MTE – Assignment (AAPS)

Q1. Explain the concept of a prefix sum array and its applications.

A **prefix sum array** is an array that holds the cumulative sum of the elements of an original array up to each index. For an array arr[], its prefix sum array prefix[] is defined as:
prefix[0] = arr[0]
prefix[i] = prefix[i-1] + arr[i] for i > 0

Applications:

- Fast range sum queries: O(1) time for sum between two indices
- Solving subarray sum problems
- Used in 2D matrix sum queries
- Optimization in dynamic programming

Q2. Program to find the sum of elements in a given range [L, R] using prefix sum array.

Algorithm:

1. Construct prefix sum array.
2. To find sum from L to R:
   sum = prefix[R] - prefix[L-1] if L > 0, else sum = prefix[R]

Code in Java:

```
int rangeSum(int[] arr, int L, int R) {
   int[] prefix = new int[arr.length];
   prefix[0] = arr[0];
   for (int i = 1; i < arr.length; i++) {
      prefix[i] = prefix[i-1] + arr[i];
   }
   return (L == 0) ? prefix[R] : prefix[R] - prefix[L - 1];
}
```

Time Complexity: O(n) for prefix, O(1) per query
Space Complexity: O(n)

Example:
arr = [2, 4, 1, 5, 3], L = 1, R = 3
prefix = [2, 6, 7, 12, 15]
sum = prefix[3] - prefix[0] = 12 - 2 = 10

Q3. Find the equilibrium index in an array.

An **equilibrium index** is where the sum of elements before it is equal to the sum after it.

Algorithm:

1. Calculate total sum.
2. Traverse and maintain left sum.
3. If leftSum == totalSum - leftSum - arr[i], return index.

Code in Java:

```
int findEquilibriumIndex(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 Complexity: O(n)
Space Complexity: O(1)

Example:
arr = [-7, 1, 5, 2, -4, 3, 0] → Output: 3

Q4. Check if array can be split so that prefix sum = suffix sum.

Algorithm:

1. Find total sum.
2. Traverse and check if prefix sum equals half of total.

Code:

```
boolean canSplit(int[] arr) {
   int total = 0, prefix = 0;
   for (int val : arr) total += val;
   for (int i = 0; i < arr.length - 1; i++) {
```

```
        prefix += arr[i];
        if (prefix == total - prefix) return true;
    }
    return false;
}
```

Time: O(n)
Space: O(1)
Example: arr = [1, 2, 3, 3] → true

Q5. Maximum sum of subarray of size K.

Algorithm: Sliding Window

1. Sum first K elements.
2. Slide the window and update max.

Java Code:

```
int maxSumK(int[] arr, int k) {
    int max = 0, sum = 0;
    for (int i = 0; i < k; i++) sum += arr[i];
    max = sum;
    for (int i = k; i < arr.length; i++) {
        sum += arr[i] - arr[i - k];
        max = Math.max(max, sum);
    }
    return max;
}
```

Time: O(n)
Space: O(1)
Example: arr = [1, 4, 2, 10, 2, 3, 1], k = 3 → Output: 16

Q6. Length of longest substring without repeating characters.

Algorithm: Sliding Window + HashSet

1. Use a set to store characters.
2. Move start and end pointers and track max length.

Code:

```
int lengthOfLongestSubstring(String s) {
   Set<Character> set = new HashSet<>();
   int left = 0, right = 0, maxLen = 0;
   while (right < s.length()) {
      if (!set.contains(s.charAt(right))) {
         set.add(s.charAt(right++));
         maxLen = Math.max(maxLen, set.size());
      } else {
         set.remove(s.charAt(left++));
      }
   }
   return maxLen;
}
```

Time: O(n)
Space: O(n)
Example: "abcabcbb" → Output: 3

Q7. Explain the sliding window technique and its use in string problems.

Sliding window is a technique to maintain a subset of data (window) and slide it across the structure (array or string) to solve problems efficiently.

Applications:

- Substring matching
- Maximum/Minimum in subarrays
- Longest substring with constraints

Example: Longest substring without repeating characters

Q8. Longest palindromic substring.

Algorithm: Expand Around Center Check every index and expand outward.

Java Code:

```
String longestPalindrome(String s) {
   String res = "";
```

```
    for (int i = 0; i < s.length(); i++) {
        res = max(res, expand(s, i, i));
        res = max(res, expand(s, i, i + 1));
    }
    return res;
}
String expand(String s, int l, int r) {
    while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
        l--; r++;
    }
    return s.substring(l + 1, r);
}
```

Time: O(n^2)
Space: O(1)
Example: "babad" → Output: "bab" or "aba"

Q9. Longest common prefix among strings.

Algorithm:

1. Sort the array.
2. Compare the first and last strings character by character.

Code:

```
String longestCommonPrefix(String[] strs) {
    Arrays.sort(strs);
    String first = strs[0], last = strs[strs.length - 1];
    int i = 0;
    while (i < first.length() && i < last.length() && first.charAt(i) == last.charAt(i)) i++;
    return first.substring(0, i);
}
```

Time: O(n log n)
Space: O(1)
Example: ["flower", "flow", "flight"] → Output: "fl"

Q10. Generate all permutations of a string.

Algorithm: Backtracking

Code:

```
void permute(String str, String ans) {
   if (str.length() == 0) {
      System.out.println(ans);
      return;
   }
   for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      String ros = str.substring(0, i) + str.substring(i + 1);
      permute(ros, ans + ch);
   }
}
```

Time: O(n!)
Space: O(n)
Example: "abc" → Output: "abc", "acb", "bac", ...

Q11. Find two numbers in a sorted array that add up to a target.

Algorithm: Two Pointer Approach

1. Start with two pointers, one at the beginning and the other at the end.
2. If the sum is equal to the target, return the pair.
3. If the sum is less, move the left pointer to the right.
4. If the sum is greater, move the right pointer to the left.

Code:

```
int[] findPair(int[] arr, int target) {
   int left = 0, right = arr.length - 1;
   while (left < right) {
      int sum = arr[left] + arr[right];
      if (sum == target) return new int[] { arr[left], arr[right] };
      if (sum < target) left++;
```

```
        else right--;
    }
    return null; // No pair found
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: arr = [1, 3, 5, 7], target = 10 → Output: [3, 7]

Q12. Rearrange numbers into the lexicographically next greater permutation.

Algorithm:

1. Find the longest non-increasing suffix.
2. Swap the pivot with the smallest number larger than it from the suffix.
3. Reverse the suffix.

Code:

```
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);
}

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

void reverse(int[] nums, int start) {
```

```
    int end = nums.length - 1;
    while (start < end) {
        swap(nums, start++, end--);
    }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example: nums = [1, 2, 3] → Output: [1, 3, 2]

Q13. How to merge two sorted linked lists into one sorted list.

Algorithm:

1. Compare the head nodes of both lists.
2. Append the smaller element to the result and move the corresponding pointer.
3. Repeat until one list is exhausted.
4. Append the remaining elements of the other list.

Code:

```
ListNode merge(ListNode l1, ListNode l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;
    if (l1.val < l2.val) {
        l1.next = merge(l1.next, l2);
        return l1;
    } else {
        l2.next = merge(l1, l2.next);
        return l2;
    }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example:
l1 = [1, 2, 4], l2 = [1, 3, 4] → Output: [1, 1, 2, 3, 4, 4]

Q14. Find the median of two sorted arrays using binary search.

Algorithm:

1. Use binary search on the smaller array.
2. Partition both arrays such that the left part of the combined arrays contains half of the elements.
3. Calculate the median from the partition.

Code:

```
double findMedianSortedArrays(int[] nums1, int[] nums2) {
   if (nums1.length > nums2.length) return findMedianSortedArrays(nums2, nums1);
   int m = nums1.length, n = nums2.length;
   int left = 0, right = m, mid = (m + n + 1) / 2;
   while (left < right) {
      int partition1 = (left + right) / 2;
      int partition2 = mid - partition1;
      if (nums1[partition1] < nums2[partition2 - 1]) left = partition1 + 1;
      else right = partition1;
   }
   int partition1 = left, partition2 = mid - left;
   int maxLeft = Math.max(partition1 == 0 ? Integer.MIN_VALUE : nums1[partition1 - 1],
                          partition2 == 0 ? Integer.MIN_VALUE : nums2[partition2 - 1]);
   if ((m + n) % 2 == 1) return maxLeft;
   int minRight = Math.min(partition1 == m ? Integer.MAX_VALUE : nums1[partition1],
                           partition2 == n ? Integer.MAX_VALUE : nums2[partition2]);
   return (maxLeft + minRight) / 2.0;
}
```

Time Complexity: O(log(min(m, n)))
Space Complexity: O(1)
Example:
nums1 = [1, 3], nums2 = [2] → Output: 2.0

Q15. Find the k-th smallest element in a sorted matrix.

Algorithm: Min-Heap or Binary Search

1. Use a min-heap to extract the smallest element from the matrix.

2. Alternatively, apply binary search over the range of matrix values.

Code (Min-Heap):

```
int kthSmallest(int[][] matrix, int k) {
   PriorityQueue<Integer> minHeap = new PriorityQueue<>();
   for (int[] row : matrix) {
      for (int num : row) {
         minHeap.add(num);
      }
   }
   while (--k > 0) minHeap.poll();
   return minHeap.poll();
}
```

Time Complexity: $O(n^2 \log n)$
Space Complexity: $O(n^2)$
Example:
matrix = [[1, 5, 9], [10, 11, 13], [12, 13, 15]], k = 8 → Output: 13

Q16. Find the majority element in an array that appears more than n/2 times.

Algorithm: Boyer-Moore Voting Algorithm

1. Initialize candidate and count variables.
2. Iterate through the array, updating candidate and count.
3. If count reaches 0, update candidate.

Code:

```
int majorityElement(int[] nums) {
   int candidate = nums[0], count = 1;
   for (int i = 1; i < nums.length; i++) {
      if (nums[i] == candidate) count++;
      else count--;
      if (count == 0) {
         candidate = nums[i];
         count = 1;
      }
```

```
    }
    return candidate;
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example:
nums = [3, 2, 3] → Output: 3

Q17. Calculate how much water can be trapped between the bars of a histogram.

Algorithm: Two-Pointer Approach

1. Use two pointers, one at the beginning and one at the end.
2. Track the maximum heights from both sides.
3. Calculate trapped water at each step.

Code:

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

Time Complexity: O(n)
Space Complexity: O(1)
Example:
height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1] → Output: 6

Q18. Find the maximum XOR of two numbers in an array.

Algorithm: Trie Approach

1. Insert elements into a Trie and compute the XOR with each inserted element to get the maximum XOR.

Code:

```
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;
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example:
nums = [3, 10, 5, 25, 2, 8] → Output: 28

Q19. Find the maximum product subarray.

Algorithm: Dynamic Programming

1. Track the maximum and minimum product up to each element.
2. Update the result whenever the current product is greater.

Code:

```
int maxProduct(int[] nums) {
    int maxProduct = nums[0], minProd = nums[0], maxProd = 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]);
        maxProduct = Math.max(maxProduct, maxProd);
    }
    return maxProduct;
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example:
nums = [2, 3, -2, 4] → Output: 6

Q20. Reverse a linked list.

Algorithm:

1. Traverse the list and reverse the links.

Code:

```
ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
```

```
    return prev;
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example:
head = [1, 2, 3, 4, 5] → Output: [5, 4, 3, 2, 1]

Q21. How to count the number of 1s in the binary representation of numbers from 0 to n.

Algorithm:

1. Iterate through numbers from 0 to n.
2. Count the number of 1s in the binary representation of each number.

Code:

```
int countOnes(int n) {
    int count = 0;
    for (int i = 0; i <= n; i++) {
        count += Integer.bitCount(i);
    }
    return count;
}
```

Time Complexity: O(n * log n)
Space Complexity: O(1)
Example:
n = 5 → Output: 7 (Binary representations: 0, 1, 10, 11, 100, 101)

Q22. How to check if a number is a power of two using bit manipulation.

Algorithm:

1. A number is a power of two if it has only one bit set in its binary form.
2. Use the expression n & (n - 1) == 0 to check.

Code:

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

Time Complexity: O(1)
Space Complexity: O(1)
Example:
n = 16 → Output: true, n = 18 → Output: false

Q23. How to find the maximum XOR of two numbers in an array.

Algorithm:

1. Use a Trie or a greedy approach to calculate the maximum XOR.
2. For each number, try to maximize the XOR by comparing it with previously seen numbers.

Code:

```
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;
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example:
nums = [3, 10, 5, 25, 2, 8] → Output: 28

Q24. Explain the concept of bit manipulation and its advantages in algorithm design.

Explanation: Bit manipulation involves performing operations directly on bits (binary digits) of numbers. Common operations include:

- AND (&), OR (|), XOR (^), NOT (~)
- Left shift (<<), Right shift (>>) Advantages:

1. **Efficiency:** Many bitwise operations are faster than their arithmetic counterparts.
2. **Compactness:** Storing and manipulating data in bits can save space.
3. **Problem-solving:** Bit manipulation is useful in solving problems involving subsets, permutations, and optimization problems.

Q25. Solve the problem of finding the next greater element for each element in an array.

Algorithm:

1. Traverse the array from right to left.
2. Use a stack to store elements and find the next greater element for each number.

Code:

```
int[] nextGreaterElements(int[] nums) {
   int n = nums.length;
   int[] result = new int[n];
   Arrays.fill(result, -1);
   Stack<Integer> stack = new Stack<>();
   for (int i = 0; i < 2 * n; i++) {
      while (!stack.isEmpty() && nums[stack.peek()] < nums[i % n]) {
         result[stack.pop()] = nums[i % n];
      }
      stack.push(i % n);
   }
   return result;
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example:
nums = [4, 5, 2, 10] → Output: [5, 10, 10, -1]

Q26. Remove the n-th node from the end of a singly linked list.

Algorithm:

1. Use two pointers: one at the start and the other n steps ahead.
2. Move both pointers until the second pointer reaches the end, then remove the node.

Code:

```
ListNode removeNthFromEnd(ListNode head, int n) {
   ListNode dummy = new ListNode(0);
   dummy.next = head;
   ListNode first = dummy, second = dummy;
   for (int i = 1; i <= n + 1; i++) first = first.next;
   while (first != null) {
      first = first.next;
      second = second.next;
   }
   second.next = second.next.next;
   return dummy.next;
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example:
head = [1, 2, 3, 4, 5], n = 2 → Output: [1, 2, 3, 5]

Q27. Find the node where two singly linked lists intersect.

Algorithm:

1. Find the lengths of both lists.
2. Align both lists and traverse to find the intersection.

Code:

```
ListNode getIntersectionNode(ListNode headA, ListNode headB) {
   int lenA = 0, lenB = 0;
   ListNode tempA = headA, tempB = headB;
   while (tempA != null) {
      lenA++;
      tempA = tempA.next;
```

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

Time Complexity: O(n)
Space Complexity: O(1)
Example:
List A: [1, 2, 3, 4], List B: [5, 6, 7, 8] → Output: Intersection at node 7

Q28. Implement two stacks in a single array.

Algorithm:

1. Use two pointers, one at the beginning and one at the end of the array.
2. Insert elements into the respective stacks.

Code:

```
class TwoStacks {
   int[] arr;
   int top1, top2;
   TwoStacks(int size) {
      arr = new int[size];
      top1 = -1;
      top2 = size;
   }
   void push1(int x) {
      if (top1 + 1 < top2) arr[++top1] = x;
   }
   void push2(int x) {
      if (top1 + 1 < top2) arr[--top2] = x;
   }
   int pop1() {
      if (top1 >= 0) return arr[top1--];
      return -1;
   }
   int pop2() {
      if (top2 < arr.length) return arr[top2++];
      return -1;
   }
}
```

Time Complexity: O(1)
Space Complexity: O(1)
Example:
Push into Stack 1: 10, 20; Push into Stack 2: 30, 40 → Pop from Stack 1: 20, Pop from Stack 2: 40

Q29. Write a program to check if an integer is a palindrome without converting it to a string.

Algorithm:

1. Reverse the second half of the number and compare it with the first half.

Code:

```
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 /= 10;
   }
   return x == reversedHalf || x == reversedHalf / 10;
}
```

Time Complexity: O(log n)
Space Complexity: O(1)
Example:
x = 121 → Output: true, x = -121 → Output: false

Q30. Explain the concept of linked lists and their applications in algorithm design.

Explanation: A linked list is a linear data structure where each element (node) contains data and a reference (link) to the next node in the sequence. There are various types:

- **Singly Linked List:** Each node points to the next node.
- **Doubly Linked List:** Each node points to both the next and previous nodes.
- **Circular Linked List:** The last node points back to the first node.

Applications:

1. **Dynamic memory allocation:** Allows efficient insertion and deletion.
2. **Representation of graphs and adjacency lists.**
3. **Stacks and Queues:** Linked lists are often used to implement these data structures.
4. **Polynomial representation:** Linked lists can be used to store polynomials where each node stores a term.

Q31. Use a deque to find the maximum in every sliding window of size K.

Algorithm:

1. Use a deque to store indices of array elements.
2. Remove elements that are out of the current window.
3. Maintain the deque in decreasing order to find the maximum in constant time.

Code:

```
int[] maxSlidingWindow(int[] nums, int k) {
   if (nums.length == 0) return new int[0];
   int[] result = new int[nums.length - k + 1];
   Deque<Integer> deque = new LinkedList<>();
   for (int i = 0; i < nums.length; i++) {
      if (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
         deque.pollFirst();
      }
      while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
         deque.pollLast();
      }
      deque.offerLast(i);
      if (i >= k - 1) result[i - k + 1] = nums[deque.peekFirst()];
   }
   return result;
}
```

Time Complexity: O(n)
Space Complexity: O(k)
Example:
nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3 → Output: [3, 3, 5, 5, 6, 7]

Q32. How to find the largest rectangle that can be formed in a histogram.

Algorithm:

1. Use a stack to store indices of histogram bars.
2. Calculate the area for each bar as the height and width of the largest rectangle with that bar.

Code:

```
int largestRectangleArea(int[] heights) {
   Stack<Integer> stack = new Stack<>();
   int maxArea = 0, index = 0;
   while (index < heights.length) {
      if (stack.isEmpty() || heights[index] >= heights[stack.peek()]) {
         stack.push(index++);
```

```
        } else {
            int top = stack.pop();
            maxArea = Math.max(maxArea, heights[top] * (stack.isEmpty() ? index : index - stack.peek() - 1));
        }
    }
    while (!stack.isEmpty()) {
        int top = stack.pop();
        maxArea = Math.max(maxArea, heights[top] * (stack.isEmpty() ? index : index - stack.peek() - 1));
    }
    return maxArea;
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example:
heights = [2, 1, 5, 6, 2, 3] → Output: 10

Q33. Explain the sliding window technique and its applications in array problems.

Explanation: The sliding window technique involves maintaining a window (or subarray) of fixed size and "sliding" it across the array. It is used to solve problems that involve finding sums, averages, or other aggregates over contiguous subarrays.

Applications:

1. **Maximum sum subarray of size k.**
2. **Finding the longest substring with at most K distinct characters.**
3. **Finding the minimum length subarray with a sum greater than or equal to a given value.**

Q34. Solve the problem of finding the subarray sum equal to K using hashing.

Algorithm:

1. Traverse the array while maintaining a cumulative sum.
2. Use a hash map to store the frequency of cumulative sums.
3. If cumulative_sum - K exists in the hash map, the subarray sum is equal to K.

Code:

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

```
    Map<Integer, Integer> map = new HashMap<>();
    map.put(0, 1);
    int count = 0, sum = 0;
    for (int num : nums) {
        sum += num;
        if (map.containsKey(sum - k)) count += map.get(sum - k);
        map.put(sum, map.getOrDefault(sum, 0) + 1);
    }
    return count;
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example:
nums = [1, 1, 1], k = 2 → Output: 2

Q35. Find the k-most frequent elements in an array using a priority queue.

Algorithm:

1. Count the frequency of each element using a hashmap.
2. Use a priority queue to store the k most frequent elements.

Code:

```
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());
    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;
}
```

Time Complexity: O(n log k)
Space Complexity: O(n)
Example:
nums = [1, 1, 1, 2, 2, 3], k = 2 → Output: [1, 2]

Q36. Generate all subsets of a given array.

Algorithm:

1. Use backtracking to generate all subsets of the array.

Code:

```
List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(result, new ArrayList<>(), nums, 0);
    return result;
}

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 Complexity: O(2^n)
Space Complexity: O(n)
Example:
nums = [1, 2, 3] → Output: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

Q37. Find all unique combinations of numbers that sum to a target.

Algorithm:

1. Use backtracking to find combinations of numbers that sum to a target.

Code:

```
List<List<Integer>> combinationSum2(int[] candidates, int target) {
   List<List<Integer>> result = new ArrayList<>();
   Arrays.sort(candidates);
   backtrack(result, new ArrayList<>(), candidates, target, 0);
   return result;
}

void backtrack(List<List<Integer>> result, List<Integer> temp, int[] candidates, int target, int start) {
   if (target == 0) {
      result.add(new ArrayList<>(temp));
      return;
   }
   for (int i = start; i < candidates.length; i++) {
      if (i > start && candidates[i] == candidates[i - 1]) continue;
      if (candidates[i] > target) break;
      temp.add(candidates[i]);
      backtrack(result, temp, candidates, target - candidates[i], i + 1);
      temp.remove(temp.size() - 1);
   }
}
```

Time Complexity: O(2^n)
Space Complexity: O(n)
Example:
candidates = [10, 1, 2, 7, 6, 5], target = 8 → Output: [[1, 2, 5], [1, 7], [2, 6]]

Q38. Generate all permutations of a given array.

Algorithm:

1. Use backtracking to generate all permutations.

Code:

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

```
    List<List<Integer>> result = new ArrayList<>();
    backtrack(result, new ArrayList<>(), nums);
    return result;
}

void backtrack(List<List<Integer>> result, List<Integer> temp, int[] nums) {
    if (temp.size() == nums.length) {
        result.add(new ArrayList<>(temp));
        return;
    }
    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 Complexity: O(n!)
Space Complexity: O(n)
Example:
nums = [1, 2, 3] → Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

Q39. Explain the difference between subsets and permutations with examples.

Explanation:

- **Subsets:** A subset is any combination of elements from the set, including the empty set and the set itself. The order of elements does not matter.
    - Example: Subsets of [1, 2] are [[], [1], [2], [1, 2]].
- **Permutations:** A permutation is an arrangement of elements where the order matters.
    - Example: Permutations of [1, 2] are [[1, 2], [2, 1]].

Q40. Write a program to check if a number is an Armstrong number.

Algorithm:

1. A number is an Armstrong number if the sum of its digits raised to the power of the number of digits is equal to the number itself.

Code:

```
boolean isArmstrong(int n) {
   int num = n, sum = 0;
   int digits = (int) Math.log10(n) + 1;
   while (num != 0) {
      int digit = num % 10;
      sum += Math.pow(digit, digits);
      num /= 10;
   }
   return sum == n;
}
```

Time Complexity: O(d) (where d is the number of digits)
Space Complexity: O(1)
Example:
n = 153 → Output: true, n = 123 → Output: false

Q41. Write a program to find the maximum subarray sum using Kadane's algorithm.

Algorithm:

1. Initialize max_sum and current_sum as the first element.
2. Traverse through the array, updating current_sum by including the current element.
3. If current_sum becomes negative, reset it to 0.
4. Update max_sum if current_sum exceeds max_sum.

Code:

```
int maxSubArray(int[] nums) {
   int maxSum = nums[0], currentSum = nums[0];
   for (int i = 1; i < nums.length; i++) {
      currentSum = Math.max(nums[i], currentSum + nums[i]);
      maxSum = Math.max(maxSum, currentSum);
   }
   return maxSum;
```

}

Time Complexity: O(n)
Space Complexity: O(1)
Example:
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4] → Output: 6

Q42. Explain the concept of dynamic programming and its use in solving the maximum subarray problem.

Explanation: Dynamic programming (DP) is a technique to solve problems by breaking them down into simpler subproblems. It avoids redundant computations by storing solutions to subproblems in a table.
For the maximum subarray problem, DP can be used by storing the maximum sum at each index, and at each step, we decide whether to include the current element in the subarray or start a new subarray.

Q43. Solve the problem of finding the top K frequent elements in an array.

Algorithm:

1. Use a hash map to store the frequency of each element.
2. Use a priority queue to extract the top K frequent elements.

Code:

```
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());
  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;
```

}

Time Complexity: O(n log k)
Space Complexity: O(n)
Example:
nums = [1, 1, 1, 2, 2, 3], k = 2 → Output: [1, 2]

Q44. How to find two numbers in an array that add up to a target using hashing.

Algorithm:

1. Use a hash map to store the complement of each element as you traverse the array.
2. If the complement of the current element is found in the map, return the pair.

Code:

```
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[] {-1, -1}; // No solution
}
```

Time Complexity: O(n)
Space Complexity: O(n)
Example:
nums = [2, 7, 11, 15], target = 9 → Output: [0, 1]

Q45. Explain the concept of priority queues and their applications in algorithm design.

Explanation: A **priority queue** is a data structure where each element is associated with a priority. Elements with higher priorities are dequeued before elements with lower priorities. It is typically implemented using heaps.
Applications:

1. **Dijkstra's Algorithm:** Used for finding the shortest path in a graph.
2. **Huffman Coding:** Used for data compression.

3. **Job Scheduling:** For scheduling tasks with different priorities.

Q46. Write a program to find the longest palindromic substring in a given string.

Algorithm:

1. Use a helper function to expand around each character in the string and check for palindromes.
2. Track the longest palindrome while iterating through the string.

Code:

```
String longestPalindrome(String s) {
   if (s.length() <= 1) return s;
   String result = "";
   for (int i = 0; i < s.length(); i++) {
      String odd = expandAroundCenter(s, i, i);
      String even = expandAroundCenter(s, i, i + 1);
      result = odd.length() > result.length() ? odd : result;
      result = even.length() > result.length() ? even : result;
   }
   return result;
}

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

Time Complexity: O(n^2)
Space Complexity: O(1)
Example:
s = "babad" → Output: "bab"

Q47. Explain the concept of histogram problems and their applications in algorithm design.

Explanation: Histogram problems involve working with arrays representing bars, and the goal is often to find areas, such as the largest rectangle or the maximum number of trapped water. The main challenge is efficiently calculating these values using stacks or other algorithms.

Applications:

1. **Largest Rectangle in Histogram:** Finding the largest rectangle that can be formed in a histogram.
2. **Trapping Rain Water:** Calculating the amount of water that can be trapped between bars.

Q48. Solve the problem of finding the next permutation of a given array.

Algorithm:

1. Find the largest index i such that nums[i] < nums[i + 1].
2. Find the largest index j greater than i such that nums[j] > nums[i].
3. Swap nums[i] and nums[j].
4. Reverse the subarray from i + 1 to the end.

Code:

```
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);
}


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

```
void reverse(int[] nums, int start) {
   int end = nums.length - 1;
   while (start < end) {
      swap(nums, start++, end--);
   }
}
```

Time Complexity: O(n)
Space Complexity: O(1)
Example:
nums = [1, 2, 3] → Output: [1, 3, 2]

Q49. How to find the intersection of two linked lists.

Algorithm:

1. Find the lengths of both linked lists.
2. Align them by moving the pointer of the longer list by the difference in lengths.
3. Traverse both lists together and return the node where they intersect.

Code:

```
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 != headB) {
      headA = headA.next;
      headB = headB.next;
```

```
    }
    return headA;
}

int getLength(ListNode head) {
    int length = 0;
    while (head != null) {
        length++;
        head = head.next;
    }
    return length;
}
```

Time Complexity: O(n + m)
Space Complexity: O(1)
Example:
headA = [4,1,8,4,5], headB = [5,0,1,8,4,5] → Output: 8

Q50. Explain the concept of equilibrium index and its applications in array problems.

Explanation: The **equilibrium index** of an array is the index at which the sum of the elements to its left is equal to the sum of the elements to its right.
Applications:

1. Finding a point of balance in arrays.
2. Used in problems where balancing or partitioning the array is required.

Algorithm:

1. Compute the total sum of the array.
2. Traverse the array while maintaining the sum of elements to the left. If the left sum equals the right sum at an index, that is the equilibrium index.

Code:

```
int equilibriumIndex(int[] nums) {
    int totalSum = 0, leftSum = 0;
    for (int num : nums) totalSum += num;
    for (int i = 0; i < nums.length; i++) {
        totalSum -= nums[i];
```

```
        if (leftSum == totalSum) return i;
        leftSum += nums[i];
    }
    return -1;
}
```

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