ASSIGNMENT MTE SEC-9

Name – **Aryan Kumar**

Adm no – **22SCSE1010567**

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;
        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 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. **Solution:**

```
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.");
    }
}
```

Time & Space Complexity:

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 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 **Solution:**

```
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.");

    }

}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(1)

6 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.

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 **Solution:**

```
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;
        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);
    }
}
```

Time & Space Complexity:

Time Complexity: O(n) Space Complexity:

O(1)

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

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.

Solution:

```
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);

    }
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(k)

8. 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

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 & Space Complexity:

Time Complexity: O(n)

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

9. Find the Longest Palindromic Substring in a Given String

Solution:

```
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);

             len2 = expandFromCenter(s, i, i + 1);

            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) {

        System.out.println("Longest palindromic substring: " +
        longestPalindrome("babad"));

    }

}
```

Time & Space Complexity:

Time Complexity: O(n^2)

Space Complexity: O(1)

10.

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.

Solution:

```
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);

    }

}
```

Time & Space Complexity:

Time Complexity: O(n * m)

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

11.

Space Complexity: O(1)

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.

Solution:

```
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);}
}
```

Time & Space Complexity:

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

12.

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

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.
    - 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).
2. Repeat until the two pointers meet or find a pair.

Solution:

```
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[] result = twoSum({1, 2, 3, 4, 5, 6}, 10);

        System.out.println("Pair: " + result[0] + ", " + result[1]);

    }

}
```

Time & Space Complexity:

13.

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

Rearrange Numbers into the Lexicographically Next Greater Permutation

Solution:

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.

Solution:

```
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;
    }
```

14.

```
    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));

    }

}
```

Time & Space Complexity:

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

14. 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.

Solution:

```
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;
        }

    }

}
```

Time & Space Complexity:

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]

15. 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.

Solution:

```
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;
    }

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

Time & Space Complexity:

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

16. 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}
        };
        System.out.println("k-th smallest: " + kthSmallest(matrix, 8));
    }
}
```

Time & Space Complexity:

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

Space Complexity: O(n * m)

17. 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));

    }

}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(1)

18. 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]. & then 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;

         leftMax[0] = height[0]; for
        (int i = 1; i < n; i++) {

            leftMax[i] = Math.max(leftMax[i - 1], height[i]);

        }

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

        }

         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));


    }

}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(n)

19. 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.

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) {

        System.out.println("Maximum XOR: " + findMaximumXOR{3, 10, 5, 25, 2, 8}));

    }

}
```

Time & Space Complexity:

Time Complexity: O(n * 32)

Space Complexity: O(n)

20. 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.

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) {

        System.out.println("Maximum product: " + maxProduct({2, 3,-2, 4}));
    }
}
```

Time & Space Complexity:

Time Complexity: O(n) Space

Complexity: O(1)

21. Count number of 1s in binary representation from 0 to n

Algorithm:

- Use Brian Kernighan’s algorithm to count bits

- For each number from 0 to n, count the set bits

Solution:

```
public class CountSetBits {     public
static int countSetBits(int n) {         int count = 0;
for (int i = 0; i <= n; i++) {            int x = i;
while (x != 0) {                x &= (x - 1);
count++;
          }
      }        return
count;
   }

   public static void main(String[] args) {
int n = 5;
      System.out.println("Total set bits from 0 to " + n + " is: " + countSetBits(n));
   }
}
```

Time & Space Complexity:

Time Complexity : O(n log n)

Space Complexity: O(1)

Example: n=5 => (0-5): 0,1,1,2,1,2 => total 7 set bits

22. Check if number is power of two using bit manipulation Solution:

```
      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(n + " is power of two? " + isPowerOfTwo(n));
   }
```

```
}
```

Time & Space Complexity:

Time Complexity: O(1),

Space Complexity: O(1)

Example: 16 -> 10000 & 01111 = 0 => true

23. Find max XOR of two numbers in an array

Solution:

```
import java.util.*;
public class MaximumXOR {
   static class TrieNode {
      TrieNode[] children = new TrieNode[2];
   }

   public static void insert(TrieNode root, 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];
      }
   }

   public static int findMaxXOR(TrieNode root, int num) {
TrieNode node = root;        int maxXOR = 0;        for (int
i = 31; i >= 0; i--) {            int bit = (num >> i) & 1;
if (node.children[1 - bit] != null) {                maxXOR |=
(1 << i);                node = node.children[1 - bit];
```

```
            } else {                node =
node.children[bit];
            }
        }
        return maxXOR;
    }

    public static int maxXOR(int[] nums) {
TrieNode root = new TrieNode();         for
(int num : nums) insert(root, num);

        int max = 0;         for (int num : nums) max = Math.max(max,
findMaxXOR(root, num));

        return max;
    }

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

Time & Space Complexity:

Time Complexity: O(n*32)

Space Complexity: O(n*32)

24. Concept of Bit Manipulation:

Solution:

Bit manipulation uses binary representations of numbers for computation using &,|,^,<<,>> etc.

Advantages:

1. Memory efficient, operates on bits
2. Faster performance (low-level)
3. Useful for optimization problems
4. Common in encryption, graphics, compression, etc.

25. Next Greater Element using stack Solution:

```
import java.util.*;
public class NextGreaterElement {     public static int[]
nextGreater(int[] nums) {         Stack<Integer> stack = new
Stack<>();         int[] result = new int[nums.length];
Arrays.fill(result, -1);         for (int i = 0; i < nums.length; i++) {
while (!stack.isEmpty() && nums[i] > nums[stack.peek()]) {
result[stack.pop()] = nums[i];
            }
stack.push(i);
        }         return
result;
    }
    public static void main(String[] args) {
int[] nums = {4, 5, 2, 25};
        System.out.println("Next Greater Elements: " + Arrays.toString(nextGreater(nums)));
    }
}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(n)

Example: [4,5,2,25] -> [5,25,25,-1]

26. Remove n-th node from end in LinkedList

Solution: class ListNode {

```
   int val;
   ListNode next;
   ListNode(int val) { this.val = val; }
}
public class RemoveNthNode {     public static ListNode
removeNthFromEnd(ListNode head, int n) {         ListNode dummy =
new ListNode(0);         dummy.next = head;
        ListNode first = dummy, second = dummy;
        for (int i = 0; i <= n; i++) first = first.next;
while (first != null) {             first = first.next;
second = second.next;
        }
        second.next = second.next.next;
return dummy.next;
    }
}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(1)

27. Intersection Node of two linked lists

Solution:

```
public class IntersectionList {     public static ListNode
getIntersectionNode(ListNode headA, ListNode headB) {         ListNode a =
```

```
headA, b = headB;          while (a != b) {              a = (a == null) ? headB : a.next;
b = (b == null) ? headA : b.next;
        }
return a;
    }
}
```

Time & Space Complexity:

Time Complexity: O(n + m)

Space Complexity: O(1)

28. Two stacks in single array

Solution: class

```
TwoStacks {
   int[] arr;     int top1,
top2;     TwoStacks(int
size) {         arr = new
int[size];         top1 = -
1;         top2 = size;
   }
   void push1(int x) {         if (top1 + 1 <
top2) arr[++top1] = x;
   }
   void push2(int x) {         if (top1 + 1
< top2) arr[--top2] = x;
   }      int pop1() {         return top1
>= 0 ? arr[top1--] : -1;
   }

   int pop2() {         return top2 < arr.length ?
arr[top2++] : -1;
   }
}
```

Time & Space Complexity:

Time Complexity: O(1)

Space Complexity: O(n).

29. Integer is Palindrome without string

Solution:

```
public class IntegerPalindrome {     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(n + " is palindrome? " + isPalindrome(n));
    }
}
```

Time & Space Complexity:

Time Complexity: O(log10(n))

Space Complexity: O(1)

30. Linked List Concept:

A LinkedList is a linear data structure where elements (nodes) point to the next.

Applications:

1. Dynamic memory allocation
2. Efficient insertion/deletion
3. Implementing stacks, queues, graphs
4. Real-time computing where memory size is unknown.

31. Sliding Window Maximum using Deque

Solution:

```
public static List<Integer> maxSlidingWindow(int[] nums, int k) {
    List<Integer> result = new ArrayList<>();
    Deque<Integer> deque = new ArrayDeque<>();

    for (int i = 0; i < nums.length; i++) {
Remove indices out of window if
(!deque.isEmpty() && deque.peek() <= i - k)
deque.poll();

Remove smaller elements while (!deque.isEmpty() &&
nums[deque.peekLast()] < nums[i])
deque.pollLast();

deque.offer(i); if (i >= k - 1)
result.add(nums[deque.peek()]);
    } return
result;
  }
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(k)

32. Largest Rectangle in Histogram Solution:

```
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 - 1 - stack.peek();                maxArea = Math.max(maxArea,
height * width);
            }
stack.push(i);
        }
        return maxArea;
    }
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(n)

33. Sliding Window Technique Explanation

Solution:

-Used when dealing with contiguous subarrays or substrings.

- Efficiently reduces time complexity from O(n^2) to O(n).

Example: max sum subarray of size k, min window with some property.

34. Subarray Sum Equals K using HashMap

Solution:

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

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(n)

35. K Most Frequent Elements

Solution:

```
public static List<Integer> topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int num : nums) map.put(num, map.getOrDefault(num, 0) + 1);
    PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>((a, b) ->
    b.getValue() a.getValue());
pq.addAll(map.entrySet());
    List<Integer> res = new ArrayList<>();
    for (int i = 0; i < k; i++)  res.add(pq.poll().getKey());
    return res;
}
```

Time & Space Complexity:

Time Complexity: O(n log k)

Space Complexity: O(n)

36. Generate All Subsets

```
public static List<List<Integer>> subsets(int[] nums) {
List<List<Integer>> result = new ArrayList<>();
backtrackSubsets(nums, 0, new ArrayList<>(), result);
    return result;
}
private static void backtrackSubsets(int[] nums, int index, List<Integer> temp, List<List<Integer>>
res) {
    res.add(new ArrayList<>(temp));
    for (int i = index; i < nums.length; i++) {
temp.add(nums[i]);
    backtrackSubsets(nums, i + 1, temp, res);
    temp.remove(temp.size()
- 1);
    }
```

```
}
```

Time & Space Complexity:

Time Complexity: O(2^n)

Space Complexity: O(2^n)

37. Combination Sum

Solution:

```
public static List<List<Integer>> combinationSum(int[] candidates, int target) {
List<List<Integer>> result = new ArrayList<>();        backtrackComb(candidates,
0, target, new ArrayList<>(), result);        return result;
}
private static void backtrackComb(int[] candidates, int start, int target, List<Integer> comb,
List<List<Integer>> res) {
    if (target == 0) {
res.add(new ArrayList<>(comb));
        return;
    }
    for (int i = start; i < candidates.length; i++) {            if
(candidates[i] <= target) {                comb.add(candidates[i]);
backtrackComb(candidates, i, target - candidates[i], comb, res);
comb.remove(comb.size() - 1);
        }
    }
}
```

Time & Space Complexity:

Time Complexity: O(2^n)

Space Complexity: O(n)

38. Permutations Solution:

```
    public static List<List<Integer>> permute(int[] nums) {
List<List<Integer>> result = new ArrayList<>();         backtrackPerm(nums, new
ArrayList<>(), new boolean[nums.length], result);
        return result;
    }
    private static void backtrackPerm(int[] nums, List<Integer> temp, boolean[] used,
List<List<Integer>> res) {         if (temp.size() == nums.length) {             res.add(new
ArrayList<>(temp));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
if (used[i]) continue;             used[i] = true;
temp.add(nums[i]);
backtrackPerm(nums, temp, used, res);
temp.remove(temp.size() - 1);             used[i]
= false;
        }
    }
```

Time & Space Complexity:

Time Complexity: O(n!)

Space Complexity: O(n!)

39. Subsets vs Permutations

Solution:

* Subsets: All combinations, order doesn't matter. (2^n subsets)

* Permutations: All arrangements, order matters. (n! permutations)

* Example: [1,2]

* Subsets: [], [1], [2], [1,2]

* Permutations: [1,2], [2,1]

40. Max Frequency Element Solution:

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

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(n)

41. Maximum Subarray Sum using Kadane's Algorithm

Solution:

```
public class MaxSubarraySum {        public static int
maxSubArray(int[] nums) {          int maxSoFar = nums[0];          int
currentMax = nums[0];            for (int i = 1; i < nums.length; i++) {
currentMax = Math.max(nums[i], currentMax + nums[i]);
maxSoFar = Math.max(maxSoFar, currentMax);
    }
    return maxSoFar;
}
public static void main(String[] args) {
int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    System.out.println("Maximum Subarray Sum: " + maxSubArray(arr));
}
```

}

Explanation:

- Kadane’s Algorithm is a dynamic programming approach.

- At each step, it keeps track of the maximum sum subarray ending at the current index.

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(1)

42. Dynamic Programming Concept

Solution:

Dynamic Programming (DP):

- Used in optimization problems like:

1. Fibonacci numbers

2. Longest common subsequence

3. Maximum subarray (Kadane's Algorithm)

- Solves complex problems by breaking them into subproblems.& Stores the results of subproblems to avoid redundant computations.

43. Top K Frequent Elements Solution: public class

```
TopKFrequent { public static List<Integer>
topKFrequent(int[] nums, int k) { Map<Integer, Integer>
freqMap = new HashMap<>(); for (int num : nums) {
freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
    }
    PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>(
        (a, b) -> b.getValue() - a.getValue());
pq.addAll(freqMap.entrySet());
    List<Integer> result = new ArrayList<>();
for (int i = 0; i < k; i++) {
result.add(pq.poll().getKey());
```

```
        }         return
result;
    }
    public static void main(String[] args) {
int[] nums = {1, 1, 1, 2, 2, 3};         int k
= 2;
        System.out.println("Top K Frequent: " + topKFrequent(nums, k));
    }
}
```

Algorithm:

1. Count frequency of each element.
2. Add entries to a max heap based on frequency & then Extract top k elements.

Time & Space Complexity:

Time Complexity: O(n log k)

Space Complexity: O(n)

44. Two Sum using Hashing

Solution:

```
public class TwoSum {     public static int[]
twoSum(int[] nums, int target) {         Map<Integer,
Integer> map = new HashMap<>();         for (int i = 0;
i < nums.length; i++) {             int complement =
target - nums[i];             if
(map.containsKey(complement)) {                 return
new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return new int[]{};
```

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

Algorithm:

1. Iterate array, store values in hashmap.
2. Check if target - current exists in map.

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(n)

46. Longest Palindromic Substring

Solution:

```
public class LongestPalindrome {
public static String longestPalindrome(String s) {
int start = 0, end = 0;       for (int i = 0; i < s.length();
i++) {             int len1 = expand(s, i, i);             int
len2 = expand(s, i, i + 1);                 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);
    }    private static int expand(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) {
        System.out.println("Longest Palindromic Substring: " + longestPalindrome("babad"));
    }
}
```

Time & Space Complexity:

Time Complexity: O(n^2)

Space Complexity: O(1)

45. Priority Queues Concept

Priority Queue:

- A data structure where each element has a priority & Elements are dequeued in order of highest priority.

Applications:

1. Dijkstra's algorithm
2. Huffman coding
3. Task scheduling

In Java: Priority Queue class.

47. Histogram Problems Concept Histogram

Problems:

- Common problem: Largest Rectangle in Histogram.
- Uses stack to track increasing heights.

Applications:

1. Image histogram analysis
2. Memory management
3. Stock span problem

48. Next Permutation Solution:

```
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, nums.length - 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) {         while (start < end) {             swap(nums,
start++, end--);
        }
    }
    public static void main(String[] args) {
int[] arr = {1, 2, 3};
nextPermutation(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(1)

49. Intersection of Two Linked Lists Solution:

```
class ListNode {
    int val;
```

```
    ListNode next;
ListNode(int x) {
val = x;        next =
null;
    }
}
public class IntersectionLinkedList {     public ListNode
getIntersectionNode(ListNode headA, ListNode headB) {
        ListNode a = headA;
        ListNode b = headB;

        while (a != b) {            a = (a ==
null) ? headB : a.next;            b = (b ==
null) ? headA : b.next;
        }
return a;
    }
}
```

Time & Space Complexity:

Time Complexity: O(n + m)

Space Complexity: O(1)

50. Equilibrium Index Solution:

```
public class EquilibriumIndex {        public static int
findEquilibrium(int[] nums) {         int total = 0,
leftSum = 0;          for (int num : nums) total +=
num;

        for (int i = 0; i < nums.length; i++) {
total -= nums[i];            if (leftSum ==
```

```
total) return i;            leftSum +=
nums[i];
        }
return -1;
    }

    public static void main(String[] args) {
int[] arr = {-7, 1, 5, 2, -4, 3, 0};
        System.out.println("Equilibrium Index: " + findEquilibrium(arr));
    }
}
```

Time & Space Complexity:

Time Complexity: O(n)

Space Complexity: O(1)

Use case: Finding balanced point in array (financial, weights, etc.)