Assignment 2

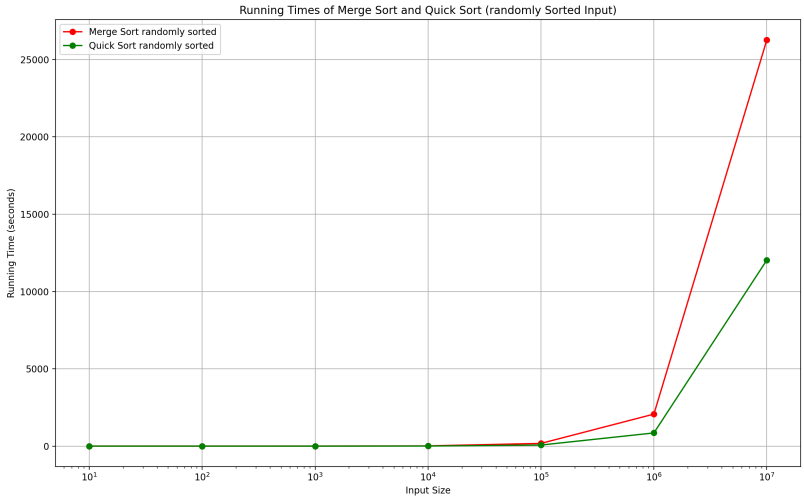
CS 514 – Algorithms

Submitted By: Aman Pandita

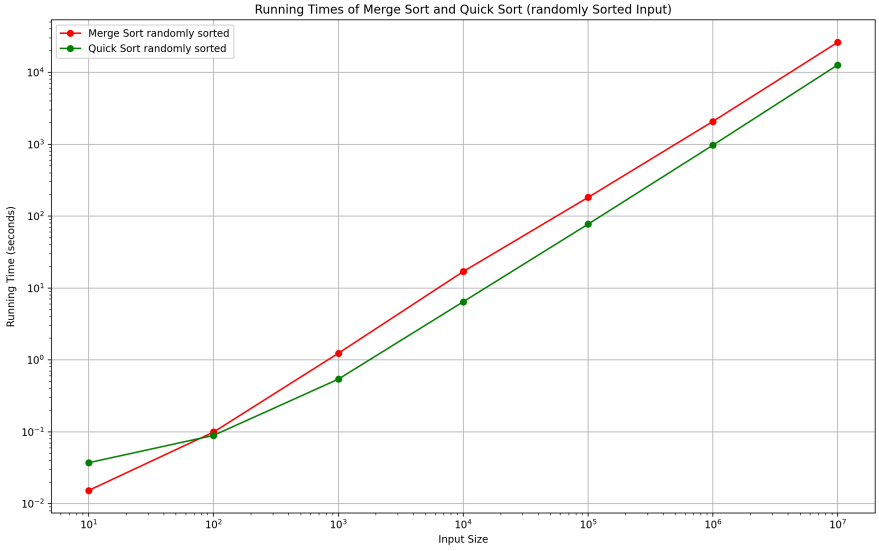
Onid: panditaa@oregonstate.edu

2. Plot the running times of the two algorithms against the *size* of the input (up to 10^7 numbers) when the input arrays are randomly sorted. Do the same when the inputs are already sorted. What functions best characterize the running times? Compare your results with what is expected from the theoretical analysis. Discuss the results and their implication.

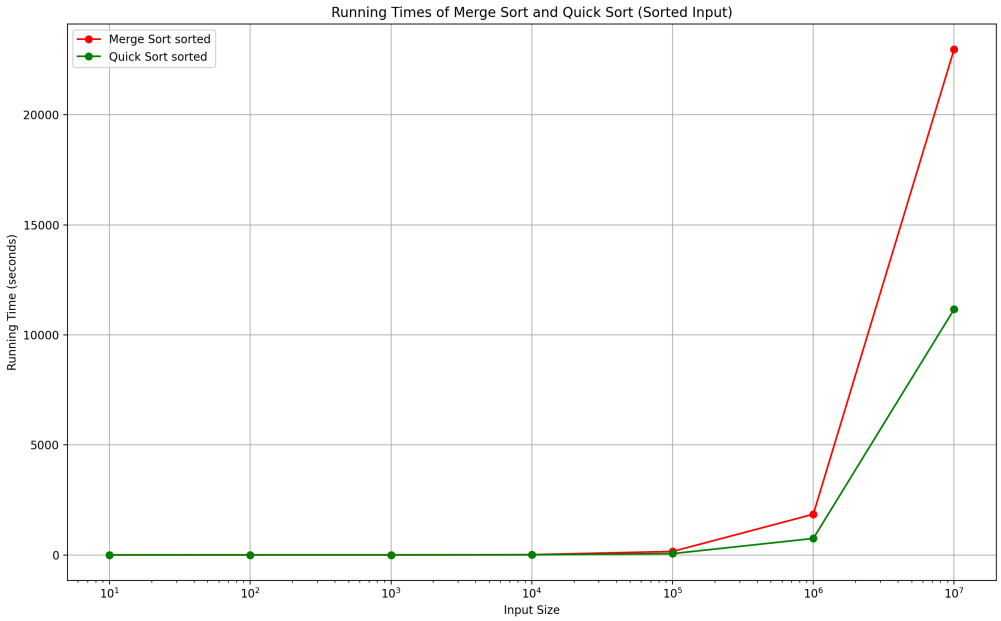
For Merge Sort vs Quick Sort when randomly sorted:



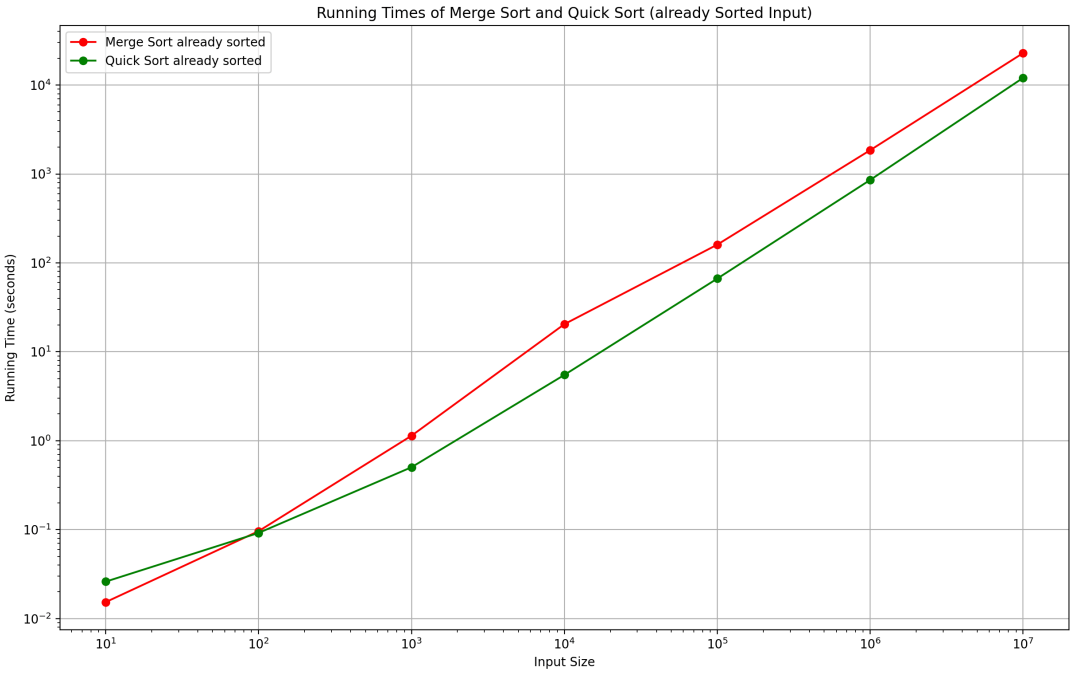
JUST A EXTRA REPRESENTATION WHERE X AND Y AXIS ARE LOG:



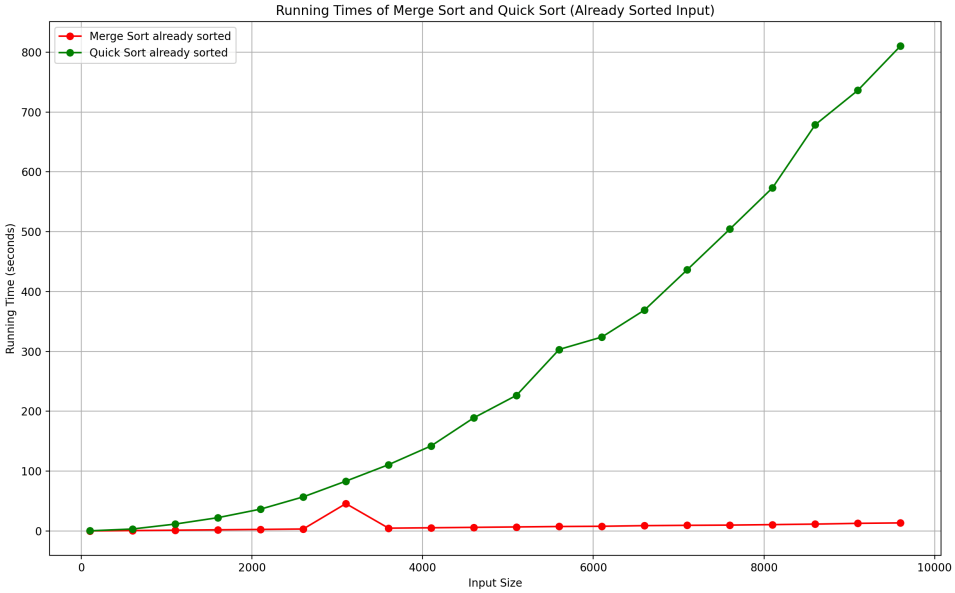
For Merge Sort vs Quick Sort when already sorted:



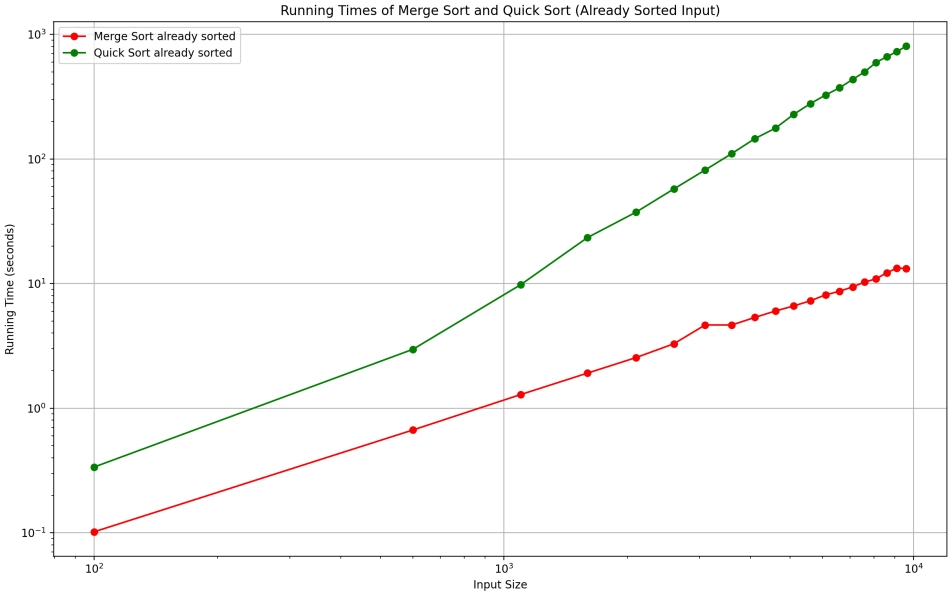
JUST A EXTRA REPRESENTATION WHERE X AND Y AXIS ARE LOG:



For Merge Sort vs Quick Sort when already sorted but the pivot was always selected as the first element (We only tested max of 10^4 input sizes):



JUST AS AN EXTRA REPRESENTATION:



Analyzing the results:

Merge Sort Expectation vs Experimental Data:

Theoretical Expectation For Randomly Sorted Data:

The time complexity of Merge Sort is expected to be **O (n log n)** for all cases of best case, average case and worst Case. The derivation is as follows:

$T(N) = 2 * T(N/2) + M(N) = 2 * T(N/2) + constant * N$

$T(N) = 2 * [2 * T(N/4) + constant * N/2] + constant * N = 4 * T(N/4) + 2 * N * constant$

.

.

.

$T(N) = 2^k * T(N/2^k) + k * N * constant$

$T(N) = N * T(1) + N * \log_2 N * constant$

$T(N) = N + N * \log_2 N$

For already Sorted Data:

As it has a O (n log n) complexity for all the cases, it does not benefit from the already sorted data.

Experimental Data:

As we can see in the graphs above, that the time required does not rise tenfold as the amount of the input does, but rather more than linearly. Given that it is greater than linear but less than quadratic, this sort of property fits with the ***n log n*** complexity of merge sort algorithm.

Quick Sort Expectation vs Experimental Data:

Theoretical Expectation for Randomly Sorted Data:

In the worst-case situation, the Quick Sort algorithm has a temporal complexity that might drop below **$O(n^2)$**. This happens in some emergency situations, such as when the pivot pick is unlucky. However, Quick Sort has an efficient time complexity of **O (n log n)** in a probabilistically average setting. It is important to emphasize that when the input data is sorted randomly, we may anticipate the method to run quite near to its average-case time complexity of O(n log n). This is because the data's randomization helps prevent a tendency toward the algorithm's worst-case performance, thus improving the sorting process.

Theoretical Expectation for Already Sorted Data:

As it has a **O (n log n)** complexity **in average and best case**, it does not benefit from the already sorted data, however if a case where the pivot selection ends in the worst case i.e, smallest or largest number, it tends to have a time complexity of $O(n^2)$.

Experimental Data:

Much like the Merge Sort algorithm, Quick Sort's computational time intricately escalates concurrent with the augmentation of the dataset's magnitude. This observable phenomenon implicates that Quick Sort maneuvers proximately aligned to its average-case complexity of O(n log n). This implies a nuanced sophistication where the algorithm demonstrates an efficient performance characteristic, managing the sorting process with a time complexity that evolves logarithmically in correspondence with the data size, showcasing a conducive balance between performance and accuracy in sorting operations.

***Note:**

The **worst-case** time complexity for previously sorted data would be **$O(n^2)$** because the pivot selection and dividing strategy are not beneficial.

(ALSO, THIS IS ONLY POSSIBLE WHEN THE FIRST ELEMENT IS CONSIDERED TO BE THE PIVOT)

Discussion and Implications:

1. Merge Sort exhibits **O (n log n)** behavior in both the previously sorted and randomly sorted circumstances, remaining consistent with the theoretical analysis.

2. Although reliable, merge sort does not make use of the array that has previously been sorted. Surprisingly, Quick Sort does not exhibit the worst-case behavior even with the array that has already been sorted, indicating that it may be employing pivot optimization or randomness to **prevent quadratic behavior**.

3. Timing-wise, it's also doing somewhat better than Merge Sort, particularly for bigger inputs.

4. If there is a worst-case scenario while sorting the already sorted array, this is where the quick sort algorithm fails disastrously. However, upon hit and trial testing, I was able to generate the output from quick sort algorithm till a input value of **10^4 at maximum**.

Conclusion:

In conclusion, both algorithms exhibit a strong correlation between the experimental outcomes and the predicted theoretical time complexities, with Quick Sort showing a minor performance advantage, especially for bigger sizes. Note that although the system architecture, compiler optimizations, real constants and lower-order terms, among other things, may have an effect on the actual execution time, the algorithms' general complexity class is unaffected.

Complexity Analysis Quick sort:

Best case: The best-case scenario for Quick Sort is when the pivot selected is the median, which causes the array to be split into two almost equal halves at each recursive call. Since the pivot is selected randomly, it's possible to get the median, but not guaranteed.

Recursion: Due to the fact that the issue of size n is split into two approximately equal-sized subproblems, it multiplies by 2: 2T(n/2).

Work Done for Partition: The partitioning procedure compares each element with the pivot for each recursion, which takes linear time, O(n).

Recurrence Relation: T(n) = 2T(n/2) + O(n)

Derivation:

Best Case :- If pivot = Mean

$T(N) = 2T\left(\frac{N}{2}\right) + N \times \text{constant}$

Here, $T\left(\frac{N}{2}\right) = 2T\left(\frac{N}{4}\right) + \frac{N}{2} \times \text{constant}$

$T(N) = 2 \times \left(2 \times T\left(\frac{N}{4}\right) + \frac{N}{2} \times \text{Constant}\right) + N \times \text{constant}$

$= 4 \times T\left(\frac{N}{4}\right) + 2 \times \text{constant} \times N$

$\Rightarrow T(N) = 2^k \times T\left(\frac{N}{2^k}\right) + k \times \text{Constant} \times N$

then $2^k = N \Rightarrow \log 2^k = \log N \Rightarrow k = \frac{\log N}{\log 2}$

$k = \log_2 N$

Putting the value of k

$T(N) = N\,T(1) + N \log_2 N$

$T(N) = N \log N$

Time Complexity would be $O(N \log N)$

Using Master Theorem: a = 2 , b = 2 , d = 1
Since these falls under Case 2 of the Master Theorem. T(n) = O(n log n)

Average Case: The average case of Quick Sort, given that the pivot is chosen randomly, is generally quite good because, on average, the partitions will be reasonably balanced.

Recursion: Due to the fact that the issue of size n is split into two approximately equal-sized subproblems, it multiplies by 2: 2T(n/2).

Work Done for Partition: The partitioning procedure compares each element with the pivot for each recursion, which takes linear time, O(n).

Recurrence Relation: T(n) = 2T(n/2) + O(n)

Derivation:

Average Case:

Let's says array divided in two parts k, N-k.

$$T(N) = T(N-k) + T(k)$$

$$= \frac{1}{N}\left[\sum_{i=1}^{N-1} T(i) + \sum_{i=1}^{N-1} T(N-i)\right]$$

[Here, $\sum_{i=1}^{N-1} T(i) = \sum_{i=1}^{N-1} T(N-i)$

Then, $T(N) = \frac{2\sum_{i=1}^{N-1} T(i)}{N}$

$$N \times T(N) = 2\sum_{i=1}^{N-1} T(i) \quad -(1)$$

we can also write it as:

$N \to N-1$

Then, $(N-1)\,T(N-1) = 2\sum_{i=1}^{N-2} T(i) \quad -(11)$

Now, Subtract (1) from (11)

$$NT(N) - (N-1)T(N-1) = 2\left[\sum_{i=1}^{N-1} T(i) - \sum_{i=1}^{N-2} T(i)\right]$$

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + N^2 \text{constant} - (N-1)^2 \times \text{constant}$$

$$N \times T(N) = T(N-1) \times (2+N-1) + \text{constant} + 2 \times N\text{constant} - \text{Constant}$$

$$= (N+1)T(N-1) + 2N\text{constant}$$

Now, Divide it by $N(N-1)$

$$\frac{T(N)}{(N+1)} = \frac{T(N-1)}{N} + \frac{2\,\text{constant}}{N+1} \quad (1)$$

Now, $N = N-1$

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2 \times \text{constant}}{N+1}$$

Let's put $N = N-1$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2\,\text{constant}}{N}$$

Using Master Theorem: a = 2 , b = 2 , d = 1
Since these falls under Case 2 of the Master Theorem. T(n) = O(n log n)

Worst Case: The worst-case scenario for Quick Sort is when the smallest or largest element is always chosen as the pivot. However, because we're choosing the pivot at random, the chances of consistently choosing the smallest or largest element are low.

Recursion: The problem of size n is divided into a problem of size (n-1) and a problem of size 1: T(n-1).

Work Done for Partition: Since all elements are compared to the pivot, it again takes linear time: O(n).

Recurrence Relation: T(n) = T(n-1) + O(n)

Derivation:

Worst Case:-

$T(N) = T(N-1) + N * \text{constant}$

$= T(N-2) + (N-1) * \text{constant} + N * \text{Constant}$

$\Rightarrow T(N-2) + 2 * N * \text{constant} - \text{constant}$

$= T(N-3) + 3 * N * \text{constant} - 2 * \text{constant} - \text{constant}$

$\Rightarrow T(N-k) + k * N * \text{Constant} - (k-1) * \text{constant} -$
$\quad\quad - 2 * \text{Constant} - \text{constant}$

$= T(N-k) + k * N * \text{constant} * (k * (k-1))/2$

Let's put $k = N$

$T(N) = T(0) + N * N * \text{constant} - \text{Constant} * (N * (N-1)/2)$

$= N^2 - N * (N-1)/2$

$= N^2/2 + N/2 \Rightarrow$ So, the complexity would be $O(N^2)$

Merge Sort	Quick Sort
Best case: O (n log n) **Average case:** O (n log n) **Worst case:** O (n log n)	**Best case:** O (n log n) **Average case:** O (n log n) **Worst case:** O (n^2)

Commentary of the analysis when inputs are randomly sorted:

The fewer hidden constant variables make Quick Sort, on average, quicker. Its built-in partitioning also helps to conserve space.
In contrast to Quick Sort, Merge Sort has a constant time complexity of **O (n log n)** for all situations.
Merge Sort would be preferred if stability was a priority.
Additionally, Quick Sort is more space-efficient because it takes up less room.
Furthermore, Quick Sort may better optimize efficiency by adjusting to different pivot selection techniques.

Commentary of the analysis when inputs are already sorted:

Merge Sort consistently performs well and retains its **O (n log n)** time complexity for inputs that have previously been sorted, making it a consistent and reliable option.

However, Quick Sort needs to be carefully considered. To prevent the **O(n^2)** worst-case situation, the pivot selection mechanism must be improved. Quick Sort can also function effectively and attain **O (n log n)** time complexity with the appropriate optimization *(e.g., picking a middle element or employing a median-of-three technique).*

Without **pivot optimization**, Quick Sort runs the danger of reaching the worst-case time complexity for inputs that have already been sorted. Thus, unless the Quick Sort implementation is well optimized, Merge Sort could be a more dependable option in such situations.

Overall, Merge Sort is always going to be more stable and consistent no matter what the array orientation is. However, when Quick Sort is used, it performs comparatively well when the input is randomly sorted considering that the random selection of pivot is not randomly selected to the worst-case scenario that is making the time complexity **O(n^2)**

A key finding was that when the input number is as high as 10^7, Quick sort used to reach maximum recursion depth, this was tackled by using a recursion depth limiter:

```
sys.setrecursionlimit(10000000)
```

AND THE MAXIMUX INPUT SIZE OF 10^4 WAS ONLY ABLE TO PROCESS.