Title: Algorithm Efficiency and Sorting

Author: Işık Özsoy

ID: 21703160

Section: 01

Assignment: 1

Description: This file contains the answers to question 1, 2 and 3

<u>Question 1. a.</u>

Show that (n) = $20n^4$ + $20n^2$ + 5 is $(n^5$)

<u>Solution 1. a.</u>

We need to find two positive constants: c and n_0 such that:

$0 \le 20n^4 + 20n^2 + 5 \le cn^5$ for all $n \ge n_0$

$0 \le 20 + 20 / n^2 + 5 / n^5 \le c$ for all $n \ge n_0$

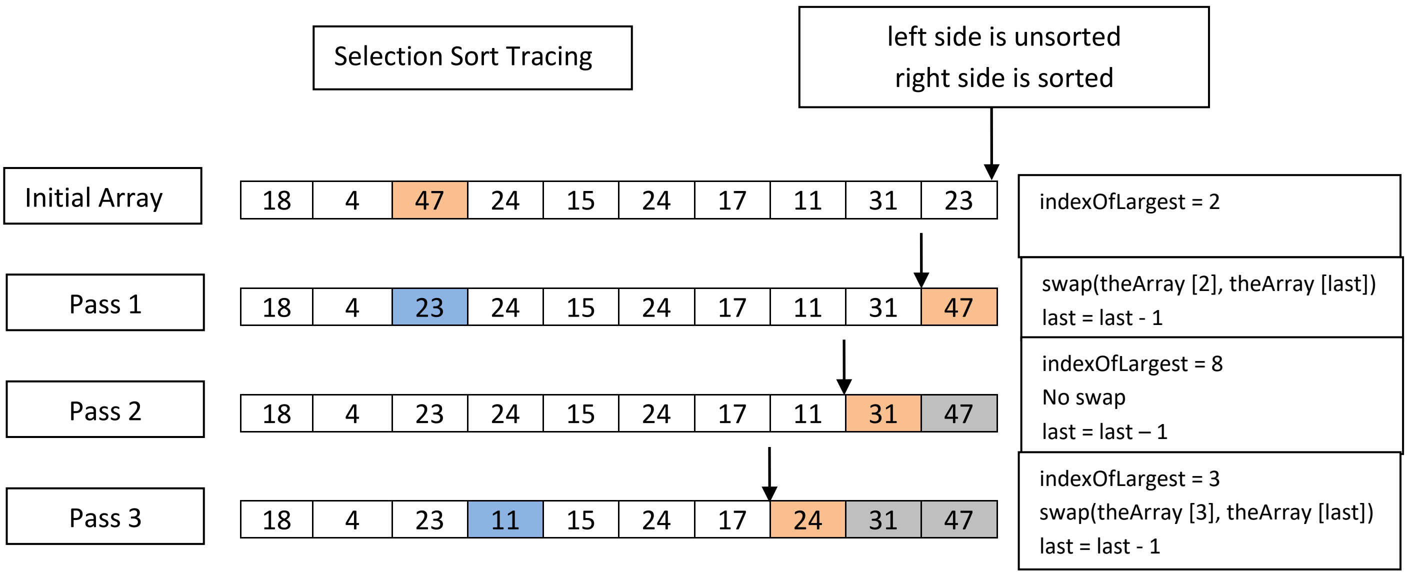
Choose $c = 45$ and $n_0 = 1$

$20n^4 + 20n^2 + 5 \le 45n^5$ for all $n \ge 1$

<u>Question 1. b.</u>

Trace the following sorting algorithms to sort the array [18, 4, 47, 24, 15, 24, 17, 11, 31, 23]

<u>Solution 1. b.</u>



Pass 4	18	4	23	11	15	17	24	24	31	47	indexOfLargest = 5 swap(theArray [5], theArray [last]) last = last - 1
Pass 5	18	4	17	11	15	23	24	24	31	47	indexOfLargest = 2 swap(theArray [2], theArray [last]) last = last - 1
Pass 6	15	4	17	11	18	23	24	24	31	47	indexOfLargest = 0 swap(theArray [0], theArray [last]) last = last - 1
Pass 7	15	4	11	17	18	23	24	24	31	47	indexOfLargest = 2 swap(theArray [2], theArray [last]) last = last - 1
Pass 8	11	4	15	17	18	23	24	24	31	47	indexOfLargest = 0 swap(theArray [0], theArray [last]) last = last - 1
Pass 9	4	11	15	17	18	23	24	24	31	47	indexOfLargest = 0 swap(theArray [0], theArray [last]) last = last - 1

Bubble Sort Tracing

Pass 1

Initial Array:	18	4	47	24	15	24	17	11	31	23	swap (theArray [0], theArray [1]) index = index + 1
	4	18	47	24	15	24	17	11	31	23	No swap index = index + 1
	4	18	47	24	15	24	17	11	31	23	swap (theArray [2], theArray [3]) index = index + 1
	4	18	24	47	15	24	17	11	31	23	swap (theArray [3], theArray [4]) index = index + 1
	4	18	24	15	47	24	17	11	31	23	swap (theArray [4], theArray [5]) index = index + 1
	4	18	24	15	24	47	17	11	31	23	swap (theArray [5], theArray [6]) index = index + 1
	4	18	24	15	24	17	47	11	31	23	swap (theArray [6], theArray [7]) index = index + 1
	4	18	24	15	24	17	11	47	31	23	swap (theArray [7], theArray [8]) index = index + 1

4	18	24	15	24	17	11	31	47	23

swap (theArray [8], theArray [9])
index = index + 1

4	18	24	15	24	17	11	31	23	47

pass = pass + 1

Pass 2

4	18	24	15	24	17	11	31	23	47

No swap
index = index + 1

4	18	24	15	24	17	11	31	23	47

No swap
index = index + 1

4	18	24	15	24	17	11	31	23	47

swap (theArray [2], theArray [3])
index = index + 1

4	18	15	24	24	17	11	31	23	47

No swap
index = index + 1

4	18	15	24	24	17	11	31	23	47

swap (theArray [4], theArray [5])
index = index + 1

4	18	15	24	17	24	11	31	23	47

swap (theArray [5], theArray [6])
index = index + 1

4	18	15	24	17	11	24	31	23	47

No swap
index = index + 1

4	18	15	24	17	11	24	31	23	47

swap (theArray [7], theArray [8])
index = index + 1

4	18	15	24	17	11	24	23	31	47

pass = pass + 1

Pass 3

4	18	15	24	17	11	24	23	31	47

No swap
index = index + 1

4	18	15	24	17	11	24	23	31	47

swap (theArray [1], theArray [2])
index = index + 1

4	15	18	24	17	11	24	23	31	47

No swap
index = index + 1

4	15	18	24	17	11	24	23	31	47

swap (theArray [3], theArray [4])
index = index + 1

4	15	18	17	24	11	24	23	31	47	swap (theArray [4], theArray [5]) index = index + 1
4	15	18	17	11	24	24	23	31	47	No swap index = index + 1
4	15	18	17	11	24	24	23	31	47	swap (theArray [6], theArray [7]) index = index + 1
4	15	18	17	11	24	23	24	31	47	pass = pass + 1

Pass 4

4	15	18	17	11	24	23	24	31	47	No swap index = index + 1
4	15	18	17	11	24	23	24	31	47	No swap index = index + 1
4	15	18	17	11	24	23	24	31	47	swap (theArray [2], theArray [3]) index = index + 1
4	15	17	18	11	24	23	24	31	47	swap (theArray [3], theArray [4]) index = index + 1
4	15	17	11	18	24	23	24	31	47	No swap index = index + 1
4	15	17	11	18	24	23	24	31	47	swap (theArray [5], theArray [6]) index = index + 1
4	15	17	11	18	23	24	24	31	47	pass = pass + 1

Pass 5

4	15	17	11	18	23	24	24	31	47	No swap index = index + 1
4	15	17	11	18	23	24	24	31	47	No swap index = index + 1
4	15	17	11	18	23	24	24	31	47	swap (theArray [2], theArray [3]) index = index + 1
4	15	11	17	18	23	24	24	31	47	No swap index = index + 1

4	15	11	17	18	23	24	24	31	47	No swap index = index + 1
4	15	11	17	18	23	24	24	31	47	pass = pass + 1

Pass 6

4	15	11	17	18	23	24	24	31	47	No swap index = index + 1
4	15	11	17	18	23	24	24	31	47	swap (theArray [1], theArray [2]) index = index + 1
4	11	15	17	18	23	24	24	31	47	No swap index = index + 1
4	11	15	17	18	23	24	24	31	47	No swap index = index + 1
4	11	15	17	18	23	24	24	31	47	pass = pass + 1

Pass 7

4	11	15	17	18	23	24	24	31	47	No swap index = index + 1
4	11	15	17	18	23	24	24	31	47	No swap index = index + 1
4	11	15	17	18	23	24	24	31	47	No swap index = index + 1
4	11	15	17	18	23	24	24	31	47	No swap index = index + 1
4	11	15	17	18	23	24	24	31	47	sorted = true

Question 2b: Sort {9, 5, 8, 15, 16, 6, 3, 11, 18, 0, 14, 17, 2, 9, 11, 7} by using insertion, merge and quick sort and print the number of key comparisons and data movements.

Solution 2b:

isik.ozsoy@dijkstra:~/cs202/CS202HW01

```
login as: isik.ozsoy
isik.ozsoy@dijkstra.ug.bcc.bilkent.edu.tr's password:
Last login: Mon Mar  9 18:42:49 2020 from 10.201.182.31
[isik.ozsoy@dijkstra ~]$ cd cs202/CS202HW01/
[isik.ozsoy@dijkstra CS202HW01]$ ls
auxArrayFunctions.cpp  CS202HW01.cbp     main.cpp   partc.h
auxArrayFunctions.h    CS202HW01.depend  obj        sorting.cpp
bin                    CS202HW01.layout  partc.cpp  sorting.h
[isik.ozsoy@dijkstra CS202HW01]$ g++ sorting.cpp auxArrayFunctions.cpp partc.cpp
 main.cpp -o hw1
[isik.ozsoy@dijkstra CS202HW01]$ ./hw1
9       5       8       15      16      6       3       11      18      0       1
4       17      2       9       11      7
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18
insertion sort: num of key comp: 74   num of movement: 89
9       5       8       15      16      6       3       11      18      0       1
4       17      2       9       11      7
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18
merge sort: num of key comp: 46   num of movement: 128
9       5       8       15      16      6       3       11      18      0       1
4       17      2       9       11      7
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18
quick sort: num of key comp: 47   num of movement: 105
[isik.ozsoy@dijkstra CS202HW01]$
```

Question 2c: Analyze the performance of the sorting algorithms that you would have implemented.

Solution 2c:

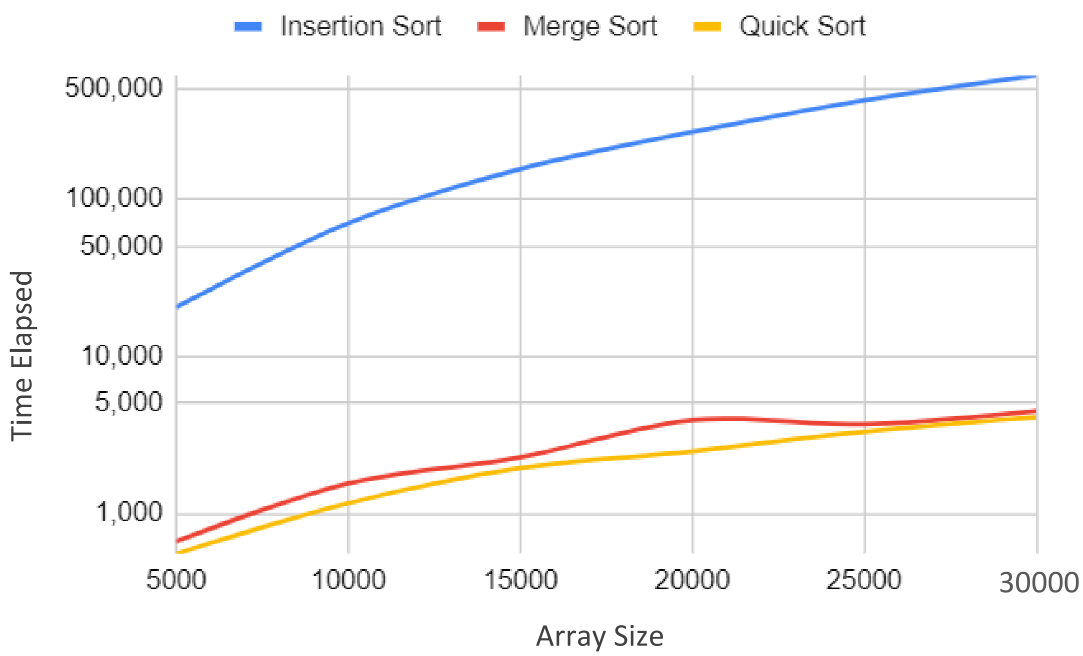
```
---------------- RANDOMLY GENERATED NUMBERS --------------------
Part c - Time analysis of Insertion Sort
Array Size      Time Elapsed            compCount       moveCount
5000            20.463 ms               6305767         6310766
10000           69.935 ms               25076768        25086767
15000           154.557 ms              56104874        56119873
20000           265.878 ms              100002179       100022178
25000           421.167 ms              156518748       156543747
30000           606.405 ms              224601102       224631101
-----------------------------------------------------
Part c - Time analysis of Merge Sort
Array Size      Time Elapsed            compCount       moveCount
5000            0.664 ms                55151           123616
10000           1.551 ms                120430          267232
15000           2.273 ms                189355          417232
20000           2.918 ms                261009          574464
25000           3.69 ms                 334318          734464
30000           4.478 ms                408560          894464
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size      Time Elapsed            compCount       moveCount
5000            0.553 ms                73805           109488
10000           1.164 ms                143686          226389
15000           1.943 ms                247488          375798
20000           2.484 ms                328523          498819
25000           3.31 ms                 429408          714066
30000           4.095 ms                557547          860601
```

```
---------------- ALREADY SORTED NUMBERS --------------------
Part c - Time analysis of Insertion Sort
Array Size      Time Elapsed            compCount       moveCount
5000            0.02 ms                 4999            9998
10000           0.041 ms                9999            19998
15000           0.063 ms                14999           29998
20000           0.083 ms                19999           39998
25000           0.102 ms                24999           49998
30000           0.122 ms                29999           59998
-----------------------------------------------------
Part c - Time analysis of Merge Sort
Array Size      Time Elapsed            compCount       moveCount
5000            0.366 ms                32004           123616
10000           0.776 ms                69008           267232
15000           1.204 ms                106364          417232
20000           1.641 ms                148016          574464
25000           2.171 ms                188476          734464
30000           2.679 ms                227728          894464
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size      Time Elapsed            compCount       moveCount
5000            23.112 ms               12497500        14997
10000           92.404 ms               49995000        29997
15000           208.271 ms              112492500       44997
20000           370.532 ms              199990000       59997
25000           579.635 ms              312487500       74997
30000           834.485 ms              449985000       89997
```

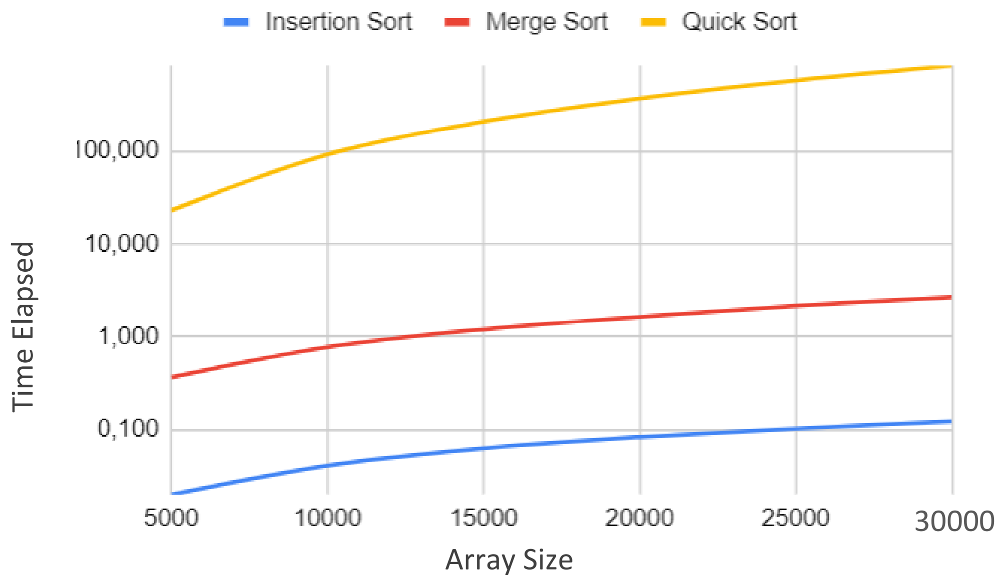
<u>Question 2d:</u> Prepare a single page report about the experimental results that you will have obtained in Question 2c.

<u>Solution 2d:</u>

<u>Randomly Generated Numbers</u>



Already Sorted Numbers



It can be observed from plots and screen shots that quick sort performs better than other sorting algorithms in the array consists of randomly generated numbers. When we compare it with merge sort, it can be seen that there isn't a huge difference between quick and merge sort as we expect. When we consider their time complexities we see that merge and quick sort work in O (n logn) in the best and average cases. Thus, we can say that our empirical results are close to the theoretical results.

On the other hand, it can be seen that the most ineffective one is the insertion sort which works in O(n), it is proportional to size in the best case and O(n^2) in the average and worst case. If we handle the first plot, we can observe that it increases like polinoms (n^2). In the insertion sort, there is a huge number of key comparison and data movement, and this situation increases the time complexity.

Furthermore, another important point about plots above is the change in the quick sort algorithm. When we have an array with randomly generated numbers, the algorithm time is the smallest one when we compare it with other sort algorithms. As mentioned before, it works in o (n logn) in the best case. However, if we have an array which is already sorted, the elapsed time increases accordingly, and becomes proportional to n^2.

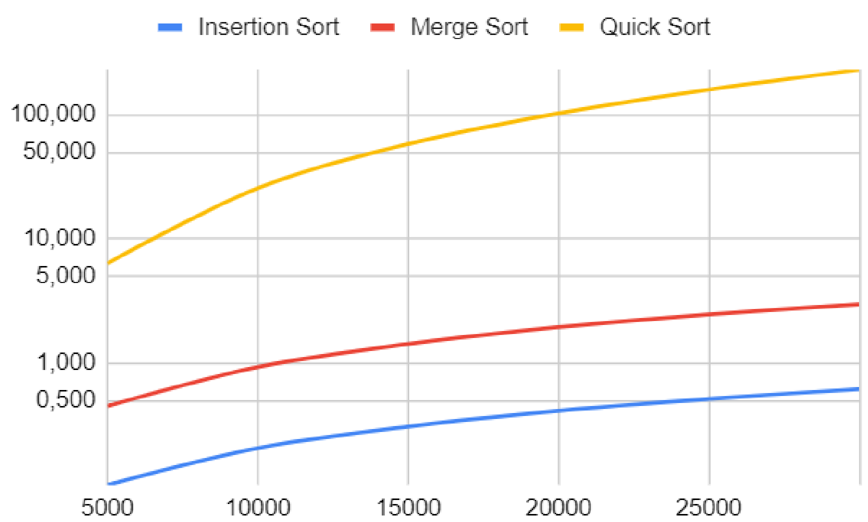
To sum up, according to the plots, it can be seen that our experimental results and theoretical results are close to each other. We also observed, they differ in terms of efficiency in different cases so it can be stated that their usage can be differ according to the purpose.

Question 3: **Prepare a single page report that discusses which algorithm among the three (i.e., the insertion sort or the merge sort or the quick sort) you should select for the most efficient solution of this problem. Discuss how the value of K (with respect to N) will affect your selection. You have to support your discussion with the experimental results.**

Solution 3:

```
-----------------NEARLY SORTED NUMBERS--------------------
Part c - Time analysis of Insertion Sort
Array Size      Time Elapsed            compCount       moveCount
5000            0.105 ms                16272           21271
10000           0.208 ms                32349           42348
15000           0.31 ms                 48420           63419
20000           0.416 ms                64906           84905
25000           0.518 ms                80443           105442
30000           0.622 ms                97115           127114
-----------------------------------------------------
Part c - Time analysis of Merge Sort
Array Size      Time Elapsed            compCount       moveCount
5000            0.453 ms                35565           123616
10000           0.933 ms                76031           267232
15000           1.433 ms                118136          417232
20000           1.956 ms                162094          574464
25000           2.478 ms                206215          734464
30000           2.99 ms                 251415          894464
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size      Time Elapsed            compCount       moveCount
5000            6.368 ms                3379409         29931
10000           25.687 ms               13758782        58722
15000           58.267 ms               31538623        87618
20000           102.719 ms              55209272        117744
25000           159.241 ms              86329603        146784
30000           230.569 ms              123473815       177909
```

Already Sorted Numbers



For the insertion sort, as the number of data moves (K) increases, algorithm time increases accordingly. From this observation, insertion sort performs better if the number of element to be sorted is small O (n). In the cases in which we have huge and unsorted data, insertion sort can be considered as inappropriate choice O (n^2).

For merge sort, as K increases, required algorithm time increases as well. However, unlikely insertion sort, it is useful since the time complexity of it is still not such big. On the other hand, in terms of required memory, it uses much more memory than others.

In quick sort, if we maximize K, as we can see in the first plot, we might still have an efficient algorithm. However, unlikely insertion sort, if the number of required data moves to sort is small, in other words, if the array is much more sorted, it performs worst than insertion sort, O (n^2) in the worst cases.

As a conclusion, it is obvious that insertion, merge and quick sorts are all can be efficient in appropriate cases, according to the given data to be sorted. We can consider quick sort as the best performing sorting algorithm if we know that the data is not nearly or completely sorted. Contrarily, if the data is nearly sorted, the time complexity of insertion sort provides better solution, however since it is proportional to size, algorithm time increases fast. Finally, merge sort can be considered as consistent algorithm, the time complexity of it doesn’t change too much so it can be considered as safe to use, if the usage of memory is not too important.