Natalia Solar

CSD-235-335-ABBOTT - S21

Assignment 5

Output:

Sorted ascending dataset

dataset size == 10

Quick Sort took 940900

Merge Sort took 1119900

Insertion Sort took 1643400

dataset size == 100

Quick Sort took 38801

Merge Sort took 85801

Insertion Sort took 3900

dataset size == 1000

Quick Sort took 1411401

Merge Sort took 826500

Insertion Sort took 43599

dataset size == 10000

Quick Sort took 1050599

Merge Sort took 2661200

Insertion Sort took 778100

dataset size == 100000

Quick Sort took 3988200

Merge Sort took 11414501

Insertion Sort took 1794800

dataset size == 1000000

Quick Sort took 21025600

Merge Sort took 90392801

Insertion Sort took 4651200

Sorted descending dataset

dataset size == 10

Quick Sort took 77200

Merge Sort took 185501

Insertion Sort took 1001

dataset size == 100

Quick Sort took 143199

Merge Sort took 299900

Insertion Sort took 35199

dataset size == 1000

Quick Sort took 393401

Merge Sort took 552699

Insertion Sort took 4075700

dataset size == 10000

Quick Sort took 1454701

Merge Sort took 2143799

Insertion Sort took 110743100

dataset size == 100000

Quick Sort took 1775301

Merge Sort took 8850000

Insertion Sort took 7225020001

dataset size == 1000000

Quick Sort took 20496999

Merge Sort took 79860300

Insertion Sort took 724753641900

Randomized dataset

dataset size == 10

Quick Sort took 1700

Merge Sort took 2400

Insertion Sort took 400

dataset size == 100

Quick Sort took 40800

Merge Sort took 18200

Insertion Sort took 5600

dataset size == 1000

Quick Sort took 100700

Merge Sort took 183600

Insertion Sort took 380300

dataset size == 10000

Quick Sort took 903300

Merge Sort took 2093200

Insertion Sort took 35124400

dataset size == 100000

Quick Sort took 9760900

Merge Sort took 18918200

Insertion Sort took 3601078999

dataset size == 1000000

Quick Sort took 116622900

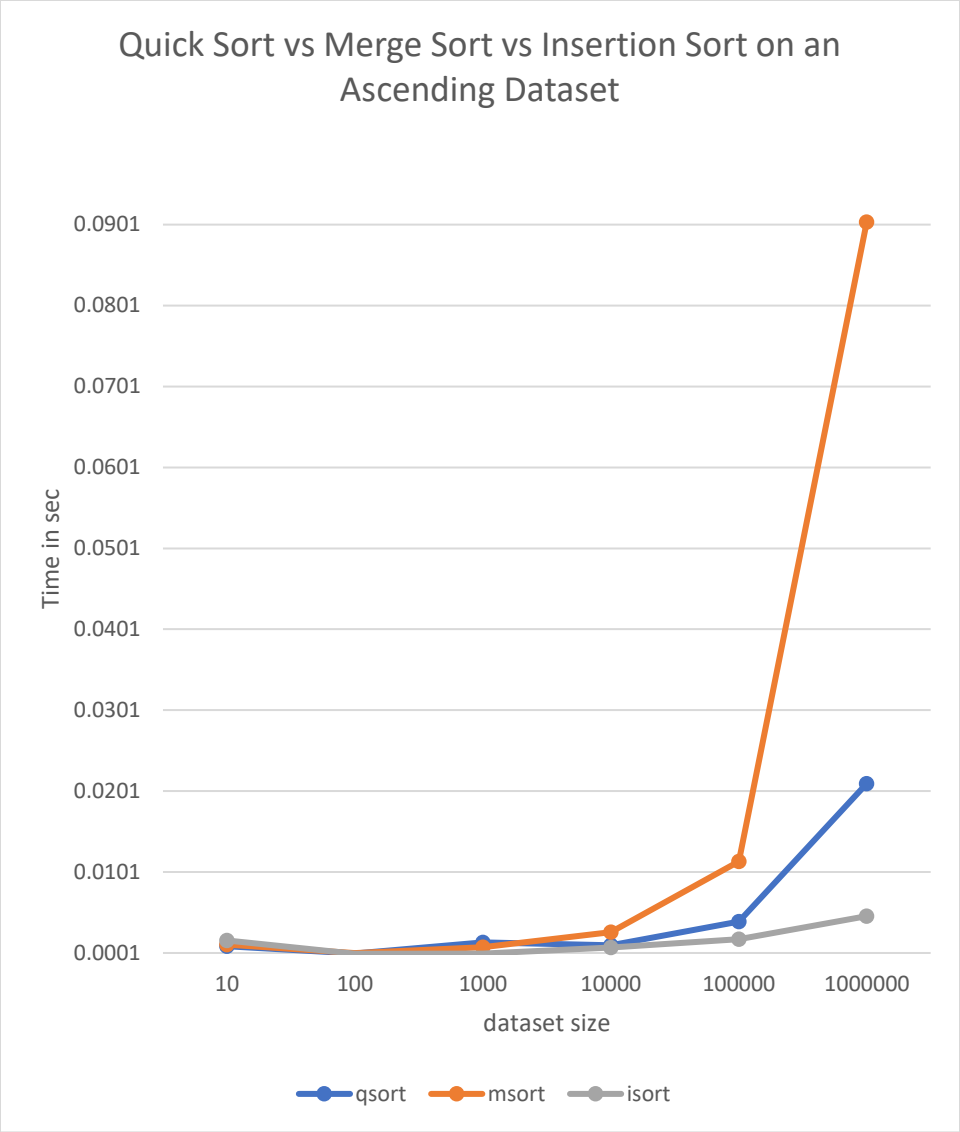
Merge Sort took 184433400

Insertion Sort took 414528533599

1. Ascending dataset

Results:

| | size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|---|
| Time in nanoseconds | qsort | 940900 | 38801 | 1411401 | 1050599 | 3988200 | 21025600 |
| | msort | 1119900 | 85801 | 826500 | 2661200 | 11414501 | 90392801 |
| | isort | 1643400 | 3900 | 43599 | 778100 | 1794800 | 4651200 |

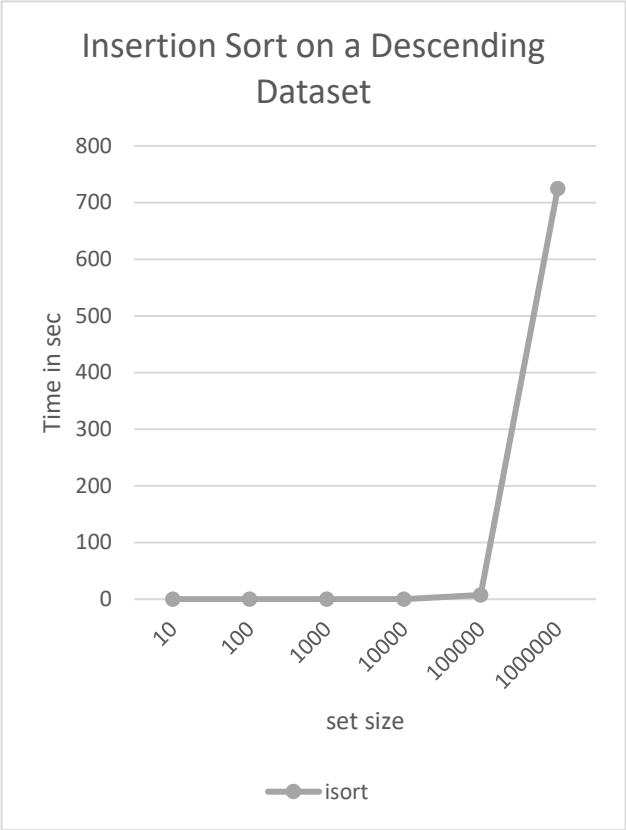


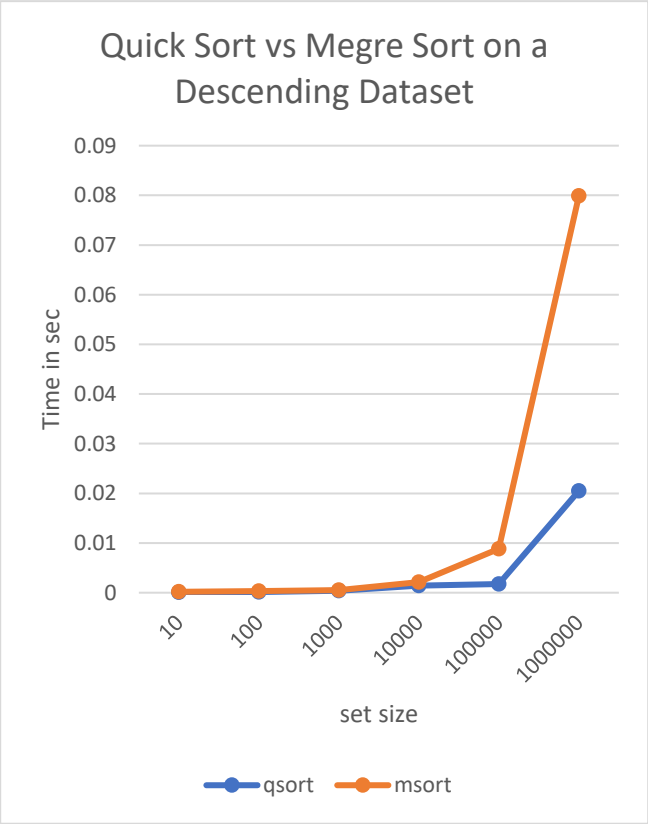

The expected most efficient sorting algorithm was quick sort. However, given the ascending sorted dataset the insertion sort had the best performance out of those three sorting algorithms. The reason for that could be that an ascending dataset is the best case for insertion sort with the time complexity of O(N). In this case insertion sort doesn't require any swaps, which is the most time-consuming procedure in this type of sort. Although, this is not the worst case for quick sort and merge sort, and the time complexity for both is O(NlogN) here, the recursion is probably the part that makes those algorithms slower. The recursion requires to invoke quicksort recursively on each partition and merge sort on each half of the array. The reason why merge sort has the worst performance here is probably that in addition to recursion, this algorithm requires allocating a new temporary array each time the merge sort has been invoked.

2. Descending dataset

Results:

| | size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|---|
| Time in nanoseconds | qsort | 77200 | 143199 | 393401 | 1454701 | 1775301 | 20496999 |
| | msort | 185501 | 299900 | 552699 | 2143799 | 8850000 | 79860300 |
| | isort | 1001 | 35199 | 4075700 | 110743100 | 7225020001 | 724753641900 |



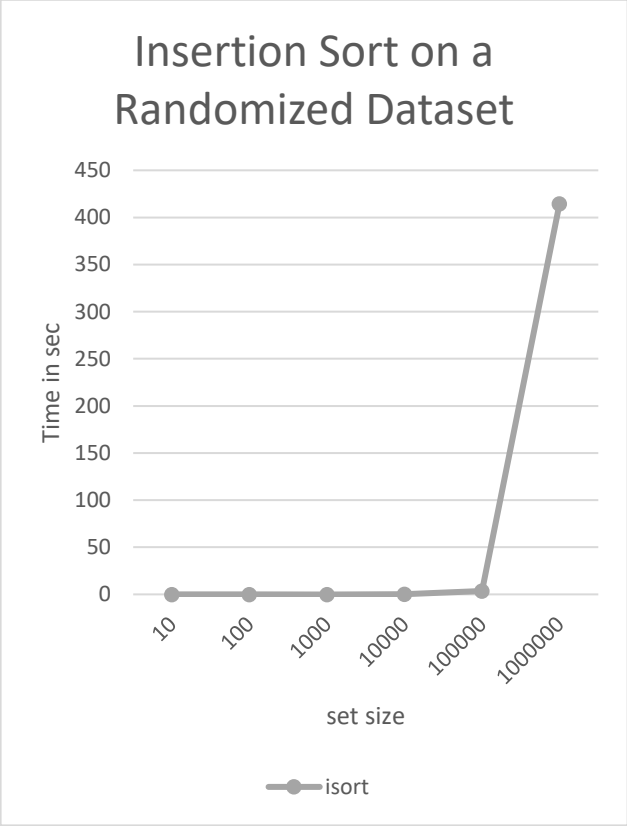

(I wasn't able to put insertion sort on a same chart with quick and merge sorts as there is a very big difference in time scale)

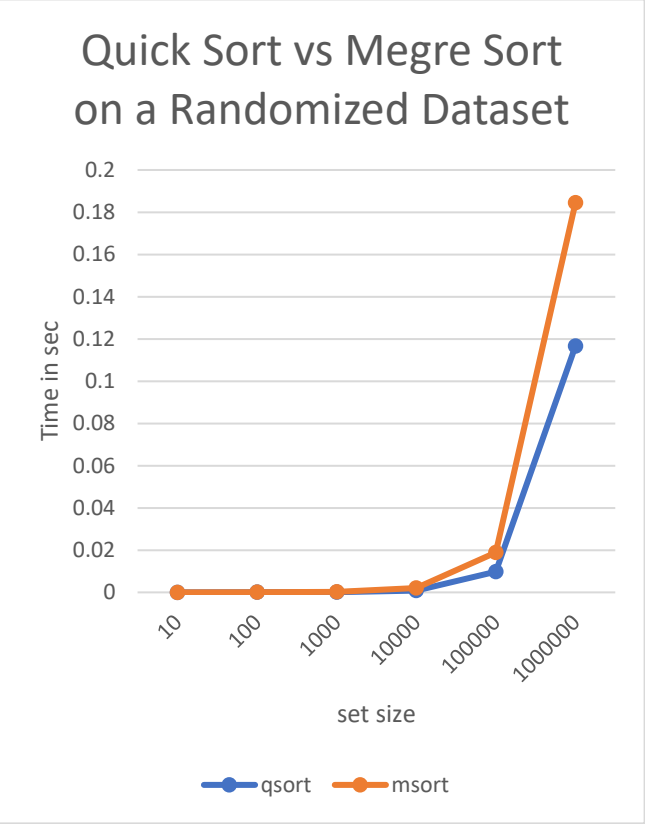
As it was expected with a descending dataset, an insertion sort had the worst average performance. A descending sorted array is the worst case for insertion sort, so the time complexity is $O(N^2)$. However, according to the results, insertion sort still took less time sorting small arrays (up to 100 elements) than quick and merge sorts. Again, in case of quick sort and merge sort, the factor that slows sorting smaller array is probably recursion. As it could be seen from the chart, quick and merge sorts had about a same performance on arrays with sizes up to 10000. The difference becomes distinguishable on 100000 and 1000000 size datasets. The quick sort performs better than merge sort. It is probably caused by the need of allocating a new temp array for an each recursive invoke of the merge sort.

3. Randomized dataset

Results:

| | size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|---|
| Time in nanoseconds | qsort | 1700 | 40800 | 100700 | 903300 | 9760900 | 116622900 |
| | msort | 2400 | 18200 | 183600 | 2093200 | 18918200 | 184433400 |
| | isort | 400 | 5600 | 380300 | 35124400 | 3601078999 | 4.14529E+11 |




The results of sorting randomized dataset are similar to a descending dataset. As with a descending dataset, an insertion sort had the worst average performance here. I would consider a randomized dataset an average case for all three sorts, which means the time complexity for insertion sort is $O(N^2)$, and for quick and merge sorts it's going to be O(NlogN). Nevertheless, an insertion sort outperforms quick and merge sorts on smaller scale arrays, and the reason for that is recursion. As with a descending dataset, quick and merge sort had about a same performance on arrays with sizes up to 10000. On larger datasets, the quick sort performs better that the merge sort, but the difference is not that big as with descending sorted datasets.

4. Comparing the results of each sorting algorithm

quick sort

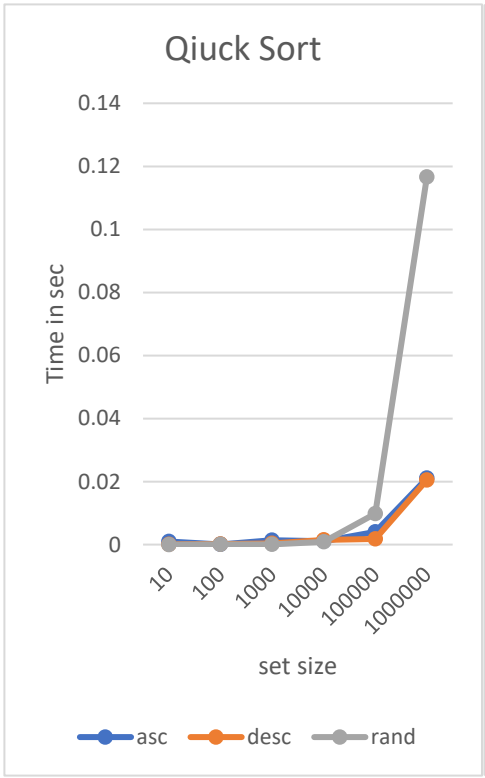
| size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|
| asc | 940900 | 38801 | 1411401 | 1050599 | 3988200 | 21025600 |
| desc | 77200 | 143199 | 393401 | 1454701 | 1775301 | 20496999 |
| rand | 1700 | 40800 | 100700 | 903300 | 9760900 | 116622900 |

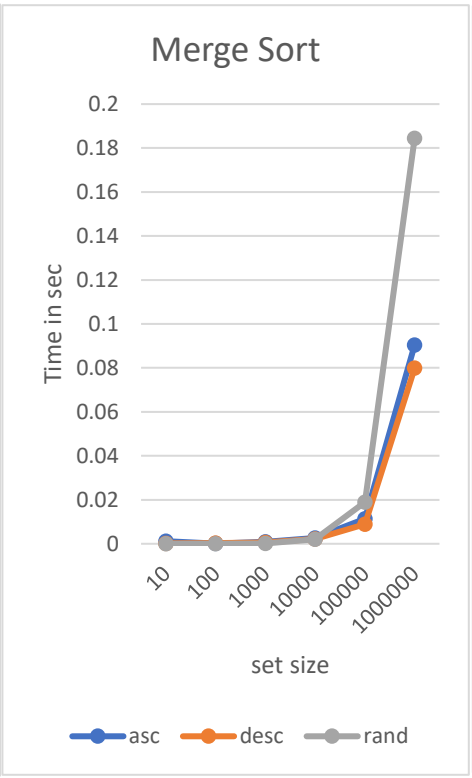
Merge sort

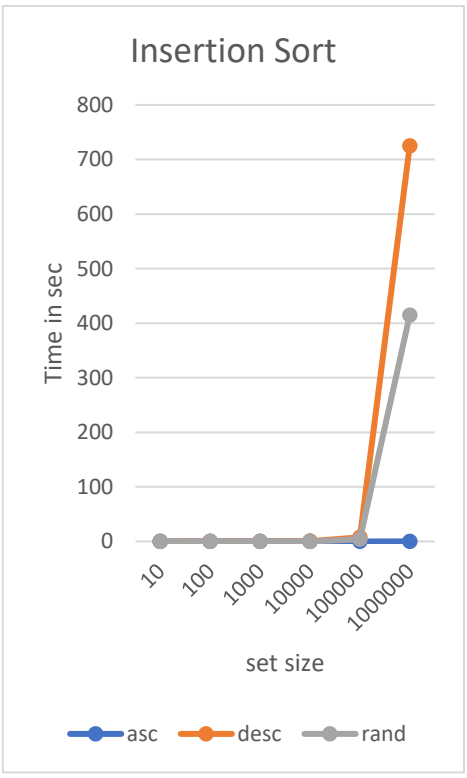
| size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|
| asc | 1119900 | 85801 | 826500 | 2661200 | 11414501 | 90392801 |
| desc | 185501 | 299900 | 552699 | 2143799 | 8850000 | 79860300 |
| rand | 2400 | 18200 | 183600 | 2093200 | 18918200 | 184433400 |

Insertion Sort

| size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|
| asc | 1643400 | 3900 | 43599 | 778100 | 1794800 | 4651200 |
| desc | 1001 | 35199 | 4075700 | 110743100 | 7225020001 | 7.24754E+11 |
| rand | 400 | 5600 | 380300 | 35124400 | 3601078999 | 4.14529E+11 |



It is interesting that there are similar patterns in quick sort and merge sort. In particular, both have relatively the same performance on ascending and descending sorted data. Both, quick and merge sorts, sort randomized dataset somewhat faster than sorted datasets with sizes up to 10000 elements. With larger arrays, sorting randomized data took more time than sorting sorted arrays. This fact has probably something to do with using recursion in those algorithms and maybe the nature of a randomized dataset, but at this point I’m struggling to explain that.

With the insertion sort, everything is pretty straightforward. An ascending array is the best case for it, so the runtimes are small. A descending dataset is the worst case for the insertion sort, and, as a result, it has the worst performance with the descending array. Sorting a randomized data is somewhere in the middle.