Advanced Algorithmic Problem Solving (R1UC601B)
Assignment for MTE

---

1. 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 contains the sum of all elements from the start (index 0) up to index i.

Formula:

If arr[] is the original array, then:

prefixSum[0] = arr[0];
prefixSum[i] = prefixSum[i-1] + arr[i];

Applications:

- Range sum queries in O(1)
- Solving subarray problems
- Finding equilibrium index
- Efficient cumulative calculations

---

2. Write a program to find the sum of elements in a given range [L, R] using a prefix sum array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Build prefix sum array.
2. To find sum in range [L, R], use:
    - prefixSum[R] - prefixSum[L - 1] if L > 0
    - prefixSum[R] if L == 0

Program:

```
public class PrefixSumRange {
    public static int[] buildPrefixSum(int[] arr) {
        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 prefix;
    }

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

    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10};
        int[] prefix = buildPrefixSum(arr);
        int sum = rangeSum(prefix, 1, 3);  // sum of 4 + 6 + 8 = 18
        System.out.println("Sum from 1 to 3: " + sum);
    }
}
```

Time and Space Complexity:

- **Build Time**: O(n)
- **Query Time**: O(1)
- **Space**: 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.

Definition:

An index i such that: sum of elements from 0 to i-1 == sum from i+1 to n-1

Algorithm:

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

Program:

```
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};
        int index = findEquilibrium(arr);
        System.out.println("Equilibrium Index: " + index);
    }
}
```

Time and Space Complexity:

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

---

4. Check if an array can be split into two parts such that the sum of the prefix equals the sum of the suffix. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Calculate total sum.
2. Iterate and track left sum.
3. If leftSum == total - leftSum, split is possible.

Program:

```
public class SplitArrayEqualSum {
    public static boolean canBeSplit(int[] arr) {
        int total = 0, leftSum = 0;
        for (int num : arr) total += num;
        for (int i = 0; i < arr.length - 1; i++) {
            leftSum += arr[i];
            if (leftSum == total - leftSum) return true;
        }
        return false;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 3};
        System.out.println("Can be split: " + canBeSplit(arr));
    }
}
```

Time and Space Complexity:

- **Time**: O(n)

- **Space**: O(1)

---

5. Find the maximum sum of any subarray of size K in a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Calculate sum of first K elements.
2. Slide the window forward:
    - Subtract outgoing, add incoming.
    - Update max.

Program:

```
public class MaxSumSubarrayK {
    public static int maxSumK(int[] arr, int k) {
        int maxSum = 0, windowSum = 0;

        for (int i = 0; i < k; i++) windowSum += arr[i];
        maxSum = windowSum;

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

        return maxSum;
    }

    public static void main(String[] args) {
        int[] arr = {1, 4, 2, 10, 2, 3, 1, 0, 20};
        int k = 4;
        System.out.println("Max sum of size " + k + ": " + maxSumK(arr, k));
    }
}
```

Time and Space Complexity:

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

---

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

Algorithm:

1. Use a sliding window with a HashSet or Map.
2. Move right pointer; if character not in set, add and update max.
3. If char is repeated, remove from left side until unique.

Program:

```
import java.util.*;

public class LongestUniqueSubstring {
    public static int lengthOfLongestSubstring(String s) {
        Set<Character> set = new HashSet<>();
        int left = 0, maxLen = 0;

        for (int right = 0; right < s.length(); right++) {
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left++));
            }
            set.add(s.charAt(right));
            maxLen = Math.max(maxLen, right - left + 1);
        }
        return maxLen;
    }

    public static void main(String[] args) {
        String s = "abcabcbb";
        System.out.println("Longest length: " + lengthOfLongestSubstring(s));
    }
}
```

Complexity:

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

---

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

Concept:

The Sliding Window is used for problems involving substrings or subarrays. Instead of checking all possible combinations (which is slow), we slide a window to process elements in O(n) time.

Uses in String Problems:

- Longest substring without repeating characters
- Anagrams in strings
- Fixed-length maximum/minimum substrings

- Longest substring with at most K distinct characters

How it Works:

- Use two pointers (start, end) representing the current window.
- Expand or shrink window based on conditions.

---

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

Algorithm:

1. Expand around every character (odd and even centers).
2. Track longest palindrome.

Program:

```
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);
    }

    public 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) {
        String s = "babad";
        System.out.println("Longest Palindrome: " + longestPalindrome(s));
    }
}
```

Complexity:

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

---

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

Algorithm:

1. Take first word as base.
2. Compare with all others, character by character.
3. Reduce prefix until all match.

Program:

```
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].indexOf(prefix) != 0) {
                prefix = prefix.substring(0, prefix.length() - 1);
                if (prefix.isEmpty()) return "";
            }
        }
        return prefix;
    }

    public static void main(String[] args) {
        String[] strs = {"flower", "flow", "flight"};
        System.out.println("Longest Common Prefix: " +
longestCommonPrefix(strs));
    }
}
```

Complexity:

- **Time**: O(n * m), where n = number of strings, m = length of shortest string
- **Space**: O(1)

---

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

Algorithm:

Use backtracking:

1. Fix one character.
2. Recur for remaining string.
3. Swap back after recursion (backtrack).

Program:

```
public class StringPermutations {
    public static void permute(String str, int l, int r) {
        if (l == r) {
            System.out.println(str);
            return;
        }
        for (int i = l; i <= r; i++) {
            str = swap(str, l, i);
            permute(str, l + 1, r);
            str = swap(str, l, i);
        }
    }

    public static String swap(String s, int i, int j) {
        char[] ch = s.toCharArray();
        char temp = ch[i];
        ch[i] = ch[j];
        ch[j] = temp;
        return new String(ch);
    }

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

Complexity:

- **Time**: O(n × n!)
- **Space**: O(n) (for recursion stack)

---

11. Find two numbers in a sorted array that add up to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm (Two-pointer technique):

1. Initialize two pointers: left = 0, right = n - 1.
2. While left < right:
    - If arr[left] + arr[right] == target, return indices.
    - If sum < target, move left forward.
    - If sum > target, move right backward.

Program:

```
public class TwoSumSorted {
    public static int[] twoSum(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[]{left, right};
            else if (sum < target)
                left++;
            else
                right--;
        }
        return new int[]{-1, -1};
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 7, 11, 15};
        int[] result = twoSum(arr, 9); // 2 + 7 = 9
        System.out.println("Indices: " + result[0] + ", " + result[1]);
    }
}
```

Complexity:

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

---

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

Algorithm:

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

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, nums.length - 1);
    }

    private static void swap(int[] arr, int i, int j) {
        int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
    }

    private static void reverse(int[] arr, int start, int end) {
        while (start < end) swap(arr, start++, end--);
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        nextPermutation(arr);
        System.out.println(Arrays.toString(arr));    }
}
```

Complexity:

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

---

13. How to merge two sorted linked lists into one sorted list. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

1. Use a dummy node.
2. Compare heads of both lists.
3. Append the smaller node to merged list.
4. Continue until both lists are merged.

Program:

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

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

public class MergeSortedLists {
    public static ListNode merge(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode tail = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                tail.next = l1; l1 = l1.next;
            } else {
                tail.next = l2; l2 = l2.next;
            }
            tail = tail.next;
        }
        tail.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }

    // Helper to print list
    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " -> ");
            head = head.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        ListNode a = new ListNode(1); a.next = new ListNode(3); a.next.next =
new ListNode(5);
        ListNode b = new ListNode(2); b.next = new ListNode(4); b.next.next =
new ListNode(6);
        ListNode merged = merge(a, b);
        printList(merged);
    }
}
```

Complexity:

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

---

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

Algorithm:

- Use binary search on the smaller array.

- Partition both arrays such that elements on the left are ≤ right.
- Use maxLeft and minRight to compute median.

Program:

```
public class MedianSortedArrays {
    public static double findMedian(int[] A, int[] B) {
        if (A.length > B.length) return findMedian(B, A);

        int x = A.length, y = B.length;
        int low = 0, high = x;

        while (low <= high) {
            int partitionX = (low + high) / 2;
            int partitionY = (x + y + 1) / 2 - partitionX;

            int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE :
A[partitionX - 1];
            int minRightX = (partitionX == x) ? Integer.MAX_VALUE :
A[partitionX];

            int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE :
B[partitionY - 1];
            int minRightY = (partitionY == y) ? Integer.MAX_VALUE :
B[partitionY];

            if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
                if ((x + y) % 2 == 0)
                    return (Math.max(maxLeftX, maxLeftY) +
Math.min(minRightX, minRightY)) / 2.0;
                else
                    return Math.max(maxLeftX, maxLeftY);
            } else if (maxLeftX > minRightY)
                high = partitionX - 1;
            else
                low = partitionX + 1;
        }
        return 0;
    }

    public static void main(String[] args) {
        int[] A = {1, 3};
        int[] B = {2};
        System.out.println("Median: " + findMedian(A, B));
    }
}
```

Complexity:

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

---

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

Assumptions:

- Each row and column is sorted.

Algorithm:

1. Use a Min Heap (PriorityQueue).
2. Insert elements from the first row.
3. Pop smallest and insert the next element in the same column.
4. Repeat K times.

Program:

```
import java.util.PriorityQueue;

class Tuple {
    int val, row, col;
    Tuple(int v, int r, int c) { val = v; row = r; col = c; }
}

public class KthSmallestMatrix {
    public static int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        PriorityQueue<Tuple> pq = new PriorityQueue<>((a, b) -> a.val -
b.val);

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

        for (int i = 0; i < k - 1; i++) {
            Tuple t = pq.poll();
            if (t.col < n - 1)
                pq.offer(new Tuple(matrix[t.row][t.col + 1], t.row, t.col +
1));
        }

        return pq.poll().val;
    }

    public static void main(String[] args) {
        int[][] matrix = {
            {1, 5, 9},
            {10, 11, 13},
            {12, 13, 15}
        };
        int k = 8;
        System.out.println("Kth smallest: " + kthSmallest(matrix, k));
    }
}
```

Complexity:

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

---

16. Find the majority element in an array that appears more than n/2 times. Write its algorithm, program. Find its time and space complexities. Explain with suitable example

Algorithm (Boyer-Moore Voting Algorithm):

1. Initialize count = 0, candidate = None.
2. Traverse the array:
    - If count == 0, set candidate = num.
    - If num == candidate, count++, else count--.
3. Return the candidate.

Code:

```
public class MajorityElement {
    public static int findMajority(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, 2, 2, 2};
        System.out.println("Majority element: " + findMajority(nums));
    }
}
```

Complexity:

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

---

17. Calculate how much water can be trapped between the bars of a histogram. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm (Two Pointer):

1. Use two pointers left and right, leftMax, rightMax.
2. While left < right, calculate trapped water based on min of leftMax and rightMax.

Code:

```
public class TrappingRainWater {
    public static int trap(int[] height) {
        int left = 0, right = height.length - 1;
        int leftMax = 0, rightMax = 0, water = 0;

        while (left < right) {
            if (height[left] < height[right]) {
                leftMax = Math.max(leftMax, height[left]);
                water += leftMax - height[left];
                left++;
            } else {
                rightMax = Math.max(rightMax, height[right]);
                water += rightMax - height[right];
                right--;
            }
        }
        return water;
    }

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

Complexity:

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

---

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

Algorithm (Using Trie or Greedy):

- Build prefix using masks and use XOR property to maximize result.

Code (Greedy with HashSet):

```
import java.util.HashSet;

public class MaximumXOR {
```

```
    public static int findMaximumXOR(int[] nums) {
        int max = 0, mask = 0;

        for (int i = 31; i >= 0; i--) {
            mask |= (1 << i);
            HashSet<Integer> set = new HashSet<>();

            for (int num : nums)
                set.add(num & mask);

            int temp = max | (1 << i);
            for (int prefix : set) {
                if (set.contains(temp ^ prefix)) {
                    max = temp;
                    break;
                }
            }
        }
        return max;
    }

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

Complexity:

- **Time**: O(n * 32)
- **Space**: O(n)

---

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

Algorithm:

- Keep track of maxProduct, minProduct at each index.
- Update result = max(result, maxProduct).

Code:

```
public class MaxProductSubarray {
    public static int maxProduct(int[] nums) {
        int maxProd = nums[0], minProd = nums[0], result = nums[0];

        for (int i = 1; i < nums.length; i++) {
            int temp = maxProd;
            maxProd = Math.max(nums[i], Math.max(maxProd * nums[i], minProd *
nums[i]));
```

```
            minProd = Math.min(nums[i], Math.min(temp * nums[i], minProd *
nums[i]));
            result = Math.max(result, maxProd);
        }

        return result;
    }

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

Complexity:

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

---

20. Count all numbers with unique digits for a given number of digits. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

- For digits 0 to n, calculate how many numbers can be formed with unique digits using permutations.

Code:

```
public class UniqueDigitNumbers {
    public static int countNumbersWithUniqueDigits(int n) {
        if (n == 0) return 1;
        int count = 10, uniqueDigits = 9, available = 9;

        for (int i = 2; i <= n && available > 0; i++) {
            uniqueDigits *= available--;
            count += uniqueDigits;
        }

        return count;
    }

    public static void main(String[] args) {
        int n = 2;
        System.out.println("Count: " + countNumbersWithUniqueDigits(n));
    }
}
```

Complexity:

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

---

21. How to count the number of 1s in the binary representation of numbers from 0 to n. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

- Use dynamic programming: countBits[i] = countBits[i >> 1] + (i & 1)
- Shifts the number right by 1 (equivalent to dividing by 2) and adds 1 if the last bit is 1.

Code:

```
import java.util.Arrays;

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;
    }

    public static void main(String[] args) {
        int n = 5;
        System.out.println("Count of 1s: " + Arrays.toString(countBits(n)));

    }
}
```

Complexity:

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

---

22. How to check if a number is a power of two using bit manipulation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm:

- A power of 2 has only one set bit: e.g., 1 (0001), 2 (0010), 4 (0100), etc.
- Use: n > 0 && (n & (n - 1)) == 0

Code:

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

    public static void main(String[] args) {
        int num = 16;
        System.out.println(num + " is power of 2? " + isPowerOfTwo(num));
}
}
```

Complexity:

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

---

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

This was **also Q18**, but here's a brief recap.

Algorithm:

- Build prefix set of numbers using a mask.
- Try to set bits from highest to lowest using XOR and check existence in the set.

Code:

```
import java.util.HashSet;

public class MaxXOR {
    public static int findMaxXOR(int[] nums) {
        int max = 0, mask = 0;
        for (int i = 31; i >= 0; i--) {
            mask |= (1 << i);
            HashSet<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[] arr = {3, 10, 5, 25, 2, 8};
        System.out.println("Max XOR: " + findMaxXOR(arr));
    }
}
```

Complexity:

- **Time**: O(n * 32)
- **Space**: O(n)

---

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

Concept:

Bit manipulation is the process of using **bitwise operators** (AND, OR, XOR, NOT, shifts) to perform operations directly on the binary representation of integers.

Common Bitwise Operators:

- & AND
- | OR
- ^ XOR
- ~ NOT
- << Left shift
- >> Right shift

Advantages:

- **Performance**: Faster than arithmetic or loop operations.
- **Memory Efficient**: Use fewer bytes to store information.
- **Real-world Use Cases**:
    - Checking even/odd: (x & 1)
    - Swapping values without temp: a ^= b; b ^= a; a ^= b;
    - Finding power of 2, counting set bits
    - Cryptography, graphics, compression

---

25. Solve the problem of finding the next greater element for each element in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm (Using Stack):

1. Traverse from **right to left**.
2. Use a stack to keep track of next greater elements.
3. For each element, pop smaller elements from the stack.
4. If the stack is empty, no greater element exists.

Code:

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

public class NextGreaterElement {
    public static int[] nextGreater(int[] nums) {
        int[] res = new int[nums.length];
        Stack<Integer> stack = new Stack<>();

        for (int i = nums.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= nums[i])
                stack.pop();
            res[i] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(nums[i]);
        }
        return res;
    }

    public static void main(String[] args) {
        int[] arr = {4, 5, 2, 25};
        System.out.println("Next Greater Elements: " +
Arrays.toString(nextGreater(arr)));
            }
}
```

Complexity:

- **Time**: O(n)
- **Space**: O(n) for stack

---

26. Remove the N-th Node from End of a Singly Linked List

Algorithm:

1. Use two pointers first and second, both starting at the head.

2. Move first pointer n steps ahead.
3. Move both pointers one step at a time until first reaches the end.
4. second will be just before the node to be removed.
5. Skip the target node using second.next = second.next.next.

Java Code:

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

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

      ListNode first = dummy;
      ListNode second = dummy;

      for (int i = 0; i < n + 1; i++)
         first = first.next;

      while (first != null) {
         first = first.next;
         second = second.next;
      }

      second.next = second.next.next;
```

```
        return dummy.next;
    }

    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);

        head = removeNthFromEnd(head, 2);

        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
    }
}
```

Complexity:

- **Time:** O(L) where L = length of list
- **Space:** O(1)

---

27. Find Intersection Node of Two Singly Linked Lists

Algorithm:

1. Calculate lengths of both lists.
2. Advance the longer list by the length difference.
3. Move both pointers together until they meet or reach null.

Java Code:

```
public class IntersectionFinder {
    public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = getLength(headA), 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;
    }

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

```
        return len;
    }
}
```

Complexity:

- **Time:** O(M + N)
- **Space:** O(1)

---

28. Two Stacks in One Array

Algorithm:

1. Use a single array.
2. Stack1 grows from start to end, Stack2 from end to start.
3. Prevent overlap using index comparison.

Java Code:

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

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

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

```
            System.out.println("Stack Overflow");
    }

    void push2(int x) {
        if (top1 + 1 < top2)
            arr[--top2] = x;
        else
            System.out.println("Stack Overflow");
    }

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

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

Complexity:

- **Time:** O(1) for push/pop
- **Space:** O(N) (single array)

---

29. Check if Integer is a Palindrome (No String Conversion)

Algorithm:

1. Negative numbers are not palindrome.
2. Reverse the second half of the number.
3. Compare with the first half.

Java Code:

```
public class PalindromeNumber {
    public 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;
    }

    public static void main(String[] args) {
        System.out.println(isPalindrome(121));  // true
        System.out.println(isPalindrome(123));  // false
    }
}
```

Complexity:

- **Time:** $O(\log_{10} N)$
- **Space:** O(1)

---

30. Linked List Concept & Applications

Concept:

A linked list is a linear data structure where each element (node) points to the next. It consists of:

- Data (value)

- Next (reference to the next node)

Types:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Applications:

- Dynamic memory allocation
- Implementing stacks, queues
- Undo operations in editors
- Graph representations
- Efficient insert/delete from middle of a list

Example (Singly Linked List in Java):

```
class Node {
   int data;
   Node next;
   Node(int data) {
      this.data = data;
      next = null;
   }
}
```

◇ **Advantages:**

- No fixed size (unlike arrays)
- Efficient insertions/deletions

◇ **Disadvantages:**

- No random access

- More memory (due to pointers)

---

31. Find Maximum in Every Sliding Window of Size K Using Deque

Algorithm:

1. Use a Deque to store indices.
2. Traverse the array:
    - Remove elements out of window bounds from front.
    - Remove smaller elements from back (they can't be maximum).
    - Add current index to deque.
    - When window size ≥ k, add front element (maximum) to result.

Java Code:

```
import java.util.*;


public class SlidingWindowMax {
   public static int[] maxSlidingWindow(int[] nums, int k) {
      if (nums.length == 0) return new int[0];
      int n = nums.length;
      int[] result = new int[n - k + 1];
      Deque<Integer> dq = new LinkedList<>();

      for (int i = 0; i < n; i++) {
         while (!dq.isEmpty() && dq.peek() < i - k + 1)
            dq.poll();

         while (!dq.isEmpty() && nums[dq.peekLast()] < nums[i])
            dq.pollLast();
```

```
            dq.offer(i);

            if (i >= k - 1)
                result[i - k + 1] = nums[dq.peek()];
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        System.out.println(Arrays.toString(maxSlidingWindow(nums, k))); // [3, 3, 5, 5, 6, 7]
    }
}
```

Time Complexity:

- **O(N) –** each element is added and removed at most once
- **Space:** O(K) for deque

---

32. Largest Rectangle in Histogram

Algorithm:

1. Use a stack to track increasing bars' indices.
2. When a shorter bar is found, pop from stack and calculate area with that bar as the smallest.
3. Repeat till the end.

Java Code:

```
import java.util.*;

public class HistogramMaxArea {
    public static int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0;
        int n = heights.length;

        for (int i = 0; i <= n; i++) {
            int h = (i == n) ? 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;
    }

    public static void main(String[] args) {
        int[] heights = {2,1,5,6,2,3};
        System.out.println(largestRectangleArea(heights)); // Output: 10
```

```
    }
}
```

Time Complexity:

- **O(N) –** each bar pushed and popped once
- **Space:** O(N) for stack

---

33. Sliding Window Technique

Concept:

Sliding window is used to reduce nested loops by maintaining a window (subset of elements) and sliding it across the array to maintain efficiency.

Types:

1. Fixed-size window – window of size k
2. Variable-size window – window changes depending on condition (e.g., sum ≤ K)

Applications:

- Maximum/Minimum in subarrays
- Longest substring without repeating characters
- Subarray sum problems
- Streaming data processing

---

34. Subarray Sum Equals K (Using Hashing)

Algorithm:

1. Use a hashmap to store prefix sums and their frequency.
2. For each prefix sum:
    - Check if prefixSum - k exists in map → it means there's a subarray with sum k.

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 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;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        int k = 3;
        System.out.println(subarraySum(nums, k)); // Output: 2
    }
}
```

Time Complexity:

- **O(N) –** one pass
- **Space: O(N) –** for hashmap

---

35. K Most Frequent Elements Using Priority Queue

Algorithm:

1. Count frequency of each element using a HashMap.
2. Use a min-heap (priority queue) to store elements by frequency.
3. Maintain heap size of k.
4. Extract elements from heap to get k most frequent.

Java Code:

```
import java.util.*;


public class KMostFrequent {
   public static int[] topKFrequent(int[] nums, int k) {
      Map<Integer, Integer> freqMap = new HashMap<>();
      for (int n : nums) freqMap.put(n, freqMap.getOrDefault(n, 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();
      }
```

```
        int[] result = new int[k];
        for (int i = k - 1; i >= 0; i--)
            result[i] = pq.poll().getKey();

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1,1,1,2,2,3};
        int k = 2;
        System.out.println(Arrays.toString(topKFrequent(nums, k))); // Output: [1, 2]
    }
}
```

Time Complexity:

- O(N log K)
- **Space:** O(N)

---

36. Generate All Subsets of an Array

Algorithm:

- Use backtracking or bit manipulation.
- For each element, either include or exclude it.

Java Code (Backtracking):

```
import java.util.*;

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

```
        List<List<Integer>> result = new ArrayList<>();
        backtrack(0, nums, new ArrayList<>(), result);
        return result;
    }

    private static void backtrack(int start, int[] nums, List<Integer> temp, List<List<Integer>>
result) {
        result.add(new ArrayList<>(temp));
        for (int i = start; i < nums.length; i++) {
            temp.add(nums[i]);
            backtrack(i + 1, nums, temp, result);
            temp.remove(temp.size() - 1);
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        System.out.println(subsets(nums)); // Output: all subsets
    }
}
```

Time Complexity:

- O(2^n)
- **Space:** O(n) for recursion stack

---

37. Unique Combinations That Sum to Target (Combination Sum)

Algorithm:

- Use backtracking to explore choices.

- Prune paths where sum exceeds target.
- Avoid duplicates by always moving forward in the array.

Java Code:

```
import java.util.*;


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


   private static void backtrack(int start, int[] candidates, int target, List<Integer> temp,
List<List<Integer>> result) {
      if (target == 0) {
         result.add(new ArrayList<>(temp));
         return;
      }
      if (target < 0) return;


      for (int i = start; i < candidates.length; i++) {
         temp.add(candidates[i]);
         backtrack(i, candidates, target - candidates[i], temp, result);
         temp.remove(temp.size() - 1);
      }
   }
```

```
    public static void main(String[] args) {
        int[] candidates = {2,3,6,7};
        int target = 7;
        System.out.println(combinationSum(candidates, target)); // Output: [[2,2,3],[7]]
    }
}
```

Time Complexity:

- O(2^target) in worst case
- **Space:** O(target)

---

38. Generate All Permutations of an Array

Algorithm:

- Use backtracking by swapping elements.

Java Code:

```
import java.util.*;

public class Permutations {
    public static List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(0, nums, result);
        return result;
    }

    private static void backtrack(int index, int[] nums, List<List<Integer>> result) {
        if (index == nums.length) {
```

```
            List<Integer> list = new ArrayList<>();
            for (int n : nums) list.add(n);
            result.add(list);
            return;
        }

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

    private static void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        System.out.println(permute(nums)); // Output: all permutations
    }
}
```

Time Complexity:

- O(n!)
- **Space:** O(n) for recursion

---

39. Difference Between Subsets and Permutations

Subsets:

- Unordered combinations of elements.
- Each element is either included or not.
- Total: 2^n subsets

Permutations:

- Ordered arrangements of elements.
- Total: n! permutations for n distinct elements

---

40. Find Element With Maximum Frequency

Algorithm:

- Count frequency using HashMap.
- Track max frequency and corresponding element.

Java Code:

```
import java.util.*;


public class MaxFrequency {
   public static int maxFrequencyElement(int[] nums) {
      Map<Integer, Integer> map = new HashMap<>();
      int maxFreq = 0, element = nums[0];


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

```
            if (freq > maxFreq) {
                maxFreq = freq;
                element = num;
            }
        }

        return element;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 2, 1, 4, 1};
        System.out.println(maxFrequencyElement(nums)); // Output: 1
    }
}
```

Time Complexity:

- O(N) – one pass
- **Space:** O(N)

---

41. Maximum Subarray Sum Using Kadane's Algorithm

Algorithm:

1. Initialize maxSoFar = nums[0], maxEndingHere = nums[0].
2. Loop through the array starting from index 1.
3. At each step, calculate:
    - maxEndingHere = max(nums[i], maxEndingHere + nums[i])
    - maxSoFar = max(maxSoFar, maxEndingHere)
4. Return maxSoFar.

Java Code:

```
public class KadaneAlgorithm {
    public static int maxSubArray(int[] nums) {
        int maxEndingHere = nums[0];
        int maxSoFar = nums[0];

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

        return maxSoFar;
    }

    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum Subarray Sum: " + maxSubArray(nums)); // Output: 6
    }
}
```

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

---

42. Dynamic Programming in Maximum Subarray

Concept:

Dynamic Programming (DP) solves problems by combining solutions to subproblems. For maximum subarray, the problem has overlapping subproblems and optimal substructure, ideal for DP.

Explanation:

- At each index i, we store the maximum sum of subarray ending at index i.
- Use the recurrence:
  dp[i] = max(nums[i], dp[i-1] + nums[i])
- Kadane's Algorithm is a space-optimized version of this DP approach.

---

43. Top K Frequent Elements in an Array

Algorithm:

1. Count frequency of elements using HashMap.
2. Use a min-heap of size k to keep top k elements.
3. Return elements from the heap.

Java Code:

```
import java.util.*;


public class TopKFrequentElements {
   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>> 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();
```

```
        }

        int[] result = new int[k];
        int i = 0;
        for (Map.Entry<Integer, Integer> entry : pq)
            result[i++] = entry.getKey();

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1,1,1,2,2,3};
        int k = 2;
        System.out.println(Arrays.toString(topKFrequent(nums, k))); // Output: [1, 2]
    }
}
```

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

---

44. Two Sum Problem Using Hashing

Algorithm:

1. Initialize a HashMap.
2. Loop through the array.
3. For each element, check if target - nums[i] exists in the map.
4. If yes, return the indices.

Java Code:

```
import java.util.*;

public class TwoSum {
   public static int[] findTwoSum(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}; // not found
   }

   public static void main(String[] args) {
      int[] nums = {2, 7, 11, 15};
      int target = 9;
      System.out.println(Arrays.toString(findTwoSum(nums, target))); // Output: [0, 1]
   }
}
```

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

---

45. Priority Queues & Applications in Algorithms

Concept:

A priority queue is a data structure where each element has a priority. Elements with higher priority are served before those with lower priority.

Java Implementation:

Java uses PriorityQueue which is a min-heap by default.

PriorityQueue<Integer> pq = new PriorityQueue<>(); // Min-Heap

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder()); // Max-Heap

---

Applications:

1. Dijkstra's Algorithm (shortest path)
2. A* Search
3. Huffman Coding
4. Job Scheduling
5. K Largest / Smallest Elements
6. Median in Data Stream

---

46. Longest Palindromic Substring

Algorithm (Expand Around Center):

1. For each index i, expand around:
    - Odd-length palindrome: center at i.
    - Even-length palindrome: center at i and i+1.
2. Track the longest palindrome found during expansions.

Java Code:

```
public class LongestPalindromeSubstring {
    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;
}

public static void main(String[] args) {
    String s = "babad";
```

```
        System.out.println("Longest Palindromic Substring: " + longestPalindrome(s)); // Output: "bab" or "aba"
    }
}
```

Time Complexity: $O(n^2)$

Space Complexity: O(1)

---

47. Histogram Problems – Concept & Applications

Concept:

Given heights of bars in a histogram, the task is to find the largest rectangle that can be formed.

Applications:

- Solving largest rectangle in a matrix
- Max area under curve problems
- In image processing or skyline silhouette
- Memory allocation problems

---

48. Next Permutation

Algorithm:

1. Find the first decreasing index from the right (i).
2. Find just larger element than arr[i] from the right (j).
3. Swap arr[i] and arr[j].
4. Reverse subarray from i+1 to end.

Java Code:

```
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, nums.length - 1);
}

private static void reverse(int[] nums, int start, int end) {
    while (start < end) swap(nums, start++, end--);
}

private static void swap(int[] nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}

public static void main(String[] args) {
```

```
        int[] arr = {1, 2, 3};
        nextPermutation(arr);
        System.out.println("Next Permutation: " + Arrays.toString(arr)); // Output: [1, 3, 2]
    }
}
```

Time Complexity: O(n)

Space Complexity: O(1)

---

49. Intersection of Two Linked Lists

Algorithm:

1. Use two pointers a and b.
2. Traverse each list and switch heads when reaching null.
3. They will either meet at the intersection node or null.

Java Code:

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

public class IntersectionOfLinkedLists {
    public static 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;
    }

    public static void main(String[] args) {
        // Sample test with shared nodes
    }
}
```

Time Complexity: O(m + n)

Space Complexity: O(1)

---

50. Equilibrium Index – Concept & Application

Concept:

An equilibrium index in an array is an index i such that the sum of elements before i is equal to the sum of elements after i.

Application:

- In balance point problems
- Load balancing in distributed systems
- Financial modeling
