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

A prefix sum array is another array **prefixSum[]** of the same size, such that **prefixSum[i]** is **arr[0] + arr[1] + arr[2] . . . arr[i]**.

**Example:**

***Input:*** *arr[] = [10, 20, 10]****Output:*** *10 30 40   
prefixSum[0] = 10,   
prefixSum[1] = 10 + 20 = 30,   
prefixSum[2] = 10 + 20 + 10 = 40*

*Applications.*

*Range sum queries.*

*Subarray sum.*

*Competitive programming and data analytics*

*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.*

*Algorithms -:*

*a. Take input array arr of size n*

*b. Build a prefix sum array:*

* *prefix[0] = arr[0]*
* *prefix[i] = prefix[i - 1] + arr[i]*

*c.*  *If L == 0, return prefix[R]*

*Else, return prefix[R] - prefix[L - 1]*

*Code -:*

*public class PrefixSum {*

*public static int[] buildPrefixSum(int[] arr) {*

*int n = arr.length;*

*int[] prefix = new int[n];*

*prefix[0] = arr[0];*

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

*prefix[i] = prefix[i - 1] + arr[i];*

*}*

*return prefix;*

*}*

*public static int rangeSum(int[] prefix, int L, int R) {*

*if (L == 0) {*

*return prefix[R];*

*} else {*

*return prefix[R] - prefix[L - 1];*

*}*

*}*

*Time complexity – O(n)*

*Space complexity – O(n)*

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

*Algorithm -:*

1. *Calculate the* ***total sum*** *of the array.*
2. *Initialize a variable leftSum = 0.*
3. *Traverse the array from left to right:*
   * *For each index i, the* ***right sum*** *is totalSum - leftSum - arr[i].*
   * *If leftSum == rightSum, return index i (equilibrium index).*
   * *Otherwise, add arr[i] to leftSum and continue.*

*Code -:*

*public class EquilibriumIndex {*

*public static int findEquilibriumIndex(int[] arr) {*

*int totalSum = 0;*

*for (int num : arr) {*

*totalSum += num;*

*}*

*int leftSum = 0;*

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

*int rightSum = totalSum - leftSum - arr[i];*

*if (leftSum == rightSum) {*

*return i;*

*}*

*leftSum += arr[i];*

*}*

*return -1;*

*}*

*Time complexity – O(n)*

*Space complexity – O(1)*

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.*

*Algorithm*-

a Calculate the **total sum** of the array.

b Initialize leftSum = 0

c Traverse the array up to n-1 (not including last element):

* Add arr[i] to leftSum
* Compute rightSum = totalSum - leftSum
* If leftSum == rightSum, return true

d If loop ends, return false

Code -:

public class SplitEqualSum {

public static boolean canBeSplit(int[] arr) {

int totalSum = 0;

for (int num : arr) {

totalSum += num;

}

int leftSum = 0;

for (int i = 0; i < arr.length - 1; i++) {

leftSum += arr[i];

int rightSum = totalSum - leftSum;

if (leftSum == rightSum) {

return true;

}

}

return false;

}

Time complexity – O(n)

Space complexity- O(1)

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.

Algorithm-:

a. Calculate the sum of the first K elements. Store it as windowSum.

b. Initialize maxSum = windowSum.

c. Slide the window from index K to n-1:

* Subtract arr[i-K] (element leaving the window)
* Add arr[i] (element entering the window)
* Update maxSum if windowSum is greater

d. Return maxSum

Code-:

public class MaxSumSubarrayK {

public static int maxSum(int[] arr, int k) {

if (arr.length < k) {

System.out.println("Invalid: Array size less than K");

return -1;

}

int windowSum = 0;

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

windowSum += arr[i];

}

int maxSum = windowSum;

// Step 2: Slide the window

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

windowSum += arr[i] - arr[i - k];

maxSum = Math.max(maxSum, windowSum);

}

return maxSum;

}

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.

Algorithm-

a. Use two pointers left and right to represent the window.

b. Use a HashSet to store characters in the current window.

c. Slide right to expand the window:

* If s[right] is not in the set, add it and update the max length.
* If it is, remove s[left] and move left forward until the character is removed.

d. Continue until right reaches the end of the string.

Code

class LongestUniqueSubstring {

static int lengthOfLongestSubstring(String s) {

int[] lastIndex = new int[256];

for (int i = 0; i < 256; i++) lastIndex[i] = -1;

int maxLength = 0, start = 0;

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

if (lastIndex[s.charAt(end)] >= start) {

start = lastIndex[s.charAt(end)] + 1;

}

lastIndex[s.charAt(end)] = end;

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

}

return maxLength;

}

}

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

Sliding window is a technique where we maintain a subset (window) of the data and slide it across the array/string.

Useful for problems requiring contiguous subarrays or substrings.

Examples:

- Longest substring without repeating characters

- Max sum of subarray of size K

- Anagram detection in string

Time Complexity: O(n), Space: O(1)/O(k)

8. Longest palindromic substring

class LongestPalindrome {

static String longestPalindrome(String s) {

if (s.length() < 1) return "";

int start = 0, end = 0;

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

int len1 = expandFromMiddle(s, i, i);

int len2 = expandFromMiddle(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);

}

static int expandFromMiddle(String s, int left, int right) {

while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {

left--;

right++;

}

return right - left - 1;

}

}

Algorithm: Expand Around Center

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

Example: "babad" -> "bab" or "aba"

9. Longest common prefix among strings

class LongestCommonPrefix {

static String longestCommonPrefix(String[] strs) {

if (strs.length == 0) return "";

String prefix = strs[0];

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

while (strs[i].indexOf(prefix) != 0) {

prefix = prefix.substring(0, prefix.length() - 1);

if (prefix.isEmpty()) return "";

}

}

return prefix;

}

}

Algorithm:

1. Start with first string as prefix.

2. Trim prefix if next string doesn't start with it.

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

10. Generate all permutations of a string

class StringPermutations {

static void permute(String str, int l, int r) {

if (l == r)

System.out.println(str);

else {

for (int i = l; i <= r; i++) {

str = swap(str, l, i);

permute(str, l + 1, r);

str = swap(str, l, i);

}

}

}

static String swap(String a, int i, int j) {

char[] charArray = a.toCharArray();

char temp = charArray[i];

charArray[i] = charArray[j];

charArray[j] = temp;

return String.valueOf(charArray);

}

}

Backtracking approach

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

Example: abc -> abc, acb, bac, bca, cab, cba

11. Find two numbers in a sorted array that add up to a target

class TwoSumSorted {

static int[] twoSum(int[] nums, int target) {

int left = 0, right = nums.length - 1;

while (left < right) {

int sum = nums[left] + nums[right];

if (sum == target) return new int[]{left, right};

if (sum < target) left++;

else right--;

}

return new int[]{-1, -1};

}

}

Two-pointer approach

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

12. Lexicographically next greater permutation

class NextPermutation {

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);

}

static void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

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)

13. Merge two sorted linked lists

class MergeSortedLists {

static class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

static ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(-1), curr = dummy;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

curr.next = l1; l1 = l1.next;

} else {

curr.next = l2; l2 = l2.next;

}

curr = curr.next;

}

curr.next = (l1 != null) ? l1 : l2;

return dummy.next;

}

}

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

14. Median of two sorted arrays using binary search

class MedianSortedArrays {

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

if (nums1.length > nums2.length) return findMedianSortedArrays(nums2, nums1);

int x = nums1.length, y = nums2.length;

int low = 0, high = x;

while (low <= high) {

int partitionX = (low + high) / 2;

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

int maxX = (partitionX == 0) ? Integer.MIN\_VALUE : nums1[partitionX - 1];

int minX = (partitionX == x) ? Integer.MAX\_VALUE : nums1[partitionX];

int maxY = (partitionY == 0) ? Integer.MIN\_VALUE : nums2[partitionY - 1];

int minY = (partitionY == y) ? Integer.MAX\_VALUE : nums2[partitionY];

if (maxX <= minY && maxY <= minX) {

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

return ((double)Math.max(maxX, maxY) + Math.min(minX, minY)) / 2;

else

return (double)Math.max(maxX, maxY);

} else if (maxX > minY) {

high = partitionX - 1;

} else {

low = partitionX + 1;

}

}

throw new IllegalArgumentException();

}

}

Time: O(log(min(n, m))), Space: O(1)

15. Find the k-th smallest element in a sorted matrix

import java.util.PriorityQueue;

class KthSmallestInMatrix {

static class Node implements Comparable<Node> {

int val, r, c;

Node(int v, int r, int c) { this.val = v; this.r = r; this.c = c; }

public int compareTo(Node o) { return this.val - o.val; }

}

static int kthSmallest(int[][] matrix, int k) {

int n = matrix.length;

PriorityQueue<Node> pq = new PriorityQueue<>();

for (int i = 0; i < n; i++) pq.offer(new Node(matrix[i][0], i, 0));

while (--k > 0) {

Node node = pq.poll();

if (node.c < n - 1)

pq.offer(new Node(matrix[node.r][node.c + 1], node.r, node.c + 1));

}

return pq.peek().val;

}

}

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

16. Find the majority element (appears more than n/2 times)

class MajorityElement {

static int majorityElement(int[] nums) {

int count = 0, candidate = -1;

for (int num : nums) {

if (count == 0) {

candidate = num;

count = 1;

} else if (candidate == num) {

count++;

} else {

count--;

}

}

return candidate;

}

}

Moore’s Voting Algorithm

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

17. Trapping Rain Water

class TrappingRainWater {

static int trap(int[] height) {

int left = 0, right = height.length - 1;

int leftMax = 0, rightMax = 0, result = 0;

while (left < right) {

if (height[left] < height[right]) {

if (height[left] >= leftMax)

leftMax = height[left];

else

result += leftMax - height[left];

left++;

} else {

if (height[right] >= rightMax)

rightMax = height[right];

else

result += rightMax - height[right];

right--;

}

}

return result;

}

}

Two-pointer approach

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

18. Maximum XOR of two numbers in an array

class MaximumXOR {

static int findMaximumXOR(int[] nums) {

int max = 0, mask = 0;

Set<Integer> set = new HashSet<>();

for (int i = 31; i >= 0; i--) {

mask |= (1 << i);

set.clear();

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;

}

}

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

19. Maximum Product Subarray

class MaxProductSubarray {

static int maxProduct(int[] nums) {

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;

}

}

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

20. Count numbers with unique digits (for given number of digits)

class CountUniqueDigits {

static int countNumbersWithUniqueDigits(int n) {

if (n == 0) return 1;

int res = 10, uniqueDigits = 9, available = 9;

while (n-- > 1 && available > 0) {

uniqueDigits \*= available;

res += uniqueDigits;

available--;

}

return res;

}

}

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

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

class CountBits {

static int[] countBits(int n) {

int[] res = new int[n + 1];

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

res[i] = res[i >> 1] + (i & 1);

}

return res;

}

}

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

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

class PowerOfTwo {

static boolean isPowerOfTwo(int n) {

return n > 0 && (n & (n - 1)) == 0;

}

}

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

23. Maximum XOR of two numbers

class MaximumXOR {

static int findMaximumXOR(int[] nums) {

int max = 0, mask = 0;

Set<Integer> set = new HashSet<>();

for (int i = 31; i >= 0; i--) {

mask |= (1 << i);

set.clear();

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;

}

}

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

24. Concept of Bit Manipulation

Bit Manipulation involves using bitwise operators (&, |, ^, ~, <<, >>) for optimizing algorithms.

Advantages:

- Faster computation

- Memory efficiency

- Useful in masks, toggling bits, and solving complex problems like subset generation.

25. Next greater element for each element in an array

class NextGreaterElement {

static int[] nextGreaterElements(int[] nums) {

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

int[] res = new int[nums.length];

Arrays.fill(res, -1);

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

while (!stack.isEmpty() && nums[stack.peek()] < nums[i]) {

res[stack.pop()] = nums[i];

}

stack.push(i);

}

return res;

}

}

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

26. Remove the n-th node from end of a singly linked list

class RemoveNthNode {

static class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

static ListNode removeNthFromEnd(ListNode head, int n) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode first = dummy, second = dummy;

for (int i = 0; i <= n; i++) first = first.next;

while (first != null) {

first = first.next;

second = second.next;

}

second.next = second.next.next;

return dummy.next;

}

}

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

**2**7. Find the node where two singly linked lists intersect

class LinkedListIntersection {

static RemoveNthNode.ListNode getIntersectionNode(RemoveNthNode.ListNode headA, RemoveNthNode.ListNode headB) {

RemoveNthNode.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)

28. Implement two stacks in a single array

class TwoStacks {

int size, top1, top2;

int[] arr;

TwoStacks(int n) {

size = n;

arr = new int[n];

top1 = -1;

top2 = n;

}

void push1(int x) {

if (top1 < top2 - 1) arr[++top1] = x;

}

void push2(int x) {

if (top1 < top2 - 1) arr[--top2] = x;

}

int pop1() {

if (top1 >= 0) return arr[top1--];

return -1;

}

int pop2() {

if (top2 < size) return arr[top2++];

return -1;

}

}

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

29. Check if integer is palindrome without converting to string

class PalindromeNumber {

static boolean isPalindrome(int x) {

if (x < 0 || (x % 10 == 0 && x != 0)) return false;

int reversed = 0;

while (x > reversed) {

reversed = reversed \* 10 + x % 10;

x /= 10;

}

return x == reversed || x == reversed / 10;

}

}

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

31. Maximum in every sliding window of size K using deque

class MaxSlidingWindow {

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

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

int[] result = new int[nums.length - k + 1];

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

while (!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;

}

}

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

32. Largest rectangle in histogram

class LargestRectangleHistogram {

static int largestRectangleArea(int[] heights) {

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

int maxArea = 0;

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

int h = (i == heights.length) ? 0 : heights[i];

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

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

int width = stack.isEmpty() ? i : i - stack.peek() - 1;

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

}

stack.push(i);

}

return maxArea;

}

}

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

34. Subarray sum equal to K using hashing

class SubarraySumEqualsK {

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

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

map.put(0, 1);

int sum = 0, count = 0;

for (int num : nums) {

sum += num;

count += map.getOrDefault(sum - k, 0);

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

}

return count;

}

}

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

36. Generate all subsets of a given array

class Subsets {

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

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

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

return result;

}

static void backtrack(List<List<Integer>> result, List<Integer> temp, int[] nums, int start) {

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

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

temp.add(nums[i]);

backtrack(result, temp, nums, i + 1);

temp.remove(temp.size() - 1);

}

}

}

Time: O(2^n), Space: O(n)

37. All unique combinations that sum to a target

class CombinationSum {

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

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

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

return result;

}

static void backtrack(List<List<Integer>> result, List<Integer> temp, int[] candidates, int remain, int start) {

if (remain == 0) result.add(new ArrayList<>(temp));

else if (remain > 0) {

for (int i = start; i < candidates.length; i++) {

temp.add(candidates[i]);

backtrack(result, temp, candidates, remain - candidates[i], i);

temp.remove(temp.size() - 1);

}

}

}

}

Time: O(2^n), Space: O(n)

38. Generate all permutations of a given array

class ArrayPermutations {

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

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

backtrack(result, new ArrayList<>(), nums);

return result;

}

static void backtrack(List<List<Integer>> result, List<Integer> temp, int[] nums) {

if (temp.size() == nums.length) {

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

} else {

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

if (temp.contains(nums[i])) continue;

temp.add(nums[i]);

backtrack(result, temp, nums);

temp.remove(temp.size() - 1);

}

}

}

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

40. Element with maximum frequency in array

class MaxFrequencyElement {

static int maxFrequency(int[] nums) {

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

int maxCount = 0, res = nums[0];

for (int num : nums) {

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

if (map.get(num) > maxCount) {

maxCount = map.get(num);

res = num;

}

}return res;

}

}

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

41. Program to Find the Maximum Subarray Sum using Kadane’s Algorithm in Java

public class KadaneAlgorithm {

public static int kadane(int[] arr) {

int maxSoFar = Integer.MIN\_VALUE;

int maxEndingHere = 0;

for (int num : arr) {

maxEndingHere = Math.max(num, maxEndingHere + num);

maxSoFar = Math.max(maxSoFar, maxEndingHere);

}

return maxSoFar;

}

public static void main(String[] args) {

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

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

}

}

**Time Complexity:** O(n), where n is the number of elements in the array.

**Space Complexity:** O(1), as we only use a constant amount of extra space.

43. Solve the Problem of Finding the Top K Frequent Elements in an Array

**Algorithm:**

1. **Count the frequency** of each element in the array.
2. Use a **min-heap** (or priority queue) to store the k most frequent elements.
3. Pop elements from the heap to maintain only the k most frequent elements.
4. Return the k elements from the heap.

**Program:**

import java.util.\*;

public class TopKFrequentElements {

public static List<Integer> topKFrequent(int[] nums, int k) {

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

for (int num : nums) {

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

}

PriorityQueue<Map.Entry<Integer, Integer>> minHeap =

new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());

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

minHeap.offer(entry);

if (minHeap.size() > k) {

minHeap.poll();

}

}

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

while (!minHeap.isEmpty()) {

result.add(minHeap.poll().getKey());

}

Collections.reverse(result); // Optional to get descending order

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("Top " + k + " frequent elements: " + result);

}

}

Time Complexity**:** O(n log k), where n is the number of elements in the array and k is the number of top elements we need to find.  
Space Complexity: O(n), where n is the number of unique elements in the array.

44. Find Two Numbers in an Array that Add Up to a Target Using Hashing

Algorithm:

1. Iterate through the array.
2. For each element, calculate its complement (target - current element).
3. If the complement exists in the hash set, return the pair.
4. Otherwise, add the current element to the hash set.

**Program:**

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 complement = target - nums[i];

if (map.containsKey(complement)) {

return new int[] { map.get(complement), i };

}

map.put(nums[i], i);

}

return new int[] {};

}

public static void main(String[] args) {

int[] nums = {2, 7, 11, 15};

int target = 9;

int[] result = twoSum(nums, target);

System.out.println("Indices of elements adding up to target: " + Arrays.toString(result));

}

}

**Time Complexity:** O(n), where n is the number of elements in the array.  
**Space Complexity:** O(n), as we are storing elements in a hash map.

45. Priority Queues and Their Applications in Algorithm Design

**Priority Queue** is a data structure where each element is associated with a priority, and the element with the highest (or lowest) priority is dequeued first. It is often implemented using heaps.

Applications in algorithm design:

1. **Dijkstra's shortest path algorithm** uses a priority queue to efficiently extract the minimum distance node.
2. **Huffman coding** for data compression uses a priority queue to build the optimal coding tree.
3. **Merge k sorted lists**: A priority queue can help merge multiple sorted lists in linear time.

**Q46. Program to Find the Longest Palindromic Substring**

**Algorithm:**

1. Use **expand around center** technique, where you check each character and expand outward to check for both odd and even length palindromes.
2. For each character, expand around it as the center and track the longest palindrome.

**Program:**

public class LongestPalindromicSubstring {

public static String longestPalindrome(String s) {

if (s == null || s.length() == 0) return "";

int start = 0, end = 0;

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

int len1 = expandAroundCenter(s, i, i); // Odd length palindrome

int len2 = expandAroundCenter(s, i, i + 1); // Even length palindrome

int len = Math.max(len1, len2);

if (len > (end - start)) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private static int expandAroundCenter(String s, int left, int right) {

while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {

left--;

right++;

}

return right - left - 1;

}

public static void main(String[] args) {

String s = "babad";

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

}

}

**Time Complexity:** O(n^2), where n is the length of the string.  
**Space Complexity:** O(1), as we are using only a constant amount of extra space.

48. Solve the Problem of Finding the Next Permutation of a Given Array

**Algorithm:**

1. Find the largest index i such that arr[i] < arr[i + 1]. If no such index exists, the array is the last permutation.
2. Find the largest index j such that arr[j] > arr[i].
3. Swap arr[i] and arr[j].
4. Reverse the subarray starting at arr[i + 1].

**Program:**

import java.util.Arrays;

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);

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), where n is the length of the array.  
**Space Complexity:** O(1), since we modify the array in-place.

49. Find the Intersection of Two Linked Lists

**Algorithm:**

1. Find the lengths of both linked lists.
2. Align the two lists by skipping the extra nodes in the longer list.
3. Traverse both lists in parallel and return the intersection node when found.

**Program:**

class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

public class IntersectionOfLinkedLists {

public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA == null || headB == null) return null;

ListNode a = headA;

ListNode b = headB;

while (a != b) {

a = (a == null) ? headB : a.next;

b = (b == null) ? headA : b.next;

}

return a;

}

public static void main(String[] args) {

ListNode headA = new ListNode(4);

headA.next = new ListNode(1);

ListNode intersection = new ListNode(8);

headA.next.next = intersection;

headB = new ListNode(5);

headB.next = intersection;

ListNode intersectionNode = getIntersectionNode(headA, headB);

System.out.println("Intersection at node: " + intersectionNode.val);

}

}

**Time Complexity:** O(n + m), where n and m are the lengths of the two linked lists.  
**Space Complexity:** O(1), as we are using constant space.