COURSE NAME: CSE230

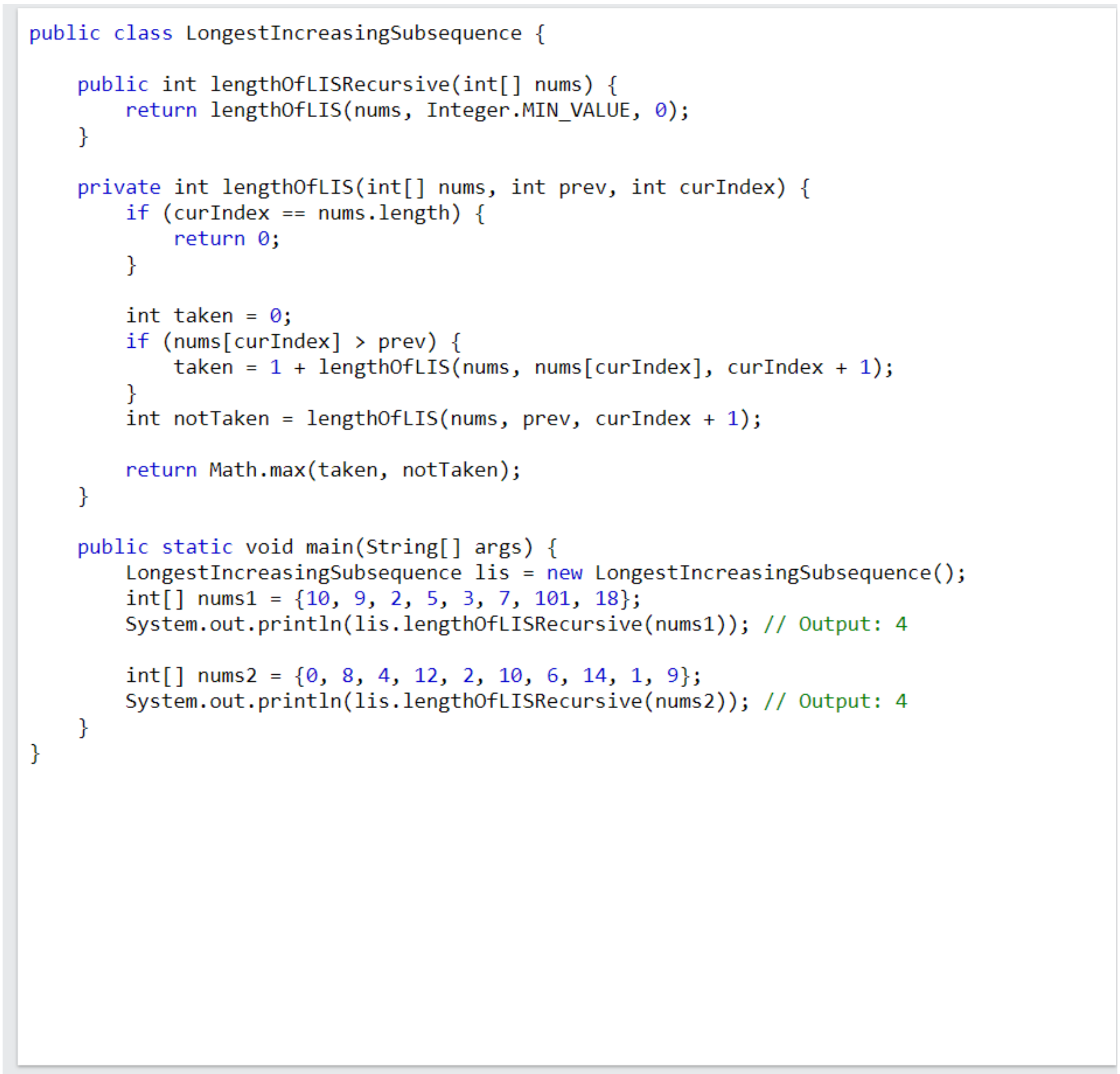
NAME: UDAY SINGH SHEKHAWAT

REG.NO: 12212334

ASSIGNENT NUMBER: 3

Q1) You are given an array of integers called ***nums***. Your task is to find the length of the longest increasing subsequence (LIS) within **nums**.

Ans1)



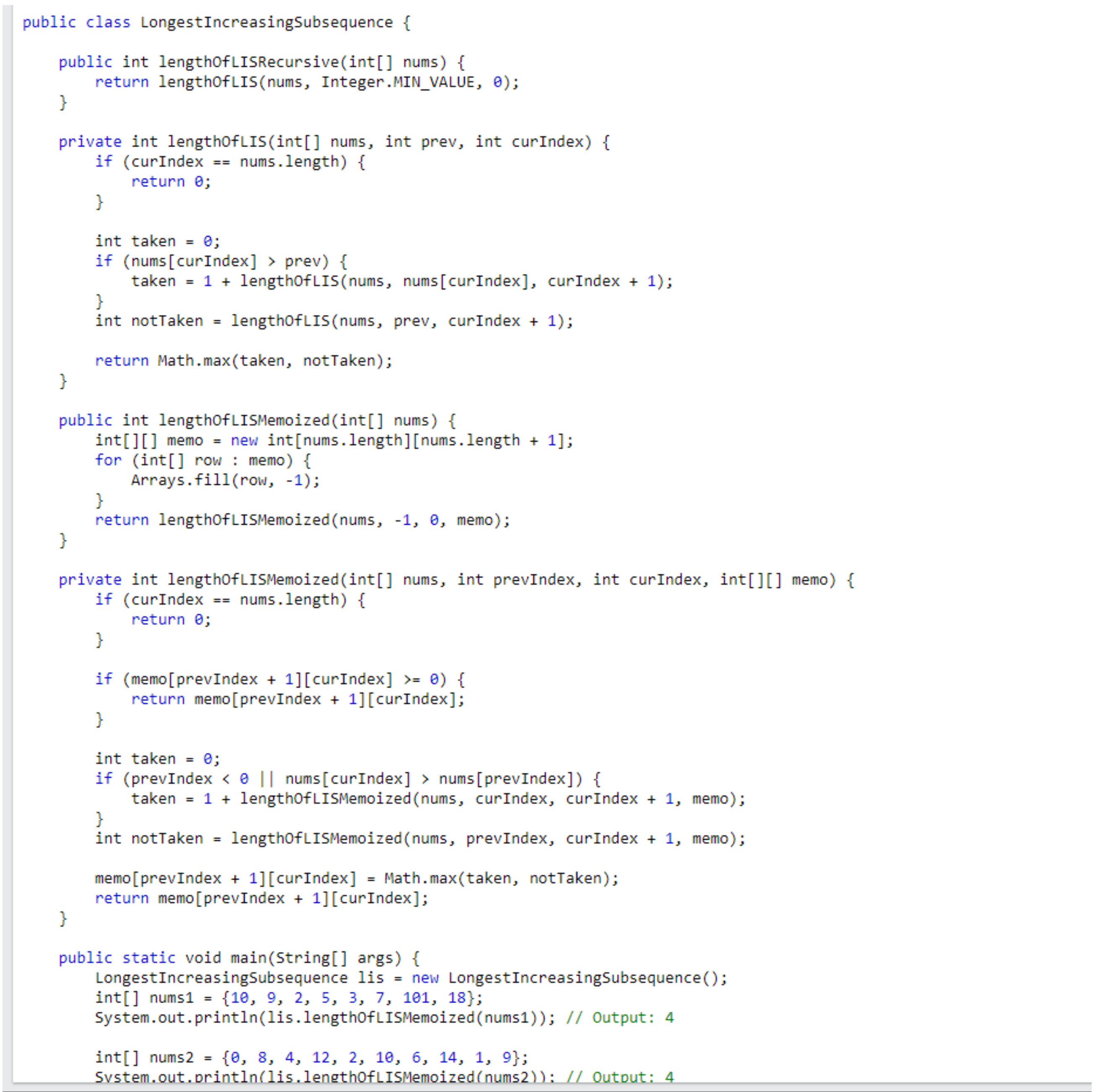
```java
public class LongestIncreasingSubsequence {

    public int lengthOfLISRecursive(int[] nums) {
        return lengthOfLIS(nums, Integer.MIN_VALUE, 0);
    }

    private int lengthOfLIS(int[] nums, int prev, int curIndex) {
        if (curIndex == nums.length) {
            return 0;
        }

        int taken = 0;
        if (nums[curIndex] > prev) {
            taken = 1 + lengthOfLIS(nums, nums[curIndex], curIndex + 1);
        }
        int notTaken = lengthOfLIS(nums, prev, curIndex + 1);

        return Math.max(taken, notTaken);
    }

    public static void main(String[] args) {
        LongestIncreasingSubsequence lis = new LongestIncreasingSubsequence();
        int[] nums1 = {10, 9, 2, 5, 3, 7, 101, 18};
        System.out.println(lis.lengthOfLISRecursive(nums1)); // Output: 4

        int[] nums2 = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9};
        System.out.println(lis.lengthOfLISRecursive(nums2)); // Output: 4
    }
}
```

[A]. Recursive Function: The function lengthOfLISRecursive takes the array nums as input and returns the length of the longest increasing subsequence. It calls the private helper function lengthOfLIS recursively.

[B]. Explanation with Example:

For the array [10, 9, 2, 5, 3, 7, 101, 18], the longest increasing subsequence starts with 2 and includes 3, 7, and 101, resulting in a length of 4.

For the array [0, 8, 4, 12, 2, 10, 6, 14, 1, 9], the longest increasing subsequence starts with 0 and includes 2, 6, and 9, resulting in a length of 4.



```java
public class LongestIncreasingSubsequence {

    public int lengthOfLISRecursive(int[] nums) {
        return lengthOfLIS(nums, Integer.MIN_VALUE, 0);
    }

    private int lengthOfLIS(int[] nums, int prev, int curIndex) {
        if (curIndex == nums.length) {
            return 0;
        }

        int taken = 0;
        if (nums[curIndex] > prev) {
            taken = 1 + lengthOfLIS(nums, nums[curIndex], curIndex + 1);
        }
        int notTaken = lengthOfLIS(nums, prev, curIndex + 1);

        return Math.max(taken, notTaken);
    }

    public int lengthOfLISMemoized(int[] nums) {
        int[][] memo = new int[nums.length][nums.length + 1];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return lengthOfLISMemoized(nums, -1, 0, memo);
    }

    private int lengthOfLISMemoized(int[] nums, int prevIndex, int curIndex, int[][] memo) {
        if (curIndex == nums.length) {
            return 0;
        }

        if (memo[prevIndex + 1][curIndex] >= 0) {
            return memo[prevIndex + 1][curIndex];
        }

        int taken = 0;
        if (prevIndex < 0 || nums[curIndex] > nums[prevIndex]) {
            taken = 1 + lengthOfLISMemoized(nums, curIndex, curIndex + 1, memo);
        }
        int notTaken = lengthOfLISMemoized(nums, prevIndex, curIndex + 1, memo);

        memo[prevIndex + 1][curIndex] = Math.max(taken, notTaken);
        return memo[prevIndex + 1][curIndex];
    }

    public static void main(String[] args) {
        LongestIncreasingSubsequence lis = new LongestIncreasingSubsequence();
        int[] nums1 = {10, 9, 2, 5, 3, 7, 101, 18};
        System.out.println(lis.lengthOfLISMemoized(nums1)); // Output: 4

        int[] nums2 = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9};
        System.out.println(lis.lengthOfLISMemoized(nums2)); // Output: 4
```

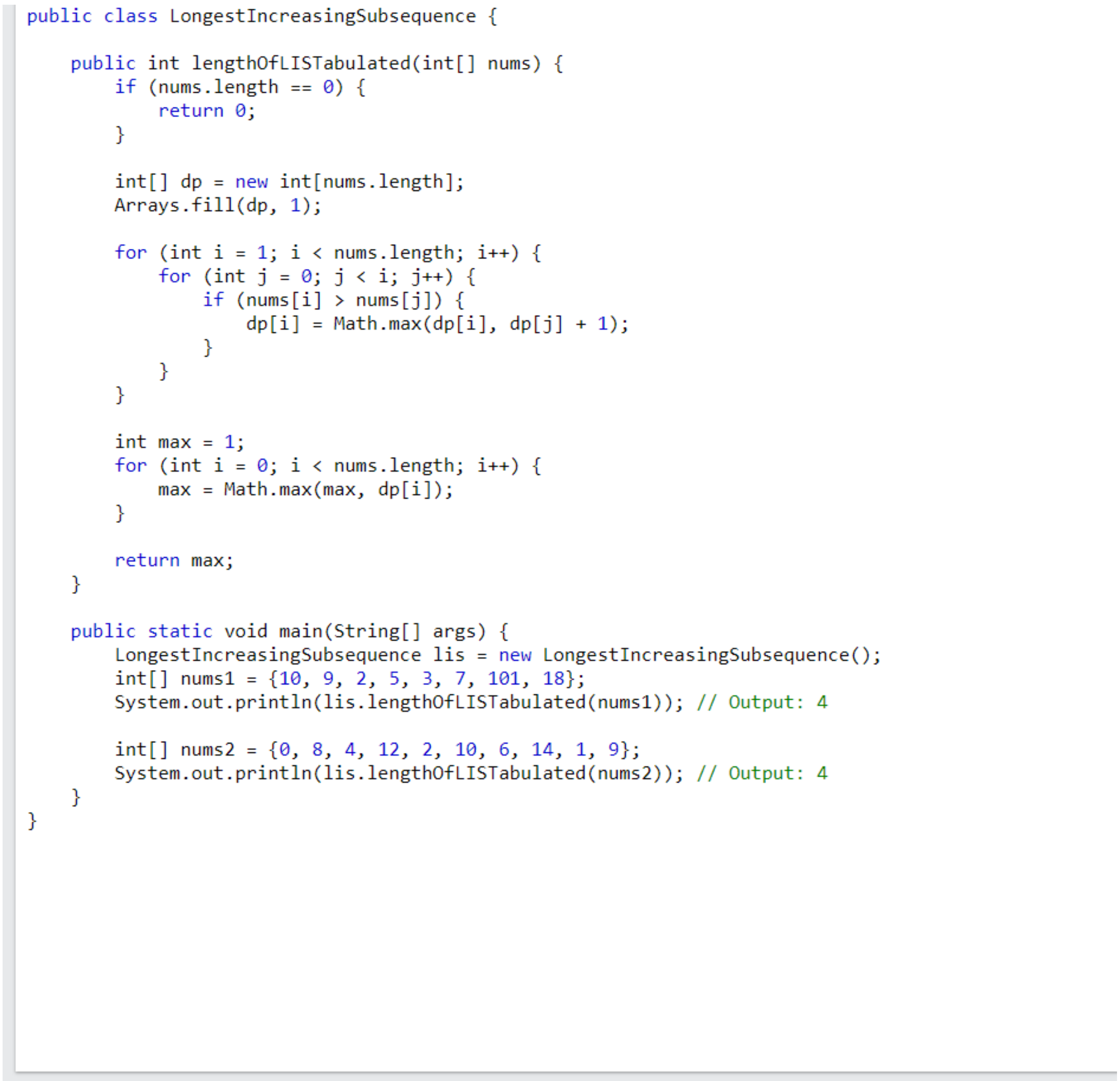
[C]. Memorized Version:

The function lengthOfLISMemoized uses memoization to store the results of overlapping subproblems, preventing redundant calculations.

[D]. Explanation with Example:

For the array [10, 9, 2, 5, 3, 7, 101, 18], when calculating the length of the LIS starting at index 0 (10), the function recursively calculates the LIS starting at index 2 (2), 3 (5), 4 (3), and 5 (7). These calculations are repeated for other starting indices. By memoizing these results, we avoid redundant calculations and improve efficiency.

For the array [0, 8, 4, 12, 2, 10, 6, 14, 1, 9], similar memoization occurs for overlapping subproblems.



```java
public class LongestIncreasingSubsequence {

    public int lengthOfLISTabulated(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }

        int[] dp = new int[nums.length];
        Arrays.fill(dp, 1);

        for (int i = 1; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
        }

        int max = 1;
        for (int i = 0; i < nums.length; i++) {
            max = Math.max(max, dp[i]);
        }

        return max;
    }

    public static void main(String[] args) {
        LongestIncreasingSubsequence lis = new LongestIncreasingSubsequence();
        int[] nums1 = {10, 9, 2, 5, 3, 7, 101, 18};
        System.out.println(lis.lengthOfLISTabulated(nums1)); // Output: 4

        int[] nums2 = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9};
        System.out.println(lis.lengthOfLISTabulated(nums2)); // Output: 4
    }
}
```

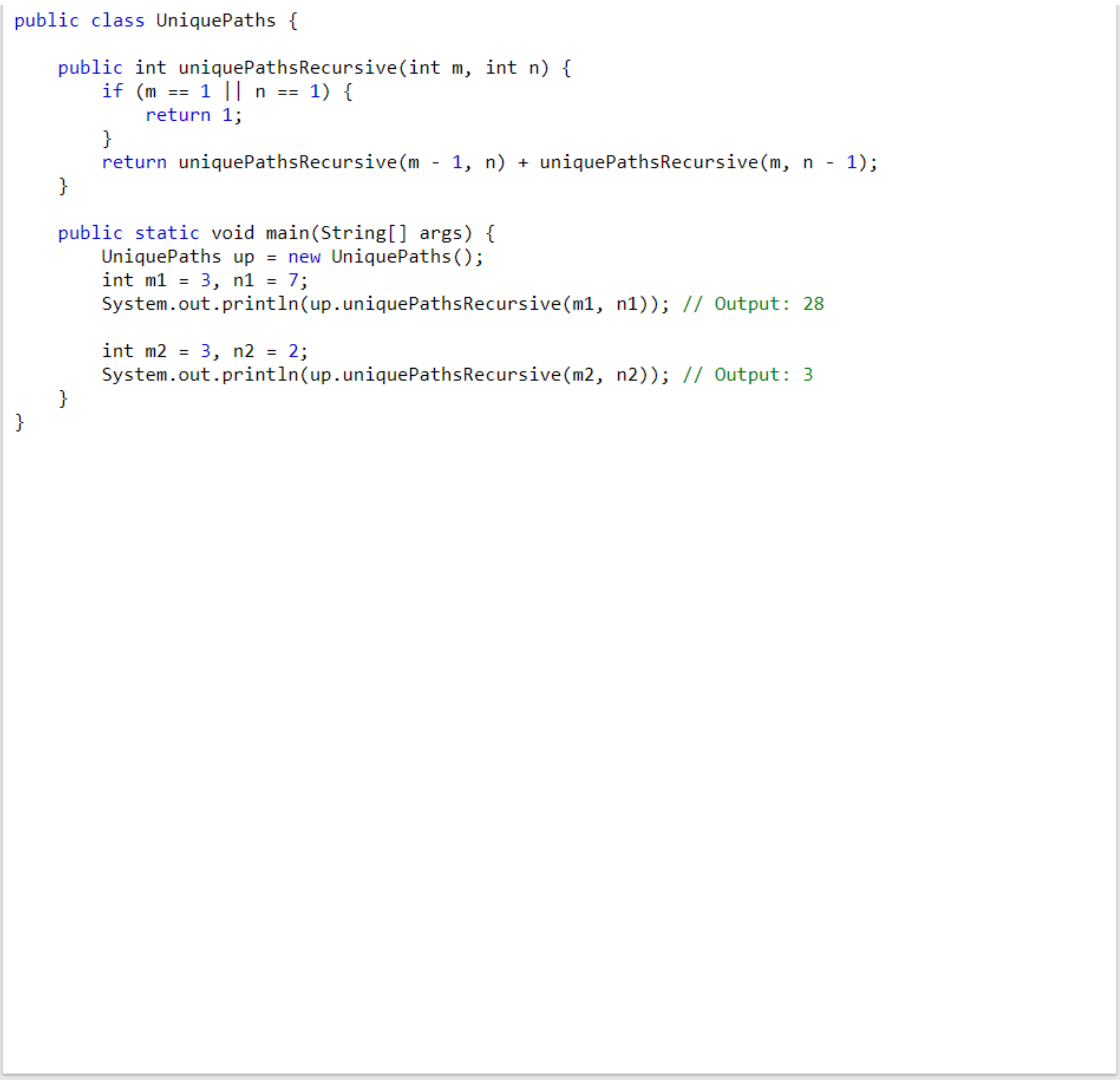
[E]. Tabulated Version:

The function lengthOfLISTabulated uses a dynamic programming approach to fill a 1D array dp, where dp[i] represents the length of the LIS ending at index i. It iterates through the array, updating dp[i] based on previous calculated values.

[F]. Explanation with Example:
For the array [10, 9, 2, 5, 3, 7, 101, 18], the dp array after computation will be [1, 1, 1, 2, 2, 3, 4, 4], where dp[6] represents the length of the LIS ending at index 6 (101), which is 4.
For the array [0, 8, 4, 12, 2, 10, 6, 14, 1, 9], the dp array after computation will also have dp[6] as 4, representing the length of the LIS ending at index 6 (10).

Q2) A robot is located at the top-left corner of a m x n grid. The robot can only move either down or right at any point in time. Your task is to determine how many unique paths there are for the robot to reach the bottom-right corner of the grid from the top-left corner.

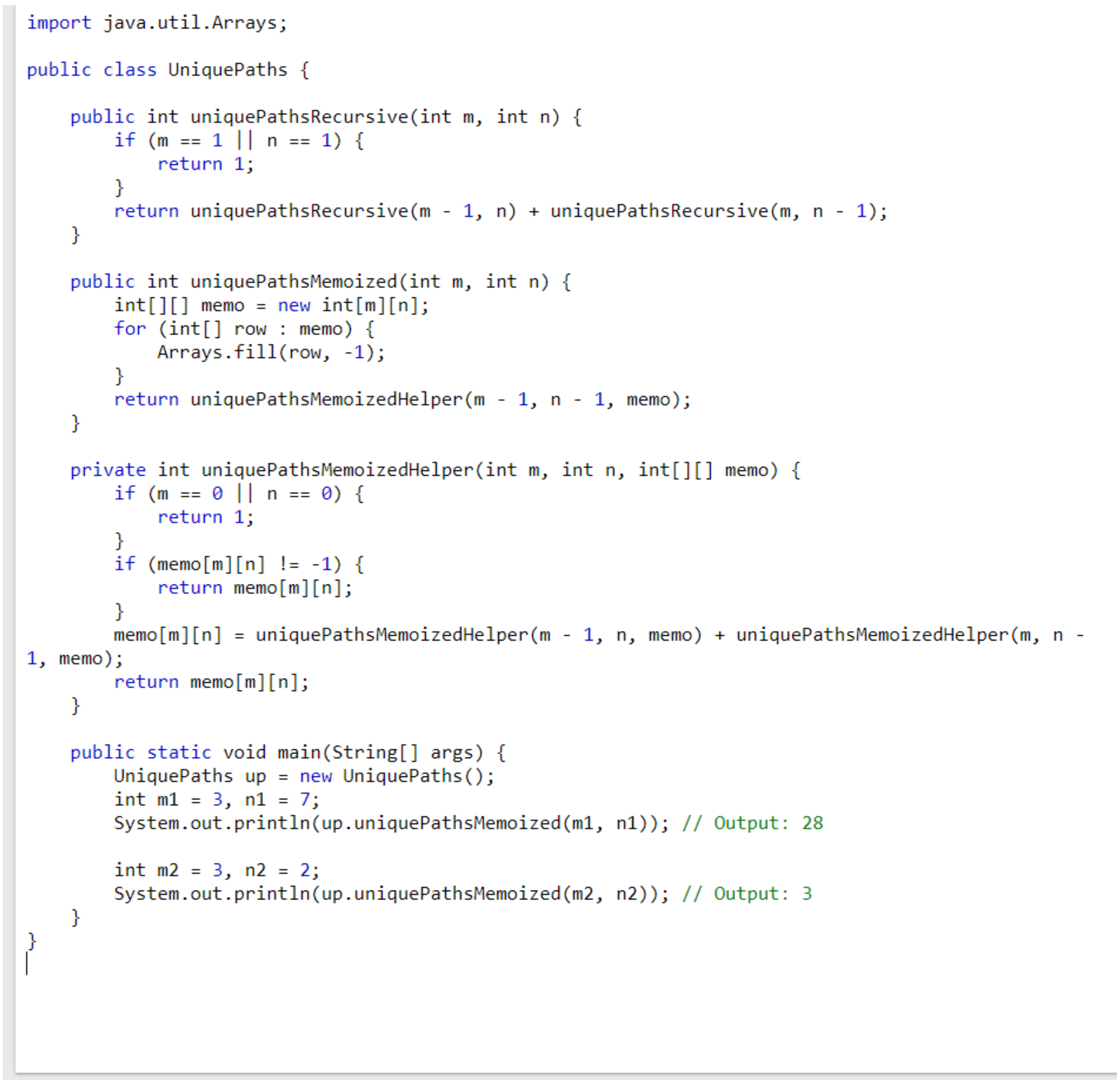
Ans2)



```java
public class UniquePaths {

    public int uniquePathsRecursive(int m, int n) {
        if (m == 1 || n == 1) {
            return 1;
        }
        return uniquePathsRecursive(m - 1, n) + uniquePathsRecursive(m, n - 1);
    }

    public static void main(String[] args) {
        UniquePaths up = new UniquePaths();
        int m1 = 3, n1 = 7;
        System.out.println(up.uniquePathsRecursive(m1, n1)); // Output: 28

        int m2 = 3, n2 = 2;
        System.out.println(up.uniquePathsRecursive(m2, n2)); // Output: 3
    }
}
```

[A]. Recursive Function:
The function uniquePathsRecursive takes two parameters m and n representing the dimensions of the grid and returns the number of unique paths from the top-left corner to the bottom-right corner. It recursively calculates the number of unique paths by exploring two possible moves: either moving down or moving right.

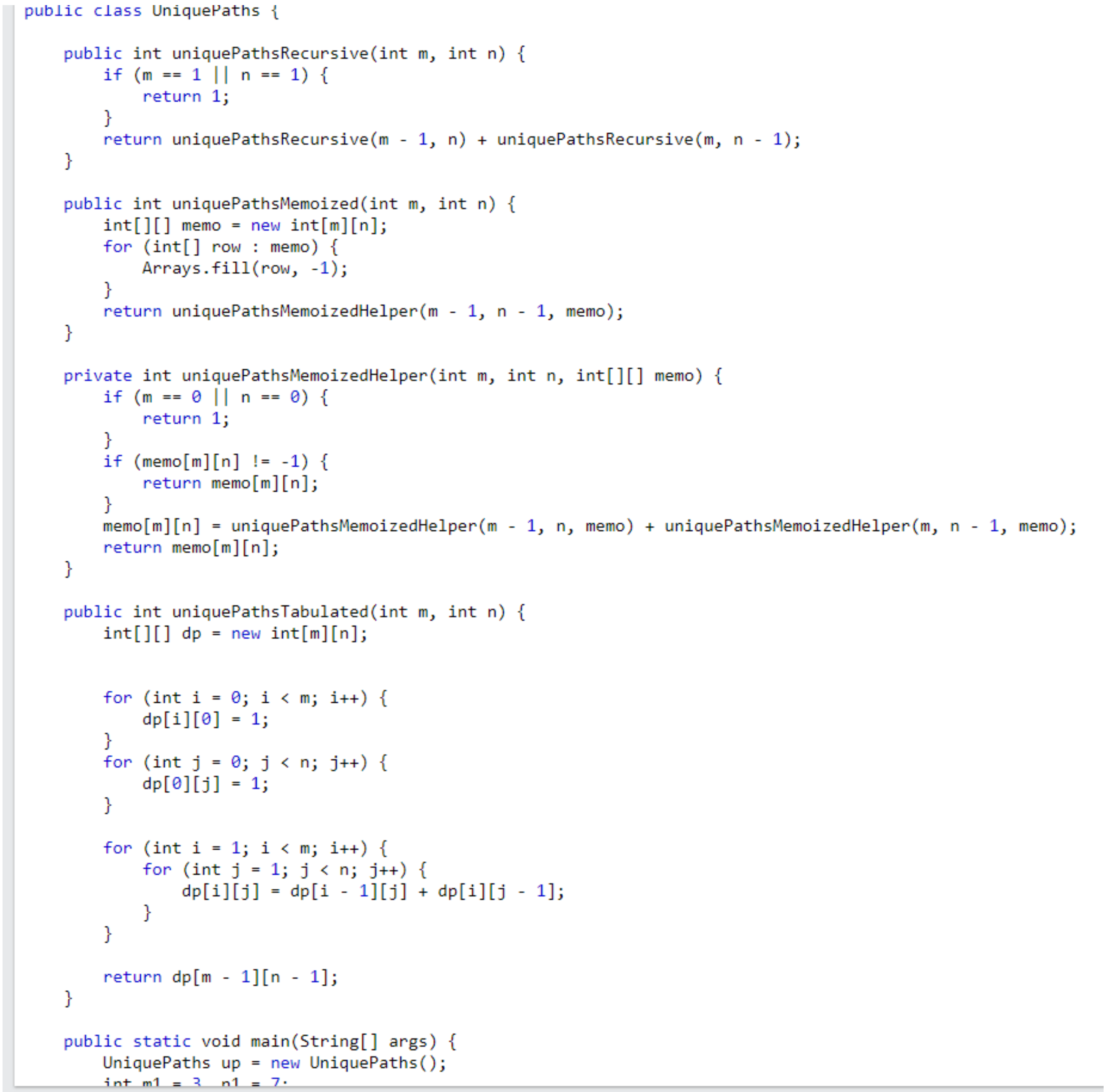
[B]. Explanation with Examples:
For a 3x7 grid, the robot can move either down or right. It needs to reach the bottom-right corner from the top-left corner. By recursively exploring all possible moves, we find that there are 28 unique paths.
For a 3x2 grid, similarly, there are 3 unique paths for the robot to reach the bottom-right corner from the top-left corner.



```java
import java.util.Arrays;

public class UniquePaths {

    public int uniquePathsRecursive(int m, int n) {
        if (m == 1 || n == 1) {
            return 1;
        }
        return uniquePathsRecursive(m - 1, n) + uniquePathsRecursive(m, n - 1);
    }

    public int uniquePathsMemoized(int m, int n) {
        int[][] memo = new int[m][n];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return uniquePathsMemoizedHelper(m - 1, n - 1, memo);
    }

    private int uniquePathsMemoizedHelper(int m, int n, int[][] memo) {
        if (m == 0 || n == 0) {
            return 1;
        }
        if (memo[m][n] != -1) {
            return memo[m][n];
        }
        memo[m][n] = uniquePathsMemoizedHelper(m - 1, n, memo) + uniquePathsMemoizedHelper(m, n -
1, memo);
        return memo[m][n];
    }

    public static void main(String[] args) {
        UniquePaths up = new UniquePaths();
        int m1 = 3, n1 = 7;
        System.out.println(up.uniquePathsMemoized(m1, n1)); // Output: 28

        int m2 = 3, n2 = 2;
        System.out.println(up.uniquePathsMemoized(m2, n2)); // Output: 3
    }
}
```

[C]. Memorized Version:
 The function uniquePathsMemoized uses memoization to store the results of overlapping subproblems, improving the efficiency of the algorithm.

[D]. Explanation with Examples:
For a 3x7 grid, when calculating the unique paths, many subproblems overlap. For example, when calculating the paths for the cell (2, 6), it depends on the paths from (1, 6) and (2, 5), which have already been computed. By memoizing these results, redundant calculations are avoided, improving the efficiency of the algorithm.
Similarly, for a 3x2 grid, memoization helps in avoiding redundant calculations and speeds up the computation of unique paths.



```java
public class UniquePaths {

    public int uniquePathsRecursive(int m, int n) {
        if (m == 1 || n == 1) {
            return 1;
        }
        return uniquePathsRecursive(m - 1, n) + uniquePathsRecursive(m, n - 1);
    }

    public int uniquePathsMemoized(int m, int n) {
        int[][] memo = new int[m][n];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return uniquePathsMemoizedHelper(m - 1, n - 1, memo);
    }

    private int uniquePathsMemoizedHelper(int m, int n, int[][] memo) {
        if (m == 0 || n == 0) {
            return 1;
        }
        if (memo[m][n] != -1) {
            return memo[m][n];
        }
        memo[m][n] = uniquePathsMemoizedHelper(m - 1, n, memo) + uniquePathsMemoizedHelper(m, n - 1, memo);
        return memo[m][n];
    }

    public int uniquePathsTabulated(int m, int n) {
        int[][] dp = new int[m][n];


        for (int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }
        for (int j = 0; j < n; j++) {
            dp[0][j] = 1;
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

        return dp[m - 1][n - 1];
    }

    public static void main(String[] args) {
        UniquePaths up = new UniquePaths();
        int m1 = 3, n1 = 7;
```

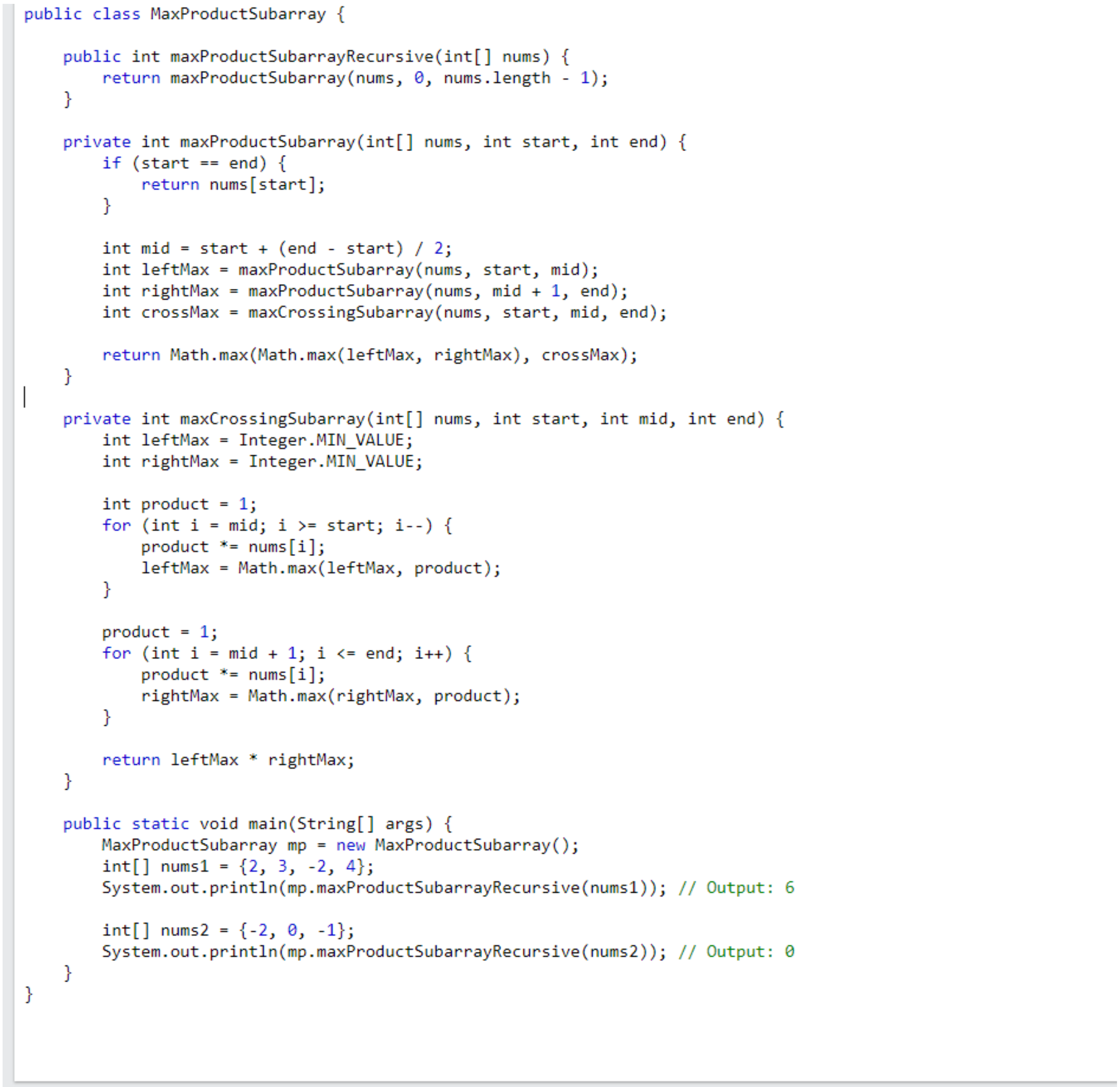
[E]. Tabulated Version:
The function uniquePathsTabulated uses dynamic programming to fill a 2D array dp, where dp[i][j] represents the number of unique paths to reach cell (i, j) from the top-left corner. It iterates through the array, filling each cell based on the number of unique paths from the cells above and to the left of it.

[F]. Explanation with Examples:
For a 3x7 grid, the tabulated version calculates the number of unique paths iteratively. It fills the dp array by considering the number of unique paths from the cells above and to the left of each cell. Finally, it returns the value in the bottom-right corner of the dp array, which represents the total number of unique paths.

Similarly, for a 3x2 grid, the tabulated version efficiently computes the number of unique paths using dynamic programming.

Q3) Given an array of integers, find the contiguous subarray with the largest
Product.

Ans3)
[A]. Recursive Function:
The function maxProductSubarrayRecursive takes the array nums as input and returns the maximum product of the contiguous subarray. It recursively divides the array into two halves and calculates the maximum product of the subarrays in the left half, right half, and the cross subarray.



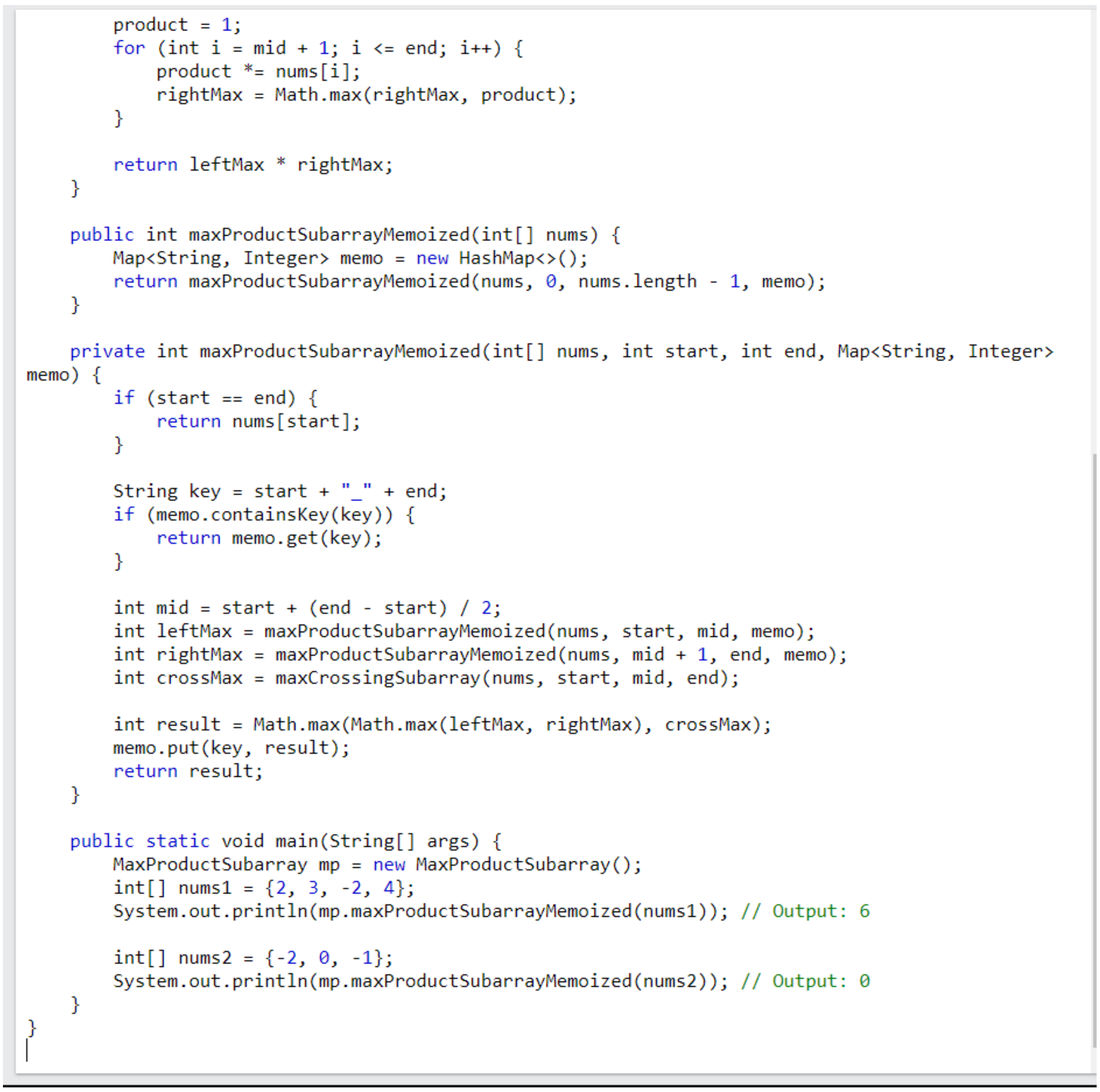
```java
public class MaxProductSubarray {

    public int maxProductSubarrayRecursive(int[] nums) {
        return maxProductSubarray(nums, 0, nums.length - 1);
    }

    private int maxProductSubarray(int[] nums, int start, int end) {
        if (start == end) {
            return nums[start];
        }

        int mid = start + (end - start) / 2;
        int leftMax = maxProductSubarray(nums, start, mid);
        int rightMax = maxProductSubarray(nums, mid + 1, end);
        int crossMax = maxCrossingSubarray(nums, start, mid, end);

        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }

    private int maxCrossingSubarray(int[] nums, int start, int mid, int end) {
        int leftMax = Integer.MIN_VALUE;
        int rightMax = Integer.MIN_VALUE;

        int product = 1;
        for (int i = mid; i >= start; i--) {
            product *= nums[i];
            leftMax = Math.max(leftMax, product);
        }

        product = 1;
        for (int i = mid + 1; i <= end; i++) {
            product *= nums[i];
            rightMax = Math.max(rightMax, product);
        }

        return leftMax * rightMax;
    }

    public static void main(String[] args) {
        MaxProductSubarray mp = new MaxProductSubarray();
        int[] nums1 = {2, 3, -2, 4};
        System.out.println(mp.maxProductSubarrayRecursive(nums1)); // Output: 6

        int[] nums2 = {-2, 0, -1};
        System.out.println(mp.maxProductSubarrayRecursive(nums2)); // Output: 0
    }
}
```

[B]. Explanation with Examples:

For the array [2, 3, -2, 4], the function recursively divides it into [2, 3] and [-2, 4]. It then calculates the maximum product of these subarrays and their cross subarray. The maximum product is found to be 6 from the subarray [2, 3].

For the array [-2, 0, -1], similar recursive steps are followed, and the maximum product is found to be 0 from the subarray [-2, 0].

```java
import java.util.HashMap;
import java.util.Map;

public class MaxProductSubarray {

    public int maxProductSubarrayRecursive(int[] nums) {
        return maxProductSubarray(nums, 0, nums.length - 1);
    }

    private int maxProductSubarray(int[] nums, int start, int end) {
        if (start == end) {
            return nums[start];
        }

        int mid = start + (end - start) / 2;
        int leftMax = maxProductSubarray(nums, start, mid);
        int rightMax = maxProductSubarray(nums, mid + 1, end);
        int crossMax = maxCrossingSubarray(nums, start, mid, end);

        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }

    private int maxCrossingSubarray(int[] nums, int start, int mid, int end) {
        int leftMax = Integer.MIN_VALUE;
        int rightMax = Integer.MIN_VALUE;

        int product = 1;
        for (int i = mid; i >= start; i--) {
            product *= nums[i];
            leftMax = Math.max(leftMax, product);
        }

        product = 1;
        for (int i = mid + 1; i <= end; i++) {
            product *= nums[i];
            rightMax = Math.max(rightMax, product);
        }

        return leftMax * rightMax;
    }

    public int maxProductSubarrayMemoized(int[] nums) {
        Map<String, Integer> memo = new HashMap<>();
        return maxProductSubarrayMemoized(nums, 0, nums.length - 1, memo);
    }
```

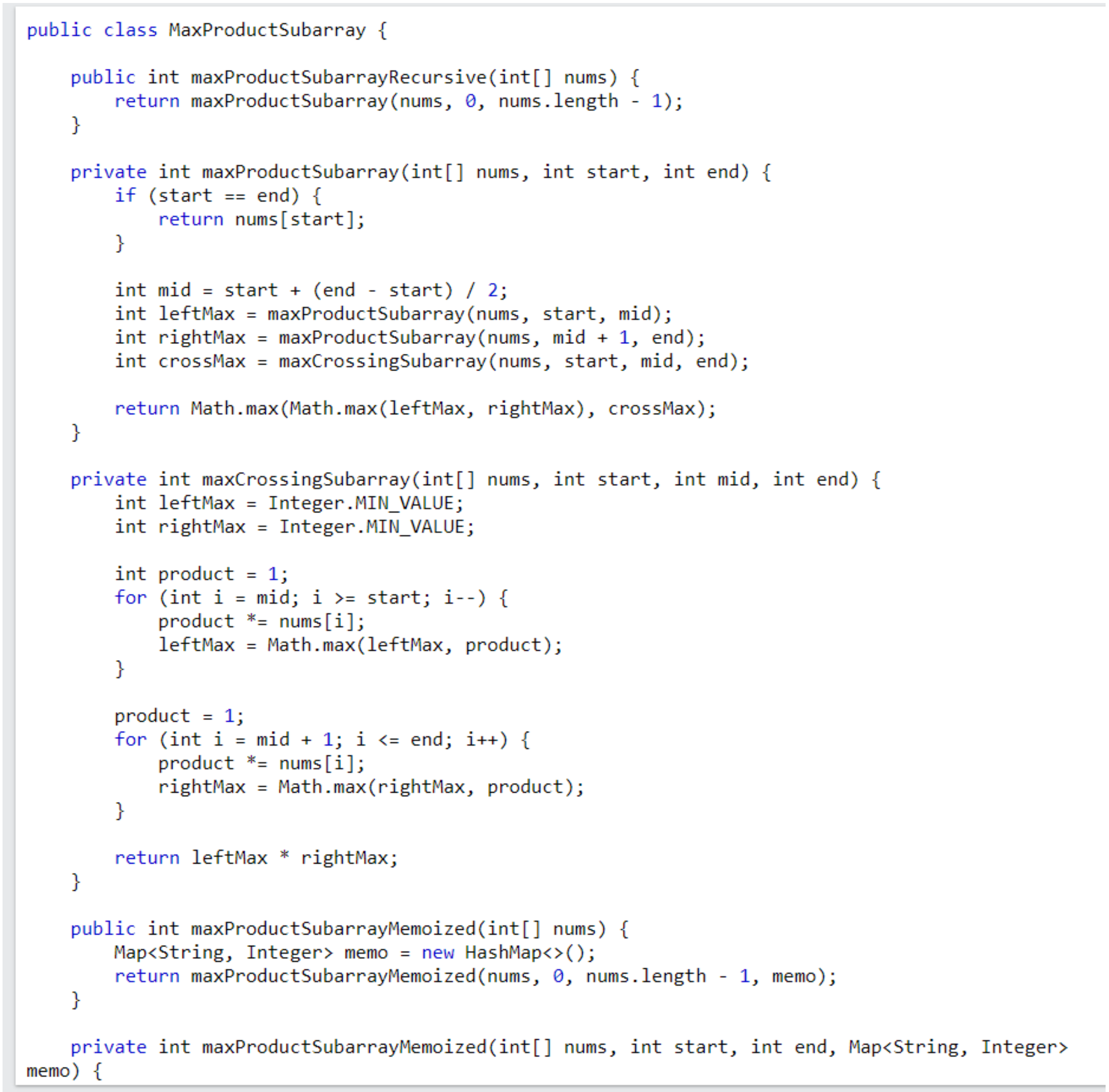


```java
        product = 1;
        for (int i = mid + 1; i <= end; i++) {
            product *= nums[i];
            rightMax = Math.max(rightMax, product);
        }

        return leftMax * rightMax;
    }

    public int maxProductSubarrayMemoized(int[] nums) {
        Map<String, Integer> memo = new HashMap<>();
        return maxProductSubarrayMemoized(nums, 0, nums.length - 1, memo);
    }

    private int maxProductSubarrayMemoized(int[] nums, int start, int end, Map<String, Integer>
memo) {
        if (start == end) {
            return nums[start];
        }

        String key = start + "_" + end;
        if (memo.containsKey(key)) {
            return memo.get(key);
        }

        int mid = start + (end - start) / 2;
        int leftMax = maxProductSubarrayMemoized(nums, start, mid, memo);
        int rightMax = maxProductSubarrayMemoized(nums, mid + 1, end, memo);
        int crossMax = maxCrossingSubarray(nums, start, mid, end);

        int result = Math.max(Math.max(leftMax, rightMax), crossMax);
        memo.put(key, result);
        return result;
    }

    public static void main(String[] args) {
        MaxProductSubarray mp = new MaxProductSubarray();
        int[] nums1 = {2, 3, -2, 4};
        System.out.println(mp.maxProductSubarrayMemoized(nums1)); // Output: 6

        int[] nums2 = {-2, 0, -1};
        System.out.println(mp.maxProductSubarrayMemoized(nums2)); // Output: 0
    }
}
```

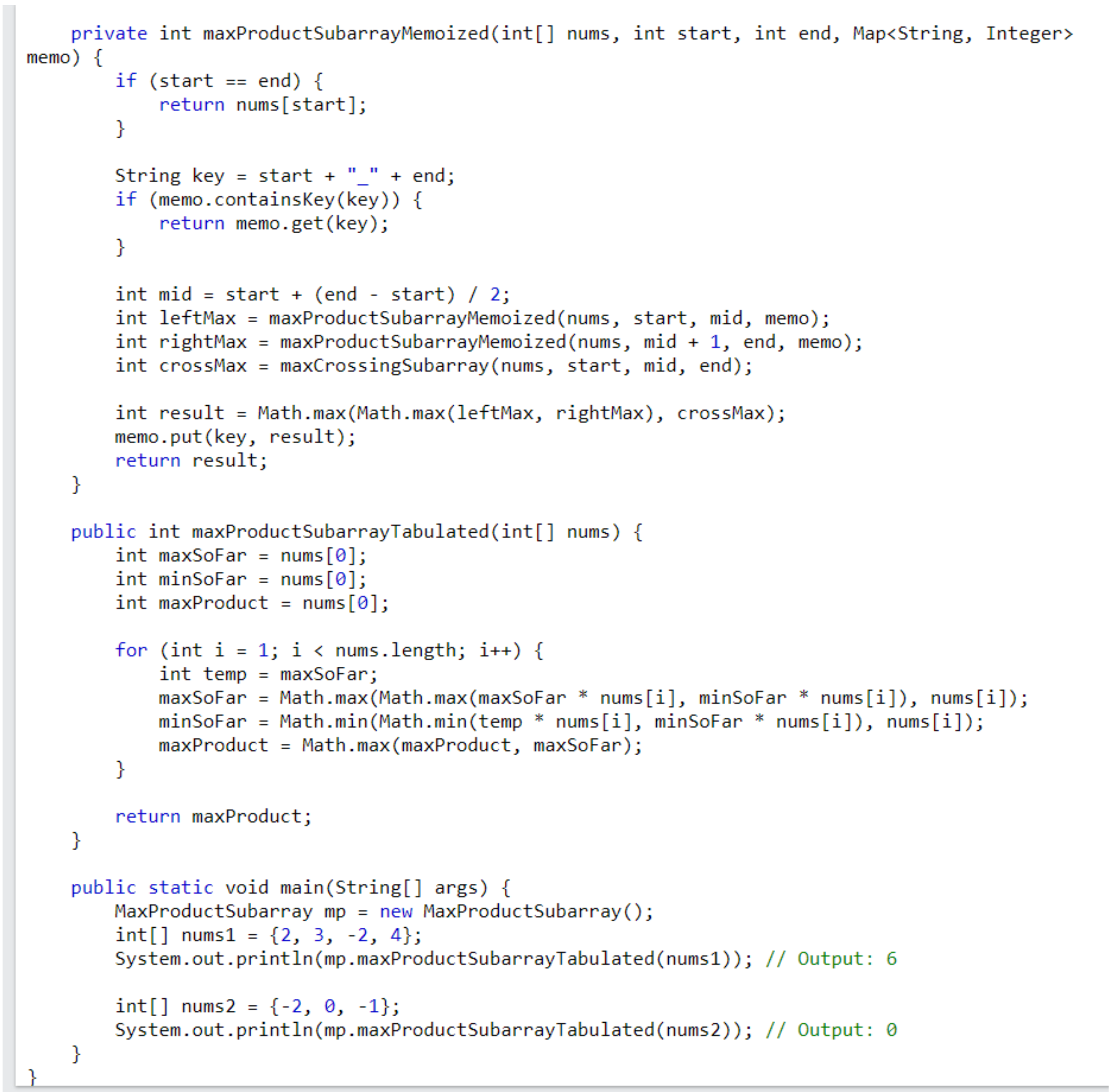
[C]. Memorized Version:
The function maxProductSubarrayMemoized uses memoization to store the results of overlapping subproblems, improving the efficiency of the algorithm.

[D]. Explanation with Examples:
For the array [2, 3, -2, 4], when calculating the maximum product of subarrays, many subproblems overlap. For example, when calculating the maximum product of the subarray [2, 3], it depends on the maximum product of the subarrays [2] and [3], which have already been computed. By memoizing these results, redundant calculations are avoided, improving the efficiency of the algorithm.

Similarly, for the array [-2, 0, -1], memoization helps in avoiding redundant calculations and speeds up the computation of the maximum product of subarrays.



```
public class MaxProductSubarray {

    public int maxProductSubarrayRecursive(int[] nums) {
        return maxProductSubarray(nums, 0, nums.length - 1);
    }

    private int maxProductSubarray(int[] nums, int start, int end) {
        if (start == end) {
            return nums[start];
        }

        int mid = start + (end - start) / 2;
        int leftMax = maxProductSubarray(nums, start, mid);
        int rightMax = maxProductSubarray(nums, mid + 1, end);
        int crossMax = maxCrossingSubarray(nums, start, mid, end);

        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }

    private int maxCrossingSubarray(int[] nums, int start, int mid, int end) {
        int leftMax = Integer.MIN_VALUE;
        int rightMax = Integer.MIN_VALUE;

        int product = 1;
        for (int i = mid; i >= start; i--) {
            product *= nums[i];
            leftMax = Math.max(leftMax, product);
        }

        product = 1;
        for (int i = mid + 1; i <= end; i++) {
            product *= nums[i];
            rightMax = Math.max(rightMax, product);
        }

        return leftMax * rightMax;
    }

    public int maxProductSubarrayMemoized(int[] nums) {
        Map<String, Integer> memo = new HashMap<>();
        return maxProductSubarrayMemoized(nums, 0, nums.length - 1, memo);
    }

    private int maxProductSubarrayMemoized(int[] nums, int start, int end, Map<String, Integer>
memo) {
```



```
    private int maxProductSubarrayMemoized(int[] nums, int start, int end, Map<String, Integer>
memo) {
        if (start == end) {
            return nums[start];
        }

        String key = start + "_" + end;
        if (memo.containsKey(key)) {
            return memo.get(key);
        }

        int mid = start + (end - start) / 2;
        int leftMax = maxProductSubarrayMemoized(nums, start, mid, memo);
        int rightMax = maxProductSubarrayMemoized(nums, mid + 1, end, memo);
        int crossMax = maxCrossingSubarray(nums, start, mid, end);

        int result = Math.max(Math.max(leftMax, rightMax), crossMax);
        memo.put(key, result);
        return result;
    }

    public int maxProductSubarrayTabulated(int[] nums) {
        int maxSoFar = nums[0];
        int minSoFar = nums[0];
        int maxProduct = nums[0];

        for (int i = 1; i < nums.length; i++) {
            int temp = maxSoFar;
            maxSoFar = Math.max(Math.max(maxSoFar * nums[i], minSoFar * nums[i]), nums[i]);
            minSoFar = Math.min(Math.min(temp * nums[i], minSoFar * nums[i]), nums[i]);
            maxProduct = Math.max(maxProduct, maxSoFar);
        }

        return maxProduct;
    }

    public static void main(String[] args) {
        MaxProductSubarray mp = new MaxProductSubarray();
        int[] nums1 = {2, 3, -2, 4};
        System.out.println(mp.maxProductSubarrayTabulated(nums1)); // Output: 6

        int[] nums2 = {-2, 0, -1};
        System.out.println(mp.maxProductSubarrayTabulated(nums2)); // Output: 0
    }
}
```

[E]. Tabulated Version:
The function maxProductSubarrayTabulated uses dynamic programming to iteratively calculate the maximum product of subarrays.

[F]. Explanation with Examples:
For the array [2, 3, -2, 4], the tabulated version efficiently computes the maximum product of subarrays. It iterates through the array, updating maxSoFar and minSoFar variables to keep track of the maximum and minimum product ending at the current position. The final result is stored in maxProduct.
Similarly, for the array [-2, 0, -1], the tabulated version efficiently computes the maximum product of subarrays using dynamic programming.