Assignment 6 - Sort a given array by finding the max and min element using divide and conquer approach.

Parag Parihar
Roll No:- IIT2016095
iit2016095@iiita.ac.in

Rakshit Sai
Roll No:- IIT2016126
iit2016126@iiita.ac.in

Adarsh Agrawal
Roll No:- IIT2016516
iit2016516@iiita.ac.in

Nilotpal Pramanik
Roll No:- IRM2016501
irm2016501@iiita.ac.in

***Abstract*—Sort a given array by continuously tracking the max and min elements at every recursive step and placing the max and min elements at the correct position. find the max and min element using divide and conquer approach.**

I. Introduction and Literature Survey

The given problem is on divide and conquer. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. Divide: Break the given problem into subproblems of same type. 2. Conquer: Recursively solve these subproblems 3. Combine: Appropriately combine the answers

The given problem for us is to sort a given array by continuously tracking the maximum and minimum elements at every recursive step and placing the maximum and minimum elements at the correct position. We need to find the maximum and minimum elements by using the divide and conquer approach. It basically means that we need to divide the array into two parts and then keep on solving the same way in subsequent steps and finally combine the array to arrive with the sorted array.

The report contains :-
2. Algorithm Design
3. Analysis and Discussion
4. Experimental Analysis
5. Conclusion

II. Algorithm Design

To implement this algorithm, it is our convention that we are sorting the array in **ascending order** through **divide and conquer approach**.So, now we have to scan an array of integers through taking the length of the array which are the inputs for this problem.After that we have to check weather the array is in ascending order or not.Thats why we will traverse the array up to half of its length and check weather the return value of the function **check** is 1 or not.For this function we are sending the constructed array according the current minimum and maximum index in each iteration of the loop.

"check" Function: Basically in this function we are checking weather the array is in ascending order or not.As the attributes we are taking the **array** itself along with the **start**: the starting index and **end**: the last index of the array.Then we are initializing a **flag** variable to 1,indicating the array is in ascending order.After that we are traversing the whole array from the starting index and check weather the current element is lesser than the previous one or not.If true then assign 0 at flag and break the loop.And from the function we are returning the value of flag.

Now if the return value of **check** is 1 then the input array is already sorted in ascending order and we have just to print that array accordingly.

Otherwise we have to store the return value of the **getMinMax** function in mx,the **struct pair**.

"struct pair" Structure: This structure only contains the **min** and **max** to integers.

"getMinMax" Function: As the attributes it takes the **array**, **low**: the starting index and **high**: the last index of the array.As well as the return type of this function is **struct pair**.In the function we are defining the **minmax**,**mml**(: struct pair for the left portion of the array),**mmr**(: struct pair for the right portion of the array) variables as the struct pair and **mid** variable as the integer data type.If the array consists only one element the the value of **low** and **high** will be equal and satisfying that condition the value of **low** will be assigned to **minmax.max** and **minmax.min** both.And return the **minmax**. If the array consists only two elements then we have only to check the value of **high** is just next to the value of **low** or not.If true then simply we have to determine the value at the **low**th index of the array is greater than the value at the **high**th index of the array and then we will assign the value of **low** to **minmax.max** and the value of **high** to **minmax.min**.Otherwise we have to assign the value of **high** to **minmax.max** and the value of **low** to **minmax.min** accordingly. When the array consists more than two elements then we are assigning the value of (low + high)/2 in **mid** and through recursive way storing the value getMinMax(arr, low, mid) and getMinMax(arr, mid+1, high) in **mml** and **mmr** respectively.Basically here we are dividing

the whole array with respect to the middle index of the array **mid** in **left** and **right**.Now we are comparing the maximum values and minimum values of the two partitions and finally determining the highest and lowest value among them and assigning their values in **minmax.max** and **minmax.min** respectively.And returning the **minmax** from this function which is containing the minimum and maximum indexes of the particular constructed array.

Now print the **minimum**: the element stored in **mx.min**th index of the array,**maximum**: the element stored in **mx.max**th index of the array followed by the range through printing the values of **0+i** and **n-1-i**.Now we have just to assign the minimum element of that particular constructed array in the lowest index and accordingly the maximum element in the highest index through swapping the values in that array.We have to follow this procedure in each iteration of the for loop traversing for the half of the length of the input array.And finally we are printing the sorted array in ascending order, according to our convention.

Algorithm 1 getMinMax function

INPUT: arr, start, end
OUTPUT: position of minimum and maximum element
Struct pair *minmax, mml, mmr*
if $start = end$ **then**
  $minmax.max \leftarrow start$
  $minmax.min \leftarrow end$
  return *minmax*
end if
if $end = start + 1$ **then**
  if $arr[start] > arr[end]$ **then**
    $minmax.max \leftarrow start$
    $minmax.min \leftarrow end$
    return *minmax*
  else
    $minmax.max \leftarrow end$
    $minmax.min \leftarrow start$
    return *minmax*
  end if
end if
$mid \leftarrow (start + end)/2$
$mml \leftarrow getMinMax(arr, start, mid)$
$mmr \leftarrow getMinMax(arr, mid + 1, end)$
if $arr[mml.min] < arr[mmr.min]$ **then**
  $minmax.min \leftarrow mml.min$
else
  $minmax.min \leftarrow mmr.min$
end if
if $arr[mml.max] < arr[mmr.max]$ **then**
  $minmax.max \leftarrow mmr.max$
else
  $minmax.max \leftarrow mml.max$
end if

Algorithm 2 Check function

INPUT: arr, start, end
OUTPUT: 1(If array is in ascending Order) and 0 Otherwise
$flag \leftarrow 1$
for $i = start + 1$ *to* end **do**
  if arr[i] < arr[i-1] **then**
    $flag \leftarrow 0$
    break;
  end if
end for
return flag

Algorithm 3 Main Function

Scan the value of n and array
for $i = 0$ *to* $n/2$ **do**
  if $check(arr, 0 + i, n - 1 - i)! = 1$ **then**
    Struct pair $mx \leftarrow getMinMax(arr, 0+i, n-1-i)$
    if $n - 1 - i = mx.min$ **then**
      $mx.min \leftarrow mx.max$
    end if
    $max\_ele \leftarrow arr[mx.max]$
    $arr[mx.max] \leftarrow arr[n - 1 - i]$
    $arr[n - 1 - i] \leftarrow max\_ele$
    $min\_ele \leftarrow arr[mx.min]$
    $arr[mx.min] \leftarrow arr[0 + i]$
    $arr[0 + i] \leftarrow min\_ele$
  else
    break
  end if
end for
print array

III. Analysis and Discussion

In this section we have analysed all the cases including time complexity and graphs and notations.

A. *Time-Complexity——-*

*1) **Best-Case**:* For the best case, we are assuming that our input array will be in ascending order in that case, we only have to use "check" function once as check function will return 1 which will fail the if condition and we will come out of that loop. Altogether we are using mainly check function.
Check function :- $5n + 7$
But overall for best case is:- $5n + 13$
Time complexity for best case is $O(n)$

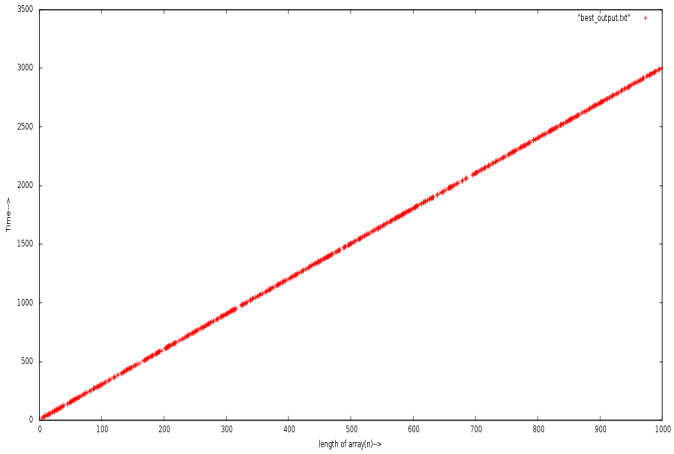

Fig.1 Length of array(n) vs Time

*2) **Worst-Case**:* As far as worst case is concerned, we are assuming that we will place minimum element at the very beginning of array and maximum element at last. In this way, our output array should be in ascending order. By above assumption we can say that our worst case occur if the input array is in descending order. As because for worst case check will never return 1 and we will have to go into divide and conquer part.
For the worst case, we require the CHECK function, The getMinMax function and the main function is mandatory.
The number of units of the time taken for the CHECK function is :- $5n + 9$
The number of units of the time taken for the getMinMax function is :- $(3n/2) - 2$ if n is power of 2 Overall time complexity of worst case is:- $\frac{n}{2}\left(\frac{3n}{2} - 2\right)$ or $O(n^2)$

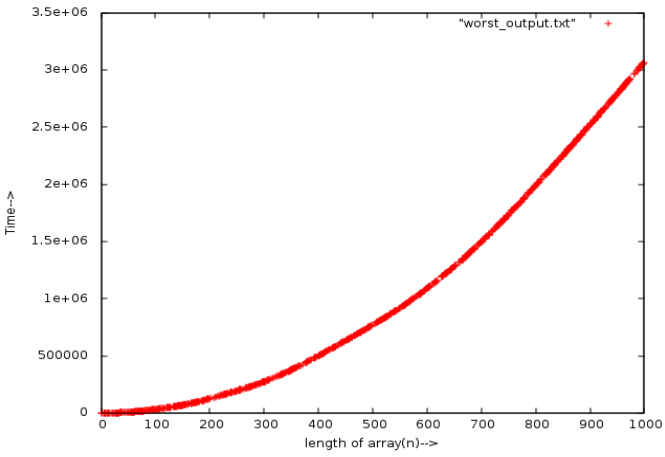

Fig.2 Length of array(n) vs Time

*3) **Average-Case**:* For average case we can say that it will lie between best and worst case but more accurately it is order of n^2 or we can say that overall time complexity for average case is $O(n^2)$.

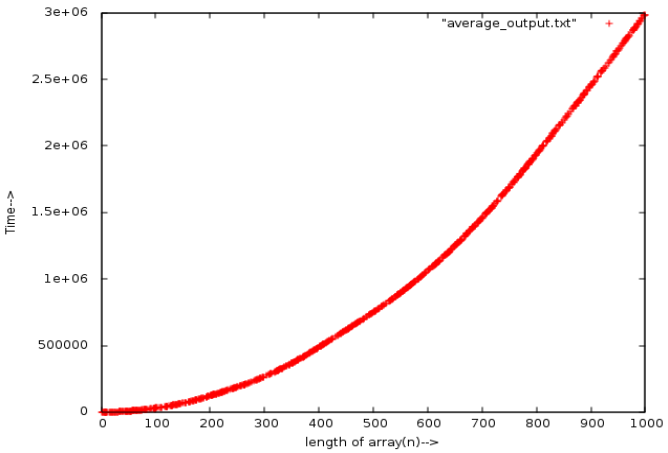

Fig.3 Length of array(n) vs Time

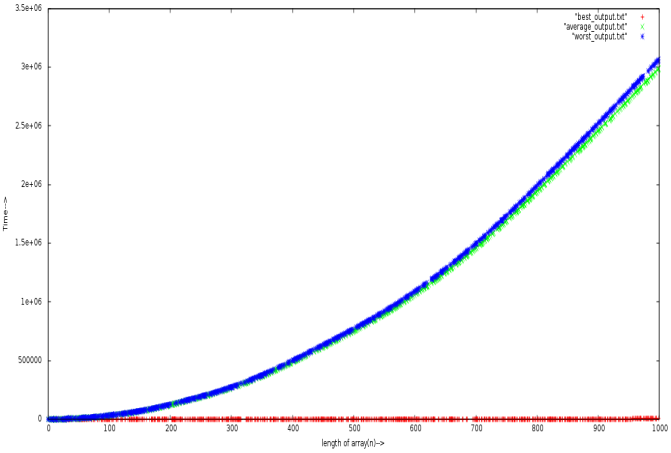

Fig.4 Length of array(n) vs Time

RED CURVE:- **best case**
GREEN CURVE:- **average case**
BLUE CURVE:- **worst case**

IV. Experimental Analysis

We revised the commands and basic functions of **GNUPlot** to plot the time complexity analysis graphs and other relevant analysis related to our algorithm.
While making this report we also learnt the basics of making reports using LATEX in IEEE format using **IEEEtran class**.

For the analysis of the Complexity of the algorithm we had to generate 3 kinds of test cases. For each of these 3 kinds our code was run on **1000** test cases of each type and The algorithm was run on these three files separately and graphs were plotted using **GNU-Plot**.The graphs have been vividly explained in the previous section. In each of these test cases the length of **array** was randomly chosen greater then 1.

For the **Best case**, we generated the test cases where array is in ascending order as we have assumed that we will put minimum element at first position and maximum element at the last position

For the **Worst case**, we generated test cases where array is in descending order so that for every iteration of loop we will not get any part of array being ascending already.So for every iteration we will have to go into divide and conquer part.

And for the **average case**, we generated test cases where array was filled with random number. So it may be possible that any point of loop we may get array in ascending order or we may never get any array in ascending order

n	**time**$_{bestcase}$	**time**$_{averagecase}$	**time**$_{worstcase}$
5	20	143	148
10	35	370	459
20	65	1477	1523
50	155	8371	8547
75	230	17707	18063
100	305	31745	32551

V. CONCLUSION

In the problem we had to sort a given array by continuously tracking the Maximum and Minimum elements at every recursive step and placing the Maximum and Minimum elements at the correct position. We had to find the Maximum and Minimum elements by the Divide and Conquer approach. For this we used **Check function** which returns 1 if the array is in ascending order else 0. This is used to check if the array is sorted properly or not. Then the **getMinMax function** is the one where we find the Minimum and Maximum positions of the array by the Divide and Conquer approach. The best case time complexity of the problem is $O(n)$. The worst case time complexity is $O(n^2)$.And accordingly for the average case it is lying in between the best and worst case but through out the experimental analysis it is basically quite nearer to the curve for worst case.Overall the solution is optimized by the Divide and Conquer approach and it is used to calculate the Minimum and Maximum.As well as according to our convention we are sorting the input array in ascending order.

REFERENCES

[1] Introduction To Algorithms (Second Edition) by Thomas H.Cormen, Charles E.Leiserson , R.L.Rivest and Clifford Stein.
[2] The C Programming Language by Brian Kernighan and Dennis Ritchie.
[3] https://www.geeksforgeeks.org/
[4] https://www.stackoverflow.com/