CSC 301

AN EXPERIMENT TO DETERMINE PRACTICAL EFFICIENCY OF INSERTION SORT, MERGE SORT & QUICK SORT

GROUP 6

AMUDA, Tosin Joseph 090805009
ADESHINA, Taofeek Akanbi 100805006
SALAMI, Shehu Tijani 090805056
SORINOLU, Olamilekan 090805062
KAZEEM, Adetunji Lukman 090805025
DOSUNMU, Olakunle Musiliu 100805027

Abstract

Sorting is a fundamental operation in computer science (many programs use it as an intermediate step), and as a result a large number of good sorting algorithms have been developed. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, and the kind of storage device to be used: main memory, disks, or tapes.

There are three reasons to study sorting algorithms. First, sorting algorithms illustrate many creative approaches to problem solving, and these approaches can be applied to solve other problems. Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays. Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

However, this paper attempt to compare the practical efficiency of three sorting algorithms – Insertion, Quick and mere Sort using empirical analysis. The result of the experiment shows that insertion sort is a quadratic time sorting algorithm and that it's more applicable to subarray that is sufficiently small. The merge sort performs better with larger size of input as compared to insertion sort. Quicksort runs the most efficiently.

TABLE OF CONTENT

1.0 Introduction

Sorting is a classic subject in computer science.

Suppose two algorithms perform the same task, in this case sorting (merge sort vs. insertion sort vs. quick sort). Which one is better? To answer this question, we started by using a theoretical approach to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, we could see how fast an algorithm's execution time increases as the input size increases, so we compared two algorithms by examining their growth rates.

We went further to confirm the practical efficiency of these algorithms by implementing these algorithms in C# and run the programs to get their execution time. In other to get a reliable data from our run-time experiment, the following conditions were meant:

1. First, we killed other running concurrent processes that could affect the execution time of the algorithm.

2. Second, since the execution time depends on specific input, we made use of sufficiently large input for the experiment.

2.0 Description of Algorithm

The C# API contains several overloaded sort methods for sorting primitive type values and objects in the System.Collections class. For simplicity, this section assumes:

1. Data to be sorted are integers,
2. Data are stored in an array, and
3. Data are sorted in ascending order.

The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in an ArrayList or a LinkedList.

There are many algorithms for sorting. This paper introduces insertion sort, quick sort and merge sort.

2.1 INSERTION SORT

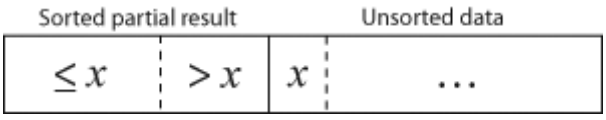
Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort or merge sort. However, insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is O($n + d$), where d is the number of inversions.
- More efficient in practice than most other simple quadratic (i.e., O (n^2)) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is O (n)
- Stable; i.e., does not change the relative order of elements with equal keys

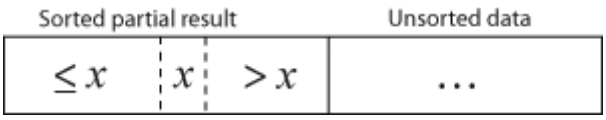
Insertion Sort Algorithm

Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain.

Sorting is typically done in-place. The resulting array after k iterations has the property where the first $k + 1$ entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result:



becomes



with each element greater than x copied to the right as it is compared against x.

The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called *Insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform an insertion sort, begin at the left-most element of the array and invoke *Insert* to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

Pseudocode

```
for j   ← 2 to length[n] do
  key  ← A [j ]
  i  ← j   - 1
  while i  > 0 and A [i ] > key do
      A [i  + 1] ← A [i ]
    i  ← i  - 1
    A [i + 1] ← key
```

Example:

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

Worse Case Analysis

The worst case input is an array sorted in reverse order. In this case, line 5 will be executed *j* times for each *j*.

```
1.    for j  ← 2 to length[n] do                    (n)
2.       key  ← A [j ]                              (n-1)
3.       i  ← j  - 1                                (n-1)
4.       while i  > 0 and A [i ] > key do           (∑_{j=2}^{n} time(j))
5.          A [i  + 1] ← A [i ]                     (∑_{j=2}^{n} (time(j) − 1))
6.        i  ← i  - 1                               (∑_{j=2}^{n} (time(j) − 1))
7.        A [i + 1] ← key                           (n-1)
```

$$\sum_{j=2}^{n} time(j) = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (time(j) - 1)) = \frac{n(n-1)}{2}$$

Let T (*n*) be total number of operations

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

$$= n + n\text{-}1 + n\text{-}1 + \frac{n^2}{2} + \frac{n}{2} - \frac{2}{2} + \frac{n^2}{2} - \frac{n}{2} + \frac{n^2}{2} - \frac{n}{2} + n\text{-}1$$

$$= \frac{3n^2}{2} + \frac{7n}{2} - 4$$

$$\approx an^2 + bn + c$$

$$\approx \Theta(n^2)$$

2.2 MERGE SORT

Next, we present the merge sort algorithm, a comparison model sorting algorithm based on the **divide and conquer** technique. Using this technique, the merge sort algorithm divides the array into two halves; recursively sorts each half and then merges the two sorted halves. Obviously it will be better if we divide a list into two sub-lists with equal length and then sort them. If they remain too large, we can continue breaking them down until we get to something very easy to sort as shown in the Figure 2.0.

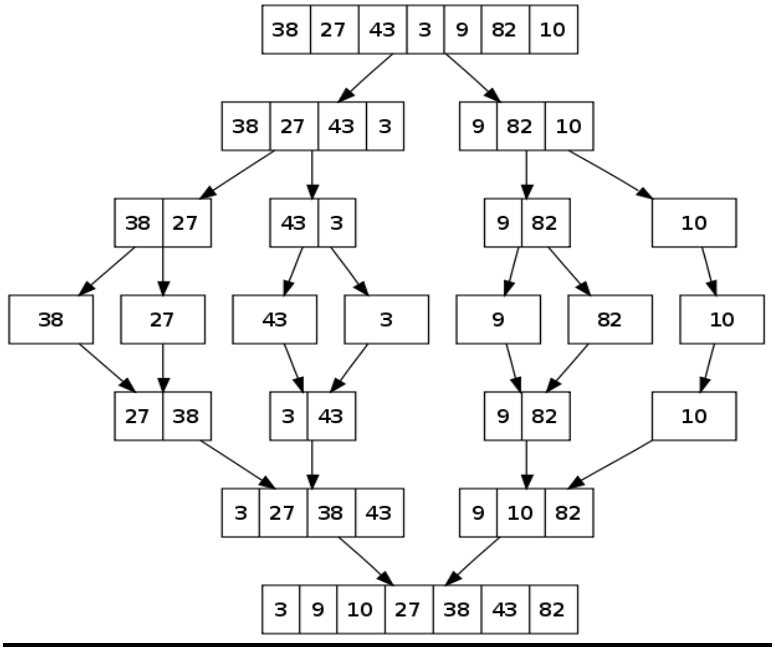


Figure2.0

Merge Sort Algorithm

We now describe the recursive approach **MergeSort** approach that takes ***array***, the array to sort, ***aux,*** an auxiliary array to use for the merge, ***left,*** the starting index of the subarray to sort, ***right***, the ending index of the subarray to sort,

Conceptually, a merge sort works as follows

1. Divide the unsorted array into two halves subarrays, just involves finding the average of the left and right indices. Then the two halves are sorted recursively.

2. We now discuss how the two sorted halves are merged. There is an index ***i*** for the current position of the left half of ***array***, an index ***j*** for the current position of the right half of ***array***, and the next position k to fill in ***aux***.

3. Until the end of either the left or right half of ***array*** is reached, whichever elements is smaller between ***a[i]*** and ***a[j]*** is copied into the next slot of ***aux,*** and the appropriate indices are incremented.

4. Finally, the remainder of the left half of ***array***, if any, is copied into ***aux,*** and finally all used portions of ***aux*** are copied back onto data.

Pseudocode

MERGE-SORT*(A, p, r)*
1 **if** *p* < *r*
2 **then** *q* ← _*(p* + *r)*/2_
3 MERGE-SORT*(A, p, q)*
4 MERGE-SORT*(A, q* + 1, *r)*
5 MERGE*(A, p, q, r)*

```
Input: array, aux, left and right
Output: array such that array[i] <= array[i+1]
MERGE-SORT(array, aux, left, right)
1. if (left < right)
2.    then mid ← ⌊(left+right)/2⌋
3. i←left; j ← mid + 1; k ← left
4. MERGE-SORT(array, aux, left, mid)
5. MERGE-SORT(array, aux, mid+1, right)
6. While (i ≤ mid AND j ≤ right) do
7.   if array[i] < array[j]
8.      then aux[k] = array[i]
9.           i ←i+1
10.     else aux[k] = array[j++]
11.          j ←j+1
12.  K ←k + 1
13. Copy rest of left half to aux
14. Copy used portion of aux back into array
```

Worse Case Analysis

We now briefly discuss the time complexity analysis for merge sort. We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which con- tributes $2T(n/2)$ to the running time.

Combine: $\Theta(n)$ is used for other computation(besides the recursive calls), so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

Using the "master theorem," it can be shown that for this recurrence $T(n) = \Theta(n \log n)$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

2.3 QUICK SORT

Quicksort, like merge sort, is based on the divide-and-conquer paradigm introduced in Section 2.0.1. Recall that merge sort splits the array into two equal halves, recursively sorts the halves, and then merges the sorted subarrays. So merge sort does very little computation before the recursive calls are made, and performs of the required work after the recursive calls have completed. In contrast, quicksort performs all of its extra computation prior to making the recursive calls.

Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p\,..\,r]$.

Divide: Partition (rearrange) the array $A[less\ .\ .\ .\ greater]$ into two (possibly empty) subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ such that each element of $A[p\,..\,q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1\,..\,r]$. Compute the index q as part of this partitioning procedure.

less greater

Conquer: Sort the two subarrays $A[less\ .\ .\ .\ pivot-1]$ and $A[pivot+1\ .\ .\ .\ greater]$ by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[less\ .\ .\ greater]$ is now sorted.

Pseudocode

The following procedure implements quicksort.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, length[A])$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subar- ray $A[p\,..\,r]$ in place.

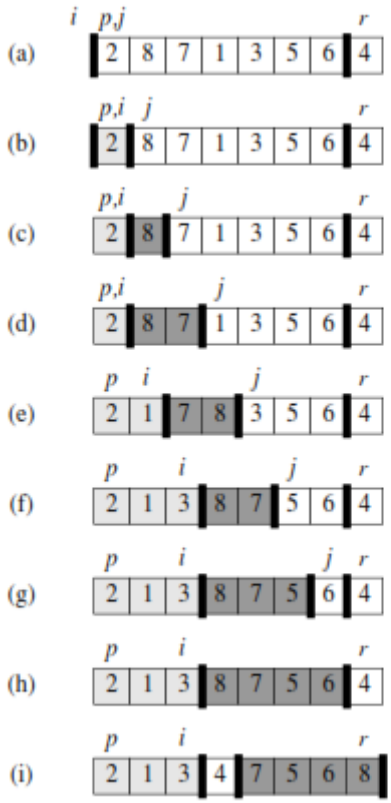
```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

At the beginning of each iteration of the loop of lines 3–6, for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Description of Quicksort

The figure below shows the operation of PARTITION on an 8-element array. PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p \,..\, r]$. As the procedure runs, the array is partitioned into four (possibly empty) regions. At the start of each iteration of the **for** loop in lines 3–6, each region satisfies certain properties, which we can state as a loop invariant:



Worse Case Analysis

Let $T(n)$ be the worst-case time for QUICK SORT on input size n. We have a recurrence

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) \qquad \text{--------- 1}$$

Where q runs from 1 to n-1, since the partition produces two regions, each having size at least 1.

Now we guess that $T(n) \le cn^2$ for some constant c.

Substituting our guess in equation 1.We get

$$\begin{aligned} T(n) &= \max_{1 \le q \le n-1} (cq^2) + c(n - q^2)) + \Theta(n) \\ &= c \max (q^2 + (n - q)^2) + \Theta(n) \end{aligned}$$

Since the second derivative of expression $q^2 + (n-q)^2$ with respect to q is positive. Therefore, expression achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints. This gives the bound max $(q^2 + (n - q)^2))\ 1 + (n-1)^2 = n^2 + 2(n-1)$.

Continuing with our bounding of $T(n)$ we get

$$\begin{aligned} T(n) &\le c\,[n^2 - 2(n-1)] + \Theta(n) \\ &= cn^2 - 2c(n-1) + \Theta(n) \end{aligned}$$

Since we can pick the constant so that the $2c(n-1)$ term dominates the $\Theta(n)$ term we have

$$T(n) \le cn^2$$

Thus the worst-case running time of quick sort is $\Theta(n^2)$.

3.0 Experimental Setup

This experiment was conducted in the following environment:

Operating System: Windows 7 Home Premium 64-bit
RAM: 8 GB
HDD: 640 GB
Processor: Intel x-64 Core i5 2.3 Ghz
Compiler: Visual C#.Net
Language: C#

The purpose of this experiment is to compare the practical efficiency of the aforementioned algorithms using empirical analysis. Consequently, using the dictate of the instructor we chose to use the time the program implementing the algorithms in question. We were able to measure the running time of the code fragment by asking for the system's time right before the fragment's start (t_{start}) and just after its completion (t_{finish}) and then computing the differene between the two ($t_{finish} - t_{start}$). In our implementation language C#, we used the code fragment below using the DateTime and TimeSpan class :

```
DateTime start = DateTime.Now;
//SortingMethod(array);
TimeSpan elapsedTime = DateTime.Now - start;
```

The sample size of input (N) was generated by doubling using the pattern (500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000).

Our test bed was generated using random number generator. We took advantage of the random number generator available in the c# programming language and using a seed of 5000 to generate the same instance of the same size. The code fragment used for this is shown below:

```
Random random = new Random(5000); //seed of 5000 is used
```

It might interest you to know that we are aware of the fact that system's time is typically not very accurate, an obvious remedy we applied is by taking our measurement 5 times and then took their average.

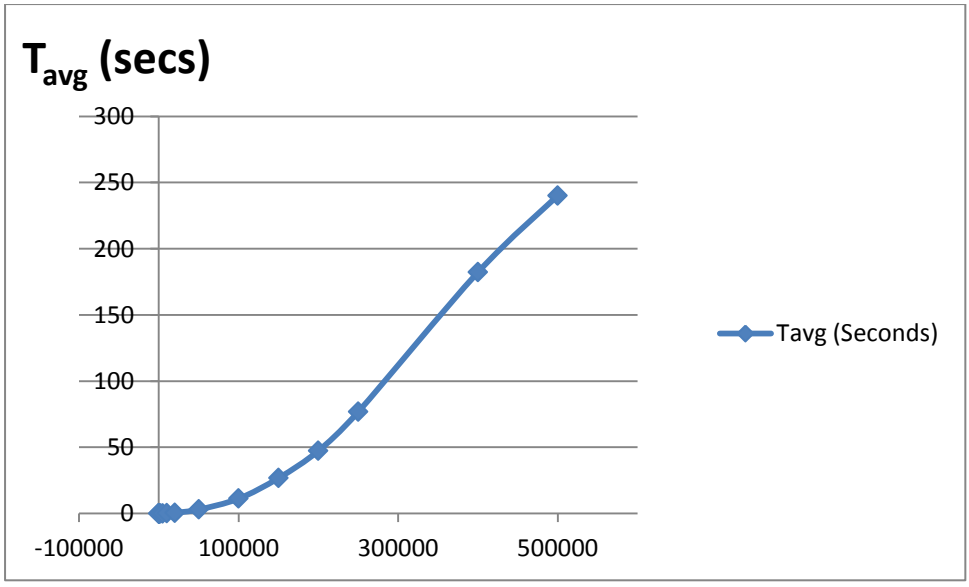
4.0 Experimental Results

In this section, using our tabulated data and respective scatterplot of the results from the experiment for the various algorithm we compared the algorithm and also confirm their theoretical complexity.

Using Microsoft Excel, we made a scatterplot from our table showing the running time (y-axis) vs. your input size (x-axis) of the algorithms on the same graph.

For insertion sort the observation from the experiment is shown in the table below

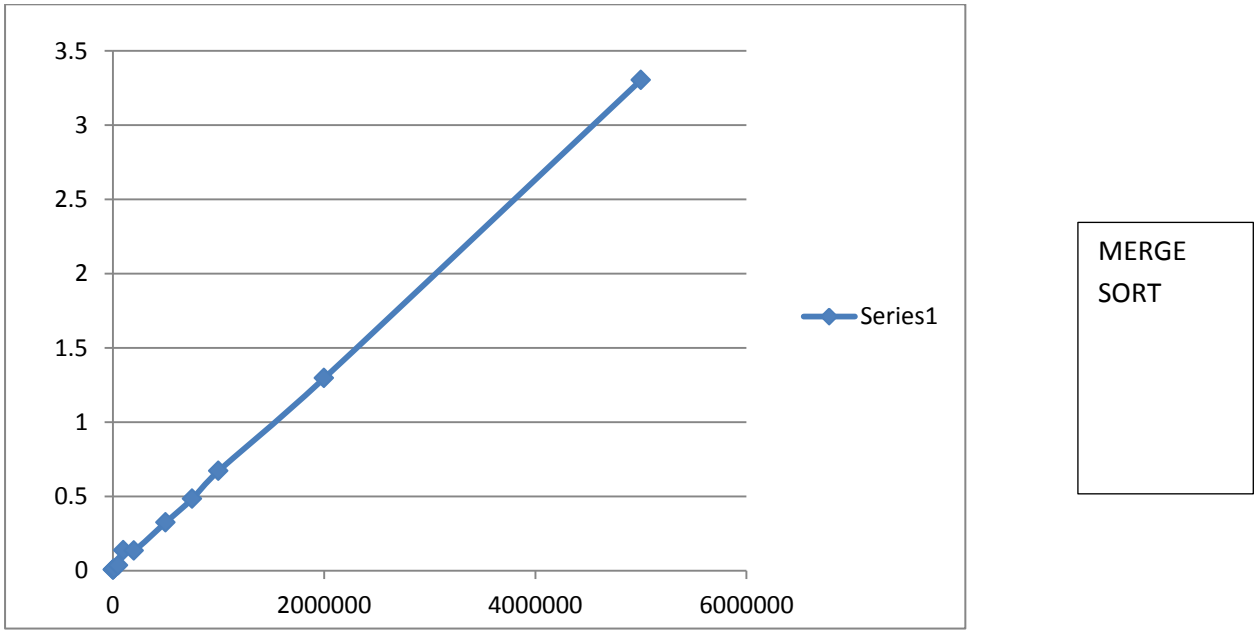
1	INSERTION SORT						
2	N	T_{avg} (Seconds)	T_1 (Seconds)	T_2 (Seconds)	T_3 (Seconds)	T_4 (Seconds)	T_5 (Seconds)
3	0	0	0	0	0	0	0
4	1	0.00220012	0.001	0.0020001	0.0030002	0.0020001	0.0030002
5	10	0.00220012	0.0020002	0.0030001	0.0020001	0.0020001	0.0020001
6	100	0.00220012	0.0020001	0.0020001	0.0030002	0.0020001	0.0020001
7	200	0.00220012	0.0020001	0.0020001	0.0030002	0.0020001	0.0020001
8	300	0.00320018	0.0020001	0.0040002	0.0030002	0.0040002	0.0030002
9	500	0.0040002	0.0040002	0.0040002	0.0040002	0.0040002	0.0040002
10	1000	0.00460028	0.0040003	0.0040002	0.0050003	0.0050003	0.0050003
11	2000	0.0082005	0.0080005	0.0070004	0.0110007	0.0080005	0.0070004
12	5000	0.03960226	0.0340019	0.0430025	0.0430025	0.035002	0.0430024
13	10000	0.1639275	0.1716003	0.1610092	0.1620093	0.1610093	0.1640094
14	20000	0.52382994	0.5440311	0.5150294	0.5170296	0.5210298	0.5220298
15	50000	3.0000776	2.6021488	2.7996901	3.3771932	2.5471457	3.6742102
16	100000	11.20784322	9.6723067	12.4410333	9.436108	11.7991776	12.6905905
17	150000	26.7486723	30.9258971	21.8972529	27.5489381	27.2221436	26.1491298
18	200000	47.32973466	47.5561564	50.5390676	43.0983721	48.5631009	46.8919763
19	250000	76.80605131	74.8940898	75.102227	78.09122	73.6993753	82.24334444
20	400000	182.2410659	178.5367876	168.9396628	195.0177432	163.4514877	205.2596482
21	500000	240.0738259	240.5352302	361.0596514	350.2110309	114.5075495	134.0556675



Insertion Sort

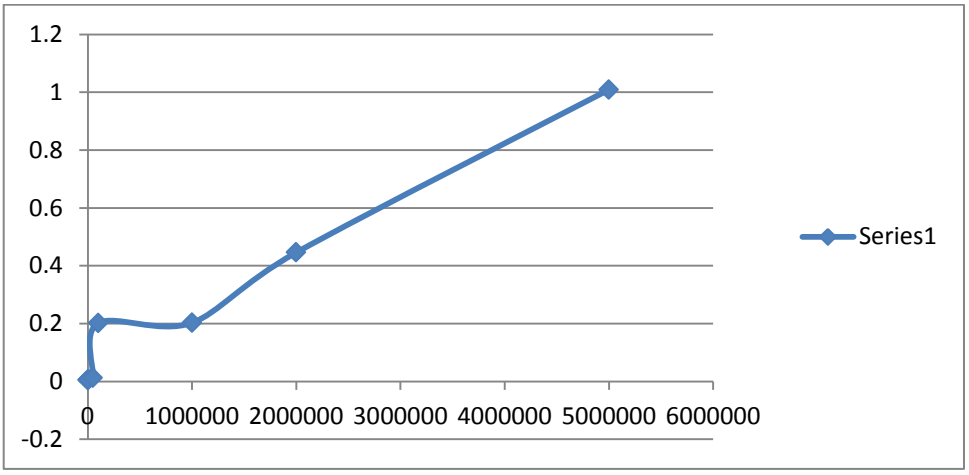
From the chart for insertion sort, we can see that time t increases by 4 as n doubles in order of n^2 which confirms the theoretical analysis.

	MERGE SORT					
N	T_{avg} (Seconds)	T_1 (Seconds)	T_2 (Seconds)	T_3 (Seconds)	T_4 (Seconds)	T_5 (Seconds)
1000	0.00560032	0.0050003	0.0060003	0.0070004	0.0050003	0.0050003
2000	0.00740038	0.0060003	0.0080004	0.0070004	0.0060003	0.0100005
10000	0.01160062	0.0110006	0.0120006	0.0090005	0.0130007	0.0130007
50000	0.03640208	0.0400023	0.035002	0.0410023	0.0330019	0.0330019
100000	0.13700784	0.0580033	0.0850048	0.0730042	0.0650038	0.4040231
200000	0.13520778	0.1320075	0.1280075	0.1290074	0.1460084	0.1410081
500000	0.3250186	0.2950169	0.3440197	0.3180182	0.3160181	0.3520201
750000	0.48342766	0.4470256	0.506029	0.472027	0.4700269	0.5220298
1000000	0.67183842	0.6080347	0.7290417	0.6850393	0.6740385	0.6630379
2000000	1.29567414	1.2650724	1.3530774	1.3010744	1.2910739	1.2680726
5000000	3.30358896	3.3861937	3.2751874	3.2781875	3.2821877	3.2961885



From the chart for merge sort, we can see that the graph is sub-linear with the trend of logn.

QUICK SORT						
N	T_{avg} (Seconds)	T_1 (Seconds)	T_2 (Seconds)	T_3 (Seconds)	T_4 (Seconds)	T_5 (Seconds)
1000	0.0050003	0.0040003	0.0050003	0.0050003	0.0070004	0.0040002
2000	0.00560034	0.0040003	0.0050003	0.0070004	0.0050003	0.0070004
5000	0.0056003	0.0050003	0.0050003	0.0050003	0.0050003	0.0080003
10000	0.0072004	0.0070004	0.0090005	0.0090005	0.0060003	0.0050003
50000	0.01240068	0.0100005	0.0140008	0.0130007	0.0130007	0.0120007
100000	0.20201156	0.1770102	0.2070118	0.2060118	0.2160124	0.2040116
1000000	0.20281158	0.1910109	0.2130121	0.1990113	0.1970113	0.2140123
2000000	0.4458255	0.3910224	0.4270245	0.4530259	0.4840276	0.4740271
5000000	1.00885768	0.9810561	1.0190583	1.0050573	1.0010573	1.0380594



From the chart for quick sort, we can see that the graph is sub-linear with the trend of logn. With N_0 = 5000

5.0 Conclusion

From the experiment we can conclusively say insertion sort would take longer time to sort very large set of data e.g. (one million set of data) but faster when the array is sufficiently small.

Merge sort and quick sort are suitable for larger input sizes but quick sort perform as slow as insertion sort for sufficiently small input.

From our graphs and experiment Quick sort algorithm is slightly faster than Merge sort algorithm and largely faster than the Insertion sort algorithm.

References

1) David S. Johnson. 2001. A Theoretician's Guide to the Experimental Analysis of Algorithms, *http://www.research.att.com/_dsj/*

2) Deitel, J. P. et al., 2012. Java How to Program, Ninth Edition

3) A practical Guide to Data structure and Algorithms Using Java

4) Fundamental of the Analysis of Algorithm Efficiency

5) Wikipedia., 2012. http://en.wikipedia.org/Sorting

Code Listing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FinalAlgorithm
{
    class Program
    {
        static void Main(string[] args)
        {
            const int size = 1000000; //size n
            Random random = new Random(5000);
            int[] array = new int[size];

            for (int i = 0; i < array.Length; i++)
                array[i] = random.Next();

            //Console.WriteLine("Unsorted Arrays are: ");
            DateTime start = DateTime.Now;
            InsertionSort(array); //change when thru with insertion sort to
MergeSort(array) and QuickSort(array, 0, array.Length -1);
            TimeSpan elapsedTime = DateTime.Now - start;
            Console.WriteLine("The elapsed time (in milliseconds) is: " +
elapsedTime.TotalMilliseconds.ToString() + " and " + (elapsedTime.TotalMilliseconds /
1000.0).ToString()+ " Seconds");
            //Console.WriteLine("The elapsed time seconds is: " + elapsedTime
.TotalSeconds);
            //Console.WriteLine("The sorted Arrays are ");


        }


        public static void InsertionSort(int[] array)
        {
            for (int i = 1; i < array.Length; i++)
            {
                /** insert array[i] into a sorted subarray array[0..i-1] so that
array[0..i] is sorted. */
                int currentElement = array[i];
                int k;
                for (k = i - 1; k >= 0 && array[k] > currentElement; k--)
                {
                    array[k + 1] = array[k];
                }

                // Insert the current element into array[k + 1]
                array[k + 1] = currentElement;

            }
        }

        #region
        public static void MergeSort(int[] array)
        {
            if (array.Length > 1)
            {
                // Merge sort the first half
                int[] firstHalf = new int[array.Length / 2];
```

```
            Array.Copy(array, 0, firstHalf, 0, array.Length / 2);
            MergeSort(firstHalf);

            // Merge sort the second half
            int secondHalfLength = array.Length - array.Length / 2;
            int[] secondHalf = new int[secondHalfLength];
            Array.Copy(array, array.Length / 2, secondHalf, 0, secondHalfLength);
            MergeSort(secondHalf);

            // Merge firstHalf with secondHalf
            int[] temp = merge(firstHalf, secondHalf);
            Array.Copy(temp, 0, array, 0, temp.Length);
        }
    }


    /** Merge two sorted arrays */
    private static int[] merge(int[] array1, int[] array2)
    {
        int[] temp = new int[array1.Length + array2.Length];
        int current1 = 0; // Current index in array1
        int current2 = 0; // Current index in array2
        int current3 = 0; // Current index in temp

        while (current1 < array1.Length && current2 < array2.Length)
        {
            if (array1[current1] < array2[current2])
                temp[current3++] = array1[current1++];

            else
                temp[current3++] = array2[current2++];
        }

        while (current1 < array1.Length)
            temp[current3++] = array1[current1++];

        while (current2 < array2.Length)
            temp[current3++] = array2[current2++];

        return temp;
    }
    #endregion

    #region
    /*Quick Sort*/
    public static void QuickSort(int[] array, int first, int last)
    {
        if (last > first)
        {
            int pivotIndex = partition(array, first, last);
            QuickSort(array, first, pivotIndex - 1);
            QuickSort(array, pivotIndex + 1, last);
        }
    }

    private static int partition(int[] array, int first, int last)
    {
        int pivot = array[first]; // Choose the first element as the pivot
        int low = first + 1; // Index for forward search
        int high = last; // Index for backward search
```

```
            while (high > low)
            {
                // Search forward from left
                while (low <= high && array[low] <= pivot)

                    low++;

                // Search backward from right
                while (low <= high && array[high] > pivot)

                    high--;
                // Swap two elements in the array
                if (high > low)
                {
                    int temp = array[high];

                    array[high] = array[low];

                    array[low] = temp;
                }
            }

            while (high > first && array[high] >= pivot)
                high--;

            // Swap pivot with array[high]
            if (pivot > array[high])
            {
                array[first] = array[high];
                array[high] = pivot;


                return high;


            }

            else
            {
                return first;

            }
        }

        #endregion

        static void PrintArray(int[] array)
        {
            int i = 0;
            foreach (int element in array)
            {
                Console.WriteLine(""+ (++i) + " "+ element + " ");
            }
        }
    }
}
```