GULSHAN KUMAR
22SCSE1012548

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

Concept of a Prefix Sum Array:

- A prefix sum array is a derived array that stores the cumulative sum of elements from the original array. Each element at index i in the prefix sum array is the sum of all elements from index 0 to i in the original array.

Formula:

Given an array arr[] of size n, the prefix sum array prefixSum[] is defined as:
prefixSum[0] = arr[0]
prefixSum[i] = prefixSum[i-1] + arr[i] for i = 1 to n-1

Example:
Let arr = [2, 4, 6, 8, 10]
The prefix sum array will be:

- prefixSum[0] = 2
- prefixSum[1] = 2 + 4 = 6
- prefixSum[2] = 6 + 6 = 12
- prefixSum[3] = 12 + 8 = 20
- prefixSum[4] = 20 + 10 = 30

So, prefixSum = [2, 6, 12, 20, 30]

Applications of Prefix Sum Arrays:

1. Efficient Range Sum Queries
   You can quickly calculate the sum of elements between two indices l and r:
   - Sum(l, r) = prefixSum[r] - prefixSum[l-1] (if l > 0)
   - Sum(0, r) = prefixSum[r] (if l == 0)

2. Subarray Problem Solving
   Useful in problems like:
   - Finding subarrays with a specific sum
   - Maximum subarray sum
   - Number of subarrays with sum divisible by k

3. 2D Prefix Sums
   — Used in matrices to efficiently calculate the sum of elements in a sub-matrix.

4. Image Processing
   — For fast filtering and averaging pixel values in a region.

5. Competitive Programming

— Speeds up time complexity from O(n) per query to O(1), after O(n) preprocessing.

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.

Algorithm

1. Take an array arr[] of size n.
2. Construct the prefixSum[] array such that:
   — prefixSum[0] = arr[0]
   — prefixSum[i] = prefixSum[i - 1] + arr[i] for i > 0
3. To find the sum from index L to R:
   - If L == 0: result = prefixSum[R]
   - Else: result = prefixSum[R] - prefixSum[L - 1]

Code

```
import java.util.Scanner; public class
PrefixSumRange { public static int[]
buildPrefixSum(int[] arr) {
    int n = arr.length; int[]
    prefixSum = new int[n];
    prefixSum[0] = arr[0];

    for (int i = 1; i < n; i++) {
      prefixSum[i] = prefixSum[i - 1] +
      arr[i];
    }

    return prefixSum;
  }
    public static int rangeSum(int[] prefixSum, int L, int R) {
      if (L == 0) return prefixSum[R];
      return prefixSum[R] - prefixSum[L -
      1];
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of elements:
    "); int n = sc.nextInt(); int[] arr = new int[n];

    System.out.println("Enter array elements:");
    for (int i = 0; i < n; i++) {
      arr[i] = sc.nextInt();
    }
    int[] prefixSum = buildPrefixSum(arr);
```

```
    System.out.print("Enter L and R (0-based indices): ");
    int L = sc.nextInt();
    int R = sc.nextInt();

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

Time Complexity

- Preprocessing (prefix sum build): O(n)
- Querying sum in range [L, R]: O(1)
- Total (1 query): O(n + 1) = O(n)

Space Complexity

- O(n) — for prefix sum array.

Example

Input:
Array: [3, 5, 2, 8, 6]
Range: L = 1, R = 3

Step-by-step:
Prefix sum array: [3, 8, 10, 18, 24]
Sum from index 1 to 3 = prefixSum[3] - prefixSum[0] = 18 - 3 = 15

Output:
Sum from index 1 to 3 is: 15

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.

What is an Equilibrium Index?
An equilibrium index in an array is an index i such that the sum of elements before it is equal to the sum of elements after it.

Formally, for index i:
arr[0] + arr[1] + ... + arr[i-1] == arr[i+1] + arr[i+2] + ... + arr[n-1]

Algorithm

1. Calculate the total sum of the array.
2. Initialize a variable leftSum = 0.
3. Loop through each element:
    - For index i, the right sum will be: totalSum - leftSum - arr[i]
    - If leftSum == rightSum, return i as equilibrium index.

- Else, add arr[i] to leftSum.

4. If no such index is found, return -1.

Code

```
import java.util.Scanner;
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) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of elements:
    "); int n = sc.nextInt(); int[] arr = new int[n];

    System.out.println("Enter array elements:");
    for (int i = 0; i < n; i++) {
      arr[i] = sc.nextInt();
    }

    int eqIndex = findEquilibriumIndex(arr);
    if (eqIndex != -1) {
      System.out.println("Equilibrium Index found at position: " + eqIndex);
    } else {
      System.out.println("No Equilibrium Index found.");
    }
  }
}
```

Time Complexity: O(n)

Space Complexity: O(1)

Example

- Input:

  Array: [1, 3, 5, 2, 2]

—Step-by-Step:

1. Total Sum = 1 + 3 + 5 + 2 + 2 = 13

2. Iterate:

- i = 0: left = 0, right = 13 - 0 - 1 = 12 → not equal
- i = 1: left = 1, right = 13 - 1 - 3 = 9 → not equal
- i = 2: left = 4, right = 13 - 4 - 5 = 4 → ✅ match found at index 2

—Output:

Equilibrium Index found at position: 2

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.

Given an array, check if it can be split into two parts such that:

sum of the prefix == sum of the suffix

Note: The two parts must not overlap, and we should find an index i such that:

arr[0] + arr[1] + ... + arr[i] == arr[i+1] + arr[i+2] + ... + arr[n-1]

Algorithm

1. Calculate the total sum of the array.
2. Initialize prefixSum = 0.
3. Loop through the array:
   - Add arr[i] to prefixSum.
   - Calculate suffixSum = totalSum - prefixSum.
   - If prefixSum == suffixSum, return true (split possible).
4. If loop ends and no match is found, return false.

Program

```
import java.util.Scanner; public
class PrefixSuffixSplit {
  public static boolean canBeSplit(int[] arr) {
    int totalSum = 0;
    for (int num : arr) {
      totalSum += num;
    }

    int prefixSum = 0;
    for (int i = 0; i < arr.length - 1; i++) { // -1 to ensure non-empty suffix
      prefixSum += arr[i]; int suffixSum
      = totalSum - prefixSum; if
      (prefixSum == suffixSum) {
```

```
            return true;
          }
        }
return false;
    }
    public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      System.out.print("Enter number of elements:
      "); int n = sc.nextInt(); int[] arr = new int[n];

      System.out.println("Enter array elements:");
      for (int i = 0; i < n; i++) {
        arr[i] = sc.nextInt();
      }

      boolean result = canBeSplit(arr);
      if (result) {
        System.out.println("Array can be split into two parts with equal sum.");
      } else {
        System.out.println("Array cannot be split into two parts with equal sum.");
      }
    }
  }
```

Time Complexity: O(n)

Space Complexity: O(1)

Example

Input 1:

— Array: [2, 4, 1, 2, 1] Step-by-step:

1. Total Sum = 10
2. Iterate:
   - i=0: prefix=2, suffix=8 → not equal • i=1: prefix=6, suffix=4 → not equal • i=2: prefix=7, suffix=3 → not equal
   - i=3: prefix=9, suffix=1 → not equal
   - End of loop: + No split point found Output:

Array cannot be split into two parts with equal sum.

Input 2:

—Array: [1, 2, 3, 3]

Total Sum = 9

Try prefix:

- i=0: prefix=1, suffix=8 → no
- i=1: prefix=3, suffix=6 → no
- i=2: prefix=6, suffix=3 → ✅ match! Output:

Array can be split into two parts with equal sum.

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.

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

Algorithm (Using Sliding Window Technique)

1. Calculate the sum of the first K elements (this is the first window).
2. Slide the window:
   - Subtract the first element of the previous window.
   - Add the next element in the current window.
   - Update max sum if the new window's sum is greater.
3. Continue this process until the end of the array.

Program

```
import java.util.Scanner;
public class MaxSubarraySum
{
  public static int maxSubarraySum(int[] arr, int k) {
    int n = arr.length;
    if (k > n) {
      System.out.println("Window size K cannot be larger than array size.");
      return -1;
    }
    int maxSum = 0;
    for (int i = 0; i < k; i++) {
      maxSum += arr[i];
    }

    int currentSum = maxSum;
    for (int i = k; i < n; i++) {
      currentSum = currentSum - arr[i - k] + arr[i];
      maxSum = Math.max(maxSum, currentSum);
    }

    return maxSum;
  }
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of elements:
    "); int n = sc.nextInt(); int[] arr = new int[n];

    System.out.println("Enter array elements:");
    for (int i = 0; i < n; i++) {
      arr[i] = sc.nextInt();
    }
```

```
        System.out.print("Enter subarray size K: ");
        int k = sc.nextInt();

        int result = maxSubarraySum(arr, k);
        if (result != -1) {
System.out.println("Maximum sum of subarray of size " + k + " is: " + result);
        }
    }
}
```

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

Example Input:

—Array: [2, 1, 5, 1, 3, 2]
—K = 3 Step-by-Step:

- Initial window (2+1+5) = 8 → maxSum = 8
- Slide: remove 2, add 1 → 1+5+1 = 7 → maxSum = 8 • Slide: remove 1, add 3 → 5+1+3 = 9 → maxSum = 9 • Slide: remove 5, add 2 → 1+3+2 = 6 → maxSum = 9

Output:
Maximum sum of subarray of size 3 is: 9

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.

Given a string, find the length of the longest substring without repeating characters.

Algorithm (Sliding Window Technique)

1. Initialize a set to store characters.
2. Use two pointers: start and end to represent the sliding window.
3. Iterate over the string with the end pointer.
4. If the character at end is not in the set:
   - Add it to the set.
   - Update the max length.
5. If the character is already in the set:
   - Remove characters from the start pointer until the duplicate is removed.
6. Repeat until you reach the end of the string.

Program

```
import java.util.HashSet;
public class LongestSubstringWithoutRepeating {
  public static int lengthOfLongestSubstring(String s) {
    int maxLength = 0;
    int start = 0;
    HashSet<Character> set = new HashSet<>();
```

```
for (int end = 0; end < s.length(); end++) {
        char current = s.charAt(end);
while (set.contains(current)) {
          set.remove(s.charAt(start));
          start++;
        }

        set.add(current);
        maxLength = Math.max(maxLength, end - start + 1);
      }

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

Time Complexity: O(n)
Space Complexity: O(min(n, m))
Example
Let's take s = "abcabcbb"

- Start from index 0.
- a, b, c → unique → length = 3
- Next a → already seen → move start to skip the first a • Now substring is bca
- Continue...
- The longest substring without repeating characters is "abc" or "bca" or "cab" → length = 3

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

The Sliding Window is a technique for reducing the time complexity of algorithms involving linear data structures like arrays or strings.

Instead of using nested loops to find a solution (which can be $O(n^2)$), the sliding window uses two pointers (start and end) to create a "window" that slides across the data to keep track of a subset of elements that satisfy a certain condition.

How It Works

- You start with two pointers: usually start and end.
- The window is the range between start and end.
- You expand the end to include new elements and contract the start when the condition is violated.
- This helps in keeping track of subarrays or substrings without recomputing things over and over.

Use in String Problems
Sliding window is widely used in string problems that involve substrings, especially when constraints like "no repeating characters" or "minimum/maximum length" are involved.

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

Given a string s, return the longest substring which is a palindrome (reads the same forwards and backwards).

Algorithm (Expand Around Center - Optimal Approach)
We consider each character (and each pair of characters) as the center of a potential palindrome and expand outward while the characters on both sides are equal.

Steps:
1. Loop through each character in the string.
2. For each character:
   - Expand around the odd-length palindrome center (i, i).
   - Expand around the even-length palindrome center (i, i+1).
3. Keep track of the longest palindrome found.

Program

```
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 CC right < s.length() CC s.charAt(left) == s.charAt(right)) {
        left--;
        right++;
      }
      return right - left - 1;
    }
public static void main(String[] args) {
      String input = "babad";
      String result = longestPalindrome(input);
      System.out.println("Longest Palindromic Substring: " + result);
    }
  }
```

Time Complexity: O(n^2)
Space Complexity: O(1)

Example
Input: "babad"

- Center at 'b': expands to "bab"
- Center at 'a': expands to "aba"
- 'aba' and 'bab' are both valid. Either can be returned.
- Final answer: "bab" or "aba"

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

Algorithm (Vertical Scanning Method – Simple s Efficient)
We compare characters column by column (i.e., index by index) across all strings until a mismatch is found.

Steps:
1. If the list is empty, return "".
2. Loop through the characters of the first string.
3. For each character:
   - Compare it with the character at the same position in all other strings.
   - If any string is shorter or there's a mismatch, stop.
4. Return the substring of the first string from 0 to the last matched index.

Program

```
public class LongestCommonPrefix {
   public static String longestCommonPrefix(String[] strs)
     { if (strs == null || strs.length == 0) return "";
for (int i = 0; i < strs[0].length(); i++) {
        char currentChar = strs[0].charAt(i);
for (int j = 1; j < strs.length; j++) { if (i >= strs[j].length() ||
        strs[j].charAt(i) != currentChar) {
            return strs[0].substring(0, i);
          }
        }
      }

      return strs[0]; // All characters matched
    }
public static void main(String[] args) {
      String[] input = {"flower", "flow", "flight"};
      String result = longestCommonPrefix(input);
      System.out.println("Longest Common Prefix: " + result);
    }
  }
```

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

Example
Input: ["flower", "flow", "flight"]

Step-by-step:

- Compare char at index 0: 'f' → matches in all
- Index 1: 'l' → matches in all
- Index 2: 'o' → mismatch in "flight" (has 'i')

Output: "fl"

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

Given a string s, generate and print all possible permutations of its characters.

Algorithm (Backtracking)
1. Use a recursive function that swaps characters to create permutations.
2. Fix one character at a time and recurse for the remaining substring.
3. When the current index reaches the end of the string, print/store the permutation.
4. Backtrack by swapping characters back to their original positions.

Program

```
public class StringPermutations {
  public static void permute(String str, int left, int
    right) { if (left == right) {
    System.out.println(str); return;
    }

    for (int i = left; i <= right;
      i++) { str = swap(str, left,
      i); permute(str, left + 1,
      right);
      str = swap(str, left, i);
    }
  } private static String swap(String s, int i,
int j) { char[] charArray = s.toCharArray();
char temp = charArray[i]; charArray[i] =
charArray[j]; charArray[j] = temp;
    return String.valueOf(charArray);
  }
 public static void main(String[] args) {
    String input = "abc";
    System.out.println("All permutations of \"" + input + "\":");
    permute(input, 0, input.length() - 1);
  }
}
```

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

Example
Input: "abc"
Permutations:
1. "abc" 2.
"acb" 3.
"bac" 4.
"bca" 5.
"cab"
6. "cba"

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.

Given a sorted array of integers and a target sum, find two numbers such that they add up to the target. Return their indices (or values).

Algorithm (Two Pointers)
1. Initialize two pointers:
- left = 0 (start of array)
- right = array.length - 1 (end of array)

2. While left < right:
- Calculate the sum: sum = arr[left] + arr[right]
- If sum == target: return the pair
- If sum < target: move left pointer to the right
- If sum > target: move right pointer to the left

3. If no pair found, return something like [-1, -1]

Program

```
public class TwoSumSorted { public static int[]
  findTwoSum(int[] arr, int target) { int left = 0;
    int right = arr.length - 1;
while (left < right) { int sum =
    arr[left] + arr[right];
if (sum == target) { return new
      int[]{left, right};
      } else if (sum < target) {
        left++;
      } else { right-
        -;
      }
    }
return new int[]{-1, -1}; // no valid pair found
  }
public static void main(String[] args) {
  int[] arr = {1, 2, 3, 4, 6};
    int target = 7;
int[] result = findTwoSum(arr, target);
```

```
    if (result[0] != -1) {
      System.out.println("Pair found at indices: " + result[0] + " and " + result[1]);
      System.out.println("Values: " + arr[result[0]] + " + " + arr[result[1]] + " = " + target);
    } else {
      System.out.println("No pair found.");
    }
  }
}
```

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

Example Input:
arr = {1, 2, 3, 4, 6}
target = 7

Execution: 1 + 6 = 7 → ✅ found

Output:
Indices: 0 and 4
Values: 1 + 6 = 7

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

Given an array of numbers representing a permutation, rearrange them to the lexicographically next greater permutation.
If such an arrangement is not possible (i.e., it is the last permutation), rearrange it to the lowest possible order (i.e., sorted in ascending order).

Algorithm
1. Find the first decreasing element from the right:
   - Let i be the largest index such that nums[i] < nums[i + 1].
   - If no such i exists, the array is in descending order. Reverse the whole array.
2. Find the next larger element from the right:
   - Find index j such that nums[j] > nums[i] (smallest element > nums[i] to the right).
3. Swap nums[i] and nums[j]
4. Reverse the subarray from i + 1 to the end

Program

```
import java.util.Arrays;

public class NextPermutation {

  public static void nextPermutation(int[] nums) {
    int n = nums.length;
    int i = n - 2; while (i >= 0 CC nums[i]
    >= nums[i + 1]) { i--;
    }
```

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

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

Example
Input: [1, 2, 3]

- Step 1: Find first decreasing: 2 at index 1
- Step 2: Find number just greater than 2: 3
- Step 3: Swap 2 and 3 → [1, 3, 2]
- Step 4: Reverse from index 2 to end → [1, 3, 2]

Output: [1, 3, 2]

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.

Given two sorted singly linked lists, merge them into a single sorted linked list.

Algorithm (Two-Pointer Technique)
We use two pointers to iterate through the input lists, compare their values, and build the merged list step by step.

1. Create a dummy node to simplify the merge logic.
2. Use a pointer current starting at the dummy node.
3. While both lists are non-empty:

- Compare the values of the current nodes.
- Append the smaller node to current.next.
- Move the pointer in the list where the node was taken from.

4. Append the remaining part of the non-empty list (if any).
5. Return dummy.next as the head of the merged list.

Program

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

public class MergeSortedLists {

  public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(-1); // Dummy node
    ListNode current = dummy;
while (l1 != null CC l2 != null) { if
    (l1.val <= l2.val) {
        current.next = l1;
        l1 = l1.next;
      }       else       {
        current.next = l2;
        l2 = l2.next;
      }
      current = current.next;
    }
    current.next = (l1 != null) ? l1 : l2;

    return dummy.next;
  }
  public static void printList(ListNode head) {
    while (head != null) {
      System.out.print(head.val + " ");
      head = head.next;
    }
  }
public static void main(String[] args) {
  ListNode l1 = new ListNode(1);
  l1.next = new ListNode(3);
  l1.next.next = new ListNode(5);
  ListNode l2 = new ListNode(2);
  l2.next = new ListNode(4);
    l2.next.next = new ListNode(6);

    ListNode merged = mergeTwoLists(l1,
    l2); System.out.print("Merged List: ");
    printList(merged);
  }
```

}

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

Example Input:

- List 1: 1 → 3 → 5
- List 2: 2 → 4 → 6

Output:

- Merged List: 1 → 2 → 3 → 4 → 5 → 6

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.

Given two sorted arrays nums1 and nums2, find the median of the two sorted arrays.
You must achieve this in O(log(min(n, m))) time.

Algorithm (Binary Search on the Smaller Array)

1. Let A and B be the two arrays. Make sure A is the smaller array.
2. Use binary search on A to find the correct partition such that:
   - maxLeftA <= minRightB • maxLeftB <= minRightA
3. If partition is correct:
   - If total number of elements is even, median is:

$$\text{median} = \frac{\max(\text{maxLeftA}, \text{maxLeftB}) + \min(\text{minRightA}, \text{minRightB})}{2}$$

   - If odd, median is:

$$\text{median} = \max(\text{maxLeftA}, \text{maxLeftB})$$

Program

```
public class MedianOfSortedArrays {

   public static double findMedianSortedArrays(int[] nums1, int[] nums2) {
      if (nums1.length > nums2.length) { return
      findMedianSortedArrays(nums2, nums1);
      }

      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
               ((double)Math.max(maxLeftX, maxLeftY) + Math.min(minRightX, minRightY)) / 2;
               } else {
                  return (double)Math.max(maxLeftX, maxLeftY);
               }
            } else if (maxLeftX > minRightY) { high
               = partitionX - 1;
            } else { low = partitionX
               + 1;
            }
         }

         throw new IllegalArgumentException("Input arrays are not sorted or not valid");
      }

      public static void main(String[] args) {
         int[] nums1 = {1, 3}; int[] nums2 =
         {2};
         System.out.println("Median: " + findMedianSortedArrays(nums1, nums2));
int[] nums3 = {1, 2}; int[]
         nums4 = {3, 4};
         System.out.println("Median: " + findMedianSortedArrays(nums3, nums4));
      }
   }
```

Time Complexity: O(log(min(n, m)))
Space Complexity: O(1)
Example
Input: nums1
= [1, 3]
nums2 = [2]

- Combined array: [1, 2, 3]
- Median = 2

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.

You are given an n x n matrix where each row and column is sorted in ascending order. Find the k-th smallest element in the matrix.

Optimal Algorithm: Binary Search on Value Range
We don't merge all elements into one list (that would be $O(n^2)$).
Instead, we binary search on the range of values (matrix[0][0] to matrix[n-1][n-1]) and count how many elements are ≤ mid.

Algorithm Steps:

1. Set low = matrix[0][0] and high = matrix[n-1][n-1]
2. While low < high:
   - Set mid = (low + high) / 2
   - Count how many elements in matrix are <= mid
   - If count < k → move low = mid + 1
   - Else → move high = mid
3. When loop ends, low is the k-th smallest number

Program

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

    private static int countLessOrEqual(int[][] matrix, int target) {
        int count = 0; int n = matrix.length;
        int row = n - 1; // start from bottom-left corner 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;
        int result = kthSmallest(matrix, k);
```

```
        System.out.println("The " + k + "-th smallest element is: " + result);
    }
}
```

Time Complexity: O(n * log(max - min))
Space Complexity: O(1)

Example
Input: matrix
= [
[1, 5, 9],
  [10, 11, 13],
  [12, 13, 15]
]
k = 8
Output:
The 8th smallest element is: 13

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.

You are given an array of size n. Find the majority element, which is the element that appears more than n/2 times.
Algorithm Steps:
1. Initialize a candidate and a counter.
2. Traverse the array:
    - If the counter is 0, set the current element as the candidate.
    - If the current element is the candidate, increment the counter.
    - Else, decrement the counter.
3. The final candidate is the majority element.

Code:

```
public class MajorityElement {

    public static int findMajorityElement(int[] nums) {
        int count = 0; int
        candidate = 0; for
        (int num : nums) { if
        (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }
        count = 0; for (int
        num : nums) {
            if (num == candidate) {
                count++;
            }
        }

        if (count > nums.length / 2) {
            return candidate;
```

```
        } else { throw new IllegalArgumentException("No majority element
            found");
        }
    }

    public static void main(String[] args) { int[]
        arr = {2, 2, 1, 1, 2, 2, 2};
        System.out.println("Majority Element: " + findMajorityElement(arr));
    }
}
```

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

Example
Input: [2, 2, 1, 1, 2, 2, 2]
Length = 7 ⇒ n/2 = 3.5, so majority element must occur more than 3 times.
Step-by-step (Boyer-Moore):

- Start with count = 0
- 2 → count = 1 (candidate = 2)
- 2 → count = 2
- 1 → count = 1
- 1 → count = 0
- 2 → count = 1 (candidate = 2)
- 2 → count = 2
- 2 → count = 3

🟩 Final candidate = 2, occurs 5 times ⇒ Majority element.

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.

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Algorithm (Efficient Two-Pointer Approach)
We'll use the two-pointer technique, which is efficient and easy to implement.

1. Initialize two pointers: left and right at the ends of the array.
2. Maintain two variables: leftMax and rightMax to store the maximum height seen so far from left and right.
3. Move the pointers inward:
   - If height[left] < height[right]:
     - If height[left] >= leftMax: update leftMax ○ Else: water trapped at left = leftMax - height[left] ○ Move left forward
   - Else:
     - If height[right] >= rightMax: update rightMax ○ Else: water trapped at right = rightMax - height[right] ○ Move right backward

Program

```
public class TrappingRainWater {

    public static int trap(int[] height) { int
        left = 0, right = height.length - 1; int
        leftMax = 0, rightMax = 0; int
        trappedWater = 0;
while (left < right) { if (height[left] <
        height[right]) { if (height[left] >=
        leftMax) { leftMax = height[left];
              } else { trappedWater += leftMax -
                 height[left];
              }
              left++;
           } else { if (height[right] >=
              rightMax) { rightMax =
              height[right];
              } else { trappedWater += rightMax -
                 height[right];
              } right--
              ; }
        }

        return trappedWater;
    }

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

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

Example
Input: height =
[0,1,0,2,1,0,1,3,2,1,2,1]
Visualization:

```
      #
  #   #
  #  ###  #
 ########
```

How Water is Trapped:

- Between bars 2 and 3 → 1 unit • Between bars 3 and 7 → 6 units
- And so on...

🟩 Output: 6 + 1 + 2 + 1 + 2 = 6 units of water

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.

Given an array of integers, find the maximum XOR value of any two elements.

Algorithm Steps:

1. Convert each number to binary and insert into a trie (from MSB to LSB).
2. For each number:
   - Traverse the trie trying to pick the opposite bit at each level (to maximize XOR).
   - Calculate the XOR with the best match in the trie.
3. Track the maximum XOR encountered.

Code

```
import java.util.*;
public class MaximumXOR { static
   class TrieNode {
      TrieNode[] children = new TrieNode[2]; // 0 and 1
   }
   private static void insert(TrieNode root, int num) { TrieNode
      node = root;
      for (int i = 31; i >= 0; i--) { int bit
         = (num >> i) & 1; if
         (node.children[bit] == null) {
         node.children[bit] = new
         TrieNode();
         }
         node = node.children[bit];
      }
   }
   private static int findMaxXOR(TrieNode root, int num) {
      TrieNode node = root; int maxXor = 0;
      for (int i = 31; i >= 0; i--) { int bit = (num
      >> i) & 1; int toggledBit = 1 - bit; if
      (node.children[toggledBit] != null) {
      maxXor |= (1 << i);
            node = node.children[toggledBit];
         } else { node =
            node.children[bit];
         }
      }
      return maxXor;
   }
   public static int findMaximumXOR(int[] nums) {
      TrieNode root = new TrieNode();

      // Build Trie for (int
      num : nums) {
         insert(root, num);
      }
      int maxResult = 0; for
      (int num : nums) {
         int xor = findMaxXOR(root, num);
         maxResult = Math.max(maxResult, xor);
      }

      return maxResult;
   }
```

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

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

Example
Input: nums = [3, 10, 5, 25, 2, 8] **Binary Format:**

- 3 → 000...0011
- 10 → 000...1010
- 5 → 000...0101 • 25 → 000...11001 • **...**

Pair with Max XOR:

- 5 (0101) ^ 25 (11001) = 28
- 0101 XOR 11001 = 11100 (which is 28)

🟩 **Output:** 28

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

Given an integer array nums, find the contiguous subarray (containing at least one number) that has the largest product, and return the product.

Efficient Approach: Dynamic Programming
This problem is tricky because:

- A negative number can turn a large positive product into a negative one.
- But two negative numbers can make a positive product!

So, we keep track of both:

- maxProductSoFar
- minProductSoFar

Algorithm Steps
1. Initialize:
   - maxSoFar = nums[0]
   - currMax = nums[0]
   - currMin = nums[0]
2. Loop from index 1 to end:
   - If the current number is negative, swap currMax and currMin
   - Update currMax = max(num, currMax * num)
   - Update currMin = min(num, currMin * num)
   - Update maxSoFar = max(maxSoFar, currMax)
3. Return maxSoFar

Code

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

       int maxSoFar = nums[0]; int
       currMax = nums[0];
       int currMin = nums[0];

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

          if (num < 0) {
             // Swap currMax and currMin
             int temp = currMax; currMax
             = currMin;
             currMin = temp;
          }

          currMax = Math.max(num, currMax * num); currMin
          = Math.min(num, currMin * num);

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

       return maxSoFar;
   }

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

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

Example
nput: [2, 3, -2, 4]

Step-by-Step:

- Start with currMax = 2, currMin = 2, maxSoFar = 2
- At 3:
  - currMax = max(3, 2×3) = 6 ○ currMin = min(3, 2×3) = 3 ○ maxSoFar = 6
- At -2: (swap max and min) ○ currMax = max(-2, 3×-2) = -2 ○ currMin = min(-2, 6×-2) = -12
- At 4:
  - currMax = max(4, -2×4) = 4 ○ currMin = min(4, -12×4) = -48 •
    Final maxSoFar = 6

Output: 6 (from subarray [2, 3])

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.

Given an integer n, return the count of all numbers with unique digits, where the number of digits is less than or equal to n.

Algorithm

We'll use combinatorics to count the possibilities.

- For 1-digit numbers: 10 (0–9)
- For 2-digit numbers:
  - First digit: 9 choices (1–9)
  - Second digit: 9 choices (excluding the first digit)

So:

- Count for 2 digits = 9 × 9 = 81
- Count for 3 digits = 9 × 9 × 8 = 648
- Count for 4 digits = 9 × 9 × 8 × 7
- ...

Formula: For each k from 1 to n:

- If k = 1: 10
- Else: 9 × (9) × (8) × ... × (11 - k)

Program

```
public class CountUniqueDigits {
    public static int countNumbersWithUniqueDigits(int n) { if
        (n == 0) return 1;
        if (n > 10) n = 10; // Cannot have more than 10 unique digits (0-9)

        int total = 10; // For n = 1 int
        uniqueDigits = 9;
        int available = 9;

        for (int i = 2; i <= n; i++) {
            uniqueDigits *= available;
            total += uniqueDigits;
            available--;
        }

        return total;
    }

    public static void main(String[] args) { int
        n = 3;
        System.out.println("Count of numbers with unique digits for n = " + n + ": " +
countNumbersWithUniqueDigits(n));
    }
}
```

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

Example

- Let's calculate for n = 2
- For 1-digit numbers: 10 (0–9)
- For 2-digit numbers:
  - First digit: 9 choices (1–9)
  - Second digit: 9 choices (excluding first)
  - Total: 9 × 9 = 81

Total = 10 + 81 = 91

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.

Given an integer n, return an array of length n + 1 where the value at each index i is the number of 1s in the binary representation of i.

Algorithm Steps

1. Create an array res of size n + 1. 2. Initialize res[0] = 0 3. Loop i from 1 to n:
   - res[i] = res[i >> 1] + (i & 1)
4. Return res

Program

```
import java.util.Arrays; public class CountBits { 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("Number of 1s from 0 to " + n + ": " + Arrays.toString(result));
    }
  }
```

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

Example
Input: n = 5
Binary Representations:

- 0 → 0 → 0 one
- 1 → 1 → 1 one
- 2 → 10 → 1 one
- 3 → 11 → 2 ones

- 4 → 100 → 1 one
- 5 → 101 → 2 ones

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

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.

Check whether a given integer n is a power of two.

Algorithm

Bit Manipulation Trick Core

Idea:

A number n is a power of two if it has exactly one bit set to 1 in its binary representation.

Example:

1 → 0001 ✅

2 → 0010 ✅

4 → 0100 ✅

6 → 0110 + (has two 1s)

Bitwise Formula: n > 0 && (n & (n - 1)) == 0 **Explanation:**

- For power of 2: n has only one bit set.
- n - 1 flips all bits after that one bit.
- ANDing n and n - 1 results in 0.

Program

```
public class PowerOfTwoChecker {

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

   public static void main(String[] args) { int[]
      testCases = {1, 2, 3, 4, 5, 8, 16, 18};

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

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

Example

Let's check n = 8:

- Binary: 1000
- n - 1 = 7 → 0111
- 1000 & 0111 = 0000 ✅ → So, 8 is a power of two

Let's check n = 6:

- Binary: 0110

- n - 1 = 0101
- 0110 & 0101 = 0100 ≠ 0 + → So, 6 is not a power of two

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.

Given an array of integers nums, return the maximum result of nums[i] ^ nums[j], where 0 ≤ i, j < nums.length and i ≠ j.

Optimal Solution: Using Trie (Prefix Tree)
To compare binary representations of numbers bit by bit and try to maximize the XOR from most significant bit (MSB) to least significant bit (LSB).

Algorithm
1. Build a binary trie from the array:
   - Each node represents a binary bit (0 or 1).
   - Insert all numbers in binary form into the trie.

2. For each number:
   - Traverse the trie and at each level, try to go the opposite bit to maximize the XOR.
   - Keep track of the max XOR found.

Program

```
public class MaxXORInArray {

   static class TrieNode {
      TrieNode[] children = new TrieNode[2]; // index 0 for bit 0, index 1 for bit 1
   }
   private static void insert(TrieNode root, int num) { TrieNode
      node = root;
      for (int i = 31; i >= 0; i--) { int bit = (num
         >> i) & 1; if (node.children[bit] == null)
         { node.children[bit] = new TrieNode();
         }
         node = node.children[bit];
      }
   }
   private static int findMaxXOR(TrieNode root, int num) {
      TrieNode node = root; int
      maxXOR = 0;
      for (int i = 31; i >= 0; i--) {
         int bit = (num >> i) & 1;
         int toggle = 1 - bit;
         if (node.children[toggle] != null) { maxXOR
            |= (1 << i); // set the bit in result node =
            node.children[toggle];
         }     else     {     node     =
            node.children[bit];
         }
      }
      return maxXOR;
   }
```

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

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

        return maxResult;
    }

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

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

Example
Input: [3, 10, 5, 25, 2, 8]

Binary Representations:

- 3 → 000...0011
- 10 → 000...1010
- 5 → 000...0101
- 25 → 000...11001

Best XOR Pair:

- 5 ^ 25 = 28
- 0101 ^ 11001 = 11100 → Decimal = 28

Output: 28

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

Bit manipulation is the act of directly operating on bits (0s and 1s) of data using bitwise operators. It allows you to perform operations at the binary level of integers, which is how data is actually stored and processed in memory.

Common Bitwise Operators:

Operator	Symbol	Description	Example (a = 5, b = 3)
AND	&	Bitwise AND	a & b = 1
OR	|	Bitwise OR	`a
XOR	^	Bitwise XOR	a ^ b = 6
NOT	~	Bitwise NOT (one's complement)	~a = -6

Left Shift	<<	Shifts bits to the left	a << 1 = 10
Right Shift	>>	Shifts bits to the right	a >> 1 = 2

Advantages of Bit Manipulation

1. Speed
   - Bitwise operations are blazingly fast – usually take just 1 CPU cycle.
   - Much faster than arithmetic or logical operations.

2. Memory Efficiency
   - You can use a single integer (int) to store multiple flags or states by using individual bits.
   - Example: Representing 32 on/off switches with just one 32-bit integer.

3. Powerful for Certain Algorithms Used in:
   - Checking powers of 2: n > 0 && (n & (n - 1)) == 0
   - Finding single/non-repeating elements: Using XOR
   - Subsets generation
   - Trie-based XOR problems
   - Bitmasking in DP and graph problems

4. Elegant Mathematical Tricks
   - Swapping two numbers without a temporary variable using XOR
   - Setting, unsetting, and toggling bits precisely

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.

Given an array of integers nums, for each element nums[i], find the next greater element to its right. If no such element exists, use -1.

Example

Input: [4, 5, 2, 10, 8]
Output: [5,10,10,-1,-
1] Explanation:

- 4 → next greater is 5
- 5 → next greater is 10
- 2 → next greater is 10
- 10 → no greater element → -1
- 8 → no greater element → -1
-

Efficient Solution: Using Stack (Monotonic Stack)

We use a stack to keep track of potential "next greater elements". We traverse the array from right to left, which allows us to:

- Maintain a stack of decreasing elements.
- Pop from the stack until we find a greater element.

Algorithm

1. Initialize a stack and result array of same size.
2. Traverse the array from right to left.
3. For each nums[i]:

- While the stack is not empty and stack.peek() <= nums[i], pop the stack.
- If the stack is empty, set result[i] = -1.
- Else, result[i] = stack.peek().
- Push nums[i] into the stack.

Program

```
import java.util.Stack; import
java.util.Arrays;

public class NextGreaterElement {

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

        for (int i = n - 1; i >= 0; i--) {

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

            // Push current element into the stack stack.push(nums[i]);
        }

        return result;
    }

    public static void main(String[] args) { int[]
        nums = {4, 5, 2, 10, 8};
        int[] result = nextGreaterElements(nums);

        System.out.println("Next greater elements: " + Arrays.toString(result));
    }
}
```

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.

Algorithm (Two-pointer technique)
To remove the n-th node from the end:
1. Use two pointers, first and second.
2. Move first pointer n steps ahead.
3. Move both first and second one step at a time until first reaches the end.
4. Now, second is just before the node to delete. Update second.next = second.next.next

Program

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

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

public class RemoveNthNode {

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

        ListNode first = dummy; ListNode
        second = dummy;
        for (int i = 0; i <= n; i++) { first
            = first.next;
        }
        while (first != null) { first
            = first.next;
            second = second.next;
        }
        second.next = second.next.next;

        return dummy.next;
    }

    // Utility function to print 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) { 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; // Remove 2nd node from end head
        = removeNthFromEnd(head, n);

        System.out.print("After removing " + n + "th node from end: "); printList(head);
    }
```

}

3Time Complexity: O(L) Space
Complexity: O(1)

Example
Input: 1 -> 2 -> 3 -> 4 -> 5, n = 2
Goal: Remove the 2nd node from the end → Node 4.
Steps:

- Move first 2+1 = 3 steps ahead → lands on node 3.
- Move both first and second together until first reaches the end.
- second now points to node 3, so skip node 4: second.next = second.next.next.

Output: 1 -> 2 -> 3 -> 5

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

Algorithm (Two-Pointer Technique) To find the intersection:

1. Use two pointers pA and pB, starting at the heads of the two lists.
2. Traverse through the lists.
   - When pA reaches the end of list A, set it to headB.
   - When pB reaches the end of list B, set it to headA.
3. Continue moving both pointers. If the lists intersect, pA and pB will meet at the intersection node. If not, both will reach null.

Why this works:

- Each pointer travels exactly lengthA + lengthB, so they align perfectly on a second pass.

Program

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

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

      ListNode pA = headA;
      ListNode pB = headB;

      while (pA != pB) { pA = (pA == null) ?
         headB : pA.next; pB = (pB == null) ?
         headA : pB.next;
      }
```

```
        return pA; // Could be null or the intersection node
    }
    public static void printIntersection(ListNode node) { if
        (node != null) {
            System.out.println("Intersection at node with value: " + node.val);
        } else {
            System.out.println("No intersection found.");
        }
    }

    public static void main(String[] args) {
        ListNode common = new ListNode(8);
        common.next = new ListNode(9);

        ListNode headA = new ListNode(1); headA.next
        = new ListNode(2);
        headA.next.next = common;

        ListNode headB = new ListNode(4);
        headB.next = new ListNode(5);
        headB.next.next = common;

        ListNode intersection = getIntersectionNode(headA, headB);
        printIntersection(intersection);
    }
}
```

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

Example:
List A: 1 → 2 → 8 → 9
List B: 4 → 5 → 8 → 9
The intersection starts at node 8. Since both lists share 8 → 9, the reference is common. Both pointers will eventually align at node 8 after switching lists once.

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

Algorithm (Two Stack in One Array)
Key Idea:

- Use a single array arr[] of size n.
- Maintain two pointers:
  - top1 for Stack 1 (starts from beginning, grows right).
  - top2 for Stack 2 (starts from end, grows left).
- Ensure top1 < top2 always.

Operations:

- Push1(x):
  - If space exists (top1 < top2 - 1), then arr[++top1] = x
- Push2(x):
  - If space exists (top1 < top2 - 1), then arr[--top2] = x
- Pop1():

- If top1 >= 0, return arr[top1--]
- Pop2(): ○ If top2 < size, return arr[top2++]

Program class TwoStacks { int size;
int[] arr; int top1, top2;
TwoStacks(int n) {
size = n; arr = new int[n]; top1 = -1; top2 = n;
}

void push1(int x) {
if (top1 < top2 - 1) {
arr[++top1] = x;
} else {
System.out.println("Stack Overflow on Stack 1");
}
}
void push2(int x) { if (top1 < top2 - 1) {
arr[--top2] = x;
} else {
System.out.println("Stack Overflow on Stack 2");
}
}
int pop1() {
if (top1 >= 0) {
return arr[top1--];
} else {
System.out.println("Stack Underflow on Stack 1"); return -1;
}
}
int pop2() {
if (top2 < size) {
return arr[top2++];
} else {
System.out.println("Stack Underflow on Stack 2"); return -1;
}
}
}

public class Main { public static void main(String[] args) {
TwoStacks ts = new TwoStacks(10);

ts.push1(5);
ts.push1(10);

```
        ts.push2(100);
        ts.push2(200);

        System.out.println("Popped from Stack 1: " + ts.pop1()); // 10
        System.out.println("Popped from Stack 2: " + ts.pop2()); // 200
    }
}
```

Time Complexity for all operations:

- Push: O(1)
- Pop: O(1)

Space Complexity: O(n)

Example

Array Size = 10

Initial:
top1 = -1, top2 = 10

ts.push1(5) => top1 = 0, arr[0] = 5 ts.push2(100)
=> top2 = 9, arr[9] = 100

ts.push1(10) => top1 = 1, arr[1] = 10 ts.push2(200)
=> top2 = 8, arr[8] = 200

pop1() => 10, top1 = 0 pop2()
=> 200, top2 = 9

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.

Algorithm (Reverse Half of the Number)

To avoid overflow and unnecessary work, we only reverse half of the number:

1. Negative numbers are not palindromes.
2. If the number ends with 0 (but is not 0), it's not a palindrome.
3. Reverse the second half of the digits.
4. Compare the original first half with the reversed second half.

Steps:

- While x > reversedHalf:
    - Get last digit: digit = x % 10
    - Append to reversedHalf: reversedHalf = reversedHalf * 10 + digit ○ Remove digit from x: x /= 10
- Finally: ○ If x == reversedHalf (even digits) or x == reversedHalf/10 (odd digits), return true.

Program

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

       int reversedHalf = 0;
       while (x > reversedHalf) {
       int digit = x % 10;
          reversedHalf = reversedHalf * 10 + digit; x
          = x / 10;
       }
       return (x == reversedHalf || x == reversedHalf / 10);
   }

   public static void main(String[] args) {
       int num1 = 121; int num2 = 123;

       System.out.println(num1 + " is palindrome? " + isPalindrome(num1)); // true
       System.out.println(num2 + " is palindrome? " + isPalindrome(num2)); // false
   }
}
```

Time Complexity: O(log10(n))
Space Complexity: O(1)

Example:
Input: x = 1221

- Initial: x = 1221, reversedHalf = 0 •
  Iteration 1: ○ Digit = 1, reversedHalf = 1, x =
  122 • Iteration 2:
  ○ Digit = 2, reversedHalf = 12, x = 12

Now x == reversedHalf, so it's a palindrome.

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 not stored in contiguous memory. Each node contains:

1. Data
2. Pointer (or reference) to the next node

Types of Linked Lists:

1. Singly Linked List
   - Each node points to the next node only.

2. Doubly Linked List
   - Each node points to both the previous and next nodes.

3. Circular Linked List
   - The last node points back to the first node.

Why Use Linked Lists?

- Dynamic Size: Can grow or shrink at runtime easily.

- Efficient Insert/Delete: O(1) time for inserting/removing elements at head or middle (if you have the pointer).

- Memory Usage: Doesn't require contiguous memory like arrays.

Applications in Algorithm Design

1. Stacks & Queues Implementation
   - Singly linked lists are used to implement stacks (LIFO) and queues (FIFO) efficiently.

2. Dynamic Memory Management
   - Operating systems use linked lists to keep track of memory chunks (free and used blocks).

3. Graph Representations
   - Adjacency lists in graphs are often implemented using linked lists.

4. Polynomial Arithmetic
   - Polynomials are stored as linked lists for easier manipulation of terms.

5. Hash Tables with Chaining
   - In collision resolution, buckets can be linked lists.

6. Undo/Redo Functionality
   - Doubly linked lists are useful in text editors for tracking history.

7. Browser History
   - Doubly linked lists can track backward and forward navigation.

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.

Algorithm Using Deque

Use a deque to store indices of elements in the current window, in decreasing order of their values.

Steps:

1. Create an empty deque dq and a list result.
2. Iterate through each element nums[i]:
   - Remove indices from back of deque while nums[i] > nums[dq.back()] (they're useless).
   - Add current index i to back of deque.
   - Remove front if it's out of the window (i.e., dq.front() <= i - k).
   - If window size ≥ k, add nums[dq.front()] to result.
3. Return result.

Program

```
import java.util.*;

public class SlidingWindowMaximum {

    public static List<Integer> maxSlidingWindow(int[] nums, int k) {
        Deque<Integer> dq = new LinkedList<>();
        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < nums.length; i++) {
```

```
            if (!dq.isEmpty() && dq.peekFirst() <= i - k) {
                dq.pollFirst();
            }
            while (!dq.isEmpty() && nums[i] >= nums[dq.peekLast()]) {
                dq.pollLast();
            }

            dq.offerLast(i); // Add current index if
            (i >= k - 1) {
                result.add(nums[dq.peekFirst()]);
            }
        }

        return result;
    }

    public static void main(String[] args) { int[]
        nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        List<Integer> output = maxSlidingWindow(nums, k);
        System.out.println("Sliding window maximums: " + output);
    }
}
```

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

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

Windows and max:
[1, 3, -1] → max = 3
[3, -1, -3] → max = 3
[-1, -3, 5] → max = 5
[-3, 5, 3] → max = 5
[5, 3, 6] → max = 6
[3, 6, 7] → max = 7

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.

Algorithm (Using Stack)
For each bar, find the first smaller bar to the left and right.
Use a stack to maintain increasing bar indices.
For each bar popped, compute area with it as the smallest (limiting) height.

Steps:
1. Initialize an empty stack and maxArea = 0.
2. Loop through each index i in heights[]:
    - While stack is not empty and heights[i] < heights[stack.top()]:
    - Pop from stack and calculate the area:
    - height = heights[top]
    - width = i - stack.peek() - 1 (if stack not empty), else i

- update maxArea
- Push i to the stack.

3. After the loop, clear the stack:
   - Do similar area calculations for remaining bars.

Program

```
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;
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 = 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};
        int maxArea = largestRectangleArea(heights);
        System.out.println("Maximum area: " + maxArea); // Output: 10
    }
}
```

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

Example:
Input: [2, 1, 5, 6, 2,
3] Stack Simulation:
i = 0: stack = [0]
i = 1: heights[1] < heights[0] → pop → area = 2*1 = 2 → stack = [], push 1
i = 2: stack = [1, 2] i = 3: stack = [1, 2, 3]
i = 4: heights[4] < heights[3] → pop 3 → area = 6*1 = 6
→ pop 2 → area = 5*2 = 10 (max so far) → push 4
...

Final max area: 10

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

The Sliding Window is a technique used to reduce time complexity in problems involving linear data structures like arrays or strings.

Instead of recalculating results for every subarray or substring from scratch, you reuse previous computations by "sliding" a window over the input.

When to Use It?

- When you're dealing with contiguous subarrays or substrings
- When you're asked to find things like:
  - Maximum/Minimum in a subarray
  - Sum/Product of a subarray
  - Longest substring with specific conditions

Types of Sliding Window

1. Fixed Size Window

You know the size of the window k (like sum or max in a window of size k).

Example:
Find the maximum sum of any subarray of size k

Input: arr = [2, 1, 5, 1, 3, 2], k = 3
Output: 9 → [5, 1, 3]

Logic:

- Start with first window, then slide by one element at a time.
- Subtract the element going out, add the new element coming in.

2. Variable Size Window

The window size changes based on a condition.

Example:
Find the longest substring without repeating characters

Input: s = "abcabcbb"
Output: 3 → "abc"

Logic:

- Use a set or map to track characters.
- Shrink or expand the window based on duplicates.

Applications of Sliding Window

Problem Type	Description
Maximum/Minimum Subarray Sum	Find max/min sum of window of size k
Longest Substring	Longest substring with no repeats, at most K distinct characters
Anagram Detection	Check if one string is an anagram of another within a window
Median/Max in Sliding Window	Use Deque or Heap for efficient computation
Subarrays with Product < K	Count subarrays meeting a condition

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.

Efficient Approach (Using Hashing)

Key Idea:

- Use prefix sum + HashMap to store how often a particular cumulative sum has occurred so far.

Let's say:

- prefixSum = sum of nums[0..i]
- Then if prefixSum - k has occurred before, it means the subarray (some subarray ending at i) has sum k.

Algorithm:

1. Initialize prefixSum = 0, count = 0.
2. Use a HashMap<Integer, Integer> to store frequency of prefix sums.
3. Initialize map with {0: 1} to handle case where prefixSum == k.
4. For each number in nums:
    - Add it to prefixSum
    - If (prefixSum - k) exists in map, add its frequency to count
    - Update the map: increment frequency of prefixSum 5. Return count.

Program

```
import java.util.HashMap;
public class SubarraySumEqualsK {

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

        for (int num : nums) {
            prefixSum += num;

            if (map.containsKey(prefixSum - k)) { count
                += map.get(prefixSum - k);
            }

            map.put(prefixSum, map.getOrDefault(prefixSum, 0) + 1);
        }

        return count;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3, -2, 5}; int k =
        5;
```

```
        System.out.println("Total subarrays with sum " + k + ": " + subarraySum(nums, k));
    }
}
```

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

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

Prefix Sum Steps:
i = 0: sum = 1 → map = {0:1, 1:1}
i = 1: sum = 3 → (3 - 3 = 0) found in map → count = 1 → map = {0:1, 1:1, 3:1} i =
2: sum = 6 → (6 - 3 = 3) found in map → count = 2 → map = {0:1, 1:1, 3:1, 6:1}

Output: 2

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.

Algorithm Using Priority Queue
Key Idea:
We can use a frequency map to count the occurrences of each element in the array, and then use a min-heap (priority queue) to efficiently keep track of the k most frequent elements.

Steps:
1. Count frequencies of each element in the array using a hash map (or a frequency map).
2. Create a min-heap that stores the elements based on their frequency.
    - The heap will store pairs of (frequency, element).
3. If the size of the heap exceeds k, remove the element with the least frequency from the heap.
4. After processing all elements, the heap will contain the k most frequent elements.
5. Extract and return the elements from the heap

Program

```
import java.util.*;

public class KMostFrequentElements {

    public static List<Integer> topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> freqMap = new HashMap<>(); for
        (int num : nums) { freqMap.put(num,
        freqMap.getOrDefault(num, 0) + 1);
        }
        PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>(
            (a, b) -> a.getValue() - b.getValue() // Min-heap based on frequency
        );
        for (Map.Entry<Integer, Integer> entry : freqMap.entrySet())
            { pq.offer(entry); if (pq.size() > k) { pq.poll();
            }
        }
```

```
        List<Integer> result = new ArrayList<>();
        while (!pq.isEmpty()) {
        result.add(pq.poll().getKey());
        }
        return result;
    }

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

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

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

1. Frequency Map: {1: 3, 2: 2, 3: 1}
2. Heap Operations:
    - Insert (1, 3) → Heap: [(1, 3)]
    - Insert (2, 2) → Heap: [(2, 2), (1, 3)]
    - Insert (3, 1) → Heap: [(1, 3), (2, 2), (3, 1)] → Remove (3, 1) → Heap: [(2, 2), (1, 3)]
3. Extract from Heap:
    - Extract (2, 2) → Result: [2]
    - Extract (1, 3) → Result: [2, 1]

Output: [1, 2]

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

Algorithm (Recursive / Backtracking Approach):
1. Start with an empty list to store subsets.
2. Use backtracking to explore all possibilities:
    - For each element, you have two choices: include it in the current subset or exclude it.
3. Recursively build all subsets by exploring both choices for each element.

Example:
For input array: {1, 2, 3} Generated
subsets are:
[]
[1]
[1, 2]
[1, 2, 3]

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

Program:

```
import java.util.*;

public class SubsetsGenerator { public static List<List<Integer>>
   generateSubsets(int[] nums) {
      List<List<Integer>> result = new ArrayList<>();
      backtrack(0, nums, new ArrayList<>(), result);
      return result;
   }
   private static void backtrack(int index, int[] nums, List<Integer> current, List<List<Integer>> result) {

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

      for (int i = index; i < nums.length; i++) current.add(nums[i]);
         backtrack(i + 1, nums, current, result);
         current.remove(current.size() - 1);
      }
   }
   public static void main(String[] args) { int[]
      array = {1, 2, 3};
      List<List<Integer>> subsets = generateSubsets(array);

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

Time Complexity = O(n * 2^n)
Space Complexity:

- Auxiliary Space (Stack): O(n) due to recursion.
- Result Storage: O(n * 2^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.

Given an array of positive integers (candidates) and a target number, find all unique combinations where the chosen numbers sum to the target.

- You may use the same number multiple times.
- All numbers (including target) are positive integers.
- The solution set must not contain duplicate combinations.

Algorithm (Backtracking):

1. Sort the input array (optional but helpful for optimization).
2. Use backtracking to try all combinations:
   - At each step, either include the current number or move to the next one.

- Reduce the target by the chosen number.
- Stop when the target becomes negative.
- If the target is 0, add the current combination to the result.

Example:
For input: candidates =
[2, 3, 6, 7] target = 7

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

Program:

```
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;
      }
      current.add(candidates[index]);
      backtrack(index, candidates, target - candidates[index], current, result);
      current.remove(current.size() - 1); // Backtrack 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> combination : combinations) {
      System.out.println(combination);
      }
   }
}
```

Time Complexity = O(2^target)
Space Complexity = O(target * number of combinations)

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

Algorithm (Backtracking Approach):

1. Use backtracking to generate all possible orderings.
2. At each recursion level:
   - Swap the current element with every element after it (including itself).
   - Recurse on the next index.
   - After recursion, backtrack (swap back to restore original state).

This generates all permutations by fixing one element at a time and recursively permuting the rest.

Example:

Input: {1, 2, 3}
Output:
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 2, 1]
[3, 1, 2]
Total permutations = 3! = 6

Program:

```
import java.util.*;

public class PermutationsGenerator {

    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) {
            List<Integer> permutation = new ArrayList<>(); for
            (int num : nums) {
                permutation.add(num);
            }
            result.add(permutation);
            return;
        }

        for (int i = index; i < nums.length; i++) {
            swap(nums, index, i);
            backtrack(index + 1, nums, result);
            swap(nums, index, i);
        }
    }
```

```
    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[]
        array = {1, 2, 3};
        List<List<Integer>> permutations = permute(array);

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

Time Complexity = O(n × n!)
Space Complexity:

- Recursion stack = O(n) (depth of recursion).
- Result storage = O(n × n!) (to store all permutations).

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

1. Subsets
A subset is any selection of elements from a set — order doesn't matter, and elements may be included or not.

Key Properties:

- Each element is either included or excluded.
- The number of subsets for a set of size n is 2^n.
- Order does NOT matter.

Example:
Given the array: {1, 2, 3} All subsets ($2^3 = 8$ subsets):

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

Note: [1, 2] and [2, 1] are considered the same subset.

2. Permutations
A permutation is a rearrangement of elements, considering the order.

Key Properties:
- All elements are used (typically).
- The number of permutations of n unique elements is n! (factorial).
- Order matters.

Example:
Given the array: {1, 2, 3}
All permutations (3! = 6 permutations):
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
Note: [1, 2, 3] and [3, 2, 1] are different permutations.

Subsets vs Permutations:

Feature	Subsets	Permutations
Order matters	+ No	Yes
Duplicates allowed?	+ No (usually unique elements)	+ No (in basic case)
Number of results	2^n	$n!$
Use case	Choosing elements	Rearranging elements
Example	`[1, 2]` = `[2, 1]` (same)	`[1, 2]` ≠ `[2, 1]` (different)

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.

Algorithm:
1. Create a frequency map (HashMap):
   - Iterate over the array.
   - For each element, count how many times it appears.
2. Find the element with the highest count:
   - Iterate over the map entries to find the key with the maximum value.

Example:
Given array:
{1, 3, 2, 1, 4, 1, 3, 3, 3} Frequencies:
- 1 → 3 times
- 2 → 1 time
- 3 → 4 times
- 4 → 1 time

Output:

Element with maximum frequency: 3

Program:

```
import java.util.*;

public class MaxFrequencyElement { public static int
   findMaxFrequencyElement(int[] arr) { Map<Integer, Integer>
   frequencyMap = new HashMap<>(); for (int num : arr) {
         frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
      }
      int maxFreq = 0; int maxFreqElement = arr[0]; // Default to first
      element for (Map.Entry<Integer, Integer> entry :
      frequencyMap.entrySet()) { if (entry.getValue() > maxFreq) {
      maxFreq = entry.getValue();
            maxFreqElement = entry.getKey();
         }
      }

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

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

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

Kadane's Algorithm
1. Initialize: maxSoFar = nums[0] — the maximum subarray sum found so far.
currentMax = nums[0] — the maximum sum of subarray ending at the current index.
2. Iterate through the array starting from index 1:

- At each step, update currentMax as:
  - currentMax = Math.max(nums[i], currentMax + nums[i]);
- Update maxSoFar as: ○ maxSoFar = Math.max(maxSoFar, currentMax);

3. Return maxSoFar.

Example:
Input array: {-2, 1, -3, 4, -1, 2, 1, -5, 4}
The maximum subarray is: [4, -1, 2, 1] → sum = 6 Output:
Maximum subarray sum is: 6

Program:

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

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

      return maxSoFar;
   }
   public static void main(String[] args) { int[]
      array = {-2, 1, -3, 4, -1, 2, 1, -5, 4}; int
      result = maxSubarraySum(array);
      System.out.println("Maximum subarray sum is: " + result);
   }
}
```

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 (DP) is an optimization technique used to solve problems that can be broken down into overlapping subproblems with optimal substructure.

Key Concepts:
1. Optimal Substructure: A problem has optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.
2. Overlapping Subproblems: The problem can be divided into subproblems which are solved multiple times.

How DP Works:
- Store the solutions to subproblems (using arrays or tables).
- Reuse them to build up the final answer (avoiding recomputation).

Maximum Subarray Problem (Using Kadane's Algorithm)
Problem Statement:
Given an integer array (can have negative numbers), find the contiguous subarray with the maximum sum.

How This Fits DP: Subproblem:
Let's define:
- dp[i] = maximum subarray sum ending at index i Then: dp[i] = max(arr[i], dp[i-1] + arr[i])

This means:
- Either start a new subarray at index i
- Or extend the subarray ending at i-1

Optimal Substructure:

- Best subarray at i depends on the best subarray at i-1

Overlapping Subproblems:

- We reuse the solution of dp[i-1] to calculate dp[i]

DP-Based Implementation (Kadane's):

```
public class MaxSubarrayDP {

    public static int maxSubarraySum(int[] arr) {
        int n = arr.length; int[] dp = new int[n];
        dp[0] = arr[0];
        int maxSum = arr[0];
for (int i = 1; i < n; i++) { dp[i] = Math.max(arr[i],
        dp[i - 1] + arr[i]); maxSum =
        Math.max(maxSum, dp[i]);
        }

        return maxSum;
    }

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

Example:

For array: {−2, 1, −3, 4, −1, 2, 1, −5, 4}

DP values (dp[]):

Index	Value	dp[i] (max subarray ending at i)
0	-2	-2
1	1	max(1, -2+1) = 1
2	-3	max(-3, 1-3) = -2
3	4	max(4, -2+4) = 4
4	-1	max(-1, 4-1) = 3 5 2 max(2, 3+2) = 5
6	1	max(1, 5+1) = 6 ← max
7	-5	max(-5, 6-5) = 1 8 4 max(4, 1+4) = 5

Maximum subarray sum = 6

Time & Space Complexity:

Complexity	Value
Time	O(n)
Space (DP array)	O(n)

Complexity	Value
Time	O(n)
Space (DP array)	O(n)

Space (Kadane's) O(1) Space (Kadane's) 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.

Algorithm:
Step-by-step Approach:
1. Frequency Map:
   - Count the frequency of each element using a HashMap.
2. Min-Heap (Priority Queue):
   - Use a min-heap (priority queue) of size k to keep track of the top k frequent elements.
3. Extract Top K Elements:
   - The heap will contain the k elements with the highest frequencies.

Example:
Input: nums = {1, 1, 1, 2, 2, 3, 4, 4, 4, 4}, k = 2

Frequencies:
- 1 → 3 times
- 2 → 2 times
- 3 → 1 time
- 4 → 4 times

Output: Top 2 frequent elements: [1, 4] or [4, 1]

Program:

```
import java.util.*;

public class TopKFrequentElements {

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

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

        for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet())
            { minHeap.add(entry); if (minHeap.size() > k) { minHeap.poll();
            // Remove element with lowest frequency
            }
        }
        List<Integer> result = new ArrayList<>(); while
        (!minHeap.isEmpty()) {
        result.add(minHeap.poll().getKey());
        }
        Collections.reverse(result);
        return result;
    }
```

```
    public static void main(String[] args) { int[]
        nums = {1, 1, 1, 2, 2, 3, 4, 4, 4, 4}; int k
        = 2;

        List<Integer> topK = topKFrequent(nums, k);
        System.out.println("Top " + k + " frequent elements: " + topK);
    }
}
```

Time Complexity: O(n log k)
Space Complexity: HashMap: O(n) Heap:
O(k)

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.

Algorithm:

1. Create an empty HashMap (Map<Integer, Integer>) to store (value, index).
2. Loop through the array:
   - Calculate complement = target - arr[i]
   - If complement exists in the map, return the pair.
   - Else, put arr[i] in the map.
3. If no pair is found, return an indication (e.g., null or empty).

Program:

```
import java.util.*;

public class TwoSumHashing {

    public static int[] findTwoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>(); // value -> index
for (int i = 0; i < nums.length; i++) { int
        complement = target - nums[i];

            if (map.containsKey(complement)) { return new
               int[]{map.get(complement), i}; // indices
            }
            map.put(nums[i], i);
        }

        return new int[]{}; // No pair found
    }

    public static void main(String[] args) {
        int[] array = {2, 7, 11, 15}; int target
        = 9;
        int[] result = findTwoSum(array, target); if
        (result.length == 2) {
            System.out.println("Two elements found at indices: " + result[0] + " and " + result[1]);
        } else {
            System.out.println("No pair found.");
```

```
      }
    }
}
```
Example:
Input: array = {2, 7, 11, 15} target
= 9

Explanation:

- 2 + 7 = 9 ✅
- So the output is: indices 0 and 1

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

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

A Priority Queue is a special type of queue in which each element is associated with a priority, and elements are served based on their priority (not just their order of arrival).

Key Properties:

- Elements with higher priority are dequeued before elements with lower priority.
- If two elements have the same priority, the one inserted first is served first (FIFO order for equal priority).

How Is It Implemented?
Internally, priority queues are commonly implemented using heaps (typically binary heaps):

- Min-Heap: The element with the lowest value (highest priority) comes first.
- Max-Heap: The element with the highest value comes first.

In Java:

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
```

Applications in Algorithms:
1. Dijkstra's Algorithm (Shortest Path)
Uses a min-heap to efficiently get the node with the smallest distance.

2. A* Search Algorithm
Uses a priority queue based on f(n) = g(n) + h(n) (cost + heuristic).

3. Huffman Encoding (Data Compression)
Builds a tree using a min-heap based on character frequency.
4. Top K Problems
Like "Top K Frequent Elements" — use a min-heap to track top k values.

5. CPU Scheduling / Task Management
Priority queues simulate job scheduling based on task priority.

6. Merging K Sorted Lists / Arrays

Use a min-heap to repeatedly extract the minimum of all heads.

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.

You are given a string S. You need to find the longest palindromic substring within S.

Example:
Input: "babad"
Output: "bab" or "aba" (Both are correct since they are longest palindromes)

Algorithm (Expand Around Center Approach):
1. Initialize a variable to store the start index and maximum length of the palindrome.
2. Loop through the string, treating each character as a center.
3. For each center, try to expand outwards to check for the longest palindrome around it.
4. Do this for both:
   - Odd length palindromes (centered at one character) •
     Even length palindromes (centered between two characters)
5. Track the longest palindrome substring during expansion.
6. Return the substring from start index with the recorded maximum length.

Program:

```
public class LongestPalindrome { public static int expandAroundCenter(String
    s, int left, int right) { while (left >= 0 && right < s.length() && s.charAt(left) ==
    s.charAt(right)) { left--;
            right++;
        }
        return right - left - 1;
    }

    public static 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);
    }

    public static void main(String[] args) {
        String input = "babad";
        String result = longestPalindrome(input);
        System.out.println("Longest Palindromic Substring: " + result);
```

```
    }
}
```

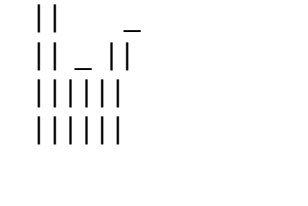
Time Complexity: O(n)
Space Complexity: O(1)

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

In programming and algorithm design, a histogram problem usually refers to dealing with a bar chart where each bar has a certain height, and you're asked to solve questions like:

Classic Problem: Largest Rectangle in a Histogram
Given an array of integers representing the heights of bars in a histogram (where each bar has width = 1), find the area of the largest rectangle that can be formed within the bounds of the histogram.

Example:
Input: [2, 1, 5, 6, 2, 3]
Visual: Each number is a bar height _



Index: 0 1 2 3 4 5

The largest rectangle has area = 10, from bars of height 5 and 6 (width = 2).

Algorithm Design Techniques Stack-Based
Approach (Efficient)
Idea: Use a monotonic stack to store indices of the histogram bars in increasing order. This helps us find:

- The nearest smaller element to the left and right of every bar.
- This way we calculate the maximum width a bar can extend to form the largest rectangle.

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

Applications in Algorithm Design
Histogram problems are not just about bar charts. The core idea is about area, range, boundaries, or constraints — very useful in:

1. Maximal Rectangle in a Binary Matrix

-

  You convert each row of the matrix into a histogram.
- Then solve the largest rectangle histogram problem for each row.
- Application: Image processing, terrain mapping, 2D data analysis.

2. Monotonic Stack Problems
   - Histogram logic inspires a whole class of problems that use monotonic increasing or decreasing stacks to track local maximums/minimums.

Examples:
- Trapping Rain Water Problem
- Stock Span Problem
- Next Greater Element / Previous Smaller Element

3. Dynamic Programming + Histogram
   - In some matrix problems, histograms are used with DP to track continuous 1s or heights over rows.

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.

Given an array of integers (representing a permutation), rearrange them into the next lexicographically greater permutation.
If such an arrangement is not possible (i.e., the array is in descending order), rearrange it to the lowest possible order (i.e., sorted in ascending order).

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

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

Input: [1, 1, 5]
Output: [1, 5, 1]

Algorithm (Step-by-step):
1. Find the first decreasing element from the right (i.e., find the pivot).
   - Let index i be the last position such that nums[i] < nums[i + 1].
2. If no such index exists, the array is in descending order:
   - Reverse the entire array and return (it's the last permutation).
3. Find the element just larger than nums[i] from the right:
   - Let index j be the last element such that nums[j] > nums[i].
4. Swap nums[i] and nums[j].
5. Reverse the suffix starting from i + 1 to the end of the array.

Program:
```
import java.util.Arrays;
```

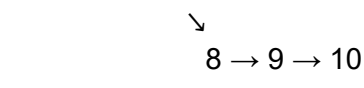
•

```
public class NextPermutation {

    public static void nextPermutation(int[] nums) { int
        n = nums.length;




        int i = n - 2;
        while (i >= 0 && nums[i] >= nums[i + 1]) {
        i--; } if (i >= 0) { int j = n - 1;
            // Step 3: Find element just larger than nums[i]
            while (j >= 0 && nums[j] <= nums[i]) { j--;
            }
            swap(nums, i, j);
        }
        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));
    }
}
```

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.

Example:



↗
List B: 3 → 6 → 7
Intersection Point: Node with value 8.

Algorithm (Optimal Approach – Length Difference):

1. Find the lengths of both linked lists: lenA and lenB.
2. Calculate the difference d = |lenA - lenB|.
3. Move the pointer of the longer list forward by d steps.
4. Now, traverse both lists together until:
   - You find a common node (intersection)
   - Or reach the end (no intersection)

Program:

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

public class IntersectionOfLinkedLists {
    public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = getLength(headA);
        int lenB = getLength(headB);
        while (lenA > lenB) {
            headA = headA.next;
            lenA--;
        }

        while (lenB > lenA) {
            headB = headB.next;
            lenB--;
        }
        while (headA != null && headB != null) {
            if (headA == headB) return headA;
            headA = headA.next; headB =
            headB.next;
        }

        return null; // No intersection
    }

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

```
    •

      return length;
   }

   public static void main(String[] args) {
      ListNode common = new ListNode(8);
      common.next = new ListNode(9);
      common.next.next = new ListNode(10);
      ListNode headA = new ListNode(1);
      headA.next = new ListNode(2);
      headA.next.next = common;

      ListNode headB = new ListNode(3);
      headB.next = new ListNode(6);
      headB.next.next = new ListNode(7);
      headB.next.next.next = common;

      ListNode intersection = getIntersectionNode(headA, headB); if
      (intersection != null)
         System.out.println("Intersection at node with value: " + intersection.val); else
         System.out.println("No intersection found.");
   }
}
```

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

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

In an array, an equilibrium index is an index i such that:

- Sum of elements to the left of i = Sum of elements to the right of i

Mathematically:
For index i in array A[0..n-1], it's an equilibrium index if:

- A[0] + A[1] + ... + A[i-1] = A[i+1] + A[i+2] + ... + A[n-1]

Note: The element at index i is not included in either sum.

Example:
Input: A = [-7, 1, 5, 2, -4, 3, 0]
↑
Output: Index 3 is an equilibrium index because:
Left sum = (-7 + 1 + 5) = -1
Right sum = (-4 + 3 + 0) = -1

Algorithm to Find Equilibrium Index

1. Compute the total sum of the array.
2. Initialize leftSum = 0
3. Traverse the array:
    - Subtract the current element from total to get rightSum
    - If leftSum == rightSum, it's an equilibrium index
    - Add current element to leftSum for next iteration

Applications of Equilibrium Index

1. Load Balancing
    - Useful in distributing tasks or load equally across systems (e.g., in distributed computing or networking).

2. Partitioning Problems

    Used to find a balance point where data can be split evenly (e.g., memory management, file partitioning).

3. Game Theory / Strategy Games
    - Analyzing balance points in strategic decisions where left and right paths hold equal weight.

4. Finance
    - Used in algorithms to find break-even points in stock analysis or cost-benefit computations.

5. Interview Problems
    - Common question in coding interviews due to its mix of logic, optimization, and array manipulation.