Advanced Algorithmic Problem Solving (R1UC601B)

Assignment for MTE Solutions

By: Aditri Ashish

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

Concept: A prefix sum array is an array where each element at index i is the sum of all elements from the start to index i of the original array. It is useful in range-sum queries and helps reduce time complexity from O(n) to O(1) for such queries.

Applications:

- Range sum queries
- Checking subarray sum existence
- Solving equilibrium index problems
- Histogram area problems

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

Algorithm:

1. Create a prefix sum array prefix[].
2. prefix[0] = arr[0]
3. For i = 1 to n-1, prefix[i] = prefix[i-1] + arr[i]
4. To find sum in range [L, R]:
    - If L == 0: return prefix[R]
    - Else: return prefix[R] - prefix[L-1]

Java Code:

```
public class PrefixSumRange {
    public static 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];
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        System.out.println(rangeSum(arr, 1, 3)); // Output: 9
    }
}
```

Time Complexity: O(n) for prefix array creation, O(1) for query **Space Complexity:** O(n)

Q3. Equilibrium Index in an Array

Definition: An index where sum of elements before it equals the sum after it.

Algorithm:

1. Compute total sum of array.
2. Traverse the array while maintaining leftSum.
3. At each index i, check: leftSum == totalSum - leftSum - arr[i]

Java Code:

```
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;
    }
    public static void main(String[] args) {
        int[] arr = {-7, 1, 5, 2, -4, 3, 0};
        System.out.println(findEquilibrium(arr)); // Output: 3
    }
}
```

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

Q4. Split Array Into Two Equal Sums

Algorithm:

1. Compute total sum.
2. Traverse array while maintaining left sum.
3. If left sum * 2 == total - arr[i], a split is possible.

Java Code:

```
public class EqualSplit {
    public static boolean canSplit(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 true;
            leftSum += arr[i];
        }
```

```java
        return false;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 3};
        System.out.println(canSplit(arr)); // Output: true
    }
}
```

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

Q5. Maximum Sum Subarray of Size K

Algorithm (Sliding Window):

1. Calculate sum of first K elements.
2. Slide the window forward, update sum by subtracting element left behind and adding new element.

Java Code:

```java
public class MaxSumK {
    public static int maxSum(int[] arr, int k) {
        int max = 0, windowSum = 0;
        for (int i = 0; i < k; i++) windowSum += arr[i];
        max = windowSum;
        for (int i = k; i < arr.length; i++) {
            windowSum += arr[i] - arr[i - k];
            max = Math.max(max, windowSum);
        }
        return max;
    }
```

```java
    public static void main(String[] args) {
        int[] arr = {1, 4, 2, 10, 23, 3, 1, 0, 20};
        System.out.println(maxSum(arr, 4)); // Output: 39
    }
}
```

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

Q6. Longest Substring Without Repeating Characters

Algorithm: Sliding window with hashmap to store last seen index.

Java Code:

```java
import java.util.*;
public class LongestUniqueSubstring {
    public static int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int maxLen = 0, start = 0;
        for (int end = 0; end < s.length(); end++) {
            char c = s.charAt(end);
            if (map.containsKey(c)) start = Math.max(map.get(c) + 1, start);
            map.put(c, end);
            maxLen = Math.max(maxLen, end - start + 1);
        }
        return maxLen;
    }
    public static void main(String[] args) {
        System.out.println(lengthOfLongestSubstring("abcabcbb")); // Output: 3
    }
```

```
}
```

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

Q7. Sliding Window Technique in String Problems

Concept: A technique to maintain a moving range (window) over data.

Use Cases:

- Longest substring without repeating characters
- Anagrams in strings
- Maximum sum substring of size k

Advantage: Reduces time complexity from O(n^2) to O(n) for many problems.

Q8. Longest Palindromic Substring

Algorithm: Expand around center (check every index and expand both sides)

Java Code:

```java
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;
      }
```

```java
        }
        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;
    }
    public static void main(String[] args) {
        System.out.println(longestPalindrome("babad")); // Output: "bab" or "aba"
    }
}
```

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

Q9. Longest Common Prefix among Strings

Algorithm: Compare characters of all strings at each position.

Java Code:

```java
public class LongestCommonPrefix {
    public 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].startsWith(prefix)) {
                prefix = prefix.substring(0, prefix.length() - 1);
```

```
            if (prefix.isEmpty()) return "";
        }
    }
    return prefix;
  }
  public static void main(String[] args) {
    String[] arr = {"flower", "flow", "flight"};
    System.out.println(longestCommonPrefix(arr)); // Output: "fl"
  }
}
```

Time Complexity: O(n * m) where n = no. of strings, m = average length **Space Complexity:** O(1)

Q10. Generate All Permutations of a String

Algorithm: Backtracking with swapping

Java Code:

```
public class Permutations {
  public 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); // backtrack
      }
    }
```

```
    }

    private static String swap(String a, int i, int j) {
        char[] ch = a.toCharArray();
        char temp = ch[i];
        ch[i] = ch[j];
        ch[j] = temp;
        return String.valueOf(ch);
    }

    public static void main(String[] args) {
        String str = "ABC";
        permute(str, 0, str.length() - 1);
    }
}
```

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

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

Algorithm: Use two pointers from both ends of the array.

Java Code:

java

CopyEdit

```
public class TwoSumSorted {
    public 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};
            else if (sum < target) left++;
```

```
            else right--;
        }
        return new int[]{};
    }
}
```

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

Q12. Lexicographically Next Greater Permutation

Algorithm:

1. Find the first decreasing element from the right.
2. Swap it with the next bigger element to its right.
3. Reverse the remaining suffix.

Java Code:

java

CopyEdit

```
import java.util.*;
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--;
            int temp = nums[i];
            nums[i] = nums[j];
```

```
            nums[j] = temp;
        }
        reverse(nums, i + 1);
    }

    private static void reverse(int[] nums, int start) {
        int i = start, j = nums.length - 1;
        while (i < j) {
            int temp = nums[i++];
            nums[i - 1] = nums[j];
            nums[j--] = temp;
        }
    }
}
```

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

Q13. Merge Two Sorted Linked Lists

Java Code:

java

CopyEdit

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

```
public class MergeSortedLists {
    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        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;
        }
        current.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
}
```

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

Q14. Median of Two Sorted Arrays

Use binary search on the smaller array to partition both arrays.
Due to complexity, code is generally long; use standard implementations (like Leetcode 4).

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

Q15. K-th Smallest Element in Sorted Matrix

Java Code:

java

CopyEdit

```
import java.util.*;

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

public class KthSmallestMatrix {
    public static int kthSmallest(int[][] matrix, int k) {
        PriorityQueue<Cell> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.val));
        for (int i = 0; i < matrix.length; i++)
            pq.add(new Cell(matrix[i][0], i, 0));
        while (--k > 0) {
            Cell cell = pq.poll();
            if (cell.c + 1 < matrix[0].length)
                pq.add(new Cell(matrix[cell.r][cell.c + 1], cell.r, cell.c + 1));
        }
        return pq.poll().val;
    }
}
```

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

Q16. Majority Element (> n/2 times)

Java Code:

java

CopyEdit

```
public class MajorityElement {
   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;
   }
}
```

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

Q17. Trapping Rain Water

Java Code:

java

CopyEdit

```
public class RainWaterTrap {
   public static 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)

Q18. Maximum XOR of Two Numbers in an Array

Java Code (Brute-force):

java

CopyEdit

```
public class MaxXOR {
    public static int findMaximumXOR(int[] nums) {
        int max = 0;
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                max = Math.max(max, nums[i] ^ nums[j]);
```

```
        }
      }
      return max;
   }
}
```

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

Q19. Maximum Product Subarray

Java Code:

java

CopyEdit

```
public class MaxProductSubarray {
   public static int maxProduct(int[] nums) {
      int max = nums[0], min = nums[0], result = nums[0];
      for (int i = 1; i < nums.length; i++) {
         int temp = max;
         max = Math.max(nums[i], Math.max(max * nums[i], min * nums[i]));
         min = Math.min(nums[i], Math.min(temp * nums[i], min * nums[i]));
         result = Math.max(result, max);
      }
      return result;
   }
}
```

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

Q20. Count All Numbers with Unique Digits

Java Code:

java

CopyEdit

```
public class UniqueDigitsCount {
    public static int countNumbersWithUniqueDigits(int n) {
        if (n == 0) return 1;
        int count = 10, product = 9, available = 9;
        for (int i = 2; i <= n && available > 0; i++) {
            product *= available--;
            count += product;
        }
        return count;
    }
}
```

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

Q21. Count 1s in Binary Representation from 0 to n

Algorithm: Use dynamic programming

Java Code:

```
public class CountBits {
    public 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 Complexity: O(n) **Space Complexity:** O(n)

Q22. Check Power of Two using Bit Manipulation

Java Code:

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

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

Q23. Max XOR in Array

[Same as Q18, already included above]

Q24. Bit Manipulation Concept

Explanation: Bit manipulation is the act of algorithmically manipulating bits or binary digits. It's faster and uses less memory. Useful in:

- Set/Clear/Test bit
- Subsets
- Power of two

Q25. Next Greater Element for Each Array Element

Algorithm: Use stack to keep track of next greater

Java Code:

```
import java.util.*;

public class NextGreaterElement {

    public 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[i] > nums[stack.peek()]) {

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

            }

            stack.push(i);

        }

        return res;

    }

}
```

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

Q26. Remove N-th Node from End of Linked List

Algorithm: Use two pointers with a gap of n

Java Code:

```
class ListNode {

    int val;

    ListNode next;

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

```
}

public class RemoveNthNode {
    public 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 Complexity: O(n) **Space Complexity:** O(1)

Q27. Intersection of Two Singly Linked Lists

Algorithm: Use two pointers that reset after reaching end.

Java Code:

```
public class LinkedListIntersection {
    public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        ListNode a = headA, b = headB;
        while (a != b) {
            a = (a == null) ? headB : a.next;
```

```java
            b = (b == null) ? headA : b.next;
        }
        return a;
    }
}
```

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

Q28. Two Stacks in One Array

Java Code:

```java
class TwoStacks {
    int size;
    int top1, top2;
    int[] arr;

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

    void push1(int x) {
        if (top1 + 1 < top2) arr[++top1] = x;
    }

    void push2(int x) {
```

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

    int pop1() {
        return (top1 >= 0) ? arr[top1--] : -1;
    }

    int pop2() {
        return (top2 < size) ? arr[top2++] : -1;
    }
}
```

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

Q29. Palindrome Integer (Without String Conversion)

Java Code:

```
public class PalindromeNumber {
    public static boolean isPalindrome(int x) {
        if (x < 0 || (x % 10 == 0 && x != 0)) return false;
        int rev = 0;
        while (x > rev) {
            rev = rev * 10 + x % 10;
            x /= 10;
        }
        return x == rev || x == rev / 10;
    }
}
```

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

Q30. Linked Lists and Applications

Explanation: A linked list is a linear data structure where elements point to the next. Types: singly, doubly, circular.

Applications:

- Dynamic memory allocation
- Implementing queues, stacks
- Efficient insertion/deletion

Q31. Maximum in Every Sliding Window of Size K (Using Deque)

Algorithm: Use a deque to store indices of useful elements.

Java Code:

```
import java.util.*;

public class SlidingWindowMax {

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

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

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

Q32. Largest Rectangle in Histogram

Algorithm: Use stack to keep track of bars.

Java Code:

```
import java.util.*;
public class LargestRectangle {
    public 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 Complexity: O(n) **Space Complexity:** O(n)

Q33. Sliding Window Technique in Array Problems

Explanation: A technique for reducing the nested loop $O(n^2)$ to linear O(n) by maintaining a window that "slides" across the array.

Applications:

- Maximum/minimum in subarrays
- Average/sum of subarrays
- Longest substring with constraints

Q34. Subarray Sum Equals K (Using Hashing)

Java Code:

```
import java.util.*;

public class SubarraySumEqualsK {

    public 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;

            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)

Q35. K Most Frequent Elements (Using Priority Queue)

Java Code:

```
import java.util.*;

public class KMostFrequent {

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

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

        for (int num : nums) freq.put(num, freq.getOrDefault(num, 0) + 1);

        PriorityQueue<Integer> heap = new PriorityQueue<>((a, b) -> freq.get(a) - freq.get(b));

        for (int key : freq.keySet()) {

            heap.add(key);

            if (heap.size() > k) heap.poll();

        }

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

        Collections.reverse(result);

        return result;

    }

}
```

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

Q36. Generate All Subsets of an Array

Java Code:

```
import java.util.*;

public class SubsetsGenerator {

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

```java
        List<List<Integer>> result = new ArrayList<>();
        result.add(new ArrayList<>());
        for (int num : nums) {
            int n = result.size();
            for (int i = 0; i < n; i++) {
                List<Integer> subset = new ArrayList<>(result.get(i));
                subset.add(num);
                result.add(subset);
            }
        }
        return result;
    }
}
```

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

Q37. Unique Combinations That Sum to a Target

Algorithm: Backtracking

Java Code:

```java
import java.util.*;
public class CombinationSum {
    public static List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(candidates, 0, target, new ArrayList<>(), result);
        return result;
    }
```

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

Time Complexity: Exponential (worst case O(2^n)) **Space Complexity:** O(n)

Q38. Generate All Permutations of an Array

Java Code:

```java
import java.util.*;

public class PermutationsArray {
    public static List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        backtrack(nums, new ArrayList<>(), res, new boolean[nums.length]);
        return res;
    }
```

```java
    private static void backtrack(int[] nums, List<Integer> temp, List<List<Integer>> res, boolean[] used) {
        if (temp.size() == nums.length) {
            res.add(new ArrayList<>(temp));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (used[i]) continue;
            used[i] = true;
            temp.add(nums[i]);
            backtrack(nums, temp, res, used);
            used[i] = false;
            temp.remove(temp.size() - 1);
        }
    }
}
```

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

Q39. Difference Between Subsets and Permutations

Explanation:

- **Subsets:** Unordered collection of elements (e.g., [], [1], [1,2])
- **Permutations:** Ordered arrangements (e.g., [1,2], [2,1])

Q40. Element with Maximum Frequency in Array

Java Code:

```java
import java.util.*;
```

```java
public class MaxFrequencyElement {
    public static int mostFrequent(int[] nums) {
        Map<Integer, Integer> map = new HashMap<>();
        int maxFreq = 0, result = nums[0];
        for (int num : nums) {
            int freq = map.getOrDefault(num, 0) + 1;
            map.put(num, freq);
            if (freq > maxFreq) {
                maxFreq = freq;
                result = num;
            }
        }
        return result;
    }
}
```

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

Q41. Maximum Subarray Sum using Kadane's Algorithm

Java Code:

```java
public class KadanesAlgorithm {
    public static int maxSubArray(int[] nums) {
        int maxSoFar = nums[0], maxEndingHere = nums[0];
        for (int i = 1; i < nums.length; i++) {
            maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
            maxSoFar = Math.max(maxSoFar, maxEndingHere);
        }
```

```java
        return maxSoFar;
    }
}
```

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

Q42. Concept of Dynamic Programming in Max Subarray

Explanation: Kadane's Algorithm is a classic example of dynamic programming where we build the solution to a larger problem using the solution to sub-problems.

- dp[i] = max(dp[i-1] + nums[i], nums[i])
- Only last subproblem result is used, optimizing space.

Q43. Top K Frequent Elements in an Array

Java Code:

```java
import java.util.*;

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

        PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> freq.get(a) - freq.get(b));
        for (int key : freq.keySet()) {
            pq.offer(key);
            if (pq.size() > k) pq.poll();
        }

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

```
        Collections.reverse(result);
        return result;
    }
}
```

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

Q44. Two Sum Using Hashing

Java Code:

```
import java.util.*;
public class TwoSumHashing {
    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[]{};
    }
}
```

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

Q45. Concept of Priority Queues

Explanation: A priority queue is an abstract data type where each element is associated with a priority and is served based on priority. Implemented using heap.

Applications:

- Scheduling tasks
- Dijkstra's shortest path
- Top K elements

Q46. Longest Palindromic Substring (Revisited)

Java Code:

```
public class LongestPalindromeString {
    public 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 = 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 Complexity: $O(n^2)$
Space Complexity: $O(1)$

Q47. Histogram Problem and Applications

Explanation:

- Largest rectangle in histogram is a classic problem for stack usage.
- Applications include: Rainwater trapping, Skyline problem, Area chart analysis.

Q48. Next Permutation of an Array

[Already covered in Q12]

Q49. Intersection of Two Linked Lists

[Already covered in Q27]

Q50. Equilibrium Index and Applications

[Already covered in Q3]

Applications:

- Useful in array balancing problems
- Partitioning arrays

- Related to prefix/suffix sums