CS 202, Spring 2021
NAME: N. Onur Vural
ID: 21902330
SECTION: 2
Homework 1 – Algorithm Efficiency and Sorting

Question 1
(a) Showing **$f(n) = 5n^3 + 4n^2 + 10 = O(n^4)$**

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

$0 \le 5n^3 + 4n^2 + 10 \le cn^4$ for all $n \ge n_0$

$5/n + 4/n^2 + 10/n^4 \le c$ for all $n \ge n_0$

Choose $c = 3$ and $n_0 = 3$

- $5n^3 + 4n^2 + 10 \le 3n^4$ for all $n \ge 3$

(b) Tracing sorting algorithms to sort integer array [24, 8, 51, 28, 20, 29, 21, 17, 38, 27] in ascending order.

- **Insertion sort ($O(n^2)$)**

```
void     insertionSort( int arr[], int arrSize) {
         for ( int notSorted = 1; notSorted < arrSize; ++notSorted) {
                  int     key = arr[notSorted]; // get current item to compare
                  int     index = notSorted;    // get location
                  for ( ; (index > 0) && (arr[index-1] > key);  --index)
                           arr[index] = arr[index-1]; // shift while item is bigger than key
                  arr[index] = key;  // insert current key
         }
}
```

[24, 8, 51, 28, 20, 29, 21, 17, 38, 27]

- Select first element as sorted (arr[0]) **// notSorted = 1**

[24, 8, 51, 28, 20, 29, 21, 17, 38, 27]

- Key to compare => 8 **// key = arr[1] & index = 1**
- Shift 24 to right, insert 8 to index 0 **// as arr[0] > 8 (1 comparison)**
- Increase sorted part by 1 **// ++notSorted (2)**

[8, 24, 51, 28, 20, 29, 21, 17, 38, 27]

- Key to compare => 51 **// key = arr[2] & index = 2**
- Compare once, there exists no larger element in sorted part **// arr[1] is not smaller than 51 (1 comparison)**

- Increase sorted part by 1 (continue iteration) **++notSorted (3)**

[8, 24, 51, 28, 20, 29, 21, 17, 38, 27]

- Key to compare => 28 **// key = arr[3] & index = 3**
- Shift 51 to right, insert 28 to index 2 **// as arr[2] > 28, arr[1] not greater than 28 (2 comparisons)**
- Increase sorted part by 1 **// ++notSorted (4)**

[8, 24, 28, 51, 20, 29, 21, 17, 38, 27]

- Key to compare => 20 **// key = arr[4] & index = 4**
- Shift 51, 28, 24 to right respectively, insert 20 to index 1 **// up to arr[0] not greater than 20 (4 comparisons)**
- Increase sorted part by 1 **// ++notSorted (5)**

[8, 20, 24, 28, 51, 29, 21, 17, 38, 27]

- Key to compare => 29 **// key = arr[5] & index = 5**
- Shift 51 to right, insert 29 to index 4 **// up to arr[3] not greater than 29 (2 comparisons)**
- Increase sorted part by 1 **// ++notSorted (6)**

[8, 20, 24, 28, 29, 51, 21, 17, 38, 27]

- Key to compare => 21 **// key = arr[6] & index = 6**
- Shift 51, 29, 28, 24 to right respectively, insert 21 to index 2 **// up to arr[3] not greater than 21 (5 comparisons)**
- Increase sorted part by 1 **// ++notSorted (7)**

[8, 20, 21, 24, 28, 29, 51, 17, 38, 27]

- Key to compare => 17 **// key = arr[7] & index = 7**
- Shift 51, 29, 28, 24, 21, 20 to right respectively, insert 17 to index 1 **// up to arr[0] not greater than 17 (7 comparisons)**
- Increase sorted part by 1 **// ++notSorted (8)**

[8, 17, 20, 21, 24, 28, 29, 51, 38, 27]

- Key to compare => 38 **// key = arr[8] & index = 8**
- Shift 51 to right, insert 38 to index 7 **// up to arr[6] not greater than 38 (2 comparisons)**
- Increase sorted part by 1 **// ++notSorted (9)**

[8, 17, 20, 21, 24, 28, 29, 38, 51, 27]

- Key to compare => 27 **// key = arr[9] & index = 9**
- Shift 51, 38, 29, 28 to right, insert 27 to index 5 **// up to arr[4] not greater than 38 (5 comparisons)**
- Increase sorted part by 1 **// ++notSorted (10)**

[8, 17, 20, 21, 24, 27, 28, 29, 38, 51] **//THE ARRAY IS SORTED… (notSorted == arrSize)**

- **Bubble sort ($O(n^2)$)**

// the characteristic of the algorithm is making comparisons with next element
// it is useful as it gives the opportunity to exit immediately

```
void    bubbleSort( int arr[], int arrSize) {

        flagSorted = false; // helps for immediate exit

        for ( int cont = 1; (cont< arrSize) && ! flagSorted; ++cont) {

                flagSorted = true;

                for ( int index = 0; arrSize - cont;  ++index) {

                        int nextIndex = index + 1;

                        if ( arr[index] > arr[nextIndex]) {

                                swap( arr[index], arr[nextIndex]);

                                flagSorted = false; // change signal

                         }

                }

        }

 }
```

[24, 8, 51, 28, 20, 29, 21, 17, 38, 27]

- flagSorted = false **// initialize**
- **cont = 1**, flagSorted = true, index = 0, **inner loop: for (0->9)**
- nextIndex = 1
- swap 24 & 8 (arr[0] > arr[1]) => [**8, 24**, 51, 28, 20, 29, 21, 17, 38, 27]
- **flagSorted = false**, index++ (1), nextIndex (2)
- no swap 24 & 51 !(arr[1] > arr[2]) => [8, 24, 51, 28, 20, 29, 21, 17, 38, 27]
- index++ (2), nextIndex (3)
- swap 51 & 28 (arr[2] > arr[3]) => [8, 24, **28, 51**, 20, 29, 21, 17, 38, 27]
- flagSorted = false, index++ (3), nextIndex (4)
- swap 51 & 20 (arr[3] > arr[4]) => [8, 24, 28, **20, 51**, 29, 21, 17, 38, 27]
- flagSorted = false, index++ (4), nextIndex (5)
- swap 51 & 29 (arr[4] > arr[5]) => [8, 24, 28, 20, **29, 51**, 21, 17, 38, 27]
- flagSorted = false, index++ (5), nextIndex (6)
- swap 51 & 21 (arr[5] > arr[6]) => [8, 24, 28, 20, 29, **21, 51**, 17, 38, 27]
- flagSorted = false, index++ (6), nextIndex (7)
- swap 51 & 17 (arr[6] > arr[7]) => [8, 24, 28, 20, 29, 21, **17, 51**, 38, 27]
- flagSorted = false, index++ (7), nextIndex (8)
- swap 51 & 38 (arr[7] > arr[8]) => [8, 24, 28, 20, 29, 21, 17, **38, 51,** 27]
- flagSorted = false, index++ (8), nextIndex (9)
- swap 51 & 27 (arr[8] > arr[9]) => [8, 24, 28, 20, 29, 21, 17, 38, **27, 51**]
- flagSorted = false, index++ (9) **(exit inner loop as !(index<9))**

- **cont = 2, flagSorted = true**, index = 0, arrSize - cont = 8
- **inner loop: for (0->8)**   [**8, 24, 28, 20, 29, 21, 17, 38, 27, 51**] **(51 is fixed)**
- nextIndex = 1
- no swap 8 & 24 !(arr[0] > arr[1]) => [8, 24, 28, 20, 29, 21, 17, 38, 27, 51]
- index++ (1), nextIndex (2)

- no swap 24 & 28 !(arr[1] > arr[2]) => [8, 24, 28, 20, 29, 21, 17, 38, 27, 51]
- index++ (2), nextIndex (3)
- swap 28 & 20 (arr[2] > arr[3]) => [8, 24, **20, 28**, 29, 21, 17, 38, 27, 51]
- **flagSorted = false**, index++ (3), nextIndex (4)
- no swap 28 & 29 !(arr[3] > arr[4]) => [8, 24, 20, 28, 29, 21, 17, 38, 27, 51]
- index++ (4), nextIndex (5)
- swap 29 & 21 (arr[4] > arr[5]) => [8, 24, 20, 28, **21, 29**, 17, 38, 27, 51]
- index++ (5), nextIndex (6)
- swap 29 & 17 (arr[5] > arr[6]) => [8, 24, 20, 28, 21, **17, 29**, 38, 27, 51]
- index++ (6), nextIndex (7)
- no swap 29 & 38 !(arr[6] > arr[7]) => [8, 24, 20, 28, 21, 17, 29, 38, 27, 51]
- index++ (7), nextIndex (8)
- swap 38 & 27 (arr[7] > arr[8]) => [8, 24, 20, 28, 21, 17, 29, **27, 38**, 51]
- index++ (8) **(exit inner loop as !(index<8))**

- **cont = 3, flagSorted = true**, index = 0, arrSize - cont = 7
- **inner loop: for (0->7)** **[8, 24, 20, 28, 21, 17, 29, 27, 38, 51] (38 is fixed)**
- nextIndex = 1
- no swap 8 & 24 !(arr[0] > arr[1]) => [8, 24, 20, 28, 21, 17, 29, 27, 38, 51]
- index++ (1), nextIndex (2)
- swap 24 & 20 (arr[1] > arr[2]) => [8, **20, 24**, 28, 21, 17, 29, 27, 38, 51]
- **flagSorted = false**, index++ (2), nextIndex (3)
- no swap 24 & 28 !(arr[2] > arr[3]) => [8, 20, 24, 28, 21, 17, 29, 27, 38, 51]
- index++ (3), nextIndex (4)
- swap 28 & 21 (arr[3] > arr[4]) => [8, 20, 24, **21, 28,** 17, 29, 27, 38, 51]
- index++ (4), nextIndex (5)
- swap 28 & 17 (arr[4] > arr[5]) => [8, 20, 24, 21, **17, 28,** 29, 27, 38, 51]
- index++ (5), nextIndex (6)
- no swap 28 & 29 !(arr[5] > arr[6]) => [8, 20, 24, 21, 17, 28, 29, 27, 38, 51]
- index++ (6), nextIndex (7)
- swap 29 & 27 (arr[6] > arr[7]) => [8, 20, 24, 21, 17, 28, **27, 29,** 38, 51]
- index++ (7) **(exit inner loop as !(index<7))**

- **cont = 4, flagSorted = true**, index = 0, arrSize - cont = 6
- **inner loop: for (0->6)** **[8, 20, 24, 21, 17, 28, 27, 29, 38, 51] (29 is fixed)**
- nextIndex = 1
- no swap 8 & 20 !(arr[0] > arr[1]) => [8, 20, 24, 21, 17, 28, 27, 29, 38, 51]
- index++ (1), nextIndex (2)
- no swap 20 & 24 !(arr[0] > arr[1]) => [8, 20, 24, 21, 17, 28, 27, 29, 38, 51]
- index++ (2), nextIndex (3)
- swap 24 & 21 (arr[2] > arr[3]) => [8, 20, **21, 24,** 17, 28, 27, 29, 38, 51]
- **flagSorted = false**, index++ (3), nextIndex (4)
- swap 24 & 17 (arr[3] > arr[4]) => [8, 20, 21, **17, 24,** 28, 27, 29, 38, 51]
- index++ (4), nextIndex (5)
- no swap 24 & 28 !(arr[4] > arr[5]) => [8, 20, 21, 17, 24, 28, 27, 29, 38, 51]
- index++ (5), nextIndex (6)
- swap 28 & 27 (arr[5] > arr[6]) => [8, 20, 21, 17, 24, **27, 28,** 29, 38, 51]
- index++ (6) **(exit inner loop as !(index<6))**

- **cont = 5, flagSorted = true**, index = 0, arrSize - cont = 5
- **inner loop: for (0->5) [8, 20, 21, 17, 24, 27, 28, 29, 38, 51] (28 is fixed)**
- nextIndex = 1
- no swap 8 & 20 !(arr[0] > arr[1]) => [8, 20, 21, 17, 24, 27, 28, 29, 38, 51]
- index++ (1), nextIndex (2)
- no swap 20 & 21 !(arr[1] > arr[2]) => [8, 20, 21, 17, 24, 27, 28, 29, 38, 51]
- index++ (2), nextIndex (3)
- swap 21 & 17 (arr[2] > arr[3]) => [8, 20, **17, 21,** 24, 27, 28, 29, 38, 51]
- **flagSorted = false**, index++ (3), nextIndex (4)
- no swap 21 & 24 !(arr[3] > arr[4]) => [8, 20, 17, 21, 24, 27, 28, 29, 38, 51]
- index++ (4), nextIndex (5)
- no swap 24 & 27 !(arr[4] > arr[5]) => [8, 20, 17, 21, 24, 27, 28, 29, 38, 51]
- index++ (5) **(exit inner loop as !(index<5))**

- **cont = 6, flagSorted = true**, index = 0, arrSize - cont = 4
- **inner loop: for (0->4) [8, 20, 17, 21, 24, 27, 28, 29, 38, 51] (27 is fixed)**
- nextIndex = 1
- no swap 8 & 20 !(arr[0] > arr[1]) => [8, 20, 17, 21, 24, 27, 28, 29, 38, 51]
- index++ (1), nextIndex (2)
- swap 20 & 17 (arr[1] > arr[2]) => [8, **17, 20,** 21, 24, 27, 28, 29, 38, 51]
- **flagSorted = false**, index++ (2), nextIndex (3)
- no swap 20 & 21 !(arr[2] > arr[3]) => [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]
- index++ (3), nextIndex (4)
- no swap 21 & 24 !(arr[3] > arr[4]) => [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]
- index++ (4) **(exit inner loop as !(index<4))**

- **cont = 7, flagSorted = true**, index = 0, arrSize - cont = 3
- **inner loop: for (0->3) [8, 17, 20, 21, 24, 27, 28, 29, 38, 51] (24 is fixed)**
- nextIndex = 1
- no swap 8 & 17 !(arr[0] > arr[1]) => [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]
- index++ (1), nextIndex (2)
- no swap 17 & 20 !(arr[1] > arr[2]) => [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]
- index++ (2), nextIndex (3)
- no swap 20 & 21 !(arr[2] > arr[3]) => [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]
- index++ (3) **(exit inner loop as !(index<3))**

- ! flagSorted condition is no longer functioning as flagSorted = true after inner loop thus the outer loop terminates

[8, 17, 20, 21, 24, 27, 28, 29, 38, 51] **//THE ARRAY IS SORTED**

Question 2

-Run your executable and add the screenshot of the console to the solution of Question 2 in the pdf submission.

```
[onur.vural@dijkstra SotingAlgortihms]$ ./SortMake
The array before selection sort:
12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8,
comp num is: 120
move num is: 45
The array after selection sort:
3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

The array before merge sort:
12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8,
comp num is: 46
move num is: 128
The array after merge sort:
3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

The array before quick sort:
12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8,
comp num is: 45
move num is: 102
The array after quick sort:
3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

The array before radix sort:
12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8,
The array after radix sort:
3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

THE ANALYSIS IS BEING DONE. PLEASE WAIT...
Analysis of Selection Sort
-Array Size-     -Elapsed Time-       -compCount-    -moveCount-
```

-The `performanceAnalysis` function needs to produce an output similar to the one given on the next page. Include this output to the answer of Question 2 in the pdf submission.

```
Analysis of Selection Sort
-Array Size-     -Elapsed Time-        -compCount-    -moveCount-
Randomized Inputs:
6000               80ms                17997000       17997
10000               240ms                49995000       29997
14000               470ms                97993000       41997
18000               780ms                161991000       53997
22000               1160ms                241989000       65997
26000               1610ms                337987000       77997
30000               2140ms                449985000       89997
Ascending Inputs:
6000               90ms                17997000       17997
10000               250ms                49995000       29997
14000               490ms                97993000       41997
18000               820ms                161991000       53997
22000               1230ms                241989000       65997
26000               1710ms                337987000       77997
30000               2290ms                449985000       89997
Descending Inputs:
6000               90ms                17997000       17997
10000               250ms                49995000       29997
14000               490ms                97993000       41997
18000               790ms                161991000       53997
22000               1190ms                241989000       65997
26000               1670ms                337987000       77997
30000               2210ms                449985000       89997
------------------------------------------------------------
```

```
------------------------------------------------------------
Analysis of Merge Sort
-Array Size-     -Elapsed Time-       -compCount-    -moveCount-
Randomized Inputs:
6000               0ms                67827       151616
10000               10ms                120389       267232
14000               10ms                175419       387232
18000               10ms                231986       510464
22000               10ms                290108       638464
26000               10ms                348868       766464
30000               10ms                408597       894464
Ascending Inputs:
6000               0ms                39152       151616
10000               0ms                69008       267232
14000               0ms                99360       387232
18000               0ms                130592       510464
22000               0ms                165024       638464
26000               10ms                197072       766464
30000               10ms                227728       894464
Descending Inputs:
6000               0ms                36656       151616
10000               0ms                64608       267232
14000               0ms                94256       387232
18000               10ms                124640       510464
22000               10ms                154208       638464
26000               10ms                186160       766464
30000               10ms                219504       894464
------------------------------------------------------------
```

```
------------------------------------------------------------
Analysis of Quick Sort
-Array Size-     -Elapsed Time-      -compCount-   -moveCount-
Randomized Inputs:
6000                0ms                87053      151863
10000               0ms                152142      262255
14000               0ms                220119      382840
18000               10ms               311268      493286
22000               10ms               370141      615472
26000               0ms                436221      743165
30000               10ms               525538      893423
Ascending Inputs:
6000                80ms               17997000      23996
10000               210ms                49995000      39996
14000               410ms                97993000      55996
18000               680ms                161991000      71996
22000               1020ms                241989000      87996
26000               1420ms                337987000      103996
30000               1890ms                449985000      119996
Descending Inputs:
6000                160ms               17997000      27023996
10000               440ms                49995000      75039996
14000               860ms                97993000      147055996
18000               1430ms                161991000      243071996
22000               2140ms                241989000      363087996
26000               2990ms                337987000      507103996
30000               3980ms                449985000      675119996
------------------------------------------------------------
```

```
--------------------------------------------------------------
Analysis of Radix Sort
-Array Size-     -Elapsed Time-
Randomized Inputs:
6000                10ms
10000               10ms
14000               10ms
18000               20ms
22000               20ms
26000               30ms
30000               30ms
Ascending Inputs:
6000                0ms
10000               10ms
14000               10ms
18000               20ms
22000               20ms
26000               20ms
30000               30ms
Descending Inputs:
6000                10ms
10000               10ms
14000               10ms
18000               10ms
22000               20ms
26000               20ms
30000               30ms
--------------------------------------------------------------
```

Question 3

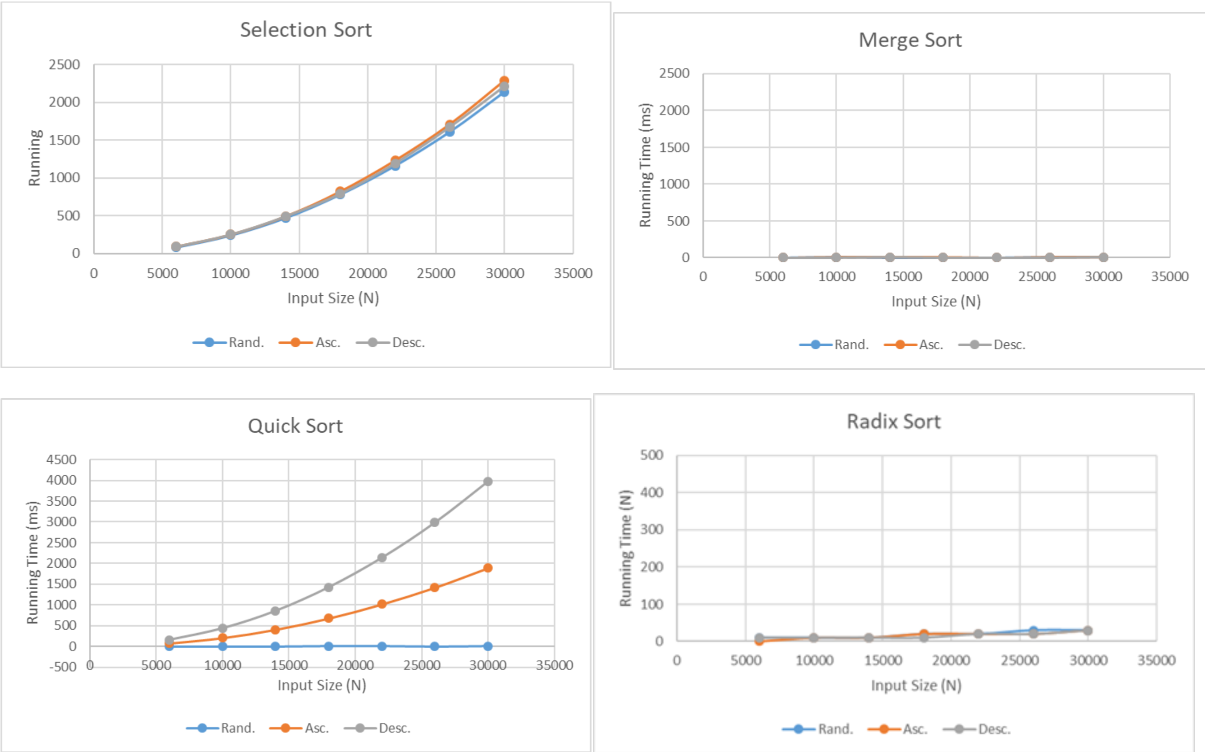

Figure 1: The relation between input size and running time for each sorting algorithm

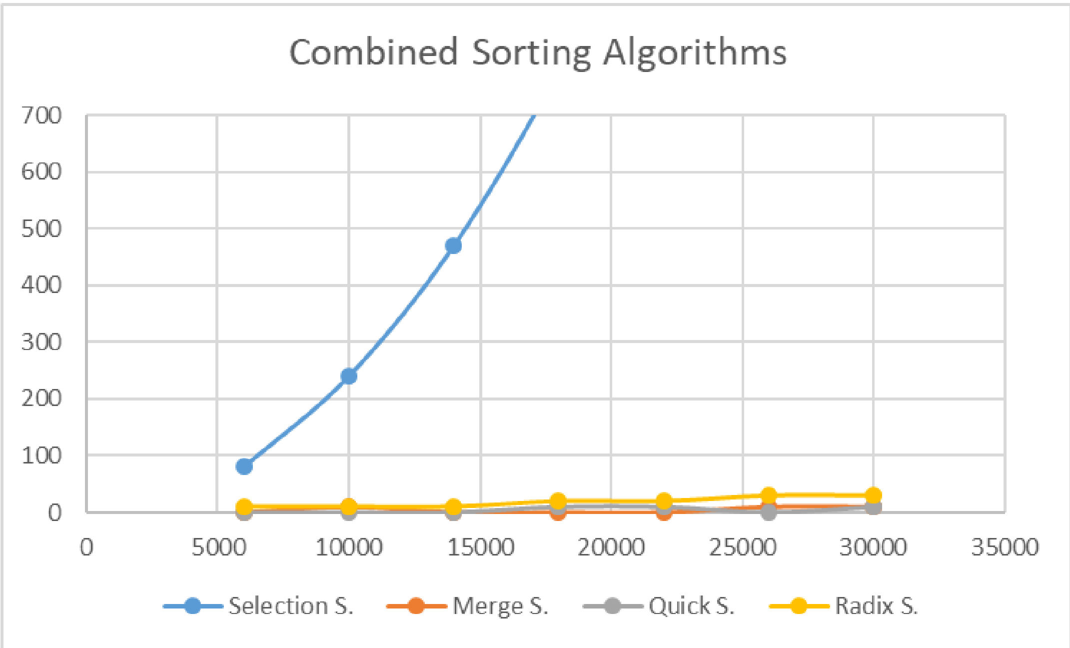

Figure 2: The sorting algorithms combined (for random inputs)

ANALYSIS REPORT:

For Selection Sort, it can be observed from the experimental results that the algorithm takes the highest running time among the four and therefore becomes very slow for very large inputs. This is in confirmatory with theoretical expectations as the algorithm proceeds by comparing all the items with each other while iterating inside two loop structure and thus becoming $O(n^2)$. This exponential relation is clearly noticable in the corresponding graph. Besides, it must be stated that the algorithm takes $O(n^2)$ for all three cases; random data, ascending data, descending data respectively. This is mostly because the algorithm does not depend on the initial organization of the data.

When it comes to Merge Sort, it turns out to be a highly efficent algorithm just as expected showing remarkably low running time overall. At this point it must be noted that although it almost looks like a constant relation *O(1)* (the reason is that it proceeds higly efficiently under large inputs when compared to Selection Sort and to understand it's full behaviour much larger inputs than 30000 must be supplied) in the graph, it is actually *O(nlogn)* for all three cases as the algorithm proceeds under binary recursion, splitting the array into two halves continously. Altough all cases have have same time complexity, ascending and descending arrays result in a decrease on the number of key comparisons in a considerable scale as seen on the running time data on Q2.

Quick Sort Algorithm shows parallelity with theoretical expectations as it is expected to be $O(n^2)$ for worst case and *O(nlogn)* else. The key concept in Quick Sort is not the initial order of the data but the value of the pivot. To provide more efficiency the pivot, which sepates the smaller values to one side and larger to another, must split the array in a more balanced fashion (50-50 split is idealized) while entering into recursion. This explains why ascending and descending values show $O(n^2)$ and randomized *O(nlogn)* in the experimental findings. In a list which is already sorted, pivot value taken as the first entry means that (it is the largest or smallest value in the entire list) the split will completely be one parted (n-1 sub-list size) wheras random values show the characteristics of averge case *O(nlogn)* with the randomized pivot value.

The experimental results for Radix Sort demonstrate that it is also a highly efficient algorithm as the graph shows a linear relation *O(cn)*(where c is digit number in this case) meaning that it grows relatively slower for high inputs when compared to algorithms with time complexity of $O(n^2)$ such as Bubble Sort or Selection Sort. But at this point it has to mentioned that although it provides remarkable efficiency, it requires to form groups that in order to hold the original data for each, resulting large memory usage (size of data * digit number). The algorithm is not concerned with the former organization of data thus for random, ascending and descending data result in same time complexity of *O(cn)*. Different from all the others, Radix Sort is not only dependant on total size but also a constant.

As a final comparison it can be observed from the graphs that the Selection Sort is the least efficient algorithm overall among others, growing exponentially with respect to input size. When it comes to random inputs, Merge Sort and Quick Sort act very similarly, and behave as *O(nlogn)* which results in little execution time (QS is a bit faster). But for ascending and descending data, in other words sorted data, the experimental findings clearly demonstrate that Merge Sort outperfoms Quick Sort as Quick Sort grows exponentially just like Selection Sort due to the selection of pivot value. In our data range (8000-30000), Radix Sort produces similar runtime results to Merge Sort and average case of Quick Sort but as it is linear (*O(cn)*), the theoretical information suggests that for more and more larger values it will grow much slower than the two and take less time if c, constant value, is not high (apart from the fact that it will consume much more space).