Bayram Barış Sarı
150130123

Analysis of Algorithms I
Assignment II
Report

Bayram Barış Sarı
150130123
Date:08/11/2017

Bayram Barış Sarı
150130123

In this assignment, we were supposed to implement quick sort and use it for given data set for different cases. I used pseudo code from lecture slides and this what I wrote:

```
int partition(Residence *R, int first, int last){
    Residence temp;
   Residence pivot = R[last];
   int i = first-1;
   for(int j=first; j<last; j++){
      if(R[j].population< pivot.population ||
(R[j].population==pivot.population && R[j].geo_id<pivot.geo_id)){
         i++;
         //swapping A[i] and A[j]
         temp = R[i];
         R[i] = R[j];
         R[j] = temp;
      }
   }
   //swapping A[i+1] and A[last]
   temp = R[i+1];
   R[i+1] = R[last];
   R[last] = temp;
   return i+1;
}
void quickSort(Residence *R,int first_index, int last_index){
   if (first_index < last_index){
      int q = partition(R,first_index,last_index);
      quickSort(R, first_index, q-1);
      quickSort(R, q+1, last_index);
   }
}
```

For finding the asymptotic upper bound on the running time for Quicksort, we should consider worst case for it. **Worst-case is splitting arrays every time to 0:n-1 sized subarrays.** So, we can calculate upper bound with this recursive equation:

$T(n) = T(0) + T(n-1) + O(n)$ $O(n)$ absorbs $T(0) = 1$. So equation becomes to

$T(n) = T(n-1) + O(n)$

$T(n-1) = T(n-2) + O(n-1)$

$T(n-2) = T(n-3) + O(n-2)$

.

.

.

$T(2) = T(1) + O(2)$

When we sum all of these equations, it returns to the equation below:

$$T(n) = T(1) + \sum_{i=2}^{n} O(i) = O(n*(n+1)/2-1) = O(n^2)$$

As a result, asymptotic upper bound on the running time for Quicksort is $O(n^2)$.

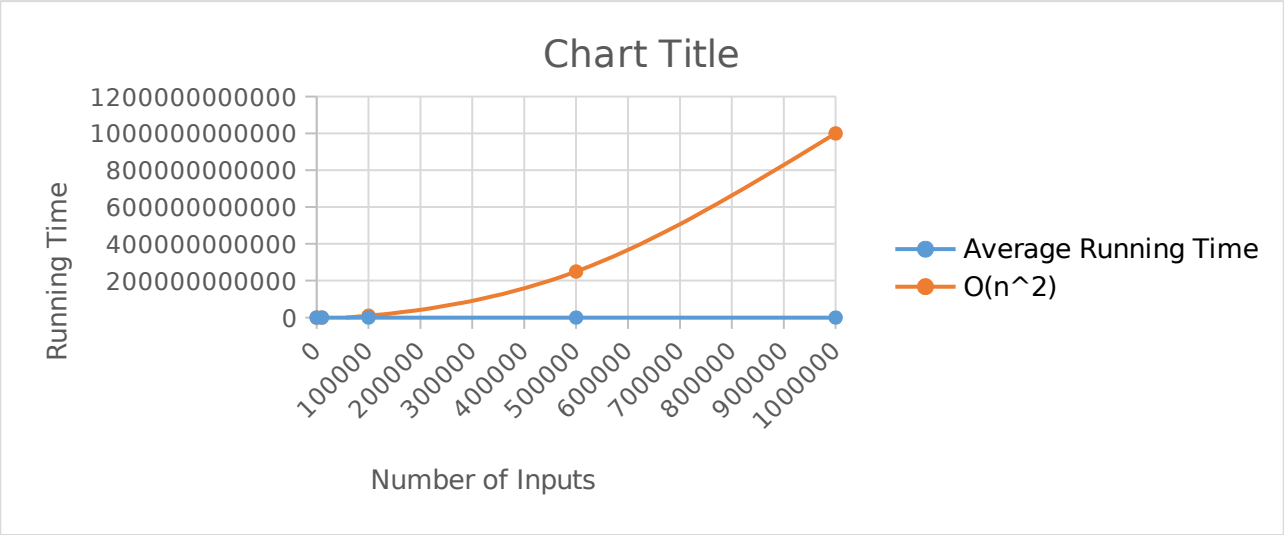
Bayram Barış Sarı
150130123

Testing

We're asked to test our code for different size of arrays and calculate average running times. The table which contains measured running times is:

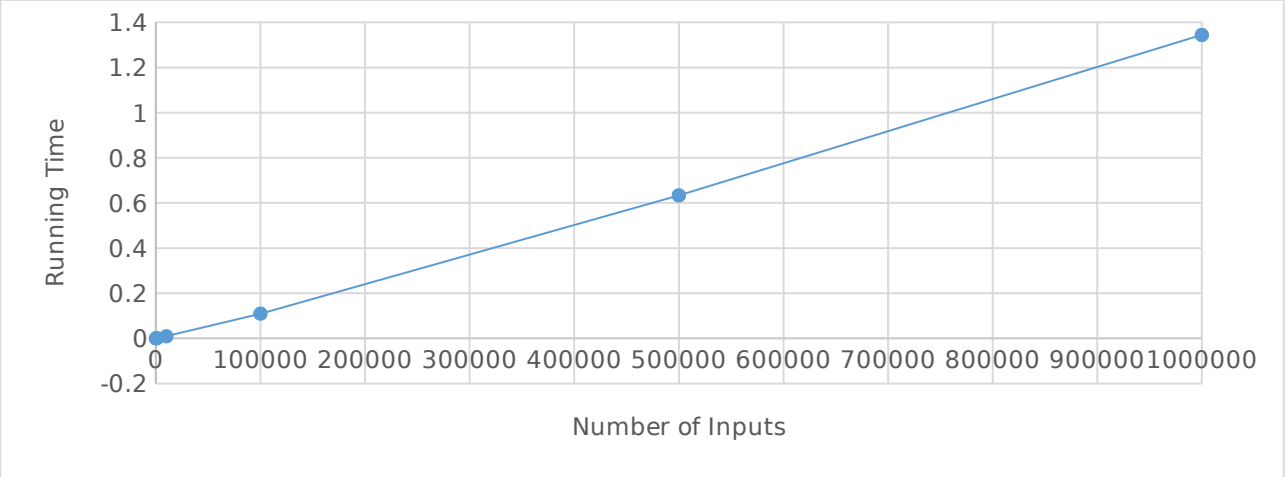
N=10	N=100	N=1000	N=10000	N=100000	N=500000	N=1000000	
5e-06 s	3.5e-05 s	0.000551 s	0.008749 s	0.11142 s	0.62968 s	1.34036 s	
5e-06 s	6e-05 s	0.000627 s	0.008448 s	0.110808 s	0.616328 s	1.27995 s	
4e-06 s	5.4e-05 s	0.000548 s	0.00897 s	0.107126 s	0.643067 s	1.34887 s	
4e-06 s	4e-05 s	0.000656 s	0.012251 s	0.112127 s	0.649626 s	1.32986 s	
5e-06 s	4.5e-05 s	0.00054 s	0.009335 s	0.111306 s	0.623603 s	1.30351 s	
7e-06 s	5e-05 s	0.000696 s	0.009195 s	0.106286 s	0.631918 s	1.39136 s	
6e-06 s	4.1e-05 s	0.00068 s	0.008689 s	0.10962 s	0.635592 s	1.38084 s	
4e-06 s	4.7e-05 s	0.00065 s	0.008188 s	0.110874 s	0.630514 s	1.46867 s	
3e-06 s	4.6e-05 s	0.000642 s	0.008875 s	0.108794 s	0.616693 s	1.3269 s	
5e-06 s	3.9e-05 s	0.000627 s	0.008241 s	0.105706 s	0.656643 s	1.27093 s	
4,8*10^{-6}s	**4,57*10^{-5}s**	**6,217* 10^{-4}s**	**0,0090941 s**	**0,1094067 s**	**0,6333664 s**	**1,344125 s**	**Average**

Note: For taking different array groups, I skipped lines arbitrarily from the beginning and then parsed N lines. After that, I sorted the data and measured running time during the Quicksort process. When we translate these measured values to a graph, it looks like below:



Pay attention that, average running time is always beneath of worst case O(n^2).
When we look at closer to Average running time:

Bayram Barış Sarı
150130123



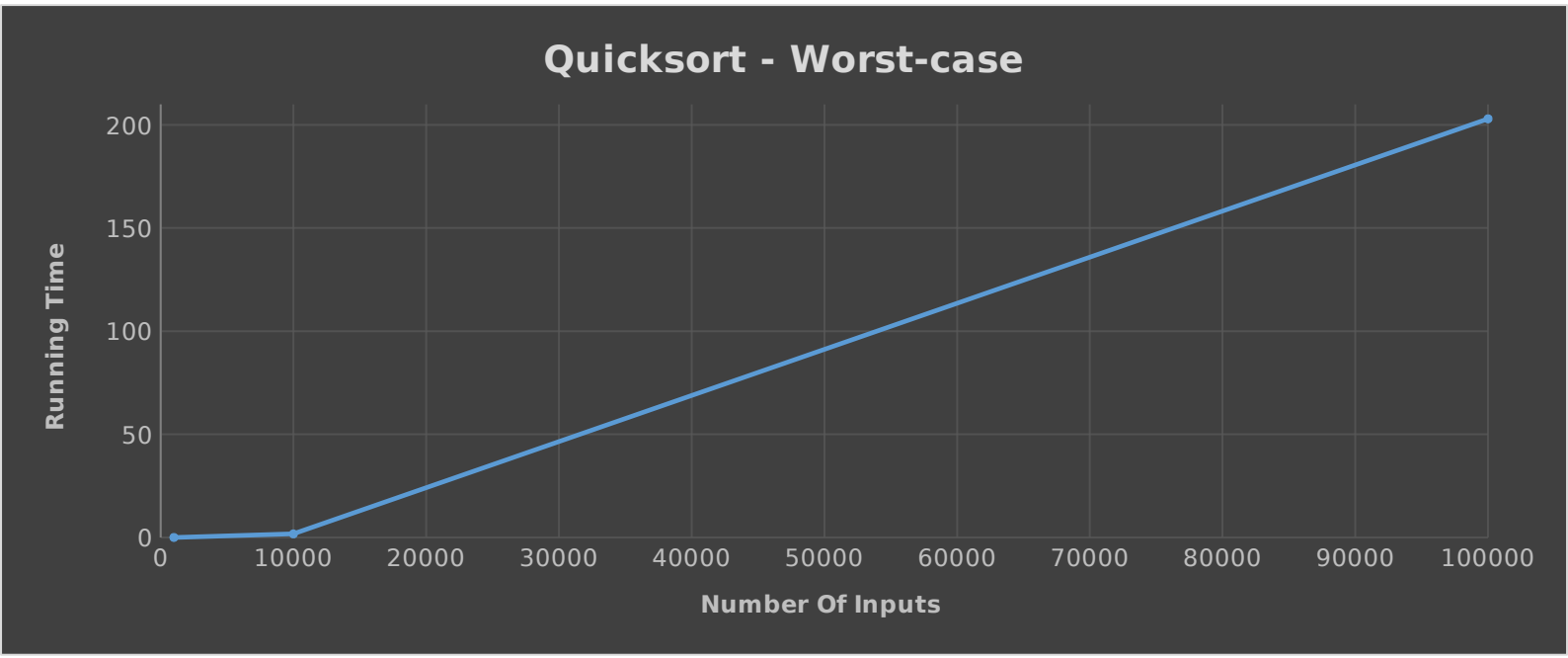
For the second part of testing, I created new data sets for simulating worst-case for Quicksort. **Every time, partition split 0:n-1 size of subarrays.** Here is the measured running time for this case:

N=10	**N=100**	**N=1000**	**N=10000**	**N=100000**	**N=500 K**	**N=1 M**	
6e-06 s	5.1e-05 s	0.005256 s	0.463531 s	63.2568 s			
6e-06 s	6e-05 s	0.00818 s	0.385319 s	100.464 s			
7e-06 s	6.6e-05 s	0.001477 s	0.980894 s	292.312 s			
8e-06 s	4.7e-05 s	0.008431 s	2.95864 s	290.773 s			
10e-06 s	10.3e-05 s	0.035456 s	2.9792 s	221.948 s			
9e-06 s	62.5e-05 s	0.036195 s	2.27637 s	258.004 s			
12e-06 s	39.5e-05 s	0.024403 s	2.69917 s	137.782 s			
6e-06 s	25.5e-05 s	0.02909 s	1.18646 s	185.237 s			
7e-06 s	34.3e-05 s	0.01241 s	1.81208 s	198.188 s			
9e-06 s	15.9e-05 s	0.020548 s	0.590088 s	281.859 s			
$8*10^{-6}$s	**$21.04*10^{-5}$ s**	**0,0157043 s**	**1,7031752 s**	**202,98238 s**			**Average**

I couldn’t find any results for 500K and 1M for the worst-case because I faced with the error of segmentation fault. I tried 2 times for 500K and once for 1M, but after 4-5 hours, program stopped executing. That’s why last 2 columns of the table aren’t filled.

Bayram Barış Sarı
150130123

The graph below is plotted according to the obtained data:



Although the runtime for inputs of 10-100-1000 is only about 10 times higher, execution time for sorting rises almost 1000 times for 10K and 100K inputs. **For handling worst-case situation, we should select random pivot rather than first or last element**. It's called randomized Quicksort and it gives better result than normal Quicksort algorithm.

Stability

Quicksort is not a stable sort algorithm. We can prove that by illustrating how it works for sorting 10 inputs.
I used this data for showing that Quicksort is not stable;

```
0,5,9,male,99927,8600000US99927
24,5,9,male,62356,8600000US62356
1,21,21,male,786,8600000US00786
24,40,44,male,62355,8600000US62355
1,0,0,female,906,8600000US00906
0,25,29,female,99927,8600000US99927
24,30,34,male,62355,8600000US62355
2,60,61,female,12738,8600000US12738
2,22,24,male,12738,8600000US12738
24,40,44,female,62358,8600000US62358
3,70,74,female,49863,8600000US49863
```

Bayram Barış Sarı
150130123

I printed the array in each iteration and this is the result:
Note: This table doesn't have gender and geo_id values.

Note-2 : The last element, 3,70,74,49863 , is selected as a pivot.

1,21,21,786	**1,21,21,786**	**1,21,21,786**	**1,21,21,786**	**1,21,21,786**	**1,21,21,786**
24,5,9,62356	**1,0,0,906**	**1,0,0,906**	**1,0,0,906**	**1,0,0,906**	**1,0,0,906**
24,40,44,62355	**24,40,44,62355**	**0,25,29,99927**	**0,25,29,99927**	**0,25,29,99927**	**0,25,29,99927**
1,0,0,906	**24,5,9,62356**	**24,5,9,62356**	**2,60,61,12738**	**2,60,61,12738**	**2,60,61,12738**
0,25,29,99927	**0,25,29,99927**	**24,40,44,62355**	**24,40,44,62355**	**2,22,24,12738**	**2,22,24,12738**
24,30,34,62355	**24,30,34,62355**	**24,30,34,62355**	**24,30,34,62355**	**24,30,34,62355**	**3,70,74,49863**
2,60,61,12738	**2,60,61,12738**	**2,60,61,12738**	**24,5,9,62356**	**24,5,9,62356**	**24,5,9,62356**
2,22,24,12738	**2,22,24,12738**	**2,22,24,12738**	**2,22,24,12738**	**24,40,44,62355**	**24,40,44,62355**
24,40,44,62358	**24,40,44,62358**	**24,40,44,62358**	**24,40,44,62358**	**24,40,44,62358**	**24,40,44,62358**
3,70,74,49863	**3,70,74,49863**	**3,70,74,49863**	**3,70,74,49863**	**3,70,74,49863**	**24,30,34,62355**

The result of residences with equal number of population, 24, proves the Quicksort isn't stable. Because after the sort, they are rearranged in each other.