# ESO207A – PROGRAMMING ASSIGNMENT 5


ASSIGNMENT DONE BY

Hamza Masood
210403

Amay Raj
210116


## THE OBJECTIVE OF THE ASSIGNMENT

The followings are the objectives of this assignment.

1. To investigate whether Quick sort has better/poor performance than Merge sort in reality?

2. To investigate how often Quick sort deviates from its average-case behaviour in reality?

## BACKGROUND OF THE ASSIGNMENT

We know that the average number of comparisons during Quick Sort on n distinct numbers is nearly 40% more than Merge sort. Quick Sort can take $\Theta(n^2)$ comparisons in the worst case whereas Merge Sort always perform O(n log n) comparisons. Given these facts and the fact that Quick Sort was invented after Merge Sort, why does C library have an implementation of Quick Sort but not that of Merge sort?
We explored the answer to this question through this assignment.

# TASK 1: QUICK SORT VS MERGE SORT

Note: 1) Number of iterations per value of n: 500
2) Units of time is microseconds (μs)

## 1.1 COMPARISONS

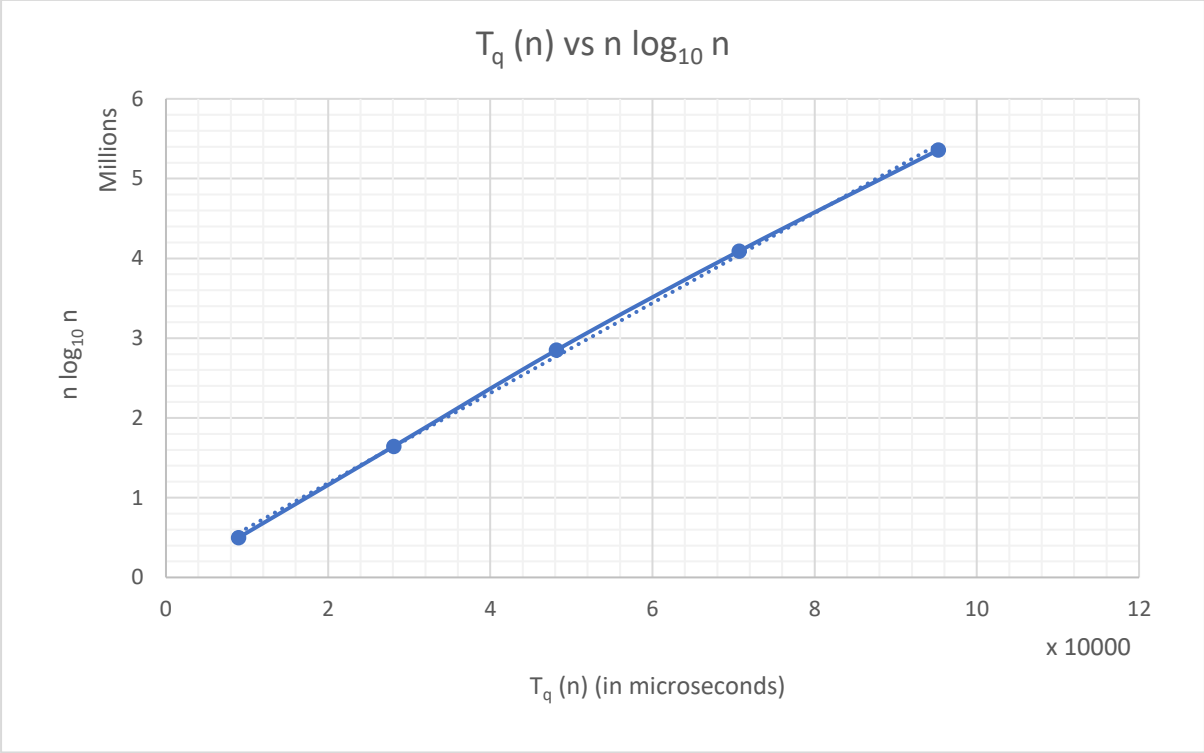
| n –→ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|
| Average number of comparisons during Quick Sort | 649 | 10997 | 156412 | 2040362 | 35116153 |
| $2n \log_e n$ | 921 | 13816 | 184207 | 2302585 | 27631021 |
| Average number of comparisons during Merge Sort | 542 | 8707 | 120453 | 1536370 | 18674168 |
| $n \log_2 n$ | 664 | 9966 | 132877 | 1660964 | 19931569 |

**Inference:**
Average number of comparisons during Quick Sort are greater than that of Merge Sort (around 1.3 – 1.4 times that of Merge Sort).
The average number of comparisons in Quick Sort are nearly equal to $2n \log_e n$ and that of Merge Sort are nearly equal to $n \log_2 n$.
In Quick Sort, during partition every element from start to end is compared with the pivot whereas during Merge Sort, we compare two elements at a time and move on from one of them, so we are comparing two halves. Therefore, the number of comparisons in Quick Sort are more.

## 1.2 NUMBER OF COMPARISONS AND TIME COMPLEXITY OF QUICK SORT

| n –→ | $10^5$ | $3*10^5$ | $5*10^5$ | $7*10^5$ | $9*10^5$ |
|---|---|---|---|---|---|
| Average running time of Quick Sort | 8966 | 28082 | 48138 | 70698 | 95254 |

**Plot of $T_q(n)$ versus n $\log_{10}$ n for the values in the table given above, where $T_q(n)$ is the average running time of quick sort on input of size n**:




**Inference:**

The average running time of Quick Sort is almost proportional to n log n as discussed in the lecture since the graph comes out to be a straight line.
Hence, the average time complexity of Quick Sort is n log n.

## 1.3 TIME COMPLEXITY

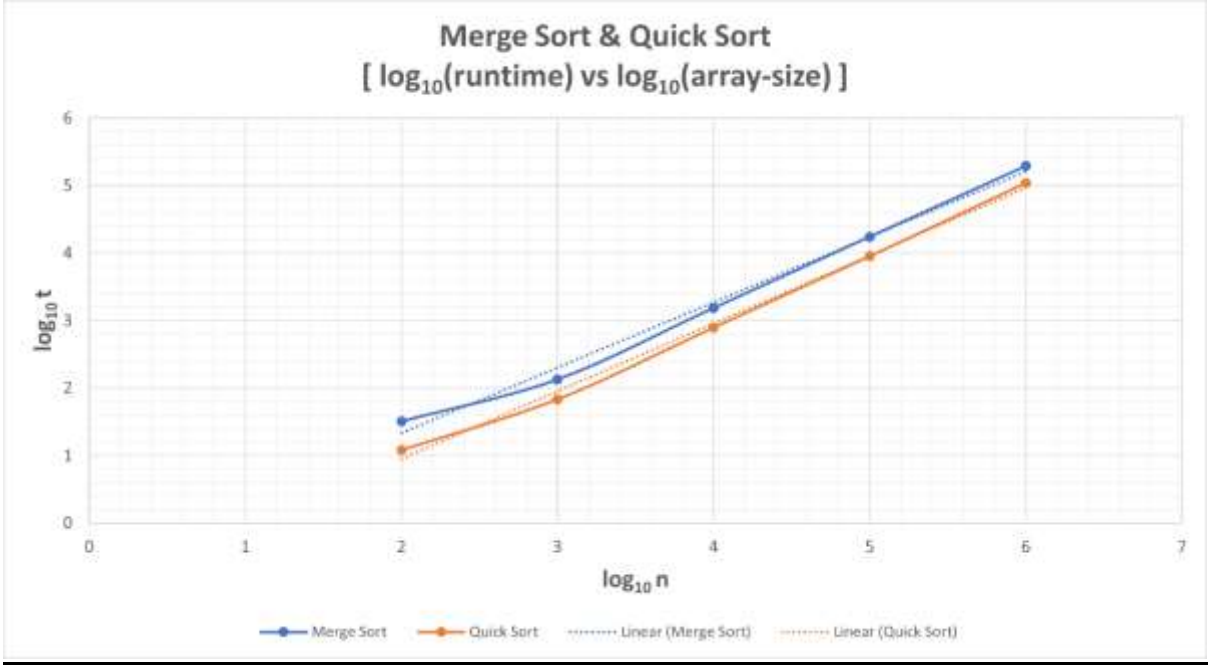
| n –→ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|
| Average running time of Quick Sort | 0 | 68 | 786 | 8960 | 108220 |
| Average running time of Merge Sort | 32 | 134 | 1536 | 17348 | 193674 |
| Number of times Merge Sort outperformed Quick Sort | 0 | 10 | 52 | 14 | 0 |

**Inference:**

Though the average number of comparisons in Quick Sort were more than that of Merge Sort, Quick Sort outperforms Merge Sort in maximum cases. Thus, Quick Sort is more efficient than Merge Sort and the number of comparisons does not directly influence the time complexity in this case. For larger values of n, since the array elements are randomised,

Quick Sort easily beats Merge Sort in all cases. The number of times we run the Quick Sort algorithm depends totally on the elements of the array and the element we choose as pivot. A suitable pivot can drastically reduce the running time of the algorithm. Whereas, in Merge Sort, the algorithm runs for a fixed number of times ($\log_2 n$) irrespective of the array elements.
One other reason for the efficiency of Quick Sort is that it sorts the array in place and takes only O(1) extra space.

**Plots for $\log_{10}$(running time) vs $\log_{10}$(array size):**

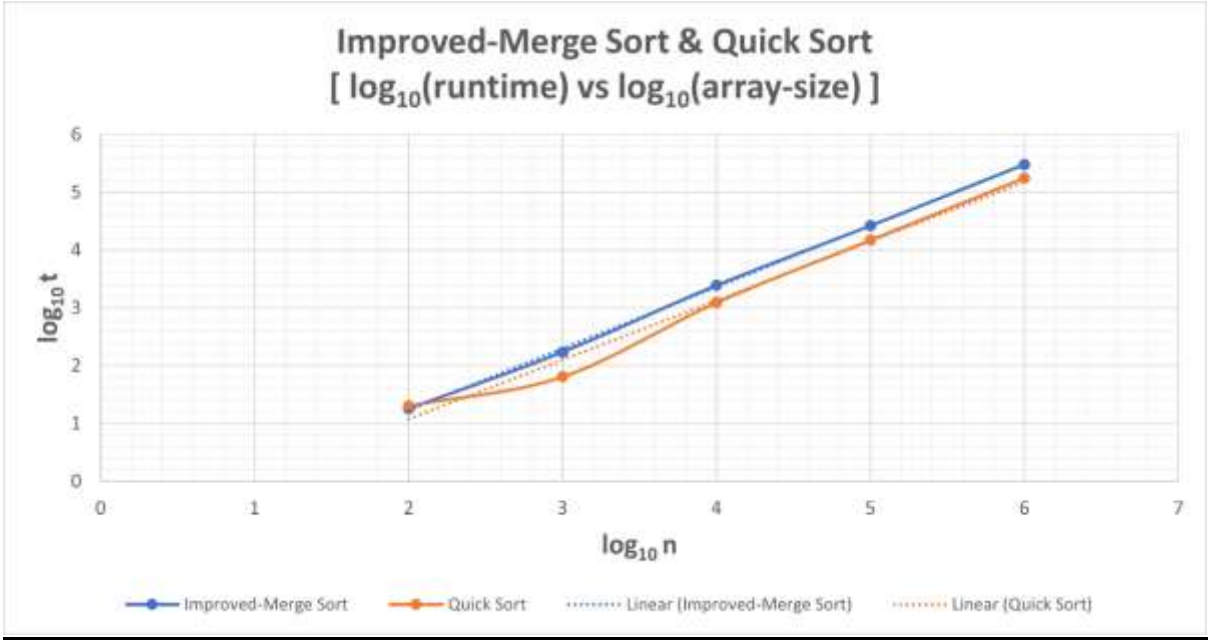



**Inference:**
The graph clearly depicts that the average running time of Quick Sort for a randomly generated array of numbers is better than that of Merge Sort.

## 1.4 IMPROVED MERGE SORT

| n –→ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|
| Average running time of Quick Sort | 20 | 64 | 1230 | 14830 | 176834 |
| Average running time of Merge Sort | 18 | 172 | 2460 | 26402 | 302900 |
| Number of times Merge Sort outperformed Quick Sort | 1 | 8 | 93 | 25 | 0 |

**Plots for $\log_{10}$(running time) vs $\log_{10}$(array size):**




**Inference:** On making slight changes in the Merge Sort algorithm (not invoking merge() if the all the elements of the left sub-array are less than that of right sub-array), the Improved Merge Sort was successful in beating Quick Sort more number of times and the running time of the Improved Merge Sort also got closer to the runtime of Quick Sort, as can be seen from the graph. However, this Merge Sort could still not beat Quick Sort for large values of n and is still quite inefficient compared to Quick Sort.

## TASK 2: RELIABILITY OF QUICK SORT

Note: 1) Number of iterations per value of n: 500
2) Units of time is microseconds (μs)
3) Data used is from Task 1.3

| n –→ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|
| Average running time of Quick Sort | 0 | 68 | 786 | 8960 | 108220 |
| Number of cases when the run time exceeds average by 5% | 1 | 10 | 220 | 222 | 2 |
| Number of cases when the run time exceeds average by 10% | 1 | 10 | 220 | 222 | 2 |
| Number of cases when the run time exceeds average by 20% | 1 | 10 | 220 | 126 | 1 |
| Number of cases when the run time exceeds average by 30% | 1 | 10 | 52 | 43 | 0 |
| Number of cases when the run time exceeds average by 50% | 1 | 10 | 52 | 11 | 0 |
| Number of cases when the run time exceeds average by 100% | 1 | 10 | 52 | 0 | 0 |

**Inference:**

From the above data, we conclude that quick sort does not deviate much from its average time complexity when we have an array of uniformly generated random numbers. Even the cases where the run time exceeds average by 30% are very less. So even though the worst-case time complexity of Quick Sort is $O(n^2)$, it is rarely achieved when we have randomly generated arrays. Also, for arrays of larger size, since the numbers are randomly generated, the probability of encountering arrays which make the algorithm deviate from its average time are even lesser.