

Title: Sorting and Algorithm Efficiency

Author: Arman Engin Sucu

ID: 21801777

Section: 2

Assignment: 1

Description: Contains answers for questions 1, 2 and 3

QUESTION 1.a

Show that $f(n) = 5n^3 + 4n^2 + 10$ is $O(n^4)$

Condition:

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \times f(N)$ for all $N \geq n_0$
- $5n^3 + 4n^2 + 10 \leq c \times n^4$ for $n \geq n_0$
- Choosing $c = 5$ and $n_0 = 2 \implies 5n^3 + 4n^2 + 10 \leq 5 \times n^4$ for all $n \geq 2$

Insertion Sort

QUESTION 1.b

Sorted		Unsorted								
24	8	51	28	20	29	21	17	38	27	Copy 8, shift 24, insert 8
8	24	51	28	20	29	21	17	38	27	Copy 51, no shifting 28
8	24	51	28	20	29	21	17	38	27	Copy 28, shift 51, insert 28
8	24	28	51	20	29	21	17	38	27	Copy 20, shift 51, shift 28, shift 24, insert 20
8	20	24	28	51	29	21	17	38	27	Copy 29, shift 51, insert 29
8	20	24	28	29	51	21	17	38	27	Copy 21, shift 51, shift 29, shift 28, shift 24 insert 21
8	20	21	24	28	29	51	17	38	27	Copy 17, shift 51, shift 29, shift 28, shift 24, shift 21, shift 20, insert 17
8	17	20	21	24	28	29	51	38	27	Copy 38, shift 51, insert 38
8	17	20	21	24	28	29	38	51	27	Copy 27, shift 51, shift 38, shift 29, shift 28, insert 27
8	17	20	21	24	27	28	29	38	51	Sorted!!!

PROCEDURE

1. The sorted boundary starts from 0 and unsorted boundary is 1 to n.
2. Copy the first unsorted element.
3. Iterate through sorted boundary as long as sorted elements bigger than the copy.
4. Shift sorted elements to the right.
5. When the copy bigger than the sorted elements stop the iteration.
6. Insert the copy.
7. Iterate for other unsorted elements.
8. When the last index inserted to its place sorting finishes.

Sorted

Compared

Bubble Sort

[illegible]

Pass1, swap index 0 – index 1

Pass1, no swap

Pass1, swap index 2 – index 3

Pass1, swap index 3 – index 4

Pass1, swap index4 – index 5

Pass1, swap index 5 – index 6

Pass1, swap index 6 – index7

Pass1, swap index 7 – index 8

Pass1, swap index 8 – index 9

Pass2, no swap

Pass2, no swap

Pass2, swap index 2 – index 3

Pass2, no swap

Pass2, swap index 4 – index5

Pass2, swap index 5 – index 6

Pass2, no swap

Pass2, swap index 7 – index 8

Pass3, no swap

Pass3, swap index 1 – index 2

Pass3, no swap

Pass3, swap index 3 – index 4

Pass3, swap index 4 – index 5

Pass3, no swap

Pass3, swap index 6 – index 7

Pass4, no swap

Pass4, no swap

Pass4, swap index 2 – index 3

8	20	21	24	17	28	27	29	38	51	Pass4, swap index 3 – index 4
8	20	21	17	24	28	27	29	38	51	Pass4, no swap
8	20	21	17	24	28	27	29	38	51	Pass4, swap index 5 – index 6
8	20	21	17	24	27	28	29	38	51	Pass5, no swap
8	20	21	17	24	27	28	29	38	51	Pass5, no swap
8	20	21	17	24	27	28	29	38	51	Pass5, swap index 2 – index 3
8	20	17	21	24	27	28	29	38	51	Pass5, no swap
8	20	21	17	24	27	28	29	38	51	Pass5, no swap
8	20	21	17	24	27	28	29	38	51	Pass6, no swap
8	20	21	17	24	27	28	29	38	51	Pass6, no swap
8	20	21	17	24	27	28	29	38	51	Pass6, swap index 2 – index 3
8	20	17	21	24	27	28	29	38	51	Pass6, no swap
8	20	17	21	24	27	28	29	38	51	Pass7, no swap
8	20	17	21	24	27	28	29	38	51	Pass7, swap index 1– index 2
8	17	20	21	24	27	28	29	38	51	Sorted!!!

PROCEDURE


1. While starting the iteration initialize sorted as true
2. Comparison starts from index 0 and pass starts from 1.
3. If index greater than index +1 swap the elements.
4. Swap the elements until the index is size - 1 - pass.
5. Initialize sorted as false if swap happens
6. When the size – 1 – pass is reached increment pass one time.
7. Continue to iteration until sorted becomes true or pass becomes size.

Compared

Not
Compared

Sorted

QUESTION 2

 engin.sucu@dijkstra:~

```
For selection sort number of key comparisons are 135 and number of data moves are 81  
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }
```

```
For merge sort number of key comparisons are 46 and number of data moves are 128  
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }
```

```
For quick sort number of key comparisons are 45 and number of data moves are 114  
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }
```

Radix Sort

```
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }-----
```

Analysis of Selection Sort with random arrays.

Array Size	Elapsed time	compCount	moveCount
6000	1140 ms	18002999	73107
10000	3140 ms	50004999	122978
14000	6080 ms	98006999	158870
18000	10070 ms	162008999	241532
22000	14960 ms	242010999	273036
26000	20880 ms	338012999	317021
30000	27760 ms	450014999	397850

Analysis of Selection Sort with ascending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	1120 ms	18002999	45256
10000	3090 ms	50004999	73719
14000	6050 ms	98006999	99695
18000	9990 ms	162008999	144995
22000	14920 ms	242010999	155346
26000	20840 ms	338012999	213342
30000	27710 ms	450014999	221279

Analysis of Selection Sort with descending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	1120 ms	18002999	38629
10000	3090 ms	50004999	69403
14000	6040 ms	98006999	101980
18000	9990 ms	162008999	129172
22000	14910 ms	242010999	172368
26000	20820 ms	338012999	196697
30000	27710 ms	450014999	225232

Analysis of Merge Sort with random arrays.

Array Size	Elapsed time	compCount	moveCount
6000	80 ms	67842	151616
10000	120 ms	120446	267232
14000	200 ms	175398	387232
18000	300 ms	231933	510464

engin.sucu@dijkstra:~

Analysis of Merge Sort with random arrays.

Array Size	Elapsed time	compCount	moveCount
6000	80 ms	67842	151616
10000	120 ms	120446	267232
14000	200 ms	175398	387232
18000	300 ms	231933	510464
22000	410 ms	290133	638464
26000	550 ms	348846	766464
30000	660 ms	408667	894464

Analysis of Merge Sort with ascending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	70 ms	68189	151616
10000	120 ms	120863	267232
14000	200 ms	175724	387232
18000	300 ms	232360	510464
22000	410 ms	291167	638464
26000	550 ms	350393	766464
30000	670 ms	409233	894464

Analysis of Merge Sort with descending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	60 ms	67683	151616
10000	120 ms	120377	267232
14000	200 ms	175158	387232
18000	300 ms	231976	510464
22000	410 ms	289607	638464
26000	550 ms	348697	766464
30000	670 ms	408462	894464

Analysis of Quick Sort with random arrays.

Array Size	Elapsed time	compCount	moveCount
6000	30 ms	92864	165057
10000	40 ms	147524	260169
14000	60 ms	220271	332430
18000	80 ms	294636	503667
22000	110 ms	366818	637968
26000	120 ms	454206	762963
30000	150 ms	540040	862374

Analysis of Quick Sort with ascending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	20 ms	83228	153084
10000	50 ms	152590	252273

engin.sucu@dijkstra:~

18000	300 ms	231976	510464
22000	410 ms	289607	638464
26000	550 ms	348697	766464
30000	670 ms	408462	894464

Analysis of Quick Sort with random arrays.

Array Size	Elapsed time	compCount	moveCount
6000	30 ms	92864	165057
10000	40 ms	147524	260169
14000	60 ms	220271	332430
18000	80 ms	294636	503667
22000	110 ms	366818	637968
26000	120 ms	454206	762963
30000	150 ms	540040	862374

Analysis of Quick Sort with ascending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	20 ms	83228	153084
10000	50 ms	152590	252273
14000	60 ms	214985	380664
18000	80 ms	302529	525933
22000	110 ms	381831	656532
26000	130 ms	448762	815013
30000	150 ms	543236	960039

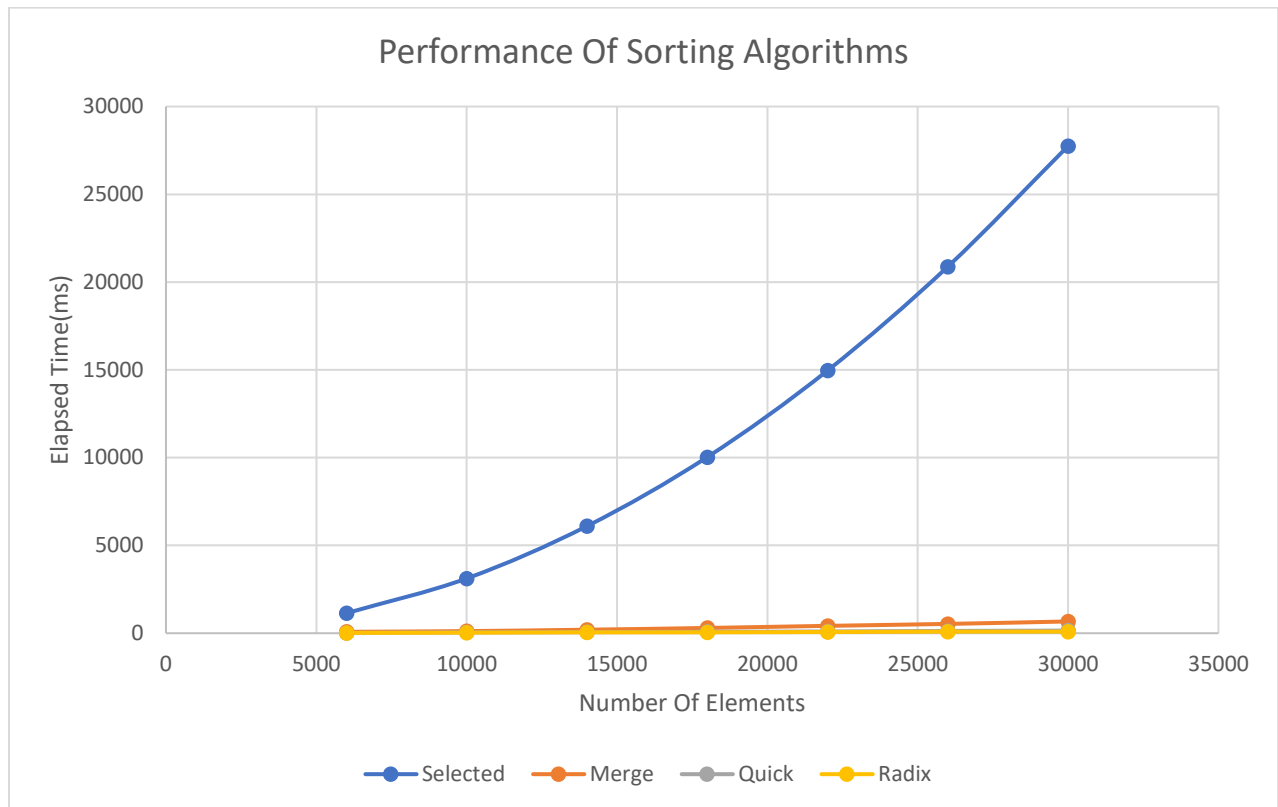
Analysis of Quick Sort with descending arrays.

Array Size	Elapsed time	compCount	moveCount
6000	30 ms	86041	148899
10000	40 ms	157954	244869
14000	60 ms	227938	361773
18000	90 ms	303304	507291
22000	100 ms	377668	621882
26000	130 ms	478797	839037
30000	150 ms	541696	847095

Analysis of Radix Sort with random arrays.

Array Size	Elapsed time
6000	20 ms
10000	30 ms
14000	50 ms
18000	50 ms
22000	70 ms
26000	90 ms
30000	90 ms

QUESTION 3



DISCUSSION

- It can be seen from the graph that the selected sort has the worst time complexity. However, in the graph, there is no obvious difference between the complexity of the remaining algorithms. Nevertheless, the graph suggests that quicksort performs better than the merge sort and although it can not be seen clearly (since to make it obvious much more elements are needed) from the graph, radix sort outperforms both quick sort and merge sort.
- Theoretical data suggest that quicksort is $O(N \log(n))$ for the best-case and average-case, however, $O(N^2)$ for the worst case, on the other hand, merge sort is $O(N \log(n))$ for both best-case, average-case and worst-case.

Question:

Why while theoretical data suggesting that merge sort has better time complexity efficiency, empirical results suggest the reverse, namely quick sort has slightly better time complexity efficiency?

- The underlying reason for this is: In most cases, average time complexity occurs which quick sort has the same time complexity as merge sort in that case. However, since for merge sort, an extra array has to be created, the time to create that auxiliary array is the reason for that slight difference between these algorithms. This result also can be interpreted from the

comparison and movement counts. Although quicksort has greater total comparison and movement counts its time is smaller than merge sort which is explained by the time allocated to create the auxiliary array.

- For selection sort, the time is nearly the same for random, ascending, and descending arrays since, the iteration has not any special case for any of these array types. Moreover, the comparison count is the same for all random, ascending, and descending arrays in the selection sort.
For merge sort the time complexity does not change dependently to the random, ascending, or descending arrays. This also is observed by, equal movement counts for 3 array types and very close comparison numbers for 3 arrays. These close numbers are the indication of a similar elapsed time.
For quick sort on ascending and descending the worst case occurs, since in those cases the array can not be divided, the advantage of divide and conquer is lost.
Radix sort has $O(Nk)$ time complexity for all average, best and worst cases. Moreover, for ascending, descending and random arrays the efficiency of the algorithm is the same since it is a non-comparative algorithm.