CS202: Fundamentals of Computer Science II

HOMEWORK ASSIGNMENT - 1

Algorithm Efficiency and Sorting

Zeynep Begüm Kara
ID: 22003880
CS202 - 01

October 23, 2022

Question 1

a) Let $n > n_0 = 2$ and choose $C = 3$. Then, for every $n > n_0$ we have

$2n < n^3$
$4n^2 < n^3$
$3n^2 < n^3$

This implies $\underbrace{2n + 4n^2 + 3n^2}_{f(n)} < 3n^3$ call $n^3 = g(n)$.

$f(n) < 3\,g(n)$ observe $C = 3$

Therefore $f(n)$ is $O(g(n))$ which means $f(n)$ is $O(n^3)$.

b) i. $T(n) = T(n-1) + n^2$ and we know $T(1) = 1$

Then, $T(n-1) = T(n-2) + (n-1)^2$. Substituting it, we have

$T(n) = T(n-2) + (n-1)^2 + n^2$ We also know $T(n-2) = T(n-3) + (n-2)^2$.

$T(n) = T(n-3) + (n-2)^2 + (n-1)^2 + n^2$ Similarly continuing this process

$\vdots$

$T(n) = T(n-k) + (n-(k-1))^2 + \dots + (n-1)^2 + n^2$ for a $k < n$.

$\vdots$

When $k = (n-1)$ we have the base case $T(1)$. Therefore, let $k = (n-1)$ and rewrite the recurrence relation. Hence,

$T(n) = T(1) + (n-(n-2))^2 + (n-(n-3))^3 + \dots + (n-1)^2 + n^2$.

Therefore,

$T(n) = T(1) + 2^2 + 3^2 + \dots + (n-1)^2 + n^2$ since $T(1) = 1$

$T(n) = 1 + 2^2 + 3^2 + \dots + (n-1)^2 + n^2$

This finite sum is equal to

$T(n) = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3 + 3n^2 + n}{6}$. Let $n > n_0 = 3$. Then,

$2n^3 < n^3$
$3n^2 < n^3$
$n < n^3$

This implies $\underbrace{\frac{2n^3 + 3n^2 + n}{6}}_{f(n)} < 4n^3$ call $n^3 = g(n)$

$f(n) < 4n^3$ observe $C = 4$

Therefore $f(n)$ is $O(g(n))$ meaning that $O(n^3)$.

i. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2}$ and $T(1) = 1$

Then, $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2^2}$ Substituting it $T(n)$ equal to

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2^2}\right] + \frac{n}{2} = 2^2 T\left(\frac{n}{2^2}\right) + \frac{2n}{2^2} + \frac{n}{2} = 2^2 T\left(\frac{n}{2^2}\right) + \frac{2 \cdot n}{2}$$

Then, $T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^3}$ substituting it

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^3}\right] + n = 2^3 T\left(\frac{n}{2^3}\right) + \frac{3n}{2}$$

$$\vdots$$

$$T(k) = 2^{k+1} T\left(\frac{n}{2^k}\right) + \frac{(k+1)n}{2} \quad \text{for a } k < n.$$

When $\frac{n}{2^k} = 1$ we have the base case $T(1)$. Therefore, let $k = \log_2 n$ and rewrite the recurrence relation. Hence,

$$T(n) = 2^{\log_2 n + 1} T(1) + \frac{(\log_2 n + 1)n}{2} \quad \text{since } T(1) = 1$$

$$T(n) = 2^{\log_2 2n} + \frac{n \log_2 2n}{2} \quad \text{using the properties of logarithms.}$$

$$T(n) = 2n + \frac{n}{2}\log_2 2n = 2n + \frac{n}{2}\frac{\log 2n}{\log 2} = n\left(\log 2^5 + \log n\right)$$

$= 5n \log 2 + n \log n$ since $5 \cdot \log 2 \approx 1.5$ we have

$\approx \frac{3}{2}n + n \log n$ since $n < n \log n$ $n > n_0 = 10$

Then, $\frac{3}{2}n < \frac{3n}{2}\log n$ Hence $\underbrace{\frac{3}{2}n + n\log n}_{f(x)} < \underbrace{\frac{5}{2}n\log n}_{\text{call it } g(x).}$

$\frac{5}{2}$ ↓ Choose $c = 3$ for simplicity

It means that $f(n)$ is $O(g(n))$ This implies $f(n)$ is $O(n \log n)$.

c) i- Selection sort

Consider the array as if it splitted by an invisible line |.
This line separates the unsorted and sorted sublists. (unsorted | sorted)
In the beginning everything is unsorted.

21 9 58 28 36 18 27 19 4 25 | (unsorted | sorted)

Then iterate the unsorted list and select the biggest value by comparing other elements in same unsorted list.

21 9 58 28 36 18 27 19 4 25 |

Swap it with the element at the end of the unsorted list.
move the seperator

21 9 25 28 36 18 27 19 4 | 58
(unsorted | sorted)

Now, selected element is in the correct place
We have the same problem with remaining unsorted list. Then again iterate the unsorted list to select the biggest

21 9 25 28 36 18 27 19 4 | 58

swap it with the last element of unsorted list and move seperator

21 9 25 28 4 18 27 19 | 36 58
(unsorted | sorted)

following the same procedure until unsorted sublist became empty
Thus, we have the following steps;

21 9 25 28 4 18 27 19 | 36 58
21 9 25 19 4 18 27 | 28 36 58
21 9 25 19 4 18 | 27 28 36 58
21 9 18 19 4 | 25 27 28 36 58
4 9 18 19 | 21 25 27 28 36 58
4 9 18 | 19 21 25 27 28 36 58
4 9 | 18 19 21 25 27 28 36 58
4 | 9 ...

ii - Insertion Sort

Consider the array as if it is splitted by an invisible line | . This line separates the unsorted and sorted sublists. In the beginning everything is unsorted.

| 21 9 58 28 36 18 27 19 4 25

sorted — unsorted.

Then iterate the unsorted list and select the element at the beginning of the unsorted list

| 21 9 58 28 36 18 27 19 4 25

Then comparing it with the elements of the sorted list, place it to its appropriate place. Since there is no element in sorted list nothing to compare. After placing it move the separator.

21 | 9 58 28 36 18 27 19 4 25

sorted — unsorted.

Now selected element is in correct place. We have the same problem with remaining unsorted list. Again choose the first element of the unsorted list.

21 | 9 58 28 36 18 27 19 4 25

Comparing it with the elements of sorted list. find its place. $9 < 21$. Then look the other element in the list. No element to compare than, place it and move the line and select the next

9 21 | 58 28 36 18 27 19 4 25

sorted — unsorted.

Repeat the same procedure until the unsorted sublist is empty.

9 21 58 | 28 36 18 27 19 4 25

While comparing selected with sorted elements shift the biggerones to right to arrange an appropriate place for selected. Thus shift 58.

9 21 28 58 | 36 18 27 19 4 25

9 21 28 36 58 | 18 27 19 4 25

9 18 21 28 36 58 | 27 19 4 25

9 18 21 27 28 36 58 | 19 4 25

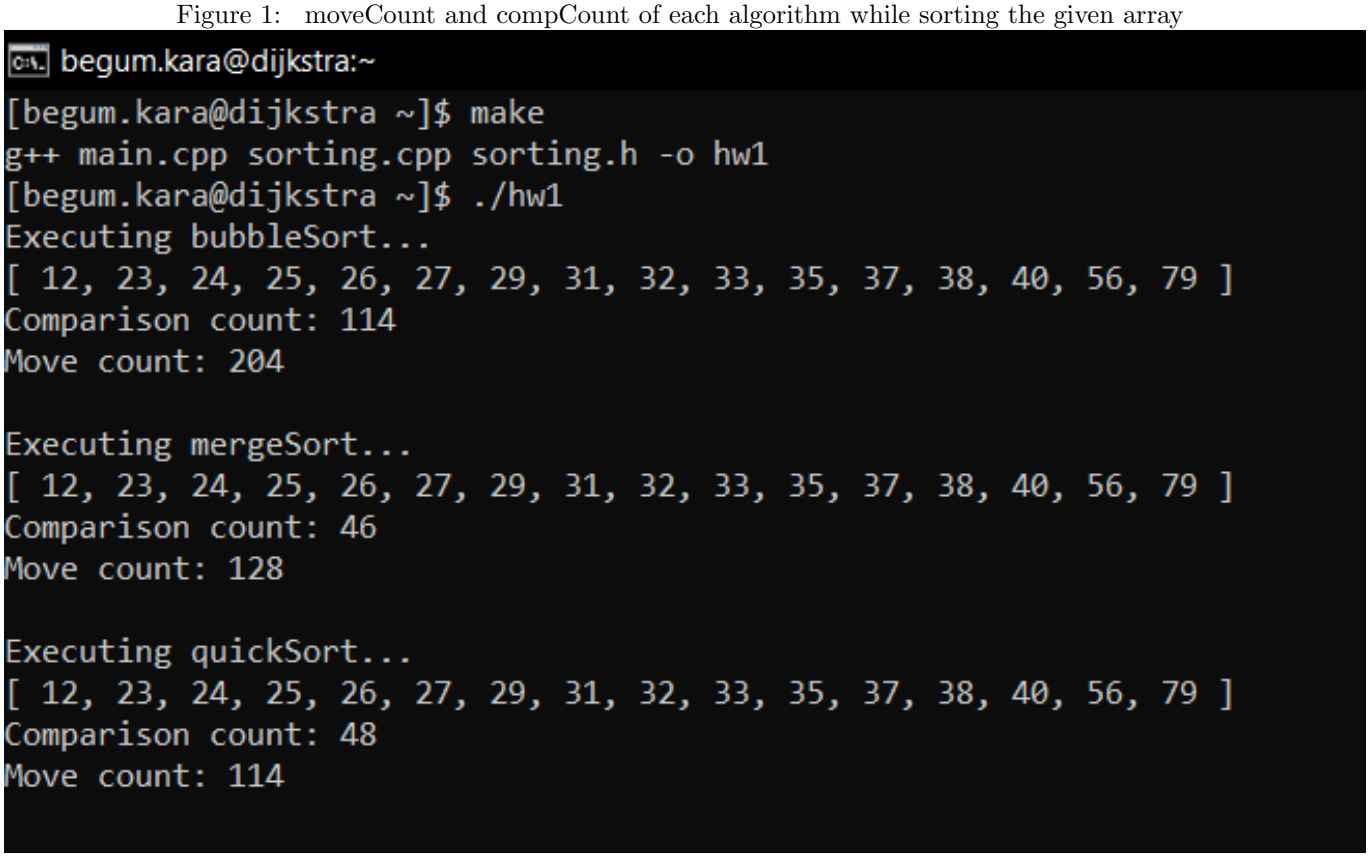
9 18 19 21 27 28 36 58 | 4 25

4 9 18 19 21 27 28 36 58 | 25

4 9 18 19 21 25 27 28 36 58 |

Question 2

Figure 1: moveCount and compCount of each algorithm while sorting the given array

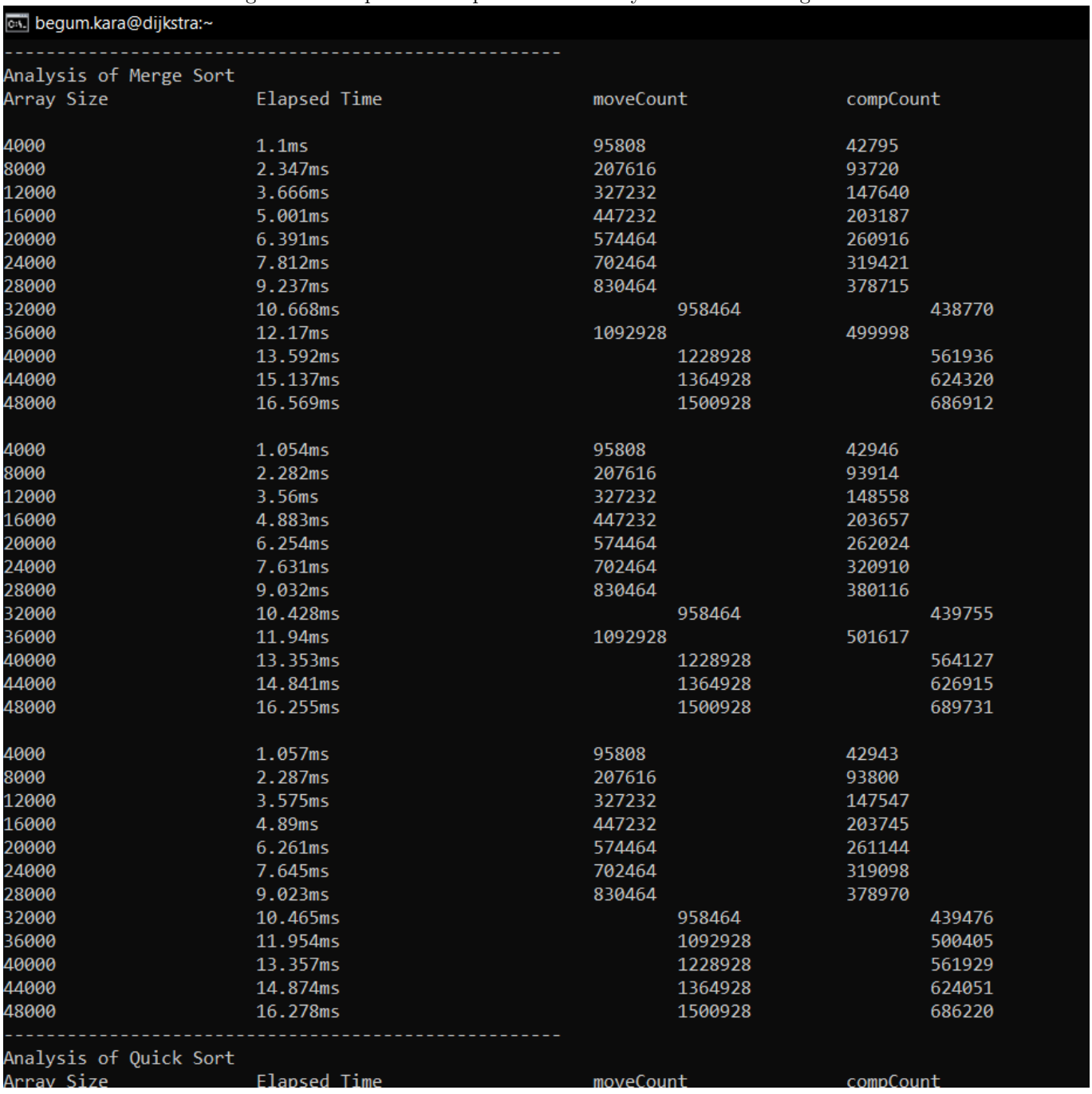


Initial array for each algorithm: {40, 25, 29, 56, 37, 27, 24, 32, 79, 12, 35, 38, 23, 31, 33, 26}

Figure 2: Outputs of the performanceAnalysis function: Bubble Sort

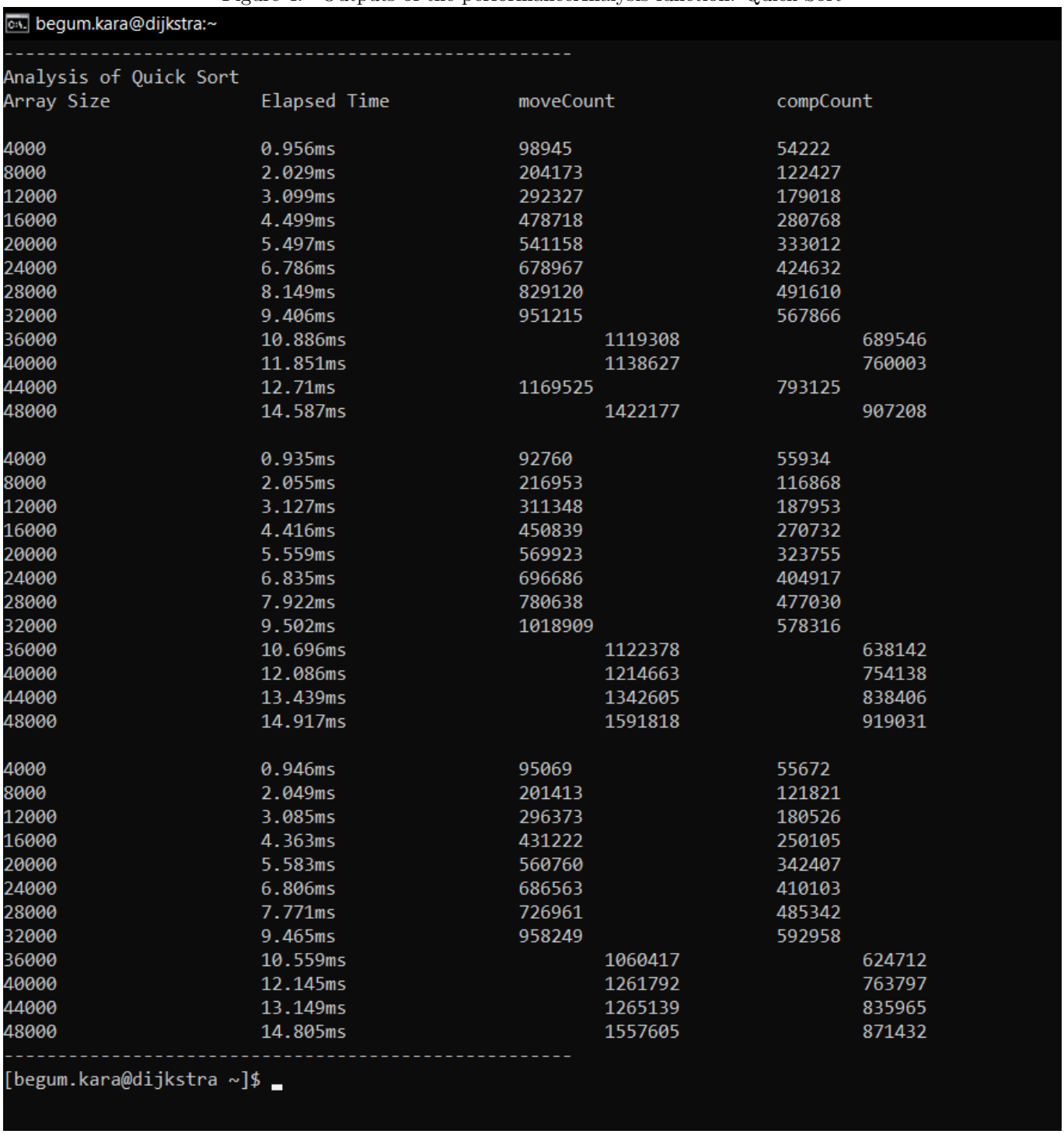
```
Seç "C:\Users\zbegu\Desktop\2.2 Fall\CS202\HW1\202HW1\bin\Debug\202HW1.exe"
Analysis of Bubble Sort
Array Size              Elapsed Time            moveCount               compCount

4000                    43                      11903766                        7996775
8000                    197                     47836737                        31974679
12000                   490                     107351733                       71993220
16000                   857                     192411960                       127960625
20000                   1339                    300814854                       199972980
24000                   1945                    430635036                       287966679
28000                   2624                    586029792                       391945530
32000                   3539                    772066740                       511962885
36000                   4466                    974949810                       647976644
40000                   5585                    1199645205                      799939530
44000                   6786                    1449514941                      967962424
48000                   7978                    1720735710                      1151960247

4000                    0                       0                       3999
8000                    0                       0                       7999
12000                   0                       0                       11999
16000                   0                       0                       15999
20000                   0                       0                       19999
24000                   0                       0                       23999
28000                   0                       0                       27999
32000                   0                       0                       31999
36000                   0                       0                       35999
40000                   0                       0                       39999
44000                   0                       0                       43999
48000                   1                       0                       47999

4000                    50                      23994000                        7998000
8000                    196                     95988000                        31996000
12000                   448                     215982000                       71994000
16000                   776                     383976000                       127992000
20000                   1218                    599970000                       199990000
24000                   1721                    863963997                       287988000
28000                   2361                    1175957997                      391986000
32000                   3088                    1535951991                      511984000
36000                   3917                    1943946000                      647982000
40000                   4912                    -1895027299                     799980000
44000                   5818                    -1391033302                     967978000
48000                   6907                    -839039305                      1151976000
```

Figure 3: Outputs of the performanceAnalysis function: Merge Sort



```
begum.kara@dijkstra:~
-------------------------------------------------
Analysis of Merge Sort
Array Size              Elapsed Time                moveCount              compCount

4000                    1.1ms                       95808                  42795
8000                    2.347ms                     207616                 93720
12000                   3.666ms                     327232                 147640
16000                   5.001ms                     447232                 203187
20000                   6.391ms                     574464                 260916
24000                   7.812ms                     702464                 319421
28000                   9.237ms                     830464                 378715
32000                   10.668ms                    958464                 438770
36000                   12.17ms                     1092928                499998
40000                   13.592ms                    1228928                561936
44000                   15.137ms                    1364928                624320
48000                   16.569ms                    1500928                686912

4000                    1.054ms                     95808                  42946
8000                    2.282ms                     207616                 93914
12000                   3.56ms                      327232                 148558
16000                   4.883ms                     447232                 203657
20000                   6.254ms                     574464                 262024
24000                   7.631ms                     702464                 320910
28000                   9.032ms                     830464                 380116
32000                   10.428ms                    958464                 439755
36000                   11.94ms                     1092928                501617
40000                   13.353ms                    1228928                564127
44000                   14.841ms                    1364928                626915
48000                   16.255ms                    1500928                689731

4000                    1.057ms                     95808                  42943
8000                    2.287ms                     207616                 93800
12000                   3.575ms                     327232                 147547
16000                   4.89ms                      447232                 203745
20000                   6.261ms                     574464                 261144
24000                   7.645ms                     702464                 319098
28000                   9.023ms                     830464                 378970
32000                   10.465ms                    958464                 439476
36000                   11.954ms                    1092928                500405
40000                   13.357ms                    1228928                561929
44000                   14.874ms                    1364928                624051
48000                   16.278ms                    1500928                686220
-------------------------------------------------
Analysis of Quick Sort
Array Size              Elapsed Time                moveCount              compCount
```

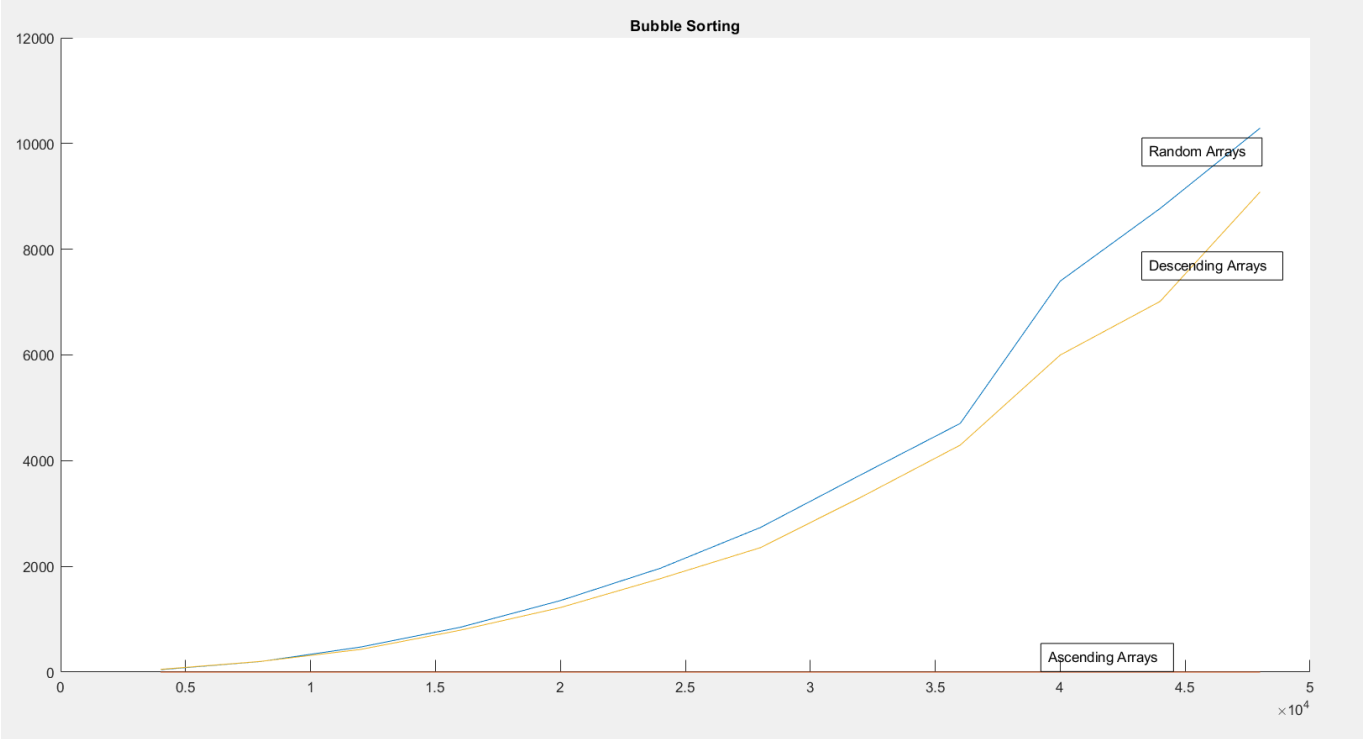
Figure 4: Outputs of the performanceAnalysis function: Quick Sort



```
begum.kara@dijkstra:~
-------------------------------------------------
Analysis of Quick Sort
Array Size              Elapsed Time            moveCount               compCount

4000                    0.956ms                 98945                   54222
8000                    2.029ms                 204173                  122427
12000                   3.099ms                 292327                  179018
16000                   4.499ms                 478718                  280768
20000                   5.497ms                 541158                  333012
24000                   6.786ms                 678967                  424632
28000                   8.149ms                 829120                  491610
32000                   9.406ms                 951215                  567866
36000                   10.886ms                1119308                 689546
40000                   11.851ms                1138627                 760003
44000                   12.71ms                 1169525                 793125
48000                   14.587ms                1422177                 907208

4000                    0.935ms                 92760                   55934
8000                    2.055ms                 216953                  116868
12000                   3.127ms                 311348                  187953
16000                   4.416ms                 450839                  270732
20000                   5.559ms                 569923                  323755
24000                   6.835ms                 696686                  404917
28000                   7.922ms                 780638                  477030
32000                   9.502ms                 1018909                 578316
36000                   10.696ms                1122378                 638142
40000                   12.086ms                1214663                 754138
44000                   13.439ms                1342605                 838406
48000                   14.917ms                1591818                 919031

4000                    0.946ms                 95069                   55672
8000                    2.049ms                 201413                  121821
12000                   3.085ms                 296373                  180526
16000                   4.363ms                 431222                  250105
20000                   5.583ms                 560760                  342407
24000                   6.806ms                 686563                  410103
28000                   7.771ms                 726961                  485342
32000                   9.465ms                 958249                  592958
36000                   10.559ms                1060417                 624712
40000                   12.145ms                1261792                 763797
44000                   13.149ms                1265139                 835965
48000                   14.805ms                1557605                 871432
-------------------------------------------------
[begum.kara@dijkstra ~]$
```

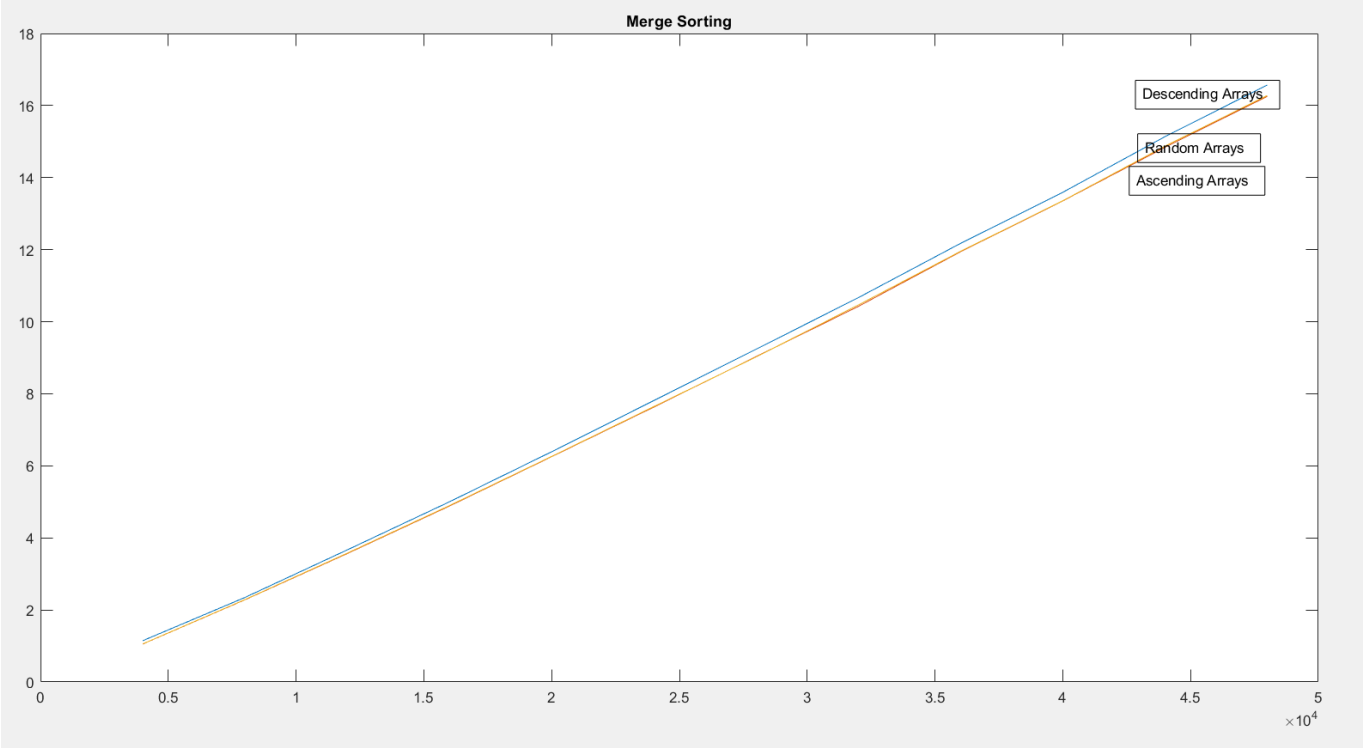
Each algorithm respectively executed for random, descending, and ascending arrays

Figure 1: Bubble Sort: Array Size vs. Time Elapsed



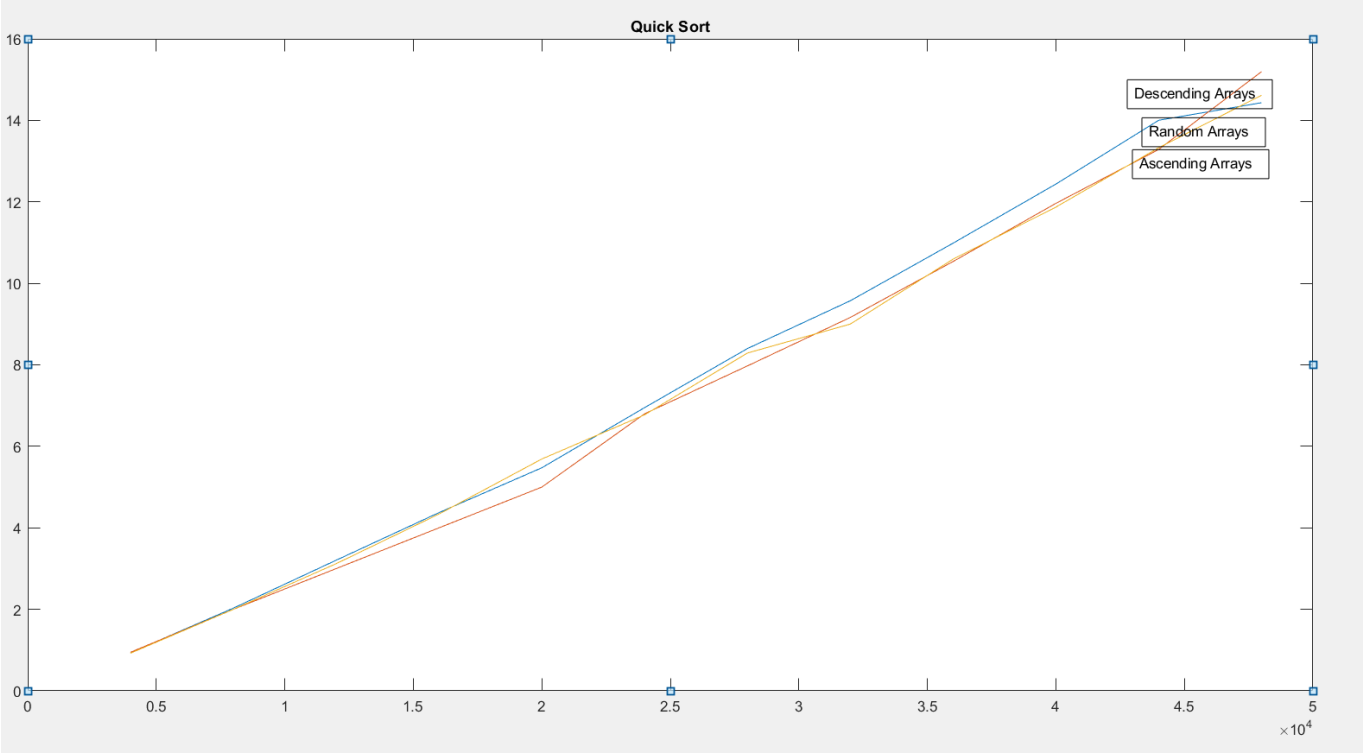
In bubble sort, since it's complexity is $O(n^2)$, we theoretically expect the time elapsed is directly proportional to n^2 where n is the size of the array. The best case of it is when the array is already sorted. In such cases, the algorithm only compares two pairs of elements in array and doesn't make any move as can be seen from the data. Therefore, if comparison speed of the computer is high, it lasts very very short time. Furthermore, in my experiment, comparison time was too small that I couldn't measure it significantly. Thus it is almost zero as seen in the graph. The worst case of this algorithm is when the array is sorted in a descending way. In such cases both number of comparison and number of moves is also order of n^2 as can be seen from the data. By definition, since it is the worst case no other arrangement can exceed it. Unexpectedly, in my experiment, time elapsed in random arrays are more than descending array. Although significance is not too much, the error must be resulted from the computer itself due to its other operations performed. Also, note that last three moveCount is exceeding the int storage, therefore negative due to overflow. Overall, the graph suits $O(n^2)$ complexity except the descending array error.

Figure 2: Merge Sort: Array Size vs. Time Elapsed



In merge sort, since it's complexity is $O(n * logn)$, we theoretically expect the time elapsed is directly proportional to $n * logn$ where n is the size of the array. In each case, where array is random, descending or ascending, algorithm outputs are very consistent as can be seen from the data. It was as expected because it is both worst case and avarage case senarios are in $O(n * logn)$. We observe in each array case its number of moves are almost twice of it. Since we knew Since we knew the key comparisons in worst case is $2n - 1$, and number of moves is $4n$, it is consistent to have nearly twice data in experiment for moveCount and compCount. Overall, the graph looks like $O(n)$ rather than $O(n * logn)$ but I think it is a result of pseudo random numbers and their arrangements in arrays.

Figure 3: Quick Sort: Array Size vs. Time Elapsed



In quick sort, since it's complexity is $O(n^2)$, we theoretically expect the time elapsed is directly proportional to n^2 where n is the size of the array. I choose pivot as the first element of array. Therefore, I expect worst elapsed time when the array is already sorted. However, results didn't turn out that, but yet it is close. This difference might be caused from the execution environment. Our graph is not as consistent as merge sort since the cases are much more make difference in this algorithm. Especially for large n's we see fluctuations. Therefore, results would be much more accurate and probably closer to the theoretical values when we examine results for much more bigger sized arrays. Overall, we expect $O(n * logn)$ complexity for best case which is consistent to graph but our expectation of worst case which is $O(n^2)$ has failed. This might be resulted from execution environment and unprecise measurement of time elapsed.