Assignment MTE SEC-9

Name – Dharmraj Kumar

Addmission No – 23SCSE1011587

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

Ans:- A **prefix sum array** is an array where each element at index i stores the sum of all elements from index 0 to i of the original array.

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

Applications:

- Fast range sum queries (e.g., sum from index l to r)
- Solving subarray sum problems
- Used in competitive programming for optimization
- Cumulative frequency analysis in data processing

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:-

Algorithm:

1. Input the array and range [L, R].
2. Create a prefix sum array.
3. Use the formula:
    - If L > 0 → sum = prefix[R] - prefix[L - 1]

- If L == 0 → sum = prefix[R]

4. Output the sum.

Java Program:

```
public class PrefixSumRange {
   public static void main(String[] args) {
      int[] arr = {2, 4, 6, 8, 10};
      int L = 1, R = 3;

      // Step 1: Create prefix sum array
      int[] prefix = new int[arr.length];
      prefix[0] = arr[0];
      for (int i = 1; i < arr.length; i++) {
         prefix[i] = prefix[i - 1] + arr[i];
      }

      // Step 2: Calculate sum in range [L, R]
      int sum = (L > 0) ? prefix[R] - prefix[L - 1] : prefix[R];

      System.out.println("Sum from index " + L + " to " + R + " is: " + sum);
   }
}
```

Output: Sum from index 1 to 3 is: 18.

Time Complexity:

- Prefix Array Creation: O(n)
- Query: O(1)

Space Complexity:

- O(n) for the prefix sum array

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.**

Algorithm:

1. Calculate the total sum of the array.
2. Initialize leftSum = 0.
3. For each index i:
    - rightSum = totalSum - leftSum - arr[i]
    - If leftSum == rightSum, return i (equilibrium index)
    - Update leftSum += arr[i]
4. If no index found, return -1.

Java Program:

```
public class EquilibriumIndex {
    public static void main(String[] args) {
        int[] arr = {3, 1, 5, 2, 2};
        int totalSum = 0, leftSum = 0;

        for (int num : arr)
```

```
            totalSum += num;

        for (int i = 0; i < arr.length; i++) {
            int rightSum = totalSum - leftSum - arr[i];
            if (leftSum == rightSum) {
                System.out.println("Equilibrium index is: " + i);
                return;
            }
            leftSum += arr[i];
        }

        System.out.println("No equilibrium index found.");
    }
}
```

Output:

Equilibrium index is: 2

Time Complexity: O(n)

Space Complexity: O(1)

5. **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:-

Algorithm:

1. Calculate the total sum of the array.
2. Initialize leftSum = 0.
3. Traverse the array:
    - Add arr[i] to leftSum
    - Calculate rightSum = totalSum - leftSum
    - If leftSum == rightSum, return true
4. If loop ends, return false

Java Program:

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

        int totalSum = 0, leftSum = 0;

        for (int num : arr)
            totalSum += num;

        for (int i = 0; i < arr.length - 1; i++) {
            leftSum += arr[i];
            int rightSum = totalSum - leftSum;
```

```
            if (leftSum == rightSum) {
                System.out.println("Can be split at index " + i);
                return;
            }
        }

        System.out.println("Cannot be split into equal prefix and
suffix.");
    }
}
```

Output: Can be split at index 1

Time Complexity: O(n)

Space Complexity: O(1)

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:-

Algorithm (Sliding Window):

1. Compute the sum of the first K elements → windowSum.
2. Initialize maxSum = windowSum.
3. Slide the window from index K to n-1:
    - windowSum = windowSum - arr[i - K] + arr[i]

- Update maxSum if windowSum is greater

4. Return maxSum

Java Program:

```
public class MaxSubarraySumK {
    public static void main(String[] args) {
        int[] arr = {1, 4, 2, 10, 2, 3, 1, 0, 20};
        int k = 4;

        int maxSum = 0, windowSum = 0;

        // Step 1: sum of first k elements
        for (int i = 0; i < k; i++)
            windowSum += arr[i];

        maxSum = windowSum;

        // Step 2: slide the window
        for (int i = k; i < arr.length; i++) {
            windowSum += arr[i] - arr[i - k];
            if (windowSum > maxSum)
                maxSum = windowSum;
        }
```

```
        System.out.println("Maximum sum of subarray of size " + k + " is: " + maxSum);
    }
}
```

Output:

Maximum sum of subarray of size 4 is: 24

Time Complexity: O(n)

Space Complexity: O(1)

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:-

Algorithm (Sliding Window + HashSet):

1. Use a sliding window with two pointers start and end.
2. Use a HashSet to track characters in the current window.
3. Expand end, and if a character is repeated, shrink from start until the window is valid.
4. Track the maximum window size.

Java Program:

```
public class LongestUniqueSubstring {
```

```
    public static int lengthOfLongestUniqueSubstring(String s) {
        HashSet<Character> set = new HashSet<>();
        int maxLen = 0, start = 0;

        for (int end = 0; end < s.length(); end++) {
            while (set.contains(s.charAt(end))) {
                set.remove(s.charAt(start));
                start++;
            }
            set.add(s.charAt(end));
            maxLen = Math.max(maxLen, end - start + 1);
        }

        return maxLen;
    }

    public static void main(String[] args) {
        String input = "abcabcbb";
        int result = lengthOfLongestUniqueSubstring(input);
        System.out.println("Length of longest substring without
repeating characters: " + result);
    }
}
```

Output:

Length of longest substring without repeating characters: 3

Time Complexity: O(n)

Space Complexity: O(k)

Where k is the number of unique characters (at most 26 for lowercase letters).

7. Sliding Window Technique and Its Use in String Problems

✅ **Definition:**

The Sliding Window Technique is a method for handling problems that involve contiguous sequences like substrings or subarrays by using a window defined by two pointers (start and end).

How It Works:

1. Initialize two pointers to define a window.
2. Expand the window by moving end.
3. Shrink the window by moving start if a condition is violated.
4. Update result during traversal.

Use in String Problems:

- Longest substring without repeating characters
- Find all anagrams in a string

- Minimum window substring
- Substring with at most K distinct characters

These problems require checking character patterns or frequency within a moving window.

Advantages:

- Optimizes time from $O(n^2)$ to $O(n)$
- Avoids repeated calculations

Example:

For "abcabcbb" → longest unique substring is "abc" with length 3.

Time Complexity: O(n)

Space Complexity: O(k) (unique characters**)**

8. Find the Longest Palindromic Substring in a Given String

Algorithm (Expand Around Center):

1. For each character in the string, expand around that character as a center.
2. Consider both odd and even length centers.
3. Track the longest palindrome found during expansion.
4. Return the longest palindromic substring.

Java Program:

```
public class LongestPalindromeSubstring {

    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++) {
            int len1 = expandFromCenter(s, i, i);     // odd length
            int len2 = expandFromCenter(s, i, i + 1); // even length
            int len = Math.max(len1, len2);

            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }

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

    public static int expandFromCenter(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 input = "babad";
        String result = longestPalindrome(input);
        System.out.println("Longest palindromic substring: " + result);
    }
}
```

Output:

Longest palindromic substring: bab

(Note: For input "babad", both "bab" and "aba" are valid answers.)

Time Complexity: O(n^2)

Space Complexity: O(1)

9. Find the Longest Common Prefix Among a List of Strings

Algorithm (Vertical Scanning):

1. Take the first string as the reference.
2. Compare each character of the first string with the same index in all other strings.
3. If any mismatch occurs or a string ends, return the prefix found so far.

Java Program:

```
public class LongestCommonPrefix {


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


      for (int i = 0; i < strs[0].length(); i++) {
         char c = strs[0].charAt(i);


         for (int j = 1; j < strs.length; j++) {
            if (i >= strs[j].length() || strs[j].charAt(i) != c) {
               return strs[0].substring(0, i);
            }
         }
      }
      return strs[0];
   }
```

```
    public static void main(String[] args) {
        String[] input = {"flower", "flow", "flight"};
        String result = longestCommonPrefix(input);
        System.out.println("Longest Common Prefix: " + result);
    }
}
```

Output: Longest Common Prefix: fl

Time Complexity: O(n * m)

- n = number of strings
- m = length of the shortest string

Space Complexity: O(1)

10. Generate All Permutations of a Given String

Algorithm (Backtracking):

1. Convert the string to a character array.
2. Use recursion and swapping to fix each character at a position.
3. Swap characters back after recursive calls (backtrack).
4. Print each permutation when the end is reached.

Java Program:

```
public class StringPermutations {

    public static void generatePermutations(char[] chars, int index) {
        if (index == chars.length - 1) {
            System.out.println(String.valueOf(chars));
            return;
        }

        for (int i = index; i < chars.length; i++) {
            swap(chars, index, i);
            generatePermutations(chars, index + 1);
            swap(chars, index, i); // backtrack
        }
    }

    public static void swap(char[] chars, int i, int j) {
        char temp = chars[i];
        chars[i] = chars[j];
        chars[j] = temp;
    }

    public static void main(String[] args) {
        String input = "abc";
        generatePermutations(input.toCharArray(), 0);
```

```
    }
}
```

Output:

```
nginx
CopyEdit
abc
acb
bac
bca
cba
cab
```

Time Complexity: O(n!)

(Each of the n! permutations takes O(n) time to print)

Space Complexity: O(n) (for recursion stack)

11. Find Two Numbers in a Sorted Array That Add Up to a Target

---

Algorithm:

1. Initialize two pointers: left at the beginning and right at the end of the array.
2. Check the sum of the elements at left and right:

- If the sum equals the target, return the pair.
- If the sum is less than the target, move the left pointer one step to the right (increase the sum).
- If the sum is greater than the target, move the right pointer one step to the left (decrease the sum).

3. Repeat until the two pointers meet or find a pair.

Java Program:

```
public class TwoSumSortedArray {


    public static int[] twoSum(int[] nums, int target) {
        int left = 0, right = nums.length - 1;


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


            if (sum == target) {
                return new int[]{nums[left], nums[right]};
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
```

```
        return new int[]{-1, -1}; // No pair found
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3, 4, 5, 6};
        int target = 10;
        int[] result = twoSum(nums, target);
        System.out.println("Pair: " + result[0] + ", " + result[1]);
    }
}
```

✅ **Output:** Pair: 4, 6

Time Complexity: O(n)

Space Complexity: O(1)

Example:

Input: [1, 2, 3, 4, 5, 6], Target: 10
→ Pair: 4 and 6

---

12. Rearrange Numbers into the Lexicographically Next Greater Permutation

---

Algorithm:

1. Find the largest index i such that nums[i] < nums[i + 1].
2. Find the largest index j such that nums[j] > nums[i].
3. Swap nums[i] and nums[j].
4. Reverse the subarray from i + 1 to the end.

Java Program:

```
public class NextPermutation {

    public static void nextPermutation(int[] nums) {
        int i = nums.length - 2;

        // Step 1: Find the first decreasing element
        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }

        if (i >= 0) {
            int j = nums.length - 1;
            // Step 2: Find the number to swap with
            while (nums[j] <= nums[i]) {
                j--;
            }

            // Step 3: Swap the numbers
```

```
            swap(nums, i, j);
        }

        // Step 4: Reverse the subarray
        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--);
        }
    }

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

```
        System.out.println("Next permutation: " +
Arrays.toString(nums));
    }
}
```

Output: Next permutation: [1, 3, 2]

Time Complexity: O(n)

Space Complexity: O(1)

Example:

Input: [1, 2, 3]
→ Next permutation: [1, 3, 2]

---

13. Merge Two Sorted Linked Lists into One Sorted List

Algorithm:

1. Initialize a dummy node to track the merged list.
2. Compare the elements from both lists, appending the smaller element to the merged list.
3. Once one list is exhausted, append the remaining elements from the other list.

Java Program:

```
public class MergeTwoSortedLists {
```

```
static class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    while (l1 != null && l2 != null) {
        if (l1.val <= l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    if (l1 != null) current.next = l1;
    else current.next = l2;
```

```
        return dummy.next;
    }

    public static void main(String[] args) {
        ListNode l1 = new ListNode(1);
        l1.next = new ListNode(2);
        l1.next.next = new ListNode(4);

        ListNode l2 = new ListNode(1);
        l2.next = new ListNode(3);
        l2.next.next = new ListNode(4);

        ListNode result = mergeTwoLists(l1, l2);
        while (result != null) {
            System.out.print(result.val + " ");
            result = result.next;
        }
    }
}
```

✅ **Output:** 1 1 2 3 4 4

Time Complexity: O(m + n)

- Where m and n are the lengths of the two lists.

Space Complexity: O(1) (excluding the space for the result list)

Example:

Input: l1 = [1, 2, 4], l2 = [1, 3, 4]
→ Merged list: [1, 1, 2, 3, 4, 4]

---

14. Find the Median of Two Sorted Arrays Using Binary Search

Algorithm:

1. Ensure the first array is the smaller one.
2. Perform binary search on the smaller array.
3. Partition both arrays such that left half and right half combined have equal elements.
4. Compute the median based on the partitioned elements.

Java Program:

```
public class MedianOfTwoSortedArrays {


   public static double findMedianSortedArrays(int[] nums1, int[] nums2) {
       if (nums1.length > nums2.length) {
           return findMedianSortedArrays(nums2, nums1);  // Make nums1 the smaller array
       }
```

```
        int m = nums1.length, n = nums2.length;
        int left = 0, right = m;

        while (left <= right) {
            int partition1 = (left + right) / 2;
            int partition2 = (m + n + 1) / 2 - partition1;

            int maxLeft1 = (partition1 == 0) ? Integer.MIN_VALUE :
nums1[partition1 - 1];
            int minRight1 = (partition1 == m) ? Integer.MAX_VALUE :
nums1[partition1];

            int maxLeft2 = (partition2 == 0) ? Integer.MIN_VALUE :
nums2[partition2 - 1];
            int minRight2 = (partition2 == n) ? Integer.MAX_VALUE :
nums2[partition2];

            if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
                if ((m + n) % 2 == 0) {
                    return (Math.max(maxLeft1, maxLeft2) +
Math.min(minRight1, minRight2)) / 2.0;
                } else {
                    return Math.max(maxLeft1, maxLeft2);
                }
            } else if (maxLeft1 > minRight2) {
```

```
            right = partition1 - 1;
        } else {
            left = partition1 + 1;
        }
    }

    return 0.0; // Should never reach here
  }

  public static void main(String[] args) {
    int[] nums1 = {1, 3};
    int[] nums2 = {2};
    System.out.println("Median: " + findMedianSortedArrays(nums1, nums2));
  }
}
```

✅ **Output:** Median: 2.0

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

- m and n are the lengths of the two arrays.

Space Complexity: O(1)

Example:

Input: nums1 = [1, 3], nums2 = [2]
→ Median: 2.0

---

15. Find the k-th Smallest Element in a Sorted Matrix

Algorithm:

1. Use a min-heap to store elements from the matrix.
2. Extract the smallest element and continue until the k-th smallest element is found.

Java Program:

```
public class KthSmallestInMatrix {
    public static int kthSmallest(int[][] matrix, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                minHeap.offer(matrix[i][j]);
            }
        }

        int count = 0;
        while (count < k - 1) {
            minHeap.poll();
```

```
            count++;
        }

        return minHeap.poll();
    }

    public static void main(String[] args) {
        int[][] matrix = {
            {1, 5, 9},
            {10, 11, 13},
            {12, 13, 15}
        };
        int k = 8;
        System.out.println("k-th smallest: " + kthSmallest(matrix, k));
    }
}
```

Output: k-th smallest: 13

Time Complexity: O(n * m * log(n * m))

- Where n and m are the dimensions of the matrix.

Space Complexity: O(n * m)

Example:

Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
→ k-th smallest: 13

---

16. Find the Majority Element in an Array That Appears More Than n/2 Times

---

Algorithm:

1. Use Boyer-Moore Voting Algorithm to find the candidate.
2. Verify if the candidate occurs more than n/2 times.

Java Program:

```
public class MajorityElement {

    public static int majorityElement(int[] nums) {
        int count = 0, candidate = -1;

        for (int num : nums) {
            if (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }

        return candidate;
```

```
    }

    public static void main(String[] args) {
        int[] nums = {3, 2, 3};
        System.out.println("Majority element: " +
majorityElement(nums));
    }
}
```

Output: Majority element: 3

Time Complexity: O(n)

Space Complexity: O(1)

17. Calculate How Much Water Can Be Trapped Between the Bars of a Histogram

---

Algorithm:

1. Initialize two arrays: leftMax[] and rightMax[] to store the maximum height seen from the left and right for each bar.
2. Populate the leftMax[] array by iterating from left to right, keeping track of the highest bar encountered.
3. Populate the rightMax[] array by iterating from right to left.
4. For each bar, calculate the water trapped above it as min(leftMax[i], rightMax[i]) - height[i].

5. Sum the water trapped at each index.

Java Program:

```
public class WaterTrapping {

    public static int trap(int[] height) {
        if (height == null || height.length == 0) return 0;

        int n = height.length;
        int[] leftMax = new int[n];
        int[] rightMax = new int[n];
        int waterTrapped = 0;

        // Fill leftMax array
        leftMax[0] = height[0];
        for (int i = 1; i < n; i++) {
            leftMax[i] = Math.max(leftMax[i - 1], height[i]);
        }

        // Fill rightMax array
        rightMax[n - 1] = height[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            rightMax[i] = Math.max(rightMax[i + 1], height[i]);
        }
```

```
        // Calculate water trapped
        for (int i = 0; i < n; i++) {
            waterTrapped += Math.min(leftMax[i], rightMax[i]) - height[i];
        }

        return waterTrapped;
    }

    public static void main(String[] args) {
        int[] height = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
        System.out.println("Water trapped: " + trap(height));
    }
}
```

✅ **Output:** Water trapped: 6

Time Complexity: O(n)

Space Complexity: O(n)

Example:

Input: [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
→ Water trapped: 6

---

18. Find the Maximum XOR of Two Numbers in an Array

Algorithm:

1. Initialize a variable maxXOR to store the maximum XOR found.
2. Use a Trie or HashSet to store the prefixes of the numbers in the array.
3. For each number, calculate the maximum possible XOR by comparing it with each prefix stored in the Trie.
4. Update maxXOR if a larger XOR is found.

Java Program:

```
public class MaximumXOR {


   public static int findMaximumXOR(int[] nums) {
      int maxXOR = 0;
      int mask = 0;


      for (int i = 31; i >= 0; i--) {
         mask |= (1 << i);
         HashSet<Integer> prefixes = new HashSet<>();


         for (int num : nums) {
            prefixes.add(num & mask);
         }
```

```
            int candidate = maxXOR | (1 << i);
            for (int prefix : prefixes) {
                if (prefixes.contains(prefix ^ candidate)) {
                    maxXOR = candidate;
                    break;
                }
            }
        }

        return maxXOR;
    }

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

Output:

Maximum XOR: 28

Time Complexity: O(n * 32)

- Where n is the number of elements and 32 is the number of bits in an integer.

Space Complexity: O(n)

Example:

Input: [3, 10, 5, 25, 2, 8]
→ Maximum XOR: 28

---

19. How to Find the Maximum Product Subarray

Algorithm:

1. Initialize variables for maxProduct, minProduct, and result.
2. Traverse the array. For each element:
    - If it's negative, swap maxProduct and minProduct because multiplying a negative number can turn a small product into a large one.
    - Update maxProduct and minProduct based on the current element.
    - Update the result if maxProduct is larger than the current result.

Java Program:

```
public class MaximumProductSubarray {


   public static int maxProduct(int[] nums) {
```

```
        int maxProduct = nums[0], minProduct = nums[0], result =
nums[0];

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < 0) {
                int temp = maxProduct;
                maxProduct = minProduct;
                minProduct = temp;
            }

            maxProduct = Math.max(nums[i], maxProduct * nums[i]);
            minProduct = Math.min(nums[i], minProduct * nums[i]);

            result = Math.max(result, maxProduct);
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {2, 3, -2, 4};
        System.out.println("Maximum product: " + maxProduct(nums));
    }
}
```

Output: Maximum product: 6

Time Complexity: O(n)

Space Complexity: O(1)

21. How to Count the Number of 1s in the Binary Representation of Numbers from 0 to n

---

Algorithm:

1. Iterate through each number from 0 to n.
2. Use bitwise operations to count the 1s in the binary representation of each number.

Java Program:

```
public class CountOnes {


   public static int countSetBits(int n) {
      int count = 0;
      for (int i = 0; i <= n; i++) {
         count += Integer.bitCount(i);
      }
      return count;
   }
```

```
    public static void main(String[] args) {
        int n = 5;
        System.out.println("Total 1's in binary representations: " +
countSetBits(n));
    }
}
```

✅ **Output:** Total 1's in binary representations: 7

Time Complexity: O(n * log n)

Space Complexity: O(1)

Example:

Input: n = 5
→ Total 1's: 7 (Binary representations: 0, 1, 10, 11, 100, 101)

---

22. How to Check if a Number is a Power of Two Using Bit Manipulation

---

Algorithm:

1. A number is a power of two if it has only one 1 bit in its binary representation.
2. Use the expression n & (n - 1) which clears the lowest set bit. If the result is 0, n is a power of two.

Java Program:

```
public class PowerOfTwo {

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

    public static void main(String[] args) {
        int n = 16;
        System.out.println("Is power of two: " + isPowerOfTwo(n));
    }
}
```

Output:

Is power of two: true

Time Complexity: O(1)

Space Complexity: O(1)

Example:

Input: n = 16
→ 16 is a power of two: true

---

23. How to Find the Maximum XOR of Two Numbers in an Array

Algorithm:

1. Build a Trie to store prefixes of all numbers.
2. For each number, check the Trie to find the best possible XOR with the prefixes.

Java Program:

```
public class MaximumXOR {


   public static int findMaximumXOR(int[] nums) {
      int maxXOR = 0;
      int mask = 0;


      for (int i = 31; i >= 0; i--) {
         mask |= (1 << i);
         HashSet<Integer> prefixes = new HashSet<>();


         for (int num : nums) {
            prefixes.add(num & mask);
         }


         int candidate = maxXOR | (1 << i);
         for (int prefix : prefixes) {
```

```
            if (prefixes.contains(prefix ^ candidate)) {
                maxXOR = candidate;
                break;
            }
        }
    }

    return maxXOR;
  }

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

Output: Maximum XOR: 28

Time Complexity: O(n * 32)

Space Complexity: O(n)

Example:

Input: [3, 10, 5, 25, 2, 8]
→ Maximum XOR: 28

---

24. Explain the Concept of Bit Manipulation and Its Advantages in Algorithm Design

---

Explanation:

Bit Manipulation involves directly manipulating individual bits of data (0s and 1s). Common operations include AND, OR, XOR, NOT, shifting, and toggling bits. These operations are extremely fast and can help optimize algorithms.

Advantages:

1. Efficiency: Operations on bits are faster than arithmetic or comparison operations.
2. Memory Optimization: Storing data in bitwise form can use less memory (e.g., using a bit array instead of a boolean array).
3. Compactness: Some problems, such as subset generation or checking powers of two, can be solved in constant time using bit operations.
4. Parallelism: Bit-level operations can be done in parallel in hardware, speeding up computations.

---

25. Solve the Problem of Finding the Next Greater Element for Each Element in an Array

---

Algorithm:

1. Use a stack to keep track of elements for which the next greater element is yet to be found.
2. Traverse the array from right to left.
3. For each element, pop elements from the stack that are smaller or equal.
4. If the stack is not empty, the top of the stack will be the next greater element for the current element.
5. Push the current element onto the stack**.**

Java Program:

```
public class NextGreaterElement {


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


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


      return result;
```

```
    }

    public static void main(String[] args) {
        int[] nums = {4, 5, 2, 10, 8};
        int[] result = nextGreater(nums);
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

Output: 10 10 10 -1 -1

Time Complexity: O(n)

Space Complexity: O(n)

Example:

Input: [4, 5, 2, 10, 8]
→ Next Greater Element: [10, 10, 10, -1, -1]

---

26. Remove the n-th Node from the End of a Singly Linked List

---

Algorithm:

1. Use two pointers: first and second.
2. Move the first pointer n steps ahead.
3. Move both first and second pointers one step at a time until first reaches the end.
4. second will be at the node before the one to remove.

Java Program:

```
public class RemoveNthNode {


   public static ListNode removeNthFromEnd(ListNode head, int n) {
      ListNode dummy = new ListNode(0);
      dummy.next = head;
      ListNode first = dummy;
      ListNode second = dummy;


      for (int i = 1; i <= n + 1; i++) {
         first = first.next;
      }


      while (first != null) {
         first = first.next;
         second = second.next;
      }
```

```
        second.next = second.next.next;
        return dummy.next;
    }

    public static void main(String[] args) {
        // Example usage
    }
}
```

Time Complexity: O(n)

Space Complexity: O(1)

27. Find the Node Where Two Singly Linked Lists Intersect

Algorithm:

1. Find the lengths of both linked lists.
2. Move the pointer of the longer list ahead by the difference in lengths.
3. Move both pointers step by step until they meet or both reach the end.

Java Program:

```
public class IntersectionOfLinkedLists {
```

```
    public static ListNode getIntersectionNode(ListNode headA,
ListNode headB) {
        int lengthA = getLength(headA);
        int lengthB = getLength(headB);

        while (lengthA > lengthB) {
            headA = headA.next;
            lengthA--;
        }

        while (lengthB > lengthA) {
            headB = headB.next;
            lengthB--;
        }

        while (headA != null && headB != null) {
            if (headA == headB) {
                return headA;
            }
            headA = headA.next;
            headB = headB.next;
        }
```

```
        return null;
    }

    private static int getLength(ListNode head) {
        int length = 0;
        while (head != null) {
            length++;
            head = head.next;
        }
        return length;
    }
}
```

Time Complexity: O(m + n)

Space Complexity: O(1)

✅ **Example:**

Input: Two linked lists that intersect at a node.
→ Returns the node where the two lists intersect.

---

28. Implement Two Stacks in a Single Array

Algorithm:

1. Use a single array to represent both stacks.

2. One stack grows from the left and the other from the right.
3. Ensure the two stacks do not overlap by tracking their indices.

Java Program:

```
public class TwoStacksInArray {

    private int[] arr;
    private int top1, top2;

    public TwoStacksInArray(int size) {
        arr = new int[size];
        top1 = -1;
        top2 = size;
    }

    public void push1(int value) {
        if (top1 + 1 == top2) {
            System.out.println("Stack 1 Overflow");
            return;
        }
        arr[++top1] = value;
    }

    public void push2(int value) {
```

```
        if (top2 - 1 == top1) {
            System.out.println("Stack 2 Overflow");
            return;
        }
        arr[--top2] = value;
    }

    public int pop1() {
        if (top1 == -1) {
            System.out.println("Stack 1 Underflow");
            return -1;
        }
        return arr[top1--];
    }

    public int pop2() {
        if (top2 == arr.length) {
            System.out.println("Stack 2 Underflow");
            return -1;
        }
        return arr[top2++];
    }
}
```

Time Complexity: O(1) for both push and pop operations

Space Complexity: O(n)

Example:

Input:
Two stacks implemented in a single array, operations push1(1), push2(2), pop1(), and pop2().
→ Outputs the results of popping elements from both stacks.

---

29. Write a Program to Check if an Integer is a Palindrome Without Converting It to a String

---

Algorithm:

1. Reverse the second half of the number and compare it with the first half.
2. If both halves are equal, it is a palindrome.

Java Program:

```
public class PalindromeNumber {


   public static boolean isPalindrome(int x) {
      if (x < 0 || (x % 10 == 0 && x != 0)) return false;


      int reversed = 0;
      while (x > reversed) {
```

```
            reversed = reversed * 10 + x % 10;
            x /= 10;
        }
        return x == reversed || x == reversed / 10;
    }

    public static void main(String[] args) {
        int n = 121;
        System.out.println("Is palindrome: " + isPalindrome(n));
    }
}
```

Output: Is palindrome: true

Time Complexity: O(log n)

Space Complexity: O(1)

---

30. Explain the Concept of Linked Lists and Their Applications in Algorithm Design

---

Explanation:

Linked Lists are linear data structures where each element (node) contains data and a reference (pointer) to the next node in the sequence. Linked lists are flexible in terms of memory allocation since they do not require contiguous memory locations.

Applications in Algorithm Design:

1. Dynamic Memory Allocation: Linked lists can dynamically allocate memory, making them useful when the number of elements is not known in advance.
2. Efficient Insertion/Deletion: Inserting and deleting nodes are efficient (O(1)) if the node to be modified is known.
3. Implementing Other Data Structures: Linked lists are used to implement other data structures such as stacks, queues, and graphs.

---

31. Use a Deque to Find the Maximum in Every Sliding Window of Size K

Algorithm:

1. Use a deque to store indices of useful elements in the sliding window.
2. The deque will store elements in decreasing order to maintain the maximum at the front.
3. Slide the window by adding the next element and removing elements that fall outside the window.

Java Program:

```
public class MaxInSlidingWindow {


   public static int[] maxSlidingWindow(int[] nums, int k) {
      if (nums == null || nums.length == 0) return new int[0];
```

```
        int n = nums.length;
        int[] result = new int[n - k + 1];
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < n; i++) {
            while (!deque.isEmpty() && nums[deque.peekLast()] <=
nums[i]) {
                deque.pollLast();
            }
            deque.offer(i);

            if (deque.peek() <= i - k) {
                deque.poll();
            }

            if (i >= k - 1) {
                result[i - k + 1] = nums[deque.peek()];
            }
        }

        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.println(Arrays.toString(result));
    }
}
```

Output: [3, 3, 5, 5, 6, 7]

Time Complexity: O(n)

Space Complexity: O(k)

---

32. How to Find the Largest Rectangle that Can Be Formed in a Histogram

---

Algorithm:

1. Use a stack to store the indices of the histogram bars.
2. For each bar, if it is smaller than the bar at the top of the stack, calculate the area with the top bar as the smallest bar.
3. Keep track of the maximum area.

Java Program:

```
public class LargestRectangleInHistogram {
```

```
    public static int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0;
        int index = 0;

        while (index < heights.length) {
            if (stack.isEmpty() || heights[index] >= heights[stack.peek()]) {
                stack.push(index++);
            } else {
                int top = stack.pop();
                maxArea = Math.max(maxArea, heights[top] *
(stack.isEmpty() ? index : index - stack.peek() - 1));
            }
        }

        while (!stack.isEmpty()) {
            int top = stack.pop();
            maxArea = Math.max(maxArea, heights[top] * (stack.isEmpty()
? index : index - stack.peek() - 1));
        }

        return maxArea;
    }
```

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

Output:

Largest Rectangle Area: 10

Time Complexity: O(n)

Space Complexity: O(n)

---

33. Explain the Sliding Window Technique and Its Applications in Array Problems

---

Explanation:

Sliding Window Technique is used to maintain a subset of data (a window) in an array, usually of fixed size. It slides over the array to compute some value or condition efficiently.

Applications:

1. Subarray problems: Finding subarrays with certain properties (sum, max/min, etc.).

2. String problems: Finding substrings with unique characters, palindromes, etc.
3. Efficiency: Reduces time complexity from O(n^2) to O(n) by maintaining the window.

---

34. Solve the Problem of Finding the Subarray Sum Equal to K Using Hashing

---

Algorithm:

1. Maintain a running sum while iterating through the array.
2. Use a hash map to store the frequency of running sums.
3. If the running sum minus k exists in the hash map, it means there is a subarray whose sum is k.

Java Program:

```
public class SubarraySumEqualToK {


   public static int subarraySum(int[] nums, int k) {
      HashMap<Integer, Integer> map = new HashMap<>();
      map.put(0, 1);
      int count = 0, sum = 0;


      for (int num : nums) {
         sum += num;
         if (map.containsKey(sum - k)) {
```

```
            count += map.get(sum - k);
        }
        map.put(sum, map.getOrDefault(sum, 0) + 1);
    }

    return count;
  }

  public static void main(String[] args) {
    int[] nums = {1, 1, 1};
    int k = 2;
    System.out.println("Subarray Sum Equal to K: " +
subarraySum(nums, k));
  }
}
```

Output: Subarray Sum Equal to K: 2

Time Complexity: O(n)

Space Complexity: O(n)

35. Find the k-most frequent elements in an array using a priority queue

---

Algorithm:

1. Count the frequency of each element using a hash map.
2. Use a priority queue (max heap) to store elements by their frequency.
3. Extract the top k elements from the heap.

Java Program:

```
public class KMostFrequentElements {


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


      PriorityQueue<Map.Entry<Integer, Integer>> maxHeap =
         new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());


      maxHeap.addAll(frequencyMap.entrySet());


      List<Integer> result = new ArrayList<>();
      while (k-- > 0) {
         result.add(maxHeap.poll().getKey());
```

```
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        System.out.println(topKFrequent(nums, k));
    }
}
```

Output: [1, 2]

Time Complexity: O(n log k)

Space Complexity: O(n)

---

36. Generate All Subsets of a Given Array

Algorithm:

1. Use backtracking to generate all possible subsets.
2. For each element, include or exclude it in the current subset.

Java Program:

```
public class GenerateSubsets {
```

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

    private static void backtrack(List<List<Integer>> result,
List<Integer> tempList, int[] nums, int start) {
        result.add(new ArrayList<>(tempList));
        for (int i = start; i < nums.length; i++) {
            tempList.add(nums[i]);
            backtrack(result, tempList, nums, i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        System.out.println(subsets(nums));
    }
}
```

✅ **Output:** [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

Time Complexity: O(2^n)

Space Complexity: O(n)

---

37. Find All Unique Combinations of Numbers That Sum to a Target

---

Algorithm:

1. Use backtracking to generate combinations that sum to the target.
2. If the sum exceeds the target, prune the branch.
3. Ensure no duplicate combinations by not revisiting previous elements.

Java Program:

```
public class CombinationSum {


    public static List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), candidates, target, 0);
        return result;
    }


    private static void backtrack(List<List<Integer>> result, List<Integer> tempList, int[] candidates, int remain, int start) {
```

```
        if (remain < 0) return;
        if (remain == 0) {
            result.add(new ArrayList<>(tempList));
            return;
        }
        for (int i = start; i < candidates.length; i++) {
            tempList.add(candidates[i]);
            backtrack(result, tempList, candidates, remain - candidates[i],
i);  // not i + 1 because we can reuse the same element
            tempList.remove(tempList.size() - 1);
        }
    }

    public static void main(String[] args) {
        int[] candidates = {2, 3, 6, 7};
        int target = 7;
        System.out.println(combinationSum(candidates, target));
    }
}
```

Output: [[2, 2, 3], [7]]

Complexity: O(2^n)

Space Complexity: O(target)

---

38. Generate All Permutations of a Given Array

Algorithm:

1. Use backtracking to generate all possible permutations.
2. Swap elements to generate permutations and backtrack after generating each one.

Java Program:

```
public class GeneratePermutations {


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


   private static void backtrack(List<List<Integer>> result,
List<Integer> tempList, int[] nums) {
      if (tempList.size() == nums.length) {
         result.add(new ArrayList<>(tempList));
      } else {
         for (int i = 0; i < nums.length; i++) {
            if (tempList.contains(nums[i])) continue;
```

```
                tempList.add(nums[i]);
                backtrack(result, tempList, nums);
                tempList.remove(tempList.size() - 1);
            }
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        System.out.println(permute(nums));
    }
}
```

Output:

[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

Time Complexity: O(n!)

Space Complexity: O(n)

---

39. Explain the Difference Between Subsets and Permutations with Examples

---

Explanation:

- Subsets: A subset is any combination of elements from the original set, including the empty set and the set itself. The order of elements in a subset does not matter.
    - Example: Subsets of [1, 2]: [[], [1], [2], [1, 2]]
- Permutations: A permutation is a rearrangement of elements where the order matters. Each element is used exactly once.
    - Example: Permutations of [1, 2]: [[1, 2], [2, 1]]

---

40. Solve the Problem of Finding the Element with Maximum Frequency in an Array

---

Algorithm:

1. Use a hash map to count the frequency of each element.
2. Find the element with the highest frequency.

Java Program:

```
public class MaxFrequencyElement {


   public static int maxFrequencyElement(int[] nums) {
      Map<Integer, Integer> frequencyMap = new HashMap<>();
      for (int num : nums) {
         frequencyMap.put(num, frequencyMap.getOrDefault(num, 0)
+ 1);
      }
```

```
        int maxFreq = 0;
        int maxElement = -1;
        for (Map.Entry<Integer, Integer> entry :
frequencyMap.entrySet()) {
            if (entry.getValue() > maxFreq) {
                maxFreq = entry.getValue();
                maxElement = entry.getKey();
            }
        }
        return maxElement;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 2, 2, 5, 1, 2};
        System.out.println(maxFrequencyElement(nums));
    }
}
```

✅ **Output:** 2

Time Complexity: O(n)

Space Complexity: O(n)

---

41. Write a Program to Find the Maximum Subarray Sum Using Kadane's Algorithm

---

Algorithm:

1. Iterate through the array while maintaining the current maximum sum and the global maximum.
2. Update the global maximum whenever the current sum exceeds it.

Java Program:

```
public class MaximumSubarraySum {

    public static int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }

        return maxSum;
    }

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

```
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println(maxSubArray(nums));
    }
}
```

✅ **Output:** 6

Time Complexity: O(n)

Space Complexity: O(1)

---

42. Explain the Concept of Dynamic Programming and Its Use in Solving the Maximum Subarray Problem

---

Explanation:

Dynamic programming (DP) is a technique for solving problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations. For the maximum subarray problem, the DP approach stores the current maximum subarray sum at each step, allowing us to solve the problem efficiently.

In Kadane's algorithm, the dynamic programming approach is used to keep track of the current sum of the subarray and update the global maximum sum whenever necessary.

---

43. Solve the Problem of Finding the Top K Frequent Elements in an Array

Algorithm:

1. Count the frequency of each element using a hash map.
2. Use a priority queue (max heap) to store elements by their frequency.
3. Extract the top k elements from the heap.

Java Program:

```
public class KMostFrequentElements {


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


      PriorityQueue<Map.Entry<Integer, Integer>> maxHeap =
         new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());


      maxHeap.addAll(frequencyMap.entrySet());


      List<Integer> result = new ArrayList<>();
      while (k-- > 0) {
         result.add(maxHeap.poll().getKey());
      }
```

```
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        System.out.println(topKFrequent(nums, k));
    }
}
```

✅ **Output:**

[1, 2]

Time Complexity: O(n log k)

Space Complexity: O(n)

---

44. How to Find Two Numbers in an Array That Add Up to a Target Using Hashing

---

Algorithm:

1. Create a hash set to store the numbers as you iterate through the array.
2. For each number, check if the difference between the target and the number exists in the hash set.

3. If it does, return the pair; otherwise, add the current number to the hash set.

Java Program:

```
public class TwoSumUsingHashing {

    public static boolean findTwoSum(int[] nums, int target) {
        Set<Integer> seen = new HashSet<>();
        for (int num : nums) {
            if (seen.contains(target - num)) {
                return true; // Found the pair
            }
            seen.add(num);
        }
        return false;
    }

    public static void main(String[] args) {
        int[] nums = {2, 7, 11, 15};
        int target = 9;
        System.out.println(findTwoSum(nums, target)); // Output: true
    }
}
```

Output: true

Time Complexity: O(n)

Space Complexity: O(n)

---

45. Explain the Concept of Priority Queues and Their Applications in Algorithm Design

---

Explanation:

A priority queue is a data structure where each element is associated with a priority. Elements are dequeued in order of priority, with the highest (or lowest) priority element being dequeued first.

- **Applications:**
    - Heap Sort: Used in sorting algorithms.
    - Dijkstra's Algorithm: For finding the shortest path in a graph.
    - Huffman Coding: For data compression.
    - Task Scheduling: To execute tasks based on priority.

Priority queues are commonly implemented using a heap (binary heap), which allows efficient insertion and deletion operations.

46. Write a Program to Find the Longest Palindromic Substring in a Given String

---

Algorithm:

1. Iterate through each character in the string as the center of the palindrome.

2. Expand around the center while the substring remains a palindrome.
3. Keep track of the longest palindrome found during the expansion.

Java Program:

```
public class LongestPalindromicSubstring {


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

        int start = 0, maxLength = 1;

        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);
            int len2 = expandAroundCenter(s, i, i + 1);
            int len = Math.max(len1, len2);

            if (len > maxLength) {
                maxLength = len;
                start = i - (maxLength - 1) / 2;
            }
```

```
        }

        return s.substring(start, start + maxLength);
    }

    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 s = "babad";
        System.out.println(longestPalindrome(s)); // Output: "bab"
    }
}
```

Output:

"bab" (or "aba" as both are valid palindromes)

Time Complexity: O(n^2)

Space Complexity: O(1)

---

47. Explain the Concept of Histogram Problems and Their Applications in Algorithm Design

---

Explanation:

A histogram problem is often used to represent bars with different heights, usually related to data analysis or graphical visualization. The problem focuses on determining properties or solving problems like:

- Largest Rectangle in Histogram: Find the largest rectangular area that can be formed using the bars.
- Trapping Rain Water: Calculate how much water can be trapped between the bars.

Applications:

- Bar Graph Representation: Histograms are useful in analyzing frequency distributions.
- Geospatial Analysis: Represent elevation maps or terrain.
- Histogram Equalization in Image Processing: Improve image contrast by adjusting pixel intensity distributions.

---

48. Solve the Problem of Finding the Next Permutation of a Given Array

---

Algorithm:

1. Identify the rightmost pair of consecutive elements where the first is smaller than the second (i.e., find the "pivot").

2. Swap the pivot with the smallest element that is larger than it to ensure the next higher permutation.
3. Reverse the elements to the right of the pivot index to get the next smallest arrangement.

Java Program:

```
public class NextPermutation {


    public static void nextPermutation(int[] nums) {
        int i = nums.length - 2;


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


        if (i >= 0) {
            int j = nums.length - 1;
            while (nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }


        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--);
        }
    }

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

Output:

[1, 3, 2].

Time Complexity: O(n)

Space Complexity: O(1)

---

49. How to Find the Intersection of Two Linked Lists

Algorithm:

1. Get the lengths of both linked lists.
2. Advance the pointer of the longer list by the difference in lengths.
3. Move both pointers simultaneously until they intersect.

Java Program:

```
public class LinkedListIntersection {
   public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
      if (headA == null || headB == null) {
         return null;
      }

      ListNode pointerA = headA;
      ListNode pointerB = headB;

      while (pointerA != pointerB) {
```

```
            pointerA = (pointerA == null) ? headB : pointerA.next;
            pointerB = (pointerB == null) ? headA : pointerB.next;
        }

        return pointerA; // If they meet, return the intersection node.
    }

    public static void main(String[] args) {
        // Example setup: Define two linked lists with an intersection.
    }
}

class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

Output:

```
List A: 1 -> 2 -> 3 -> 4
List B: 5 -> 6 -> 3 -> 4
→ Intersection node: 3
```

✅ Time Complexity: O(m + n)

✅ Space Complexity: O(1)

---

50. Explain the Concept of Equilibrium Index and Its Applications in Array Problems

---

Explanation:

An equilibrium index is an index where the sum of the elements on its left is equal to the sum of the elements on its right.

Applications:

- Balancing problems: Such as dividing an array into two parts with equal sums.
- Efficient problem-solving: Finding this index reduces the time complexity of solving related array partition problems.