CS 202 Homework 1

Question 1a [10 points]

Sort the functions below in the increasing order of their asymptotic complexity.

1. $f_1(n) = 10^n$
2. $f_4(n) = \log n$;
3. $f_5(n) = n^{0:0001}$;
4. $f_3(n) = \sqrt{n}$;
5. $f_2(n) = n$;
6. $f_9(n) = \log(n!)$;
7. $f_6(n) = n \log n$;
8. $f_7(n) = 2^n$;
9. $f_8(n) = n!$;
10. $f_{10}(n) = n^n$;

Question 1b [10 points]

Express the running time complexity (using asymptotic notation) of each loop separately. Show all the steps clearly.

```
// Loop A - 3 points
for (i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
                sum = sum + 1;
```

The first loop runs from 1 to n (up to equal n) so it will run n times through this loop. Therefore the first loop is O(n);

The second loop runs from 1 to i. The second loop will run i times. It has a complexity of O(i).

The third line is a constant time operation (addition + assignment to the variable). Here the complexity is O(1).

To see the complexity of the overall execution we simply have to determine how many times the constant operation in the third line is run. It will run i times at each loop. However i is changing at each iteration of the first loop. So the number of times the third line is executed is given below:

NoRunTimes = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + n-1 + n

The addition of the number of times the constant operation is executed yields:

NoRunTimes = n*(n-1)/2.

Therefore the complexity of the overall execution is $\mathbf{O(n^2)}$.

```
// Loop B - 3 points
i = 1;
while (i < n) {
            for (j = 1; j <= i; j++)
                        sum = sum + 1;
            i *= 2;
}
```

The while loop is executed n-1 times, since i starts from 1 and goes up to n(not inclusive). This means that the complexity of this loop is O(n-1) which is O(n).

The for loop is executed i times each time the while loop is run. So each individual for loop has a complexity of O(i).

The third line and fourth line represent constant time operations. This means each time they will be run they will consume O(1) time.

To determine the overall execution complexity we should determine what is the number of times the third line gets executed. The number of run times is given below:

NoRunTimes = 1 + 2 + 4 + 8 +16 ...... + 2^k.

Here 2^k represents a number for which 2^{k+1} >= n. For ease of computation we will say 2^{k+1}= n. The resulting NoRunTimes has k additive factors which means that all loops will be finished in O(k) time. From the above assertions we have k = log(n) -1, which means that the overall execution will be **O(logn).**

```
// Loop C - 4 points
i = 1;
while (i < n) {
        j = 1;
        while (j < i*i) {
          for (k = 1; k <= n; k++)
               sum = sum + 1;
          j *= 2;
         }
          i *= 2;
}
```

The first while loop will run logn number of times because i gets incremented by a multiplication with a scalar (it experiences an exponential increase).

The second while loop will run log(i^2) which is 2*(log(i)), which means that we are executing j for 2*log(logn) which is upper bounded by logn. We can estimate the runtime of this part to be O(logn).

The third loop will run n number of times so it is of order n, O(n).

The sixth line represents a constant time execution of O(i).

Therefore the overall execution is **O(n$(logn)^2$).**

Question 1c [5 points]

Note that in the following we have taken the approximation that n = 2^k for simplicity.

The recurrence relation for MergeSort is given as follows:

T(n) = T(n/2) + T (n/2) + O(n)

- ⇨ T(n) = 2*T(n/2) + O(n)
- ⇨ T(n/2) = 2* T(n/4) + O(n/2), T(n) = 2*2*T(n/4) + O(n)
- ⇨ T(n/4) = 2* T(n/8) + O(n/4), T(n) = 2*2*2*T(n/4) + O(n)
- ⇨ T(n/8) = 2* T(n/16) + O(n/8), T(n) = 2*2*2*2*T(n/8) + O(n)
- ⇨ ....
- ⇨ ....
- ⇨ T(n) = 2^{k+1}*T(n/2^k) + k O(n)
- ⇨ T(n) = 2*T(1) + log(n)O(n)
- ⇨ T(n) = O(nlogn)

The recurrence relation for QuickSort is given as follows:

Since the worst case for quicksort means that we sort only 1 item at a time then every time we have n-1 left to be sorted and we have spent a traversal of O(n) in the proccess

T(n) = T(n-1) + O(n)

- ⇨ T(n-1) = T(n-2) + O(n), T(n) = T(n-2) + 2* O(n)
- ⇨ T(n-2) = T(n-3) + O(n), T(n) = T(n-3) + 3* O(n)
- ⇨ T(n-4) = T(n-4) + O(n), T(n) = T(n-4) + 4 * O(n)
- ⇨ ....
- ⇨ ....
- ⇨ T(n) = T(1) + n *O(n)
- ⇨ T(n) = O(n^2)

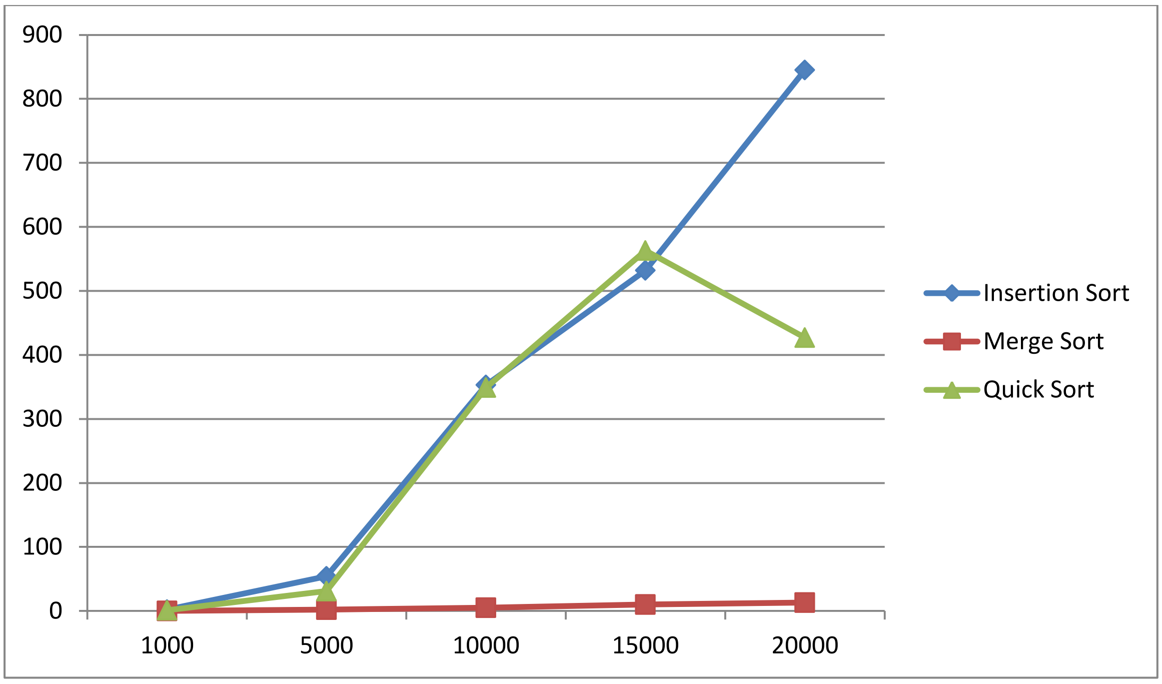
Question 3 [25 points]

In the following the table of results and the graphs of these results are given.

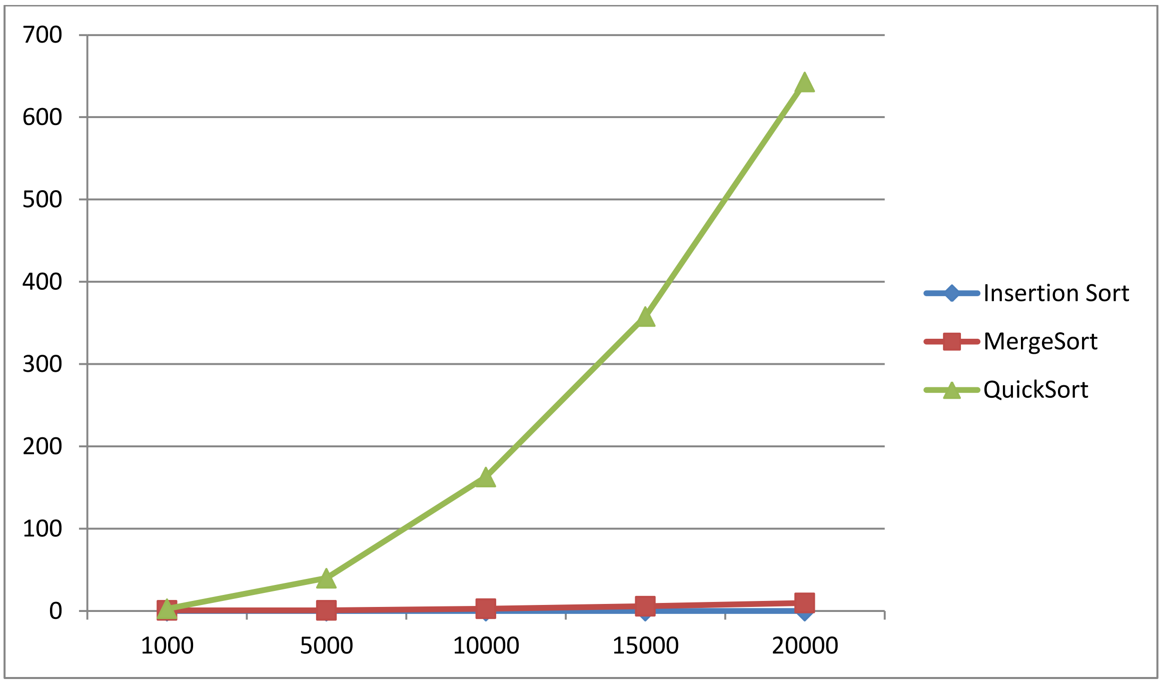
Table1.0 Sorting Algorithms Performance Table

Array	Elapsed Time (in milliseconds)			Number of Comparisons			Number of Data Moves		
	Insertion Sort	MergeSort	QuickSort	Insertion Sort	MergeSort	QuickSort	Insertion Sort	MergeSort	QuickSort
R1K	2	1	1	249144	8715	109299	250143	19952	303985
R10K	209	4	210	25024152	120376	18207386	25034151	267232	54529111
R20K	782	11	434	99926434	260806	37659081	99946433	574464	112683322
A1k	0	0	1	999	5044	500499	1998	19952	6997
A10K	0	3	157	9999	69008	50004999	19998	267232	69997
A20K	0	10	674	19999	148016	200009999	39998	574464	139997
D1K	4	0	-	500499	4932	-	501498	19952	-
D10K	448	4	-	50004999	64608	-	50014998	267232	-
D20K	1568	12	-	200009999	139216	-	200029998	574464	-

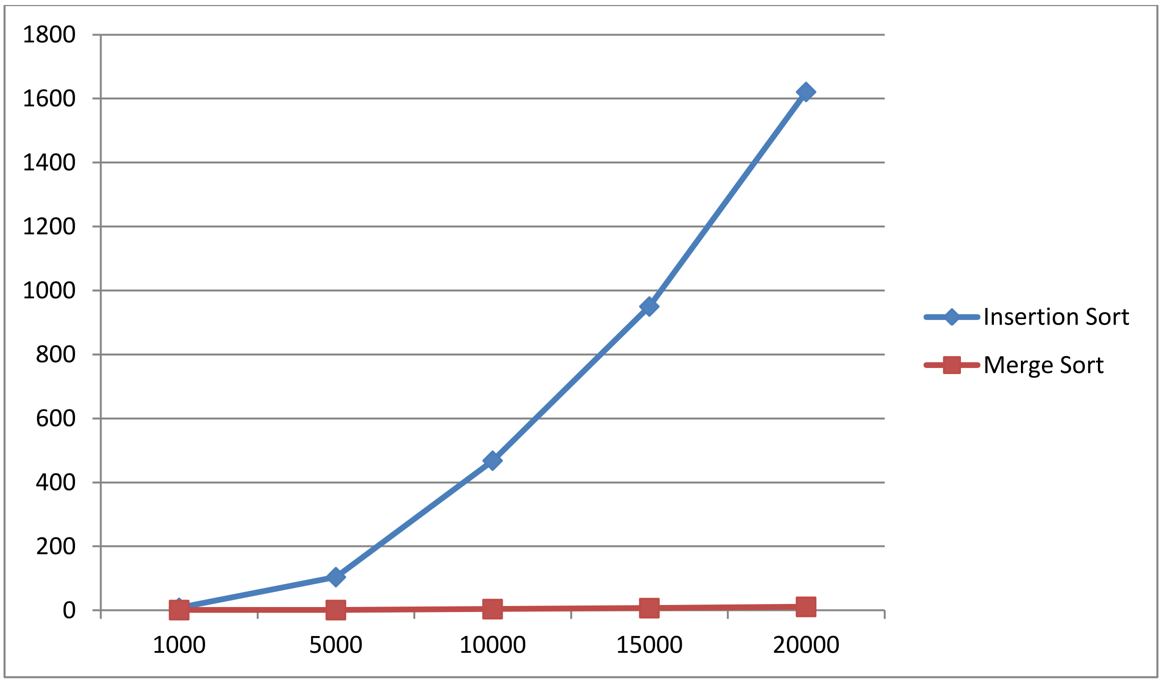
1.0 Graph of Random Integers Array



2.0 Graph of Ascending Integer Array



3.0 Graph of Descending Integer Array



Interpretation of the results

1. In the random integer array execution the sorting algorithms behave as in theory. Insertion Sort seems to tend to an execution of O(n^2) whereas mergeSort is much slower and tends to O(nlogn). However depending on the execution the quick Sort seems to be in between the running times of the two other sorting algorithms. This was to be expected since for Quick sort the worst case is O(n^2) and the best case is O(nlogn).
2. In the ascending integer dataset the insertion and merge sort perform better than quick sort. In this case quickSort approaches its worst case running time since all the partitions will be of size 1 and n-1. In a similar way we except the same thing to happen to the descending array case as well where quickSort unlike the other two sorting methodologies loses its performance.
3. In the third graph quicksort does not have any data because my computer was not able to perform the quicksort for the array sizes declared in there. I was able to create an array up until size 993. Anything more than that made my program crash and not execute properly.
4. The number of comparisons in quicksort is relatively higher, which was to be excepted since the quicksort algorithm will partition the array and then try to work around its parts.
5. The number of moves however seems to vary by the input type. In this way both insertion sort and quicksort will make a large number of moves when the data is in an ascending fashion.

When should insertion sort algorithm be preferred over merge sort and quick sort algorithms?

Insertion sort might be preferred when the data is already in a sorted in an ascending order since this way we will have less data moves within the array. In the same way the execution time is also low since there is not much work done by the algorithm. Insertion sort might also be reasonable for relatively small sizes of the array, since the implementation of insertion sort is straightforward and easy unlike mergesort and quicksort which require a recursive solution. It might also be advantageous when compared to mergesort since mergesort will require an additional array, and when you are running out of space (perhaps other computations and programs might be using) the allocation of twice the size of the original array might turn out to be overkill.

When should merge sort algorithm be preferred over quick sort algorithm?

Merge Sort algorithm is preferred over quick sort algorithm in cases when the array that is to be sorted is already sorted by itself in some way. Merge Sort will ensure that the run time of any array will be O(nlogn) in the worst case. However the quick sort does not have this property. Quicksort will resolve to O(n^2) for the case when the array is already sorted since the algorithm will partition the array into sizes of 1 and n-1 each time.