Volume 3, Issue 3, March 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Comparison of Sorting Algorithms (On the Basis of Average Case)

Pankaj Sareen

Department of Computer Applications,

Baddi University of Emerging Sciences & Technology, India.

Abstract:- Sorting is an important data structure operation, which makes easy searching, arranging and locating the information. I have discussed about various sorting algorithms with their comparison to each other. I have also tried to show this why we have required another sorting algorithm, every sorting algorithm have some advantage and some disadvantage. This paper also shows how to find the running time of an algorithm with the help of C sharp Programming language. I have compared five sorting algorithms (Selection Sort, Insertion Sort, Bubble Sort, Merge Sort and Quick Sort) by comparing their running times calculated by a Program developed in C Sharp Language. I have also compared the sorting algorithms on the basis of various important factors, like complexity, memory, method etc.

Keywords: -Sorting, Quick Sort, Merge Sort, Selection Sort, Insertion Sort, Bubble Sort, Run-time Analysis, Stop Watch.

I. Sorting Algorithm

In computer science, a sorting algorithm is an efficient algorithm which performs an important task that puts elementsof a list in a certain order or arranges a collection of items into a particular order. Sorting data has been developed to arrange the array values in various ways for a database. For instance, sorting will order an array of numbers from lowest to highest or from highest to lowest, or arrange an array of strings into alphabetical order. Typically, it sorts an array into increasing or decreasing order. Most simple sorting algorithms[1] involve two steps which are compare two items and swap two items or copy one item. It continues executing over and over until the data is sorted.

Sorting algorithms are an important part of managing data. Most sorting algorithms work by comparing the data being sorted. In some cases, it may be desirable to sort a large chunk of data (for instance, a struct containing a name and address) based on only a portion of that data. The piece of data actually used to determine the sorted order is called the key. Sorting algorithms are usually judged by their efficiency. In this case, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort. Most of the algorithms in use have an algorithmic efficiency of either O (n^2) or O(n*log(n)).

A. Criteria for Comparison

Many algorithms that have the same efficiency do not have the same speed on the same input.

First, algorithms must be judged based on their average case, best case, and worst case efficiency[2]. Some algorithms, such as quick sort, perform exceptionally well for some inputs, but horribly for others. Other algorithms, such as merge sort, are unaffected by the order of input data.

A second factor is the "constant term". As big-O notation abstracts away many of the details of a process, it is quite useful for looking at the big picture. But one thing that gets dropped out is the constant in front of the expression: for instance, O(c*n) is just O(n). In the real world, the constant, c, will vary across different algorithms. A well-implemented quicksort should have a much smaller constant multiplier than heap sort.

A second criterion for judging algorithms is their space requirement -- do they require scratch space or can the array be sorted in place (without additional memory beyond a few variables)? Some algorithms never require extra space, whereas some are most easily understood when implemented with extra space (heap sort, for instance, can be done in place, but conceptually it is much easier to think of a separate heap). Space requirements may even depend on the data structure used (merge sort on arrays versus merge sort on linked lists, for instance).

A third criterion is stability -- does the sort preserve the order of keys with equal values? Most simple sorts do just this, but some sorts, such as heap sort, do not.

The following table compares sorting algorithms on the

Various criteria outlined above Table I

II. Comparison Of Sorting Algorithms

	Time					
Sort	Avg.	Best	Worst	Space	Stability	Remarks
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Merge Sort	O(n*logn)	O(n*logn)	O(n*logn)	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quick Sort	O(n*logn)	O(n*logn)	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

- Some algorithms (selection, bubble) work by moving elements to their final position, one at a time. You sort an array of size N, put 1 item in place, and continue sorting an array of size N – 1
- Some algorithms (insertion, quick sort) put items into a temporary position, close(r) to their final position. You rescan, moving items closer to the final position with each iteration.
- One technique is to start with a "sorted list" of one element, and merge unsorted items into it, one at a time.
- Complexity and running time[3]
- ▪ Factors: algorithmic complexity, startup costs, additional space requirements, use of recursion (function calls are expensive and eat stack space), worst-case behavior, assumptions about input data, caching, and behavior on already-sorted or nearly-sorted data
- ▪ O (N) time is possible if we make assumptions about the data and don't need to compare elements against each other (i.e., we know the data falls into a certain range or has some distribution).O(N) clearly is the minimum sorting time possible, since we must examine every element at least once

A. Bubble Sort [Best: O (n), Worst: O (N^2)]

Starting on the left, compare adjacent items and keep "bubbling" the larger one to the right (it's in its final
Place). Bubbles sort [4] the remaining N -1 item.

- Though "simple" I found bubble sort nontrivial. In general, sorts where you iterate backwards (decreasing some index) were counter-intuitive for me. With bubble-sort, either you bubble items "forward" (left-to-right) and move the endpoint backwards (decreasing), or bubble items "backward" (right-to-left) and increase the left endpoint. Either way, some index is decreasing.
- You also need to keep track of the next-to-last endpoint, so you don't swap with a non-exist ant item.

***1) Advantage**:* Simplicity and ease-of-implementation

2) Disadvantage: Code inefficient

B. Selection Sort [Best/Worst: O(N^2)]

Scan all items and find the smallest. Swap it into position as the first item. Repeat the selection sort[5] on the remaining N-1 items.

• I found this the most intuitive and easiest to implement — you always iterate forward (i from 0 to N-1), and swap with the smallest element (always i).

1) Advantage: Simple and easy to implement

2) Disadvantage: Inefficient for large lists, so similar to the more efficient insertion sort, the insertion sort should be used in its place.

C. Insertion Sort [Best: O(N), Worst: O(N^2)]

Start with a sorted list of 1 element on the left, and N-1 unsorted items on the right. Take the first unsorted item (element #2) and insert it into the sorted list, moving elements as necessary [6]. We now have a sorted list of size 2, and N -2 unsorted elements. Repeat for all elements.

- Like bubble sort, I found this counter-intuitive because you step "backwards"
- This is a little like bubble sort for moving items, except when you encounter an item smaller than you, you stop. If the data is reverse-sorted, each item must travel to the head of the list, and this becomes bubble-sort.
- There are various ways to move the item leftwards — you can do a swap on each iteration, or copy each item over its neighbor

1) Advantage: Relative simple and easy to implement. Twice faster than bubble sort.

2) Disadvantage: Inefficient for large lists.

D.Quicksort [Best: O(N lg N), Avg: O(N lg N), Worst:O(N^2)]

There are many versions of Quicksort[7], which is one of the most popular sorting methods due to its speed (O (N lgN) average, but O (N^2) worst case). Here's a few:

1) Using external memory:

- Pick a "pivot" item
- Partition the other items by adding them to a "less than pivot" sublist, or "greater than pivot" sublist
- The pivot goes between the two lists
- Repeat the quicksort on the sublists, until you get to a sublist of size 1 (which is sorted).
- Combine the lists — the entire list will be sorted

2) Using in-place memory:

- Pick a pivot item and swap it with the last item. We want to partition the data as above, and need to get the pivot out of the way.
- Scan the items from left-to-right, and swap items greater than the pivot with the last item (and decrement the "last" counter). This puts the "heavy" items at the end of the list, a little like bubble sort.
- Even if the item previously at the end is greater than the pivot, it will get swapped again on the next iteration.
- Continue scanning the items until the "last item" counter overlaps the item you are examining – it means everything past the "last item" counter is greater than the pivot.
- Finally, switch the pivot into its proper place. We know the "last item" counter has an item greater than the pivot, so we swap the pivot there.
- Now, run quicksort again on the left and right subset lists. We know the pivot is in its final place (all items to left are smaller; all items to right are larger) so we can ignore it.

3) Using in-place memory with two pointers:

- Pick a pivot and swap it out of the way
- Going left-to-right, find an oddball item that is greater than the pivot
- Going right-to-left, find an oddball item that is less than the pivot
- Swap the items if found, and keep going until the pointers cross — re-insert the pivot
- Quicksort the left and right partitions
- Note: this algorithm gets confusing when you have to keep track of the pointers and where to swap in the pivot

4) Advantage: Fast and efficient

5) Disadvantage: Show horrible result if list is already sorted.

E. Merge Sort [Best: O(N lg N), Avg: O(N lg N), Worst:O(N^2)]

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems[8].

To sort $A[p .. r]$:

1) Divide Step If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2) Conquer Step

Conquer by recursively sorting the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.

3) ***Combine Step***

Combine the elements back in *A*[*p* .. *r*] by merging the two sorted subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

4) Advantage: Well suited for large data set.

5) Disadvantage: At least twice the memory requirements than other sorts

III. Comparison By Using Code Written In C Sharp Language

Now, I will determine the efficiency of the various sorting algorithms according to the time[9] by using randomized trails. The build environment will be built using the C# language in Asp.Net Framework. I will discuss and implement several sorting algorithms such as bubble sort[10], selection sort, and insertion sort and will also include complexity sort such as quick sort and merge sort. I will represent these algorithms as a way to sort an array or integers and run random trails of length. The research will provide the runtime of each sorting algorithm.

To investigate, I create a namespace called "ConsoleApplication1" which contains one class "SortComparison". This class contains various Functions for Selection Sort, Insertion Sort, Quick Sort, Bubble Sort and Merge Sort. In Main () function I will be using Random Number Generator for generating the number of elements. I will be using the StopWatch[11] Class of the System.Diagnostics Namespace which will help me to find the running time of the algorithm in microseconds.

```
int[] arr_selection = new int[10000];          //Number of elements are 10000
int[] arr_insertion = new int[10000];
int[] arr_bubble = new int[10000];
int[] arr_merge = new int[10000];
Random rn = new Random();
for (int i = 0; i < arr_selection.Length; i++)
{
   arr_selection[i] = rn.Next(1, 10000          // Random Number for generating 10000 elements
}
```

Similarly we can generate random numbers for different sorting algorithms

```
System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
sw.Start();
          // Sorting Function to be called
sw.Stop();
long timeselection = sw.ElapsedTicks/(System.Diagnostics.Stopwatch.Frequency/(1000L * 1000L));
```

Table II

timeselection is the time in the microseconds. I will be calling each sorting function to find the running time of that sorting algorithm so that I can compare the running time of the algorithms. For this I passed different number of elements (N=10, 100, 1000, 10000) to the sorting Functions. I ran the program five times for each value of N (i.e. 10 or 100 or 1000 or 10000) and tried to find the running time of each sorting algotithm).

Table II shows the running time of each algorithm for first, second, third, fourth and Fifth Run. I have also calculated the average running time (In Microseconds) based upon the running time.

I have used three charts for comparing the sorting algorithms. First Chart (fig.1) compares all the sorting algorithms for the small values of N=10 and N=100. Second Chart (fig.2) compares all the sorting algorithms for the large values of N=1000 and Third Chart (fig.3) compares all the sorting algorithms for the extra large values of N=10000.

First Run(Time in Microseconds)					
N	Selection Sort	Insertion Sort	Bubble Sort	Merge Sort	Quick Sort
10	267	223	208	535	318
100	298	250	290	598	337
1000	3686	3286	8393	3416	605

10000	348511	263315	809311	91848	3316
Second Run(Time in Microseconds)					
10	307	219	206	543	320
100	563	442	519	1073	609
1000	3924	2805	8308	4410	617
10000	343363	264876	809943	103267	3366
Third Run(Time in Microseconds)					
10	265	318	223	601	337
100	337	245	291	598	335
1000	3728	2939	8402	4005	609
10000	336569	294503	802286	142748	3335
Fourth Run(Time in Microseconds)					
10	341	241	208	550	319
100	351	250	285	604	335
1000	6799	5424	15120	7207	1105
10000	339655	263865	802645	104050	3342
Fifth Run(Time in Microseconds)					
10	279	224	207	550	337
100	341	245	283	597	334
1000	7009	5340	10071	3701	608
10000	338079	264992	812777	92377	3675
Average					
10	291.8	245	210.4	555.8	326.2
100	378	286.4	333.6	694	390
1000	5029.2	3958.8	10058.8	4547.5	708.8
10000	341235.4	270310.2	807392.4	106858	3406.8

Fig.I

We can see in Fig. that for the small values of N i.e N=10, 100; Merge Sort is taking the maximum time.

For N=10000, again Bubble Sort and Selection Sort are taking the Maximum Time as shown in Fig. III. We can observe from the fig. that Quick Sort is taking the least time in all the cases. So we can say that from all the sorting algorithms we have compared, Quick Sort is efficient.

IV. Conclusion

In this study we have studied about various sorting algorithms and their comparison. There is advantage and disadvantage in every sorting algorithm. To find the running time of each sorting algorithm I used one Program for comparing the running time (in Microseconds). After running the same program on five different runs (for each different value of N=10, 100, 1000, 10000), I calculated the average running time for each algorithm and then showed the result with the help of a chart. From the chart I can conclude that Quick Sort is the most efficient algorithm

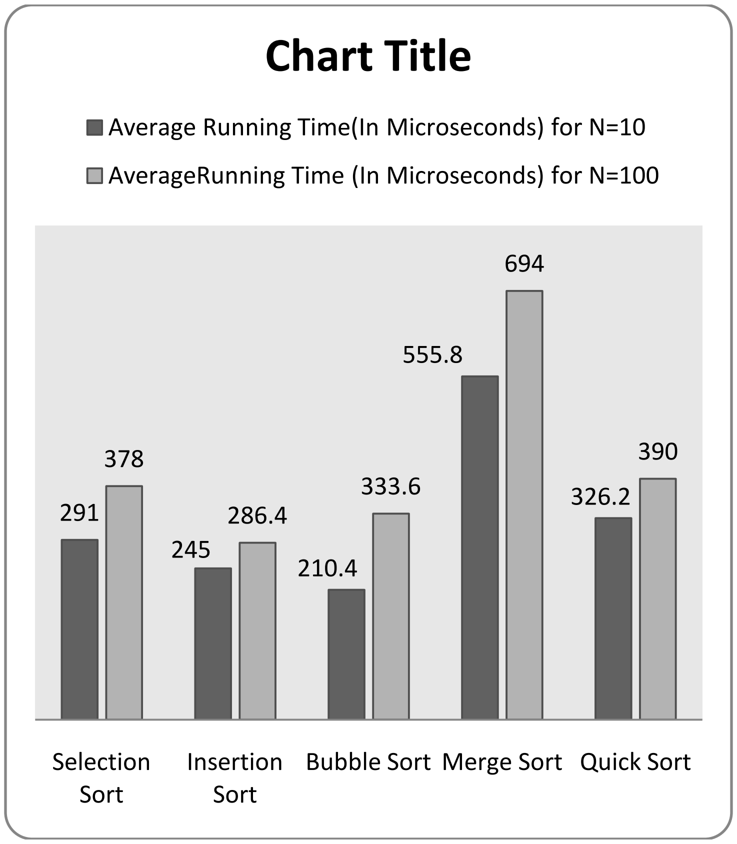

Fig. I: We can see in Fig. that for the small values of N i.e N=10, 100; Merge Sort is taking the maximum time.

For N=1000, Bubble Sort is taking the Maximum Time as shown in Fig. II.

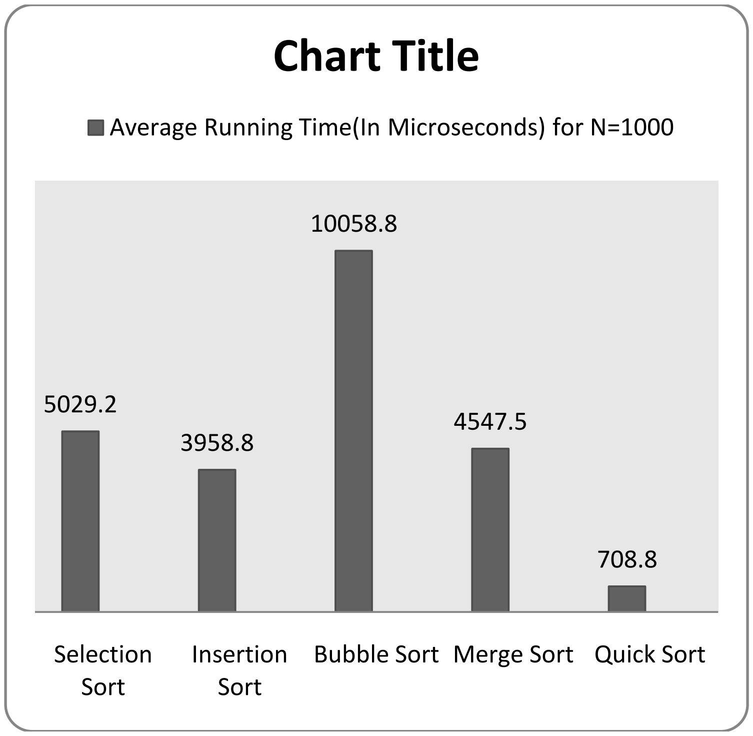

Fig.II

Another efficient algorithm for the large values of N is Merge Sort. But Merge sort has at least twice memory requirements than other sorting algorithms.

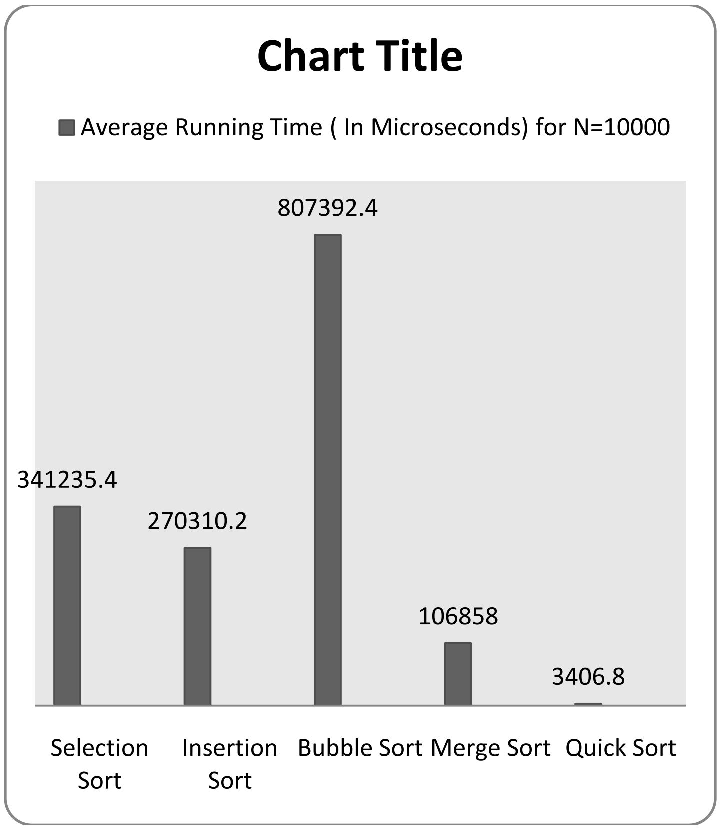

Fig.III

References

[1] Sedgewick, Algorithms in C++, pp.96-98, 102, ISBN 0-201-51059-6 ,Addison-Wesley , 1992
[2] http://www.cs.ucf.edu/courses/cop3502/nihan/spr03/sort.pdf
[3] http://www.cs.manchester.ac.uk/ugt/COMP26912/lecture/lecture- sorting.pdf
[4] http://en.wikipedia.org/wiki/Bubble_sort
[5] http://en.wikipedia.org/wiki/Selection_sort
[6] http://en.wikipedia.org/wiki/Insertion_sort
[7] http://en.wikipedia.org/wiki/Quicksort
[8] http://en.wikipedia.org/wiki/Merge_sort
[9] http://www.scribd.com/doc/45996720/Run-Time-Analysis-of-Insertion- Sort-and-Quick-Sort
[10] Owen Astrachan, Bubble Sort: An Archaeological Algorithmic Analysis, SIGCSE 2003, http://www.cs.duke.edu/~ola/papers/bubble.pdf
[11] http://www.dotnetperls.com/stopwatch
[12] http://www.c-sharpcorner.com/uploadfile/9f4ff8/use-of-stopwatch-class- in-C-Sharp/

Appendix

/* Program that will show the use of Sorting Algorithms (Selection Sort , Insertion Sort , Bubble Sort , Merge Sort and Quick Sort) and compares the running time of these algorithms with the help of StopWatch Class of System.Diagnostics NameSpace*/

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

namespace ConsoleApplication1
{
```

```
class SortComparison
{
    static void Main(string[] args)
    {
        int[] arr_selection = new int[10];   // change the value here if you want to run the code for 100 or 1000 elements
        int[] arr_insertion = new int[10];
        int[] arr_bubble = new int[10];
        int[] arr_merge = new int[10];
        int[] arr_quick = new int[10];
        Random rn = new Random();
        for (int i = 0; i < arr_selection.Length; i++)
        {
            arr_selection[i] = rn.Next(1, 10000);
        }

        /********************* Selection Sort
*************************************************************************/
        System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
        sw.Start();
        selectionsort(arr_selection,arr_selection.Length);
        sw.Stop();
        long timeselection = sw.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
        Console.WriteLine("time taken by selection sort is:{0} microseconds",timeselection);

/***************************************************************************************************
**********/

        /********************* Insertion Sort
*************************************************************************/
        Random rn1 = new Random();
        for (int i = 0; i < arr_insertion.Length; i++)
        {
            arr_insertion[i] = rn1.Next(1, 10000);
        }
        System.Diagnostics.Stopwatch sw1 = new System.Diagnostics.Stopwatch();
        sw1.Start();
        insertionsort(arr_insertion, arr_insertion.Length);
        sw1.Stop();
        long timeinsertion = sw1.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
        Console.WriteLine("time taken by insertion sort is:{0} microseconds", timeinsertion);

/***************************************************************************************************
**********/

        /******************** Bubble Sort
*************************************************************************/

        Random rn2 = new Random();
        for (int i = 0; i < arr_bubble.Length; i++)
        {
            arr_bubble[i] = rn2.Next(1, 10000);
        }
        System.Diagnostics.Stopwatch sw2 = new System.Diagnostics.Stopwatch();
        sw2.Start();
        bubblesort(arr_bubble, arr_bubble.Length);
        sw2.Stop();
        long timebubble = sw2.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
        Console.WriteLine("time taken by bubble sort is:{0} microseconds", timebubble);
```

```
/*************************************************************************************************
**********/

        /********************* Merge Sort
***********************************************************************/

        Random rn3 = new Random();
        System.Diagnostics.Stopwatch sw3 = new System.Diagnostics.Stopwatch();
        for (int i = 0; i < arr_merge.Length; i++)
        {
          arr_merge[i] = rn3.Next(1, 10000);
        }
        sw3.Start();
        mergesort(arr_merge, 0, arr_merge.Length - 1);
        sw3.Stop();
        long timemerge = sw3.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
        Console.WriteLine("time taken by merge sort is:{0} microseconds", timemerge);

/*************************************************************************************************
**********/

        /********************* Quick Sort
***********************************************************************/

        Random rn4 = new Random();
        System.Diagnostics.Stopwatch sw4 = new System.Diagnostics.Stopwatch();
        for (int i = 0; i < arr_quick.Length; i++)
        {
          arr_quick[i] = rn3.Next(1, 10000);
        }
        sw4.Start();
        quicksort(arr_quick, 0, arr_quick.Length - 1);
        sw4.Stop();
        long timequick= sw4.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
        Console.WriteLine("time taken by quick sort is:{0} microseconds", timequick);

/*************************************************************************************************
**********/
        Console.ReadKey();
      }
      static void selectionsort(int[] arr, int size)
      {
        int small, pos, tmp, i, j;
        for (i=0; i<size; i++)
        {
           small = arr[i];
           pos = i;
           for (j=i+1; j<size; j++)
             {
                if (arr[j] < small)
                {
                  small = arr[j];
                    pos = j;
                }
             }
           tmp = arr[i];
          arr[i] = arr[pos];
          arr[pos] = tmp;
          }
       }
      static void insertionsort(int[] a, int n)
       {
           int i, k, y;
          for (k=1; k<n; k++)
          {
               y = a[k];
              for (i=k-1; i>=0 && y<a[i]; i--)
                    a[i+1] = a[i];
             a[i+1] = y;
           }
       }
      static void bubblesort(int[] a, int size)
       {
           int temp, i, j;
           for (i=0; i<size-1; i++)
              for (j=0; j<size-1-i; j++)
                 if (a[j]>a[j+1])
                 {
                        temp = a[j];
                        a[j] = a[j+1];
                         a[j+1] = temp;
                 }
       }


static void mergesort(int[] arr,int low,int high)
{
   int mid;
   if(low<high)
            {
      mid=(low+high)/2;
               mergesort(arr,low,mid);
      mergesort(arr,mid+1,high);
               sort(arr,low,mid,high);
  }
}
static void sort(int[] arr, int low, int mid, int high)
{
   int[] b = new int[arr.Length];
   int i, j, k, l;
   l = low;
   i = low;
   j = mid + 1;
   while ((l <= mid) && (j <= high))
   {
      if (arr[l] <= arr[j])
      {
         b[i] = arr[l];
         l++;
      }
      else
      {
         b[i] = arr[j];
         j++;
      }
      i++;
   }
```

```
    if (l > mid)
    {
        for (k = j; k <= high; k++)
        {
            b[i] = arr[k];
            i++;
        }
    }
    else
    {
        for (k = l; k <= mid; k++)
        {
            b[i] = arr[k];
            i++;
        }
    }
    for (k = low; k <= high; k++)
    {
        arr[k] = b[k];
    }
}
 static void quicksort(int[] x,int first,int last){
    int pivot,j,temp,i;

     if(first<last){
         pivot=first;
         i=first;
         j=last;

         while(i<j){
             while(x[i]<=x[pivot]&&i<last)
                 i++;
             while(x[j]>x[pivot])
                 j--;
             if(i<j){
                 temp=x[i];
                  x[i]=x[j];
                  x[j]=temp;
             }
         }

         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,first,j-1);
         quicksort(x,j+1,last);

    }
}


    }
}
```