# BLG-335E PROJECT 2 REPORT

**ONAT ŞAHİN - 150150129**

## Notes About My Code

In my code, I used the stoi() function to convert strings to integers. This function is a C++11 function. Therefore, my code should be compiled with "g++ -std=c++11 main.cpp residence.cpp" command. I used this command to get the runtime values for part b. However, for part c, I used the "g++ -std=c++11 main.cpp residence.cpp -O2" to further optimize the code and get the results quicker since the algorithm takes a very long time for large number of values in the worst case. Also, I found the results using my own computer since ssh server has a 5 minute time limit and it was not sensitive enough to calculate the time in miliseconds for some reason. Finally, I chose the pivot element to be the last element in the array for my quicksort algorithm.

**a.**

Cost of partition = Θ(n)

In the worst case, when an array is partitioned, one subarray contains n-1 of n elements while the other subarray is empty. Therefore the recurrence for the worst case is:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \ldots + \Theta(2) + \Theta(1) \\ &= \Theta(n(n+1)/2) \\ &= \Theta(n^2) \end{aligned}$$

Therefore the upper bound for the worst case of quicksort is $O(n^2)$.

In the best case, when an array is partitioned, both of the subarrays contain n/2 of the n elements. Therefore the recurrence for the best case is:

$$T(n) = 2T(n/2) + \Theta(n)$$

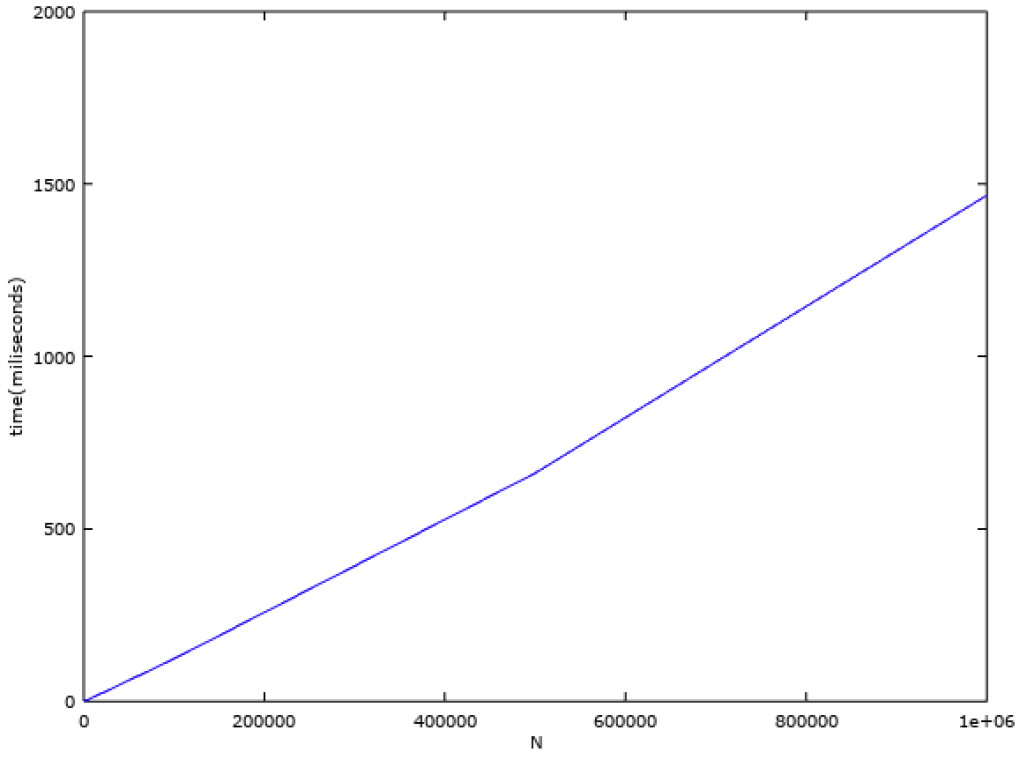
Using master theorem case 2, this recurrence can be solved.

$$T(n) = \Theta(n\log n)$$

Therefore the upper bound for the best case of quicksort is $O(nlogn)$.

**b.**

| N | Average Runtime (miliseconds) |
|---|---|
| 10 | 0.0111 |
| 100 | 0.1125 |
| 1000 | 1.0722 |
| 10000 | 10.8128 |
| 100000 | 122.8614 |
| 500000 | 661.5961 |
| 1000000 | 1467.007 |

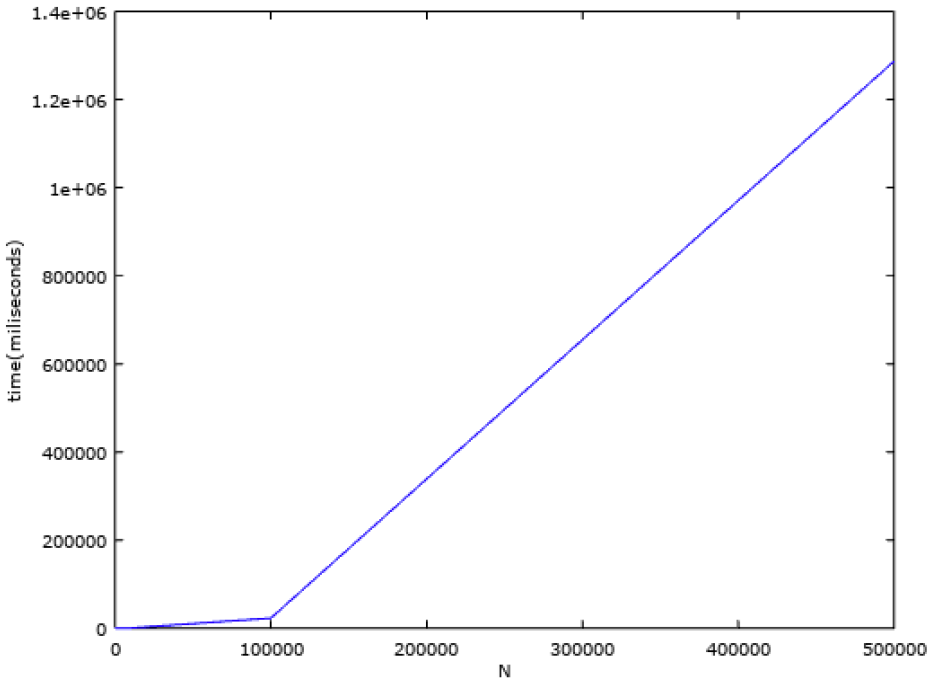



This results are very close to the $O(nlogn)$ result that we found. This result is true for the best and average cases. However the results are not %100 accurate since runtime is affected by many other things other than complexity.

**c.**

In the worst case for quicksort, when an array is partitioned, one subarray contains n-1 of n elements while the other subarray is empty. We can achieve this by constructing a dataset where the elements are sorted or reverse sorted since we select the pivot from the beginning or the end. The results below are found using a sorted dataset. Compilation is done with the "g++ -std=c++11 main.cpp residence.cpp -O2" command to further optimize the code and get the results quicker since the algorithm takes a very long time for large number of values in the worst case.

| N | Average Runtime (miliseconds) |
|---|---|
| 10 | 0.0034 |
| 100 | 0.0289 |
| 1000 | 3.1364 |
| 10000 | 114.0323 |
| 100000 | 22785.38 |
| 500000 | 1287180 |




It can be seen that in the worst case the runtime increases exponentially and is really close to $O(n^2)$. To

overcome the worst case, we can select the pivot randomly instead of always selecting the first or the last element. This will minimize the odds of encountering the worst case.

**d.**

Quicksort is not stable. Consider this dataset where the last 2 elements are equal:

2,20,20,male,99927,8600000US99927
4,80,84,female,51562,8600000US51562
1,70,74,male,99923,8600000US99923
0,40,44,female,906,8600000US906
0,80,84,female,906,8600000US906

The order below is the order after the first partition is done where the last element is the pivot.

0,80,84,female,906,8600000US906
4,80,84,female,51562,8600000US51562
1,70,74,male,99923,8600000US99923
0,40,44,female,906,8600000US906
2,20,20,male,99927,8600000US99927

The order of the equal elements are changed. And since this is a reverse sorted set and the pivot is now the first element, next calls of the quicksort function will be from indexes 0 to -1 and 1 to 4. This means the order of the equal elements won't change in the next partitions, keeping the order of the equal elements different from the input order. This proves that quicksort is not a stable sorting algorithm.