Advanced Algorithmic Problem Solving (R1UC601B)
Assignment for MTE

Ashish Nayak
22SCSE1012746

1. Explain the concept of a prefix sum array and its applications.
prefix sum:
A **prefix sum array** is an array where each element at index `i` contains the sum of all elements from the beginning up to index `i` of the original array.
Example:
 arr = [2, 4, 1, 3, 5]
 prefix = [2, 6, 7, 10, 15]

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.
CODE:

```
import java.util.*;

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

        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i - 1] + arr[i];
        }
        return prefix;
    }
    public static int rangeSum(int[] prefix, int L, int R) {
        if (L == 0)
            return prefix[R];
        return prefix[R] - prefix[L - 1];
    }

    public static void main(String[] args) {
        int[] arr = {3, 5, 2, 6, 1};
        int L = 1, R = 3;

        int[] prefix = buildPrefixSum(arr);
        int sum = rangeSum(prefix, L, R);

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

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

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.

CODE:

```
public class EquilibriumIndex {
    public static int findEquilibriumIndex(int[] arr) {
        int totalSum = 0;
        for (int num : arr) {
            totalSum += num;
        }
        int leftSum = 0;
        for (int i = 0; i < arr.length; i++) {
            int rightSum = totalSum - leftSum - arr[i];
            if (leftSum == rightSum) {
                return i;
            }
            leftSum += arr[i];
        }

        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 2, 2};
        int index = findEquilibriumIndex(arr);
        if (index != -1) {
            System.out.println("Equilibrium Index: " + index);
        } else {
            System.out.println("No Equilibrium Index Found");
        }
    }
}
```

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

4. Check if an array can be split into two parts such that the sum of the prefix equals the sum of the suffix. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: nums = [1,5,11,5]
Output: true

```
class Solution {
    public boolean canPartition(int[] nums) {
        int totalSum = 0;
        for (int i = 0; i < nums.length; i++) {
            totalSum = totalSum + nums[i];
        }
        if (totalSum % 2 != 0) {
            return false;
        }
        int target = totalSum / 2;
        boolean[] dp = new boolean[target + 1];
        dp[0] = true;

        for (int num : nums) {
            for (int j = target; j >= num; j--) {
                dp[j] = dp[j] || dp[j - num];
            }
        }

        return dp[target];
    }
}
```

Time Complexity: O(n × sum/2)
Space Complexity: O(sum/2)

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.
CODE:

Input: arr[] = [100, 200, 300, 400] , k = 2

Output: 700

```
class Solution {
    public int maximumSumSubarray(int[] arr, int k) {
                int n = arr.length;
        int sum = 0 ;
        int maxSum = 0;
        int index = 0;
        while ( index < n && index < k) {
            sum += arr[index];
            index++;
        }
        maxSum = sum ;

        for (int i = 1 ; i< n-k+1 ; i++) {
            int prevEle = arr[i-1];
            int nextEle = arr[i+k-1];

            sum = sum - prevEle + nextEle;
           maxSum = Math.max(maxSum , sum );
        }
        return maxSum ;
    }
}
```

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.
CODE:
Input: s = "abcabcbb"
Output: 3

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        int n = s.length();
        int i = 0, j = 0;
        int maxLength = 0;
        HashMap<Character, Integer> map = new HashMap<>();

        while (j < n) {
            char ch = s.charAt(j);

            if (map.containsKey(ch) && map.get(ch) >= i) {
                i = map.get(ch) + 1;
            }

            map.put(ch, j);
            maxLength = Math.max(maxLength, j - i + 1);
            j++;
        }
        return maxLength;
    }
}
```

7. Explain the sliding window technique and its use in string problems.

ANS:

The sliding window technique is a method used to efficiently solve problems involving arrays or strings, especially when you're looking for subarrays or substrings that satisfy certain conditions.

USE :

Finding the longest/shortest substring with certain properties.

Checking for anagrams.

Finding unique characters, or character frequencies.

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

CODE:

Input: s = "babad"

Output: "bab"

```
class Solution {
    int max = 0;
    int start = 0;
    int end = 0;
    public boolean isPalindrome (String s , int i , int j ) {

        while (i  < j)
        {
            char ch1 = s.charAt(i);
            char ch2 = s.charAt(j);
            if (ch1 != ch2){
                return false;
            }
             i++;
            j--;
        }
        return true;
    }
    public String longestPalindrome(String s) {

        for ( int i = 0 ; i< s.length(); i++)
        {
            for ( int j = i ; j< s.length() ; j++ )
            {
                if(isPalindrome(s,i,j) == true)
                {
                   if((j-i+1)> max) {
                    max = j-i+1;
                    start = i;
                    end = j;
                   }
                }
            }
        }
        return s.substring(start , end+1);
    }
}
```
The total time complexity is:
O(n^3)
(Because you have O(n^2) pairs and O(n) time to check if each substring is a palindrome.)

Space Complexity : O(1)

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

Input: strs = ["flower","flow","flight"]
Output: "fl"

```
class Solution {
    public String longestCommonPrefix(String[] strs) {
        StringBuilder ans = new StringBuilder();
        Arrays.sort(strs);
        String first = strs[0];
        String last= strs[strs.length -1];

        for (int i =0; i<Math.min(first.length(), last.length()) ; i++){
            if (first.charAt(i) != last.charAt(i)) {
                return ans.toString();
            }
            ans.append(first.charAt(i));
                }
            return ans.toString();
    }
}
```

Time Complexity	O(m × log n + k)
Space Complexity	O(k)

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

CODE:

Input: nums = [1,2,3]

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

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

    private void backtrack(int nums[], List<Integer> current,
List<List<Integer>> result) {
        if (current.size() == nums.length) {
            result.add(new ArrayList<>(current));
            return;
        }

        for (int num : nums) {
            if (current.contains(num))
                continue;
            current.add(num);
            backtrack(nums, current, result);
            current.remove(current.size() - 1);
        }
    }

}
```

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

11. Find two numbers in a sorted array that add up to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int start = 0;
        int end = numbers.length -1;

        while(start < end) {
            int sum = numbers[start]+ numbers[end];

            if (sum == target) {
                return new int [] {start+1, end+1};
            }else if (sum < target) {
                start++;
            }else {
                end--;
            }
        }
        return  new int[]{-1,-1};
    }
}
```

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

12. Rearrange numbers into the lexicographically next greater permutation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: nums = [1,2,3]
Output: [1,3,2]

```
class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 1;
        while (i > 0 && nums[i-1] >= nums[i]) {
            i--;
        }
        if (i == 0) {
            reverse(nums, 0, nums.length-1);
            return;
        }
        int j = nums.length - 1;
        while (j >= i && nums[j] <= nums[i-1]) {
            j--;
        }
        swap(nums, i-1, j);
        reverse(nums, i, nums.length-1);
    }

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

    private void reverse(int[] nums, int start, int end) {
        while (start < end) {
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }

}
```

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

13. How to merge two sorted linked lists into one sorted list. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
      ListNode dummyNode = new ListNode (-1);
      ListNode temp =   dummyNode ;
```

```
        while  (list1 != null && list2 != null){
          if(list1.val < list2.val){
              temp.next = list1 ;
               list1 = list1.next ;
          } else {
              temp.next = list2;
              list2 = list2.next;
          }
          temp = temp.next;
        }
          if(list1 != null){
              temp.next = list1;

          }else {
              temp.next = list2;

          }
        return dummyNode.next;
      }
}
```

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

14. Find the median of two sorted arrays using binary search. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {

        int n1 = nums1.length;
        int n2 = nums2.length;
        int n = n1 + n2;
        int [] new_arr = new int[n];

        int i=0, j=0, k= 0;

        while (i<=n1 &&  j<=n2) {
            if (i == n1) {
                while(j<n2) new_arr[k++] = nums2[j++];
                break;

            }
            else if(j == n2) {
                while ( i<n1) new_arr[k++] = nums1[i++];
                break;
            }
            if (nums1[i] < nums2[j]) {
                new_arr[k++] = nums1[i++];
            }
            else{
                new_arr[k++] = nums2[j++];
            }
        }
        if (n%2==0) return (float)(new_arr[n/2-1] + new_arr[n/2])/2;
        else return new_arr[n/2];
    }
}
```

Time Complexity : O(n1 + n2)
Space Complexity : O(n1 + n2)

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

Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
Output: 13

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

        while(low < high){
            int mid = low + (high - low)/2;
            int count = lessEqual(matrix,mid);
            if(count < k){
                low = mid+1;
            }
            else{
                high = mid;
            }
        }
        return low;

    }


    public int lessEqual(int[][] matrix, int target){
        int count = 0 , len = matrix.length, i = len-1, j=0;
        while(i >=0 && j<len){
            if(matrix[i][j] > target){
                i--;
            }
            else
            {
                count = count + i +1;
                j++;
            }
        }
        return count;
    }
}
```

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

16. Find the majority element in an array that appears more than n/2 times. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: nums = [3,2,3]
Output: 3

```
class Solution {
    public int majorityElement(int[] nums) {
        int n = nums.length ;
      for( int i = 0 ; i< n ;i++){
        int count = 0 ;
        for( int j = 0; j< n ; j++){
            if(nums[i]== nums[j]) {
                count ++;
            }
        }
        if( count >  n/2){
            return nums[i];
        }
      }
      return -1;
    }
}
```

Time Complexity: O(n²)
Space Complexity: O(1)

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

CODE:

Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6

```
class Solution {
    public int trap(int[] height) {
        int water = 0 ;
        int max = 0;
        for (int i = 0 ; i< height.length; i++) {
            if(max < height[i]){
                max = height[i];
            }

        }
            int lMax = -1;
        for ( int i = 0 ; i<= max[i] ; i++) {
            if (height[i] < lMax) {
             water += (lMax - height[i]);
            } else {
                left = height[i];
            }
                if (height[i] > max) {
                    water += (height[i] - max);
                }else {
                    max = height[i];
                }

        }
        return water ;
    }
}
```

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

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

CODE:
Input: nums = [3,10,5,25,2,8]
Output: 28

```
class Solution {
    public int findMaximumXOR(int[] nums) {
        int res = 0;
        int mask = 0;
        int n = nums.length;

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

            for (int j = 0; j < n; j++) {
                set.add(nums[j] & mask);
            }

            int target = res | (1 << i);

            for (int prefix : set) {
                if (set.contains(prefix ^ target)) {
                    res = target;
                    break;
                }
            }
        }

        return res;
    }
}
```

19. How to find the maximum product subarray. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

CODE:
Input: nums = [2,3,-2,4]
Output: 6

```
class Solution {
    public int maxProduct(int[] nums) {
        int max_product = nums[0];
        int n = nums.length;
        for (int i = 0; i< n; i++)
        {
            int product = 1;
            for (int j = 0 ; j <n ; j++)
            {
              product *= nums[j];
              max_product = Math.max(max_product, product);

            }
        }
        return max_product;
    }
}
```

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

CODE:

Input: n = 2

Output: 91

```
class Solution {
    public int countNumbersWithUniqueDigits(int n) {
        int temp = 1;
        if(n>0){
            for(int i=0; i<n; i++){
                int p = 1;
                for(int j=0; j<=i; j++){
                    if(j==0){
                        p*=9;
                    }
                    else{
```

```
                    p*=10-j;
                }
            }
            temp += p;
        }
    }
    return temp;
  }
}
```

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. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:

```
import java.util.Arrays;

public class Solution {
    public static int[] countBits(int n) {
        int[] res = new int[n + 1];
        res[0] = 0;

        for (int i = 1; i <= n; i++) {
            res[i] = res[i >> 1] + (i & 1);
        }

        return res;
    }

    public static void main(String[] args) {
        int n = 5;
        int[] result = countBits(n);
        System.out.println("Count of 1s from 0 to " + n + ": " +
Arrays.toString(result));
    }
}
```

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

22. How to check if a number is a power of two using bit manipulation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: n = 1
Output: true

```
public class Solution {
    public boolean isPowerOfTwo(int n) {
        for (int i = 0; i < 31; i++) {
            int ans = (int) Math.pow(2, i);
            if (ans == n) {
                return true;
            }
        }
        return false;
    }
}
```

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

CODE:
Input: nums = [0,1,2,3,4], queries = [[3,1],[1,3],[5,6]]
Output: [3,3,7]

```
class Solution {
     class Node {
        Node[] links = new Node[2];

        public Node() {

        }

        boolean containsKey(int bit){
            return links[bit] != null;
        }

        Node get(int bit){
            return links[bit];
        }

        void put(int bit, Node node){
            links[bit] = node;
        }
    }
    class Trie {
        private Node root;

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

        public void insert(int num){
            Node node = root;
            for(int i = 31; i >= 0; i--){
                int bit = (num >> i) & 1;
                if(!node.containsKey(bit)){
                    node.put(bit, new Node());
                }

                node = node.get(bit);
            }
        }

        public int getMax(int num){
            Node node = root;
            int maxNum = 0;
            for(int i = 31; i >= 0; i--){
                int bit = (num >> i) & 1;
                if(node.containsKey(1 - bit)){
                    maxNum = maxNum | (1 << i);
                    node = node.get(1 - bit);
                }else{
                    node = node.get(bit);
                }
            }

            return maxNum;
        }
    }
    public int[] maximizeXor(int[] nums, int[][] queries) {
        int queriesLength = queries.length;
        int[] ans = new int[queriesLength];
        int[][] temp = new int[queriesLength][3];
        for (int i = 0; i < queriesLength; i++) {
            temp[i][0] = queries[i][0];
            temp[i][1] = queries[i][1];
            temp[i][2] = i;
        }

        Arrays.sort(temp, (a, b) -> {
            return a[1] - b[1];
        });

        Arrays.sort(nums);

        Trie trie = new Trie();

        int ind = 0;
        for(int[] q : temp){
            while(ind < nums.length && nums[ind] <= q[1]){
                trie.insert(nums[ind]);
                ind++;
            }
            if(ind == 0){
                ans[q[2]] = -1;
            }else{
                ans[q[2]] = trie.getMax(q[0]);
            }
        }

        return ans;
    }
}
```

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

Bit manipulation is the act of algorithmically manipulating bits or binary digits (0 and 1), which are the most basic form of data in computing. It involves using bitwise operators such as AND (`&`), OR (`|`), XOR (`^`), NOT (`~`), left shift (`<<`), and right shift (`>>`) to perform operations directly on the bits of integers.

Advantages of Bit Manipulation in Algorithm Design:

1. Speed and Performance

    - Bitwise operations are very fast and often take just one CPU cycle.

    - They can significantly improve performance in time-critical applications.

2. Memory Efficiency

    - Bitwise operations are very fast and often take just one CPU cycle.

    - They can significantly improve performance in time-critical applications.

    - Bitwise operations are very fast and often take just one CPU cycle.

    - They can significantly improve performance in time-critical applications.

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

CODE:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2]

Output: [-1,3,-1]

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        HashMap<Integer, Integer> nextGreaterMap = new
HashMap<>();


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


        for (int num : nums2) {
            while (!stack.isEmpty() && num > stack.peek()) {
                nextGreaterMap.put(stack.pop(), num);
            }
            stack.push(num);
        }
        while (!stack.isEmpty()) {
            nextGreaterMap.put(stack.pop(), -1);
        }
        int[] result = new int [nums1.length];


        for (int i = 0; i < nums1.length; i++) {
            result[i] = nextGreaterMap.get(nums1[i]);
        }
        return result;
    }
}
```

Time Complexity : O(n + m)

Space Complexity: O(n + m)

26. Remove the n-th node from the end of a singly linked list. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

CODE:
Input: head = [1,2,3,4,5], n = 2

Output: [1,2,3,5]

```
class Solution {

    public ListNode removeNthFromEnd(ListNode head, int n) {

        ListNode res = new ListNode(0, head);

        ListNode dummy = res;


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

            head = head.next;

        }

        while (head != null){

            head = head.next;

            dummy = dummy.next;

        }

        dummy.next = dummy.next.next;


        return res.next;

    }
}
```

Time Complexity:

First loop: move head forward by n → O(n)

Second loop: move head and dummy until end → O(L - n), where L = total number of nodes

So overall, you traverse the list once:

Time Complexity: O(L), where L is the number of nodes in the list.

Space Complexity: O(1)

27. Find the node where two singly linked lists intersect. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

CODE:

```
class ListNode {

    int val;

    ListNode next;

    ListNode(int x) {

        val = x;

        next = null;

    }

}
```

```
public class Solution {

    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 Complexity: O(m + n)

Space Complexity: O(1)

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

CODE:

Input: target = [1,3], n = 3

Output: ["Push","Push","Pop","Push"]

```
class Solution {
    public List<String> buildArray(int[] target, int n) {
        List<String> result = new ArrayList<>();
        int num = 1, index = 0;

        while (num <= n && index < target.length) {
            if (num == target[index]) {
                result.add("Push");
                index++;
            } else {
                result.add("Push");
                result.add("Pop");
            }
            num++;
        }
        return result;
    }
}
```

Time complexity: O(n)

Space complexity: O(n)

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

CODE:

Input: x = 121

Output: true

```
class Solution {
    public boolean isPalindrome(int x) {
        if(x < 0){
            return false ;
        }
        long reversed = 0;
        long temp = x;

        while (temp !=0) {
            int digit = (int) (temp % 10);
            reversed = reversed * 10 + digit;
            temp /=10;
        }
        return (reversed == x);
    }
}
```

Time Complexity: O(log x)

Space Complexity: O(1)

30. Explain the concept of linked lists and their applications in algorithm design.

A linked list is a linear data structure where elements (called nodes) are stored in non-contiguous memory locations. Each node contains:

Types:

1. Singly Linked List – each node points to the next node only
2. Doubly Linked List – each node points to both the next and previous nodes
3. Circular Linked List – the last node links back to the first node

Operation	**Time Complexity**
Insert at head	O(1)
Insert at tail	O(n)
Delete Node	O(n)
Search	O(n)

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

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

```
public class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || k <= 0) return new int[0];

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

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

            if (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
                deque.pollFirst();
            }


            while (!deque.isEmpty() && nums[deque.peekLast()] <
nums[i]) {
                deque.pollLast();
            }


            deque.offerLast(i);


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

        return result;
    }
}
```

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

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

Input: heights = [2,1,5,6,2,3]
Output: 10

```
class Solution {
    public int largestRectangleArea(int[] heights) {
         int n = heights.length;
        int maxArea = 0 ;
        int nsr[] = new int [n];
        int nsl[] = new int [n];

        //Next smaller right
        Stack <Integer> s = new Stack<>();

        for ( int i = n-1; i >= 0 ; i--) {
            while ( !s.isEmpty() && heights[s.peek()] >= heights[i]) {
                s.pop();
            }
            if(s.isEmpty() ) {
                nsr[i] = n ;
            } else {
                nsr[i] = s.peek();
            }
            s.push(i);
        }

        //next samller left
        s = new Stack<>();

        for (int i = 0 ; i< n ; i++) {
            while (!s.isEmpty() && heights[s.peek()] >= heights[i]) {
                s.pop();
            }
            if(s.isEmpty()) {
                nsl[i] = -1;
            } else {
                nsl[i]= s.peek();
            }
        s.push(i);
        }

        //current Area

        for ( int i = 0 ; i< n ; i++) {
            int height = heights[i];
            int width = nsr[i]-nsl[i]-1;
            int currArea = height * width;
            maxArea = Math.max(currArea , maxArea);
        }
        return maxArea;
    }
}
```

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

33. Explain the sliding window technique and its applications in array problems.

The sliding window technique is used to optimize problems that require analyzing contiguous subarrays or substrings.

Types of Sliding Windows:

1. Fixed Size Window

Window size `k` is constant

Example: Maximum sum of `k` consecutive elements

2. Variable Size Window

Window expands and shrinks dynamically based on conditions

Example: Longest substring without repeating characters.

34. Solve the problem of finding the subarray sum equal to K using hashing. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:

Input: nums = [1,1,1], k = 2
Output: 2

Time complexity: O(n)
Space complexity: O(n)class Solution {

```
    public int subarraySum(int[] nums, int k) {
        HashMap<Integer, Integer> subNum = new HashMap<>();
        subNum.put(0, 1);
        int total = 0, count = 0;

        for (int n : nums) {
            total += n;

            if (subNum.containsKey(total - k)) {
                count += subNum.get(total - k);
            }

            subNum.put(total, subNum.getOrDefault(total, 0) + 1);
        }

        return count;
    }
}
```

Time complexity: O(n)
Space complexity: O(n)

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

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

        PriorityQueue<int[]> maxHeap = new PriorityQueue<>(
            (a, b) -> b[0] - a[0]
        );
        for (Map.Entry<Integer, Integer> entry : freq.entrySet()) {
            maxHeap.add(new int[]{entry.getValue(), entry.getKey()});
        }

        List<Integer> result = new ArrayList<>();
        while (k-- > 0) {
            result.add(maxHeap.poll()[1]);
        }

        return result;
    }
}
```

Time complexity: O(nlogn)
Space complexity: O(n)

36. Generate all subsets of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        result.add(new ArrayList<>());

        for(int num : nums){
            int size = result.size();
            for(int i = 0 ; i< size ; i++){
                List<Integer> subset = new ArrayList<>(result.get(i));
                subset.add(num);
                result.add(subset);
            }
        }
        return result;
    }
}
```

Time Complexity : O(n × 2^n)
Space Complexity: O(n × 2^n)

37. Find all unique combinations of numbers that sum to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]

```
class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int
target) {
        List<List<Integer>> res = new ArrayList<>();

        makeCombination(candidates, target, 0, new ArrayList<>(), 0,
res);
        return res;
    }

    private void makeCombination(int[] candidates, int target, int idx,
List<Integer> comb, int total, List<List<Integer>> res) {
        if (total == target) {
            res.add(new ArrayList<>(comb));
            return;
        }

        if (total > target || idx >= candidates.length) {
            return;
        }

        comb.add(candidates[idx]);
        makeCombination(candidates, target, idx, comb, total +
candidates[idx], res);
        comb.remove(comb.size() - 1);
        makeCombination(candidates, target, idx + 1, comb, total,
res);
    }
}
```

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

CODE:

Input: nums = [1,2,3]

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

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        if (nums.length == 1) {
            List<Integer> singleList = new ArrayList<>();
            singleList.add(nums[0]);
            res.add(singleList);
            return res;
        }
        for (int i = 0; i < nums.length; i++) {
            int n = nums[i];
            int[] remainingNums = new int[nums.length - 1];
            int index = 0;
            for (int j = 0; j < nums.length; j++) {
                if (j != i) {
                    remainingNums[index] = nums[j];
                    index++;
                }
            }

            List<List<Integer>> perms = permute(remainingNums);
            for (List<Integer> p : perms) {
                p.add(n);
            }

            res.addAll(perms);
        }

        return res;
    }
}
```
Time complexity: O(n * n!)
Space complexity: O(n)

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

Subsets:
A subset is a selection of elements from a set where order doesn't matter.

Example: For {1, 2}, subsets are: [], [1], [2], [1, 2].

Permutations:

A permutation is an ordered arrangement of elements.

Example: For {1, 2}, permutations are: [1, 2], [2, 1].

Key Difference:

Subsets → Order doesn't matter

Permutations → Order does matter

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

CODE:

Input: nums = [1,2,2,3,1,4]

Output: 4

```
class Solution {
    public int maxFrequencyElements(int[] nums) {
        HashMap<Integer, Integer> h=new HashMap<>();
        for(int num:nums){
            if(h.containsKey(num)){
                h.put(num,h.get(num)+1);
            }
            else{
                h.put(num,1);
            }
        }
        int mx=0;
        for(int val:h.values()){
            mx=Math.max(val,mx);
        }
        int res=0;
        for(int frq:h.values()){
            if(frq==mx){
                res+=frq;
            }
        }
        return res;
    }
}
```

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

41. Write a program to find the maximum subarray sum using Kadane’s algorithm.
CODE:
Input: arr[] = [2, 3, -8, 7, -1, 2, 3]
Output: 11

```
class Solution {
    public int maxSubArray(int[] nums) {
        int n = nums.length ;
        int max_sum = Integer.MIN_VALUE;
        int sum = 0 ;

        for ( int i = 0 ; i < n ; i++)
        {
            sum += nums[i];

            if(  max_sum < sum  )
            {
                max_sum = sum ;
            }

             if (sum < 0)
            {
               sum = 0;
            }
        }
        return max_sum;
    }
}
```

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

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

Dynamic Programming:

Dynamic Programming is a method used to solve problems by breaking them down into overlapping subproblems, solving each subproblem only once, and storing the result for future use (usually in a table or by using variables).

Solves the problem in linear time (O(n)) using constant space (O(1)).

43. Solve the problem of finding the top K frequent elements in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

```
class Solution {
    class Pair implements Comparable<Pair> {
        int num, count;
        Pair(int num, int count) {
            this.num = num;
            this.count = count;
        }
        public int compareTo(Pair b) {
            return this.count - b.count;
        }
    }

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

        PriorityQueue<Pair> pq = new PriorityQueue<>();
        Set<Integer> keys = mp.keySet();

        for (int key : keys) {
            if (pq.size() < k) {
                pq.add(new Pair(key, mp.get(key)));
            } else if (pq.peek().count < mp.get(key)) {
                pq.poll();
                pq.add(new Pair(key, mp.get(key)));
            }
        }

        int[] arr = new int[k];
        int i = k - 1;
        while (i >= 0) {
            arr[i--] = pq.poll().num;
        }
        return arr;
    }
}
```

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. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]

```
    public static int[] twoSum_bf(int[] numbers, int target) {
        int len = numbers.length;
        for (int i = 0; i < len; i++) {
            for (int j = i + 1; j < len; j++) {
                if (numbers[i] + numbers[j] == target) {
                    return new int[]{i + 1, j + 1};
                }
            }
        }

        return new int[0];
    }
```

45. Explain the concept of priority queues and their applications in algorithm design.
A Priority Queue is a special type of queue where each element has a priority, and elements are served based on priority, not just order of insertion.

In most implementations , it behaves like a min-heap by default — the smallest (or highest-priority) element comes out first.

Operation	Description	Time Complexity
Insert()	Add an element	O(log n)
peek()	View the top-priority element	O(1)
remove()	Remove the top-priority element	O(log n)

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

```
Input: s = "babad"
Output: "bab"

class Solution {
    int max = 0;
    int start = 0;
    int end = 0;
    public boolean isPalindrome (String s , int i , int j ) {


        while (i  < j)
        {
            char ch1 = s.charAt(i);
            char ch2 = s.charAt(j);
            if (ch1 != ch2){
                return false;

            }
             i++;
            j--;
        }
        return true;
    }
    public String longestPalindrome(String s) {

        for ( int i = 0 ; i< s.length(); i++)
        {
            for ( int j = i ; j< s.length() ; j++ )
            {
                if(isPalindrome(s,i,j) == true)
                {
                   if((j-i+1)> max) {
                    max = j-i+1;
                    start = i;
                    end = j;
                   }
                }
            }
        }
        return s.substring(start , end+1);
    }
}
```

Time Complexity : O(n^3)
Space Complexity : O(1)

47. Explain the concept of histogram problems and their applications in algorithm design.
A histogram problem typically involves analyzing a bar chart-like structure, where you are given an array of bar heights and asked to

find patterns or solve problems related to area, visibility, or containment.
Solving Histogram Problems:

- Use a stack to keep track of indices of increasing bar heights.
- When a smaller bar is found, pop from the stack and calculate possible rectangle areas.
- This reduces the brute-force $O(n^2)$ solution to O(n).

48. Solve the problem of finding the next permutation of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

CODE:
Input: nums = [1,2,3]
Output: [1,3,2]

```
class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 1;
        while (i > 0 && nums[i-1] >= nums[i]) {
            i--;
        }
        if (i == 0) {
            reverse(nums, 0, nums.length-1);
            return;
        }
        int j = nums.length - 1;
        while (j >= i && nums[j] <= nums[i-1]) {
            j--;
        }
        swap(nums, i-1, j);
        reverse(nums, i, nums.length-1);
    }

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

    private void reverse(int[] nums, int start, int end) {
        while (start < end) {
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }

}
```

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

49. How to find the intersection of two linked lists. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.
CODE:
intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3
Output: Intersected at '8'

```
class Solution {
  public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
      if(headA == null || headB == null){
        return null;
      }
      ListNode a = headA;
      ListNode b = headB;

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

}
```

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

50. Explain the concept of equilibrium index and its applications in array problems.

An equilibrium index in an array is an index i such that the sum of elements on the left of i is equal to the sum of elements on the right of i.

Time & Space Complexity:

Time: O(n)

Space: O(1)

Applications:

- Load Balancing: In distributed systems or scheduling, where tasks need to be evenly balanced.
- Financial Modeling: When tracking balance points in profit/loss arrays.
- Sensor Data Analysis: Finding steady-state points in fluctuating datasets.
- Game Development: Finding balance points in health, score, or resource arrays.