BLG35E – Analysis of Algorithms 1, Fall 2017

Project 2

Yunus Güngör 150150701

Compiling: compiled with command: `g++ main.cpp -std=c++11`

a. Analysis of Quicksort

Quicksort is an algorithm that runs with recurrence. Quicksort has a divide and conquer approach for sorting an array. First step of quicksort is to partition the array based on a pivot. Everything smaller than pivot goes left of the pivot and everything bigger than pivot goes right of the pivot. Since quicksort is an in-place sorting algorithm that uses no extra space, array will have two unsorted parts, one on the left of the pivot and one on the right of the pivot. Using same partition algorithm again and again until there is only one element left, solves sorting problem very efficiently. Pseudocode is given below.

```
QUICKSORT(A,p,r)
1  if p < r
2     then q ← PARTITION(A,p,r)
3          QUICKSORT(A,p,q-1)
4          QUICKSORT(A,q+1,r)
```

```
PARTITION(A,p,r)
1  x ← A[r]
2  i ← p - 1
3  for j ← p to r-1
4       do if A[j] ≤ x
5          then i ← i + 1
6               exchange A[i] ↔ A[j]
7  exchange A[i+1] ↔ A[r]
8  return i+1
```

Figure 1: Pseudocode for quicksort (Analysis of Algorithms 1 Course Slides ITU, Dr. Çataltepe & Dr. Ekenel)

This algorithm called with "quicksort(A,0,length(A))". In figure 1, A is the array we want to order. p is the starting point of the array that we want to put in order. And r is the ending point of the array we want to put in order. Using recurrence calls to function itself, p and r values changed constantly according to the divide and conquer approach and divides array. Partition function works as explained above. Partition function places items according to pivot. After completing its task function returns new location of the pivot. Using that new location as ending point and after that as starting point helps to solve problem by creating smaller arrays until an array with one element achieved. Array with one element will be sorted already, so combining all smaller arrays with one element through partition function will result in a sorted array.

According to the for loop in partition function, partition function is $\odot(p-r)$. Since there is two different calls to the quicksort functions as a recurrence, partition function will work twice with parameters p, q-1 and q+1,r therefore partition function will have $\odot(n)$.

Worst case for quicksort is when partition algorithm returns an i value such that, divides array into two with $n-1$ elements and 0 elements. This happens when pivot is maximum or minimum. Since quicksort using last element as pivot when the given array is in order, worst case will happen. In this case time it will take is:

$T(n) = T(n-1) + \Theta(n)$

$T(n) = \Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(1)$

$T(n) = \Theta(\sum_{k=1}^{n} n) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$

Best case for quicksort is when partition algorithm divides the array into two with equal number of elements. Each part of the array will have n/2 elements (Assuming there is even number of elements). In this case time it will take is:

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

Using recurrence trees:

$\log_2 n$ steps

$T(n)$ $+\Theta(n)$

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

$T(n/2)$ $T(n/2)$ $T(n/2)$ $T(n/2)$ $+4\,\Theta(n/4) = \Theta(n)$

… … … …

$T(1)$ … $T(1)$ … $T(1)$ … $T(1)$ … $T(1)$ … $T(1)$

$$T(n) = \Theta(n \log_2 n) = \Theta(nlogn)$$

Assuming partition will algorithm will divide the array into two with $9n/10$ and $n/10$ elements. In this case time it will take is:

$T(n) = T(9n/10) + T(n/10) + \Theta(n)$

Using recurrence trees:

$\log_{10} n$ steps

$T(n)$ $+\Theta(n)$

$T(9n/10)$ $T(n/10)$ $+\Theta(9n/10) + \Theta(n/10) = \Theta(n)$

$T(81n/100)$ $T(9n/100)$ $T(n/100)$ $T(9n/100)$ $+\Theta(n)$

… … … … … … … …

$T(1)$ … $T(1)$ … $T(1)$ … $T(1)$ … $T(1)$ … $T(1)$ … $T(1)$

$$T(n) = \Theta(n \log_{10} n) = \Theta(nlogn)$$

In average and best case quicksort will be $O(nlogn)$ and in worst case quicksort will be $O(n^2)$.

b. Experiments Done with Quicksort

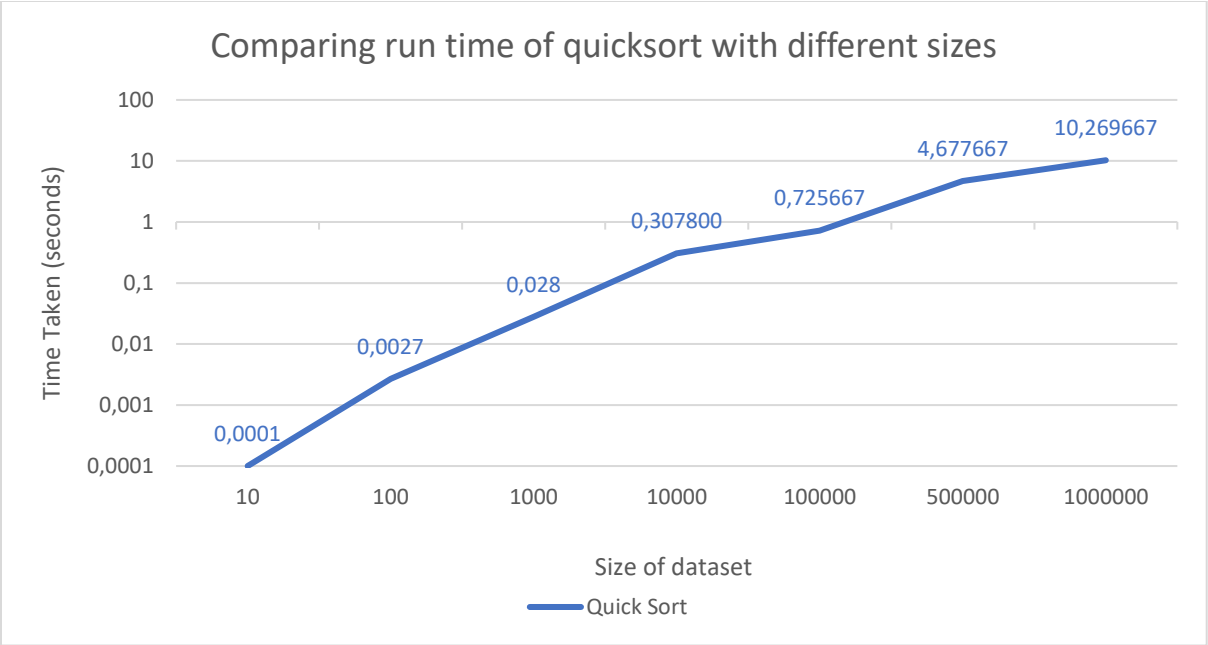

Figure 2: Logarithmic graph for run time of quicksort.

We can see in the graph there is almost a linear line. Since graph is based on logarithmic values (time takes is in logarithmic values.) it is possible to see $\Theta(nlogn)$ time to run.

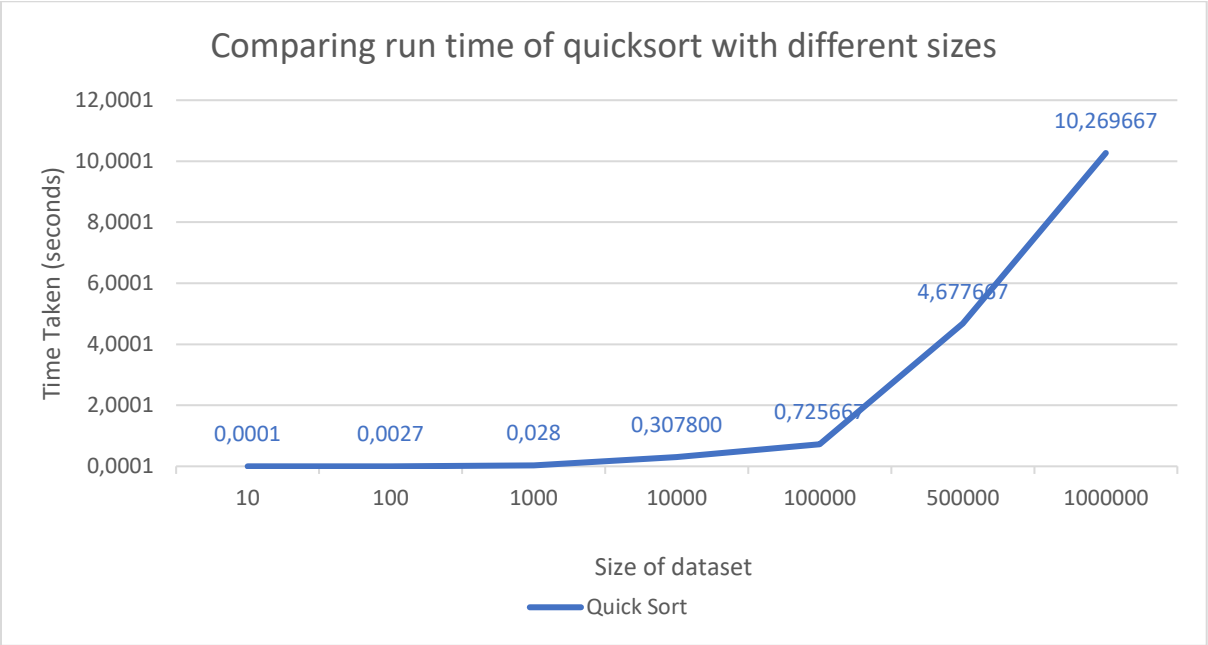

Figure 3: Regular graph for run time of quicksort.

In regular graph we can see the logarithmic part of $\Theta(nlogn)$ in action. With the increasing number of digits in size of the dataset, time takes to run algorithm increases rapidly.

c. Worst Case for Quicksort

As mentioned at part a in this report, worst case for quicksort is when partition algorithm returns an i value such that, divides array into two with $n-1$ elements and 0 elements. This happens when pivot is maximum or minimum. Since quicksort using last element as pivot when array is ordered worst case will happen. In this case time it will take is:

$$T(n) = T(n-1) + \Theta(n)$$

$$T(n) = \Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(1)$$

$$T(n) = \Theta(\sum_{k=1}^{n} n) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Using a randomized partition algorithm that will pick pivot randomly with equal probabilities for each element will solve this issue, and will make worst case unlikely to happen.

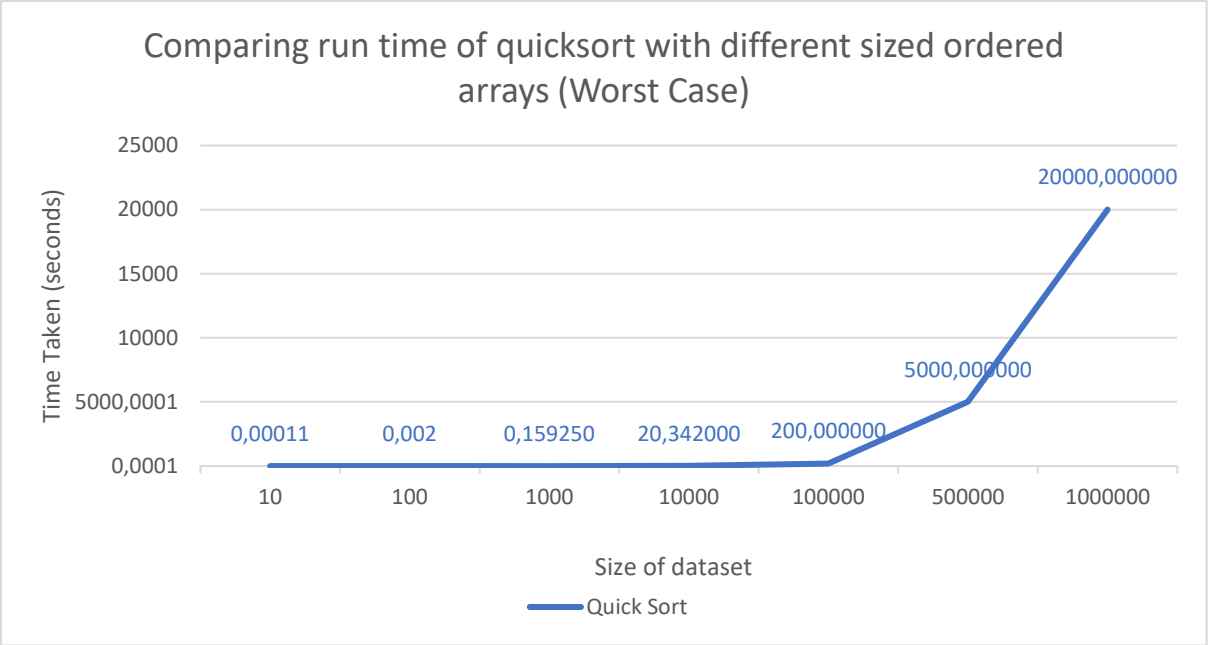

Figure 4: Regular graph for run time of quicksort in worst case (Values for 100000,500000 and 1000000 are guessed based on previous data and $\Theta(n^2)$)

We can see how worst-case effects quick sort in figure 4. Since it will take too much time for datasets with size 100000, 500000 and 1000000, those values guessed based on previous data.

d. Stability of Quicksort

Quicksort is not a stable algorithm. For example sorting given dataset, will change order in the input if there is equal keys.

population	minimum_age	maximum_age	gender	zipcode	**geo_id**
50	30	34	female	61747	**8600000US64120**
50	85		male	64120	**8600000US64120**
50	30	34	male	95117	**8600000US95117**
230	60	61	female	74074	**8600000US74074**
230	0	4	female	58042	**8600000US74074**
524	22	24	female	75241	**8600000US75241**
10	60	61	female	70631	**8600000US70631**
13	80	84	female	17260	**8600000US17260**
1	80	84	female	80612	**8600000US80612**

Figure 5: Custom dataset designed to demonstrate stability on quicksort

population	minimum_age	maximum_age	gender	zipcode	**geo_id**
1	80	84	female	80612	**8600000US80612**
10	60	61	female	70631	**8600000US70631**
13	80	84	female	17260	**8600000US17260**
50	85		male	64120	**8600000US64120**
50	30	34	female	61747	**8600000US64120**
50	30	34	male	95117	**8600000US95117**
230	60	61	female	74074	**8600000US74074**
230	0	4	female	58042	**8600000US74074**
524	22	24	female	75241	**8600000US75241**

Figure 6: Output produced by quick sort according to population then geo_id from figure 5. Notice how data order changed for data with same values for populations and geo_id.