# ALGORITHMS TASK

## SPRING 2023

### TASK NUMBER 5

### K-TH ELEMENT OF TWO SORTED ARRAYS


NAMES

| | |
|---|---|
| 1. محي الدين ماهر | 20210887 |
| 2. محمود عبد الدايم أبو المجد | 20210867 |
| 3. مروة عمر محمد محمود | 20210900 |
| 4. مريم محمود سيد محمود | 20210926 |
| 5. محمود حسام الدين محمود | 20210859 |
| 6. محمد مصطفى محمد عبد ربه | 20210836 |

# MERGE SORT APPROACH

## A. PSEUDOCODE

```
mergeSort( arr1 , size1 ,arr2 ,size2 , finalArr ,i , j, k) :

 if i<size1 && j<size2
   if arr1[i]<arr2[j]
    finalArr[k] = arr1[i]
    i++

   else
     finalArr[k] = arr2[j]
     j++

   k++
   mergeSort( arr1 , size1 ,arr2 ,size2 , finalArr ,i , j, k)

 else
    while i<size1
      finalArr[k] = arr1[i]
      i++
      k++

   while j<size2
      finalArr[k] = arr2[ij]
      j++
      k++

end
```

## B. SOURCE CODE

- [GITHUB LINK]

## C. CODE ANALYSIS

**The function takes in two arrays arr1, arr2, their sizes s1, s2,and an array finalArray to hold the merged and sorted result. It also takes pointers i, j, and k to keep track of the current index being compared in both arrays and merged array respectively. The function recursively merges the two arrays until one of the arrays has been fully traversed using the following: If arr1[i] is less than arr2[j], arr1[i] is added to the final array,i and k are incremented. If arr1[i] is greater than or equal to arr2[j], arr2[j] is added to finalArray, j and k are incremented. In the main function, two sorted arrays ar1 and ar2, and an integer k are initialized. The sizeof operator is used to find the lengths of the arrays ar1 and ar2 respectively, and the sum of these lengths is stored in the variable tot. An integer array finalar of size s1+s2 is initialized to hold the concatenated arrays and the mergeSort function is called with pointers i, j, and k initialised to 0. The k-th element of the sorted concatenated array, finalar, is printed to the console. Note that k-1 is used since finalar is**

## D. TIME COMPLEXITY:

## O(n)

## E.OUTPUT SCREENSHOT

```
int main()
 {
 int ar1[]={100,112,256,349,770};
 int ar2[]={72,86,119,265,445,892};
 int s1 = sizeof(ar1)/sizeof(ar1[0]);
 int s2 = sizeof(ar2)/sizeof(ar2[0]);
 int finalar[s1+s2];

 int k = 7;
 mergeSort(ar1,s1,ar2,s2,finalar,0,0,0);

 cout<<finalar[k-1]; // because the array is 0 indexe
return 0;
  }
```

```
E:\testinggggg_c\Algorithm_Task\bin\Debug\Algorithm_Task.exe
265
Process returned 0 (0x0)   execution time : 0.139 s
Press any key to continue.
```

# NON-RECURSIVE APPROAC

## A. PSEUDOCODE

```
nonRecursive( arr1 , size1 ,arr2 ,size2 , finalArr) :

int i=0,j=0,k=0
 while i<size1 && j<size2
    if arr1[i]<arr2[j]
     finalArr[k] = arr1[i]
     i++
     k++

    else
      finalArr[k] = arr2[j]
      j++
      k++


  while i<size1
     finalArr[k] = arr1[i]
     i++
     k++

  while j<size2
     finalArr[k] = arr2[ij]
     j++
     k++

end
```

## B. SOURCE CODE

- GITHUB LINK

## C. CODE ANALYSIS

The nonRecusize function takes in two arrays arr1, arr2, their respective lengths s1, s2, and an integer array finalArr to hold the merged and sorted result. It then uses three pointers i, j, and k to keep track of the current index being compared in both arrays and the merged array respectively. The function merges the two arrays in a while loop using the following as a guideline: If arr1[i] is less than arr2[j], arr1[i] is added to finalArr, i is incremented, and k is incremented. If arr1[i] is greater than or equal to arr2[j], arr2[j] is added to finalArr, j is incremented, and k is incremented. Once one of the arrays has been fully traversed, the remaining elements in the other array are copied to finalArr. In the main function, two sorted arrays a and b, and an integer k are initialized. The sizeof operator is used to find the lengths of the arrays a and b respectively, and the sum of these lengths is stored in the variable s1+s2. An integer array f of size s1+s2 is initialized to hold the concatenated arrays and the nonRecusize function is called with pointers i, j, and k initialised to 0. The k-th element of the sorted concatenated array, f, is printed to the console. Note that k-1 is used since f is zero-indexed

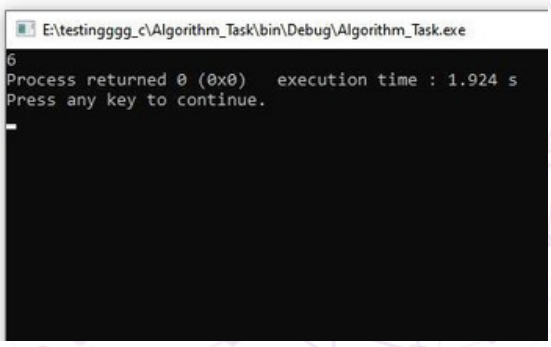
## D. TIME COMPLEXITY:

**O(n)**

## E.OUTPUT SCREENSHOT

```
int main()
 {
fast();
int a[] = {2,3,6,7,9};
int b[] = {1,4,8,10};
int s1 = sizeof(a)/sizeof(a[0]);
int s2 = sizeof(b)/sizeof(b[0]);
int f[s1+s2];
nonRecusize(a,s1,b,s2,f);

int k=5;
cout<<f[k-1];
return 0;
  }
```



# DIVIDE AND CONQUER APPROACH

## A. PSEUDOCODE

```
Algorithm kth_element(arr1,arr2,last1,last2, k)

  if arr1 == last1
    return arr2[k]
  if arr2 == last2
    return arr1[k]

     mid1 ← (last1 - arr1)/2
   mid2 ← (last2 - arr2)/2

  if mid1 + mid2 < k
    if arr1[mid1] > arr2[mid2]
      return kth_element(arr1, arr2+mid2+1, last1, last2, k-mid2-1)
    else
      return kth_element(arr1+mid1+1, arr2, last1, last2, k-mid1-1)
  else
    if arr1[mid1] > arr2[mid2]
      return kth_element(arr1, arr2, arr1 + mid1, last2, k)
    else
        return kth_element(arr1, arr2, last1 ,arr2 + mid2, k)
  end
```



## B. SOURCE CODE

- GITHUB LINK

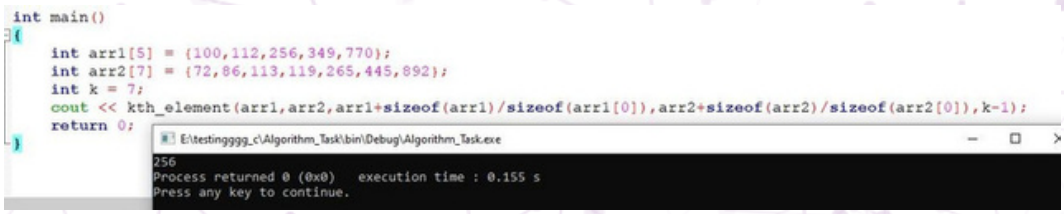
## C. CODE ANALYSIS

The kth_element function takes 5 parameters: two sorted arrays (arr1 and arr2) , two pointers to the last element of each array (last1 and last2), and an int k representing the index of the desired element. If the length of one of the arrays is 0, then the function returns the k-th element of the other array. Two variables, mid1 and mid2, are set to the middle index of each array. If the sum of mid1 and mid2 is less than k, the function recursively calls itself with either arr2+mid2+1 (if arr1[mid1] is greater than arr2[mid2]) or arr1+ mid1+1 (if arr1[mid1] is less than or equal to arr2[mid2]), and an updated value of k. If k is less than the sum of mid1 and mid2, the function recursively calls itself with either arr1+mid1 and last2 (if arr1[mid1] is greater than arr2[mid2]) or last1 and arr2+mid2 (if arr1[mid1] is less than or equal to arr2[mid2]). The main function initializes two sorted arrays (arr1 and arr2) and an integer k. It then calls the kth_element function with the two arrays, their last elements, and k-1 (since arrays are 0-indexed). Finally, the kth_element function returns the kth element of the combined arrays.

## D. TIME COMPLEXITY:

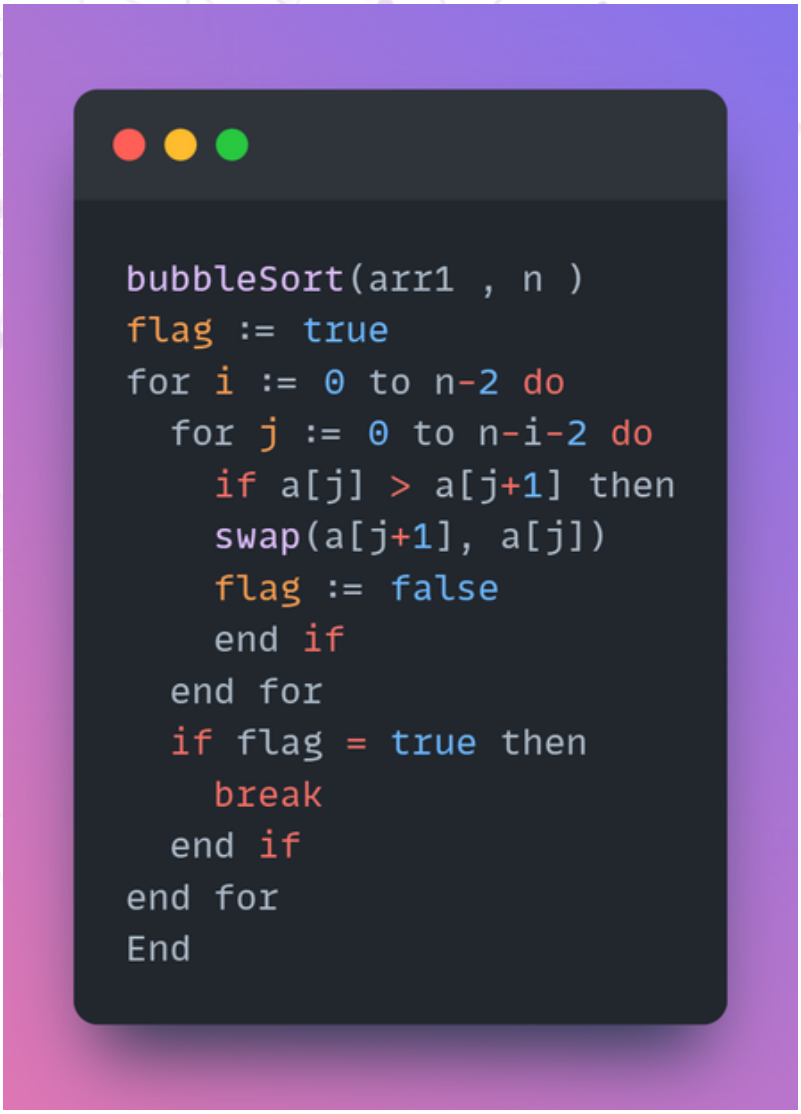
**O(log m + log n)**

## E.OUTPUT SCREENSHOT



# BUBBLE SORT APPROACH

## A. PSEUDOCODE



```
bubbleSort(arr1 , n )
flag := true
for i := 0 to n-2 do
  for j := 0 to n-i-2 do
    if a[j] > a[j+1] then
    swap(a[j+1], a[j])
    flag := false
    end if
  end for
  if flag = true then
    break
  end if
end for
End
```

## B. SOURCE CODE

- GITHUB LINK

## C. CODE ANALYSIS

The bubbleSort function takes in an array a and its length n, and sorts it using bubble sort. It sets flag to true initially, which acts as a checker for whether the array is sorted. The outer loop goes from 0 to i<n-1 and the inner loop, from j=0 to j<n-i-1. If a[j] is greater than a[j+1], the function swaps the two elements and sets flag to false. If flag is still true after the inner loop finishes, it means that the array is already sorted, so the function breaks out of the outer loop. In the main function, two sorted arrays a and b, and an integer k are initialized. The sizeof operator is used to find the lengths of the arrays a and b, and the sum of these lengths is stored in the variable tot. An integer array finalarray of size tot is initialized, and the first s1 elements
of finalarray are filled with the elements of a. The remaining elements
of finalarray are filled with the elements of b, starting from finalarray[s1]. The bubbleSort function is called with finalarray and tot, to ensure that the
concatenated array is sorted. The k-th element of the sorted concatenated array, finalarray, is printed to the console. Note that k-1 is used since finalarray is 0-indexed.

## D. TIME COMPLEXITY:

**$O(n^2)$**

## E.OUTPUT SCREENSHOT

```
int main()
{
  int a[]={2,3,6,7,9};
  int b[]={1,4,8,10};
  int k=5;
  int s1 = sizeof(a)/sizeof(a[0]);
  int s2 = sizeof(b)/sizeof(b[0]);
  ll tot = s1 + s2; //the size of the final array
  int finalarray[tot];
  for(int i=0;i<s1;i++){
    finalarray[i]=a[i];
  }
  for(int i=s1;i<tot;i++){
    finalarray[i]=b[i-s1];
  }
  bubbleSort(finalarray,tot);
  cout<<finalarray[k-1] ; //because the array is 0 indexed
return 0;
  }
```

```
E:\testingggg_c\Algorithm_Task\bin\Debug\Algorithm_Task.exe
6
Process returned 0 (0x0)   execution time : 1.620 s
Press any key to continue.
```

# INSERTION SORT APPROACH

## A. PSEUDOCODE

```
insertionSort(ar , n)
  for i := 1 to n-1 do
    key := ar[i]
    j := i-1
    while j ⩾ 0 and ar[j] > key do
      ar[j+1] := ar[j]
      j := j-1
    end while
    ar[j+1] := key
  end for
End
```

## B. SOURCE CODE

- GITHUB LINK

## C. CODE ANALYSIS

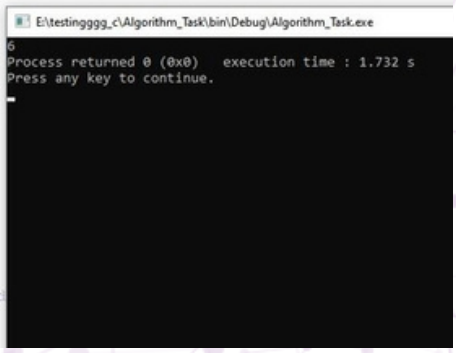
The insertionSort function takes an array ar of length n. It iterates through each element of the array, starting from index 1. For each element, it compares it with the previous elements to find the correct position to insert it into the sorted subarray that precedes it. The while loop moves each element that is greater than key one position to the right until the correct position for key is found. Once the correct position is found, key is inserted into the array. In the main function, two sorted arrays a and b are initialized, along with an integer k to find the k-th element of the sorted concatenated array. The sizeof operator is used to find the lengths of the arrays a and b respectively, and the sum of these lengths is stored in tot. An integer array finalarray of size tot is declared to hold the concatenated arrays. The elements of a are copied into finalarray using a for loop, and the elements of b are copied into finalarray using another for loop that starts from index s1, where s1 is the length of a. The insertionSort function is called with finalarray and tot as arguments to sort the concatenated array. The k-th element of the sorted concatenated array, finalarray, is printed to the console. Note that k-1 is used since finalarray is zero-indexed.

## D. TIME COMPLEXITY:

**$O(n^2)$**

## E.OUTPUT SCREENSHOT

```
int main()
 {
 int a[]={2,3,6,7,9};
 int b[]={1,4,8,10};
 int k=5;
 int s1 = sizeof(a)/sizeof(a[0]);
 int s2 = sizeof(b)/sizeof(b[0]);
 ll tot = s1 + s2; // the tot size of the final array
 int finalarray[tot];
 for(int i=0;i<s1;i++){
   finalarray[i]=a[i];
 }
 for(int i=s1;i<tot;i++){
   finalarray[i]=b[i-s1];
 }
 insertionSort(finalarray,tot);
 cout<<finalarray[k-1] ; // because the array is 0 indexed
return 0;
  }
```



# Comparing between approaches from the complexity

| Merge Sort | Non-Recursive | Divide and Conquer | Bubble Sort | Insertion Sort |
|---|---|---|---|---|
| O(n) | O(n) | O(log m + log n) | $O(N^2)$ | $O(N^2)$ |