ALGORITHMS TASK

Task Number 7
K-th Element of Two Sorted Arrays

NAMES

روان ربيع عباس محمد	20230215
اروي محمد ابراهيم فهيم	20230069
جانا اشرف ابوسريع مصطفي	20230143
كريم محمد محمد محمد	20230419
محمد حربي عبدالعاطي	20230462
عبدالرحمن محمد احمد شفيق	20230318

Introduction

Problem Statement

Given two sorted arrays of sizes m and n , find the element that would be at the k-th position in their merged sorted array.

Examples:

Input:

Array 1 = [2, 3, 6, 7, 9]

Array2 = [1, 4, 8, 10]

k = 5

Output: 6 (Merged array: [1, 2, 3, 4, 6, 7, 8, 9, 10])

PROBLEMS SOLUTION

Solution 1

NON-RECURSIVE APPROACH

Solution 2

Merge Sort (RECURSIVE APPROACH)

Solution 3

Bubble Sort (RECURSIVE APPROACH)

Solution 4

Insertion Sort (RECURSIVE APPROACH)

NON-RECURSIVE APPROACH

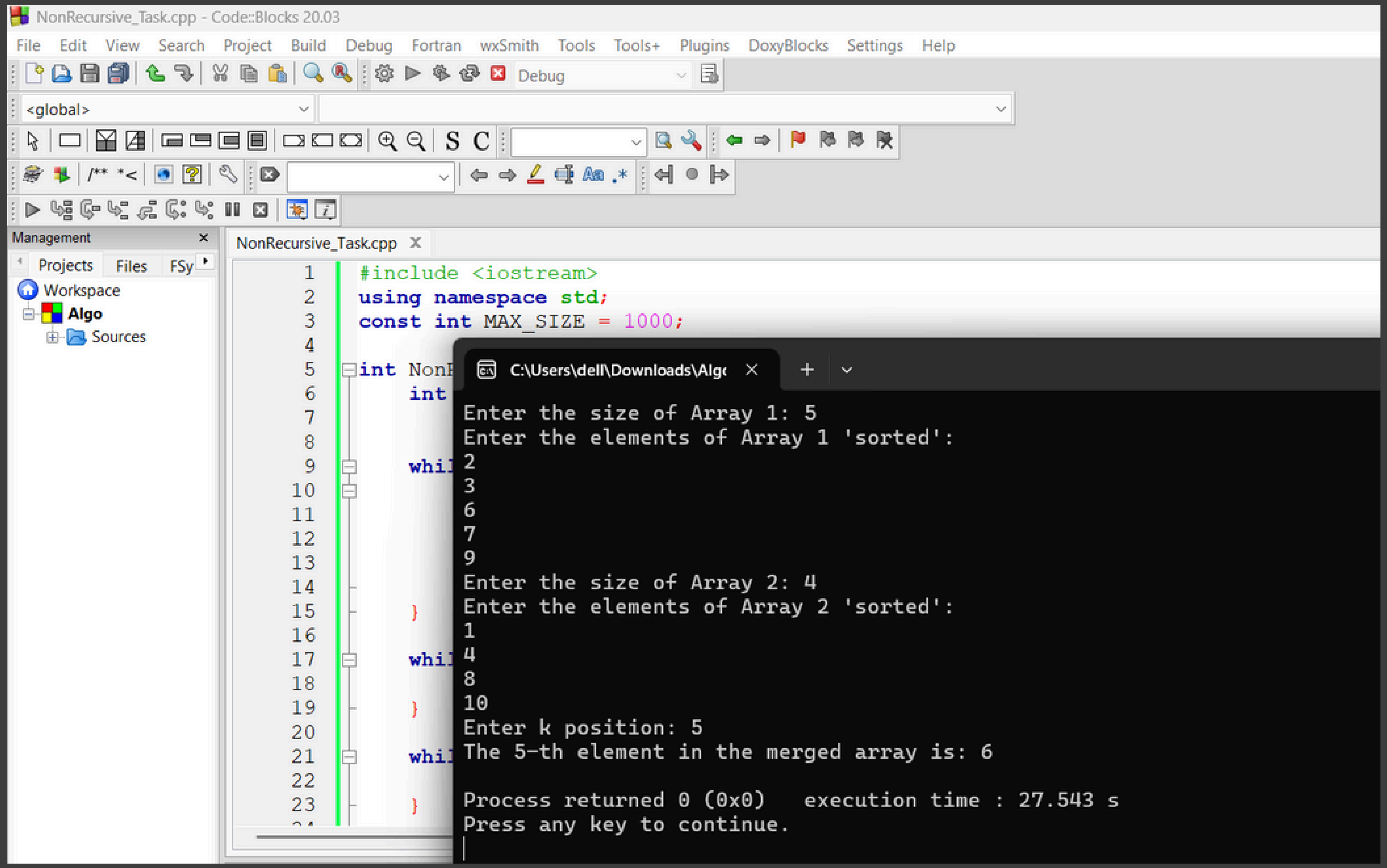
1. Pseudo-code

```
function NonRec_Fun(arr1, size1, arr2, size2, k):
    i ← 0, j ← 0, count ← 0
    while i < size1 and j < size2:
        if arr1[i] < arr2[j]:
            element ← arr1[i]
            i ← i + 1
        else:
            element ← arr2[j]
            j ← j + 1
        count ← count + 1
        if count == k:
            return element

    while i < size1:
        element ← arr1[i]
        i ← i + 1
        count ← count + 1
        if count == k:
            return element

    while j < size2:
        element ← arr2[j]
        j ← j + 1
        count ← count + 1
        if count == k:
            return element

    return -1  // k is out of bounds
```

02 Source code

→ GitHub Link :

03 Sample of output



04 Time Complexity

$O(n)$

05 Space Complexity

$O(1)$

06 Advantages

1- Simple and straightforward implementation
2- Easy to understand and verify correctness
3- No additional memory allocation needed
4- Handles edge cases well (empty arrays, k out of bounds)

07 Disadvantages

1- Not optimal for large arrays when k is large
2- Linear time complexity could be improved with a binary search approach

1. Pseudo-code

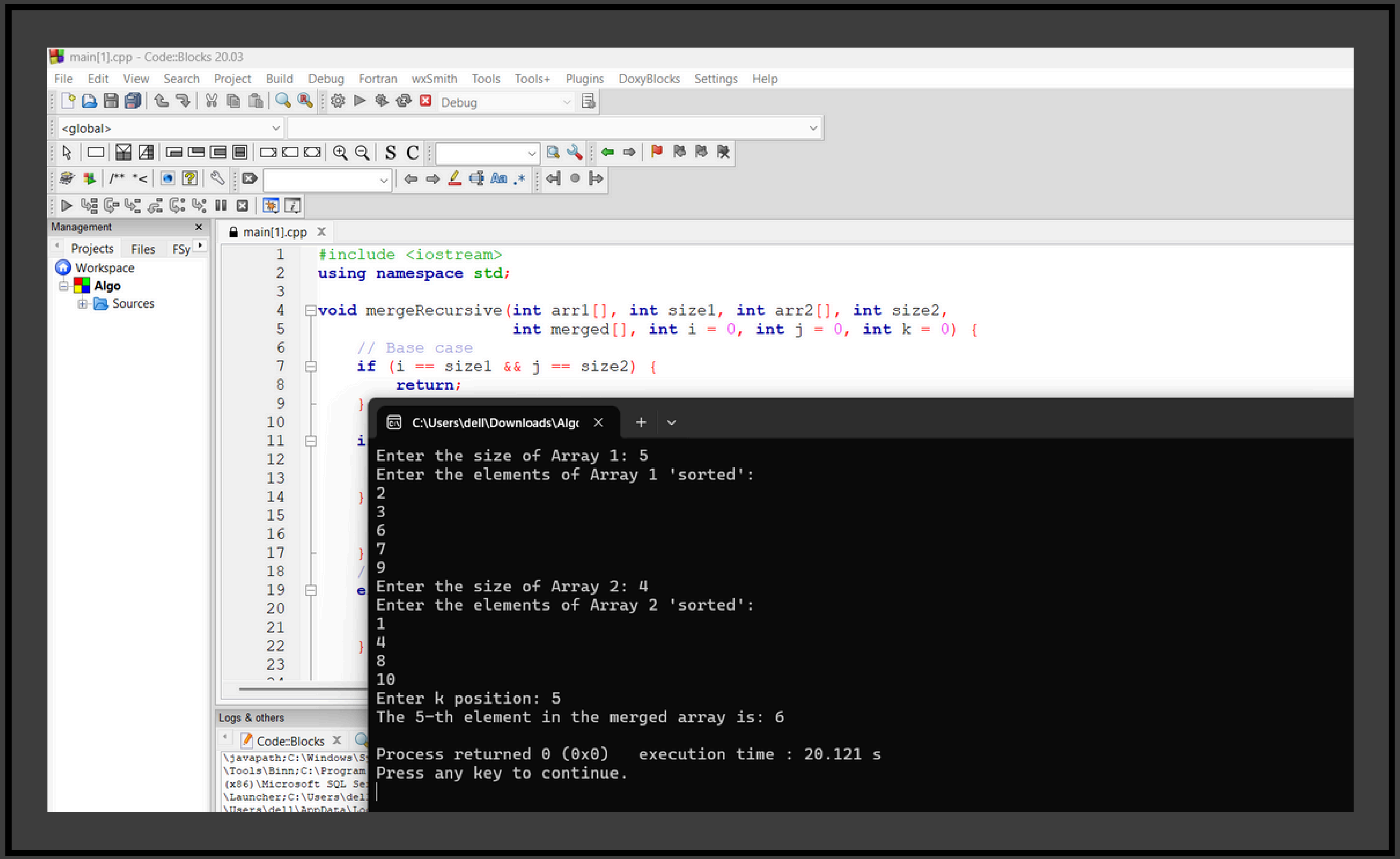
Merge Sort (RECURSIVE APPROACH)

```
function mergeRecursive(arr1, size1, arr2, size2, merged, i = 0, j = 0, k = 0):
    if i == size1 AND j == size2:  // Base case: both arrays fully processed
        return

    if i == size1:  // arr1 exhausted, take from arr2
        merged[k] = arr2[j]
        mergeRecursive(arr1, size1, arr2, size2, merged, i, j + 1, k + 1)

    else if j == size2:  // arr2 exhausted, take from arr1
        merged[k] = arr1[i]
        mergeRecursive(arr1, size1, arr2, size2, merged, i + 1, j, k + 1)

    else if arr1[i] < arr2[j]:  // arr1 has smaller element
        merged[k] = arr1[i]
        mergeRecursive(arr1, size1, arr2, size2, merged, i + 1, j, k + 1)

    else:  // arr2 has smaller or equal element
        merged[k] = arr2[j]
        mergeRecursive(arr1, size1, arr2, size2, merged, i, j + 1, k + 1)
```

02 Source code

→ GitHub Link :

03 Sample of output



04 Time Complexity

Best Case: O(m+n) - When all elements need to be processed
Average Case: O(m+n)
Worst Case: O(m+n)

05 Space Complexity

O(m+n) for the merged array
O(m+n) for recursion stack in worst case
(though tail recursion optimization may reduce this)

06 Advantages

1- Simple recursive implementation
2-Clearly demonstrates the divide-and-conquer approach
3- No need for explicit loop management

07 Disadvantages

1- Recursion overhead for large arrays
2- Potential stack overflow for very large arrays
3- Slightly more difficult to understand than iterative version

Bubble Sort (RECURSIVE APPROACH)

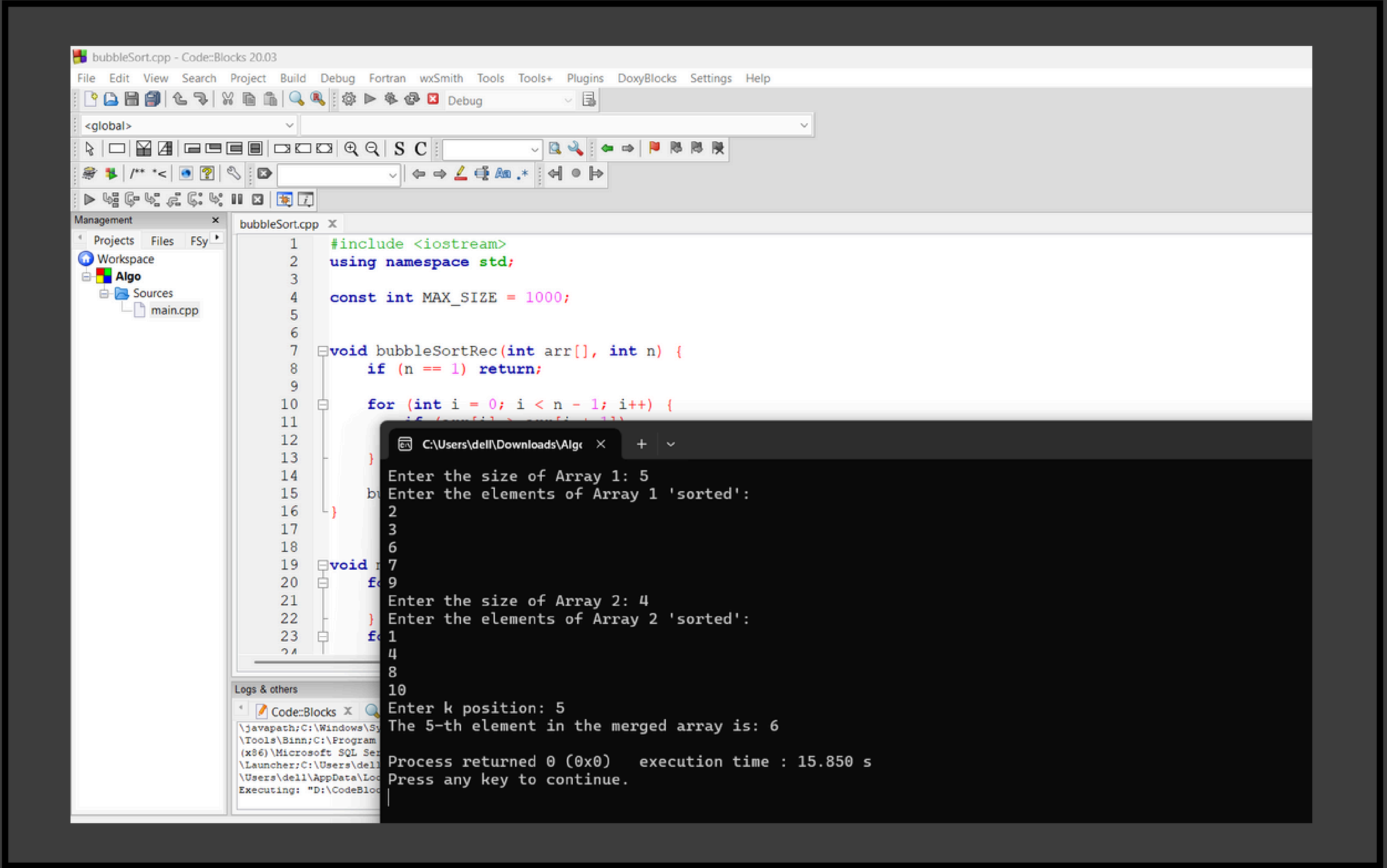
1. Pseudo-code

```
function bubbleSortRec(arr, n):
    if n == 1:
        return

    // Perform one full pass of bubble sort
    for i from 0 to n - 2:
        if arr[i] > arr[i + 1]:
            swap(arr[i], arr[i + 1])

    // Recurse on the remaining array
    bubbleSortRec(arr, n - 1)
end function
```

02 Source code

→ GitHub Link :

03 Sample of output



04 Time Complexity

Worst-case: $O(n^2)$ - When the array is sorted in reverse order

Best-case: $O(n)$ - When the array is already sorted (with optimized version)

Average-case: $O(n^2)$

05 Space Complexity

$O(n)$

06 Advantages

1- Simple to understand and implement
2- Stable sorting algorithm
3- In-place sorting

07 Disadvantages

1- Consistently poor performance (even on sorted input)
2- No practical advantages over other sorts
3- Recursion provides no benefit over iterative version
4- Least efficient of all four approaches

iNSERTION Sort (RECURSIVE APPROACH)

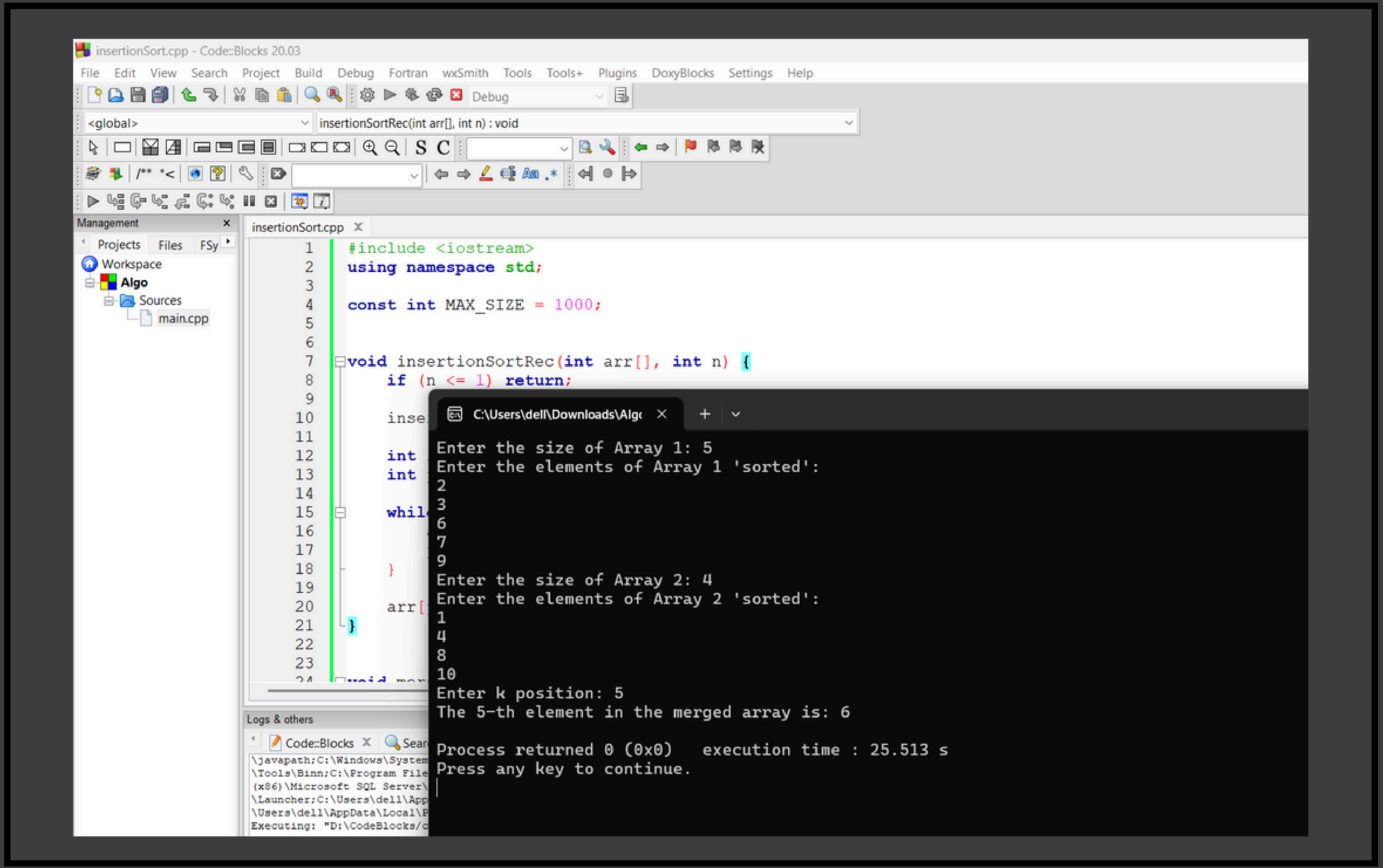
1. Pseudo-code

```
function insertionSortRec(arr, n):
    if n <= 1:
        return

    insertionSortRec(arr, n-1)  // Sort first n-1 elements

    last = arr[n-1]             // Last element to be inserted
    j = n - 2

    while j >= 0 and arr[j] > last:
        arr[j + 1] = arr[j]
        j = j - 1

    arr[j + 1] = last
end function
```

02 Source code

→ GitHub Link :

03 Sample of output



04 Time Complexity

Worst-case: $O(n^2)$ - When the array is sorted in reverse order

Best-case: $O(n)$ - When the array is already sorted

Average-case: $O(n^2)$

05 Space Complexity

$O(n)$

06 Advantages

1- Efficient for small or nearly-sorted arrays
2- Stable algorithm (preserves order of equal elements)
3- In-place sorting (no additional space needed beyond recursion stack)

07 Disadvantages

1- Quadratic time makes it impractical for large arrays
2- Poor performance on reverse-sorted input
3- Recursion adds unnecessary overhead vs iterative version

PERFORMANCE COMPARISON

Algorithm	Time Complexity	Space Complexity	Best For
Non-recursive Merge	O(n)	O(1)	Large sorted arrays
Recursive Merge	O(n)	O(n)	Medium sorted arrays
Recursive Insertion Sort	$O(n^2)$	O(n)	Small arrays
Recursive Bubble Sort	$O(n^2)$	O(n)	Educational purposes

Conclusion

The comparison clearly shows that when merging two already sorted arrays, the merge approaches (especially the non-recursive one) are vastly superior to the sorting approaches. The sorting approaches become useful only when the input arrays aren't guaranteed to be sorted, though even then more efficient sorting algorithms than insertion or bubble sort would typically be preferred. This exercise demonstrates how algorithm choice dramatically affects performance, with the merge approaches being $O(n)$ while the sorting approaches are $O(n^2)$ for this problem.

THANK YOU