);
```

```
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] test1 = {1, 2, 3};
        nextPermutation(test1);
        System.out.println("Next permutation of [1,2,3]: " + Arrays.toString(test1));

        int[] test2 = {3, 2, 1};
        nextPermutation(test2);
        System.out.println("Next permutation of [3,2,1]: " + Arrays.toString(test2));

        int[] test3 = {1, 1, 5};
        nextPermutation(test3);
        System.out.println("Next permutation of [1,1,5]: " + Arrays.toString(test3));
    }
}
```


Output

```
Next permutation of [1,2,3]: [1, 3, 2]
Next permutation of [3,2,1]: [1, 2, 3]
Next permutation of [1,1,5]: [1, 5, 1]

=== Code Execution Successful ===
```