

## Assignment 1 – Algorithm Efficiency and Sorting

### Question 1

This question will contain the tracing of 5 different sorting algorithms. First, a summary of the algorithm will be provided. Then, the outputs of the array at each pass and loop step will be observed. The sorted elements of the array will be colored in **blue**. The next element to be sorted will be colored in **red**. For each sorting algorithm trace, the array [4, 8, 3, 7, 6, 2, 1, 5] will be used.

#### a) Insertion Sort

As the name suggests, this algorithm sequentially inserts the item from the unsorted part of the array to the sorted part of the array. At the start, the first element of the array is assumed to be the sorted part of the array, hence the outer for loop starts with the index 1 (unsorted = 1). This value continuously represents the first item of the unsorted part of the list. Another variable, loc, is created to traverse the sorted part of the array. The element which is the first of the unsorted part also is assigned to the variable nextItem. The inner for loop, with the help of the variable loc, traverses the sorted array (by decreasing loc by one at each turn) and shifts each element (right for ascending order) and the loop continues as long as the condition is met (ex: the right element being bigger than nextItem for ascending order). After the inner loop finishes execution, the nextItem is assigned to wherever the loc variable points to. Hence, the algorithm inserts the first item of the unsorted part of the array to the desirable position in the sorted part. This way the whole array gets sorted. The code for the Insertion Sort algorithm can be found below.

```
void insertionSort(DataType theArray[], int n) {  
  
    for (int unsorted = 1; unsorted < n; ++unsorted) {  
  
        DataType nextItem = theArray[unsorted];  
        int loc = unsorted;  
  
        for ( ; (loc > 0) && (theArray[loc-1] > nextItem); --loc)  
            theArray[loc] = theArray[loc-1];  
  
        theArray[loc] = nextItem;  
    }  
}
```

```
}  
}
```

Trace of Insertion Sort							
4	8	3	7	6	2	1	5
4	8	3	7	6	2	1	5
3	4	8	7	6	2	1	5
3	4	7	8	6	2	1	5
3	4	6	7	8	2	1	5
2	3	4	6	7	8	1	5
1	2	3	4	6	7	8	5
1	2	3	4	5	6	7	8

## b) Selection Sort

This algorithm sorts the array by first finding the biggest element and then swapping that element with the last element of the unsorted array. In addition to the main function which houses the selection sort algorithm, it uses two other methods. The first one finds the biggest element (in value) of the unsorted array (it takes the array and a size value, the size value decreases hence this method has access to only the unsorted part of the array), the second one swaps two elements of the array (it changes the values of the pointers). The biggest element of the unsorted array gets swapped with the last element of the unsorted array, and the unsorted arrays size value gets decreased. Hence the element becomes the first element of the sorted array. This can be understood further by inspecting the given code below and the trace example that will be given.

```
void selectionSort( DataType theArray[], int n) {  
    for (int last = n-1; last >= 1; --last) {  
        int largest = indexOfLargest(theArray, last+1);  
        swap(theArray[largest], theArray[last]);  
    }  
}
```

```
}  
}
```

```
int indexOfLargest(const DataType theArray[], int size) {  
    int indexSoFar = 0;  
    for (int currentIndex=1; currentIndex<size;++currentIndex)  
    {  
        if (theArray[currentIndex] > theArray[indexSoFar])  
            indexSoFar = currentIndex;  
    }  
    return indexSoFar;  
}
```

```
void swap(DataType &x, DataType &y) {  
    DataType temp = x;  
    x = y;  
    y = temp;  
}
```

Trace of Selection Sort							
4	8	3	7	6	2	1	5
4	5	3	7	6	2	1	8
4	5	3	1	6	2	7	8
4	5	3	1	2	6	7	8
4	2	3	1	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

c) Bubble Sort

This algorithm sorts the array by comparing 2 elements (hence the bubble) and then swapping the preceding element with the prior if the prior element is smaller. This way at each bubble pass, the sorted part of the array increases in size by one, and the unsorted part decreases in size. By comparing two elements in bubbles, the biggest element of the unsorted array becomes the first element of the sorted array. Hence the array gets sorted. Below the code for this algorithm, as well as each pass can be observed below. Similar to prior traces, the sorted array will be colored in blue and the bubbled elements will be colored in red.

```
void bubbleSort( DataType theArray[], int n) {
    bool sorted = false;

    for (int pass = 1; (pass < n) && !sorted; ++pass) {

        sorted = true;
        for (int index = 0; index < n-pass; ++index) {

            int nextIndex = index + 1;
            if (theArray[index] > theArray[nextIndex]) {

                swap(theArray[index], theArray[nextIndex]);
                sorted = false;
            }
        }
    }
}
```

Trace of Bubble Sort: 1 <sup>st</sup> Pass							
4	8	3	7	6	2	1	5
4	8	3	7	6	2	1	5
4	3	8	7	6	2	1	5
4	13	7	8	6	2	1	5
4	3	7	6	8	2	1	5
4	3	7	6	2	8	1	5

4	3	7	6	2	1	8	5
4	3	7	6	2	1	5	8
Trace of Bubble Sort: 2 <sup>nd</sup> Pass							
4	3	7	6	2	1	5	8
3	4	7	6	2	1	5	8
3	4	7	6	2	1	5	8
3	4	7	6	2	1	5	8
3	4	6	7	2	1	5	8
3	4	6	2	7	1	5	8
3	4	6	2	1	7	5	8
3	4	6	2	1	5	7	8
Trace of Bubble Sort: 3 <sup>rd</sup> Pass							
3	4	6	2	1	5	7	8
3	4	6	2	1	5	7	8
3	4	6	2	1	5	7	8
3	4	2	6	1	5	7	8
3	4	2	1	6	5	7	8
3	4	2	1	5	6	7	8
Trace of Bubble Sort: 4 <sup>th</sup> Pass							
3	4	2	1	5	6	7	8
3	4	2	1	5	6	7	8
3	2	4	1	5	6	7	8
3	2	1	4	5	6	7	8
Trace of Bubble Sort: 5 <sup>th</sup> Pass							
3	2	1	4	5	6	7	8
2	3	1	4	5	6	7	8
2	1	3	4	5	6	7	8
Trace of Bubble Sort: 6 <sup>th</sup> Pass							
2	1	3	4	5	6	7	8
1	2	3	4	5	6	7	8

#### d) Merge Sort

Merge Sort is a quicker sorting algorithm compared to the three above. It uses the principle of divide and conquer. It first recursively divides the array until each element of the array is isolated. Then, the recursion starts unfolding and each step of recursion calls the merge method, which merges the isolated elements and sorts them. The algorithm is hard to understand solely by using words, but the trace, in addition to the order of calls should give an idea of how this algorithm manages to sort the array. Below the code for merge and mergesort functions, as well as the trace for the array can be found.

```
void mergesort( DataType theArray[], int first, int last) {
```

```
    if (first < last) {  
        int mid = (first + last)/2;  
  
        mergesort(theArray, first, mid);  
        mergesort(theArray, mid+1, last);  
  
        merge(theArray, first, mid, last);  
    }  
}
```

```
const int MAX_SIZE = maximum-number-of-items-in-array;
```

```
void merge( DataType theArray[], int first, int mid, int last) {
```

```
    DataType tempArray[MAX_SIZE];    //MAX_SIZE: max size of array, predefined  
    int first1 = first;    // beginning of first subarray  
    int last1 = mid;    // end of first subarray  
    int first2 = mid + 1;    // beginning of second subarray  
    int last2 = last;    // end of second subarray  
    int index = first1;    // next available location in tempArray
```

```

for ( ; (first1 <= last1) && (first2 <= last2); ++index) {
    if (theArray[first1] < theArray[first2]) {
        tempArray[index] = theArray[first1];
        ++first1;
    }
    else {
        tempArray[index] = theArray[first2];
        ++first2;
    }
}
for ( ; first1 <= last1; ++first1, ++index)
    tempArray[index] = theArray[first1];
for ( ; first2 <= last2; ++first2, ++index)
    tempArray[index] = theArray[first2];

for (index = first; index <= last; ++index)
    theArray[index] = tempArray[index];

}

```

mergesort							
4	8	3	7	6	2	1	5
mergesort				mergesort: 12 <sup>th</sup> call			
4	8	3	7	6	2	1	5
mergesort		mergesort		mergesort		mergesort	
4	8	3	7	6	2	1	5
mergesort	mergesort	mergesort	mergesort	mergesort	mergesort	mergesort	mergesort
4	8	3	7	6	2	1	5

merge		merge		merge		merge	
4	8	3	7	2	6	1	5
merge				merge			
3	4	7	8	1	2	5	6
merge							
1	2	3	4	5	6	7	8

### Call List for Merge Sort

1. mergesort(array, 0, 7)
2. mergesort(array, 0, 3)
3. mergesort(array, 0, 1)
4. mergesort(array, 0, 0)
5. mergesort(array, 1, 1)
6. merge(array, 0, 0, 1)
7. mergesort(array, 2, 3)
8. mergesort(array, 2, 2)
9. mergesort(array, 3, 3)
10. merge(array, 2, 2, 3)
11. merge(array, 0, 1, 3)
12. mergesort(array, 4, 7)
13. mergesort(array, 4, 5)
14. mergesort(array, 4, 4)
15. mergesort(array, 5, 5)
16. merge(array, 4, 4, 5)
17. mergesort(array, 6, 7)
18. mergesort(array, 6, 6)
19. mergesort(array, 7, 7)
20. merge(array, 6, 6, 7)
21. merge(array, 4, 5, 7)
22. merge(array, 0, 3, 7)

e) Quick Sort



Quick Sort is similar to the Merge Sort algorithm in the way that both use divide and conquer to sort a given array. The difference between the two is when the comparisons happen. In Quick Sort, the array elements get compared first; hence, this algorithm does the hard part of sorting first. It uses recursion and a function called partition, which partitions the array given a pivot. The trace of this algorithm and the code will make the operation more understandable, and both can be found below. For the trace, the first item is chosen as the pivot and will be displayed with the color red.

```
void quicksort(DataType theArray[], int first, int last) {
    int pivotIndex;
    if (first < last) {
        partition(theArray, first, last, pivotIndex);
        quicksort(theArray, first, pivotIndex-1);
        quicksort(theArray, pivotIndex+1, last);
    }
}

void partition(DataType theArray[], int first, int last, int &pivotIndex) {
    choosePivot(theArray, first, last);
    DataType pivot = theArray[first];
    int lastS1 = first;
    int firstUnknown = first + 1;
    for ( ; firstUnknown <= last; ++firstUnknown) {
        if (theArray[firstUnknown] < pivot) {
            ++lastS1;
            swap(theArray[firstUnknown], theArray[lastS1]);
        }
    }
    swap(theArray[first], theArray[lastS1]);
    pivotIndex = lastS1;
}
```

quicksort
-----------

4	8	3	7	6	2	1	5
partition							
1	3	2	4	6	8	7	5
quicksort			sorted	quicksort			
1	3	2	4	6	8	7	5
partition			sorted	partition			
1	3	2	4	5	6	7	8
quicksort	quicksort		sorted	quicksort	sorted	quicksort	
1	3	2	4	5	6	7	8
sorted	partition		sorted			partition	
1	2	3	4	5	6	7	8
sorted	quicksort	sorted					quicksort
1	2	3	4	5	6	7	8
sorted array							
1	2	3	4	5	6	7	8

### Call List for Quick Sort

1. quicksort(array, 0, 7)
2. partition(array, 0, 7, 0)
3. quicksort(array, 0, 2)
4. partition(array, 0, 2, 0)
5. quicksort(array, 1, 2)
6. partition(array, 1, 2, 1)
7. quicksort(array, 2, 2)
8. quicksort(array, 4, 7)
9. partition(array, 4, 7, 4)
10. quicksort(array, 4, 6)
11. partition(array, 4, 6, 4)
12. quicksort(array, 5, 6)
13. partition(array, 5, 6, 5)

- 14. quicksort(array, 5, 6)
- 15. partition(array, 5, 6, 5)
- 16. quicksort(array, 6, 6)

## Question 2

### Merge Sort

The recurrence relation for the worst case of merge sort is same as the average case, which is

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Where  $2T(n/2)$  represents the divided array and  $n$  represents the merging of the divided arrays. This relation will be solved by repeated substitutions method.

$$T(n) = 2T\left(\frac{n}{2}\right) + n, \quad T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

Substituting the two together will give

$$T(n) = 4T\left(\frac{n}{4}\right) + 2n$$

Although we have a glimpse of understanding the patters, to be sure let's substitute once more to see the relation clearly.

$$T(n) = 4T\left(\frac{n}{4}\right) + 2n, \quad T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$
$$T(n) = 8T\left(\frac{n}{8}\right) + 3n$$

Now it is clear that there are only three changing multipliers, this will give us the general relation below

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

It is known that for an array of size 1,  $T(1)$  is the resulting equation. This can be rewritten as  $T\left(\frac{n}{n}\right)$  which leads to the conclusion that in this case  $n = 2^k$ . This is the last substitution of the general relation, substituting this will give

$$T(n) = 2^k T(1) + n * k$$

To find the time complexity, we need to find what  $k$  is, and from the above equivalence we find that  $k = \log_2 n$ . Additionally,  $T(1) * 2^k = c$  since  $k$  is also a constant (another constant value). Putting these into the above equation will give

$$T(n) = c + n * \log_2 n$$

From this we can conclude that the time complexity of the worst case for merge sort is

$$\Theta(T(n)) = \Theta(c + n * \log_2 n) = \Theta(n \log_2 n)$$

$$\Theta(n \log n)$$

### Quick Sort

The recurrence relation for the worst case of quick sort is

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

The expression  $T(n - 1)$  represents the array after partitioning (in the worst case for quick sort the array will be sorted, so when the first element is taken as the pivot, the remainder of the array will just be the original array minus the pivot) and  $n$  represents the partitioning. Similar to merge sort, let's substitute 2 steps of the relation to get an understanding of the general relation and how it connects to size  $n = 1, T(1)$ .

$$T(n) = T(n - 1) + n, \quad T(n - 1) = T(n - 2) + n - 1$$

$$T(n) = T(n - 2) + n + n - 1, \quad T(n - 2) = T(n - 3) + n - 2$$

$$T(n) = T(n - 3) + n + n - 1 + n - 2$$

...

...

...

$$T(n) = T(1) + n + n - 1 + \dots + n - (n - 1)$$

Considering the fact that  $T(1) = c$ , the general relation can be written as

$$T(n) = c + n * (n - 1) - \sum_{k=1}^{n-1} k$$

The series represents the addition of  $k$  from 1 to  $n-1$ . This can be re-written as an understandable expression. This makes the final form of the expression

$$T(n) = c + n(n-1) - \frac{n(n-1)}{2} = c + \frac{n(n-1)}{2} = c + \frac{n^2}{2} - \frac{n}{2}$$

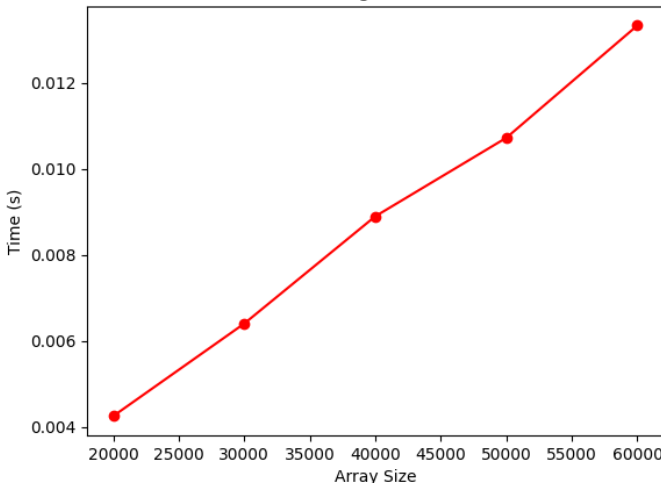
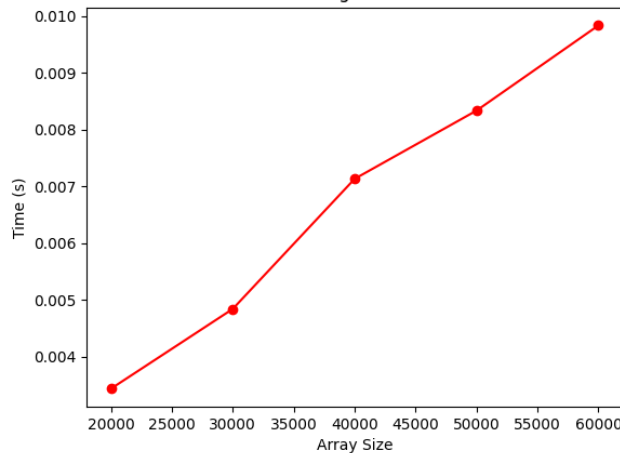
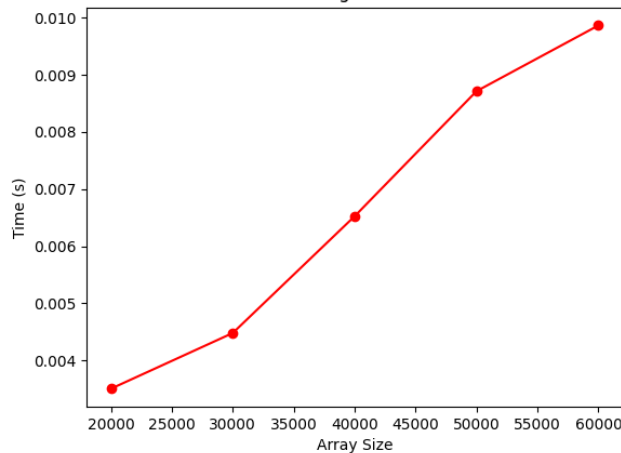
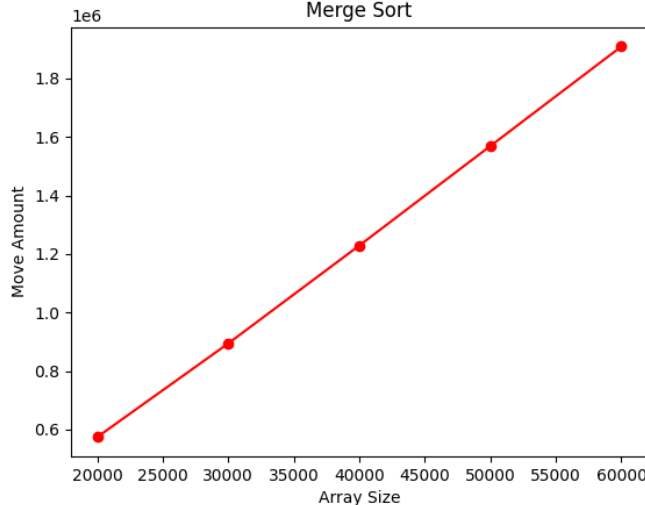
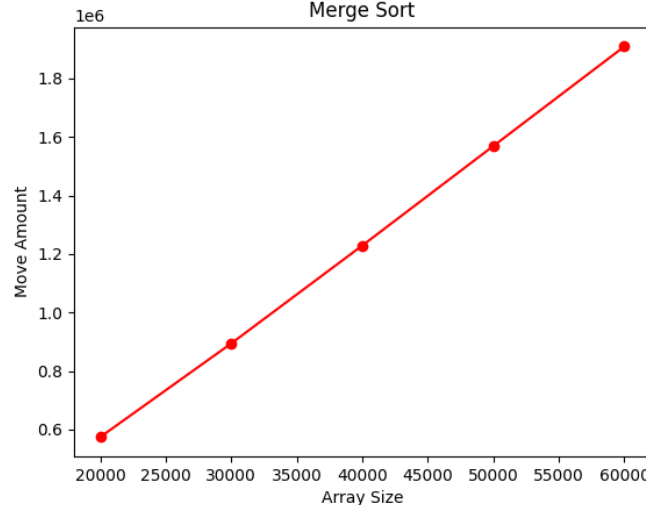
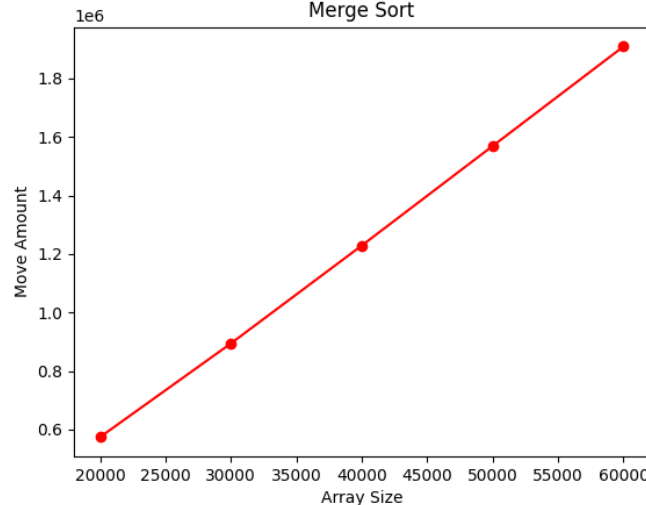
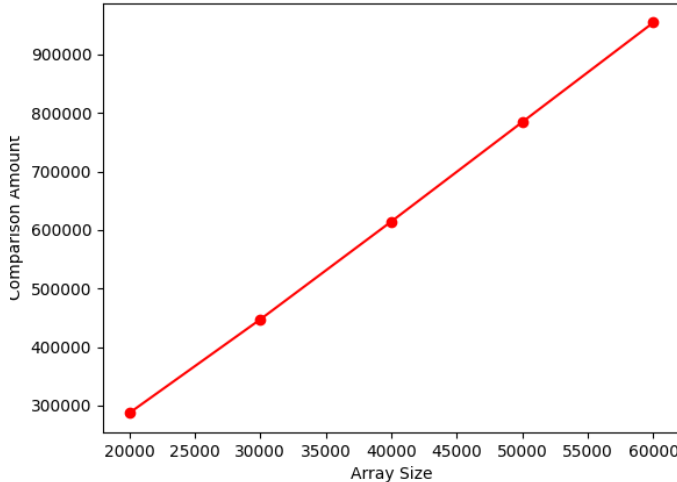
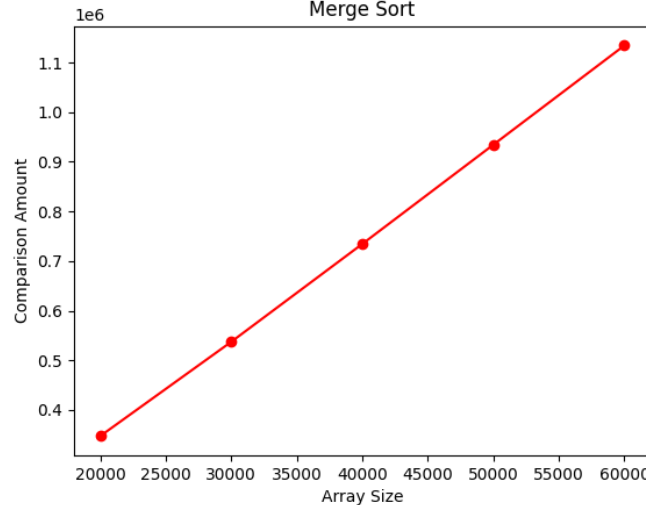
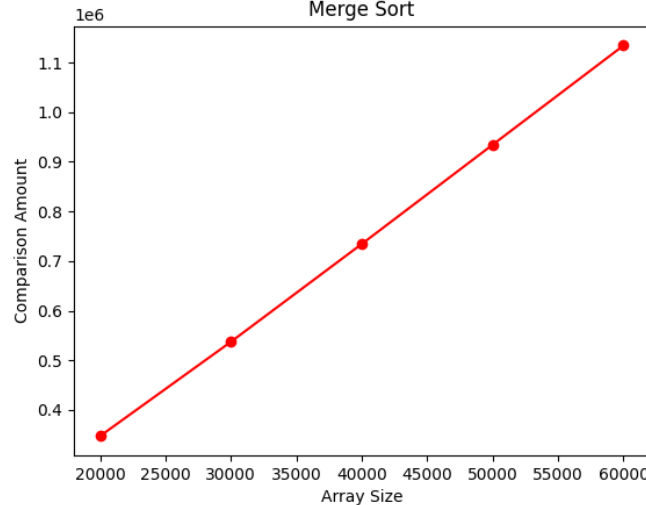
From the above expression we can find the worst time complexity of quick sort as

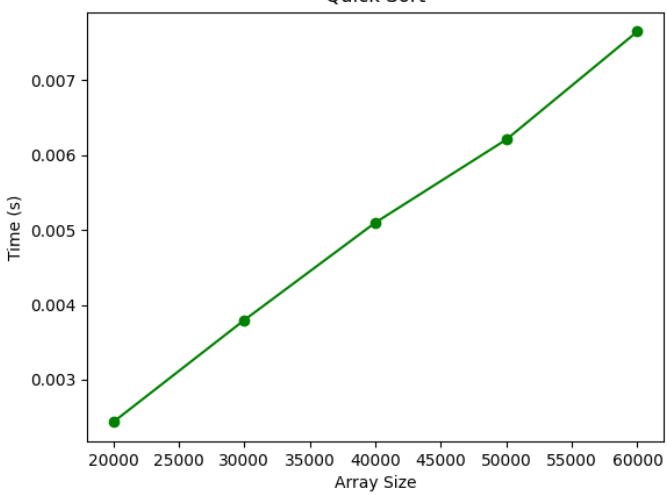
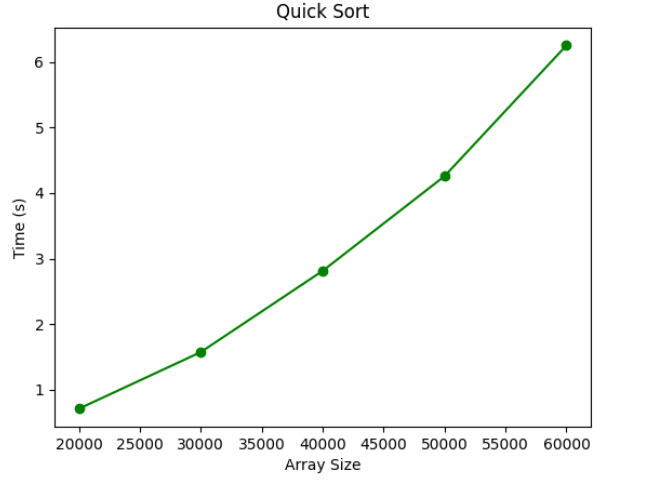
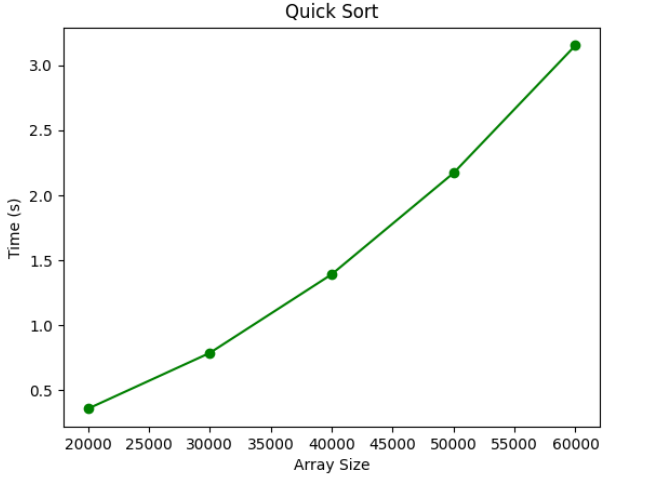
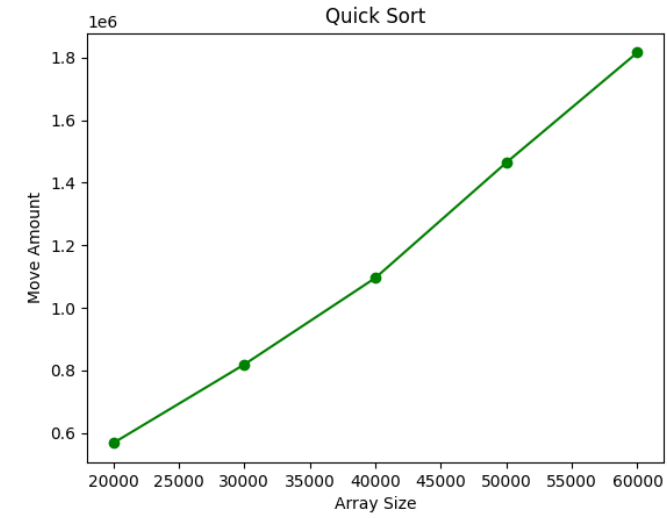
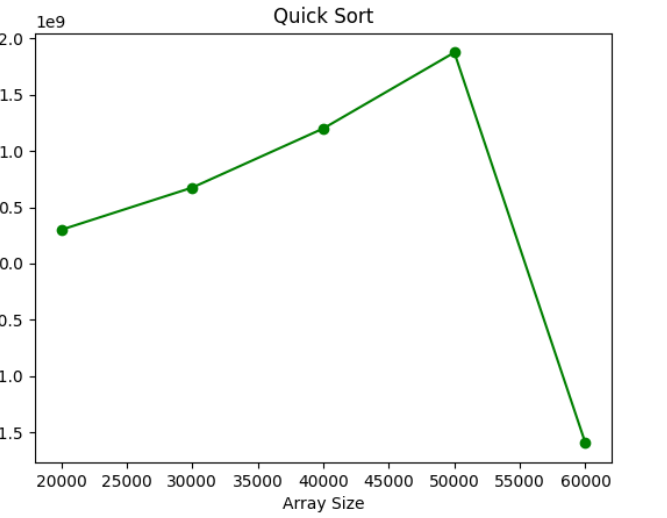
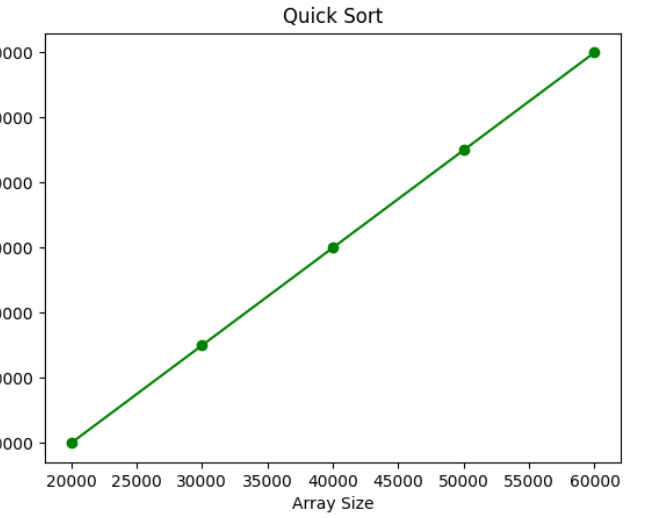
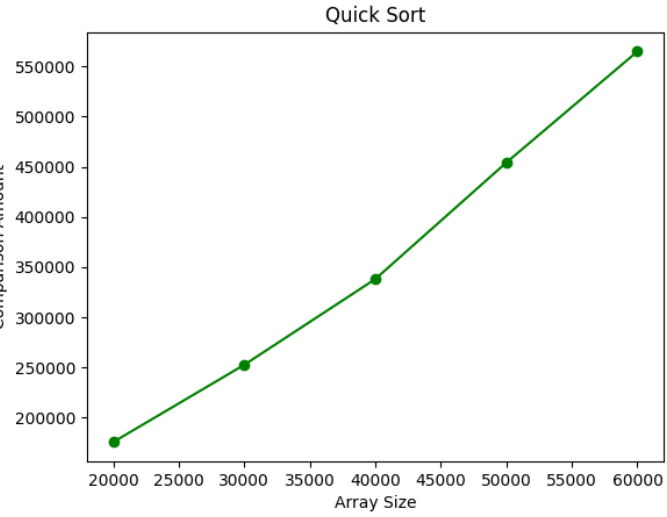
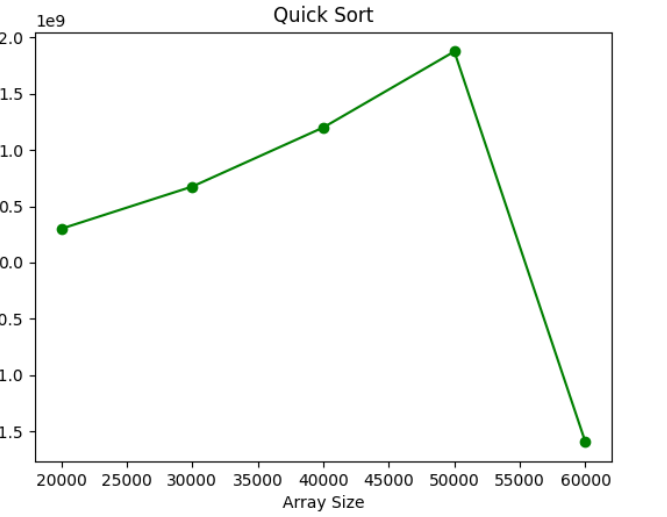
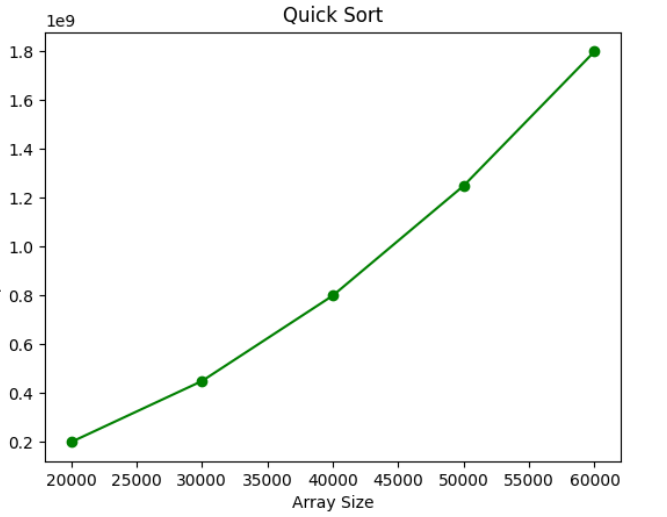
$$\Theta(T(n)) = \Theta\left(c + \frac{n^2}{2} - \frac{n}{2}\right)$$
$$\Theta(n^2)$$

### Question 3

This section will display the figures which were obtained from the 3 sorting algorithms that were implemented. 3 different experiments were conducted; an array with all random elements, an array with sorted elements in descending order and an array with sorted elements in ascending order. Array sizes were as follows: 20000, 30000, 40000, 50000, 60000 (5 different sizes). Python was used to plot the data; it should be noted that some figures have the degree of 10 above the y axis (ex: 1e8 means the y axis is to be multiplied with 10<sup>8</sup>). A discussion of these results will be on the next question.

INSERTION SORT	ARRAY WITH RANDOM INTEGERS	SORTED ARRAY (DESCENDING ORDER)	SORTED ARRAY (ASCENDING ORDER)																																				
TIME	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.25</td></tr><tr><td>30000</td><td>0.58</td></tr><tr><td>40000</td><td>1.02</td></tr><tr><td>50000</td><td>1.55</td></tr><tr><td>60000</td><td>2.25</td></tr></table></div>	Array Size	Time (s)	20000	0.25	30000	0.58	40000	1.02	50000	1.55	60000	2.25	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.5</td></tr><tr><td>30000</td><td>1.15</td></tr><tr><td>40000</td><td>2.0</td></tr><tr><td>50000</td><td>3.0</td></tr><tr><td>60000</td><td>4.4</td></tr></table></div>	Array Size	Time (s)	20000	0.5	30000	1.15	40000	2.0	50000	3.0	60000	4.4	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.000082</td></tr><tr><td>30000</td><td>0.000105</td></tr><tr><td>40000</td><td>0.000138</td></tr><tr><td>50000</td><td>0.000182</td></tr><tr><td>60000</td><td>0.000218</td></tr></table></div>	Array Size	Time (s)	20000	0.000082	30000	0.000105	40000	0.000138	50000	0.000182	60000	0.000218
Array Size	Time (s)																																						
20000	0.25																																						
30000	0.58																																						
40000	1.02																																						
50000	1.55																																						
60000	2.25																																						
Array Size	Time (s)																																						
20000	0.5																																						
30000	1.15																																						
40000	2.0																																						
50000	3.0																																						
60000	4.4																																						
Array Size	Time (s)																																						
20000	0.000082																																						
30000	0.000105																																						
40000	0.000138																																						
50000	0.000182																																						
60000	0.000218																																						
MOVE COUNT	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Move Amount</th></tr><tr><td>20000</td><td>1e8</td></tr><tr><td>30000</td><td>2.2e8</td></tr><tr><td>40000</td><td>4e8</td></tr><tr><td>50000</td><td>6.2e8</td></tr><tr><td>60000</td><td>9e8</td></tr></table></div>	Array Size	Move Amount	20000	1e8	30000	2.2e8	40000	4e8	50000	6.2e8	60000	9e8	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Move Amount</th></tr><tr><td>20000</td><td>0.2e9</td></tr><tr><td>30000</td><td>0.45e9</td></tr><tr><td>40000</td><td>0.8e9</td></tr><tr><td>50000</td><td>1.25e9</td></tr><tr><td>60000</td><td>1.8e9</td></tr></table></div>	Array Size	Move Amount	20000	0.2e9	30000	0.45e9	40000	0.8e9	50000	1.25e9	60000	1.8e9	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Move Amount</th></tr><tr><td>20000</td><td>20000</td></tr><tr><td>30000</td><td>30000</td></tr><tr><td>40000</td><td>40000</td></tr><tr><td>50000</td><td>50000</td></tr><tr><td>60000</td><td>60000</td></tr></table></div>	Array Size	Move Amount	20000	20000	30000	30000	40000	40000	50000	50000	60000	60000
Array Size	Move Amount																																						
20000	1e8																																						
30000	2.2e8																																						
40000	4e8																																						
50000	6.2e8																																						
60000	9e8																																						
Array Size	Move Amount																																						
20000	0.2e9																																						
30000	0.45e9																																						
40000	0.8e9																																						
50000	1.25e9																																						
60000	1.8e9																																						
Array Size	Move Amount																																						
20000	20000																																						
30000	30000																																						
40000	40000																																						
50000	50000																																						
60000	60000																																						
COMPARISON COUNT	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Comparison Amount</th></tr><tr><td>20000</td><td>1e8</td></tr><tr><td>30000</td><td>2.2e8</td></tr><tr><td>40000</td><td>4e8</td></tr><tr><td>50000</td><td>6.2e8</td></tr><tr><td>60000</td><td>9e8</td></tr></table></div>	Array Size	Comparison Amount	20000	1e8	30000	2.2e8	40000	4e8	50000	6.2e8	60000	9e8	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Comparison Amount</th></tr><tr><td>20000</td><td>0.2e9</td></tr><tr><td>30000</td><td>0.45e9</td></tr><tr><td>40000</td><td>0.8e9</td></tr><tr><td>50000</td><td>1.25e9</td></tr><tr><td>60000</td><td>1.8e9</td></tr></table></div>	Array Size	Comparison Amount	20000	0.2e9	30000	0.45e9	40000	0.8e9	50000	1.25e9	60000	1.8e9	<div><p>Insertion Sort</p><table><tr><th>Array Size</th><th>Comparison Amount</th></tr><tr><td>20000</td><td>20000</td></tr><tr><td>30000</td><td>30000</td></tr><tr><td>40000</td><td>40000</td></tr><tr><td>50000</td><td>50000</td></tr><tr><td>60000</td><td>60000</td></tr></table></div>	Array Size	Comparison Amount	20000	20000	30000	30000	40000	40000	50000	50000	60000	60000
Array Size	Comparison Amount																																						
20000	1e8																																						
30000	2.2e8																																						
40000	4e8																																						
50000	6.2e8																																						
60000	9e8																																						
Array Size	Comparison Amount																																						
20000	0.2e9																																						
30000	0.45e9																																						
40000	0.8e9																																						
50000	1.25e9																																						
60000	1.8e9																																						
Array Size	Comparison Amount																																						
20000	20000																																						
30000	30000																																						
40000	40000																																						
50000	50000																																						
60000	60000																																						

MERGE SORT	ARRAY WITH RANDOM INTEGERS	SORTED ARRAY (DESCENDING ORDER)	SORTED ARRAY (ASCENDING ORDER)																																				
TIME	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.0043</td></tr><tr><td>30000</td><td>0.0064</td></tr><tr><td>40000</td><td>0.0089</td></tr><tr><td>50000</td><td>0.0107</td></tr><tr><td>60000</td><td>0.0133</td></tr></table>	Array Size	Time (s)	20000	0.0043	30000	0.0064	40000	0.0089	50000	0.0107	60000	0.0133	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.0035</td></tr><tr><td>30000</td><td>0.0048</td></tr><tr><td>40000</td><td>0.0071</td></tr><tr><td>50000</td><td>0.0083</td></tr><tr><td>60000</td><td>0.0098</td></tr></table>	Array Size	Time (s)	20000	0.0035	30000	0.0048	40000	0.0071	50000	0.0083	60000	0.0098	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.0035</td></tr><tr><td>30000</td><td>0.0045</td></tr><tr><td>40000</td><td>0.0065</td></tr><tr><td>50000</td><td>0.0087</td></tr><tr><td>60000</td><td>0.0098</td></tr></table>	Array Size	Time (s)	20000	0.0035	30000	0.0045	40000	0.0065	50000	0.0087	60000	0.0098
Array Size	Time (s)																																						
20000	0.0043																																						
30000	0.0064																																						
40000	0.0089																																						
50000	0.0107																																						
60000	0.0133																																						
Array Size	Time (s)																																						
20000	0.0035																																						
30000	0.0048																																						
40000	0.0071																																						
50000	0.0083																																						
60000	0.0098																																						
Array Size	Time (s)																																						
20000	0.0035																																						
30000	0.0045																																						
40000	0.0065																																						
50000	0.0087																																						
60000	0.0098																																						
MOVE COUNT	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Move Amount (1e6)</th></tr><tr><td>20000</td><td>0.57</td></tr><tr><td>30000</td><td>0.89</td></tr><tr><td>40000</td><td>1.23</td></tr><tr><td>50000</td><td>1.57</td></tr><tr><td>60000</td><td>1.90</td></tr></table>	Array Size	Move Amount (1e6)	20000	0.57	30000	0.89	40000	1.23	50000	1.57	60000	1.90	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Move Amount (1e6)</th></tr><tr><td>20000</td><td>0.57</td></tr><tr><td>30000</td><td>0.89</td></tr><tr><td>40000</td><td>1.23</td></tr><tr><td>50000</td><td>1.57</td></tr><tr><td>60000</td><td>1.90</td></tr></table>	Array Size	Move Amount (1e6)	20000	0.57	30000	0.89	40000	1.23	50000	1.57	60000	1.90	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Move Amount (1e6)</th></tr><tr><td>20000</td><td>0.57</td></tr><tr><td>30000</td><td>0.89</td></tr><tr><td>40000</td><td>1.23</td></tr><tr><td>50000</td><td>1.57</td></tr><tr><td>60000</td><td>1.90</td></tr></table>	Array Size	Move Amount (1e6)	20000	0.57	30000	0.89	40000	1.23	50000	1.57	60000	1.90
Array Size	Move Amount (1e6)																																						
20000	0.57																																						
30000	0.89																																						
40000	1.23																																						
50000	1.57																																						
60000	1.90																																						
Array Size	Move Amount (1e6)																																						
20000	0.57																																						
30000	0.89																																						
40000	1.23																																						
50000	1.57																																						
60000	1.90																																						
Array Size	Move Amount (1e6)																																						
20000	0.57																																						
30000	0.89																																						
40000	1.23																																						
50000	1.57																																						
60000	1.90																																						
COMPARISON COUNT	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Comparison Amount</th></tr><tr><td>20000</td><td>285000</td></tr><tr><td>30000</td><td>450000</td></tr><tr><td>40000</td><td>615000</td></tr><tr><td>50000</td><td>780000</td></tr><tr><td>60000</td><td>950000</td></tr></table>	Array Size	Comparison Amount	20000	285000	30000	450000	40000	615000	50000	780000	60000	950000	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Comparison Amount (1e6)</th></tr><tr><td>20000</td><td>0.35</td></tr><tr><td>30000</td><td>0.54</td></tr><tr><td>40000</td><td>0.73</td></tr><tr><td>50000</td><td>0.93</td></tr><tr><td>60000</td><td>1.15</td></tr></table>	Array Size	Comparison Amount (1e6)	20000	0.35	30000	0.54	40000	0.73	50000	0.93	60000	1.15	<p>Merge Sort</p>  <table><tr><th>Array Size</th><th>Comparison Amount (1e6)</th></tr><tr><td>20000</td><td>0.35</td></tr><tr><td>30000</td><td>0.54</td></tr><tr><td>40000</td><td>0.73</td></tr><tr><td>50000</td><td>0.93</td></tr><tr><td>60000</td><td>1.15</td></tr></table>	Array Size	Comparison Amount (1e6)	20000	0.35	30000	0.54	40000	0.73	50000	0.93	60000	1.15
Array Size	Comparison Amount																																						
20000	285000																																						
30000	450000																																						
40000	615000																																						
50000	780000																																						
60000	950000																																						
Array Size	Comparison Amount (1e6)																																						
20000	0.35																																						
30000	0.54																																						
40000	0.73																																						
50000	0.93																																						
60000	1.15																																						
Array Size	Comparison Amount (1e6)																																						
20000	0.35																																						
30000	0.54																																						
40000	0.73																																						
50000	0.93																																						
60000	1.15																																						

QUICK SORT	ARRAY WITH RANDOM INTEGERS	SORTED ARRAY (DESCENDING ORDER)	SORTED ARRAY (ASCENDING ORDER)																																				
TIME	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.0025</td></tr><tr><td>30000</td><td>0.0038</td></tr><tr><td>40000</td><td>0.0051</td></tr><tr><td>50000</td><td>0.0062</td></tr><tr><td>60000</td><td>0.0075</td></tr></table>	Array Size	Time (s)	20000	0.0025	30000	0.0038	40000	0.0051	50000	0.0062	60000	0.0075	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.8</td></tr><tr><td>30000</td><td>1.6</td></tr><tr><td>40000</td><td>2.8</td></tr><tr><td>50000</td><td>4.3</td></tr><tr><td>60000</td><td>6.2</td></tr></table>	Array Size	Time (s)	20000	0.8	30000	1.6	40000	2.8	50000	4.3	60000	6.2	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Time (s)</th></tr><tr><td>20000</td><td>0.4</td></tr><tr><td>30000</td><td>0.8</td></tr><tr><td>40000</td><td>1.4</td></tr><tr><td>50000</td><td>2.2</td></tr><tr><td>60000</td><td>3.2</td></tr></table>	Array Size	Time (s)	20000	0.4	30000	0.8	40000	1.4	50000	2.2	60000	3.2
Array Size	Time (s)																																						
20000	0.0025																																						
30000	0.0038																																						
40000	0.0051																																						
50000	0.0062																																						
60000	0.0075																																						
Array Size	Time (s)																																						
20000	0.8																																						
30000	1.6																																						
40000	2.8																																						
50000	4.3																																						
60000	6.2																																						
Array Size	Time (s)																																						
20000	0.4																																						
30000	0.8																																						
40000	1.4																																						
50000	2.2																																						
60000	3.2																																						
MOVE COUNT	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Move Amount (1e6)</th></tr><tr><td>20000</td><td>0.55</td></tr><tr><td>30000</td><td>0.82</td></tr><tr><td>40000</td><td>1.1</td></tr><tr><td>50000</td><td>1.45</td></tr><tr><td>60000</td><td>1.8</td></tr></table>	Array Size	Move Amount (1e6)	20000	0.55	30000	0.82	40000	1.1	50000	1.45	60000	1.8	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Move Amount (1e9)</th></tr><tr><td>20000</td><td>0.3</td></tr><tr><td>30000</td><td>0.7</td></tr><tr><td>40000</td><td>1.2</td></tr><tr><td>50000</td><td>1.9</td></tr><tr><td>60000</td><td>-1.6</td></tr></table>	Array Size	Move Amount (1e9)	20000	0.3	30000	0.7	40000	1.2	50000	1.9	60000	-1.6	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Move Amount</th></tr><tr><td>20000</td><td>60000</td></tr><tr><td>30000</td><td>90000</td></tr><tr><td>40000</td><td>120000</td></tr><tr><td>50000</td><td>150000</td></tr><tr><td>60000</td><td>180000</td></tr></table>	Array Size	Move Amount	20000	60000	30000	90000	40000	120000	50000	150000	60000	180000
Array Size	Move Amount (1e6)																																						
20000	0.55																																						
30000	0.82																																						
40000	1.1																																						
50000	1.45																																						
60000	1.8																																						
Array Size	Move Amount (1e9)																																						
20000	0.3																																						
30000	0.7																																						
40000	1.2																																						
50000	1.9																																						
60000	-1.6																																						
Array Size	Move Amount																																						
20000	60000																																						
30000	90000																																						
40000	120000																																						
50000	150000																																						
60000	180000																																						
COMPARISON COUNT	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Comparison Amount</th></tr><tr><td>20000</td><td>180000</td></tr><tr><td>30000</td><td>250000</td></tr><tr><td>40000</td><td>340000</td></tr><tr><td>50000</td><td>450000</td></tr><tr><td>60000</td><td>550000</td></tr></table>	Array Size	Comparison Amount	20000	180000	30000	250000	40000	340000	50000	450000	60000	550000	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Comparison Amount (1e9)</th></tr><tr><td>20000</td><td>0.3</td></tr><tr><td>30000</td><td>0.7</td></tr><tr><td>40000</td><td>1.2</td></tr><tr><td>50000</td><td>1.9</td></tr><tr><td>60000</td><td>-1.6</td></tr></table>	Array Size	Comparison Amount (1e9)	20000	0.3	30000	0.7	40000	1.2	50000	1.9	60000	-1.6	<p>Quick Sort</p>  <table><tr><th>Array Size</th><th>Comparison Amount (1e9)</th></tr><tr><td>20000</td><td>0.2</td></tr><tr><td>30000</td><td>0.45</td></tr><tr><td>40000</td><td>0.8</td></tr><tr><td>50000</td><td>1.25</td></tr><tr><td>60000</td><td>1.8</td></tr></table>	Array Size	Comparison Amount (1e9)	20000	0.2	30000	0.45	40000	0.8	50000	1.25	60000	1.8
Array Size	Comparison Amount																																						
20000	180000																																						
30000	250000																																						
40000	340000																																						
50000	450000																																						
60000	550000																																						
Array Size	Comparison Amount (1e9)																																						
20000	0.3																																						
30000	0.7																																						
40000	1.2																																						
50000	1.9																																						
60000	-1.6																																						
Array Size	Comparison Amount (1e9)																																						
20000	0.2																																						
30000	0.45																																						
40000	0.8																																						
50000	1.25																																						
60000	1.8																																						



## Question 4

The first thing that I would like to note is the integer overflow that happens in the descending order section of the quick sort algorithm. The number of counts goes into negative; however, that only indicates that the code cannot represent the integer in positive int values. Taking that into mind, the graphs clearly show a growing trend in comparisons and move count, which is to be expected in what is accepted to be the worst case for quicksort. I could've changed the variables from int to long but didn't think this one case would be a big problem since it only happens in 2 out of 18 count graphs.

Starting with the first experiment, which can be accepted as the average case for all three sorting algorithms, is the experiment that was done with arrays of random integers. Theoretically we have insertion sort with  $O(n^2)$  and merge sort and quick sort with  $O(n \log n)$ . The theoretical complexities are parallel to the experimental results. Insertion sort takes seconds to sort the array, while quick and merge sort only take several milliseconds. Similar results can be seen at the move and comparison count plots as well. Insertion sort makes  $10^8$  counts and moves, while merge and quick sort only make hundred thousands of comparisons and millions of moves. Hence, we can conclude that for an array with random elements merge and quick sort are much more efficient compared to insertion sort, both in terms of time but also the amount of key operation the program makes.

For the second experiment, which was done with sorted arrays of descending order, the theoretical complexity of quick sort changed, since this case is the worst case for this algorithm. For merge sort, the complexity does not change since for any array it does the same operation of dividing the array and then merging it back. For insertion sort, the complexity stays the same; however, this array is the worst case for this algorithm as well. Hence, we now have insertion and quicksort at  $O(n^2)$  and merge sort at  $O(n \log n)$ . For this array, quick sort serves the worst time efficiency out of the three algorithms. Insertion sort is a close second and as it was mentioned, since this is the worst case for insertion sort the time spent on sorting is more than the first experiment. Merge sort is somewhat unaffected in terms of time but is lower than the first experiment. In terms of comparisons and move counts, quick sort is again highly inefficient. So much so that an int overflow happens. Insertion sort is not much different than quick sort but only requires a little moves and comparisons. Merge sort again requires the same amount of moves but increases in comparison amount which is to be expected since the array

For the third experiment, a sorted array of ascending order was to be sorted. The time complexity of insertion sort in this case changed to  $O(n)$ , which is the best case of insertion sort. However, quick sort and merge sort remained the same with  $O(n^2)$  and  $O(n \log n)$ . This can again be seen in the time graphs. Insertion sort is extremely efficient even lower than 1 millisecond. Merge sort is again unchanged, but quicksort again is inefficient. This complies with the theoretical complexities. For move and comparison counts, insertion sort only compares

and moves as much as the array size, merge sort makes the same amount of moves and comparisons as the prior experiment. Quick sort is inefficient in this area as well but is much better than the prior experiment.

Taking all these in mind, merge sort seems to be the most efficient sorting algorithm out of the three. This is because of its consistency and time efficiency (although there may be worries of space efficiency since it allocates an extra size array, but that is not the focus of these experiments). Quick sort would be second since although at most cases it is more efficient than merge sort, it is not consistent and hence might be considered unpredictable. Insertion sort would come at third position since it is the least efficient of all.