Equally partitioning the sequence into increasing , decreasing and increasing pattern and combine them to form a resulting sequence.

Akshay Gupta IIT2017505, Naman Deept IIT2017507 Snigdha Dobhal IIT2017506

March 13, 2019

1 Abstract

This paper introduces algorithm(s) to arrange a given sequence (array) of sequences in such a way that one third of it is increasing remaining one third is decreasing and remaining again increasing.To solve the problem, this paper uses the *Merge Sort* algorithm which is based on the *Divide and Conquer* approach. To solve the problem, the given array has been broken down in three parts from the middle and then first one third has been sorted in ascending order while the other one third has been sorted in descending order and the remaining one third again in ascending order. This paper also analyzed various aspects of the presented algorithm, such as time complexity, space complexity, etc. In this paper, outcomes of the algorithm is been presented with the experimental results.

Keywords: *Sorting*, *Merge Sort*, *Sequences*, Divide and Conquer.

2 Introduction

Merge Sorting: Merge Sort is a divide and conquer algorithm. It works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.The simple idea in the Merge Sort is to divide the array elements into 2 halves till a stage where total number in the array equals 1 (divide) and to continue merging those elements in a new array (conquer) .The idea of merging goes like this : Suppose we have 2 pack of cards both sorted and placed in a desk with its face facing upwards in a stack and we wish to merge both the cards into a new pack of deck facing downwards facing downwards. We pick the smallest of the upward facing card from among the 2 cards which are facing upwards and then put it into the empty deck of the card to be faced downward, thus if we continue doing this till both the cards ends, then we get the desired merged sorted deck of another cards. But how to check whether the card facing upward in the deck is empty or not, the idea for this will be to place a sentinel card, and we mark it (∞), So that when this card is shown up then we can say that our deck of cards which was facing upward is empty.

Recurrence Relation on merge sort: The merge sort algorithm is based on the divide and conquer algorithm, thus the time taken by the array to sort it's n elements is the twice the time taken by half the length of array to sort the elements whose size is exactly halved and also the time it takes to merge the 2 arrays. So for the array of size n .The time taken by this array to sort will be given as :

$$T(n) = 0 \text{ (when } n = 1)$$
$$T(n) = 1 \text{ (when } n = 2)$$
$$T(n) = 2\times\mathrm{T}(\tfrac{n}{2})+\mathrm{n} \ (when \ \mathrm{n}\geq 2)$$

3 Proposed Method

Input: Given any general array of size n, obtain the sequence in the specific format in such a way that one third part of the array is increasing, one third

is decreasing and the remaining one third is again increasing.

3.1 Approach for this problem

3.1.1 Merge-Sorting Algorithm:

As we have already seen the idea of merge sort is based on the divide and conquer algorithm and it has also been demonstrated taking the example of cards, we will now establish the algorithm of how to merge the two given sorted elements of an array considering the same example as that of cards where we need to consider the sentinel cards as well.

3.1.2 Merging Algorithm:

```
procedure MERGE(A, p, q, r)
    n1 ← q − p + 1
    n2 ← r − q
    Create Array L[1...n1 + 1] and R[1...n2 + 1]
    for i ← 1 to n1 do
        L[i] ← A[p + i − 1]
    end for
    for i ← 1 to n2 do
        R[i] ← A[q + i]
    end for
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        end if
        if L[i] > R[j] then
            A[k] ← R[j]
            j ← j + 1
        end if
    end for
end procedure
```

.

3.1.3 Merge Sort:

Here we sort the array by dividing it into 2 parts and going on till a stage is reached where the size of every array becomes 1.The algorithm for this is as follows:

```
1: procedure MERGESORT(a)
2:     if n == 1 then
3:         return a
4:     end if
5:     var left as array = a[0]...a[n/2]
6:     var right as array = a[n/2 + 1]...a[n]
7:     left = mergesort(left)
8:     right = mergesort(right)
9:     return merge(l1, l2)
10: end procedure
```

3.1.4 Getting the desired sequence:

To obtain the desired sequence we just need to divide our initial array elements into 3 equal portions (maybe not equal when the size is not divisible by 3) ,and just sort these divided arrays using the merge sort technique ,thereafter we only need to check if $(\frac{n}{3})$th element is lesser than $(\frac{n}{3}+1)$th element,then we only need to swap those elements and if $(\frac{2n}{3})$th element is greater than $(\frac{2n}{3}+1)$th element, then we also need to swap those elements.

Algorithm for the main function: Here A is our initial array provided, and b,c,d are the partitioned arrays.

```
procedure MAIN(A)
    for i ← 0 to n/3 do
        b[i] ← A[i]
        c[i] /gets A[i + n/3]
        d[i] /gets A[i + 2n/3]
    end for
    MERGESORT(b)
    REVERSE(MERGESORT(c))
    MERGESORT(d)
    if b[n/3] ≤ c[0] then
        temp ← b[n/3]
        b[n/3] ← c[0]
        c[0] ← temp
    end if
    if c[n/3] ≥ d[0] then
        temp ← c[n/3]
        c[n/3] ← d[0]
        d[0] ← temp
    end if
```

4 Experimental Results

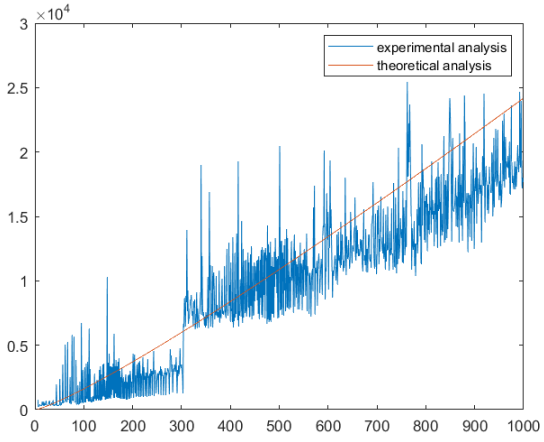

Figure 1: Graph showing the comparisons of the experimental performance of the algorithm with the theoretical analysis .

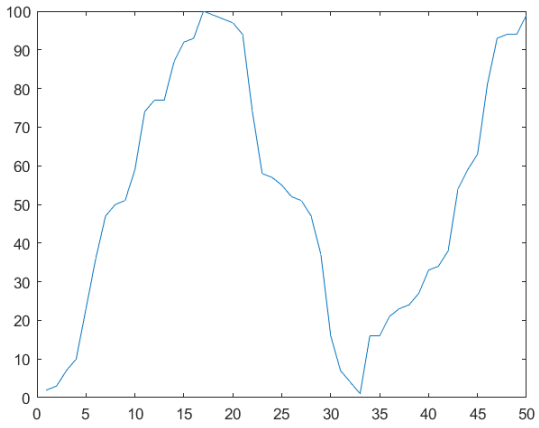

Figure 2: Graph showing arrangement of a random sequence by the above algorithm .

4.1 Complexity Analysis and Explanation

The above algorithm uses merge sort to sort array sequences and hence the worst case , the best case come out to be same as proportional to $\mathbf{n}log(n)$. We can conclude to it theoretically as we know that from equation of time complexity that
$\mathrm{T(n)} \leftarrow 2T(n/2) + n$
$T(n) \leftarrow 2^k \mathrm{T(n/2^k)} + \sum_{i=1}^{k} 2^i$
when $2^k \leftarrow \log n then$
$T(n) \propto nlog(n)$
So, we conclude that the given approach solve the problem in Θ(nlog(n)).
Since , the worst case and the base case scenario are different factors of nlog(n) and hence graph of experimental analysis shows varying proportionality of nlog(n) because of having numbers in the sequence which is randomly generated and thus can be concluded that the graph shows exact picture of the theoretical analysis.

5 Discussion and Future Work

One more way could be to solve the problem is by first sorting the whole sequence in increasing order and then placing the highest two elements to $n/3^{th}$ and

n^{th} index . And then placing the lowest two elements at 1^{st} and $2n/3^{th}$ index . Now picking 3 elements from the sorted sequence after the first two elements and placing them in order of first at the neighborhood of 1^{st} index and then the left and right of $2n/3^{th}$ index will also generate the desired the sequence . This way only one time sorting is required and the no reversal of the sequence is done .

6 Conclusion

In this paper we introduced the Merge Sort algorithm to sort an array of integers in such an order that the value of the integers increase from the beginning to one third of the array, and then decrease from one-third to two-thirds of the array, and then again increase from two-third to the end of the array. In order to achieve this, we partitioned the array into 3 halves and then just sorted their contents using merge sort technique and swapped their elements wherever neccessary . Merge Sort algorithm is a Divide-and-Conquer approach. That implies that we first divide the array into two sub arrays, and perform the same operation on the sub arrays until a base-case is reached. In this case, when the size of sub array reaches one, it is already sorted. Then merging starts happening.
We have shown that the merge sort has time complexity of O(n×log(n)). And it perfectly matches with the experimental results presented. We have seen that there are no best and worst cases in this algorithm. Although the algorithm sorts a large array in very less time compared to other Divide-and-Conquer algorithms yet it could not remove drawback of the Merge Sort algorithm. It still requires a stack space of O(n).

7 References

1..Merge Sort:In GeeksforGeeks **https://www.geeksforgeeks.org/merge-sort/**

2.Merge Sort:In Tutorialspoint **https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm**

3.Introduction to Algorithms(Second Edition)
Author:Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein
Chapter 3:Growth of Functions, Page 41