Data Structures and Algorithms

1. Arrays and Strings

Concept

Arrays are collections of elements that are stored in contiguous memory locations. Strings are arrays of characters. Both allow efficient access and manipulation of data.

Example Problem: Merge Strings Alternately

Problem Statement: Given two strings, merge them by alternating characters. For example, given "abc" and "def", the result should be "adbcef".

Mathematical Logic:

1. Initialize an empty string (or StringBuilder) for the result.
2. Use two indices to track the position in each string.
3. Continue until both strings are fully traversed.
4. Append characters from both strings alternatively.

Java Code:

```
public class MergeStrings {
    public static String mergeAlternately(String word1, String word2) {
        StringBuilder merged = new StringBuilder();
        int i = 0, j = 0;

        // Traverse both strings until we reach the end of one
        while (i < word1.length() || j < word2.length()) {
            if (i < word1.length()) {
                merged.append(word1.charAt(i++));
            }
            if (j < word2.length()) {
                merged.append(word2.charAt(j++));
            }
        }
        return merged.toString();
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the length of the longer string.
- **Space Complexity:** O(n) for storing the merged result.

2. Two-Pointer Technique

Concept

The two-pointer technique involves using two pointers to traverse a data structure from both ends or different starting points to efficiently solve problems.

Example Problem: Container with Most Water

Problem Statement: Given an array representing the heights of lines, find two lines that, along with the x-axis, form a container that holds the most water. For example, for the heights [1,8,6,2,5,4,8,3,7], the maximum area is 49.

Mathematical Logic:

1. Initialize two pointers: one at the beginning and one at the end of the array.
2. Calculate the area formed by the lines at these two pointers.
3. Move the pointer pointing to the shorter line inward (as this may help find a taller line that can hold more water).
4. Repeat until the two pointers meet.

Java Code:

```java
public class MaxArea {
    public static int maxArea(int[] height) {
        int left = 0, right = height.length - 1;
        int maxArea = 0;

        // Iterate until the two pointers meet
        while (left < right) {
            int area = Math.min(height[left], height[right]) * (right - left);
            maxArea = Math.max(maxArea, area);
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }
        return maxArea;
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the number of heights.
- **Space Complexity:** O(1), as no additional space is used.

3. Sliding Window Technique

Concept

The sliding window technique uses a moving window over a data structure to optimize subarray queries, usually for problems involving contiguous elements.

Example Problem: Maximum Number of Vowels in a Substring of Given Length

Problem Statement: Given a string, find the maximum number of vowels in any substring of length k. For example, for the string "abciiidef" and k = 3, the result is 3.

Mathematical Logic:

1. Initialize a count of vowels in the first window of size k.
2. Slide the window one character at a time, updating the count by removing the character that is left out and adding the new character.
3. Keep track of the maximum count encountered during this process.

Java Code:

```java
public class MaxVowels {
    public static int maxVowels(String s, int k) {
        int maxCount = 0, count = 0;

        // Count vowels in the first window
        for (int i = 0; i < k; i++) {
            if ("aeiou".indexOf(s.charAt(i)) != -1) {
                count++;
            }
        }
        maxCount = count;

        // Slide the window
        for (int i = k; i < s.length(); i++) {
            if ("aeiou".indexOf(s.charAt(i)) != -1) {
                count++;
            }
            if ("aeiou".indexOf(s.charAt(i - k)) != -1) {
                count--;
            }
            maxCount = Math.max(maxCount, count);
        }
        return maxCount;
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the length of the string.
- **Space Complexity:** O(1), since we are only using a few variables.

4. Prefix Sum

Concept

Prefix sum is a technique that allows the quick calculation of the sum of elements in a range by storing cumulative sums.

Example Problem: Find Pivot Index

Problem Statement: Find the index of an element in an array where the sum of all elements to the left is equal to the sum of all elements to the right. For example, in the array [1, 7, 3, 6, 5, 6], the pivot index is 3.

Mathematical Logic:

1. Calculate the total sum of the array.
2. Initialize a variable to keep track of the left sum.
3. Iterate through the array, updating the left sum, and check if it equals the total sum minus the left sum minus the current element.

Java Code:

```java
public class PivotIndex {
    public static int pivotIndex(int[] nums) {
        int totalSum = 0, leftSum = 0;

        // Calculate total sum of the array
        for (int num : nums) {
            totalSum += num;
        }

        // Check for pivot index
        for (int i = 0; i < nums.length; i++) {
            if (leftSum == totalSum - leftSum - nums[i]) {
                return i;
            }
            leftSum += nums[i];
        }
        return -1;
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the number of elements in the array.
- **Space Complexity:** O(1), as we only use a few additional variables.

5. Hash Maps / Sets

Concept

Hash maps and sets allow efficient storage and retrieval of data using a hash-based approach, making operations like lookup, insertion, and deletion fast.

Example Problem: Find the Difference of Two Arrays

Problem Statement: Given two arrays, find the elements that are not common between them. For example, for nums1 = [1,2,3] and nums2 = [2,4,6], the output should be [[1,3],[4,6]].

Mathematical Logic:

1. Use two hash sets to store elements from each array.
2. Use the methods of sets to find the difference, which allows easy identification of unique elements.

Java Code:

```java
import java.util.HashSet;
import java.util.ArrayList;
import java.util.List;

public class ArrayDifference {
    public static List<List<Integer>> findDifference(int[] nums1, int[] nums2) {
        HashSet<Integer> set1 = new HashSet<>();
        HashSet<Integer> set2 = new HashSet<>();

        // Add elements of nums1 to set1
        for (int num : nums1) {
            set1.add(num);
        }
        // Add elements of nums2 to set2
        for (int num : nums2) {
            set2.add(num);
        }

        // Find unique elements
        List<Integer> diff1 = new ArrayList<>(set1);
        List<Integer> diff2 = new ArrayList<>(set2);
        diff1.removeAll(set2); // Remove elements of nums2 from nums1
        diff2.removeAll(new HashSet<>(List.of(nums1))); // Remove elements of
nums1 from nums2

        return List.of(diff1, diff2);
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n + m), where n and m are the lengths of the two arrays.
- **Space Complexity:** O(n + m) for storing the sets.

6. Stack and Queue

Concept

A stack follows the Last In First Out (LIFO) principle, while a queue follows the First In First Out (FIFO) principle. Both are essential data structures for managing order in collections.

Example Problem: Removing Stars from a String

Problem Statement: Process the string by removing characters that are preceded by a star. For example, "leet**c**o**de**" becomes "leeto".

Mathematical Logic:

1. Use a stack to keep track of characters.
2. When encountering a star, pop the last character from the stack.
3. Continue building the result with characters left in the stack.

Java Code:

```java
import java.util.Stack;

public class RemoveStars {
    public static String removeStars(String s) {
        Stack<Character> stack = new Stack<>();

        // Traverse each character in the string
        for (char c : s.toCharArray()) {
            if (c == '*') {
                if (!stack.isEmpty()) {
                    stack.pop(); // Remove the last character
                }
            } else {
                stack.push(c); // Add the character to the stack
            }
        }

        // Build the final result
        StringBuilder result = new StringBuilder();
        while (!stack.isEmpty()) {
            result.append(stack.pop());
        }
        return result.reverse().toString(); // Reverse to maintain original order
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the length of the string.
- **Space Complexity:** O(n) in the worst case (if there are no stars).

7. Trees

Concept

Trees are hierarchical structures that consist of nodes, with each node having zero or more children. They are used to represent data with a parent-child relationship.

Example Problem: Maximum Depth of Binary Tree

Problem Statement: Find the maximum depth of a binary tree. The depth is the number of nodes along the longest path from the root node down to the farthest leaf node. For example, for a tree with root value 3, left child 9, and right child 20 (with right child having children 15 and 7), the maximum depth is 3.

Mathematical Logic:

1. Use a recursive approach to explore both left and right subtrees.
2. The depth of a node is 1 plus the maximum depth of its children.

Java Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class MaxDepth {
    public static int maxDepth(TreeNode root) {
        if (root == null) return 0; // Base case: if the node is null
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right)); // 1 +
max depth of children
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the number of nodes in the tree.
- **Space Complexity:** O(h) for the recursion stack, where h is the height of the tree.

---

8. Graphs

Concept

Graphs consist of vertices (nodes) and edges (connections between nodes). They are used to represent relationships and are fundamental in many applications.

Example Problem: Number of Islands

Problem Statement: Given a 2D grid of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. For example, the grid below has 3 islands:

```
11000
11000
00100
00011
```

Mathematical Logic:

1. Traverse the grid using Depth First Search (DFS) to explore all connected '1's, marking them as visited by changing them to '0'.
2. Count how many times a DFS starts, which corresponds to the number of islands.

Java Code:

```java
public class NumberOfIslands {
    public static int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;
        int count = 0;

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (grid[i][j] == '1') {
                    count++;
                    dfs(grid, i, j); // Call DFS to mark the entire island
                }
            }
        }
        return count;
    }

    private static void dfs(char[][] grid, int i, int j) {
        // Check bounds and whether the cell is land
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i]
[j] == '0') {
            return;
        }
        grid[i][j] = '0'; // Mark the cell as visited

        // Explore all four directions
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n * m), where n is the number of rows and m is the number of columns in the grid.
- **Space Complexity:** O(n * m) in the worst case for the recursion stack.

---

9. Binary Search

Concept

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half.

Example Problem: Search in Rotated Sorted Array

Problem Statement: Given a rotated sorted array, search for a target value. If found, return its index; otherwise, return -1. For example, in the array [4,5,6,7,0,1,2] with target 0, the index is 4.

Mathematical Logic:

1. Identify the mid-point of the current search range.
2. Determine which side of the array (left or right) is sorted.
3. Decide which side to continue the search based on the sorted side and the target.

Java Code:

```java
public class SearchRotatedArray {
    public static int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid; // Target found
            }

            // Determine the sorted side
            if (nums[left] <= nums[mid]) { // Left side is sorted
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1; // Search in left
                } else {
                    left = mid + 1; // Search in right
                }
            } else { // Right side is sorted
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1; // Search in right
                } else {
                    right = mid - 1; // Search in left
                }
            }
        }
        return -1; // Target not found
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(log n), where n is the number of elements in the array.
- **Space Complexity:** O(1), since we are using only a few variables.

---

10. Dynamic Programming

Concept

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations.

Example Problem: Climbing Stairs

Problem Statement: You are climbing a staircase with n steps, and you can either take 1 step or 2 steps at a time. Calculate the number of distinct ways to reach the top. For n = 3, the result is 3 (ways: 1+1+1, 1+2, 2+1).

Mathematical Logic:

1. Let `dp[i]` represent the number of ways to reach step i.
2. The recurrence relation is `dp[i] = dp[i-1] + dp[i-2]`, as you can reach step i from either step i-1 or step i-2.
3. Base cases: `dp[0] = 1`, `dp[1] = 1`.

Java Code:

```java
public class ClimbingStairs {
    public static int climbStairs(int n) {
        if (n <= 1) return 1; // Base cases
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2]; // Calculate ways to reach step i
        }
        return dp[n];
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the number of steps.
- **Space Complexity:** O(n) for the dp array.

---

11. Backtracking

Concept

Backtracking is an algorithmic technique for solving problems incrementally, trying partial solutions and eliminating those that fail to satisfy the conditions of the problem.

Example Problem: Permutations

Problem Statement: Given an array of distinct integers, return all possible permutations. For example, for the array [1, 2, 3], the permutations are [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]].

Mathematical Logic:

1. Use a recursive function to build permutations.
2. At each step, try adding each unused number to the current permutation.
3. When the current permutation reaches the desired length, add it to the result.

Java Code:

```java
import java.util.ArrayList;
import java.util.List;

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

    private static void backtrack(List<List<Integer>> result, List<Integer>
tempList, int[] nums) {
        if (tempList.size() == nums.length) {
            result.add(new ArrayList<>(tempList)); // Found a permutation
            return;
        }
        for (int num : nums) {
            if (tempList.contains(num)) continue; // Skip used numbers
            tempList.add(num); // Choose the number
            backtrack(result, tempList, nums); // Recurse
            tempList.remove(tempList.size() - 1); // Undo the choice
        }
    }
}
```

Complexity Analysis:

- **Time Complexity:** O(n!), where n is the number of elements in the array.
- **Space Complexity:** O(n) for the recursion stack.