Question 1

a-) Sorting Functions

$$O(n^n) > O(n!) > O(2^n) > O(nlogn) = O(log(n!)) > O(n) > O(n^{1/2}) > O(n^{0.0001}) > O(logn) > O(10^\pi)$$

b-) Time Complexity

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

Times
$n+1$
$(n+2)*(n+3)/2-1$
$n*(n+1)/2$

Total : $n+1+(n+2)*(n+3)/2-1+n*(n+1)/2 = n^2+3n+3$
Complexity : ***$O(n^2)$***

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

Times
1
$log(n)+1$
$log(n)+1+2^{log(n)+1}-1$
$2^{log(n)+1}-1$
$log(n)$

Total :$1+logn+1+logn+1+2^{log(n)+1}-1+2^{log(n)+1}-1+logn = 1+log(n^3)+2^{log(4n)} = 4n+log(n^3)+1$
Complexity : ***$O(n)$***

```
// 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;
}
```

Times
1
$log(n)+1$
$log(n)$
$(logn*(logn+1))/2$
$(n+1)*((logn*(logn-1))/2)$
$n*((logn*(logn-1))/2)$
$(logn*(logn-1))/2$
$logn$

Total : $1+logn+1+logn+(logn*(logn+1))/2+(n+1)*(logn*(logn-1))/2+n*((logn*(logn-1))/2)+(logn*(logn-1))/2+logn = (2n+4)logn+(2n+2)log^2(n)+2$
Complexity : ***$O(nlog^2(n))$***

c-) Recurrence Relations

Merge Sort

$T(n) = 2T(n/2)+O(n)+O(1)$
$T(n) = 4T(n/4)+2O(n/2)+O(n)$
$T(n) = 2^kT(n/2^k)+kO(n)$
$T(n) = nT(1)+log(n)O(n)$
T(n) =O(nlogn)

Quick Sort

$T(n) = T(n-1)+nO(1)$
$T(n-1) = T(n-2)+(n-1)O(1)$
$T(2) = T(2)+2O(1)$
$T(1) = T(1)+1O(1)$
$T(n) = nO(1)+(n-1)O(1)+...+2O(1)+O(1)$
$$T(n) = \frac{n(n+1)}{2}O(1)$$
T(n) =O($\mathbf{n^2}$)

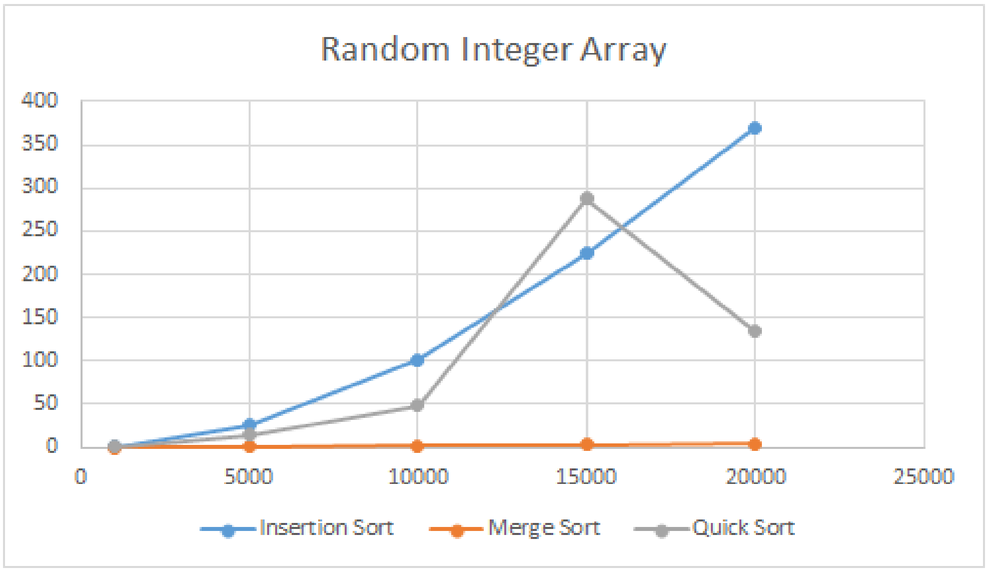
Question 3

Table 1: Sorting Algorithms Performance Table

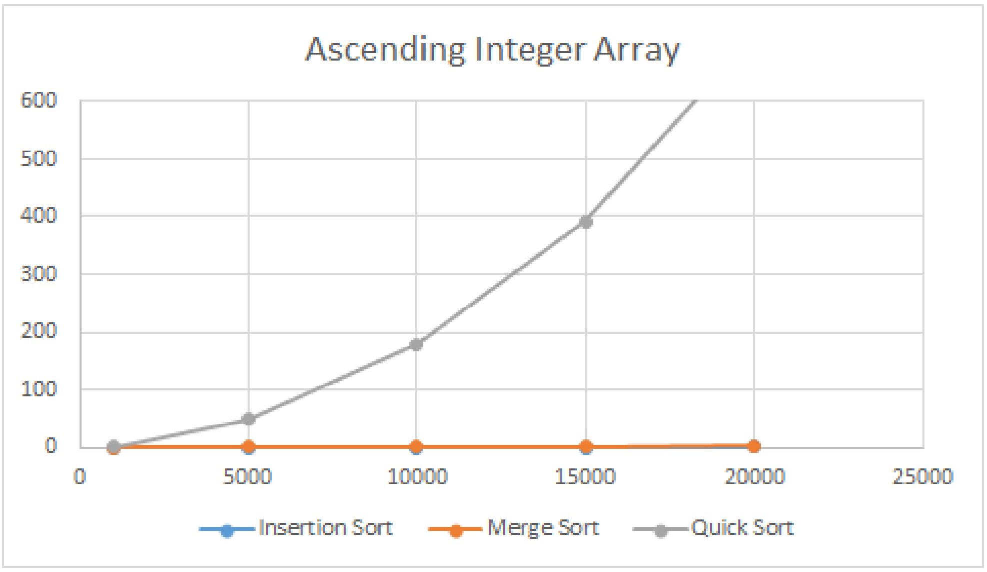
	Elapsed Time (in milliseconds)			Number of Comparisons			Number of Data Moves		
Array	Insertion Sort	Merge Sort	Quick Sort	Insertion Sort	Merge Sort	Quick Sort	Insertion Sort	Merge Sort	Quick Sort
R1K	0.001131	0.000152	0.000693	250160	4398	474	250160	19952	4407
R10K	0.101326	0.002076	0.048523	24837527	61263	31197	24837527	267232	123555
R20K	0.37043	0.003834	0.134473	99796026	132574	39825	99796026	574464	179442
A1K	0.000005	0.000132	0.001767	999	5044	249500	999	19952	75000
A10K	0.000079	0.001061	0.162365	9999	69008	24995000	9999	267232	75000000
A20K	0.000114	0.002254	0.652358	19999	148016	99990000	19999	574464	300000000
D1K	0.001888	0.000078	0.001327	500499	4932	499500	500499	19952	2997
D10K	0.192748	0.001033	0.142652	50004999	64608	49995000	50004999	267232	29997
D20K	0.761946	0.002183	0.53426	200009999	139216	199990000	200009999	574464	59997

This table shows some statistics of the most common sorting algorithms. 3 types of array are used in the experiment; array with random integers, array with ascending integers and array with descending integers. Also each experiment is done with 3 different sized arrays. Insertion Sort and Quick Sort are affected by the condition of the array however Merge Sort is not affected.

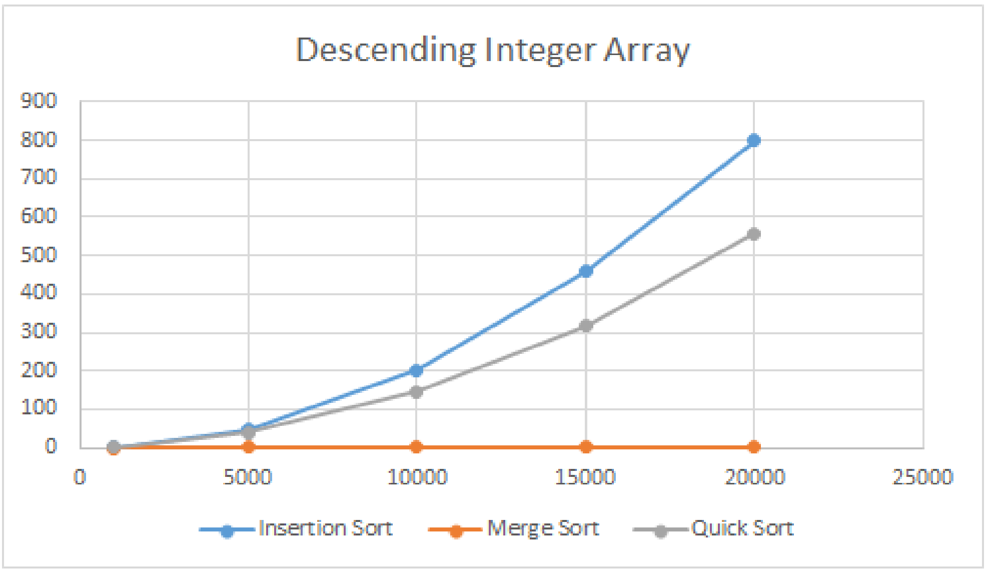
Following graphs are the running times of the sorting algorithms (Insertion Sort, Merge Sort, Quick Sort) in different conditions.



This is the graph of the sort of random integer arrays with 3 different sorting algorithm. When array is completely random, it is assumed that this is the average case for sorting algorithms theoretically. However, there are also other factors which affect the algorithm's actual run time, for instance choosing the pivot is really important in the Quick Sort. In the graph, Insertion Sort and Merge Sort behave as they are supposed to be but Quick Sort has similar run times with Insertion Sort. Both of their worst case time complexity is $O(n^2)$ and this case is the worst case for the Quick Sort and it is stemmed from the selection of the pivot. If proper pivot was selected, Quick Sort could have the similar run times with Merge Sort because both of their average case time complexity is $O(nlogn)$.



This is the graph of the sort of ascending integer arrays with 3 different sorting algorithm. Ascending integer arrays are the worst cases for the algorithms like Quick Sort. Because we select always the last element as pivot, in each partition $nsized$ array is divided into two parts sized $n-1$ and 1. This partition repeats n times therefore Quick Sort have $O(nlogn)$ time complexity if the array is ascending. For the Insertion sort, because sorted array is the best case for it, it is time complexity is $O(n)$. Merge Sort is always has $O(nlogn)$ complexity.



This is the graph of the sort of descending integer arrays with 3 different sorting algorithm. As like ascending integer arrays, descending integer arrays are also the worst cases for the algorithms like Quick Sort. Because we select always the last element as pivot, in each partition $nsized$ array is divided into two parts sized $n-1$ and 1. This partition repeats n times therefore Quick Sort have $O(nlogn)$ time complexity if the array is ascending. For the Insertion sort, it is the worst case because in every iteration the last element should go to the beginning of the unsorted list and this operation is done by n times, it is time complexity is $O(n^2)$. Merge Sort is always has $O(nlogn)$ complexity.

These algorithms show that state of the array, different implementations of them and size of the array could affect the behaviour of the algorithms. Especially, there are many factors in Quick Sort like pivot selection, partitioning etc. We see that when array is in ascending order, comparing to descending order state, number of the data moves get higher. This means Quick Sort algorithm is inefficient when the array is sorted. Nonetheless; behaviour of the Merge Sort algorithm is not affected by the state of the array. Also it has specific implementation which means implementation does not change its efficiency therefore Merge Sort is the most consistent sorting algorithm compared to the others.

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

–As we see from the results, when array is in sorted state(ascending integer array) or partially sorted, Insertion Sort is the best option. It has $O(n)$ (linear) time complexity when array is sorted. It is preffered to Merge Sort because it does not require extra memory.

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

–Merge Sort is better than Quick Sort when the state of the array is unknown. Quick Sort's running time could be chaged if pivot is selected different. Therefore Merge Sort is more consistent than the Quick Sort, its worst,best,average cases have same complexity $O(nlogn)$. Also Merge Sort guarantees less comparisons than the Quick Sort. Despite of its memory requirement, Merge Sort is commonly used because of its consistency.