# A02G48Q25 : Sorting a sequence in 'N' format

Harsh Kochar (IIT2018049)

Madhu(IIT2018068)

*IV Semester B.Tech, Department of Studies in Information Technology,*

*Indian Institute of Information Technology Allahabad, India*

***Abstract: In this paper we have first sorted the sequence by tim sort algorithm and then rearranged it in a manner such that, when plotted it's index against the values it makes 'N' shaped pulse.***

## I. INTRODUCTION

Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort. Because merge function performs well when sizes subarrays are powers of 2 and insertion sort performs well for small arrays.
We divide the Array into blocks known as Run. We sort those runs using insertion sort one by one and then merge those runs using a combined function used in merge sort. If the size of Array is less than run, then Array get sorted just by using Insertion Sort. Now we merge them one by one. We double the size of merged subarrays after every iteration. After this sorting we rearrange it by reversing the mid two third region, ie swapping the elements.
The algorithm works is described in detail in part II.

## II. ALGORITHM DESCRIPTION

This algorithm works in mainly 4 stages.
If n is the size of sequence.
Stage 1: We have a suitable run size (32 this time). Then we divide our sequence into n/32 parts.

Stage 2: Run an insertion sort algorithm for every run (parts). It operates by assuming first i elements are sorted. The inner loop moves element A[i] to its correct place so that after this i+1th element is sorted.

Stage 3: Merging the sorted sequence in pairs by making a new sequence of twice the size. .This can also exploit the parallel processing hence divide and conquer to give complexity of $\Theta(log^3(n))$ .

Stage 4: Rearrangement by swapping each (n/3+i)th term with (2n/3-i)th term.

## III. ALGORITHM AND ANALYSIS

**Main**
Input: array of integer
Output: sorted array

```
i← 0                                          Step 1
run← 32                                       Step 2
for  i<n, i← i+run                            Step 3
        insertion(arr[], i,min(i+run,n))      Step 4
size← run                                     Step 5
while size<n                                  Step 6
  left← 0                                     Step 7
  while left<n                                Step 8
    mid← left + size -1                       Step 9
    right ← min((left + 2*size - 1), (n-1))  Step
10
    merge(arr, left, mid, right)              Step
11
    left← size*2
Step 12
```

```
  size← size*2                          Step
13
rearrange(arr,n/3,2*n/3)                Step
14
return arr
```

**Insertion sort**

input : array of a fixed range (less than equal to run) **A**

```
i ← 1                                   Step
15
while i < length(A)                     Step
16
        x ← A[i]
Step 17
        j ← i - 1                       Step
18
        while j >= 0 and A[j] > x       Step
19
          A[j+1] ← A[j]                 Step
20
          j ← j - 1                     Step
21
        end while                       Step
22
        A[j+1] ← x[3]                   Step
23
        i ← i + 1
Step 24
end while                               Step
25
```

**Merge sort**

intput : array **A**, left bound **l**, mid bound **m**, end bound **r**

```
  index1← l , index2← m+1               Step
26
  newlist←  empty list                  Step
27
  while index1<m and index2<r           Step
28
    If A[index1] <= A[index2]           Step
29
        newlist.append(A[index1])       Step
30
        index1← index1+1                Step
31
    else
        newlist.append(A[index2])       Step
32
        index2← index2+1                Step
33
i← 0                                    Step
34
while l<m:                              Step
35
    A[l]← newlist[i]                    Step
36
    l← l+1, i← i+1                      Step
37
```

**Rearrange**

Input an array **A,** left bound **l,**right bound **r**

```
for l<(l+r)/2                           Step
38
        swap(A[l],A[r])                 Step
39
```

Step 1,2,5: have a frequency of 1, and will take 1 unit of time.

Step 3: The loop has a frequency of O(n), as it runs for n/Run times.

Step 4: It is insertion sort of 32 pisces, O(1). (Should have been O(n2), but n in here is Run, which is 32, so constant)

Step 6: Loop will run for log(n) times (with the base 2, as size←size*2).

Step 7,13: Will just have the loop frequency, and will take 1 unit of time.

Step 8: This loop will run n/2*size times, where size changes from 32, 64,128… to nearest value of 2^([log(n)]), which means log(n) - log(32) number of times. Giving a frequency of O(log(n)).

Step 9,10,12: Will take 1,2,1 unit of time.

Step 11: It will run for Size*2 times giving a complexity of O(n). (And Space complexity =Size*2) Which is best as it is merging of two sorted array of same size. But with a best complexity of O(1)

>>This merging could have exploited parallel processing, giving a complexity of log(n). This would make overall complexity (log(n))^3.

This results in overall complexity to be:

T $\alpha$ n + log(n)*n
T best $\alpha$ n + log(n)
Which gives average time complexity of O(nlog(n)).
And Space complexity of O(n), as n is the size of extra space (maximum) was used.
It happened while merging.

## III (a). Apriori Analysis

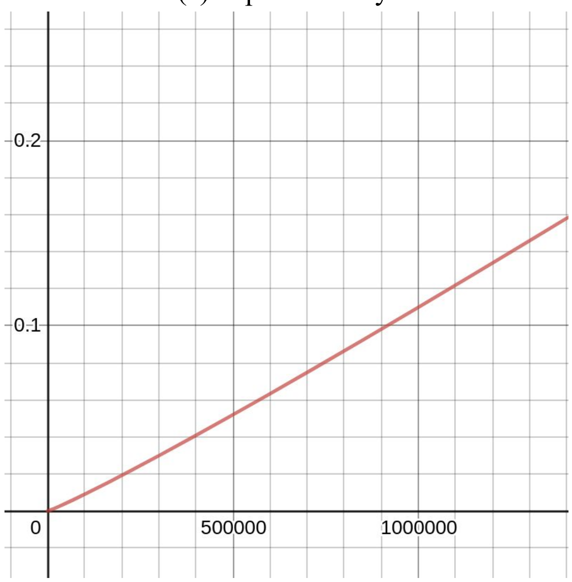
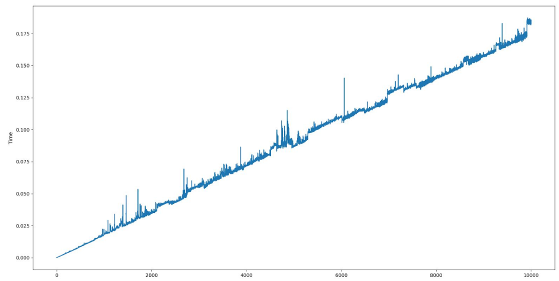



Figure1:Time complexity graph for apriori analysis.

Table1: time complexity for Apriori analysis

| | Time | FreqBest | Frequency |
|---|---|---|---|
| Step 1 | 1 | 1 | 1 |
| Step 2 | 1 | 1 | 1 |
| Step 3 | 2 | n/run | n/run |
| Step 4 | x | n/run | n/run |
| Step 5 | 1 | 1 | 1 |
| Step 6 | 1 | $log_2(run)$ | $log_2(run)$ |
| Step7 | 1 | 1 | 1 |
| Step 8 | 1 | 1 | 1 |
| Step 9 | 1 | 2 | 2 |
| Step 10 | 3 | 2 | 2 |
| Step 11 | y | 2 | 2 |
| Step12 | 1 | 2 | 2 |
| Step13 | 1 | 1 | 1 |
| Step14 | z | 1 | 1 |
| Step15 | 1 | 1 | 1 |
| Step16 | 1 | run | run |
| Step 17 | 1 | 1 | 1 |
| Step18 | 1 | 1 | 1 |
| Step19 | 2 | 1 | n |
| Step20 | 1 | 1 | 1 |
| Step21 | 1 | 1 | 1 |
| Step22 | 0 | 0 | 0 |
| Step23 | 1 | 1 | 1 |
| Step24 | 1 | 1 | 1 |
| Step25 | 0 | 0 | 0 |
| Step26 | 2 | 1 | 1 |
| Step27 | 1 | 1 | 1 |
| Step28 | 2 | run | run |
| Step29 | 1 | 1 | 1 |
| Step30 | 1 | 0 | 1/2 |
| Step31 | 1 | 0 | 1/2 |
| Step32 | 1 | 0 | 1/2 |
| Step33 | 1 | 0 | 1/2 |
| Step34 | 1 | 1 | 1 |
| Step35 | 1 | run | run |
| Step36 | 1 | 1 | 1 |
| Step37 | 2 | 1 | 1 |
| Step38 | 3 | n/3 | n/3 |
| Step39 | 3 | 1 | 1 |

## IV. EXPERIMENTAL ANALYSIS AND PROFILING.




In the above graph, the input 'n' is considered along x-axis and time along y-axis. The graph represents Time v/s n. The above graph shows that the APosteriori analysis is consistent with Apriori analysis.

## CONCLUSION

From this paper we can conclude that algorithmic best is O(n), worst and average is O(nlog(n)) and space complexity is O(n)

## REFERENCES

[1]https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort
[2]https://en.wikipedia.org/wiki/Insertion_sort#Best,_worst,_and_average_cases
[3]https://www.i-programmer.info/news/181-algorithms/12111-timsort-really-is-onlogn.html

[4]https://en.wikipedia.org/wiki/Timsort#Analysis

[5]https://www.geeksforgeeks.org/timsort/

[6]https://en.wikipedia.org/wiki/Merge_sort

[7]https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_merge_sort.htm

[8]https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort