Merge Sorted Arrays

DAA ASSIGNMENT-2 , GROUP 7

Medha Balani
IIT2019021

Saksham Aggarwal
IIT2019022

Utkarsh Gangwar
IIT2019023

a) ***Abstract: In this paper we have devised an algorithm which finds sorted components in a given array and then merges those sorted components efficiently to sort the original array.***
We have discussed the time and memory complexity of the algorithm by both Apriori and Apostriori analysis.

I. INTRODUCTION

Let's first formally define what a process of sorting is.
Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
For eg ,if the array Arr [6,4,8,3,1] is given , after sorting in ascending order, array Arr will be [1,3,4,6,8].
Now we define the Merge operation for two sorted arrays
Merge algorithms are a family of algorithms that take multiple sorted lists as input and produce a single list as output, containing all the elements of the inputs lists in sorted order. These algorithms are used as subroutines in various sorting algorithms, most famous of which is merge sort.
For eg , We have 2 sorted arrays $Arr1$ [2,5,7,9] and $Arr2$ [1,3,5,6,10].So , after applying merge operation on these two arrays , the output merged array will be [1,2,3,5,5,6,7,9,10].
Set
Sets are containers that store unique elements following a specific order.

In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.
Sets are typically implemented as binary search trees.
Associative
Elements in associative containers are referenced by their key and not by their absolute position in the container.
Ordered
The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.
Set
The value of an element is also the key used to identify it.
Unique keys
No two elements in the container can have equivalent keys.

II. ALGORITHM DESIGN

A. Algorithm used for finding $Sorted\_Components$

This algorithm takes an array, Arr of n elements as input and returns an array $Sorted\_Components$ of sorted(*in non-decreasing order*) arrays.
We use a temporary array $Temp$ to store the current $Sorted\_Component$
Now to detect $Sorted\_Components$ we traverse the given input array Arr from left to right , for the current element $Arr[i]$ there are two cases :

1) If the current element $Arr[i] \geq Temp.back()$, we append $Arr[i]$ to $Temp$.
2) Else we append $Temp$ to $Sorted\_Components$, clear the temporary array $Temp$ and append $Arr[i]$ to $Temp$.

After the traversal , we have all the $Sorted\_Components$.

B. Algorithm used for merging $Sorted\_Components$

Here we describe 2 possible approaches :

Approach 1

Keep merging 2 $Sorted\_Components$ into 1 using the merge operation of the merge sort algorithm until we end up with 1 single $Sorted\_Component$.

Merge Operation for merging 2 $Sorted\_Components$ using 2 pointer method.

First pointer l points to the current element of first array $Arr1[l]$ and Second pointer r points to the current element of second array $Arr2[r]$.
$While$ l does not point to end of first array or r does not point to the end of second array , we have 2 possible cases :

- If r points to the end of second array or $Arr1[l] \leq Arr2[r]$ we append $Arr1[l]$ to $Answer$ array and increment l by 1.
- Else we append $Arr2[r]$ to $Answer$ array and increment r by 1.

Approach 2

Use better Data Structures
For implementing this approach we can use a Data Structure like a Min-Heap (Priority Queue in $C++STL$) or a Set . We describe our approach using the Set Data Structure.
Firstly Let the size of $Sorted\_Components$ vector be k.
For each element $Sorted\_Components[i][j]$ to be inserted into the set we store 3 things

- the element itself , $Sorted\_Components[i][j]$
- index of its sorted array , i

- index of that element in its sorted array , j

Firstly for each of the k sorted arrays in $Sorted\_Components$ we insert into the set , the first element of each array along with the array and element indices , $(Sorted\_Components[i][j], i, j)$
Now ,Until the set doesn't become empty we extract an element from the beginning of the set and do the following two steps :

1) Append $Sorted\_Components[i][j]$ to $Answer$ array.
2) If $j+1 < Sorted\_Components[i].size()$, ie the current element index is less than its sorted component size , we insert the next element $(Sorted\_Components[i][j+1], i, j+1)$ into the set.

The $Answer$ array contains the final sorted array.

Algorithm 1: Find Sorted Components

Input: vector Arr of size n
Output: 2-D vector Sorted_Components

```
Function FindSortedComponents(Arr,n):
    vector Temp
    Temp.append(Arr[0])
    for i ← 1 to n do
        if (Arr[i] ≥ Temp.back()) then
            Temp.append(Arr[i])
        else
            Sorted_Components.append(Temp)
            Temp.clear()
            Temp.append(Arr[i])
    if (Temp.empty() is false) then
        Sorted_Components.append(Temp)
    return Sorted_Components
```

Algorithm 2: Merge Sorted Components

Input: 2 arrays $Arr1$ and $Arr2$
Output: sorted array Arr

```
Function MergeSortedComponents(Arr1,Arr2):
    n1 ← Arr1.size()
    n2 ← Arr2.size()
    l ← 0
    r ← 0
    vector Arr
    while (l ≠ n1 or r ≠ n2) do
        if (r = n2 or Arr1[l] ≤ Arr2[r]) then
            Arr.append(Arr1[l])
            l ← (l + 1)
        else
            Arr.append(Arr2[r])
            r ← (r + 1)
    return Arr
```

Algorithm 3: Merge Sorted Components

Input: 2-D vector $Sorted\_Components$ of size k
Output: sorted array $Answer$

```
Function MergeSortedComponents(Sorted_Components,k):
    Set My_Set
    Vector Answer
    for i ← 0 to k do
        My_Set.insert(Sorted_Components[i][0], i, 0)
    while Mt_Set.empty() is false do
        Cur ← My_Set.begin().first
        i ← My_Set.begin().second
        j ← My_Set.begin().third
        Delete(My_Set.begin())
        Answer.append(Cur)
        if j + 1 < Sorted_Components[i].size() then
            My_Set.insert(Sorted_Components[i][j + 1], i, j + 1)
    return Answer
```

III. ALGORITHM ANALYSIS

A. Time Complexity

The above described algorithm to find sorted components from a given array takes O(n) complexity in both the best and worst cases.
For merging $Sorted\_Components$ we have described 2 algorithms. First algorithm which merges 2 sorted arrays into 1 using 2pointer technique takes O(n) for each merge operation therefore for k $Sorted\_Components$ a total of $k-1$ merge operations would be required. So, this algorithm takes O($n*k$) complexity where n is the size of input array.
In Second Algorithm using Set data structure , since every element is pushed into the set only once and it is extracted only once , also the size of set never grows beyond k , so the total time complexity comes out to be O($n*logk$) where n is the size of input array.

B. Space Complexity

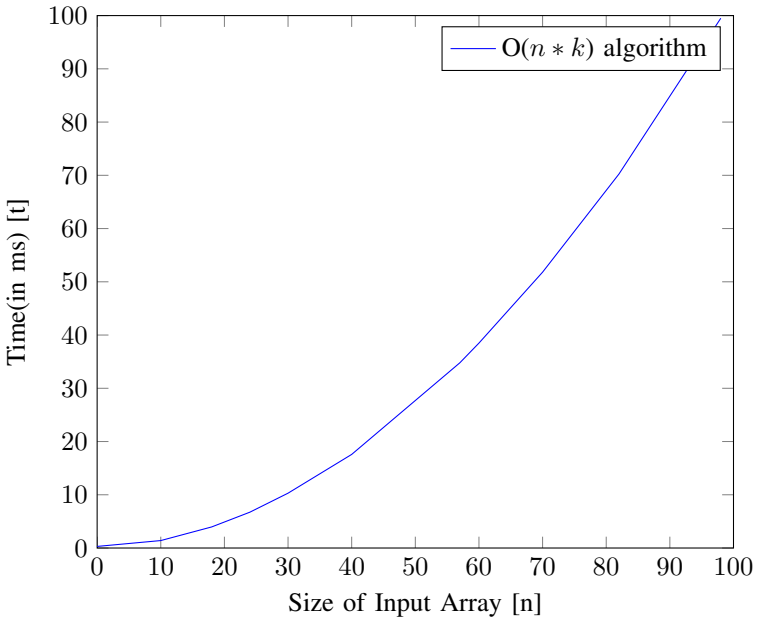
The space complexity of this algorithm is O(n) where n is the size of input array.

IV. EXPERIMENTAL STUDY

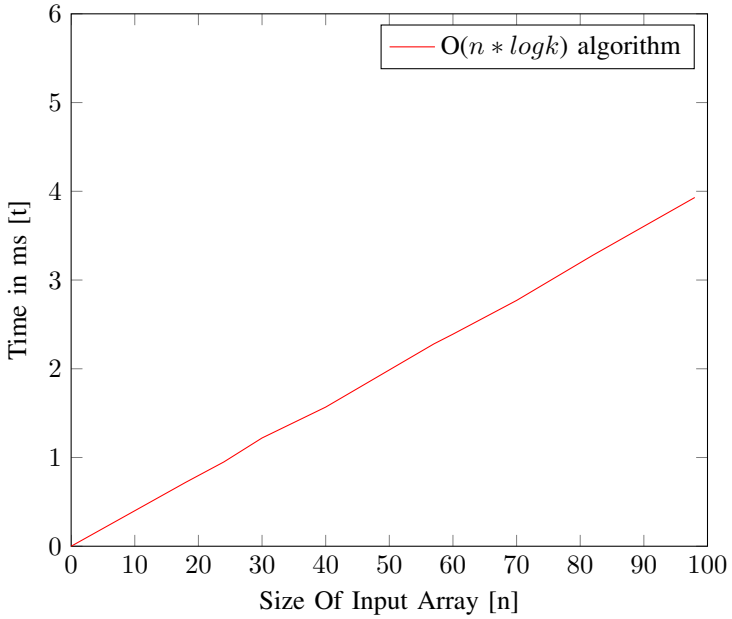
In the worst case when the input array is sorted in decreasing order($k \simeq n$) the naive O($n*k$) algorithm would be equivalent to O(n^2) which is clearly inefficient for input of the order of 10^5, this algorithm would take around 100 seconds for execution.

While our O($n*logk$)algorithm , when the input array is sorted in decreasing order ,($k \simeq n$),the complexity would be O($n*logn$) and it is pretty efficient even for inputs of the order 10^5,this algorithm would run in around 16 milliseconds which is far-far better than the first algorithm.

Complexity analysis For Naive O($n * k$) algorithm



Complexity analysis for O($n * logk$) algorithm



V. ILLUSTRATION

Let us consider our initial array to be 2,1,4,3. Now applying our optimal proposed approach:

1) First, traversing the set and appending sorted components (non-decreasing) to our sorted components vector. So , our sorted components vector would be {{2},{1,4},{3}}.
2) Now, we declare a set and insert first element of each sorted components along with index of that sorted component and index of that first element in the sorted component i.e 0. Here set would be {{1,{1,0}}, {2,{0,0}}, {3,{2,0}}}.
3) Next, we declare a ans vector and then iterate our set until it is empty. While iterating the set, we do following two things:
   - We append the smallest element to our ans vector i.e ans.append(*st.begin().first) and erase from our set.
   - Check if that particular sorted component contain more elements. If yes, we append to our set.

   So, here inserting '1'(*st.begin().first) to our ans. We see the size of (*st.begin().second.first) sorted component which is two here which is greater than (*st.begin.second.second)+1. So, we insert it and our set becomes {{2,{0,0}},{3,{2,0}},{4,{1,1}}}.
   Now inserting '2' to our ans({1,2}). The 0th index sorted component has no further elements so inserting '3' to ans ({1,2,3}), which again has no further elements so inserting '4' to ans ({1,2,3,4}) thereby emptying the set.

VI. CONCLUSION

Finding sorted components takes same time and space in both the proposed algorithms. But for merging those sorted components, the later approach while using set data structure comes optimal with time complexity O($n * logk$), where n is size of input set/array and k is number of sorted components. So, in worst case too, time complexity would be O($n * logn$).

VII. REFERENCES

1) https://www.geeksforgeeks.org/merge-k-sorted-arrays-set-2-different-sized-arrays/
2) https://www.geeksforgeeks.org/set-in-cpp-stl/
3) https://www.giss.nasa.gov/tools/latex/ltx-2.html
4) https://www.cplusplus.com/reference/set/set/