BLG 335E – Analysis of Algorithms I

Homework 1

QuickSort

150170092 : Barış İncesu

10 DECEMBER 2020

Part 2. Report

2.a.Worst Case:

The worst case behaviour for quicksort occurs when the partitioning routine produces one region with *n - 1* elements and one with only *1* element. Let us assume that this unbalanced partitioning arises at every step of the algorithm.

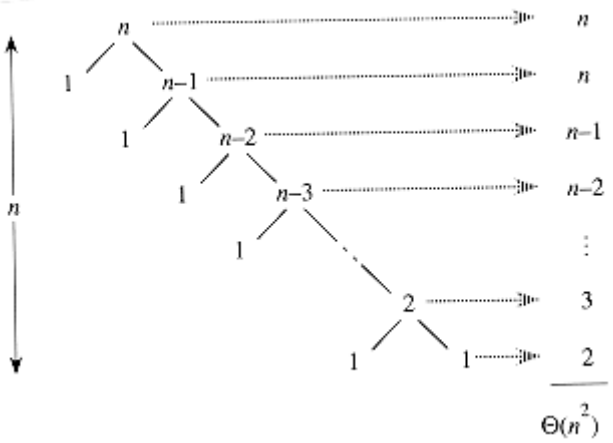
Since partitioning costs $\Theta(n)$ time and $T(1) = \Theta(1)$, the recurrence for the running time is

$$\boldsymbol{T(n) = T(n - 1) + \Theta(n).}$$

To evaluate this recurrence, we observe that $T(1) = \Theta(1)$ and then iterate:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) \ . \end{aligned}$$

a recursion tree for this worst-case execution of quicksort.



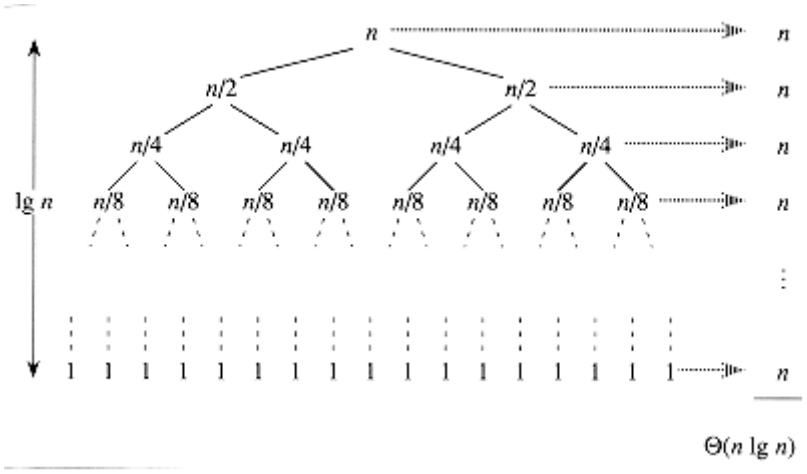
Thus, if the partitioning is maximally unbalanced at every recursive step of the algorithm, the running time is $\boldsymbol{\Theta(n^2)}$.

a.Best Case

If the partitioning procedure produces two regions of size *n/2*, quicksort runs much faster. The recurrence is then

$$\boldsymbol{T(n) = 2T(n/2) + \Theta(n),}$$

which by case 2 of the master theorem has solution $T(n) = \Theta(n \log n)$. Thus, this best-case partitioning produces a much faster algorithm.



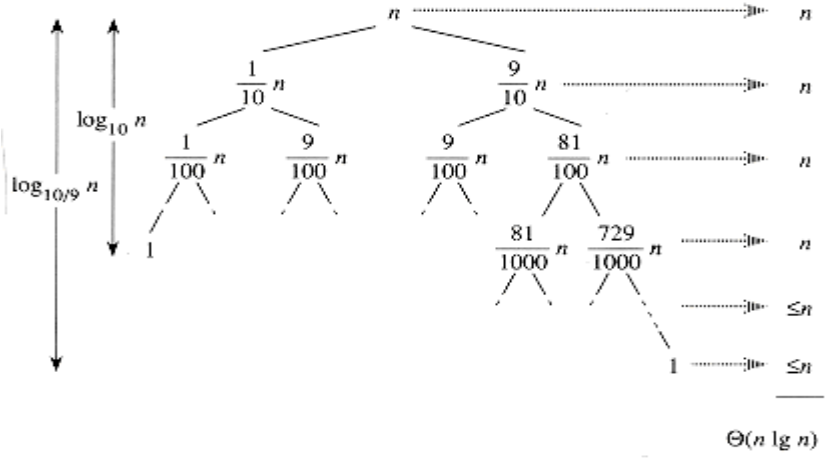
a.Average Case

The average case running time of quicksort is much closer to the best case than to the worst case. The key to understanding why this might be true is to understand how the balance of the partitioning is reflected in the recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

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

on the running time of quicksort, where we have replaced $\Theta(n)$ by n for convenience. Notice that every level of the tree has cost n, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most n. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. **The total cost of quicksort is therefore $\Theta(n \lg n)$**. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $\Theta(n \lg n)$ time asymptotically the same as if the split were right down the middle. In fact, even a 99-to-1 split yields an $O(n \lg n)$ running time. The reason is that any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $\Theta(n \lg n)$ whenever the split has constant proportionality.



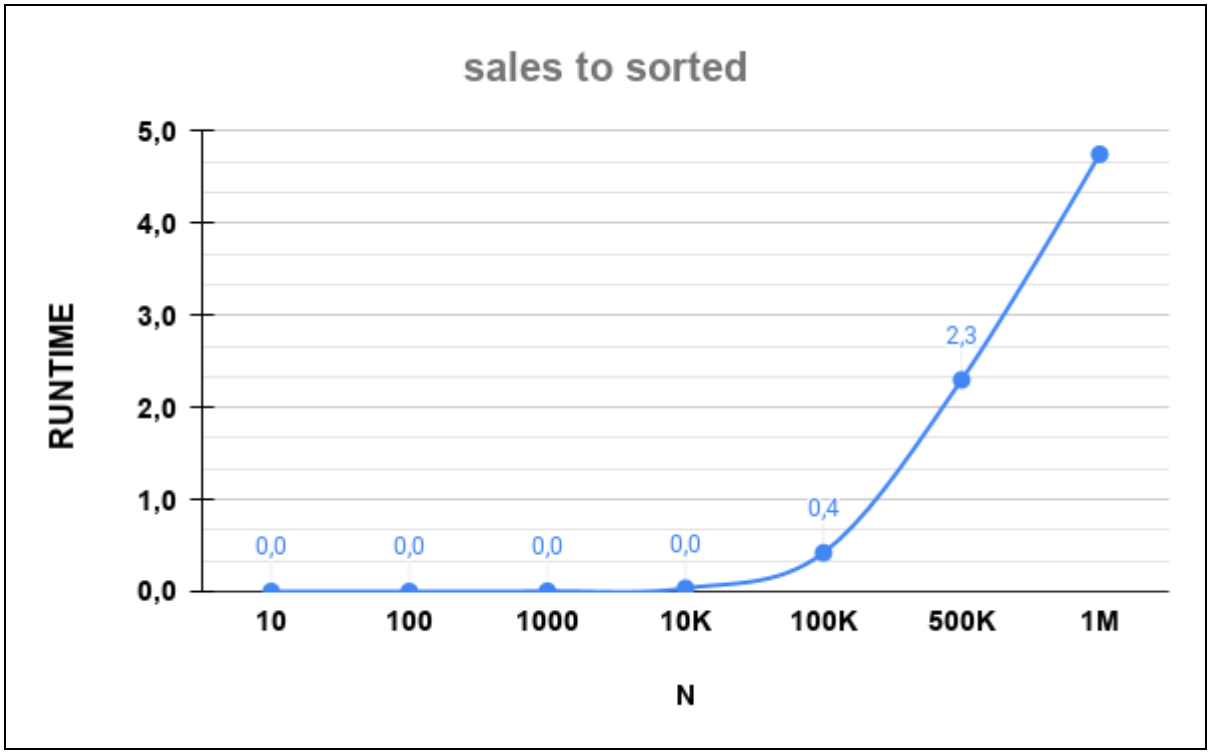
b.1.) This application did not give the same result as we did on implementation (Part 1). The reason for this is the wrong positioning of the pivot in the first partition operation. In case the pivot value is selected from the end in the *sorted_by_profit.txt* file, it is placed in front of the order that has the same letter and has more profits than itself. This also shows that quicksort is not a stable algorithm. The simulation is shown below on a small dataset.

```
vector<order> v = {[Algeria,100], [Turkey,90], [Frane,80], [Zimbabwe,70],
[Belgium,60], [Djibouti,50], [France,40]}

low = 0, high = 6, pivot = v[6] = France

Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since v[j] < pivot, do i++ and swap(v[i], v[j])
i = 0
v    =    {[Algeria,100],    [Turkey,90],    [Frane,80],    [Zimbabwe,70],
[Djibouti,60], [Belgium,50], [France,40]}
// No change as v[0] and v[0] are same

j = 1 : Since v[j] > pivot, do nothing // No change in i and v[j]

j = 2 : Since v[j] = pivot, do nothing // No change in i and v[j]

j = 3 : Since v[j] > pivot, do nothing // No change in i and v[j]

j = 4 : Since v[j] < pivot, do i++ and swap(v[i], v[j])
i = 1
v = {[Algeria,100], [Belgium,60], [Frane,80], [Zimbabwe,70], [Turkey,90],
[Djibouti,50], [France,40]}
// We swap [Turkey,90] and [Belgium,60]

j = 5 : Since v[j] < pivot, do i++ and swap(v[i], v[j])
i = 2
v    =    {[Algeria,100],    [Belgium,60],    [Djibouti,50],    [Zimbabwe,70],
[Turkey,90], [Frane,80], [France,40]}
// We swap [France,80] and [Djibouti,50]

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
v[i+1] and v[high] (or pivot)
v    =    {[Algeria,100],    [Belgium,60],    [Djibouti,50],    [France,40]},
[Turkey,90], [Frane,80], [Zimbabwe,70]}
// [Zimbabwe,70] and [France,40] swapped

Now [France,40] is not at its correct place. Algeria, Belgium and
Djibouti, Turkey and Zimbabwe on correct position but on the other hand
France,80 should be on in front of France,40.
```

b.2.) Selection Sort, Insertion Sort and Bubble Sort are **stable sort algorithms** can execute this operation.

2.c.)

	10	100	1000	10K	100K	500K	1M
1	0,000013	0,000251	0,003578	0,032312	0,412580	2,26559	4,70326
2	0,000015	0,000259	0,003861	0,030687	0,416043	2,21048	4,82042
3	0,000013	0,000259	0,003457	0,031430	0,423875	2,36501	4,84636
4	0,000013	0,000241	0,003996	0,043991	0,415963	2,28220	4,99166
5	0,000014	0,000241	0,004276	0,035259	0,418630	2,30962	4,73488
6	0,000014	0,000240	0,003691	0,035259	0,458535	2,31816	4,73148
7	0,000018	0,000322	0,004586	0,034751	0,412064	2,33858	4,74954
8	0,000014	0,000311	0,003162	0,037558	0,435361	2,28798	4,80690
9	0,000010	0,000311	0,003807	0,035323	0,410187	2,30283	4,54327
10	0,000012	0,000250	0,004947	0,035323	0,408246	2,31141	4,57572
AVG	**0,000014**	**0,000269**	**0,003936**	**0,035189**	**0,421148**	**2,299186**	**4,750349**



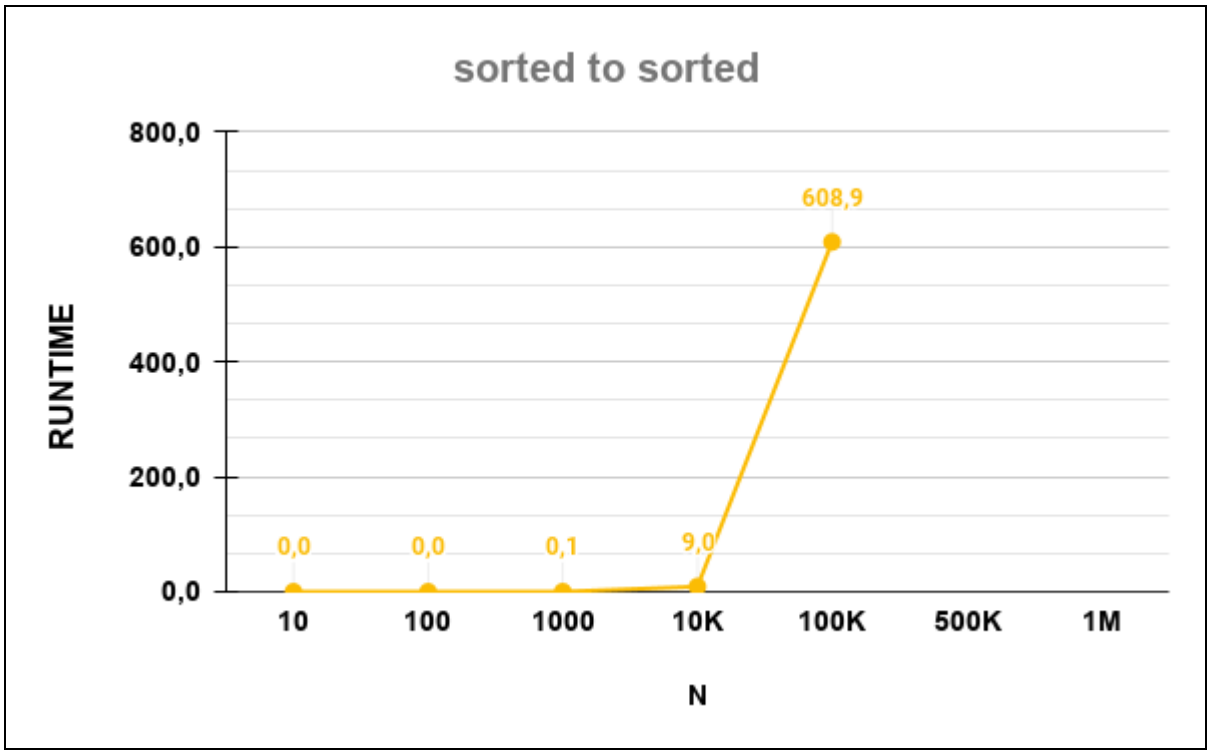
When looking at this graph by considering the $\Theta(n \log n)$ and $\Theta(n^2)$ values found for the QuickSort in **Part a**, it can be said that the relationship between runtime (s) and N is steepened at a slower rate than the increase in the $\Theta(n^2)$ graph. From here, it can be deduced that this process is one of the **average cases**.

2.d.1.)

Unlike the graph in **Part c**, it is seen that the graph is getting upright very quickly in this part. The fact that the situation in the form of **pivot-partition** seen from part a is too high is the main factor in the increase in duration. This situation can be called the **worst case** in **Part a**. Especially when the number of elements to be sorted increases, it is seen in the graphics that this time increases significantly.

In fact, the results of the trials with 1M element did not come out, although it was expected for half an hour. These values are shown in the table as **n/a** and are not included in the graph.

	10	100	1000	10K	100K	500K	1M
1	0,000027	0,001702	0,106351	8,42194	518,45	-	-
2	0,000023	0,001850	0,106642	8,63027	617,23	-	-
3	0,000024	0,001702	0,103315	9,10012	713,22	-	-
4	0,000014	0,001913	0,097508	9,06306	523,89	-	-
5	0,000022	0,001848	0,105994	9,19950	606,24	-	-
6	0,000025	0,001715	0,103925	9,06221	687,34	-	-
7	0,000024	0,001969	0,105557	8,93706	627,64	-	-
8	0,000023	0,001813	0,103337	9,22483	597,73	-	-
9	0,000022	0,001689	0,097129	9,09806	623,43	-	-
10	0,000022	0,001914	0,105909	9,10619	573,45	-	
AVG	**0,000023**	**0,001812**	**0,103567**	**8,984324**	**608,86200**	**n/a**	**n/a**



d.2.)

As can be seen in the application, **giving the data sequentially from** large to small or small to large has left QuickSort in a very difficult situation. Likewise, increasing the number of **same values** in the data will increase the execution time.

d.3.)

Not choosing the pivot from the beginning or the end to get rid of this worst case, choosing a random pivot will improve the situation a bit.