Uğur Erdem Seyfi
21801744
CS202-01

HOMEWORK 01

Solution for Question 1:

We are asked to trace the following sorting algorithms to sort the array [4, 8, 3, 7, 6, 2, 1, 5] into ascending order.

a) Insertion sort

Color Yellow stands for sorted part of the array.

Initial: 4, 8, 3, 7, 6, 2, 1, 5
Iteration 1: 4, 8, 3, 7, 6, 2, 1, 5
Iteration 2: 3, 4, 8, 7, 6, 2, 1, 5
Iteration 3: 3, 4, 7, 8, 6, 2, 1, 5
Iteration 4: 3, 4, 6, 7, 8, 2, 1, 5
Iteration 5: 2, 3, 4, 6, 7, 8, 1, 5
Iteration 6: 1, 2, 3, 4, 6, 7, 8, 5
Iteration 7: 1, 2, 3, 4, 5, 6, 7, 8

b) Selection sort

Color Yellow stands for sorted part of the array.
Color Light Purple stands for the minimum element of the unsorted (right) part of the array.

Initial: 4, 8, 3, 7, 6, 2, 1, 5
Iteration 1: 1, 8, 3, 7, 6, 2, 4, 5
Iteration 2: 1, 2, 3, 7, 6, 8, 4, 5
Iteration 3: 1, 2, 3, 7, 6, 8, 4, 5
Iteration 4: 1, 2, 3, 4, 6, 8, 7, 5
Iteration 5: 1, 2, 3, 4, 5, 8, 7, 6
Iteration 6: 1, 2, 3, 4, 5, 6, 7, 8
Iteration 7: 1, 2, 3, 4, 5, 6, 7, 8
Iteration 8: 1, 2, 3, 4, 5, 6, 7, 8

c) Bubble sort

Color Gold stands for the couples that might be swapped after each iteration.

Pass 01: 4, 8, 3, 7, 6, 2, 1, 5
Initial: 4, 8, 3, 7, 6, 2, 1, 5
Iteration 1: 4, 8, 3, 7, 6, 2, 1, 5

Iteration 2: 4, 3, 8, 7, 6, 2, 1, 5
Iteration 3: 4, 3, 7, 8, 6, 2, 1, 5
Iteration 4: 4, 3, 7, 6, 8, 2, 1, 5
Iteration 5: 4, 3, 7, 6, 2, 8, 1, 5
Iteration 6: 4, 3, 7, 6, 2, 1, 8, 5
Iteration 7: 4, 3, 7, 6, 2, 1, 5, 8

Pass 02: 4, 3, 7, 6, 2, 1, 5, 8
Initial: 4, 3, 7, 6, 2, 1, 5, 8
Iteration 1: 3, 4, 7, 6, 2, 1, 5, 8
Iteration 2: 3, 4, 7, 6, 2, 1, 5, 8
Iteration 3: 3, 4, 6, 7, 2, 1, 5, 8
Iteration 4: 3, 4, 6, 2, 7, 1, 5, 8
Iteration 5: 3, 4, 6, 2, 1, 7, 5, 8
Iteration 6: 3, 4, 6, 2, 1, 5, 7, 8
Iteration 7: 3, 4, 6, 2, 1, 5, 7, 8

Pass 03: 3, 4, 6, 2, 1, 5, 7, 8
Initial: 3, 4, 6, 2, 1, 5, 7, 8
Iteration 1: 3, 4, 6, 2, 1, 5, 7, 8
Iteration 2: 3, 4, 6, 2, 1, 5, 7, 8
Iteration 3: 3, 4, 2, 6, 1, 5, 7, 8
Iteration 4: 3, 4, 2, 1, 6, 5, 7, 8
Iteration 5: 3, 4, 2, 1, 5, 6, 7, 8
Iteration 6: 3, 4, 2, 1, 5, 6, 7, 8
Iteration 7: 3, 4, 2, 1, 5, 6, 7, 8

Pass 04: 3, 4, 2, 1, 5, 6, 7, 8
Initial: 3, 4, 2, 1, 5, 6, 7, 8
Iteration 1: 3, 4, 2, 1, 5, 6, 7, 8
Iteration 2: 3, 2, 4, 1, 5, 6, 7, 8
Iteration 3: 3, 2, 1, 4, 5, 6, 7, 8
Iteration 4: 3, 2, 1, 4, 5, 6, 7, 8
Iteration 5: 3, 2, 1, 4, 5, 6, 7, 8
Iteration 6: 3, 2, 1, 4, 5, 6, 7, 8
Iteration 7: 3, 2, 1, 4, 5, 6, 7, 8

Pass 05: 3, 2, 1, 4, 5, 6, 7, 8
Initial: 3, 2, 1, 4, 5, 6, 7, 8
Iteration 1: 2, 3, 1, 4, 5, 6, 7, 8
Iteration 2: 2, 1, 3, 4, 5, 6, 7, 8
Iteration 3: 2, 1, 3, 4, 5, 6, 7, 8
….
Iteration 7: 2, 1, 3, 4, 5, 6, 7, 8

Pass 06: 2, 1, 3, 4, 5, 6, 7, 8
Initial: 2, 1, 3, 4, 5, 6, 7, 8
Iteration 1: 1, 2, 3, 4, 5, 6, 7, 8
….
Iteration 7: 1, 2, 3, 4, 5, 6, 7, 8

Pass 07: 1, 2, 3, 4, 5, 6, 7, 8
Initial: 1, 2, 3, 4, 5, 6, 7, 8

….
Iteration 7: 1, 2, 3, 4, 5, 6, 7, 8

d) Merge sort

Color Gold stands for the subarray that is mergeSort is called for.
Color Light gray 1 stands for subarrays that are parameters of the merge function.
Color Light indigo 4 stands for the merged (and sorted) subarrays of the array.

Initial:	4, 8, 3, 7, 6, 2, 1, 5	
Step 1:	4, 8, 3, 7, 6, 2, 1, 5	mergeSort is called for left subarray
Step 2:	4, 8, 3, 7, 6, 2, 1, 5	mergeSort is called for left subarray
Step 3:	4, 8, 3, 7, 6, 2, 1, 5	mergeSort is called for left subarray
Step 4:	4, 8, 3, 7, 6, 2, 1, 5	mergeSort is called for right subarray
Step 5:	4, 8, 3, 7, 6, 2, 1, 5	merge is called for left & right subarrays
Step 6:	4, 8, 3, 7 6, 2, 1, 5	mergeSort is called for right subarray
Step 7:	4, 8, 3, 7, 6, 2, 1, 5	mergeSort is called for left subarray
Step 8:	4, 8, 3, 7, 6, 2, 1, 5	mergeSort is called for right subarray
Step 9:	4, 8, 3, 7, 6, 2, 1, 5	merge is called for left & right subarrays
Step 10:	4, 8, 3, 7, 6, 2, 1, 5	merge is called for left & right subarrays
Step 11:	3, 4, 7, 8, 6, 2, 1, 5	mergeSort is called for the right subarray
Step 12:	3, 4, 7, 8, 6, 2, 1, 5	mergeSort is called for the left subarray
Step 13:	3, 4, 7, 8, 6, 2, 1, 5	mergeSort is called for the left subarray
Step 14:	3, 4, 7, 8, 6, 2, 1, 5	mergeSort is called for the right subarray
Step 15:	3, 4, 7, 8, 6, 2, 1, 5	merge is called for left & right subarrays
Step 16:	3, 4, 7, 8, 2, 6, 1, 5	mergeSort is called for the right subarray
Step 17:	3, 4, 7, 8, 2, 6, 1, 5	mergeSort is called for the left subarray
Step 18:	3, 4, 7, 8, 2, 6, 1, 5	mergeSort is called for the left subarray
Step 19:	3, 4, 7, 8, 2, 6, 1, 5	merge is called for left & right subarrays
Step 20:	3, 4, 7, 8, 2, 6, 1, 5	merge is called for left & right subarrays
Step 21:	3, 4, 7, 8, 1, 2, 5, 6	merge is called for left & right subarrays
Final:	1, 2, 3, 4, 5, 6, 7, 8	

e) Quicksort

Color Red stands for choosen pivot.
Color Orange stands for j index (referred as firstUnknown in the slide) variable in the partition method.
Color Yellow stands for I index (referred as lastS1 in the slide) variable in the partition method.

Initial:	4, 8, 3, 7, 6, 2, 1, 5	
Step 1:	4, 8, 3, 7, 6, 2, 1, 5	**quickSort** for the whole (gray) part
Initial:	4, 8, 3, 7, 6, 2, 1, 5	pivot = 4, **partition** the array
Iteration 1:	4, 8, 3, 7, 6, 2, 1, 5	I = 0, j = 1
Iteration 2:	4, 8, 3, 7, 6, 2, 1, 5	I = 0, j = 2, **swap**(arr[++I], j)
Iteration 3:	4, 3, 8, 7, 6, 2, 1, 5	I = 1, j = 3
Iteration 4:	4, 3, 8, 7, 6, 2, 1, 5	I = 1, j = 4
Iteration 5:	4, 3, 8, 7, 6, 2, 1, 5	I = 1, j = 5, **swap**(arr[++I], j)
Iteration 6:	4, 3, 2, 7, 6, 8, 1, 5	I = 2, j = 6, **swap**(arr[++I], j)
Iteration 7:	4, 3, 2, 1, 6, 8, 7, 5	I = 3, j = 7
Final:	1, 3, 2, 4, 6, 8, 7, 5	pivot is placed into the correct place

Step 2: 1, 3, 2, 4, 6, 8, 7, 5 quickSort for the left (gray) part
Initial: 1, 3, 2 pivot = 1, **partition** the array
Iteration 1: 1, 3, 2 I = 0, j = 1
Iteration 2: 1, 3, 2 I = 0, j = 2
Final: 1, 3, 2 pivot is placed into the correct place

Step 3: 1, 3, 2, 4, 6, 8, 7, 5 quickSort for the right (gray) part
Initial: 3, 2 pivot = 3, **partition** the array
Iteration 1: 3, 2 I = 0, j = 1, **swap**(arr[++I], j)
Final: 2, 3 pivot is placed into the correct place

Step 4: 1, 2, 3, 4, 6, 8, 7, 5 quickSort for the left (gray) part

Step 5: 1, 2, 3, 4, 6, 8, 7, 5 quickSort for the right (gray) part
Initial: 6, 8, 7, 5 pivot = 6, **partition** the array
Iteration 1: 6, 8, 7, 5 I = 0, j = 1
Iteration 2: 6, 8, 7, 5 I = 0, j = 2
Iteration 3: 6, 8, 7, 5 I = 0, j = 3, **swap**(arr[++I], j)
5, 8, 7, 6
Final: 5, 6, 7, 8 pivot is placed into the correct place

Step 6: 1, 2, 3, 4, 5, 6, 7, 8 quickSort for the left (gray) part
Step 7: 1, 2, 3, 4, 5, 6, 7, 8 quickSort for the right (gray) part
Initial: 7, 8 pivot = 7, **partition** the array
Iteration 1: 7, 8 I = 0, j = 1
Final: 7. 8 pivot is placed into the correct place

Step 8: 1, 2, 3, 4, 5, 6, 7, 8 quickSort for the right (gray) part

Final: 1, 2, 3, 4, 5, 6, 7, 8

Solution for Question 2:

We are asked to write the recurrence equation for the time requirements of **mergesort** and **quicksort** algorithms in the worst case and solve them using ***repeated substitutions*** method.

a) Mergesort

Worst case for mergesort is the case where the number of comparisons is at its possible maximum value (which is 2k - 1, where k is the numer of elements in both subarrays).

If the the size of input array is 1 (n = 1), then obviously, T(n) = 1.
Otherwise, we have T(n) = 2T(n/2) + O(2k – 1 + 2k + 2)
= 2T(n/2) + O(4k + 1)
= 2T(n/2) + O(2n + 1)
= 2T(n/2) + O(n)

So we have T(n) = 2T(n/2) + O(n)
= 2(2T(n/4) + O(n/2)) + O(n)
= 2(2(2T(n/8) + O(n/4)) + O(n/2)) + O(n)

$$= 8T(n/8) + 4O(n/4) + 2O(n/2) + O(n)$$
…….
$$= 2^{(\log n)} T(1) + 2^{\log n} O(1) + 2^{(\log n - 1)} O(2) + 2^{(\log n - 2)} O(4) + \dots + O(n)$$
$$= n\,T(1) + O(n) + 2 * O(n/2) + 4 * O(n/4) + \dots + 2^{(\log n)} O(1)$$
$$= n + O(n) + 2 * O(n/2) + 4 * O(n/4) + \dots + n\,O(1)$$
$$= n + O(n) + O(n) + \dots + O(n)$$

since there are log(n) many O(n), we have:

$$= n + \log n\, O(n)$$
$$= O(n \log n)$$

b) Quicksort

The worstcase for quicksort is the case where all elements are already sorted in any (descending or ascending) order. Lets say our algorithm chooses the element at first index as pivot. Then after partitioning -which takes O(n) time complexity- only right part of the array is passed to the array. So we have:

For $n = 1$, $T(n) = 1$
Otherwise, $T(n) = O(n) + T(n-1)$

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

Sample output of the Program

2
!> EXPERIMENT FOR RANDOMLY GENERATED ARRAYS <!

!!! EXPERIMENT 1 WITH SIZE OF 25000!!!

- INSERTION SORT -
Comparison count: 311820393
Movement count: 311895391
Time passed: 0.5625 seconds.
- MERGE SORT -
Comparison count: 1330151
Movement count: 1941764
Time passed: 0 seconds.
- QUICK SORT -
Comparison count: 907039
Movement count: 1556463
Time passed: 0 seconds.
!!! EXPERIMENT 2 WITH SIZE OF 50000!!!

- INSERTION SORT -
Comparison count: 1244695143

Movement count: 1244845141
Time passed: 2.95312 seconds.
 - MERGE SORT -
Comparison count: 2837825
Movement count: 4122297
Time passed: 0 seconds.
 - QUICK SORT -
Comparison count: 1835479
Movement count: 2961703
Time passed: 0.015625 seconds.
 !!! EXPERIMENT 3 WITH SIZE OF 75000!!!

 - INSERTION SORT -
Comparison count: 2817749357
Movement count: 2817974355
Time passed: 5.40625 seconds.
 - MERGE SORT -
Comparison count: 4539011
Movement count: 6451282
Time passed: 0.015625 seconds.
 - QUICK SORT -
Comparison count: 3041139
Movement count: 5268361
Time passed: 0.015625 seconds.
 !!! EXPERIMENT 4 WITH SIZE OF 100000!!!

 - INSERTION SORT -
Comparison count: 5006461071
Movement count: 5006761069
Time passed: 10.1875 seconds.
 - MERGE SORT -
Comparison count: 6112281
Movement count: 8762917
Time passed: 0.03125 seconds.
 - QUICK SORT -
Comparison count: 4168971
Movement count: 6991925
Time passed: 0.015625 seconds.
 !!! EXPERIMENT 5 WITH SIZE OF 125000!!!

 - INSERTION SORT -
Comparison count: 7827148499
Movement count: 7827523497
Time passed: 14.2188 seconds.
 - MERGE SORT -
Comparison count: 7882351
Movement count: 11172952
Time passed: 0.015625 seconds.
 - QUICK SORT -
Comparison count: 5406003
Movement count: 9294729
Time passed: 0.03125 seconds.
 !> EXPERIMENT FOR DESCENDING ARRAYS <!

 !!! EXPERIMENT 1 WITH SIZE OF 25000!!!

 - INSERTION SORT -
Comparison count: 624999999

Movement count: 625074997
Time passed: 1.17188 seconds.
 - MERGE SORT -
Comparison count: 1299671
Movement count: 1926524
Time passed: 0 seconds.
 - QUICK SORT -
Comparison count: 625024999
Movement count: 937712491
Time passed: 1.79688 seconds.
 !!! EXPERIMENT 2 WITH SIZE OF 50000!!!

 - INSERTION SORT -
Comparison count: 2499999999
Movement count: 2500149997
Time passed: 4.76562 seconds.
 - MERGE SORT -
Comparison count: 2640577
Movement count: 4023673
Time passed: 0 seconds.
 - QUICK SORT -
Comparison count: 2500049999
Movement count: 3750424991
Time passed: 7.4375 seconds.
 !!! EXPERIMENT 3 WITH SIZE OF 75000!!!

 - INSERTION SORT -
Comparison count: 5624999999
Movement count: 5625224997
Time passed: 11.0156 seconds.
 - MERGE SORT -
Comparison count: 3985045
Movement count: 6174299
Time passed: 0.015625 seconds.
 - QUICK SORT -
Comparison count: 5625074999
Movement count: 8438137491
Time passed: 17.2656 seconds.
 !!! EXPERIMENT 4 WITH SIZE OF 100000!!!

 - INSERTION SORT -
Comparison count: 9999999999
Movement count: 10000299997
Time passed: 21.7188 seconds.
 - MERGE SORT -
Comparison count: 5374529
Movement count: 8394041
Time passed: 0.015625 seconds.
 - QUICK SORT -
Comparison count: 10000099999
Movement count: 15000849991
Time passed: 31.7656 seconds.
 !!! EXPERIMENT 5 WITH SIZE OF 125000!!!

 - INSERTION SORT -
Comparison count: 15624999999
Movement count: 15625374997
Time passed: 29.9844 seconds.

- MERGE SORT -
Comparison count: 6764405
Movement count: 10613979
Time passed: 0.015625 seconds.
- QUICK SORT -
Comparison count: 15625124999
Movement count: 23438562491
Time passed: 46.2344 seconds.
!> EXPERIMENT FOR ASCENDING ARRAYS <!

!!! EXPERIMENT 1 WITH SIZE OF 25000!!!

- INSERTION SORT -
Comparison count: 24999
Movement count: 99997
Time passed: 0 seconds.
- MERGE SORT -
Comparison count: 1316735
Movement count: 1935056
Time passed: 0.015625 seconds.
- QUICK SORT -
Comparison count: 625024999
Movement count: 312712491
Time passed: 0.796875 seconds.
!!! EXPERIMENT 2 WITH SIZE OF 50000!!!

- INSERTION SORT -
Comparison count: 49999
Movement count: 199997
Time passed: 0 seconds.
- MERGE SORT -
Comparison count: 2702409
Movement count: 4054589
Time passed: 0 seconds.
- QUICK SORT -
Comparison count: 2500049999
Movement count: 1250424991
Time passed: 3.45312 seconds.
!!! EXPERIMENT 3 WITH SIZE OF 75000!!!

- INSERTION SORT -
Comparison count: 74999
Movement count: 299997
Time passed: 0 seconds.
- MERGE SORT -
Comparison count: 4053813
Movement count: 6208683
Time passed: 0.015625 seconds.
- QUICK SORT -
Comparison count: 5625074999
Movement count: 2813137491
Time passed: 7.98438 seconds.
!!! EXPERIMENT 4 WITH SIZE OF 100000!!!

- INSERTION SORT -
Comparison count: 99999
Movement count: 399997
Time passed: 0 seconds.

- MERGE SORT -
Comparison count: 5475241
Movement count: 8444397
Time passed: 0.015625 seconds.
- QUICK SORT -
Comparison count: 10000099999
Movement count: 5000849991
Time passed: 13.2656 seconds.
!!! EXPERIMENT 5 WITH SIZE OF 125000!!!

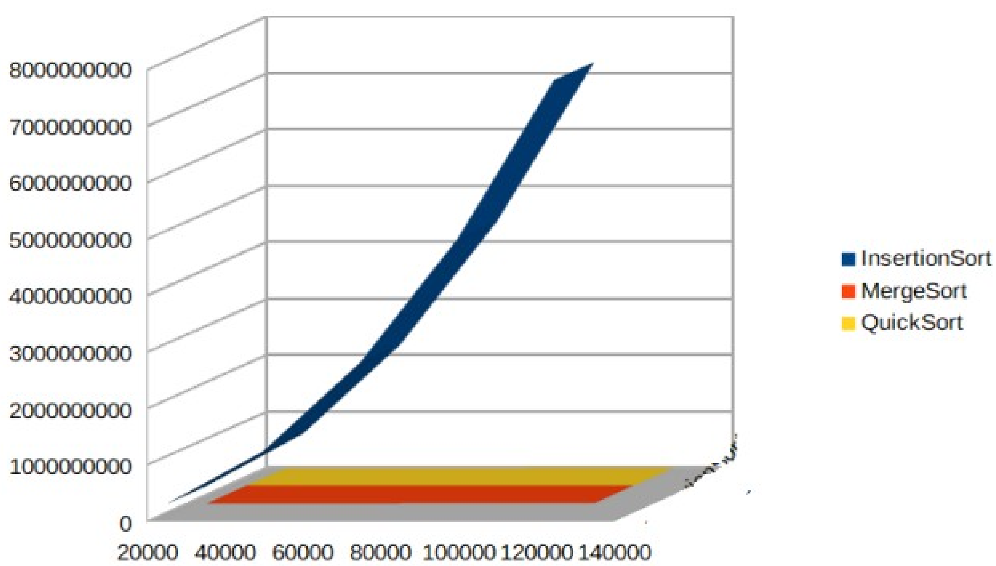
- INSERTION SORT -
Comparison count: 124999
Movement count: 499997
Time passed: 0 seconds.
- MERGE SORT -
Comparison count: 6835675
Movement count: 10649614
Time passed: 0.015625 seconds.
- QUICK SORT -
Comparison count: 15625124999
Movement count: 7813562491
Time passed: 21.625 seconds.

Graphs

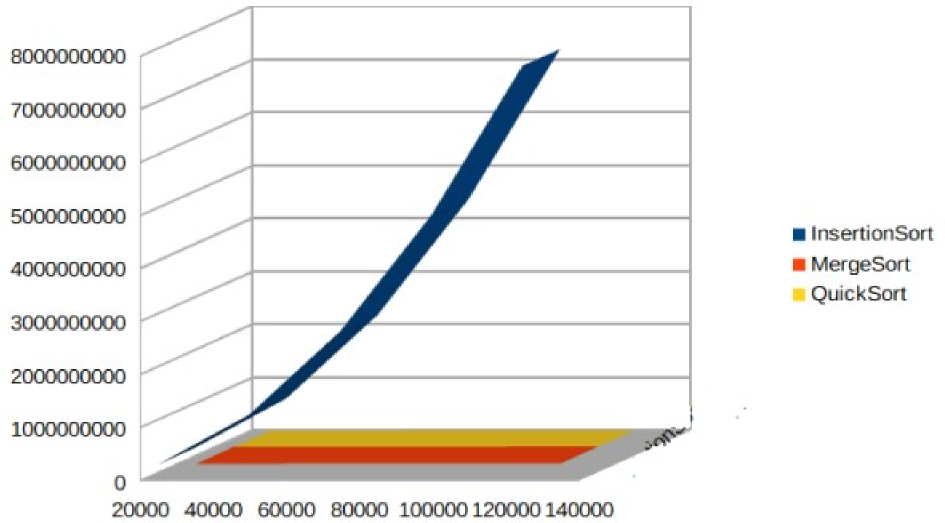
For some of the graphs below, I used 3D scattering since when lines are too close only one of the lines were appearing.

Graphs for arrays that are <u>randomly</u> created

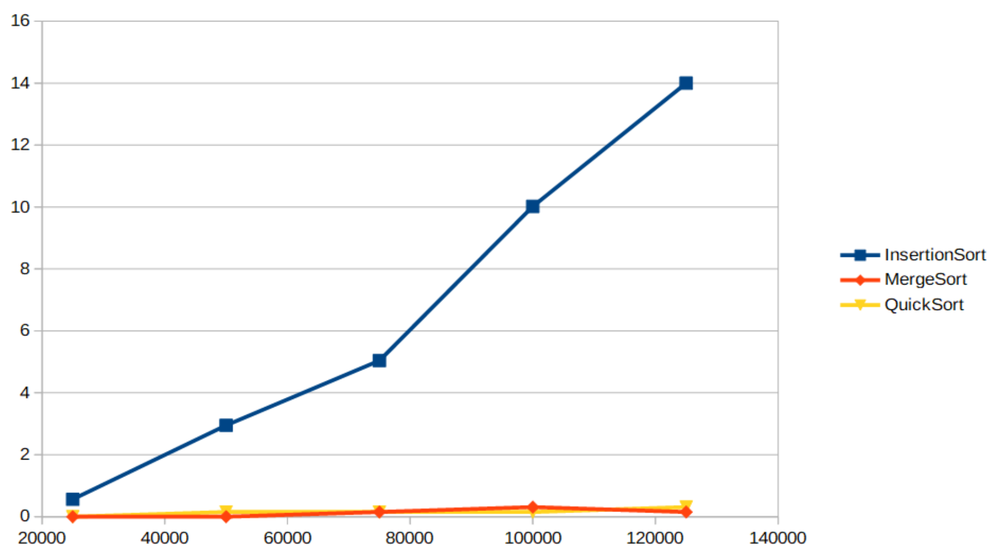
Comparison number graph:



Movement graph:

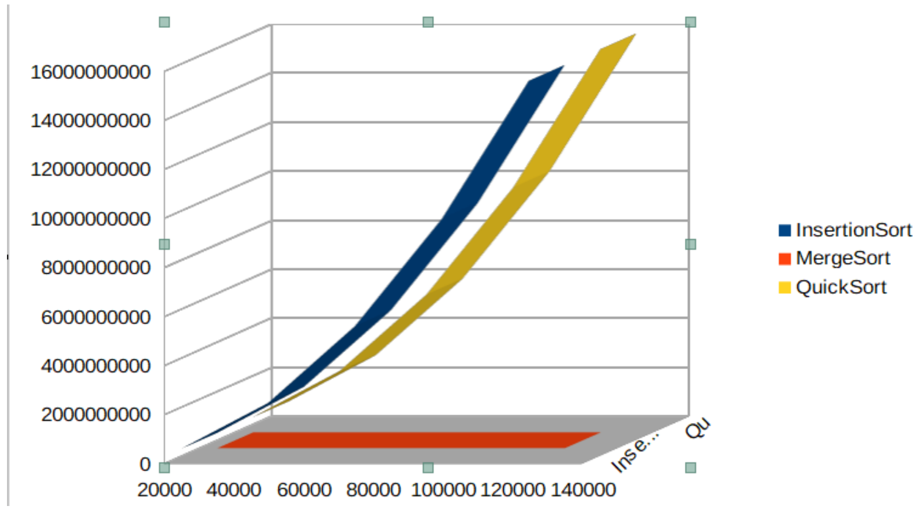


Time graph:

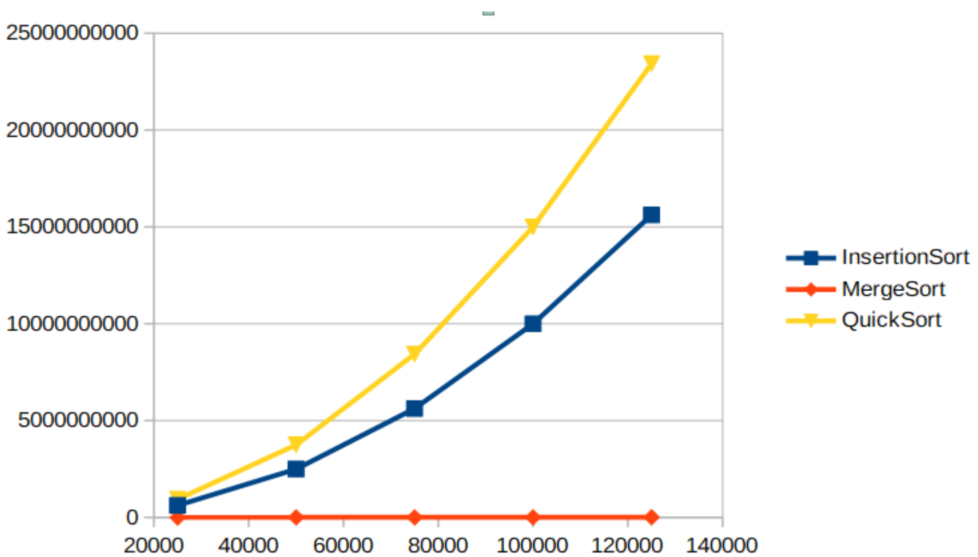


Graphs for arrays that are in <u>descending</u> order

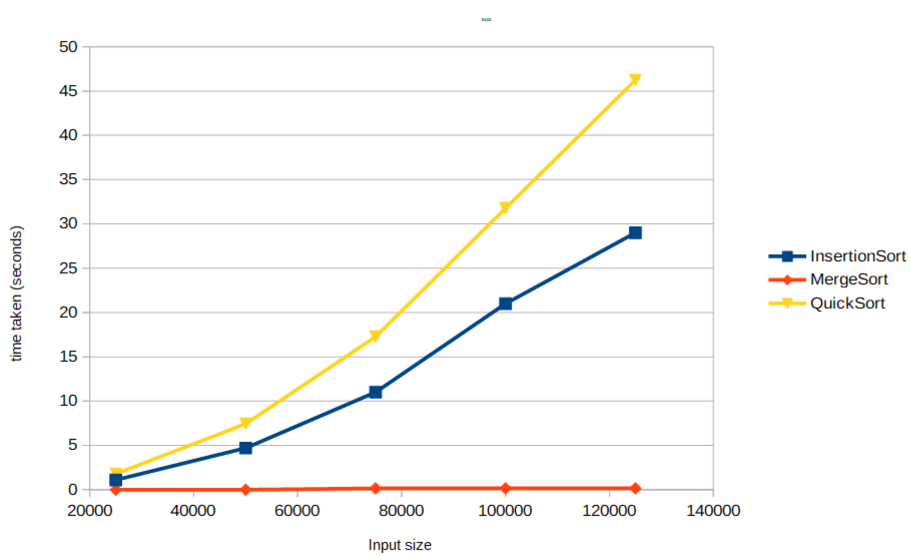
Comparison number graph:



Movement number graph:

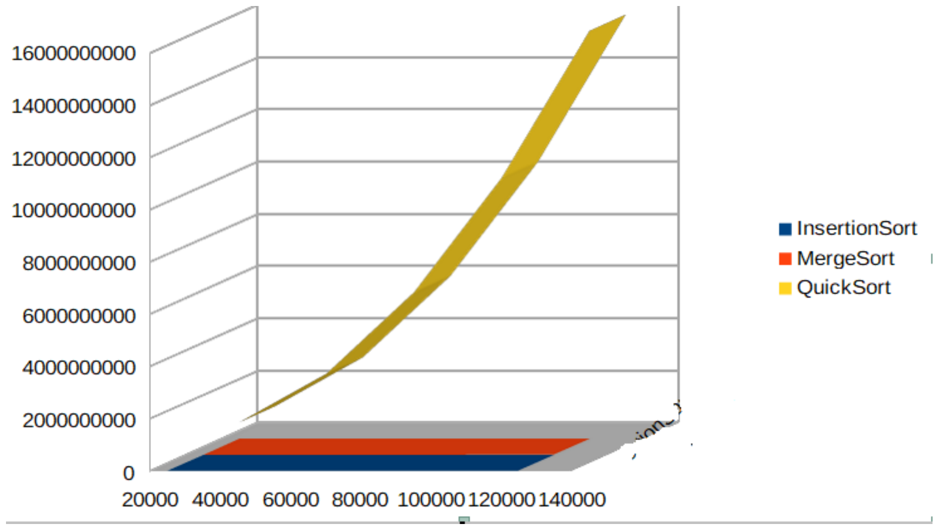


Time elapse graph:

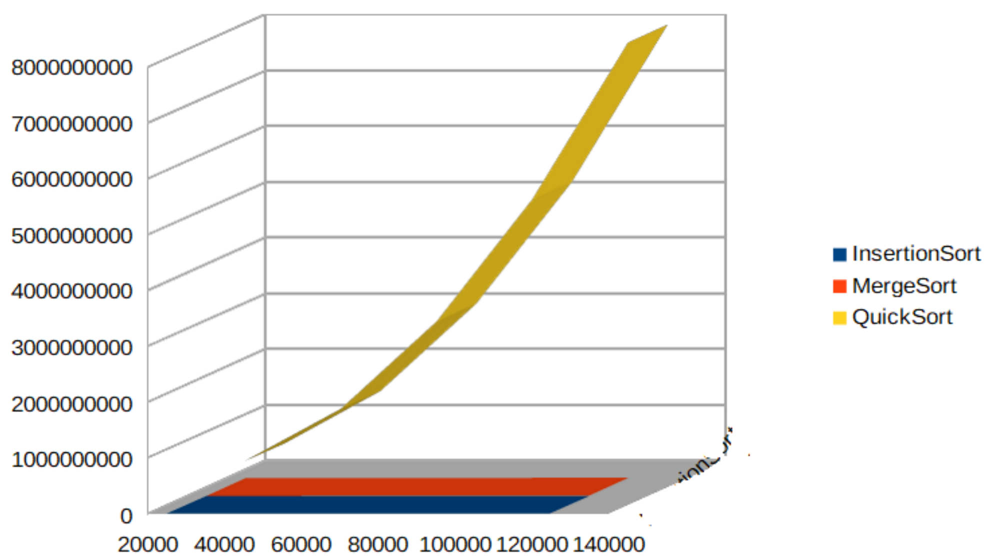


Graphs for arrays that are in ascending order

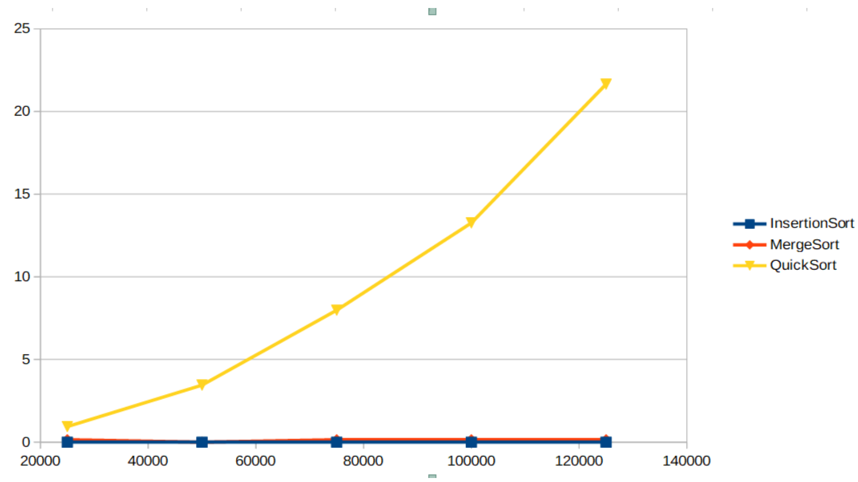
Comparison number graph:



Movement number graph:



Time elapse graph:



Question 4: Interpretation

Fortunately, there was not any conflicts between my expectations based on my theoretical knowledge and the experiments that I have made.

For randomly distributed arrays, both mergesort and quicksort go side by side each other and outperform the insertion sort.

Although for randomly generated arrays quicksort was slightly better both in terms of comparison and in terms of move numbers, their lines seemed to be very close to each other due to insertion sort's inefficient comparison and move results.

For the ascending and descending sorting of the arrays, quicksort is very inefficient and its graph seems to be at O(n^2) complexity as I expected.

Unlike quicksort and insertion sort, mergesort was more consistent in its results in different situations. According to the results that I get, mergesort seems to be working at O(n logn) complexity whether it is the worst case or not.

For ascending arrays, insertion sort is very fast and runs with O(n) complexity as expected. For randomly created and descending arrays though, its graph seems to be at O(n^2) complexity (considering the average of its move, comparison and time elapse graphs).

Overall, my expectations match with the results I get.