



ALGORITHMS TASK





Task Number 7
K-th Element of Two Sorted Arrays

.....

NAMES

20230215	روان ربیع عباس محمد
20230069	اروي محمد ابراهيم فهميم
20230143	جانا اشرف ابوسريع مصطفى
20230419	كریم محمد محمد محمد
20230462	محمد حربي عبدالعاطي
20230318	عبدالرحمن محمد احمد شفيق

.....



x x x x





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]

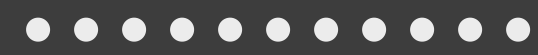
$k = 5$

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





PROBLEMS SOLUTION



Solution 1

**NON-RECURSIVE
APPROACH**

Solution 2

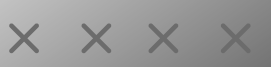
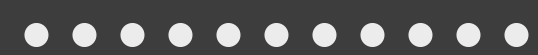
**Merge Sort
(RECURSIVE APPROACH)**

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 :

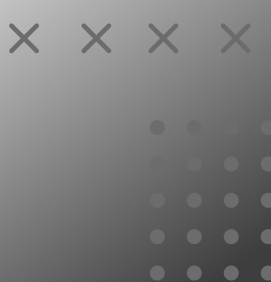
03 Sample of output

The screenshot displays the Code::Blocks IDE with a C++ project named 'NonRecursive_Task.cpp'. The source code is as follows:

```
1 #include <iostream>
2 using namespace std;
3 const int MAX_SIZE = 1000;
4
5 int NonRecursiveTask(int arr1[], int arr2[], int k)
6 {
7     int n1 = arr1[0], n2 = arr2[0];
8     int i = 1, j = 1;
9     while (i <= n1)
10     {
11         arr1[i] = i;
12         i++;
13     }
14     while (j <= n2)
15     {
16         arr2[j] = j;
17         j++;
18     }
19     int kthElement = 0;
20     while (i <= n1 || j <= n2)
21     {
22         if (i <= n1 && (j > n2 || arr1[i] < arr2[j]))
23             kthElement = arr1[i++];
24         else
25             kthElement = arr2[j++];
26     }
27     return kthElement;
28 }
```

The execution output is shown in a terminal window:

```
Enter the size of Array 1: 5
Enter the elements of Array 1 'sorted':
2
3
6
7
9
Enter the size of Array 2: 4
Enter the elements of Array 2 'sorted':
1
4
8
10
Enter k position: 5
The 5-th element in the merged array is: 6
Process returned 0 (0x0)   execution time : 27.543 s
Press any key to continue.
```





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



Functionality:

NonRec_Fun: A non-recursive function to merge two sorted arrays into a single sorted array. It uses a while loop to iterate through both arrays and merge them.

main: Takes two sorted arrays as input, merges them using NonRec_Fun, and returns the k-th smallest element from the merged array.

Key Steps:

- 1- Input: Reads two sorted arrays (arr_1 and arr_2) and an integer k.
- 2- Merge: Combines the two arrays into sortedArr using NonRec_Fun, maintaining the sorted order.
- 3- Output: Prints the k-th smallest element from the merged array



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

The screenshot displays the Code::Blocks IDE interface. The main editor window shows a C++ program named `main[1].cpp`. The code implements a recursive merge sort algorithm. It includes `<iostream>` and uses the `std` namespace. The `mergeRecursive` function takes two arrays, their sizes, and a merged array with its indices. The base case checks if the current subarray has one element. The recursive case splits the array, sorts each half, and merges them back. The `main` function prompts the user for the size and elements of two arrays, calls the merge sort function, and prints the k-th element of the merged array.

```
1 #include <iostream>
2 using namespace std;
3
4 void mergeRecursive(int arr1[], int size1, int arr2[], int size2,
5                     int merged[], int i = 0, int j = 0, int k = 0) {
6     // Base case
7     if (i == size1 && j == size2) {
8         return;
9     }
10
11     i
12
13
14
15
16
17
18
19
20
21
22
23
24 }
```

The output window shows the program's execution. It prompts for the size and elements of two arrays. The first array has size 5 and elements [2, 3, 6, 7, 9]. The second array has size 4 and elements [1, 4, 8, 10]. The user enters the position 5, and the program outputs the 5th element of the merged array, which is 6. The process returns 0 and takes 20.121 seconds to execute.

```
Enter the size of Array 1: 5
Enter the elements of Array 1 'sorted':
2
3
6
7
9
Enter the size of Array 2: 4
Enter the elements of Array 2 'sorted':
1
4
8
10
Enter k position: 5
The 5-th element in the merged array is: 6
Process returned 0 (0x0)   execution time : 20.121 s
Press any key to continue.
```





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



Functionality:

mergeRecursive: A recursive function to merge two sorted arrays into a single sorted array. It compares elements from both arrays and places the smaller one into the merged array.

main: Takes two sorted arrays as input, merges them using **mergeRecursive**, and returns the k-th smallest element from the merged array.

Key Steps:

- 1- Input: Reads two sorted arrays (A and B) and an integer k.
- 2- Merge: Combines the two arrays into **mergedArr** using **mergeRecursive**, maintaining the sorted order.
- 3- Output: Prints the k-th smallest element from the merged array.

1. Pseudo-code

Bubble Sort (RECURSIVE APPROACH)

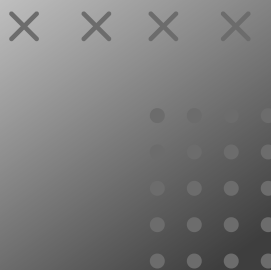
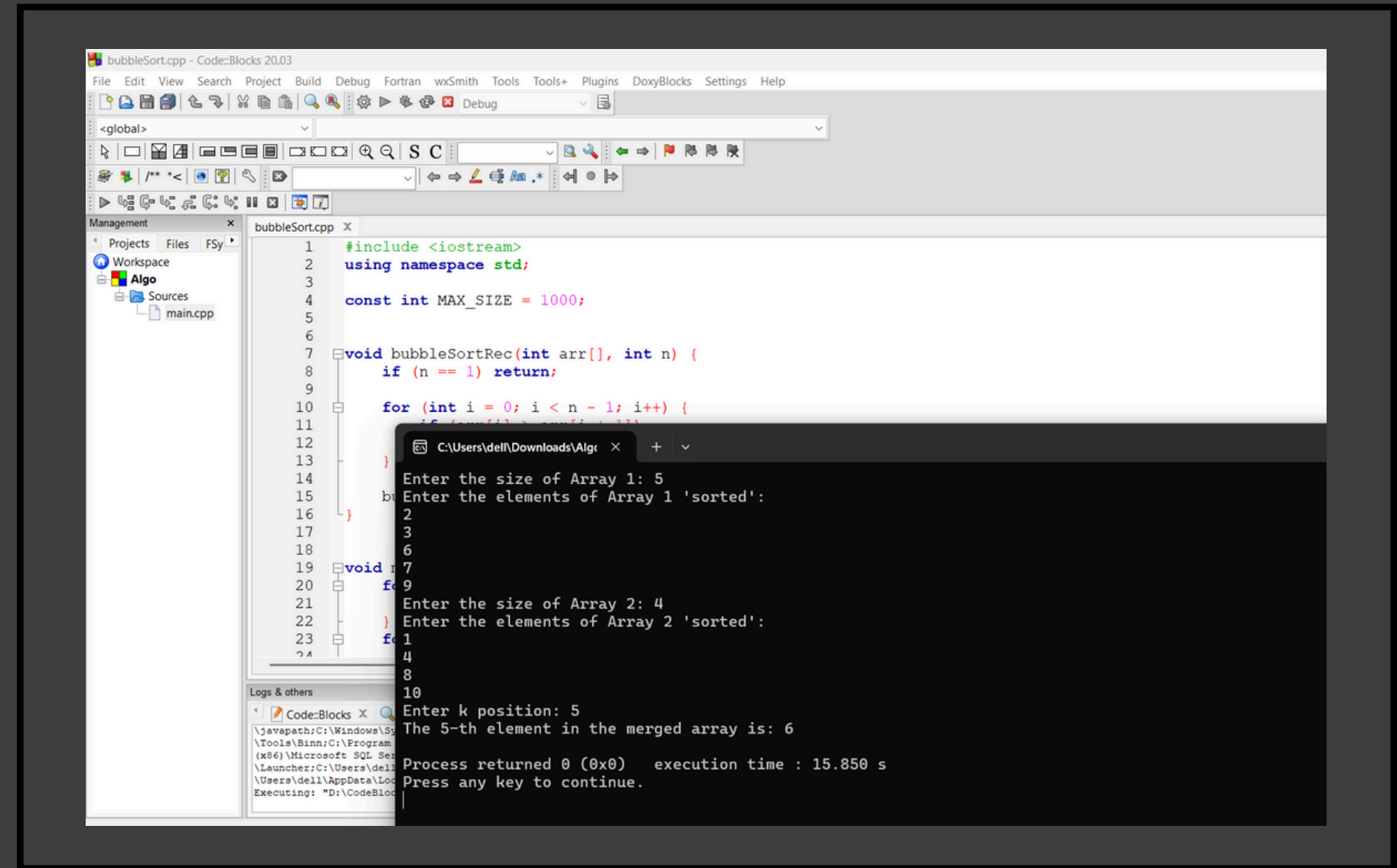
```
1  function bubbleSortRec(arr, n):
2  ✓    if n == 1:
3        return
4
5        // Perform one full pass of bubble sort
6        for i from 0 to n - 2:
7  ✓        if arr[i] > arr[i + 1]:
8            swap(arr[i], arr[i + 1])
9
10       // Recurse on the remaining array
11       bubbleSortRec(arr, n - 1)
12  end function
13
```




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



08 Code Analysis

Functionality:

`bubbleSortRec`: A recursive implementation of the bubble sort algorithm. It sorts an array by repeatedly swapping adjacent elements if they are in the wrong order.

`mergeArrays`: Merges two arrays into a single array by concatenating them.

`main`: Takes two sorted arrays as input, merges them, sorts the merged array using `bubbleSortRec`, and returns the k -th smallest element.

Key Steps:

- 1- Input: Reads two sorted arrays (`arr_1` and `arr_2`) and an integer k .
- 2- Merge: Combines the two arrays into `sortedArr` using `mergeArrays`.
- 3- Sort: Sorts the merged array recursively using `bubbleSortRec`.
- 4- Output: Prints the k -th smallest element from the sorted array.

iNSERTION Sort (RECURSIVE APPROACH)

1. Pseudo-code

```
1  function insertionSortRec(arr, n):  
2  ✓    if n <= 1:  
3        return  
4  
5        insertionSortRec(arr, n-1) // Sort first n-1 elements  
6  
7        last = arr[n-1]           // Last element to be inserted  
8        j = n - 2  
9  
10 ✓    while j >= 0 and arr[j] > last:  
11        arr[j + 1] = arr[j]  
12        j = j - 1  
13  
14        arr[j + 1] = last  
15    end function  
16
```



02 Source code

→ GitHub Link :

03 Sample of output

```
1  #include <iostream>
2  using namespace std;
3
4  const int MAX_SIZE = 1000;
5
6
7  void insertionSortRec(int arr[], int n) {
8      if (n <= 1) return;
9
10     inse
11
12     int
13     int
14
15     while
16
17
18     }
19
20     arr[
21
22
23
24 void main
```

Enter the size of Array 1: 5
Enter the elements of Array 1 'sorted':
2
3
6
7
9
Enter the size of Array 2: 4
Enter the elements of Array 2 'sorted':
1
4
8
10
Enter k position: 5
The 5-th element in the merged array is: 6
Process returned 0 (0x0) execution time : 25.513 s
Press any key to continue.





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



Functionality:

`insertionSortRec`: A recursive implementation of the insertion sort algorithm. It sorts an array by iteratively placing each element in its correct position within the already sorted part of the array.

`mergeArrays`: Merges two arrays into a single array by concatenating them.

`main`: Takes two sorted arrays as input, merges them, sorts the merged array using `insertionSortRec`, and returns the k -th smallest element.

Key Steps:

- 1- Input: Reads two sorted arrays (`arr_1` and `arr_2`) and an integer k .
- 2- Merge: Combines the two arrays into `sortedArr` using `mergeArrays`.
- 3- Sort: Sorts the merged array recursively using `insertionSortRec`.
- 4- Output: Prints the k -th smallest element from the sorted array.

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



x x x x

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

