CS 202, Fall 2023
Homework 1 - Algorithm Analysis and Sorting
Deniz Polat
22103590
Section 001

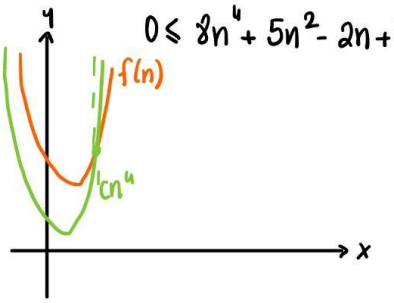
Question 1-a

Show that $f(n) = 8n^4 + 5n^2 - 2n + 4$ is $O(n^4)$ by specifying the appropriate c and n_0 values in big-O definition

Answer:

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

$0 \leq 8n^4 + 5n^2 - 2n + 4 \leq cn^4$ for all $n \geq n_0$



if we choose $c = 10$, $n_0 = 2$,

$0 \leq 8n^4 + 5n^2 - 2n + 4 \leq 10n^4$

for all $n \geq 2$.

Question 1-b

Trace the following sorting algorithms to sort:

[8, 33, 2, 10, 4, 1, 34, 7] in ascending order (küçükten büyüge)

i) insertion sort:

step 1:

	8	33	2	10	4	1	34	7
	0	1	2	3	4	5	6	7

j = 1 key = 33 33 > 8 sorted.

step 2

	2	8	33	10	4	1	34	7
	0	1	2	3	4	5	6	7

j = 2 key = 2. 2 < 33, shift 33
2 < 8, shift 8, insert 2.

step 3

	2	8	10	33	4	1	34	7
	0	1	2	3	4	5	6	7

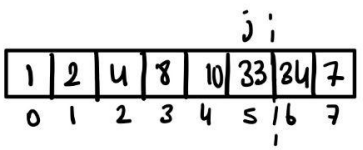
j = 3, key = 10 10 < 33, shift 33
insert 10.

step 4

	2	4	8	10	33	1	34	7
	0	1	2	3	4	5	6	7

j = 4, key = 4 4 < 33, shift 33
4 < 10, shift 10
4 < 8, shift 8, insert 4

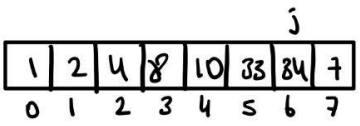
step 5



j = 5, key = 1

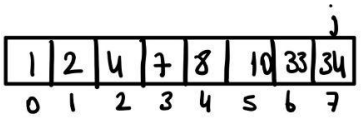
1 < 33, shift 33
1 < 10, 1 < 8, 1 < 4, 1 < 2,
shif and insert 1.

step 6



j = 6, key = 34, no shift or insert
SORTED.

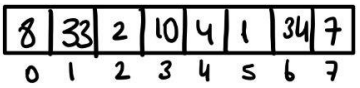
step 7

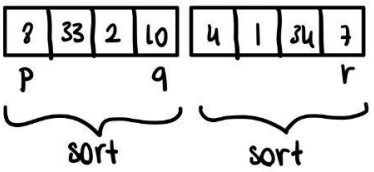


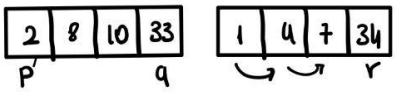
j = 7, key = 7, 7 < 34, 7 < 33, 7 < 10
7 < 8, shift and
insert 7.

sorted, done.

ii) merge sort:

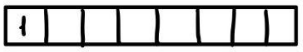




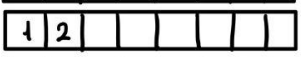


tempAccay.

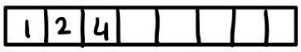
2 > 1 →



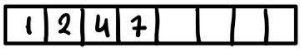
4 > 2 →



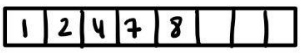
8 > 4 →



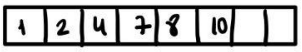
8 > 7 →



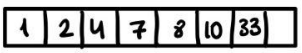
34 > 8 →



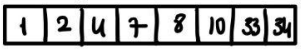
34 > 10 →



34 > 33 →



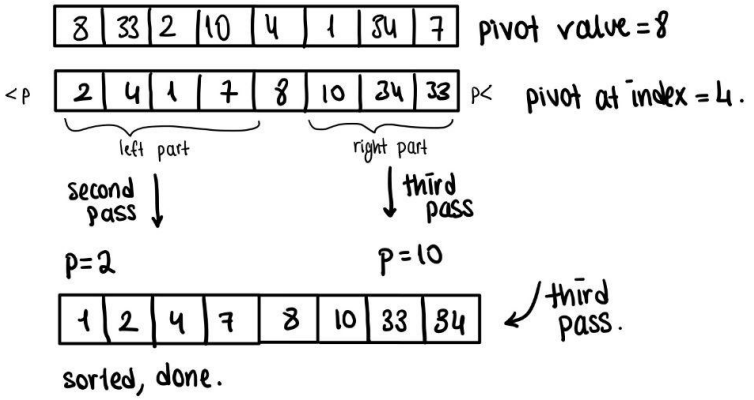
✓



sorted, done.

(iii) Quicksort:

pivot p → left subarray < p, right subarray > p. } ascending order.



Question 1-c:

$$T(n) = T(n/2) + n^2 \quad \text{where } T(1) = 1.$$

$$= T(n/4) + (n/2)^2 + n^2$$

$$= T(n/8) + (n/4)^2 + (n/2)^2 + n^2$$

Therefore:

$$T(n) = T(n/2^k) + (n/2^{k-1})^2 + \dots + (n/2^{k-k-1})^2 + (n/2^{k-k})^2$$

$$= \sum_{i=0}^{k} (n/2^{k-i})^2$$

Question 3:

After running the programs I wrote, I obtained the following data that shows the elapsed time of running each sorting algorithm for different array types and sizes.

Array	Elapsed Time (ms)					Number of Comparisons					Number of Data Moves				
	Insertion Sort	Selection Sort	Merge Sort	Quick Sort	Hybrid Sort	Insertion Sort	Selection Sort	Merge Sort	Quick Sort	Hybrid Sort	Insertion Sort	Selection Sort	Merge Sort	Quick Sort	Hybrid Sort
R1K	1.06	0.90	0.12	1.77	0.90	255998	499500	5044	499500	499500	256997	2997	19952	3996	2997
R5K	25.5	22.1	0.01	4.42	2.22	6705784	12497500	32004	12497500	12497500	6214280	14997	123616	19996	14997
R10K	102	88.5	1.52	177	88.5	37364647	49995000	69007	49995000	49995000	24892143	29997	267232	39996	29997
R20K	406	354	3.16	708	354	150312986	199990000	148015	199990000	199990000	100367982	59997	574464	79996	59997
A1K	0.02	0.10	0.13	1.77	0.10	199991236	499500	4824	499500	499500	62832	2997	19952	3996	2997
A5K	0.06	45.4	0.72	44.2	45.3	512064	12497500	30724	12497500	12497500	20560	14997	123616	19996	14997
A10K	0.20	260	1.50	177	259	125252205	49995000	6636	49995000	49995000	52701	29997	267232	39996	29997
A20K	0.41	225	3.29	708	225	50054970	199990000	142678	199990000	199990000	109966	59997	574464	79996	59997
D1K	1.90	0.93	0.13	1.77	0.91	200487177	499500	4839	499500	499500	558173	2997	19952	3996	2997
D5K	49.3	44.3	0.73	44.2	44.3	12982060	12497500	30610	12497500	12497500	12490556	14997	123616	19996	14997
D10K	200	266	1.50	177	266	62460253	49995000	66328	49995000	49995000	49987749	29997	267232	39996	29997
D20K	800	232	3.28	708	233	249915153	199990000	142795	199990000	199990000	199970149	59997	574464	79996	59997

The table shows that the elapsed time require to sort the arrays increase very fastly when the arrays size gets bigger. However, for the partially ascending ordered arrays, the elapsed time required to sort the arrays is much less than the other array types, since the algorithms are trying to sort the array in an ascending order. However, in many algorithms like insertion sort, selection sort, quick sort, and hybrid sort, the algorithms trace the array so many times, which can be seen from the number of comparisons in the table. This reduced the efficiency because even though It required less work to sort the array completely, the program compares every key value for so many times. The number of data moves, however, gave very similar results for each algorithm, except for the insertion sort for random, and partially descending arrays, which is expected. Since the partially ascending arrays require fewer data moves when compared to random and partially descending arrays, it is very expected to see such large numbers of data moves in insertion sort, because the insertion sort does many shifts in each pass. The number of shifts increases fast as the array size increases.

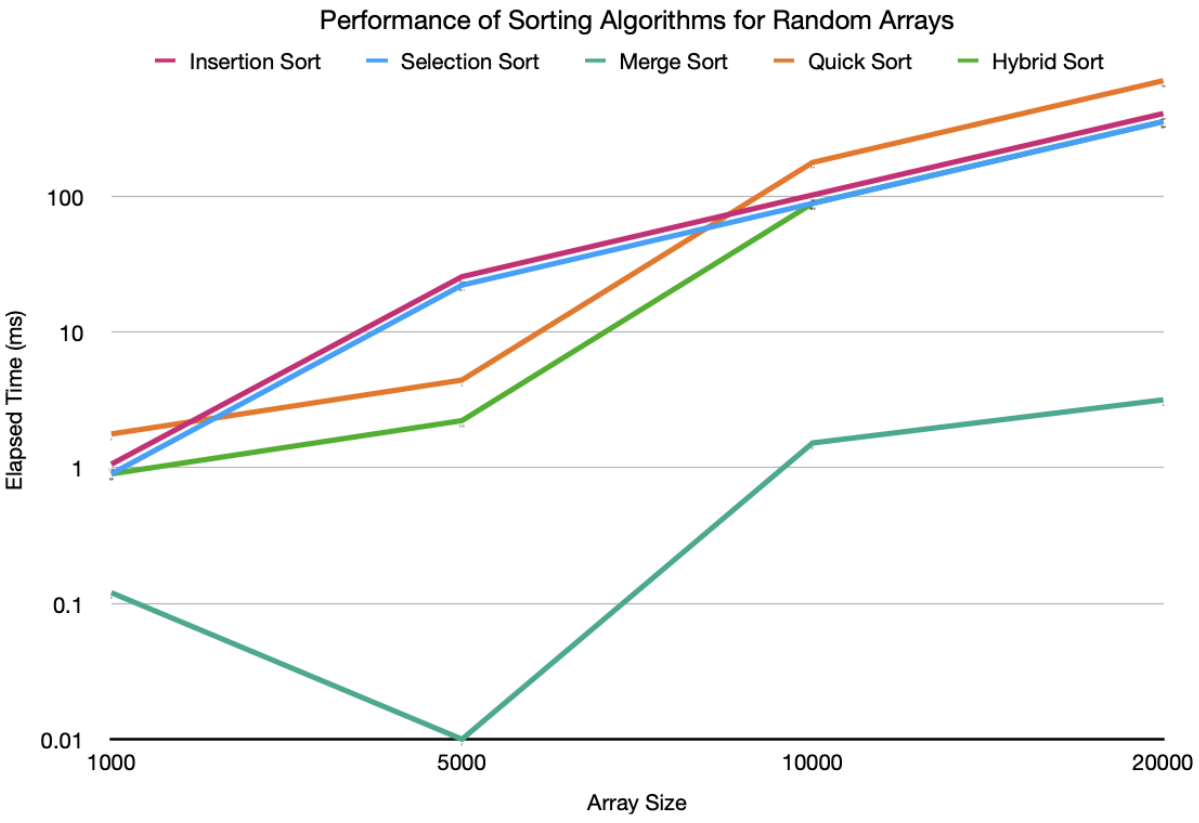
Here is the output from the terminal I filled the table with:

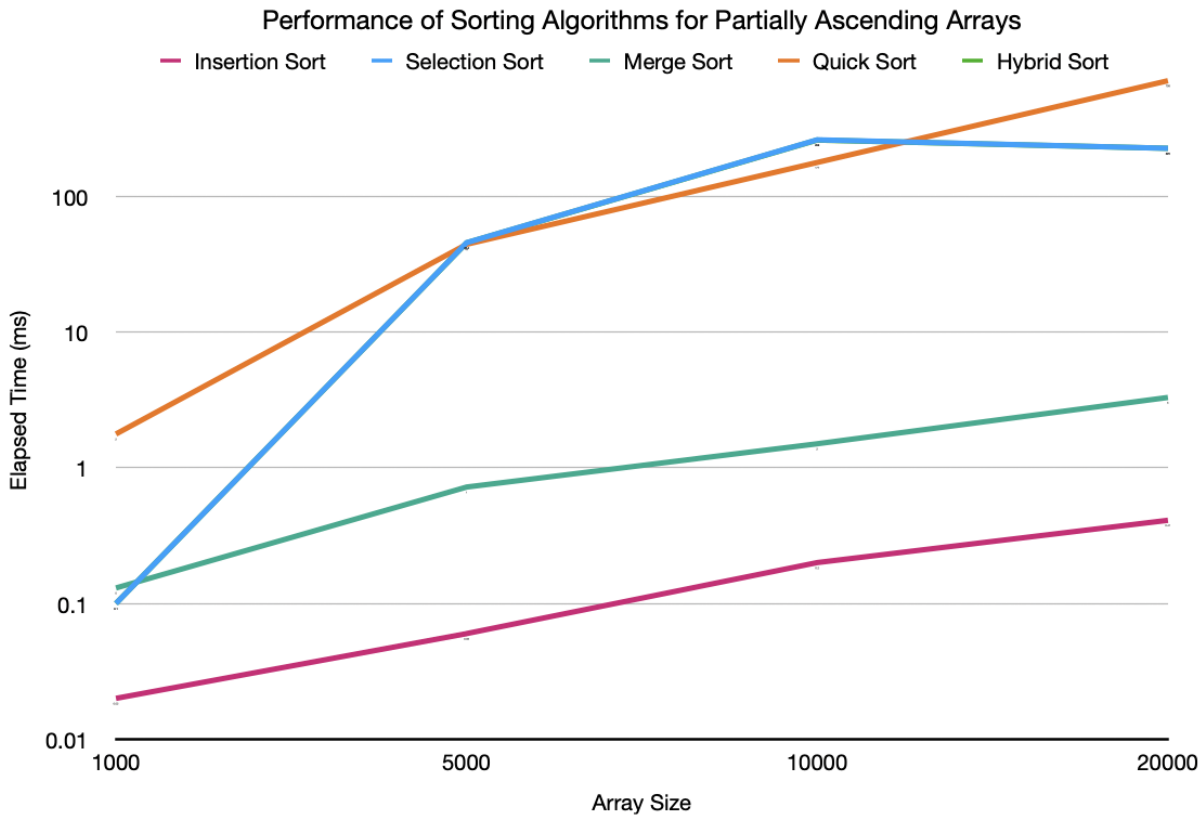
```
                        Part 2-b-1: Performance analysis of random integers array

                        Elapsed time                Comp. Count                 Move Count
Array Size: 1000
Insertion Sort          0.00106                     255998                      256997
Selection Sort          0.000895                    499500                      2997
Merge Sort              0.000124                    5044                        19952
Quick Sort              0.001768                    499500                      3996
Hybrid Sort             0.000896                    499500                      2997

Array Size: 5000
Insertion Sort          0.025511                    6705784                     6214280
Selection Sort          0.022173                    12497500                    14997
Merge Sort              0.000676                    32004                       123616
Quick Sort              0.044239                    12497500                    19996
Hybrid Sort             0.02217                     12497500                    14997

Array Size: 10000
Insertion Sort          0.101766                    37364647                    24892143
Selection Sort          0.088513                    49995000                    29997
Merge Sort              0.001525                    69007                       267232
Quick Sort              0.176764                    49995000                    39996
Hybrid Sort             0.088559                    49995000                    29997

Array Size: 20000
Insertion Sort          0.406936                    150312986                   100367982
Selection Sort          0.353938                    199990000                   59997
Merge Sort              0.00316                     148015                      574464
Quick Sort              0.707577                    199990000                   79996
Hybrid Sort             0.353772                    199990000                   59997
                        Part 2-b-2: Performance analysis of partially ascending integers array

                        Elapsed time                Comp. Count                 Move Count
Array Size: 1000
Insertion Sort          2e-05                       199991836                   62832
Selection Sort          0.000924                    499500                      2997
Merge Sort              0.000134                    4824                        19952
Quick Sort              0.001767                    499500                      3996
Hybrid Sort             0.000918                    499500                      2997

Array Size: 5000
Insertion Sort          9.6e-05                     512064                      20560
Selection Sort          0.045438                    12497500                    14997
Merge Sort              0.000723                    30724                       123616
Quick Sort              0.044168                    12497500                    19996
Hybrid Sort             0.045297                    12497500                    14997

Array Size: 10000
Insertion Sort          0.000197                    12525205                    52701
Selection Sort          0.259678                    49995000                    29997
Merge Sort              0.001502                    66336                       267232
Quick Sort              0.17675                     49995000                    39996
Hybrid Sort             0.259119                    49995000                    29997

Array Size: 20000
Insertion Sort          0.000411                    50054970                    109966
Selection Sort          1.22535                     199990000                   59997
Merge Sort              0.003285                    142678                      574464
Quick Sort              0.707597                    199990000                   79996
Hybrid Sort             1.22515                     199990000                   59997
```

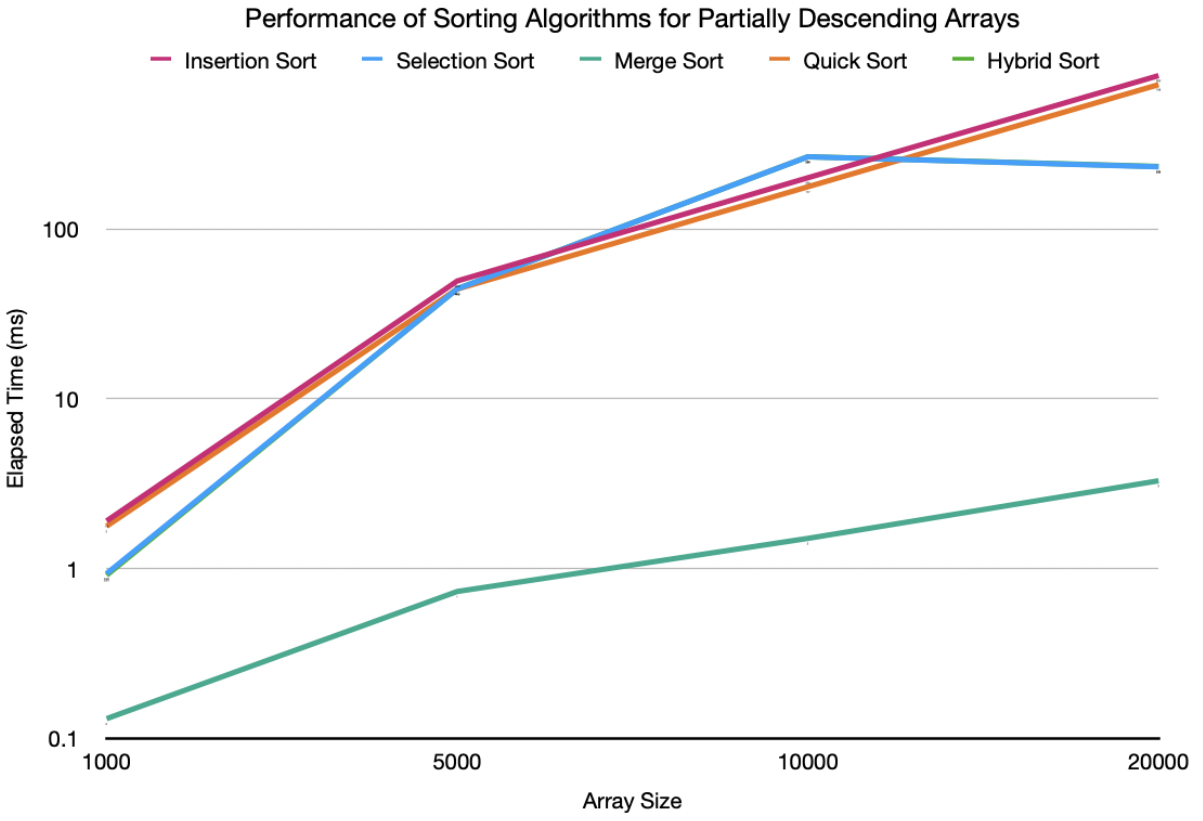
Part 2-b-2: Performance analysis of partially descending integers array

	Elapsed time	Comp. Count	Move Count
Array Size: 1000			
Insertion Sort	0.001908	200487177	558173
Selection Sort	0.000932	499500	2997
Merge Sort	0.000133	4839	19952
Quick Sort	0.001767	499500	3996
Hybrid Sort	0.000912	499500	2997
Array Size: 5000			
Insertion Sort	0.049307	12982060	12490556
Selection Sort	0.044341	12497500	14997
Merge Sort	0.000725	30610	123616
Quick Sort	0.044168	12497500	19996
Hybrid Sort	0.044312	12497500	14997
Array Size: 10000			
Insertion Sort	0.199806	62460253	49987749
Selection Sort	0.265827	49995000	29997
Merge Sort	0.001507	66328	267232
Quick Sort	0.176751	49995000	39996
Hybrid Sort	0.26605	49995000	29997
Array Size: 20000			
Insertion Sort	0.799541	249915153	199970149
Selection Sort	1.23282	199990000	59997
Merge Sort	0.003275	142795	574464
Quick Sort	0.707568	199990000	79996
Hybrid Sort	1.23304	199990000	59997

After preparing a table with my obtained values, I plotted three different graphs for each array type (random, partially ascending, and partially descending) that show the relation between the differing array sizes and the elapsed time.




Performance of Sorting Algorithms for Partially Ascending Arrays
Insertion Sort
Selection Sort
Merge Sort
Quick Sort
Hybrid Sort
100
10
1
0.1
0.01
Elapsed Time (ms)
1000
5000
10000
20000
Array Size


Performance of Sorting Algorithms for Partially Descending Arrays
Insertion Sort
Selection Sort
Merge Sort
Quick Sort
Hybrid Sort
100
10
1
0.1
Elapsed Time (ms)
1000
5000
10000
20000
Array Size

I chose my axis values to grow logarithmically rather than linear to observe and see the values in more detail, since the elapsed times for some algorithms are very small, and some are much larger.

In the first graph, it can be seen that the merge sort algorithm took much less time to sort the random arrays in ascending order compared to the other four algorithms. Insertion sort and selection sort gave very similar behavioral results, just like the relation between the quick sort and merge sort algorithms. The hybrid sort behaves as a merge sort algorithm until the partition size becomes less than or equal to 20, and then behaves as a bubble sort algorithm. This explains the large difference in the elapsed time for the two algorithms.

The random array represents the average case for each algorithm since all the values are positioned randomly in the array. The average case time complexity for insertion sort in big-O notation is $O(N^2)$, and for selection sort, it is the same. We can observe that they have the same behavior in graph 1. The average case time complexities for merge sort and quick sort are $O(N\log_2 N)$, again, since the hybrid sort behaves as the merge sort algorithm for a very long time, their behavior is almost the same which can be observed from the graph.

Hence, the most time-efficient algorithm to sort the random arrays is the merge sort.

In the second graph, the selection sort and hybrid sort algorithms behaved almost the same, and the selection sort's behavior is very hard to see since it overlaps with the hybrid sort. Quick sort, merge sort, and insertion sort algorithms showed a similar behavior, differing in the elapsed time values.

The second graph represents the best case for each algorithm since the arrays are already partially sorted as desired. For the first 4 algorithms, the time complexities for best cases are equal to the average case time complexities. However, even though the behaviors of quick sort and merge sort algorithms are similar again, insertion sort behaves like them rather than showing a behavior of a time complexity of $O(N^2)$. The insertion sort algorithm's best-case time complexity is the same as the bubble sort, and the graph shows that the hybrid sort behaved more like the bubble sort than the merge sort.

Hence, the most time-efficient algorithm to sort a partially ascending array is the insertion sort algorithm.

In the third graph, it can be seen that the elapsed time required to sort the arrays is much larger when compared to other array types, except the merge sort. Again, just like in a partially ascending array, the selection sort and hybrid sort algorithms behaved almost the same again. Quick sort and insertion sort algorithms have also very similar behaviors.

The partially descending array graph represents the worst-case for all algorithms since the array is sorted in the opposite direction. All four algorithms except the merge sort algorithm have a worst-case time complexity of $O(N^2)$. As can be observed from the graph, merge sort is very separated from other algorithms in terms of the elapsed time, which has a worst-case time complexity of $(ONlog_2N)$.

Hence, the most time-efficient algorithm to sort a partially descending array is the merge sort algorithm.