Assignment AAPS:-

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

Ans:- Prefix Sum is the Sum of Range in an Array

Applications

1. Range Sum Queries

Quickly compute the sum of a subarray arr[l...r] in O(1) time after O(n) preprocessing:

2. Solving Range-Based Problems Efficiently

Examples:

- Finding maximum/minimum sum subarrays
- Sliding window sums
- Kadane’s algorithm optimizations
- Histogram problems
- 2D Prefix sums for matrix-based sum queries

3. Binary Search + Prefix Sum

Used in problems like:

- Minimum days to ship packages
- Finding a peak point or target range using accumulated sums

4. Difference Arrays

Prefix sums are also used in reverse (through difference arrays) to apply multiple updates over a range in constant time.

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.    Construct the prefix sum array:
o     prefix[0] = arr[0]
o     For i from 1 to n-1:
prefix[i] = prefix[i-1] + arr[i]
3.    To get the sum in range [L, R]:
o     If L == 0, result = prefix[R]
o     Else, result = prefix[R] - prefix[L - 1]
```

Program :

```
import java.util.*;

public class PrefixSumRangeQuery {
  public static int rangeSum(int[] arr, int L, int R) {
        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];
        }

        if (L == 0) {
            return prefix[R];
        } else {
            return prefix[R] - prefix[L - 1];
        }
    }

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

        int result = rangeSum(arr, L, R);
        System.out.println("Sum of elements from index " + L + " to " + R + " is: " +
result);
    }
}
```

Example :
Int [] arr = [2, 4, 1, 5, 3]
prefix = [2, 6, 7, 12, 15]

Query: L = 1, R = 3

Sum = prefix[3] - prefix[0] = 12 - 2 = 10

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

```
Ans:- public class EquilibriumIndexFinder {

    public static int findEquilibriumIndex(int[] arr) {
        int totalSum = 0;
        int leftSum = 0;
        // Step 1: Calculate total sum
        for (int num : arr) {
            totalSum += num;
        }

        // Step 2: Traverse and find equilibrium index
        for (int i = 0; i < arr.length; i++) {
            int rightSum = totalSum - arr[i] - leftSum;

            if (leftSum == rightSum) {
                return i;
            }

            leftSum += arr[i];
        }

        // No equilibrium index found
        return -1;
    }

    // Sample usage
    public static void main(String[] args) {
        int[] arr = {-7, 1, 5, 2, -4, 3, 0};
        int index = findEquilibriumIndex(arr);

        if (index != -1) {
            System.out.println("Equilibrium Index: " + index);
        } else {
            System.out.println("No Equilibrium Index found.");
        }
    }
}
```

1. **Time Complexity:** O(n)
2. **Space Complexity:** O(1) — no extra data structures are used, just variables.

Output :-

```
int[] arr = {-7, 1, 5, 2, -4, 3, 0};
```

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

Ans:- **Algorithm (Efficient Linear Approach)**

Steps:

1. Compute the **total sum** of the array.
2. Initialize a variable leftSum = 0.
3. Traverse the array:
   - Add arr[i] to leftSum.
   - The remaining sum is rightSum = totalSum - leftSum.
   - If leftSum == rightSum, a valid split exists.
4. If the loop ends without finding it, return false.

Code:-

```
public class EqualPrefixSuffixSplit {


    public static boolean canBeSplit(int[] arr) {

        int totalSum = 0;

        for (int num : arr) {

            totalSum += num;

        }


        int leftSum = 0;

        for (int i = 0; i < arr.length - 1; i++) {  // stop at n - 1 to ensure both parts are non-
empty

            leftSum += arr[i];

            int rightSum = totalSum - leftSum;


            if (leftSum == rightSum) {

                return true;
            }
        }

        return false;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 3};
        if (canBeSplit(arr)) {
            System.out.println("Yes, array can be split into two parts with equal sum.");
        } else {
            System.out.println("No, array cannot be split into such parts.");
        }
    }
}
```

Example

Input:

java

CopyEdit

arr = {1, 2, 3, 3}

- Total sum = 9
- Traverse:
  - i = 0: leftSum = 1, rightSum = 8
  - i = 1: leftSum = 3, rightSum = 6
  - i = 2: leftSum = 6, rightSum = 3
  - i = 3: Not checked (we avoid splitting at the end)
- **Output:**
- Yes, array can be split into two parts with equal sum.

Time and Space Complexity

- **Time Complexity:** O(n) — one pass for sum, one for check.
- **Space Complexity:** O(1) — no extra space used.

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 Approach)
Instead of recomputing the sum every time (which takes O(K)), we **slide a window of size K** through the array and update the sum **incrementally**.
Steps:

1. Compute the sum of the **first K elements** → this is our initial window.
2. Then, move the window one element at a time:
   - Subtract the element going out of the window
   - Add the element coming into the window
3. Keep track of the **maximum sum** seen so far.

Code:-
```
public class MaxSubarraySumOfSizeK {

  public static int maxSumSubarray(int[] arr, int k) {
    if (arr.length < k) {
      throw new IllegalArgumentException("Array size is smaller than K");
    }

    // Step 1: Compute initial window sum
    int maxSum = 0;
    for (int i = 0; i < k; i++) {
      maxSum += arr[i];
    }

    int windowSum = maxSum;

    // Step 2: Slide the window
    for (int i = k; i < arr.length; i++) {
      windowSum += arr[i] - arr[i - k];
      maxSum = Math.max(maxSum, windowSum);
    }

    return maxSum;
    }

    public static void main(String[] args) {
        int[] arr = {2, 1, 5, 1, 3, 2};
        int k = 3;
        int result = maxSumSubarray(arr, k);
        System.out.println("Maximum sum of subarray of size " + k + " is: " + result);
    }
}
```

Example

Input:

arr = {2, 1, 5, 1, 3, 2}, k = 3

Window Calculations:

- [2, 1, 5] → sum = 8
- [1, 5, 1] → sum = 7
- [5, 1, 3] → sum = 9 (maximum)
- [1, 3, 2] → sum = 6

Output:

- Maximum sum of subarray of size 3 is: 9

Time & Space Complexity

- **Time Complexity:** `O(n)`
  - One pass through the array to calculate max sum.
- **Space Complexity:** `O(1)`
  - No extra space needed beyond a few variables.

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:-We use two pointers (a window) to track the current substring. As we move the right pointer to expand the window:

- If we hit a repeating character, we move the left pointer forward until the substring becomes unique again.
- Track the **maximum length** at each step.

Steps:

1. Initialize a set (or map) to store characters in the current window.
2. Use two pointers: start and end.
3. Move end pointer through the string:

- If the character is not in the set, add it and update max length.
- If it's already in the set, remove characters from the left (start) until the repeating character is removed.

Code:-

```
import java.util.HashSet;

public class LongestUniqueSubstring {

    public static int lengthOfLongestSubstring(String s) {

        HashSet<Character> seen = new HashSet<>();

        int maxLen = 0;

        int start = 0;



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

            char current = s.charAt(end);



            while (seen.contains(current)) {

                seen.remove(s.charAt(start));

                start++;

            }



            seen.add(current);

            maxLen = Math.max(maxLen, end - start + 1);

        }



        return maxLen;

    }

    public static void main(String[] args) {
        String input = "abcabcbb";

        int result = lengthOfLongestSubstring(input);

        System.out.println("Length of longest substring without repeating characters: " + result);

    }

}
```

Example

Input:

"abcabcbb"

Window Movement:

- "a" → length = 1
- "ab" → length = 2
- "abc" → length = 3 ✅
- "abca" → repeats 'a', shrink window
- Continue until end
- Max = 3

Output: Length of longest substring without repeating characters: 3

Time & Space Complexity

- **Time Complexity:** O(n)
  - Each character is added and removed from the set at most once.
- **Space Complexity:** O(min(n, m))
  - n is the length of the string, m is the size of the character set

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

It's a way of optimizing brute-force solutions that involve checking **all contiguous segments** (subarrays or substrings) of a given size or condition.

Types of Sliding Windows

1. **Fixed-size window:**
   Window has a constant size (e.g., length k).
   → Common in problems like maximum sum of subarray of size k.
2. **Variable-size window:**
   Window size changes depending on the conditions.

→ Useful in problems like longest substring without repeating characters, where the window expands and shrinks dynamically.

How Its Works.

```
Initialize window pointers: start = 0, end = 0

Initialize data structures (set/map/sum/counter/etc.)

while end < n:

    Expand the window by including arr[end]

    While window is invalid:

        Shrink the window from the start

    Update the result if needed

    Move end forward
```

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

Ans:- **Algorithm (Expand Around Center – Optimal for Interviews)**

A palindrome mirrors around its center.
There are **2n - 1 centers** (each character + space between characters).

We **expand around each center** to find the maximum palindromic substring.

Steps:

1. Loop through each character in the string.
2. For each character:
    - Expand for **odd-length** palindromes (center at one character).
    - Expand for **even-length** palindromes (center between two characters).
3. Update the longest palindrome found.

Code:-

```
public class LongestPalindromicSubstring {

    public static String longestPalindrome(String s) {

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



        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);     // Odd-length
            int len2 = expandAroundCenter(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);
    }
    private static int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;  // Length of palindrome
    }
    public static void main(String[] args) {
        String input = "babad";
        String result = longestPalindrome(input);
        System.out.println("Longest palindromic substring: " + result);
    }
}
```

Example Input:"babad"

Palindromic substrings:

- "bab" ✅
- "aba" ✅
  ("bab" or "aba" are both valid outputs)

Output: Longest palindromic substring: bab

Time and Space Complexity

- **Time Complexity:** O(n^2)
  - We expand from each center which can take up to O(n) per center.
  - There are 2n - 1 centers.
- **Space Complexity:** O(1)
  - No extra space (excluding the output substring).

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.

Ans:- ✅ Algorithm (Vertical Scanning):

```
. Take the first string as reference.
.  For each character position I:
.  Compare the character with all other strings at position i.
.  If mismatch or end of any string → return prefix till i.
.  If loop completes, whole first string is the prefix.
```

✅ Time Complexity:
. O(N × M)
. (N = number of strings, M = length of shortest string)

✅ Space Complexity:
. O(1)

. (Constant extra space)

✅ Example:
Input: ["flower", "flow", "flight"]
Output: "fl"

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

Ans:-

Algorithm (Backtracking)

Use recursion and backtracking:

- At each step, **fix a character** and **recursively permute the remaining characters**.
- Swap characters in-place to avoid extra space.
- Backtrack (i.e., **undo the swap**) after each recursive call.

Steps:

1. Convert the string into a character array.
2. Define a recursive function:
    - Base Case: If the current index == length, print/store the permutation.
    - Loop from current index to end:
        - Swap current index with loop index.
        - Recurse for next index.
        - Swap back (backtrack).

Code:-

```
public class StringPermutations {

  public static void generatePermutations(String str) {

        char[] chars = str.toCharArray();

        permute(chars, 0);

    }

    private static void permute(char[] chars, int index) {

        if (index == chars.length) {

            System.out.println(new String(chars));

            return;

        }

        for (int i = index; i < chars.length; i++) {

            swap(chars, i, index);             // Choose

            permute(chars, index + 1);         // Explore

            swap(chars, i, index);             // Un-choose (backtrack)
        }

    }

    private static void swap(char[] arr, int i, int j) {

        char temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

    public static void main(String[] args) {

        String input = "abc";

        System.out.println("Permutations of '" + input + "':");

        generatePermutations(input);

    }

}
```

Example

Input:"abc"

Output:abc

```
acb

bac

bca

cba

cab
```

Time and Space Complexity

Time Complexity O(n × n!)
Space Complexity O(n) (stack depth)

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.

Ans:-

Algorithm (Next Permutation)

This algorithm is based on the concept of **finding a pivot** and **swapping** to get the smallest greater arrangement.

Steps:

1. **Find the first decreasing element from the end**:
   - Traverse from right to left and find the first index i such that nums[i] < nums[i + 1].
2. **If no such i exists**, the array is the last permutation:
   - Reverse the entire array to get the smallest (i.e., sorted) permutation.
3. **Find the next greater element** to nums[i] on the right:
   - Find the smallest element greater than nums[i] to the right of i.
4. **Swap** nums[i] with that next greater element.
5. **Reverse the part** of the array from i + 1 to the end:
   - This ensures the next permutation is the smallest possible after i

Code:-

```
import java.util.Arrays;

public class NextPermutation {

    public static void nextPermutation(int[] nums) {

        int n = nums.length;

        int i = n - 2;



        // Step 1: Find the first decreasing element from the right

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

            i--;

        }



        if (i >= 0) {

            // Step 2: Find the next greater element on right

            int j = n - 1;
            while (nums[j] <= nums[i]) {
                j--;
            }

            // Step 3: Swap
            swap(nums, i, j);
        }
        // Step 4: Reverse the suffix
        reverse(nums, i + 1, n - 1);
    }
    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
    }
    private static void reverse(int[] arr, int start, int end) {
        while (start < end) {
            swap(arr, start++, end--);
        }
    }
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        System.out.println("Original: " + Arrays.toString(nums));
        nextPermutation(nums);
        System.out.println("Next permutation: " + Arrays.toString(nums));
    }
}
```

Example Input:[1, 2, 3]

Steps:

- From right, find first decreasing: 2 at index 1
- Find just bigger: 3 at index 2
- Swap → [1, 3, 2]
- Reverse from index 2: already just one element → Done

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

Time and Space Complexity

Time Complexity	O(n) (one pass + reverse)
Space Complexity	O(1) (in-place)

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

Ans:- ✅ Algorithm (Iterative approach):

1. Create a dummy node to serve as the head of the merged list.
2. Use a pointer tail to build the merged list.
3. Compare nodes of both lists:
4. Append the smaller node to tail.
5. Move the pointer in that list forward.
6. After loop, append the remaining nodes (if any).
7. Return dummy.next (merged list head).

✅ Time Complexity:
. O(n + m)
. (Each node from both lists is processed once)

✅ Space Complexity:
. O(1)
. (In-place merging, no extra space used)

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

Ans:- **Algorithm (Binary Search on Partition Index)**

Steps:

1. Assume nums1 is the **smaller** array (m ≤ n)
2. Binary search on nums1:
   - Partition nums1 at index i
   - Partition nums2 at index j = (m + n + 1) / 2 - i
3. Check if:

   maxLeft1≤minRight2andmaxLeft2≤minRight1

4. If so, compute the median:
   - If total is odd → median = max(left parts)
   - If even → median = average of max(left) and min(right)
5. Else adjust binary search boundaries.

Code:-

```
public class MedianSortedArrays {

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

        if (nums1.length > nums2.length) {

            return findMedianSortedArrays(nums2, nums1); // Ensure nums1 is smaller

        }



        int x = nums1.length;

        int y = nums2.length;

        int low = 0, high = x;

        while (low <= high) {

            int partitionX = (low + high) / 2;

            int partitionY = (x + y + 1) / 2 - partitionX;

            int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE : nums1[partitionX - 1];

            int minRightX = (partitionX == x) ? Integer.MAX_VALUE : nums1[partitionX];

            int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE : nums2[partitionY - 1];
            int minRightY = (partitionY == y) ? Integer.MAX_VALUE : nums2[partitionY];

            if (maxLeftX <= minRightY && maxLeftY <= minRightX) {

                if ((x + y) % 2 == 0) {

                    return (Math.max(maxLeftX, maxLeftY) + Math.min(minRightX, minRightY)) /
2.0;

                } else {

                    return Math.max(maxLeftX, maxLeftY);

                }

            } else if (maxLeftX > minRightY) {

                high = partitionX - 1;

            } else {

                low = partitionX + 1;

            }

        }

        throw new IllegalArgumentException("Input arrays are not sorted.");

    }

    public static void main(String[] args) {

        int[] nums1 = {1, 3};

        int[] nums2 = {2};

        double median = findMedianSortedArrays(nums1, nums2);

        System.out.println("Median is: " + median);

    }

}
```

Example

Input:nums1 = [1, 3]

nums2 = [2]

Combined = [1, 2, 3] → Median = 2

Output: Median is: 2.0

Time and Space Complexity

Time Complexity O(log(min(m, n)))
Space Complexity O(1) (no extra space used)

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.

Ans:-

Algorithm (Binary Search on Value Range)
Steps:

```
1. Set low = matrix[0][0] and high = matrix[n-1][n-1]
2. While low < high:
      o  Set mid = (low + high) / 2
      o  Count elements ≤ mid using a helper function
      o  If count < k → low = mid + 1
      o  Else → high = mid
3. Return low (it will converge to the k-th smallest value)
```

Code:-

```
public class KthSmallestInSortedMatrix {

  public static 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 = countLessEqual(matrix, mid, n);

      if (count < k) {
        low = mid + 1;
      } else {
        high = mid;
      }
    }
```

```
        return low;
    }

    private static int countLessEqual(int[][] matrix, int target, int n) {
        int count = 0;
        int row = n - 1;  // start from bottom-left
        int col = 0;

        while (row >= 0 && col < n) {
            if (matrix[row][col] <= target) {
                count += row + 1;
                col++;
            } else {
                row--;
            }
        }

        return count;
    }

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

Input:

```
matrix = [
  [1, 5, 9],
  [10, 11, 13],
  [12, 13, 15]
]
k = 8
```

Sorted matrix values:

[1, 5, 9, 10, 11, 12, 13, **13**, 15]

→ 8th smallest element = **13**

Output:

```
The 8-th smallest element is: 13
```

Time and Space Complexity

Time Complexity O(n * log(max - min))

Space Complexity O(1) (no extra structures)

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.

Ans:-

Algorithm (Boyer-Moore Voting Algorithm)

Steps:

1. Initialize:
   - candidate = None
   - count = 0
2. Traverse the array:
   - If count == 0, set candidate = current element
   - If current == candidate, increment count
   - Else, decrement count
3. At the end, candidate will be the **majority element**

Code:-

```
public class MajorityElementFinder {

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

      for (int num : nums) {
         if (count == 0) {
            candidate = num;
         }

         if (num == candidate) {
            count++;
         } else {
            count--;
         }
      }

      return candidate;
   }

   public static void main(String[] args) {
      int[] nums = {2, 2, 1, 1, 2, 2, 2};
      int result = majorityElement(nums);
      System.out.println("Majority element is: " + result);
   }
}
```

Input:[2, 2, 1, 1, 2, 2, 2]
Voting Process:

- 2 (count = 1)
- 2 (count = 2)
- 1 (count = 1)
- 1 (count = 0)
- 2 (count = 1, new candidate = 2)
- 2 (count = 2)
- 2 (count = 3)

Output: Majority element is: 2

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(1) (constant)

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.

Ans:-

Algorithm (Two-Pointer Technique – Optimal)

Steps:

```
1. Initialize:
    o Two pointers: left = 0, right = n - 1
    o Two variables: leftMax = 0, rightMax = 0
    o Result: water = 0
2. Loop while left < right:
    o If height[left] < height[right]:
        ▪ If height[left] >= leftMax: update leftMax
        ▪ Else: add leftMax - height[left] to water
        ▪ Move left++
    o Else:
        ▪ If height[right] >= rightMax: update rightMax
        ▪ Else: add rightMax - height[right] to water
        ▪ Move right--
```

Code:-

```
public class TrappingRainWater {

  public static int trap(int[] height) {
    int left = 0, right = height.length - 1;
    int leftMax = 0, rightMax = 0, water = 0;
```

```
        while (left < right) {
            if (height[left] < height[right]) {
                if (height[left] >= leftMax) {
                    leftMax = height[left];
                } else {
                    water += leftMax - height[left];
                }
                left++;
            } else {
                if (height[right] >= rightMax) {
                    rightMax = height[right];
                } else {
                    water += rightMax - height[right];
                }
                right--;
            }
        }

        return water;
    }

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

Input:[0,1,0,2,1,0,1,3,2,1,2,1]

Time and Space Complexity

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.

Ans:-

Algorithm (Using Trie – Optimal)

Steps:

1. **Insert all numbers** into a binary Trie (each number as a 32-bit binary).

2. For each number:
   - Traverse the Trie trying to take **opposite bits** at each position (to maximize XOR).
   - Calculate the XOR value while traversing.
3. Track the **maximum XOR value** found.

Code:-

```
import java.util.*;

public class MaxXORFinder {

  // Trie Node
  static class TrieNode {
    TrieNode[] children = new TrieNode[2]; // 0 or 1
  }

  public static int findMaximumXOR(int[] nums) {
    TrieNode root = new TrieNode();

    // Step 1: Build Trie
    for (int num : nums) {
      TrieNode node = root;
      for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        if (node.children[bit] == null) {
          node.children[bit] = new TrieNode();
        }
        node = node.children[bit];
      }
    }

    int maxXOR = 0;

    // Step 2: For each number, find max XOR pair
    for (int num : nums) {
      TrieNode node = root;
      int currXOR = 0;

      for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        int oppositeBit = 1 - bit;
```

```
            if (node.children[oppositeBit] != null) {
                currXOR |= (1 << i);
                node = node.children[oppositeBit];
            } else {
                node = node.children[bit];
            }
        }

        maxXOR = Math.max(maxXOR, currXOR);
    }

    return maxXOR;
}

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

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

Output:Maximum XOR is: 28

Time and Space Complexity

Time Complexity O(n * 32) = O(n)

Space Complexity O(n * 32) = O(n)

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

Ans:-

Algorithm (Dynamic Programming Approach)

Steps:

1. Initialize:
    - maxSoFar = nums[0]
    - maxEndingHere = nums[0]
    - minEndingHere = nums[0]
2. Traverse nums from index 1 to end:

    - Temporarily store maxEndingHere
    - Update maxEndingHere and minEndingHere using the current number
    - Update maxSoFar with the current maxEndingHere
3. Return maxSoFar

Code:-

```
public class MaxProductSubarray {

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

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

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

         int tempMax = maxEndingHere;

         maxEndingHere = Math.max(current, Math.max(current * maxEndingHere, current *
minEndingHere));
         minEndingHere = Math.min(current, Math.min(current * tempMax, current *
minEndingHere));

         maxSoFar = Math.max(maxSoFar, maxEndingHere);
      }

      return maxSoFar;
   }

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

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

Steps:

- Start: max = 2, min = 2, result = 2

- At 3:
  → max = max(3, 2×3, 2×3) = 6
  → min = min(3, 2×3, 2×3) = 3
- At -2:
  → max = max(-2, 6×-2, 3×-2) = -2
  → min = min(-2, 6×-2, 3×-2) = -12
- At 4:
  → max = max(4, -2×4, -12×4) = 4
  → min = min(4, -2×4, -12×4) = -48

Final result = **6**
Output: Maximum product subarray is: 6

Time and Space Complexity

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

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.

Ans:-

Algorithm (Combinatorics-Based)

Steps:

1. If n == 0, return 1 (only 0)
2. Initialize:
   - result = 10 (for n = 1)
3. Loop from i = 2 to n:
   - Initialize count = 9
   - Multiply by decreasing digits from 9 → (10 - i + 1)
   - Add to result
4. Return result

Code:-

```
public class UniqueDigitNumbers {
    public static int countNumbersWithUniqueDigits(int n) {
        if (n == 0) return 1;
        int result = 10; // for n = 1
        int uniqueDigits = 9;
        int availableDigits = 9;
        for (int i = 2; i <= n && availableDigits > 0; i++) {
            uniqueDigits *= availableDigits;
            result += uniqueDigits;
            availableDigits--;
        }
        return result;
    }

    public static void main(String[] args) {
        int n = 2;
        System.out.println("Count of unique-digit numbers for n = " + n + " is: " +
countNumbersWithUniqueDigits(n));
    }
}
```

Input: n = 2

Output: 10 (1-digit) + 81 (2-digit) = 91

Time and Space Complexity:-

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.

Ans:- **Algorithm (DP Based on Bit Shifts)**

1. Create an array ans[] of size n+1
2. Set ans[0] = 0
3. For each i from 1 to n:
    - ans[i] = ans[i >> 1] + (i & 1)
4. Return the ans[] array

Code:-

```java
public class CountBits {
  public static int[] countBits(int n) {
    int[] ans = new int[n + 1];
    ans[0] = 0;
    for (int i = 1; i <= n; i++) {
      ans[i] = ans[i >> 1] + (i & 1);
    }

    return ans;
  }
  public static void main(String[] args) {
    int n = 5;
    int[] result = countBits(n);
    System.out.print("Number of 1s from 0 to " + n + ": ");
    for (int i = 0; i <= n; i++) {
      System.out.print(result[i] + " ");
    }
    }
}
```

Input: n = 5

Output: [0, 1, 1, 2, 1, 2]

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(n) (for output array)

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.

Ans:-

Algorithm (Using Bit Manipulation)

Steps:

1. If n <= 0, return false (negative numbers and zero can't be powers of two).
2. If n > 0, check the condition: (n & (n - 1)) == 0
3. Return true if the condition holds, else false.

Code:-

```
public class PowerOfTwo {
    public static boolean isPowerOfTwo(int n) {
        // Return false if n is less than or equal to 0
        if (n <= 0) {
            return false;
        }
        // Check if n is a power of two
        return (n & (n - 1)) == 0;
    }
    public static void main(String[] args) {
        int num = 16;
        System.out.println(num + " is a power of two: " + isPowerOfTwo(num));


        num = 18;
        System.out.println(num + " is a power of two: " + isPowerOfTwo(num));
    }
}
```

Input: num = 16

Output: 16 is a power of two: true

Time and Space Complexity

Time Complexity O(1)

Space Complexity O(1)

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**.**

Ans:-

Algorithm (Trie-Based Approach)

Steps to Solve:

1. **Build a Trie where each node has two children, representing 0 and 1.**
2. **For each number in the array:**
    - **Convert it to a binary string of a fixed length (e.g., 32 bits for 32-bit integers).**
    - **Try to maximize the XOR by choosing the opposite bit of the current number (if available).**
    - **Update the maximum XOR with the result.**
3. **Return the maximum XOR.**

Code:-

```java
public class MaximumXOR {

    // Trie Node
    static class TrieNode {
        TrieNode[] children = new TrieNode[2]; // 0 or 1
    }
    public static int findMaximumXOR(int[] nums) {
        TrieNode root = new TrieNode();
        // Step 1: Insert all numbers into the Trie
        for (int num : nums) {
            TrieNode node = root;
            for (int i = 31; i >= 0; i--) {
                int bit = (num >> i) & 1;
                if (node.children[bit] == null) {
                    node.children[bit] = new TrieNode();
                }
                node = node.children[bit];
            }
        }
        int maxXOR = 0;
        // Step 2: For each number, find the maximum XOR pair
        for (int num : nums) {
            TrieNode node = root;
            int currXOR = 0;
            for (int i = 31; i >= 0; i--) {
                int bit = (num >> i) & 1;
                int oppositeBit = 1 - bit;
                if (node.children[oppositeBit] != null) {
```

```
                currXOR |= (1 << i);
                node = node.children[oppositeBit];
            } else {
                node = node.children[bit];
            }
        }
        maxXOR = Math.max(maxXOR, currXOR);
    }
    return maxXOR;
  }
  public static void main(String[] args) {
    int[] nums = {3, 10, 5, 25, 2, 8};
    System.out.println("Maximum XOR is: " + findMaximumXOR(nums));
  }
}
```

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

Output: Maximum XOR is: 28

Time and Space Complexity

Time Complexity O(n * 32) = O(n)

Space Complexity O(n * 32) = O(n)

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

Ans:-

Bit manipulation refers to the act of algorithmically manipulating bits (0s and 1s) at the binary level using **bitwise operators**.

Every integer in a computer is stored as a sequence of bits. Bit manipulation lets you perform operations directly on those bits, often leading to **faster and memory-efficient code**.

Advantages of Bit Manipulation in Algorithm Design

1. Performance & Speed

- Bitwise operations are **extremely fast** — typically just **one CPU cycle**.
- Great for time-critical or low-level systems.

2. Memory Efficiency

- Helps save memory using **bit flags** or **bitsets**.
- Example: Use 1 integer (32 bits) to store 32 boolean values.

3. Simplified Logic

- Many mathematical tricks and optimizations use bits.
- Example: Checking if a number is even → n & 1 == 0

4. Useful in Specialized Problems

- Cryptography
- Compression
- Graphics
- Game development
- Networking (e.g., subnet masks)
- Competitive programming

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.

Ans:-

Algorithm (Using Stack - Efficient Approach)

Steps:

1. Create a result array res[] of size n.
2. Initialize an empty stack.
3. Traverse the array arr[] from right to left:
    - While the stack is not empty and stack's top ≤ current element:
        - Pop from the stack.

```
      o If stack is not empty:
            ▪ res[i] = stack.peek();
      o Else:
            ▪ res[i] = -1;
      o Push current element onto the stack.
4. Return the result array.
```

Code:-

```java
import java.util.*;
public class NextGreaterElement {
  public static int[] nextGreaterElements(int[] arr) {
    int n = arr.length;
    int[] result = new int[n];
    Stack<Integer> stack = new Stack<>();

    for (int i = n - 1; i >= 0; i--) {
      // Pop all smaller or equal elements
      while (!stack.isEmpty() && stack.peek() <= arr[i]) {
        stack.pop();
      }
      // If stack is not empty, top is the next greater
      result[i] = stack.isEmpty() ? -1 : stack.peek();
      // Push current element
      stack.push(arr[i]);
    }
    return result;
  }
  public static void main(String[] args) {
    int[] arr = {4, 5, 2, 10, 8};

    int[] nge = nextGreaterElements(arr);

    System.out.println("Next Greater Elements:");

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

      System.out.println(arr[i] + " → " + nge[i]);

    }

  }

}
```

Input: arr = [4, 5, 2, 10, 8]

Output: 4 → 5

5 → 10

2 → 10

10 → -1

8 → -1

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(n)

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.

Ans:- **Algorithm (Two-Pointer Approach)**

Steps:

1. Create a **dummy node** pointing to the head (to handle edge cases easily).
2. Initialize two pointers: fast and slow, both pointing to dummy.
3. Move fast n + 1 steps ahead (so slow ends up just before target).
4. Move both pointers one step at a time until fast reaches the end.
5. Now, slow.next is the node to remove. Update slow.next = slow.next.next.

6. Return dummy.next.

Code:-

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}
public class RemoveNthNode {
    public static ListNode removeNthFromEnd(ListNode head, int n) {
        // Create a dummy node
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode fast = dummy;
        ListNode slow = dummy;
        // Move fast ahead by n+1 steps
        for (int i = 0; i <= n; i++) {
            fast = fast.next;
        }
        // Move both pointers until fast reaches end
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }
        // Remove the nth node
        slow.next = slow.next.next;
        return dummy.next;
    }
    // Helper function to print the list
    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " → ");
            head = head.next;
        }
        System.out.println("null");
    }
    public static void main(String[] args) {
        // Example: 1 → 2 → 3 → 4 → 5, n = 2
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);
        System.out.print("Original List: ");
        printList(head);
        int n = 2;
        head = removeNthFromEnd(head, n);
        System.out.print("After Removing " + n + "-th Node From End: ");
        printList(head);
    }
}
```

Input: List = 1 → 2 → 3 → 4 → 5, n = 2

Output: List = 1 → 2 → 3 → 5

Time and Space Complexity

Time Complexity O(L)

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.

Ans:-

Algorithm (Two-Pointer Approach)

Steps:

```
1. Initialize two pointers a and b at headA and headB.
2. While a ≠ b:
      o  If a == null, move a to headB; else move to a.next
      o  If b == null, move b to headA; else move to b.next
3. When a == b, return a (which is either intersection or null)
```

Code:-

```
class ListNode {
   int val;
   ListNode next;
   ListNode(int x) {
      val = x;
      next = null;
   }
}
public class IntersectionNode {
   public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
      ListNode a = headA;
      ListNode b = headB;
      // Traverse both lists
```

```
    while (a != b) {
        a = (a == null) ? headB : a.next;
        b = (b == null) ? headA : b.next;
    }
    return a; // Either intersection node or null
}
// Helper to print a list from a given node
public static void printFromNode(ListNode node) {
    while (node != null) {
        System.out.print(node.val + " → ");
        node = node.next;
    }
    System.out.println("null");
}
public static void main(String[] args) {
    // Shared nodes
    ListNode intersect = new ListNode(8);
    intersect.next = new ListNode(10);
    // First list: 3 → 7 → 8 → 10
    ListNode headA = new ListNode(3);
    headA.next = new ListNode(7);
    headA.next.next = intersect;
    // Second list: 99 → 1 → 8 → 10
    ListNode headB = new ListNode(99);
    headB.next = new ListNode(1);
    headB.next.next = intersect;
    ListNode result = getIntersectionNode(headA, headB);
    System.out.print("Intersection Node: ");
```

```
        System.out.println(result != null ? result.val : "null");
    }
}
```

Input:

List A = 3 → 7 → 8 → 10

List B = 99 → 1 → 8 → 10

Output: Intersection Node: 8

Time and Space Complexity

Time Complexity O(m + n) — m and n are lengths of the two lists

Space Complexity O(1) — constant space, no extra structures used

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

Ans:-

Algorithm Steps:-

1. Initialize:
   - top1 = -1 → for Stack 1 (starts from beginning).
   - top2 = n → for Stack 2 (starts from end).
2. For **push1(x)**:
   - Check if top1 < top2 - 1, then insert at ++top1.
3. For **push2(x)**:
   - Check if top1 < top2 - 1, then insert at --top2.
4. For **pop1()**:
   - If top1 >= 0, return arr[top1--].
5. For **pop2()**:
   - If top2 < size, return arr[top2++].

Code:-

```
public class TwoStacks {
```

```
int[] arr;

int size;

int top1, top2;


// Constructor

public TwoStacks(int n) {

    size = n;

    arr = new int[n];

    top1 = -1;

    top2 = n;

}

// Push to Stack 1

public void push1(int x) {

    if (top1 < top2 - 1) {

        arr[++top1] = x;

    } else {

        System.out.println("Stack Overflow in Stack 1");

    }

}

// Push to Stack 2

public void push2(int x) {

    if (top1 < top2 - 1) {

        arr[--top2] = x;

    } else {

        System.out.println("Stack Overflow in Stack 2");

    }

}
```

```
// Pop from Stack 1
public int pop1() {
    if (top1 >= 0) {
        return arr[top1--];
    } else {
        System.out.println("Stack Underflow in Stack 1");
        return -1;
    }
}
// Pop from Stack 2
public int pop2() {
    if (top2 < size) {
        return arr[top2++];
    } else {
        System.out.println("Stack Underflow in Stack 2");
        return -1;
    }
}
// Main to test
public static void main(String[] args) {
    TwoStacks ts = new TwoStacks(10);


    ts.push1(1);
    ts.push1(2);
    ts.push1(3);
    ts.push2(9);
    ts.push2(8);
    ts.push2(7);
```

```
        System.out.println("Popped from Stack 1: " + ts.pop1()); // 3
        System.out.println("Popped from Stack 2: " + ts.pop2()); // 7
    }
}
```

Time and Space Complexity :- 0(1) and 0(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.

Ans:-

Algorithm Steps

1. **Edge Cases**:
   - If x < 0: Not a palindrome (negative sign makes it invalid).
   - If x != 0 and ends with 0: Not a palindrome (10, 100 etc.).
2. **Reverse second half**:
   - While x > reversedHalf:
     - reversedHalf = reversedHalf * 10 + x % 10
     - x = x / 10
3. **Check Palindrome**:
   - If even digits: x == reversedHalf
   - If odd digits: x == reversedHalf / 10 (middle digit doesn't affect)

Code:-

```
public class PalindromeNumber {
    public static boolean isPalindrome(int x) {
        // Special cases
        if (x < 0 || (x != 0 && x % 10 == 0)) return false;
        int reversedHalf = 0;
        while (x > reversedHalf) {
            int digit = x % 10;

            reversedHalf = reversedHalf * 10 + digit;

            x = x / 10;

        }

        // For even or odd number of digits

        return (x == reversedHalf) || (x == reversedHalf / 10);

    }

    public static void main(String[] args) {

        int number = 1221;

        if (isPalindrome(number)) {

            System.out.println(number + " is a Palindrome.");

        } else {

            System.out.println(number + " is NOT a Palindrome.");

        }

    }

}
```

Input:x = 1221

Output: 1221 is a Palindrome.

Time and Space Complexity

Time Complexity O($\log_{10}$(n))

Space Complexity O(1)

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

Ans:- A **linked list** is a **linear data structure** where elements (called **nodes**) are **not stored at contiguous memory locations**. Each node contains:

1. **Data**
2. A **reference (or pointer)** to the **next** node in the sequence

There are different types of linked lists:

- **Singly Linked List**: Each node points to the next node only
- **Doubly Linked List**: Each node points to **both** next and previous nodes
- **Circular Linked List**: Last node connects back to the head (can be singly or doubly)

Applications in Algorithm Design

Linked lists are **flexible** and **powerful** in many scenarios:

1. Dynamic Memory Usage

- Unlike arrays, linked lists don't require pre-allocated size.
- Useful when the size of the structure is **unknown or changes frequently**.

2. Efficient Insertions/Deletions

- Insert/delete in O(1) time at head or tail (if pointer maintained).
- Much faster than arrays which require shifting elements.

3. Implementing Other Data Structures

- Used in:
    - **Stacks** and **Queues**
    - **Graphs** (adjacency lists)
    - **Hash Tables** (chaining for collision resolution)
    - **LRU Cache** (using doubly linked list)

4. Memory-efficient Variants

- Useful in **low-memory systems**, especially with pointer optimization.

5. Real-Time Systems

- No need to shift elements like in arrays = consistent performance.

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.

Ans:-**Algorithm Steps**

1. Initialize a deque and a result list.
2. Iterate over the array:
    - Remove indices from front if they are out of the window.
    - Remove indices from back while the current element is greater than elements at those indices.
    - Add current index to deque.
    - If i >= k - 1, add nums[deque.front] to result (current max).
3. Return the result.

Code:-

```
import java.util.*;

public class SlidingWindowMax {

    public static int[] maxSlidingWindow(int[] nums, int k) {

        if (nums.length == 0 || k == 0) return new int[0];

        int n = nums.length;

        int[] result = new int[n - k + 1];

        Deque<Integer> deque = new ArrayDeque<>();

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

            // Remove elements out of the current window

            if (!deque.isEmpty() && deque.peekFirst() <= i - k) {

                deque.pollFirst();

            }

            // Remove smaller elements in k range as they are useless

            while (!deque.isEmpty() && nums[i] >= nums[deque.peekLast()]) {

                deque.pollLast();

            }

            // Add current element's index at the back

            deque.offerLast(i);
            // Add current max to result once we reach window size
            if (i >= k - 1) {
                result[i - k + 1] = nums[deque.peekFirst()];
            }
        }
        return result;
    }
    public static void main(String[] args) {
        int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        int[] result = maxSlidingWindow(nums, k);
        System.out.println("Sliding Window Maximums:");
        for (int max : result) {
            System.out.print(max + " ");
        }
    }
}
```

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

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

Time and Space Complexity

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.

Ans:- **Algorithm Steps**

1. Initialize a stack and a variable to keep track of max area.

2. Traverse the array:
    - While the stack is **not empty** and current bar < bar at stack top:
        - Pop the stack, calculate area using popped height as the shortest.
        - Width = current index - index at new stack top - 1
    - Push current index to stack.
3. After iteration, process any remaining bars in the stack.
4. Return max area.

Code:-

```
import java.util.Stack;

public class LargestRectangleHistogram {

  public static int largestRectangleArea(int[] heights) {

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

    int maxArea = 0;

    int n = heights.length;

    // Append a 0 at the end to flush the stack

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

      int currentHeight = (i == n) ? 0 : heights[i];

      while (!stack.isEmpty() && currentHeight < heights[stack.peek()]) {

        int height = heights[stack.pop()];

        int width;

        if (stack.isEmpty()) {

          width = i;

        } else {

          width = i - stack.peek() - 1;

        }

        maxArea = Math.max(maxArea, height * width);

      }
```

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

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

Output: Largest Rectangle Area: 10

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(n)

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

Ans:-

The **Sliding Window** technique is used to **reduce the time complexity** of problems involving **linear data structures** (like arrays, lists, strings). It works by creating a **window** (subrange) that slides across the data structure to examine or compute something within that range.

Applications in Array Problems:-

1. Maximum Sum of Subarray of Size K (Fixed window)

Input: [2, 1, 5, 1, 3, 2], K = 3 → Output: 9 (5+1+3)

2. Minimum Size Subarray Sum (Variable window)

Input: [2,3,1,2,4,3], Sum ≥ 7 → Output: 2 ([4,3])

3. Longest Subarray with at Most K Distinct Numbers

Input: [1,2,1,2,3], K = 2 → Output: 4 ([1,2,1,2])

4. Maximum in Every Sliding Window of Size K

Input: [1,3,-1,-3,5,3,6,7], K = 3 → Output: [3,3,5,5,6,7]

5. Check for Anagrams in a String (Using frequency map + sliding window)

ementally.

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.

Ans:-

Algorithm Steps:-

1. Initialize a HashMap to store {prefixSum: count}. Start with {0:1}.
2. Initialize:
   - count = 0 (to count subarrays)
   - sum = 0 (to store running prefix sum)
3. Loop through the array:
   - Add current element to sum
   - If sum - k exists in map, increment count by the frequency of (sum - k)
   - Add/update sum in map
4. Return count.

Code:-

```
import java.util.HashMap;

public class SubarraySumEqualsK {


   public static int subarraySum(int[] nums, int k) {

       HashMap<Integer, Integer> map = new HashMap<>();

       map.put(0, 1); // prefix sum 0 has occurred once
```

```
        int count = 0, sum = 0;

        for (int num : nums) {

            sum += num;

            // Check if there's a prefix sum such that sum - prefix = k

            if (map.containsKey(sum - k)) {

                count += map.get(sum - k);

            }

            // Add current sum to map

            map.put(sum, map.getOrDefault(sum, 0) + 1);

        }

        return count;

    }

    public static void main(String[] args) {

        int[] nums = {1, 1, 1};

        int k = 2;

        int result = subarraySum(nums, k);

        System.out.println("Number of subarrays with sum = " + k + ": " + result);

    }

}
```

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

Subarrays:

- [1, 1] → sum = 2
- [1, 1] → sum = 2
- Total = 2 subarrays

Output: Number of subarrays with sum = 2: 2

Time and Space Complexity

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.

Ans:-

Algorithm Steps:-

1. Traverse the array to build a frequency map {element: count}.
2. Create a min-heap that stores entries (element, frequency), sorted by frequency.
3. For each entry in the map:
   - Add it to the heap.
   - If heap size > k, remove the element with the lowest frequency.
4. Extract elements from the heap and return them.

```
import java.util.*;
public class KMostFrequent {
  public static int[] topKFrequent(int[] nums, int k) {
    // Step 1: Count frequencies
    Map<Integer, Integer> freqMap = new HashMap<>();
    for (int num : nums)
      freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
    // Step 2: Min-heap based on frequency
    PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
        new PriorityQueue<>(Comparator.comparingInt(Map.Entry::getValue));

    // Step 3: Maintain top k in heap
    for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {
      minHeap.offer(entry);
      if (minHeap.size() > k)
```

```
            minHeap.poll();
        }
        // Step 4: Extract result
        int[] result = new int[k];
        for (int i = k - 1; i >= 0; i--) {
            result[i] = minHeap.poll().getKey();
        }
        return result;
    }
    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] topK = topKFrequent(nums, k);
        System.out.println("Top " + k + " frequent elements: " + Arrays.toString(topK));
    }
}
```

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

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

Time and Space Complexity

Time Complexity O(n log k)

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.

Ans:-

Algorithm (Recursive Backtracking)

Steps:

1. Use a helper function that takes:
    - The current index
    - A temporary list (current subset)
    - A result list (to store all subsets)
2. At each index:
    - Include the element and recurse
    - Exclude the element and recurse
3. When the index reaches the end, add the current subset to the result.

Code:-

```
import java.util.*;

public class SubsetGenerator {

  public static List<List<Integer>> subsets(int[] nums) {

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

    generateSubsets(0, nums, new ArrayList<>(), result);

    return result;

  }

  private static void generateSubsets(int index, int[] nums, List<Integer> current,
List<List<Integer>> result) {

    if (index == nums.length) {

      result.add(new ArrayList<>(current));

      return;

    }


    // Include the current element

    current.add(nums[index]);

    generateSubsets(index + 1, nums, current, result);

    // Backtrack and exclude the current element
```

```
        current.remove(current.size() - 1);
        generateSubsets(index + 1, nums, current, result);
    }
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        List<List<Integer>> allSubsets = subsets(nums);
        System.out.println("All subsets:");
        for (List<Integer> subset : allSubsets) {
            System.out.println(subset);
        }
    }
}
```

Input: nums = [1, 2]

```
Output (Power Set):[]
[1]
[2]
[1, 2]
```

Time and Space Complexity

Time Complexity O(2^n * n) — 2^n subsets, each takes up to n time to copy

Space Complexity O(2^n * n) — for storing all subsets

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**.**

Ans:-

Algorithm Steps:-

1. Sort the array (optional for optimization).
2. Define a recursive function with:
    - current index

   - current combination list
   - remaining target
3. If target = 0, add combination to result.
4. If target < 0 or index is out of bounds, return.
5. Loop through candidates starting from index:
   - Include candidate[i] in the current combination
   - Recurse with updated target and same index (since reuse is allowed)
   - Backtrack (remove last added element)

Code:-

```
import java.util.*;

public class CombinationSum {

    public static List<List<Integer>> combinationSum(int[] candidates, int target) {

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

        backtrack(0, candidates, target, new ArrayList<>(), result);

        return result;

    }

    private static void backtrack(int index, int[] candidates, int target, List<Integer> current,
List<List<Integer>> result) {

        if (target == 0) {

            result.add(new ArrayList<>(current));

            return;

        }

        if (target < 0 || index >= candidates.length) {

            return;

        }

        // Include current candidate

        current.add(candidates[index]);
        backtrack(index, candidates, target - candidates[index], current, result);
        // Exclude current candidate
        current.remove(current.size() - 1);
        backtrack(index + 1, candidates, target, current, result);
    }
    public static void main(String[] args) {
        int[] candidates = {2, 3, 6, 7};
        int target = 7;
        List<List<Integer>> combinations = combinationSum(candidates, target);
        System.out.println("Combinations that sum to " + target + ":");
        for (List<Integer> combo : combinations) {
            System.out.println(combo);
        }
    }
}
```

Input: candidates = [2, 3, 6, 7], target = 7

Output:[2, 2, 3]

[7]

Time and Space Complexity

Time Complexity O(2^t)

Space Complexity O(k)

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

Ans:-

Algorithm Steps:-

1. Define a recursive function permute(index, arr):
    - If index == arr.length, add the current permutation to the result.

2. For each i from index to arr.length - 1:
    - Swap arr[index] and arr[i]
    - Recurse with index + 1
    - Backtrack: swap back to restore original state

Code:-

```
import java.util.*;
public class Permutations {
  public static List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(0, nums, result);
    return result;
  }
  private static void backtrack(int index, int[] nums, List<List<Integer>> result) {
    if (index == nums.length) {
      // Convert array to list and add to result
      List<Integer> perm = new ArrayList<>();
      for (int num : nums) perm.add(num);
      result.add(perm);
      return;
    }
    for (int i = index; i < nums.length; i++) {
      swap(nums, index, i);           // Choose
      backtrack(index + 1, nums, result); // Explore
      swap(nums, index, i);           // Un-choose (backtrack)
    }
  }
  private static void swap(int[] nums, int i, int j) {
        int temp = nums[i]; nums[i] = nums[j]; nums[j] = temp;
    }
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        List<List<Integer>> allPermutations = permute(nums);
        System.out.println("All permutations:");
        for (List<Integer> perm : allPermutations) {
            System.out.println(perm);
        }
    }
}
```

Input: nums = [1, 2, 3]

Output: [1, 2, 3]

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

Time and Space Complexity

Time Complexity O(n * n!)

Space Complexity O(n)

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

Ans:-

A **subset** is any combination of elements (including the empty set and the set itself) where **order does not matter**, and each element is either **included or excluded**.

A **permutation** is a rearrangement of elements where **order matters**, and each element is used **exactly once per arrangement**.

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.

Ans:-

Algorithm Steps:-

1. Initialize a HashMap<Integer, Integer> to store {element → frequency}.
2. Loop through the array and populate the frequency map.
3. Track the element with the **highest frequency** and its count.
4. Return the element with the maximum frequency.

Code:-

```
import java.util.*;

public class MaxFrequencyElement {

  public static int findMaxFrequencyElement(int[] nums) {

    // Step 1: Count frequency using HashMap

    Map<Integer, Integer> freqMap = new HashMap<>();

    for (int num : nums) {

      freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);

    }

    // Step 2: Find element with maximum frequency

    int maxFreq = 0;

    int maxElem = nums[0]; // default to first element

    for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {

      if (entry.getValue() > maxFreq) {

        maxFreq = entry.getValue();

        maxElem = entry.getKey();

      }

    }

    return maxElem;
    }
    public static void main(String[] args) {
        int[] nums = {1, 3, 2, 3, 4, 3, 5};
        int maxFreqElement = findMaxFrequencyElement(nums);
        System.out.println("Element with maximum frequency: " + maxFreqElement);
    }
}
```

Input:[1, 3, 2, 3, 4, 3, 5]

Output: Element with maximum frequency: 3

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(n)

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

Ans:-

Kadane's Algorithm:

- **Traverse the array** from left to right.
- At each step, decide whether to:
  - Extend the **current subarray** by including the current element.
  - Start a **new subarray** from the current element.
- Keep track of:
  - currentSum: max sum ending at current index
  - maxSum: overall maximum found so far

Code:-

```
public class KadanesAlgorithm {
    public static int maxSubarraySum(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];

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

            // Decide to extend or start new subarray

            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};

        int maxSum = maxSubarraySum(nums);

        System.out.println("Maximum subarray sum is: " + maxSum);

    }

}
```

Input: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Output: Maximum subarray sum is: 6

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(1)

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

Ans:-

Dynamic Programming is a method for solving **complex problems** by breaking them down into **simpler overlapping subproblems**, solving each just once, and storing their results for future reuse.

Kadane’s Algorithm as a DP Solution:-

The **Maximum Subarray Problem** asks for the **maximum sum** of a contiguous subarray within a one-dimensional array of numbers.
This can be **solved using Dynamic Programming** – in fact, **Kadane’s Algorithm is a form of DP**!

- dp[i] = maximum subarray sum **ending at index i**

The recurrence relation:

dp[i] = max(nums[i], dp[i-1] + nums[i])

- Start a **new subarray** at i (nums[i])
- Or, **extend the previous subarray** ending at i-1 by including nums[i]

Dynamic Programming Version of Kadane’s Algorithm in Java

Code:-

```
public class MaxSubarrayDP {

  public static int maxSubarraySum(int[] nums) {

    int n = nums.length;

    int[] dp = new int[n];

    dp[0] = nums[0];

    int maxSum = dp[0];


    for (int i = 1; i < n; i++) {

      dp[i] = Math.max(nums[i], dp[i - 1] + nums[i]);

      maxSum = Math.max(maxSum, dp[i]);

    }

    return maxSum;

  }

  public static void main(String[] args) {

    int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

    int maxSum = maxSubarraySum(nums);

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

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.

Ans:-

Algorithm Steps

1. Traverse the array and store {element → frequency} in a HashMap.
2. Use a **Min-Heap** (priority queue) of size k to keep the top k frequent elements:
   - If heap size exceeds k, remove the element with the **lowest frequency**.
3. Extract elements from the heap and return them as the result.

Code:-

```
import java.util.*;
public class TopKFrequentElements {
    public static int[] topKFrequent(int[] nums, int k) {
        // Step 1: Count frequency
        Map<Integer, Integer> freqMap = new HashMap<>();
        for (int num : nums) {
            freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
        }
        // Step 2: Min-Heap based on frequency
        PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
            new PriorityQueue<>(Comparator.comparingInt(Map.Entry::getValue));
        for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {
            minHeap.offer(entry);
            if (minHeap.size() > k) {
                minHeap.poll(); // Remove least frequent
            }
        }
        // Step 3: Build result from heap
        int[] result = new int[k];
        int i = 0;
        while (!minHeap.isEmpty()) {
            result[i++] = minHeap.poll().getKey();
        }
        return result;
    }
    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] topK = topKFrequent(nums, k);
        System.out.println("Top " + k + " frequent elements: " + Arrays.toString(topK));
    }
}
```

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

Top 2 Frequent:[1, 2]

Output: Top 2 frequent elements: [2, 1] or [1, 2]

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.

Ans:-

Algorithm:-

1. **Initialize an empty HashMap to store numbers we've seen.**
2. **Loop through the array:**

    - For each element num, check if target - num is in the map.
    - If yes, return the indices of num and target - num.
    - Otherwise, add num to the map with its index.
3. If no such pair exists, return null or an appropriate message indicating no solution.

Code:-

```
import java.util.*;
public class TwoSumHashing {
  public static int[] twoSum(int[] nums, int target) {
    // Step 1: Create a map to store numbers and their indices
    Map<Integer, Integer> numMap = new HashMap<>();
    // Step 2: Iterate through the array
    for (int i = 0; i < nums.length; i++) {
      int complement = target - nums[i];
      // Step 3: Check if complement exists in the map
      if (numMap.containsKey(complement)) {
        return new int[] { numMap.get(complement), i };
      }
      // Step 4: Add the current number and its index to the map
      numMap.put(nums[i], i);
    }

    // No solution found
    return new int[] {};
  }
  public static void main(String[] args) {
    int[] nums = { 2, 7, 11, 15 };
    int target = 9;
    int[] result = twoSum(nums, target);
```

```
        if (result.length == 0) {
            System.out.println("No solution found.");
        } else {
            System.out.println("Indices of the two numbers that add up to target: " +
Arrays.toString(result));
        }
    }
}
```

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

Output: Indices of the two numbers that add up to target: [0, 1]

Time and Space Complexity

Time Complexity O(n)

Space Complexity O(n)

45.Explain the concept of priority queues and their applications in algorithm design.

Ans:-

A **priority queue** is a **special type of queue** where each element is associated with a priority. The element with the highest priority is always dequeued first. This differs from a regular queue, where elements are dequeued in the order they were enqueued (FIFO – First In, First Out).

In a priority queue, the element with the **highest priority** can either have the **largest or smallest value**, depending on the implementation. A **max-priority queue** gives the highest priority to the largest element, while a **min-priority queue** gives priority to the smallest element.

Applications of Priority Queues in Algorithm Design

Priority queues are widely used in various algorithms, especially when handling **dynamic sets** of elements that must be processed in order of priority. Here are some key applications:

1. Dijkstra’s Shortest Path Algorithm

Problem: Find the shortest path from a source node to all other nodes in a graph.

- **How it uses Priority Queue:** A priority queue is used to select the node with the **smallest distance** (shortest known path) from the source node. It helps in efficiently selecting the next node to process and updating the distances for neighboring nodes.
- **Time Complexity:** O(E log V) where V is the number of vertices and E is the number of edges.

2. Huffman Coding

Problem: Generate the optimal encoding for a set of characters based on their frequencies (used in data compression).

- **How it uses Priority Queue:** A **min-priority queue** is used to build a binary tree where characters with lower frequencies are placed deeper in the tree. The two nodes with the smallest frequencies are repeatedly merged to build the tree.
- **Time Complexity:** O(n log n) where n is the number of characters.

3. *A Search Algorithm (Pathfinding)**

Problem: Find the shortest path in a weighted graph, considering both the cost to get to a node and an estimate of the cost to reach the goal.

- **How it uses Priority Queue:** A priority queue stores the nodes in order of **f(n) = g(n) + h(n)**, where g(n) is the cost to reach node n and h(n) is the heuristic estimate to the goal. The queue helps efficiently select the next node to expand.
- **Time Complexity:** O(E log V) in graphs.

4. Merge K Sorted Lists

Problem: Merge k sorted linked lists into one sorted list.

- **How it uses Priority Queue:** A **min-priority queue** is used to keep track of the smallest element at the head of each of the k lists. The smallest element is popped and the next element from the same list is added to the queue.
- **Time Complexity:** O(N log K) where N is the total number of elements across all lists, and K is the number of lists.

5. Top K Frequent Elements (Heap-based Approach)

Problem: Find the k most frequent elements in an array.

- **How it uses Priority Queue:** A **min-priority queue** of size k can be used to keep track of the top k frequent elements. If the frequency of a new element is higher than the smallest frequency in the heap, the smallest element is removed, and the new element is added.

- **Time Complexity:** O(n log k) where n is the number of elements and k is the number of top elements to retrieve.

6. Event Simulation (e.g., Job Scheduling)

Problem: Schedule jobs based on their priority, such as jobs in a printer queue or tasks with deadlines.

- **How it uses Priority Queue:** A **max-priority queue** can be used to always process the job with the highest priority (e.g., the shortest job first, or the job with the closest deadline).
- **Time Complexity:** Depends on the scheduling mechanism, but generally involves operations on the priority queue like insertions and deletions.

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.

Ans:-

Algorithm:

1. **Initialize**:
   - A variable start to store the starting index of the longest palindrome found.
   - A variable maxLength to store the length of the longest palindrome.
2. **Iterate through the string**:
   - For each character i, consider it as a potential center of a palindrome and expand around it.
   - Expand in two ways:
     - Odd-length palindromes (single character center).
     - Even-length palindromes (two consecutive characters as the center).
3. **Expand Around Center**:
   - For both odd and even cases, keep expanding the window outwards as long as the characters on both sides are equal.
   - Update the start and maxLength whenever a longer palindrome is found.
4. **Return** the longest palindromic substring using the start and maxLength.

Time Complexity:

- **O(n^2)**, where n is the length of the string. We expand around each character (and each pair of characters), and the expansion takes linear time in the worst case.

Space Complexity:

- **O(1)**, since we are only using a few extra variables (no additional space proportional to the input size is required).

Code:-

```
public class LongestPalindromicSubstring {
  // Helper function to expand around the center
  private static String expandAroundCenter(String s, int left, int right) {
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
      left--;
      right++;
    }
    // Return the substring that is the palindrome
    return s.substring(left + 1, right);
  }
  // Function to find the longest palindromic substring
  public static String longestPalindrome(String s) {
    if (s == null || s.length() < 1) {
      return "";
    }
    String longest = "";
    for (int i = 0; i < s.length(); i++) {
      // Odd length palindromes (single character center)
      String oddPalindrome = expandAroundCenter(s, i, i);
      // Even length palindromes (two character center)
      String evenPalindrome = expandAroundCenter(s, i, i + 1);
      // Update the longest palindrome
      if (oddPalindrome.length() > longest.length()) {
        longest = oddPalindrome;
```

```
            }
            if (evenPalindrome.length() > longest.length()) {
                longest = evenPalindrome;
            }
        }
        return longest;
    }
    public static void main(String[] args) {
        String s = "babad";
        System.out.println("Longest Palindromic Substring: " + longestPalindrome(s));
    }
}
```

47.Explain the concept of histogram problems and their applications in algorithm design.

Ans:-

Algorithm: Stack-Based Approach

1. **Maintain a stack of indices** of bars in increasing height.
2. For each bar:
    - If the current bar is taller than the bar on top of the stack, push it.
    - Otherwise, pop from the stack and calculate area for the popped bar as the **smallest bar** in a rectangle.
3. Continue until the end, then clear the stack.

Time Complexity: O(n)

Because each bar is pushed and popped at most once.

Code:-

```
import java.util.Stack;
public class Histogram {
    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]; // Add 0 at the end to flush the stack
            while (!stack.isEmpty() && h < heights[stack.peek()]) {
                int height = heights[stack.pop()];
                int width = (stack.isEmpty()) ? i : i - stack.peek() - 1;
                maxArea = Math.max(maxArea, height * width);
            }
            stack.push(i);
        }
        return maxArea;
    }
    public static void main(String[] args) {
        int[] heights = {2, 1, 5, 6, 2, 3};
        System.out.println("Largest rectangle area: " + largestRectangleArea(heights));
    }
}
```

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.

Ans:-

Algorithm

1. **Find the pivot:**
    - **Traverse from the end and find the first index i where nums[i] < nums[i+1].**

  - **If no such index exists, the array is in descending order → reverse the entire array.**
2. **Find the rightmost successor:**
   - **Traverse again from the end and find the first index j such that nums[j] > nums[i].**
3. **Swap nums[i] and nums[j].**
4. **Reverse the sub-array from i + 1 to the end to get the smallest possible sequence after the pivot.**

Time & Space Complexity

- **Time Complexity: O(n)**
- **Space Complexity: O(1) (in-place)**

Code:-

```
import java.util.Arrays;

public class NextPermutation {

    public static void nextPermutation(int[] nums) {

        int n = nums.length;

        int i = n - 2;

        // Step 1: Find the first decreasing element from the right

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

            i--;

        }


        if (i >= 0) {

            // Step 2: Find element just greater than nums[i] from the end

            int j = n - 1;

            while (nums[j] <= nums[i]) {

                j--;

            }

            // Step 3: Swap nums[i] and nums[j]
            swap(nums, i, j);
        }
        // Step 4: Reverse the suffix from i + 1 to end
        reverse(nums, i + 1, n - 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[] nums = {1, 2, 3};
        nextPermutation(nums);
        System.out.println("Next permutation: " + Arrays.toString(nums));
    }
}
```

49.How to find the intersection of two linked lists. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans:-

Algorithm (Two Pointers Approach)

Steps:

1. **Initialize two pointers:**

   - **p1 at the head of List A.**
   - **p2 at the head of List B.**
2. **Traverse both lists.**
   - **When a pointer reaches the end of a list, redirect it to the head of the other list.**
3. **If the lists intersect, the pointers will eventually meet at the intersection node.**
4. **If not, both will reach null at the same time, and the loop exits.**

Code:-

```
class ListNode {
   int val;
   ListNode next;
   ListNode(int x) {
      val = x;
      next = null;
   }
}
public class IntersectionOfLinkedLists {
   public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
      if (headA == null || headB == null) return null;

      ListNode p1 = headA;
      ListNode p2 = headB;
      // Traverse both lists
      while (p1 != p2) {
         // Move to next node or switch to other list's head
         p1 = (p1 != null) ? p1.next : headB;
         p2 = (p2 != null) ? p2.next : headA;
      }
```

```
        return p1; // either the intersection node or null
    }

    // Example usage
    public static void main(String[] args) {
        // Create shared intersection
        ListNode intersect = new ListNode(7);
        intersect.next = new ListNode(9);
        // List A
        ListNode a = new ListNode(1);
        a.next = new ListNode(3);
        a.next.next = new ListNode(5);
        a.next.next.next = intersect;
        // List B
        ListNode b = new ListNode(2);
        b.next = new ListNode(4);
        b.next.next = intersect;
        ListNode result = getIntersectionNode(a, b);
        System.out.println("Intersection at node: " + (result != null ? result.val : "No
intersection"));
    }
}
```

Time & Space Complexity

- Time Complexity: O(n + m)
    - n is the length of List A.
    - m is the length of List B.
    - Each list is traversed at most twice.
- Space Complexity: O(1)
    - No extra memory used (constant space).

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

Ans:-

Algorithm:-

Efficient Approach (Time: O(n), Space: O(1))

1. Compute the total sum of the array.
2. Initialize leftSum = 0.
3. Iterate through the array:
   - Subtract arr[i] from totalSum → this gives the right sum.
   - If leftSum == rightSum, then i is an equilibrium index.
   - Add arr[i] to leftSum.

Code:-

```
public class EquilibriumIndex {
  public static void findEquilibriumIndices(int[] arr) {
    int totalSum = 0;
    for (int num : arr) {
      totalSum += num;
    }
    int leftSum = 0;
    for (int i = 0; i < arr.length; i++) {
      totalSum -= arr[i]; // totalSum now becomes right sum for index i
      if (leftSum == totalSum) {
        System.out.println("Equilibrium index found at: " + i);
      }
      leftSum += arr[i];
    }
  }
  public static void main(String[] args) {
    int[] arr = {-7, 1, 5, 2, -4, 3, 0};
```

```
        findEquilibriumIndices(arr);
    }
}
```

Time and Space Complexity

- **Time Complexity**: O(n)
    - One pass for total sum, one pass to find the index.
- **Space Complexity**: O(1)
    - Uses constant extra space.