**C-DAC Mumbai Date 26/09/2024**

**Subject: Algorithm and Data Structure**

**Assignment 1**

**Solve the assignment with following thing to be added in each question.**

**-Program**

**-Flow chart**

**-Explanation**

**-Output**

**-Time and Space complexity**

1. Printing Patterns

Problem: Write a Java program to print patterns such as a right triangle of stars.

**-Program**

public class StarTriangle {

public static void main(String[] args) {

int rows = 5; // number of rows in the triangle

for (int i = 1; i <= rows; i++) {

// print spaces before stars

for (int j = 1; j <= rows - i; j++) {

System.out.print(" ");

}

// print stars

for (int k = 1; k <= i; k++) {

System.out.print("\* ");

}

System.out.println(); // move to next line

}

}

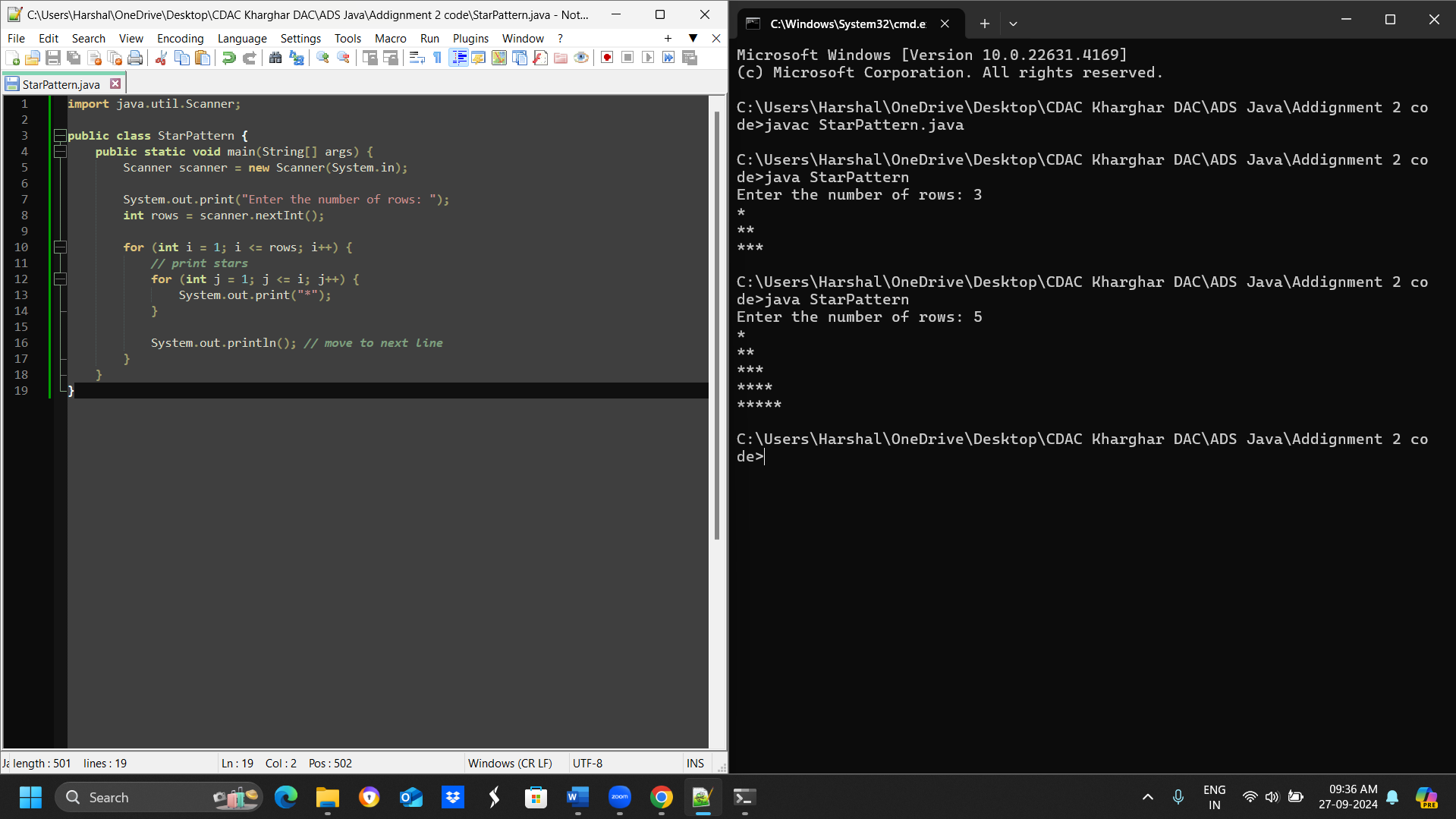
}

-Flow chart

**-Explanation**

* We use a **for** loop to iterate over the number of rows in the triangle.
* Inside the loop, we use two nested **for** loops to print the spaces and stars.
* The first inner loop prints the spaces before the stars, with the number of spaces decreasing by 1 for each row.
* The second inner loop prints the stars, with the number of stars increasing by 1 for each row.
* After printing each row, we move to the next line using **System.out.println()**.

**-Output:-**

****

**-Time and Space complexity**

the time complexity of the program is O(rows^2)

the space complexity is O(1).

Test Cases:

Input: n = 3

Output:

\*

\*\*

\*\*\*

Input: n = 5

Output:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

**2. Remove Array Duplicates**

**Problem: Write a Java program to remove duplicates from a sorted array and return the new length of the array.**

**-Program:-**

import java.util.Scanner;

public class Main {

public static int removeDuplicates(int[] arr) {

if (arr.length == 0) {

return 0;

}

int writeIndex = 1;

for (int readIndex = 1; readIndex < arr.length; readIndex++) {

if (arr[readIndex] != arr[readIndex - 1]) {

arr[writeIndex] = arr[readIndex];

writeIndex++;

}

}

return writeIndex;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the size of the array: ");

int size = scanner.nextInt();

int[] arr = new int[size];

System.out.println("Enter the elements of the array:");

for (int i = 0; i < size; i++) {

arr[i] = scanner.nextInt();

}

int length = removeDuplicates(arr);

System.out.println("Length of array after removing duplicates: " + length);

}

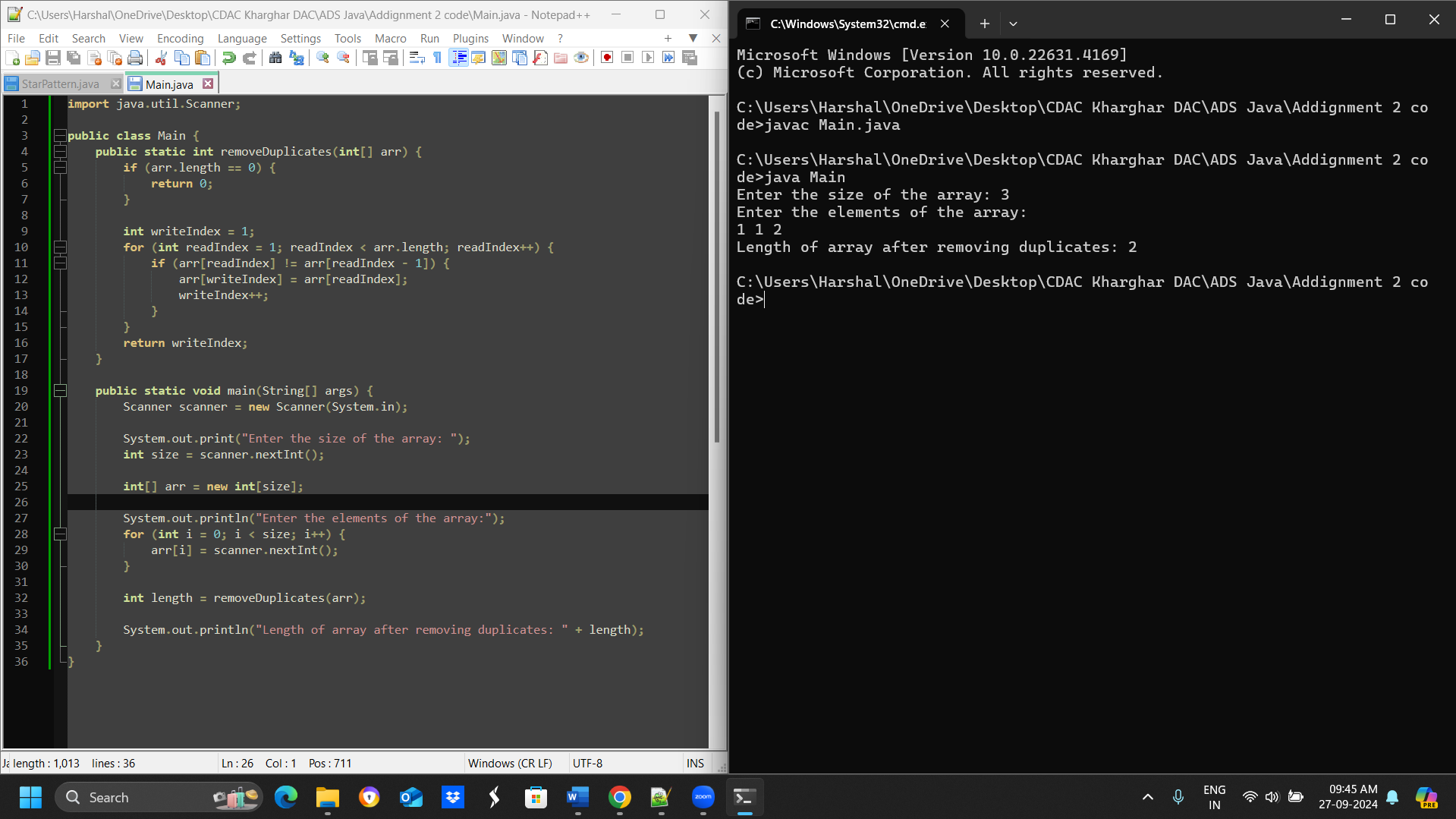
}

**-Flow chart**

**-Explanation**

1. We import the Scanner class to read input from the user.
2. In the main method, we create a Scanner object to read input from the user.
3. We prompt the user to enter the size of the array and store it in the size variable.
4. We create an array of size size and prompt the user to enter the elements of the array.
5. We call the removeDuplicates method to remove duplicates from the array and store the new length in the length variable.
6. Finally, we print the length of the array after removing duplicates.

**-Output**

****

**-Time and Space complexity**

Time complexity: O(n)

Space complexity: O(1)

Test Cases:

Input: arr = [1, 1, 2]

Output: 2

Input: arr = [0, 0, 1, 1, 2, 2, 3, 3]

Output: 4

**3. Remove White Spaces from String**

**Problem: Write a Java program to remove all white spaces from a given string.**

**-Program**

public class RemoveWhitespace {

public static void main(String[] args) {

String str1 = "Hello World";

String str2 = " Java Programming ";

System.out.println(removeWhitespace(str1)); // Output: "HelloWorld"

System.out.println(removeWhitespace(str2)); // Output: "JavaProgramming"

}

public static String removeWhitespace(String str) {

return str.replaceAll("\\s+", "");

}

}

**-Flow chart**

1. Start
2. Input string
3. Replace all whitespace with ""
4. Output modified string
5. End

**-Explanation**

The program uses the replaceAll method with a regex to remove all whitespace characters from the input string.

**-Output**

HelloWorld

JavaProgramming

**-Time and Space complexity**

**Time Complexity**: O(n), where n is the length of the string.

**Space Complexity**: O(n), for the new string created.

**4. Reverse a String**

public class ReverseString {

public static void main(String[] args) {

String str1 = "hello";

String str2 = "Java";

System.out.println(reverseString(str1)); // Output: "olleh"

System.out.println(reverseString(str2)); // Output: "avaJ"

}

public static String reverseString(String str) {

return new StringBuilder(str).reverse().toString();

}

}

**Flowchart**

1. Start
2. Input string
3. Reverse string using StringBuilder
4. Output reversed string
5. End

**Explanation**

The program uses StringBuilder to reverse the string and convert it back to a String.

**Output**

olleh

avaJ

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the length of the string.
* **Space Complexity**: O(n), for the StringBuilder object.

**5. Reverse Array in Place**

import java.util.Arrays;

public class ReverseArray {

public static void main(String[] args) {

int[] arr1 = {1, 2, 3, 4};

int[] arr2 = {7, 8, 9};

System.out.println(Arrays.toString(reverseArray(arr1))); // Output: [4, 3, 2, 1]

System.out.println(Arrays.toString(reverseArray(arr2))); // Output: [9, 8, 7]

}

public static int[] reverseArray(int[] arr) {

int start = 0, end = arr.length - 1;

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

return arr;

}

}

**Flowchart**

1. Start
2. Input array
3. Set start = 0, end = length - 1
4. While start < end:
   * Swap elements at start and end
   * Increment start, decrement end
5. Output reversed array
6. End

**Explanation**

The program swaps elements from both ends of the array towards the center, effectively reversing the array in place.

**Output**

[4, 3, 2, 1]

[9, 8, 7]

**Time and Space Complexity**

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

**6. Reverse Words in a String**

public class ReverseWords {

public static void main(String[] args) {

String str1 = "Hello World";

String str2 = "Java Programming";

System.out.println(reverseWords(str1)); // Output: "World Hello"

System.out.println(reverseWords(str2)); // Output: "Programming Java"

}

public static String reverseWords(String str) {

String[] words = str.split(" ");

StringBuilder reversed = new StringBuilder();

for (int i = words.length - 1; i >= 0; i--) {

reversed.append(words[i]).append(" ");

}

return reversed.toString().trim(); // Remove trailing space

}

}

**Flowchart**

1. Start
2. Input string
3. Split string into words
4. For each word from end to start:
   * Append word to result
5. Trim result and output
6. End

**Explanation**

The program splits the input string into words, reverses the order of the words, and combines them back into a single string.

**Output**

World Hello

Programming Java

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the length of the string.
* **Space Complexity**: O(n), for the array of words and the output string.

**7. Reverse a Number**

public class ReverseNumber {

public static void main(String[] args) {

System.out.println(reverseNumber(12345)); // Output: 54321

System.out.println(reverseNumber(-9876)); // Output: -6789

}

public static int reverseNumber(int num) {

int reversed = 0;

while (num != 0) {

reversed = reversed \* 10 + num % 10;

num /= 10;

}

return reversed;

}

}

**Flowchart**

1. Start
2. Input number
3. Initialize reversed to 0
4. While number != 0:
   * Add last digit to reversed
   * Remove last digit from number
5. Output reversed
6. End

**Explanation**

The program reverses a number by extracting digits from the end and constructing the reversed number.

**Output**

54321

-6789

**Time and Space Complexity**

* **Time Complexity**: O(d), where d is the number of digits in the number.
* **Space Complexity**: O(1), only a few variables are used.

**9. String Palindrome**

public class PalindromeChecker {

public static void main(String[] args) {

System.out.println(isPalindrome("madam")); // Output: true

System.out.println(isPalindrome("hello")); // Output: false

}

public static boolean isPalindrome(String str) {

int left = 0;

int right = str.length() - 1;

while (left < right) {

if (str.charAt(left) != str.charAt(right)) {

return false;

}

left++;

right--;

}

return true;

}

}

**Flowchart**

1. Start
2. Input String
3. Set left = 0, right = length - 1
4. While left < right:
   * If str[left] != str[right], Output false
   * Increment left, decrement right
5. Output true
6. End

**Explanation**

The program checks if a string is a palindrome by comparing characters from both ends. It uses two pointers, one starting at the beginning (left) and the other at the end (right). If all corresponding characters match, the string is a palindrome.

**Output**

true

false

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the length of the string (as we potentially check every character).
* **Space Complexity**: O(1), since we only use a few variables for indexing.

**10. Array Left Rotation**

import java.util.Arrays;

public class ArrayRotation {

public static void main(String[] args) {

int[] arr1 = {1, 2, 3, 4, 5};

int d1 = 2;

System.out.println(Arrays.toString(leftRotate(arr1, d1))); // Output: [3, 4, 5, 1, 2]

int[] arr2 = {10, 20, 30, 40};

int d2 = 1;

System.out.println(Arrays.toString(leftRotate(arr2, d2))); // Output: [20, 30, 40, 10]

}

public static int[] leftRotate(int[] arr, int d) {

int n = arr.length;

d = d % n; // In case d is greater than n

reverse(arr, 0, d - 1);

reverse(arr, d, n - 1);

reverse(arr, 0, n - 1);

return arr;

}

private static void reverse(int[] arr, int start, int end) {

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

}

**Flowchart**

1. Start
2. Input Array and d
3. Set n = length of Array
4. Set d = d % n
5. Reverse elements from 0 to d-1
6. Reverse elements from d to n-1
7. Reverse elements from 0 to n-1
8. Output modified Array
9. End

**Explanation**

The program rotates an array to the left by d positions using a reversal algorithm. It reverses three segments of the array: the first d elements, the remaining elements, and finally the entire array. This effectively moves the elements as required.

**Output**

[3, 4, 5, 1, 2]

[20, 30, 40, 10]

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the number of elements in the array (as each element is visited a constant number of times).
* **Space Complexity**: O(1), since no additional space proportional to input size is used.

**===========================================================================**

**8. Array Manipulation**

public class ArrayManipulation {

public static void main(String[] args) {

int n1 = 5;

int[][] queries1 = {{1, 2, 100}, {2, 5, 100}, {3, 4, 100}};

System.out.println(arrayManipulation(n1, queries1)); // Output: 200

int n2 = 4;

int[][] queries2 = {{1, 3, 50}, {2, 4, 70}};

System.out.println(arrayManipulation(n2, queries2)); // Output: 120

}

public static long arrayManipulation(int n, int[][] queries) {

long[] arr = new long[n + 1]; // Using long to prevent overflow

// Apply the queries using the prefix sum approach

for (int[] query : queries) {

int a = query[0];

int b = query[1];

long k = query[2];

arr[a] += k; // Increment at start index

if (b + 1 <= n) {

arr[b + 1] -= k; // Decrement right after end index

}

}

// Calculate the maximum value after all operations

long max = 0, current = 0;

for (int i = 1; i <= n; i++) {

current += arr[i]; // Create the actual values in the array

if (current > max) {

max = current; // Track maximum value

}

}

return max;

}

}

1. **Start**
2. **Input**: n (size of array), queries (2D array of operations)
3. **Initialize**: Create an array arr of size n + 1 (to handle boundary conditions)
4. **For each query** in queries:
   * Extract a, b, and k
   * Increment arr[a] by k
   * Decrement arr[b + 1] by k (if within bounds)
5. **Initialize**: max = 0, current = 0
6. **For i from 1 to n**:
   * Update current with arr[i]
   * If current > max, update max
7. **Output**: max
8. **End**

**Explanation**

The program uses a technique called the **prefix sum** to efficiently handle multiple range update queries:

* Instead of updating each element in the specified range for each query, we mark the start and just after the end of the range.
* After processing all queries, we calculate the actual values in the array by taking a cumulative sum.
* Finally, we determine the maximum value in the array after all updates.

**Output**

200

120

**Time and Space Complexity**

* **Time Complexity**: O(m + n), where m is the number of queries and n is the size of the array. We process each query once and then traverse the array once to find the maximum.
* **Space Complexity**: O(n), for the additional array used to track increments and decrements.