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

Sol.

Prefix sum array is a derived array that allows calculation of sum of elements in a subarray.

For an array ‘A’ , prefix sum array ‘P’ will be :

P[i]=A[0]+A[1]+…+A[i]

Applications –

It is mostly used in problems concerning the sum of subarray , used to calculate running sum of sub array.

1. **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.**

|  |
| --- |
| import java.util.Scanner;  class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter the size of the array: ");  int n = sc.nextInt();    int[] arr = new int[n];  System.out.println("Enter " + n + " elements:");  for(int i = 0; i < n; i++) {  arr[i] = sc.nextInt();  }    int[] prefix = new int [n];  prefix[0]=arr[0];  for(int i=1;i<n;i++){  prefix[i]=prefix[i-1]+arr[i];  }    System.out.print("Enter starting range : ");  int x = sc.nextInt();  System.out.print("Enter ending array : ");  int y = sc.nextInt();    System.out.print("The Sum of elements in range is : "+ (y-x));  }  } |

Sol.

Time complexity – O(n)

Space complexity – O(n)

Example –

[1,2,3,4,5]

Let range be [2,4]

Suffix array : [1,3,6,10,15]

Answer : suffix[4] – suffix[2] = 15-6 = 9

1. **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.**

Sol.

|  |
| --- |
| import java.util.Scanner;  class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter the size of the array: ");  int n = sc.nextInt();    int[] arr = new int[n];  System.out.println("Enter " + n + " elements:");  for(int i = 0; i < n; i++) {  arr[i] = sc.nextInt();  }    int[] prefix = new int [n];  prefix[0]=arr[0];  for(int i=1;i<n;i++){  prefix[i]=prefix[i-1]+arr[i];  }  int[] suffix = new int [n];  suffix[n-1]=arr[n-1];  for(int i=n-2;i>=0;i--){  suffix[i]=suffix[i+1]+arr[i];  }  for(int i=1;i<n-1;i++){  if(prefix[i-1]==suffix[i+1]){  System.out.print("Equilibrium index : "+i);  }  }  }  } |

Time complexity – O(n)

Space complexity – O(n)

Example –

[1,2,3,4,6]

Prefix : [1,3,6,10,16]

Suffix : [16,15,13,10,6]

Answer : equilibrium index = 3 (prefix[i-1]=suffix[i+1] for I = 3)

1. **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.**

Sol.

Yes, it can be done by comparing prefix and suffix sum at each step.

|  |
| --- |
| import java.util.Scanner;  class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter the size of the array: ");  int n = sc.nextInt();  int[] arr = new int[n];  System.out.println("Enter " + n + " elements:");  for(int i = 0; i < n; i++) {  arr[i] = sc.nextInt();  }    int[] prefix = new int [n];  prefix[0]=arr[0];  for(int i=1;i<n;i++){  prefix[i]=prefix[i-1]+arr[i];  }  int[] suffix = new int [n];  suffix[n-1]=arr[n-1];  for(int i=n-2;i>=0;i--){  suffix[i]=suffix[i+1]+arr[i];  }  for(int i=0;i<n-1;i++){  if(prefix[i]==suffix[i+1]){  System.out.print("Array should be split after index : "+i);  }  }  }  } |

Time complexity – O(n)

Space complexity – O(n)

Example –

[1,2,3,4,10]

Prefix : [1,3,6,10,20]

Suffix : [16,17,14,10]

Answer : array should be split after index = 3 (prefix[i]=suffix[i+1] for I = 3)

1. **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.**

Sol.

|  |
| --- |
| class Solution {  int maxSubarraySum(int[] arr) {  int k=arr.length;  int max=arr[0];  int ans=max;  for(int i=1;i<k;i++){  max=Math.max(arr[i]+max,arr[i]);  ans=Math.max(max,ans);  }  return ans;  }  } |

Time complexity – O(n)

Space complexity – O(1)

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

Sol.

|  |
| --- |
| import java.util.Scanner;  class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter the string: ");  String s = sc.nextLine();    int freq[] = new int [256];  int start =0; int maxlength=0;    for(int i=0;i<s.length();i++){  char c = s.charAt(i);  freq[c]++;    while(freq[c]>1){  freq[s.charAt(start)]--;  start++;  }  maxlength = Math.max(maxlength,i-start+1);  }  System.out.print("Max subarray : "+maxlength);  }  } |

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

Sol.

The sliding window technique is a way to efficiently process a subset of a sequence (like a string or array) without using nested loops.

Think of it like having a "window" that moves across a string, checking one portion (substring) at a time.

**Logic:**

1. Use two pointers: start and end
2. Start expanding the window with end
3. If there's a repeat character, shrink the window from the start
4. Keep track of the maximum window size
5. **Find the longest palindromic substring in a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.**

Sol.

|  |
| --- |
| import java.util.Scanner;  class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter the string: ");  String s = sc.nextLine();  int start =0,maxlen = 0;  for(int i=0;i<s.length();i++){  int len1=palindrome(s,i,i);  int len2=palindrome(s,i,i+1);  int len =Math.max(len1,len2);  if(len>maxlen){  maxlen=len;  start = i-(len-1)/2;    }  }  String longest = s.substring(start,start+maxlen);  System.out.println("Longest Palindromic Substring:"+longest);  }  static int palindrome(String s,int left, int right){  while(left>=0&&right<s.length()&&s.charAt(left)==s.charAt(right)){  left--;  right++;  }  return right-left-1;  }  } |

Time complexity – O(n2)

Space complexity – O(1)

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

Sol.

|  |
| --- |
| import java.util.Scanner;  public class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter number of strings: ");  int n = sc.nextInt();  sc.nextLine();  String[] strs = new String[n];  System.out.println("Enter the strings:");  for (int i = 0; i < n; i++) {  strs[i] = sc.nextLine();  }  String result = longestCommonPrefix(strs);  System.out.println("Longest Common Prefix: " + result);  }  static String longestCommonPrefix(String[] strs) {  if (strs == null || strs.length == 0) return "";  for (int i = 0; i < strs[0].length(); i++) {  char ch = strs[0].charAt(i);  for (int j = 1; j < strs.length; j++) {  // Mismatch found or index exceeds current word  if (i >= strs[j].length() || strs[j].charAt(i) != ch) {  return strs[0].substring(0, i);  }  }  }  return strs[0];  }  } |

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

Sol.

|  |
| --- |
| import java.util.Scanner;  public class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.print("Enter the string: ");  String str = sc.nextLine();  System.out.println("All permutations of the string are:");  generatePermutations(str.toCharArray(), 0);  }  static void generatePermutations(char[] chars, int index) {  if (index == chars.length - 1) {  System.out.println(new String(chars));  return;  }  for (int i = index; i < chars.length; i++) {  swap(chars, index, i);  generatePermutations(chars, index + 1);  swap(chars, index, i);  }  }  static void swap(char[] arr, int i, int j) {  char temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  } |

1. **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.**

Sol.

|  |
| --- |
| import java.util.Scanner;  class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int n = sc.nextInt();  int[] arr = new int[n];  for(int i = 0; i < n; i++)  arr[i] = sc.nextInt();  int target = sc.nextInt();  int left = 0, right = n - 1;  while(left < right) {  int sum = arr[left] + arr[right];  if(sum == target) {  System.out.println("Indices: " + left + " and " + right);  return;  }  if(sum < target)  left++;  else  right--;  }    System.out.println("No pair found");  }  } |

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

Sol.

|  |
| --- |
| import java.util.\*;  public class Main {  public static void main(String[] args) {  int[] arr = {1, 2, 3};  nextPermutation(arr);  System.out.println("Next permutation:");  for (int num : arr)  System.out.print(num + " ");  }  public static void nextPermutation(int[] arr) {  int n = arr.length;  int i = n - 2;  while (i >= 0 && arr[i] >= arr[i + 1]) {  i--;  }  if (i >= 0) {  int j = n - 1;  while (arr[j] <= arr[i]) {  j--;  }  swap(arr, i, j);  }  reverse(arr, i + 1, n - 1);  }  public static void swap(int[] arr, int i, int j) {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  public static void reverse(int[] arr, int start, int end) {  while (start < end) {  swap(arr, start++, end--);  }  }  } |

1. **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.**

Sol.

|  |
| --- |
| class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; }  }  public class Main {  public static ListNode merge(ListNode l1, ListNode l2) {  ListNode dummy = new ListNode(-1);  ListNode current = dummy;  while (l1 != null && l2 != null) {  if (l1.val < l2.val) {  current.next = l1;  l1 = l1.next;  } else {  current.next = l2;  l2 = l2.next;  }  current = current.next;  }  if (l1 != null) current.next = l1;  if (l2 != null) current.next = l2;  return dummy.next;  }  } |

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

Sol.

|  |
| --- |
| public class Main {  public static double findMedianSortedArrays(int[] A, int[] B) {  if (A.length > B.length) return findMedianSortedArrays(B, A);  int m = A.length, n = B.length, low = 0, high = m;  while (low <= high) {  int i = (low + high) / 2;  int j = (m + n + 1) / 2 - i;  int maxLeftA = (i == 0) ? Integer.MIN\_VALUE : A[i - 1];  int minRightA = (i == m) ? Integer.MAX\_VALUE : A[i];  int maxLeftB = (j == 0) ? Integer.MIN\_VALUE : B[j - 1];  int minRightB = (j == n) ? Integer.MAX\_VALUE : B[j];  if (maxLeftA <= minRightB && maxLeftB <= minRightA) {  if ((m + n) % 2 == 0)  return (Math.max(maxLeftA, maxLeftB) + Math.min(minRightA,minRightB));  else  return Math.max(maxLeftA, maxLeftB);  } else if (maxLeftA > minRightB) {  high = i - 1;  } else {  low = i + 1;  }  }  return 0.0;  }  public static void main(String[] args) {  int[] nums1 = {1, 3};  int[] nums2 = {2};  System.out.println("Median: " + findMedianSortedArrays(nums1, nums2));  }  } |

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

Sol.

|  |
| --- |
| public class Main {  public static int kthSmallest(int[][] matrix, int k) {  int n = matrix.length;  int low = matrix[0][0], high = matrix[n - 1][n - 1];  while (low < high) {  int mid = (low + high) / 2;  int count = countLessEqual(matrix, mid);  if (count < k)  low = mid + 1;  else  high = mid;  }  return low;  }  private static int countLessEqual(int[][] matrix, int mid) {  int n = matrix.length;  int count = 0, row = n - 1, col = 0;  while (row >= 0 && col < n) {  if (matrix[row][col] <= mid) {  count += row + 1;  col++;  } else {  row--;  }  }  return count;  }  public static void main(String[] args) {  int[][] matrix = {{1, 5, 9},{10, 11, 13},{12, 13, 15}};  int k = 8;  System.out.println("K-th Smallest Element: " + kthSmallest(matrix, k));  }  } |

1. **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.**

Sol.

|  |
| --- |
| public class Main {  public static int majorityElement(int[] nums) {  int count = 0, candidate = 0;  for (int num : nums) {  if (count == 0) {  candidate = num;  }  count += (num == candidate) ? 1 : -1;  }  return candidate;  }  public static void main(String[] args) {  int[] nums = {2, 2, 1, 1, 1, 2, 2};  System.out.println("Majority Element: " + majorityElement(nums));  }  } |

1. **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.**

Sol.

|  |
| --- |
| public class Main {  public static int trap(int[] height) {  int left = 0, right = height.length - 1;  int leftMax = 0, rightMax = 0;  int water = 0;  while (left < right) {  if (height[left] < height[right]) {  if (height[left] >= leftMax) {  leftMax = height[left];  } else {  water += leftMax - height[left];  }  left++;  } else {  if (height[right] >= rightMax) {  rightMax = height[right];  } else {  water += rightMax - height[right];  }  right--;  }  }  return water;  }  public static void main(String[] args) {  int[] height = {0,1,0,2,1,0,1,3,2,1,2,1};  System.out.println("Water trapped: " + trap(height));  }  } |

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

|  |
| --- |
| import java.util.\*;  public class Main {  public static int findMaximumXOR(int[] nums) {  int max = 0, mask = 0;  for (int i = 31; i >= 0; i--) {  mask = mask | (1 << i);  Set<Integer> set = new HashSet<>();  for (int num : nums) {  set.add(num & mask);  }  int temp = max | (1 << i);  for (int prefix : set) {  if (set.contains(prefix ^ temp)) {  max = temp;  break;  }  }  }  return max;  }  public static void main(String[] args) {  int[] nums = {3, 10, 5, 25, 2, 8};  System.out.println("Maximum XOR is: " + findMaximumXOR(nums));  }  } |

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

|  |
| --- |
| public class Main {  public static int maxProduct(int[] nums) {  if (nums.length == 0) return 0;  int maxProd = nums[0], minProd = nums[0], result = nums[0];  for (int i = 1; i < nums.length; i++) {  if (nums[i] < 0) {  int temp = maxProd;  maxProd = minProd;  minProd = temp;  }  maxProd = Math.max(nums[i], maxProd \* nums[i]);  minProd = Math.min(nums[i], minProd \* nums[i]);  result = Math.max(result, maxProd);  }  return result;  }  public static void main(String[] args) {  int[] nums = {2, 3, -2, 4};  System.out.println("Maximum Product Subarray: " + maxProduct(nums));  }  } |

1. **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.**

|  |
| --- |
| public class Main {  public static int countNumbersWithUniqueDigits(int n) {  if (n == 0) return 1;  if (n == 1) return 10;  int result = 10,product = 9;  for (int i = 2; i <= n; i++) {  product \*= (11 - i);  result += product;  }  return result;  }  public static void main(String[] args) {  int n = 2;  System.out.println("Count of unique numbers with " + n + " digits: " + countNumbersWithUniqueDigits(n));  }  } |

1. **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.**

|  |
| --- |
| public class Main {  public static int countSetBits(int n) {  int count = 0;  for (int i = 0; i <= n; i++) {  count += countOnes(i);  }  return count;  }  public static int countOnes(int num) {  int count = 0;  while (num > 0) {  num = num & (num - 1);  count++;  }  return count;  }  public static void main(String[] args) {  int n = 5;  System.out.println("Total number of 1's from 0 to " + n + " is: " + countSetBits(n));  }  } |

1. **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.**

|  |
| --- |
| public class Main {=  public static boolean isPowerOfTwo(int n) {  return (n > 0) && (n & (n - 1)) == 0;  }  public static void main(String[] args) {  int n = 16;  if (isPowerOfTwo(n)) {  System.out.println(n + " is a power of two.");  } else {  System.out.println(n + " is not a power of two.");  }  }  } |

1. **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.**

|  |
| --- |
| public class Main {  public static int findMaximumXOR(int[] nums) {  int maxXOR = 0, mask = 0;  for (int i = 31; i >= 0; i--) {  mask |= (1 << i);  Set<Integer> prefixes = new HashSet<>();  for (int num : nums) {  prefixes.add(num & mask);  }  int candidate = maxXOR | (1 << i);  for (int prefix : prefixes) {  if (prefixes.contains(prefix ^ candidate)) {  maxXOR = candidate;  break;  }  }  }  return maxXOR;  }  public static void main(String[] args) {  int[] nums = {3, 10, 5, 25, 2, 8};  System.out.println("Maximum XOR: " + findMaximumXOR(nums));  }  } |

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

Sol.

Bit manipulation refers to the technique of performing operations directly on the individual bits of a number. This is done using bitwise operators, which allow you to manipulate data at the binary level. These operators are efficient and are often used to solve problems related to binary representation and optimization.

**Key Bitwise Operators:**

1. **AND (&)**: Compares each bit of two numbers and returns 1 if both corresponding bits are 1, otherwise 0.
   * Example: 5 & 3 → 0101 & 0011 → 0001 (Result is 1).
2. **OR (|)**: Compares each bit of two numbers and returns 1 if at least one of the corresponding bits is 1.
   * Example: 5 | 3 → 0101 | 0011 → 0111 (Result is 7).
3. **XOR (^)**: Compares each bit of two numbers and returns 1 if the bits are different, otherwise 0.
   * Example: 5 ^ 3 → 0101 ^ 0011 → 0110 (Result is 6).
4. **NOT (~)**: Inverts each bit of the number (flips 0 to 1 and 1 to 0).
   * Example: ~5 → 1010 (Result depends on the size of the integer, typically it's the two's complement).
5. **Left Shift (<<)**: Shifts all the bits of a number to the left by a certain number of positions, effectively multiplying the number by 2^n.
   * Example: 5 << 1 → 0101 << 1 → 1010 (Result is 10).
6. **Right Shift (>>)**: Shifts all the bits of a number to the right by a certain number of positions, effectively dividing the number by 2^n and discarding bits on the right.
   * Example: 5 >> 1 → 0101 >> 1 → 0010 (Result is 2).
7. **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.**

|  |
| --- |
| import java.util.Stack;  public class Main {  public static void nextGreaterElement(int[] arr) {  int n = arr.length;  int[] result = new int[n];  Stack<Integer> stack = new Stack<>();  for (int i = n - 1; i >= 0; i--) {  while (!stack.isEmpty() && stack.peek() <= arr[i]) {  stack.pop();  }  if (!stack.isEmpty()) {  result[i] = stack.peek();  } else {  result[i] = -1;  }  stack.push(arr[i]);  }  System.out.print("Next Greater Element: ");  for (int i = 0; i < n; i++) {  System.out.print(result[i] + " ");  }  }  public static void main(String[] args) {  int[] arr = {4, 5, 2, 10, 8};  nextGreaterElement(arr);  }  } |

1. **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.**

|  |
| --- |
| class ListNode {  int val;  ListNode next;  ListNode(int val) {  this.val = val;  this.next = null;  }  }  public class Main {  public static ListNode removeNthFromEnd(ListNode head, int n) {  ListNode dummy = new ListNode(0);  dummy.next = head;  ListNode fast = dummy;  ListNode slow = dummy;  for (int i = 1; i <= n + 1; i++) {  fast = fast.next;  }  while (fast != null) {  fast = fast.next;  slow = slow.next;  }  slow.next = slow.next.next;  return dummy.next;  }  public static void printList(ListNode head) {  ListNode current = head;  while (current != null) {  System.out.print(current.val + " ");  current = current.next;  }  System.out.println();  }  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.println("Original List:");  printList(head);  int n = 2;  head = removeNthFromEnd(head, n);  System.out.println("List after removing " + n + "-th node from end:");  printList(head);  }  } |

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

|  |
| --- |
| class ListNode {  int val;  ListNode next;  ListNode(int val) {  this.val = val;  this.next = null;  }  }  public class Main {  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;  }  public static int getLength(ListNode head) {  int length = 0;  while (head != null) {  length++;  head = head.next;  }  return length;  }  public static void printList(ListNode head) {  while (head != null) {  System.out.print(head.val + " ");  head = head.next;  }  System.out.println();  }  public static void main(String[] args) {  ListNode list1 = new ListNode(1);  list1.next = new ListNode(2);  list1.next.next = new ListNode(3);  ListNode intersectNode = new ListNode(4);  list1.next.next.next = intersectNode;  intersectNode.next = new ListNode(5);  ListNode list2 = new ListNode(6);  list2.next = new ListNode(7);  list2.next.next = new ListNode(8);  list2.next.next.next = intersectNode;  ListNode intersection = getIntersectionNode(list1, list2);  if (intersection != null) {  System.out.println("Intersection at node with value: " + intersection.val);  } else {  System.out.println("No intersection found");  }  }  } |

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

|  |
| --- |
| class TwoStacks {  int[] arr;  int top1, top2;  public TwoStacks(int n) {  arr = new int[n];  top1 = -1;  top2 = n;  }  public void push1(int x) {  if (top1 + 1 < top2) {  arr[++top1] = x;  } else {  System.out.println("Stack Overflow");  }  }  public void push2(int x) {  if (top1 + 1 < top2) {  arr[--top2] = x;  } else {  System.out.println("Stack Overflow");  }  }  public int pop1() {  if (top1 >= 0) {  return arr[top1--];  } else {  System.out.println("Stack Underflow");  return -1;  }  }  public int pop2() {  if (top2 < arr.length) {  return arr[top2++];  } else {  System.out.println("Stack Underflow");  return -1;  }  }  public void display() {  System.out.print("Stack 1: ");  for (int i = 0; i <= top1; i++) {  System.out.print(arr[i] + " ");  }  System.out.println();  System.out.print("Stack 2: ");  for (int i = arr.length - 1; i >= top2; i--) {  System.out.print(arr[i] + " ");  }  System.out.println();  }  }  public class Main {  public static void main(String[] args) {  TwoStacks stacks = new TwoStacks(10);  stacks.push1(10);  stacks.push1(20);  stacks.push2(30);  stacks.push2(40);  stacks.display();  System.out.println("Popped from Stack 1: " + stacks.pop1());  System.out.println("Popped from Stack 2: " + stacks.pop2());  stacks.display();  }  } |

1. **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.**

|  |
| --- |
| public class Main {  public static void main(String[] args) {  int number = 121;  if (isPalindrome(number)) {  System.out.println(number + " is a palindrome.");  } else {  System.out.println(number + " is not a palindrome.");  }  }  public static boolean isPalindrome(int x) {  if (x < 0 || (x % 10 == 0 && x != 0)) {  return false;  }  int reversedHalf = 0;  while (x > reversedHalf) {  reversedHalf = reversedHalf \* 10 + x % 10;  x = x / 10;  }  return x == reversedHalf || x == reversedHalf / 10;  }  } |

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

Ans.

**Concept of Linked Lists:**

A **linked list** is a linear data structure where each element, called a **node**, contains two parts:

**Data**: The value or information stored in the node.

**Next**: A reference (or pointer) to the next node in the list.

The nodes are not stored in contiguous memory locations, and each node points to the next node, making it possible to efficiently insert and delete elements in the list without needing to shift elements as in arrays.

There are different types of linked lists:

**Singly Linked List**: Each node points to the next node, and the last node points to null.

**Doubly Linked List**: Each node points to both the next and the previous node, enabling traversal in both directions.

**Circular Linked List**: The last node points back to the first node, creating a circular structure.

**Applications of Linked Lists in Algorithm Design:**

**Dynamic Memory Allocation**: Used in operating systems and memory management to track free memory blocks.

**Implementing Other Data Structures**:

**Stacks and Queues**: Efficient for stack/queue operations at the beginning or end of the list.

**Graphs**: Used to represent adjacency lists in graph algorithms.

**Efficient Insertions and Deletions**: Ideal for problems with frequent data changes, like merging sorted lists or removing duplicates.

**Polynomial Arithmetic**: Represent polynomials efficiently, enabling operations like addition, multiplication, and evaluation.

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

Ans.

|  |
| --- |
| import java.util.Deque;  import java.util.LinkedList;  public static int[] maxSlidingWindow(int[] nums, int k) {  if (nums == null || nums.length == 0 || k == 0) return new int[0];  int n = nums.length;  int[] result = new int[n - k + 1];  Deque<Integer> deque = new LinkedList<>();  for (int i = 0; i < n; i++) {  if (!deque.isEmpty() && deque.peek() < i - k + 1) deque.poll();  while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {  deque.pollLast();  }  deque.offer(i);  if (i >= k - 1) result[i - k + 1] = nums[deque.peek()];  }  return result;  } |

**Algorithm**:

Use a **deque** to store indices of elements.

Maintain a decreasing order of elements in the deque.

For each window, remove out-of-bound indices and update the deque with the current element.

The element at the front of the deque is the maximum for the current window.

**Time Complexity**: O(n) (Each element is processed at most twice: once added, once removed)

**Space Complexity**: O(k) (The deque stores at most k elements)

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

Ans.

|  |
| --- |
| import java.util.Stack;  public static int largestRectangleArea(int[] heights) {  Stack<Integer> stack = new Stack<>();  int maxArea = 0;  int index = 0;  while (index < heights.length) {  if (stack.isEmpty() || heights[index] >= heights[stack.peek()]) {  stack.push(index++);  } else {  int top = stack.pop();  int area = heights[top] \* (stack.isEmpty() ? index : index - stack.peek() - 1);  maxArea = Math.max(maxArea, area);  }  }  while (!stack.isEmpty()) {  int top = stack.pop();  int area = heights[top] \* (stack.isEmpty() ? index : index - stack.peek() - 1);  maxArea = Math.max(maxArea, area);  }  return maxArea;  } |

**Algorithm**:

Use a **stack** to store indices of the histogram bars.

Traverse the histogram; for each bar, if it's higher than the bar at the top of the stack, push its index onto the stack.

When a lower bar is found, calculate the area of the rectangle using the bar at the top of the stack as the smallest height.

Continue until all bars are processed, and then process any remaining bars in the stack.

**Time Complexity**: O(n)  
Each index is pushed and popped from the stack at most once.

**Space Complexity**: O(n)  
The stack can store up to n indices.

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

Ans. **Purpose**: Efficiently solve problems involving contiguous subarrays or substrings.

**Types**:

**Fixed-size window**: Size K is constant.

**Variable-size window**: Size changes based on conditions.

**Key Applications:**

**Max/Min Sum Subarray of Size K**

**Longest Substring Without Repeating Characters**

**Find All Anagrams in a String**

**Subarrays with Sum Equal to K**

**Max Sliding Window**

**Algorithm:**

Use two pointers (start, end) to define the window.

Slide the end pointer, updating values.

Move the start pointer to maintain window size or conditions.

**Complexity:**

**Time**: O(n)

**Space**: O(k) (if using auxiliary structures)

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

Ans.

|  |
| --- |
| import java.util.HashMap;  public static int subarraySum(int[] nums, int k) {  int count = 0;  int currentSum = 0;  HashMap<Integer, Integer> map = new HashMap<>();  map.put(0, 1  for (int num : nums) {  currentSum += num;  if (map.containsKey(currentSum - k)) {  count += map.get(currentSum - k);  }  map.put(currentSum, map.getOrDefault(currentSum, 0) + 1);  }  return count;  } |

**Time Complexity**: O(n)

**Space Complexity**: O(n)

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

Ans.

|  |
| --- |
| import java.util.\*;  public static int[] 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>> minHeap = new PriorityQueue<>(  (a, b) -> a.getValue() - b.getValue()  );  for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {  minHeap.add(entry);  if (minHeap.size() > k) {  minHeap.poll();  }  }  int[] result = new int[k];  int index = 0;  while (!minHeap.isEmpty()) {  result[index++] = minHeap.poll().getKey();  }  return result;  } |

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

**Space Complexity**: O(n)

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

Ans.

|  |
| --- |
| public static List<List<Integer>> subsets(int[] nums) {  List<List<Integer>> result = new ArrayList<>();  int n = nums.length;  int total = 1 << n;  for (int i = 0; i < total; i++) {  List<Integer> subset = new ArrayList<>();  for (int j = 0; j < n; j++) {  if ((i & (1 << j)) != 0) {  subset.add(nums[j]);  }  }  result.add(subset);  }  return result;  } |

**Time Complexity**: O(n \* 2^n)

**Space Complexity**: O(n \* 2^n)

**37.Find all unique combinations of numbers that sum to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.**

Ans.

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

**Time Complexity**: O(2^t) where t = target (exponential due to recursive exploration) **Space Complexity**: O(t)

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

Ans. **Algorithm (Backtracking)**

Start with the current index.

Swap each element from current index to end with the current index.

Recur for the next index.

Backtrack (undo the swap) after each recursion.

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

**Space Complexity**: O(n)

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

Ans.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Subsets** | **Permutations** |
| **Definition** | **Selection of elements (order doesn't matter)** | **Arrangement of elements (order matters)** |
| **Count** | **2n2^n2n subsets for n elements** | **n!n!n! permutations for n elements** |
| **Duplicates** | **No duplicates in subset list (unless input has)** | **All unique arrangements are included** |
| **Order** | **Irrelevant** | **Very important** |
| **Use Case** | **Combinations, power set, feature selection** | **Ordering, sequences, routes** |

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

Ans.

|  |
| --- |
| public static int findMaxFrequencyElement(int[] nums) {  Map<Integer, Integer> freqMap = new HashMap<>();    for (int num : nums) {  freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);  }  int maxFreq = 0;  int maxElement = -1;    for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {  if (entry.getValue() > maxFreq) {  maxFreq = entry.getValue();  maxElement = entry.getKey();  }  }    return maxElement;  } |

**Time Complexity**: O(n)

**Space Complexity**: O(n)

**41. Maximum Subarray Sum using Kadane’s Algorithm**

|  |
| --- |
| public class KadaneAlgorithm {  public static int maxSubArraySum(int[] nums) { int maxSum = nums[0], currSum = nums[0]; for (int i = 1; i < nums.length; i++) {  currSum = Math.max(nums[i], currSum + nums[i]); maxSum = Math.max(maxSum, currSum);  }  return maxSum;  }  } |

**Time:** O(n), **Space:** O(1)

**42. Dynamic Programming & Maximum Subarray**

|  |
| --- |
| public class MaxSubarrayDP {  public static int maxSubArray(int[] nums) { int[] dp = new int[nums.length];  dp[0] = nums[0]; int max = dp[0];  for (int i = 1; i < nums.length; i++) {  dp[i] = Math.max(nums[i], dp[i-1] + nums[i]); max = Math.max(max, dp[i]);  }  return max;  }  } |

**Time:** O(n), **Space:** O(n)

**43. Top K Frequent Elements**

|  |
| --- |
|  |

**Time:** O(n log k), **Space:** O(n)

**44. Two Sum Using Hashing**

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

**Time:** O(n), **Space:** O(n)

**45. Priority Queue Concept**

|  |
| --- |
| import java.util.PriorityQueue;  public class PriorityQueueExample {  public static void main(String[] args) { PriorityQueue<Integer> pq = new PriorityQueue<>(); pq.add(5);  pq.add(1);  pq.add(10); System.out.println(pq.poll()); // 1  }  } |

**46. Longest Palindromic Substring**

|  |
| --- |
| public class LongestPalindrome {  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 = expand(s, i, i);  int len2 = expand(s, i, i + 1); int len = Math.max(len1, len2); if (len > end - start) {  start = i - (len - 1) / 2; end = i + len / 2;  }  }  return s.substring(start, end + 1);  }  private static int expand(String s, int left, int right) { while (left >= 0 && right < s.length() && s.charAt(left) ==  s.charAt(right)) {  left--; right++;  }  return right - left - 1;  }  } |

**Time:** O(n²), **Space:** O(1)

**47. Histogram Problems**

Explanation: Common problems include:

• Largest rectangle area in histogram

• Water trapped between bars

Techniques: Stack (for area), Two pointers (for water)

**48. Next Permutation**

|  |
| --- |
| public class NextPermutation {  public static void nextPermutation(int[] nums) { int i = nums.length - 2;  while (i >= 0 && nums[i] >= nums[i + 1]) i--;  if (i >= 0) {  int j = nums.length - 1;  while (nums[j] <= nums[i]) j--; swap(nums, i, j);  }  reverse(nums, i + 1);  }  private static void swap(int[] nums, int i, int j) {  int temp = nums[i]; nums[i] = nums[j]; nums[j] = temp;  }  private static void reverse(int[] nums, int start) { int end = nums.length - 1;  while (start < end) swap(nums, start++, end--);  }  } |

**Time:** O(n), **Space:** O(1)

**49. Intersection of Two Linked Lists**

|  |
| --- |
| public class Intersection { static class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; next = null; }  }  public ListNode getIntersectionNode(ListNode headA, ListNode headB) { ListNode a = headA, b = headB;  while (a != b) {  a = (a == null) ? headB : a.next;  b = (b == null) ? headA : b.next;  }  return a;  }  } |

**Time:** O(n), **Space:** O(1)

**50. Equilibrium Index**

|  |
| --- |
| public class EquilibriumIndex {  public static int findEquilibrium(int[] arr) { int total = 0, leftSum = 0;  for (int num : arr) total += num;  for (int i = 0; i < arr.length; i++) { total -= arr[i];  if (leftSum == total) return i; leftSum += arr[i];  }  return -1;  }  } |

**Time:** O(n), **Space:** O(1)