BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
Department of Computer Science and Engineering

Course: CSE204

Report: SORTING ALGORITHM COMPARISON (Merge Sort and Quick Sort)

ID: 1805055

Name: Shamit Fatin

COMPLEXITY ANALYSIS

Merge Sort

Merge Sort falls into divide and conquer paradigm. We can fit its procedures under the steps of a generic divide and conquer algorithm.

Divide: Divide the n-element sequence into two subsequences of n/2. This is easily obtainable in linear time. Thus, the time complexity of this step is $\theta(1)$.

Conquer: Now, as we have divided the sequence into two subsequences, we can solve them, using half amount of time for the real algorithm. If the time complexity of the original algorithm is $T(n)$, then time complexity for this step is $2T(n/2)$.

Combine: Merge the two sorted n/2-element array into one n-element array. This is a linear procedure, talking almost n-steps. Thus, the time complexity of this step is $\theta(n)$.

That means, the time complexity of the algorithm can be described as a recursive relation,

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(1) + \theta(n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) \qquad [\theta(1) \text{ can be removed from this case}]$$

Using master method, we can solve the equation.

According to master method, if $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ and $f(n) = \theta\left(n^{\log_b a}\right)$, then, $T(n) = \theta(n^{\log_b a} \lg n)$

Here, a = 2, b = 2

Therefore, $T(n) = \theta(n \lg n)$

The time complexity of merge sort is $\theta(n \lg n)$.

Quick Sort

Quick Sort also falls into divide and conquer algorithm paradigm.

Divide: Partition the array A [p ... r] into two subarrays A [p ... q – 1] and A [q + 1 … r], where A [p ... q – 1] will contain values smaller than A[q] and A [q + 1 … r] will contain values larger than A[q].

This is where the analysis of quick sort differs from Merge sort. In marge sort, we know that the divide step will divide the array into two equal sub sections, hence the step to do merge sort for both of the sub sections will be half of the steps needed for the original array. But in quick sort we cannot predict how the array will be divided, hence the steps to do merge sort for both subsections can be between the multiples of n-1 and n/2 if the parent array is n-element long.

Conquer: This is where we sort the divided subsections.

Combine: Since, we have already sorted the subsections and thus the actual array, we don't need to combine them.

So, we can see that the analysis of quick sort can be quite tricky, because of the inability to predict the division process.

However, if we want to calculate the worst-case scenario, then we can find an equation to satisfy the time complexity of quick sort,

$T(n) = T(n-1) + \theta(n)$.

To which we can apply master method and get, $T(n) = \theta(n^2)$

On the other hand, if we want to calculate for the best-case scenario, then the equation we get is,

$T(n) = 2T(n/2) + \theta(n)$.

This equation is similar to merge sort. So, the time complexity becomes, $T(n) = n \lg n$.

There is, however, a better way to predict the possibility of which time complexity quick sort will acquire. That is using probabilistic analysis.

We start by saying,

If partition in quick sort takes in total X comparisons throughout the entire process, then we can say that, for an n-element array, quick sort will take O(n + X) time. Therefore, calculating X is enough for our goal.

If we define our Array as $A\ [z_1, z_2, z_3\ \ldots\ z_n]$

$$X = \sum_{i=1}^{i=n-1} \sum_{j=i+1}^{j=n} X_{ij}$$

$$E[X] = E\left[\sum_{i=1}^{i=n-1}\sum_{j=i+1}^{j=n} X_{ij}\right] \qquad [\ taking\ expections\ of\ both\ side\ ]$$

$$= \sum_{i=1}^{i=n-1}\sum_{j=i+1}^{j=n} E[X_{ij}]$$

$$= \sum_{i=1}^{i=n-1}\sum_{j=i+1}^{j=n} \Pr\{X_{ij}\ to\ happen\} \qquad [\ E[Xi] = Pr\{i\ event\ to\ occure\}\ ]$$

Now,

$$\Pr\{X_{ij}\ to\ happen\} = \Pr\{z_i\ is\ compared\ with\ z_j\}$$

$$= \Pr\{z_i or\ z_j\ is\ first\ pivot\ chosen\ from\ Z_{ij}\}$$

$$= \Pr\{z_i\ is\ first\ pivot\ chosen\ from\ Z_{ij}\} + \Pr\{z_j\ is\ first\ pivot\ chosen\ from\ Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$

$$= \frac{2}{j-i+1}$$

$$\therefore E[X] = \sum_{i=1}^{i=n-1}\sum_{j=i+1}^{j=n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{i=n-1}\sum_{k=1}^{k=n-i} \frac{2}{k+1} \qquad [\ k = j-i\ ]$$

$$< \sum_{i=1}^{i=n-1}\sum_{k=1}^{k=n-i} \frac{2}{k}$$

$$= \sum_{i=1}^{i=n-1} O(\lg n)$$

$$= O(n \lg n)$$

$\therefore E[X] < O(n \lg n)$

$\therefore time\ complexity\ of\ quick\ sort = O(n + n \lg n) = O(n \lg n)$

Given an ascending or a descending input order to sort using quick sort algorithm, we can always expect the worst-case scenario, because in those cases, the partitioning will always be on the top or on the bottom of every element. Thus, time complexity of ascending or a descending ordered input, the complexity will be $O(n^2)$. In every other case however, we can expect complexity of O (n lg n).

DATA TABLE

To see how our calculation matched with real life example, we constructed both quick sort and merge sort algorithm in a device and ran those algorithms for various ordered inputs to get time data.

Here is the average time data of the experiment.

Notice that n means the size of the array used. Every time used in the table is in micro second unit.

The device used to run the experiment had the following components

CPU: AMD Ryzen 7 3700X 8-Core 16-Thread Processor, 3.60 GHz

Ram: 16GB DDR4 with 3200MHz

OS: Windows 10

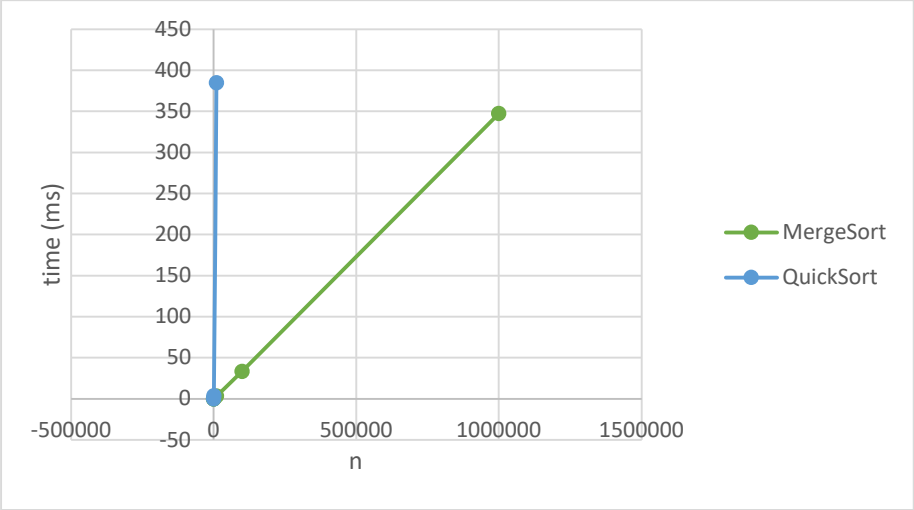
Language: Microsoft Visual C++

Input Order	**Sorting Algorithm**	***n = 10***	***n = 100***	***n = 1,000***	***n = 10,000***	***n = 100,000***	***n = 1,000,000***
Ascending	*Merge*	0.004	0.034	0.351	3.43	33.77	347.69
	Quick	0.0008	0.042	3.82	385.36	35744.9	2500000.1
Descending	*Merge*	0.004	0.034	0.352	3.46	35.60	363.10
	Quick	0.0007	0.0315	3.202	320.86	32377.12	1122170.8
Random	*Merge*	0.0049	0.0515	0.4667	4.0825	40.6042	427.481
	Quick	0.0008	0.0074	0.1221	1.3839	16.280	207.114

Graph

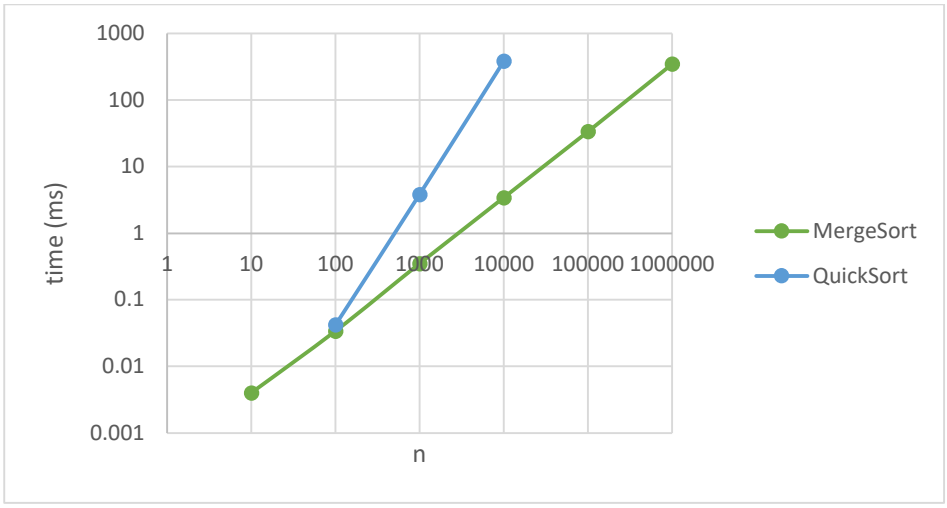
Ascending Order

In our analysis section, we have predicted that for ascending order, quick sort will take $O(n^2)$ time to complete the task, where merge sort, for any order takes O(n lg n) amount of time. If we plot the graph from our data, we can see that quick sort quickly shoots off to greater times faster than merge sort. This indicates that merge sort is efficient for ascending order than quick sort.

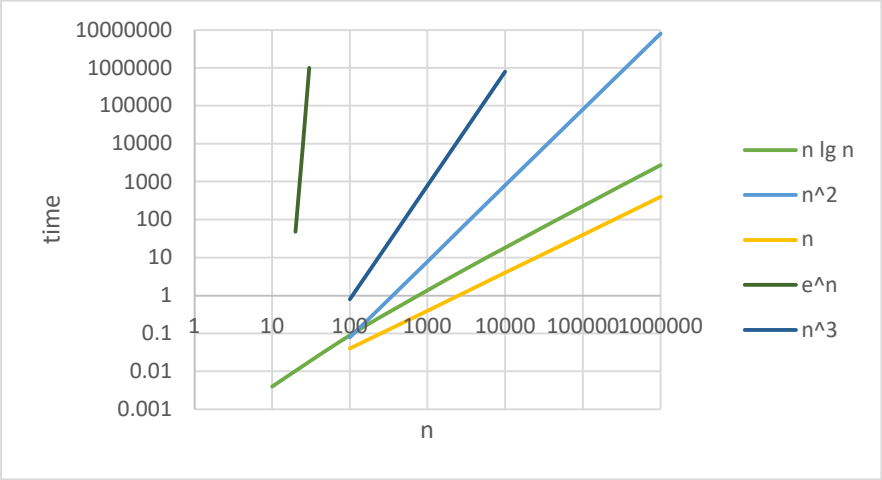


Although the graph can predict the efficiency of algorithms for ascending ordered input, it is still hard for anyone to predict the running time of the algorithms. To get a better understanding in it we can plot the graph logarithmically and compare the result with some known common functions to see which bares more resemblance to our algorithm running time.

Here's the plot of our result in logarithmic scale,



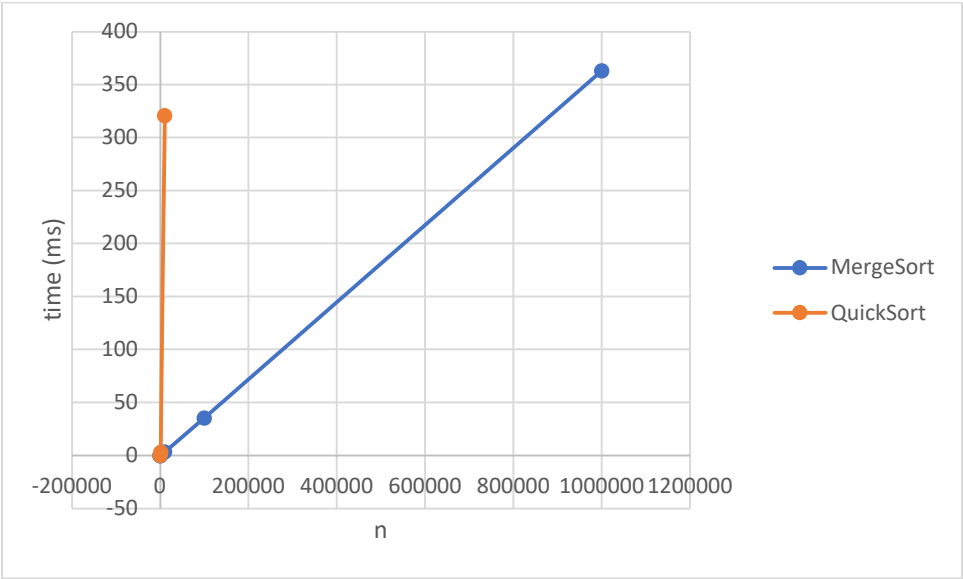
And here are some well-known functions to compare the graph to,



We can now compare the two graphs and say that merge sort behaves like O (n lg n) and quick sort behaves like O (n^2)

Descending Order

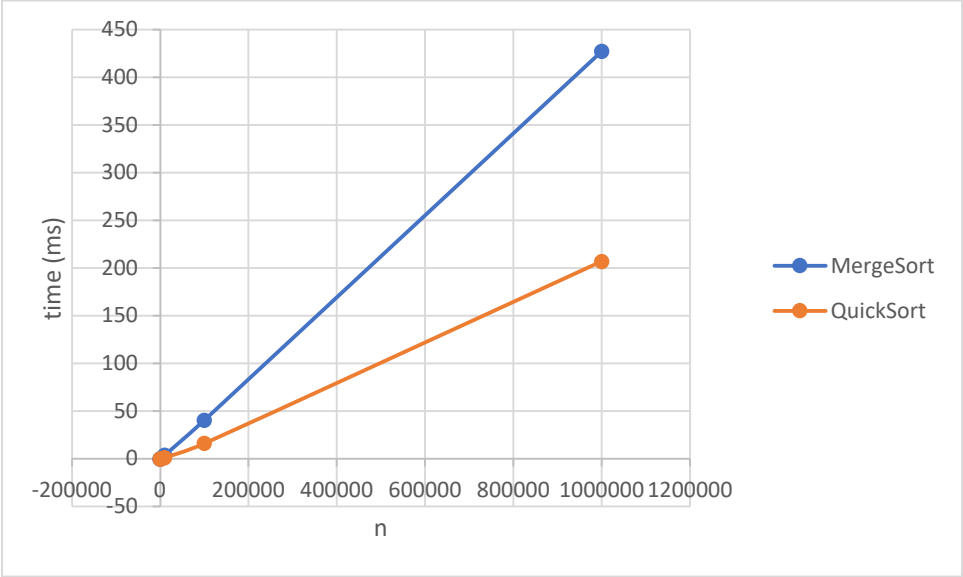
In our analysis section, we have predicted that for descending order, quick sort will take O(n^2) time to complete the task, where merge sort, for any order takes O (n lg n) amount of time. If we plot the graph from our data, we can see that quick sort quickly shoots off to greater times faster than merge sort. This indicates that merge sort is efficient for ascending order than quick sort.



This graph is similar to previous graph, so we should expect them to behave similar. Thus, our prediction is correct.

Random Order

We predicted that for random order, both quick sort and merge sort will take almost same amount of time and both of the algorithm will take n lg n runtime to complete the task. We now plot the data into the graph.



We can see that, both algorithms have almost same amount of growth rate, although quick sort in this case takes less time. This is because, although both algorithm takes O (n lg n) run time, quick sort doesn't have to do some steps that merge sort has to do. For example, merge sort has to create two new arrays in every merging step, where quick sort can sort in its existing place and thus taking fewer time. As a result, generally quick sort is used intensively rather than merge sort. Even, with clever techniques, we can make sure that the partitioning in quick sort can become almost even so that it runs always on best-case scenario.