Implement Merge sort without using recursion

DAA ASSIGNMENT-3 , GROUP 6

Shreyansh Patidar
IIT2019018

Biswajeet Das
IIT2019019

Hritik Sharma
IIT2019020

Abstract: In this paper we have devised an algorithm which is an implementation of Merge sort without using recursion.. This report further contains -
II. Algorithm Design
III. Algorithm Analysis
IV. Result
V. Conclusion

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. [1]
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.[2] 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].

II. ALGORITHM DESIGN

A. Algorithm used for dividing array into smaller parts

This algorithm takes an array, Arr of n elements as input and keeps on dividing the array into smaller components in iterative way. And as we breakdown we also keep merging the broken arrays in the order of their sizes in power of 2 in sorted fashion. Thereby the final merge will result in overall the sorted array.

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* $Arr1[l] \le Arr2[r]$ 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
Naive Approach
Here first we calculate the maximum element of both the arrays to be sorted $maxAandmaxB$. Then we find maxALL = max $maxA, maxB$.
Next, we traverse in a for loop from 1 to maxALL and check if current num is present in A or B. If present then insert it in the merged array as many times as found.

Algorithm 1: Divide into smaller parts

```
Input: vector Arr of size n
Output: last sorted array
Function divide(Arr,n):
    int $currSize$, $leftStart$
    for $currSize \leftarrow 1$ to $n-1$ do
        for $leftStart \leftarrow 0$ to $n-1$; += 2*currSize do
            int $mid$ = min(leftStart + currSize - 1, n-1)
            int $rightEnd$ = min(leftStart + 2*currSize - 1, n-1)
            merge(arr, leftStart, mid, rightEnd)
    return arr
```

Algorithm 2: Merge Sorted Components

```
Input: 2 arrays $Arr1$ and $Arr2$
Output: sorted array $Arr$
Function MergeSortedComponents($Arr1$,$Arr2$):
    $n1 \leftarrow Arr1$.size()
    $n2 \leftarrow Arr2$.size()
    $l \leftarrow 0$
    $r \leftarrow 0$
    vector $Arr$
    while ($l \neq n1$  or  $r \neq n2$) do
        if ($r = n2$  or  $Arr1[l] \le Arr2[r]$) then
            $Arr.append(Arr1[l])$
            $l \leftarrow (l+1)$
        else
            $Arr.append(Arr2[r])$
            $r \leftarrow (r+1)$
    return Arr
```

Algorithm 3: Merge Sorted Components

```
Input: 2 arrays $Arr1$ and $Arr2$
Output: sorted array $Arr$
1  Function MergeSortedComponents():
2      int $maxA$,$maxB$,$maxALL$
3      int $Arr[nA + nB]$
4      for $i$ in $A$ do
5          $maxA$ = max($maxA$ , A[i])
6      for $i$ in $B$ do
7          $maxB$ = max($maxB$ , B[i])
8      $maxALL$ = max($maxA$, $maxB$)
9      for $i \leftarrow 1$ to $maxALL$ do
10         if(i in A or i in B) Arr.append(i);
11     return Arr
```

III. ALGORITHM ANALYSIS

A. Time Complexity

The above described algorithm to divide the array into smaller components takes O($log(n)$) 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($log(n) * k$) complexity where n is the size of input array.
In Second Algorithm using naive approach , since it is brute force and each num is accessed , the total time complexity comes out to be O($log(n) * maxALL * (nA + nB)$) where nA, nB are the sizes of array A and array B. So, we can conclude that,
Approach 1:
t_{avg} = $O(n * logn)$
t_{worst} = $O(n * logn)$
t_{best} = $O(0)$
Approach 2:
t_{avg} = $O(log(n) * maxALL * (nA + nB))$
t_{worst} = $O((log(n) * maxALL * (k)))$
t_{best} = $O(0)$

B. Space Complexity

The space complexity of this algorithm for both the approaches are as follows:
Approach 1:
sp_{avg} = $O(n * logk)$
Approach 2:
sp_{avg} = $O(((nA + nB) * log(k)))$

IV. EXPERIMENTAL STUDY

Here we will see the graphs of each algorithm used.

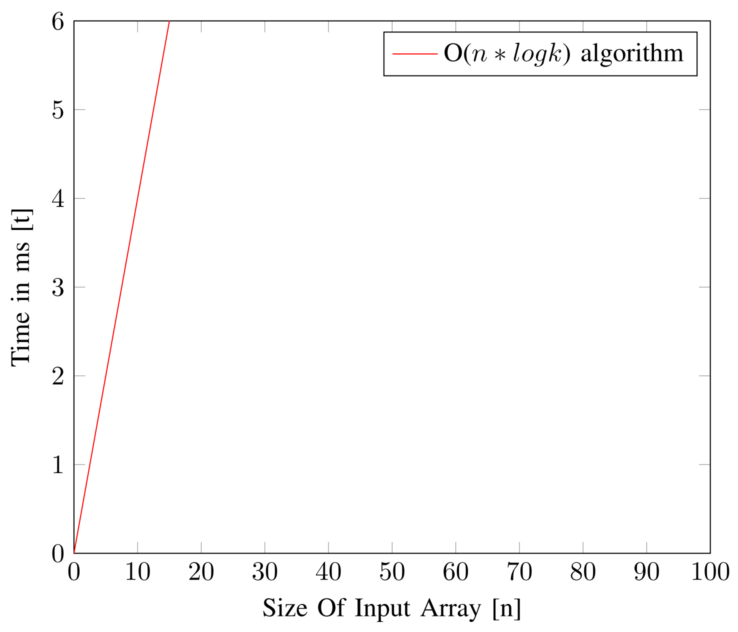

Fig 1: Time complexity for naive algorithm. Approach 2

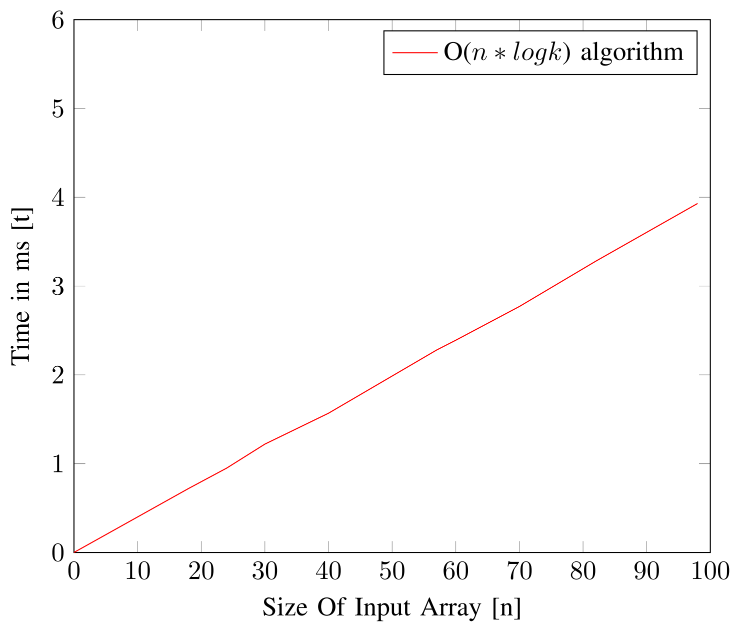

Fig 2: Time complexity for best algorithm. Approach 1

V. 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($n * logn$).

VI. REFERENCES

1) https://en.wikipedia.org/wiki/Sorting
2) https://www.sciencedirect.com/topics/computer-science/merge-operation