ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE
International Journal of Advanced Research in Computer and Communication Engineering
ICACTRP 2017
International Conference on Advances in Computational Techniques and Research Practices
Noida Institute of Engineering & Technology, Greater Noida
Vol. 6, Special Issue 2, February 2017

Performance Analysis of Various Sorting Algorithms

Deepak Kumar Pathak[1], Vivek Kumar Sharma[2], Mansi Rastogi[3]

Department of MCA, Future Group of Institutes, Bareilly (U.P.), India[1,3]

Department of C.S.E., Noida Institute of Engineering & Technology, Greater Noida (U.P.), India [2]

Abstract: Sorting is an essential data structure operation, which performs easy searching, arranging and locating the information. I have deliberated about various sorting algorithms with their performance analysis to each other. I have also strained to show this why we have required another sorting algorithm; all sorting algorithm have some advantage and some disadvantage. This paper also illustrations how to find the running time of an algorithm with the help of C Sharp Programming language. I have compared five sorting algorithms (Heap Sort, Shell Sort, Bubble Sort, Merge Sort and Quick Sort) by performance analysis their running times calculated by a Program developing in C Sharp Language. I have analyze the performance of sorting algorithms by numerous essential factors, like complexity, memory, method, etc.

Keywords: Sorting, Heap Sort, Shell Sort, Bubble Sort, Merge Sort, Quick Sort, Stop Watch, Performance Analysis.

I. INTRODUCTION

Algorithms have a vital and significant role in solving the computational problems. Generally, an algorithm is a well-formed computational procedure that takes input and provides output. Algorithm is a sequence of steps or is a tool to solve the computational problems. The presence of algorithms goes way back as they were in presence even before the presence of computers. There are numerous techniques and methodologies which are based on different kinds of algorithms. We talk about the sorting algorithms Out of all problem-solving algorithms. In sorting procedure case, it is essential to arrange a sequence of numbers into a specified order, generally non-decreasing. In computer science, an algorithm that positions elements of a list into an order is known as a sorting algorithm. Numerical order and lexicographical order are mainly used order in sorting. For making a practice of other algorithms (like search and merge algorithms) sorted lists are required to work correctly and efficiently; it is also often useful for conforming to well-established patterns or rules of data and for generating such output which is easy to read and recognise. There are two conditions recruited that output essentially satisfy.

These conditions are:

1) The output is provided in a non-decreasing order i.e. each element is greater than the earlier element in the preferred order.
2) The output is in a permutation or reordering of the input.

Since the origin of computing, the sorting problem has attracted a great deal of research, worked efficiently due to the complexity of solving it. For example, bubble sorting algorithm was analysed as early as 1956. Although many cogitate it a resolved problem, advantageous new sorting algorithms are still being invented (for example, library sort was first issued in the year 2004).

Sorting algorithms are prevailing in introductory computer science classes, where the lavishness of algorithms for the problem provides a moderate introduction to a variety of principal algorithm theories, such as big O notation, data structures, divide and conquer algorithms, randomized algorithms, best, time-space tradeoffs, worst and lower bounds, and average case analysis.

II. IN COMPUTER SCIENCE, SORTING ALGORITHMS ARE NORMALLY CLASSIFIED BY

- Computational complexity (worst, average and best behaviour) of element comparisons regarding the size of the list. For archetypal sorting algorithms good behaviour is O(n log n) and bad behaviour is O(n2).
- Memory usage (and use of other computer resources): In specific, certain sorting algorithms are "in place". it means, they need only O(1) memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in further sorting algorithms.
- Recursion: Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

A. Aims of the Algorithms:
The algorithm had several aims:

- Speed.
- Good memory utilization. The elements that can be sorted should closely approach the machine's physical limits.
- For the algorithm to be really common purpose the only operator that will be assumed is a binary comparison.
- To obtain good memory utilization, sorting of small elements linked lists are avoided. Thus, the lists of elements referred to below are implemented using arrays, deprived of any storage overhead for pointers.

III. SUMMARIES OF SORTING ALGORITHMS

A. Shell sort[1][2]
Shell sort can be thought of as a more efficient variation of insertion sort; it achieves this mainly by comparing items of varying distances apart resulting in a worst run time complexity of O ($n\log^2 n$). Shell sort is fairly straight forward but may seem somewhat confusing at first as it differs from other sorting algorithms in the way it selects items to compare.

Pros:
1) Time complexity of the algorithm is O(nlogn).
2) Auxiliary Space required for the algorithm is O(1).
3) Efficient for large element list and it requires relatively small amount of memory, extension of insertion sort.

Cons:
1) More constraints, not stable.

B. Heap sort [1][2]
Heap sort is the most efficient version of selection sort. It also sort by determining the largest (or smallest) element of the element list, placing that at the end (or beginning) of the element list, then continuing with the remaining element list, but completes this task efficiently by using a data structure called a heap, extraordinary type of binary tree. Once the data element list has been made into a heap, the root node is definite to be the largest (or smallest) element. When it is removed and positioned at the end of the element list, the heap is repositioned so the largest element remaining moves to the root. Finding the next largest element takes O(log n) time by using the heap, instead of O(n) for a linear scan as in simple selection sort. This allows, Heap sort to run in O(nlog n) time, and this is also the worst case complexity.

Pros:
1) Time complexity of the algorithm is O(nlog n).
2) Auxiliary Space required for the algorithm is O(1).
3) In-space and non-recursive makes it a good choice for large data sets.

Cons:
1) Works slowly than other such DIVIDE-AND-CONQUER sorts that also have the same O (n log n) time complexity due to cache behaviour and other factors.
2) Unable to work when dealing with linked lists due to non-convertibility of linked lists to heap structure.

C. Bubble Sort [1][2]
Bubble sort is a modest sorting algorithm. This algorithm starts at the beginning of the data element set. It compares the first two elements of data element set, and if the second is smaller than the first, then it swaps them. It continues doing this for each pair of adjacent elements to the end of the data element set. It then starts again with the first two elements, iterating until no swaps have happened on the last pass. Average and worst-case performance is O(n2) of this algorithm, so it is hardly used to sort huge, unordered, data element sets. This causes larger values to "bubble" to the end of the element list while smaller values "sink" towards the start of the element list. Bubble sort algorithm can be used to sort a small number of items (where its inefficiency is not a great penalty). Bubble sort may be efficiently used on an element list that is already sorted except for a very small number of elements list. For example, if only one element is unordered, bubble sort will take only 2n time. If two elements are unordered, bubble sort will take only at most 3n time. Bubble sort average and worst case are both $O(n^2)$.

Pros:
1) Simplicity and ease of implementation.
2) Auxiliary Space used is O(1).

Cons:
1) Very inefficient. Average complexity is O(n2) and Best case complexity is O(n).

D. Merge Sort: [1][2]
Merge sort takes advantage of the ease of merging already sorted element lists into a new sorted element list. It starts by comparing every two elements and swapping them if the first should come after second. It then merges each of the resulting element lists of two into element lists of four, and then merges those element lists of four, and so on; until at last two element lists are merged into the final sorted element list. Of the algorithms defined here, this is the first that scales well to huge element lists, because its worst running time is O (n log n).

Pros:
1) Marginally faster than the heap sort for larger sets.
2) Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

Cons:
1) At least twice the memory requirements of the other sorts because it is recursive. This is major cause for concern as its space complexity is very high. It needs about a Θ(n) auxiliary space for its working.
2) Function overhead calls (2n-1) are much more than those for quick sort (n). This causes it to take more time marginally to sort the input data.

E. Quick Sort [1][2]
Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot element is selected. All elements are moved before pivot elements which are smaller than pivot element and all greater elements are moved after it. This process can be done efficiently in linear time and in-place. The lesser and greater sub-lists of elements are then recursively sorted. Efficient implementations of quick sort (with in-place partitioning) are typically unstable sorts and to some extent complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, quick sort is one of the most standard sorting algorithms and is available in many standard programming libraries. The most complex issue in quick sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower O(n^2) performance, if at every single step the median is elected as the pivot element then the algorithm works in O(n log n). Finding the median however, is an O(n) operation on unsorted lists and therefore exacts its own penalty with sorting.

Pros:
1) One advantage of the parallel quick sort over other parallel sort algorithms is that no synchronization is compulsory. A new thread is started as quickly as a sub-list is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is complete.
2) All comparisons are being done with a single pivot element value, which can be stored in a register.
3) The list is being traversed serially, which produces very good locality of reference and cache behaviour for arrays.

Cons:
1) Auxiliary space used in the average case for implementing recursive task calls is O(log n) and hence proves to be a bit space costly, mainly when it comes to large data element sets.
2) Its worst case time complexity is O(n2) which can prove very fatal for large data sets.

TABLE I: VARIOUS SORTING ALGORITHM

Sort	Best	Average	Worst	Memory	Stable
Shell	n	$n\log^2 n$	$n\log^2 n$	nk+np	No
Heap	n log n	n log n	n log n	nk+np	No
Bubble	n	n^2	n^2	nk+np	Yes
Merge	n log n	n log n	n log n	nk+np+stack	Yes
Quick	n log n	n log n	n^2	nk+np+stack	No

IV. COMPARISON BY USING CODE WRITTEN IN C # LANGUAGE

Now, we will determine the efficiency of the various sorting algorithms according to the time by using randomized trails. The build environment will be constructed using the C# language in Asp.Net Framework. We will discuss and implement numerous sorting algorithms such as bubble, heap sort and shell sort and will also take account of complexity sort such as quick sort and merge sort. We will represent these sorting algorithms as an approach to sort an integers array and execute random trails of length.
To examine, we create a namespace called "SortAlgorithms" which contains one class "SortAlgoComparison". This class contains numerous Functions for Shell Sort, Heap Sort, Bubble Sort, Merge Sort and Quick Sort. In Main () function we will be using Random Number Generator for generating the number of elements for arrays. We will be using the StopWatch[3] Class of the System.Diagnostics Namespace which will help us to find the running time of the algorithm in microseconds. To set array size using integer type variable "N" for following arrays-

```
int N = 10000; // Set the value here if you want to run the
        code for 10,100,1000,10000 or 100000 elements
            int[] arr_shell = new int[N];
            int[] arr_heap = new int[N];
           int[] arr_bubble = new int[N];
            int[] arr_merge = new int[N];
            int[] arr_quick = new int[N];
int[] aux = new int[N];
```

These arrays fill by using "Random" class in following way-

```
            Random rn = new Random();
            for (int i = 0; i < N; i++)
                {
            arr_shell[i] = rn.Next(1, 10000);
            arr_heap[i] = rn.Next(1, 10000);
           arr_bubble[i] = rn.Next(1, 10000);
            arr_merge[i] = rn.Next(1, 10000);
            arr_quick[i] = rn.Next(1, 10000);
}
```

Similarly "SW1" is object of stopwatch to count CPU cycle and "TS" object which calculate sorting execution time in microseconds of all algorithm in following manner[3][4]-

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

```
System.Diagnostics.Stopwatch SW1 = new
System.Diagnostics.Stopwatch();
SW1.Start();
ShellSort(arr_shell);
SW1.Stop();
long TS = SW1.ElapsedTicks/
(System.Diagnostics.Stopwatch.Frequency/(1000L *
1000L));
```

We will be calling each sorting function to discover the running time of that sorting algorithm so that we can compare the running time of the algorithms. We passed a different number of elements (N=10, 100, 1000, 10000, 100000) to the sorting Functions. We executed the program five times for each value of N (i.e. 10 or 100 or 1000, 10000 or 100000) and tried to discover the running time of each sorting algorithms. Table II shows the running time of each algorithm for first, second, third, fourth and Fifth execution. We have also calculated the average running time (In Microseconds) based upon the running time. We have used five charts for comparing the sorting algorithms.

TABLE II: RUNNING TIME OF VARIOUS SORTING ALGORITHM

First Run(Time in Microseconds)					
N	Shell	Heap	Bubble	Merge	Quick
10	418	714	247	583	358
100	442	714	403	600	400
1000	723	1033	8828	1061	873
10000	5184	4785	850375	5979	5488
100000	55193	52014	82920053	62290	59764
Second Run(Time in Microseconds)					
N	Shell	Heap	Bubble	Merge	Quick
10	413	720	285	584	376
100	421	697	324	564	438
1000	725	1135	9428	1049	738
10000	4786	4865	841813	5819	5406
100000	54669	52809	83145365	62204	59440
Third Run(Time in Microseconds)					
N	Shell	Heap	Bubble	Merge	Quick
10	400	749	256	565	474
100	414	714	378	571	372
1000	698	1025	8574	1033	789
10000	6058	4870	634211	5875	5254
100000	54971	52550	83115290	63525	56951
Forth Run(Time in Microseconds)					
N	Shell	Heap	Bubble	Merge	Quick
10	464	1045	311	583	383
100	444	701	362	652	408
1000	699	1279	9101	1115	783
10000	5850	4840	849089	5804	5573
100000	63210	53626	83243948	62203	61587
Fifth Run(Time in Microseconds)					
N	Shell	Heap	Bubble	Merge	Quick
10	422	768	263	568	383
100	432	799	436	693	493
1000	767	1009	9149	1097	792
10000	4865	6926	838649	5826	5464
100000	63169	70540	83159449	62313	59937
Average Run(Time in Microseconds)					
N	Shell	Heap	Bubble	Merge	Quick
10	423.4	799.2	272.4	576.6	394.8
100	430.6	725	380.6	616	422.2
1000	722.4	1096.2	9016	1071	795
10000	5348.6	5257.2	802827.4	5860.6	5437
100000	58242.4	56307.8	83116821	62507	59535.8

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

First Chart (fig.1) compares all sorting algorithms for the small values of N=10. Second Chart (fig.2) compares all sorting algorithms for the values of N=100. Third Chart (fig.3) compares all sorting algorithms for the large values of N=1000. Chart (fig.4) compares all sorting algorithms for the large values of N=10000 and Chart (fig.5) compares all sorting algorithms for the large values of N=100000. First Chart (fig.1) compares all the sorting algorithms for the small values of N=10. For N=10, Bubble sort taking minimum execution time.

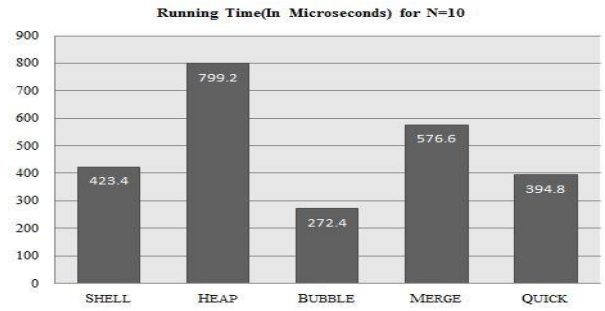

Fig. 1 running time of sorting Algorithms in microsecond for N=10

Second Chart (fig.2) compares all the sorting algorithms for the medium values of N=100. For N=100, Bubble sort taking minimum execution time.

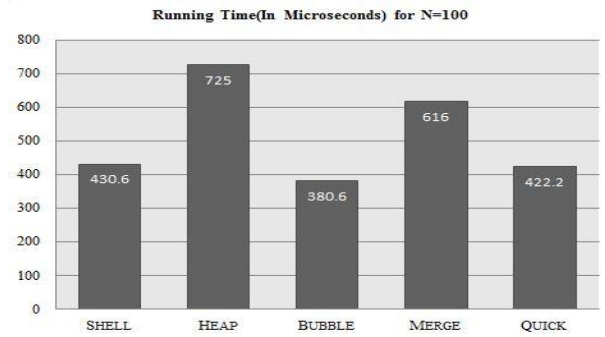

Fig. 2 running time of sorting Algorithms in microsecond for N=100

Third Chart (fig.3) compares all the sorting algorithms for the large values of N=1000. For N=1000, Shell sort taking minimum execution time.

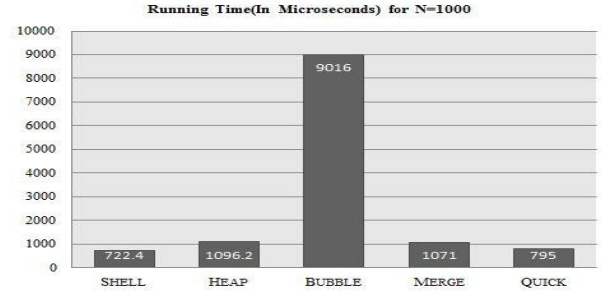

Fig. 3 running time of sorting Algorithms in microsecond for N=1000

Forth Chart (fig.4) compares all the sorting algorithms for the large values of N=10000. For N=10000, Heap sort taking minimum execution time.

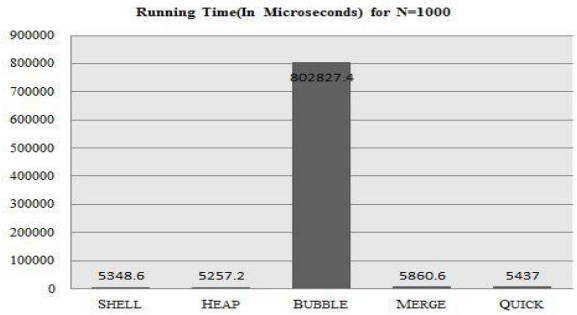

Fig. 4 running time of sorting Algorithms in microsecond for N=10000

Fifth Chart (fig.5) compares all the sorting algorithms for the large values of N=100000. For N=100000, Heap sort taking minimum execution time.

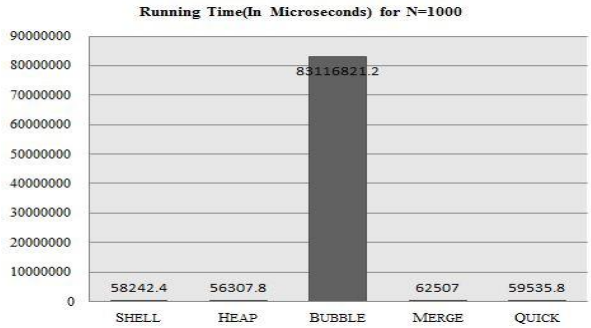

Fig. 5 running time of sorting Algorithms in microsecond for N=100000

For N=1000, 10000 and 100000, again Bubble Sort is taking the Maximum Time as shown in above figures. We can observe from the figures that Shell Sort and Quick Sort are taking the least time in all the cases but space requirement for a shell is less then Quick Short. So we can say that from all the sorting algorithms we taken for performance analysis, Shell Sort is most efficient.

V. CONCLUSIONS

In this study, we have studied about numerous sorting algorithms and their assessment. Every sorting algorithm has various advantage and disadvantage. To determine the running time of each sorting algorithm we used a Program for comparing the running time (in Microseconds). After running the similar program on five different executions (for each different value of N=10, 100, 1000, 10000, 100000), we calculated the average running time for each algorithm and then presented the result with the help of a chart. From the chart, We can conclude that Shell Sort is the most efficient algorithm.

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

REFERENCES

[1]. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms", MIT Press, Cambridge, MA, 2nd edition, 2001.
[2]. Knuth, The art of computer programming sorting and searching, 2nd edition, Addison-wesley, 1998.
[3]. http://www.dotnetperls.com/stopwatch accessed on 15 Jan, 2014.
[4]. http://www.c-sharpcorner.com/uploadfile/9f4ff8/use-of-stopwatch-class- in-c-sharp/ accessed on 15 Jan, 2014.
[5]. http://en.algoritmy.net/category/39386/Sorting-algorithms accessed on 20 Jan, 2014.

APPENDIX

/* Program that will show the use of Sorting Algorithms (Heap Sort, Shell 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;
using System.Threading.Tasks;
namespace SortAlgorithms
{
class SortAlgoComparison
{
static void Main(string[] args)
{
int N = 10000; // Set the value here if you want to run the code for 10,100,1000 or 10000 elements
int[] arr_shell = new int[N];
int[] arr_heap = new int[N];
int[] arr_bubble = new int[N];
int[] arr_merge = new int[N];
int[] arr_quick = new int[N];
int[] aux = new int[N];
Random rn = new Random();

for (int i = 0; i < N; i++)
{
arr_shell[i] = rn.Next(1, 10000);
arr_heap[i] = rn.Next(1, 10000);
arr_bubble[i] = rn.Next(1, 10000);
arr_merge[i] = rn.Next(1, 10000);
arr_quick[i] = rn.Next(1, 10000);
}

/***********SHELL SORT CALL**************/
System.Diagnostics.Stopwatch SW1 = new System.Diagnostics.Stopwatch();
SW1.Start();
ShellSort(arr_shell);
SW1.Stop();
long timeselection = SW1.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
Console.WriteLine("time taken by shell sort is:{0} microseconds", timeselection);

/************ HEAP SORT CALL **************/
System.Diagnostics.Stopwatch SW2 = new System.Diagnostics.Stopwatch();
SW2.Start();
Heapsort(arr_heap);
SW2.Stop();
long timeinsertion = SW2.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
Console.WriteLine("time taken by heap sort is:{0} microseconds", timeinsertion);

/********** BUBBLE SORT CALL *************/
System.Diagnostics.Stopwatch SW3 = new System.Diagnostics.Stopwatch();
SW3.Start();
BubbleSort(arr_bubble);
SW3.Stop();
long timebubble = SW3.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
Console.WriteLine("time taken by bubble sort is:{0} microseconds", timebubble);

/********** MEAGE SORT CALL ***********/
System.Diagnostics.Stopwatch SW4 = new System.Diagnostics.Stopwatch();
SW4.Start();
MergeSort(arr_merge, aux, 0, arr_merge.Length - 1);
SW4.Stop(); long timemerge = SW4.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
Console.WriteLine("time taken by merge sort is:{0} microseconds", timemerge);

/*********** QUICK SORT CALL **************/
System.Diagnostics.Stopwatch SW5 = new System.Diagnostics.Stopwatch();
SW5.Start();
QuickSort(arr_quick, 0, arr_quick.Length - 1);
SW5.Stop();
long timequick = SW5.ElapsedTicks / (System.Diagnostics.Stopwatch.Frequency / (1000L * 1000L));
Console.WriteLine("time taken by quick sort is:{0} microseconds", timequick);
```

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

```
Console.ReadKey();
}

/*** SHELL SORT DEFINITION START ***/
public static int[] ShellSort(int[] array)
{
int gap = array.Length / 2;
while (gap > 0)
{
for (int i = 0; i < array.Length - gap; i++)
//modified insertion sort
{
int j = i + gap;
int tmp = array[j];
while (j >= gap && tmp > array[j - gap])
{
array[j] = array[j - gap];
j -= gap;
}
array[j] = tmp;
}
if (gap == 2) //change the gap size
{
gap = 1;
}
else
{
gap = (int)(gap / 2.2);
}
}
return array;
}
/****** SHELL SORT DEFINITION END *****/


/******HEAP SORT DEFINITION START ******/
public static void Heapsort(int[] array)
{
for (int i = array.Length / 2 - 1; i >= 0; i--)
{
RepairTop(array, array.Length - 1, i);
}
for (int i = array.Length - 1; i > 0; i--)
{
HeapSwap(array, 0, i);
RepairTop(array, i - 1, 0);
}
}
/* MOVE THE TOP OF THE HEAP TO THE
CORRECT PLACE */
private static void RepairTop(int[] array, int
bottom, int topIndex)
{
int tmp = array[topIndex];
int next = topIndex * 2 + 1;
if (next < bottom && array[next] > array[next +
1]) next++;

while (next <= bottom && tmp > array[next])
{
array[topIndex] = array[next];
topIndex = next;
next = next * 2 + 1;
if (next < bottom && array[next] > array[next
+ 1]) next++;
}
array[topIndex] = tmp;
}

/*** SWAPS TWO ELEMENTS OF THE HEAP
***/
private static void HeapSwap(int[] array, int left,
int right)
{
int tmp = array[right];
array[right] = array[left];
array[left] = tmp;
}
/******* HEAP SORT DEFINITION END
*******/


/***** BUBBLE SORT DEFINITION START
******/
static void BubbleSort(int[] arr)
{
for (int i = 0; i < arr.Length - 1; i++)
{
for (int j = 0; j < arr.Length - i - 1; j++)
{
if (arr[j + 1] < arr[j])
{
int tmp = arr[j + 1];
arr[j + 1] = arr[j];
arr[j] = tmp;
}
}
}
}
/******* BUBBLE SORT DEFINITION END
*******/


/******MERGE SORT DEFINITION START
*******/
public static void MergeSort(int[] array, int[] aux, int
left, int right)
{
if (left == right) return;
int middleIndex = (left + right) / 2;
MergeSort(array, aux, left, middleIndex);
```

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering

ICACTRP 2017

International Conference on Advances in Computational Techniques and Research Practices

Noida Institute of Engineering & Technology, Greater Noida

Vol. 6, Special Issue 2, February 2017

```
MergeSort(array, aux, middleIndex + 1, right);
Merge(array, aux, left, right);

for (int i = left; i <= right; i++)
{
array[i] = aux[i];
}
}

private static void Merge(int[] array, int[] aux, int
left, int right)
{
int middleIndex = (left + right) / 2;
int leftIndex = left;
int rightIndex = middleIndex + 1;
int auxIndex = left;
while (leftIndex <= middleIndex && rightIndex
<= right)
{
if (array[leftIndex] >= array[rightIndex])
{
aux[auxIndex] = array[leftIndex++];
}
else
{
aux[auxIndex] = array[rightIndex++];
}
auxIndex++;
}
while (leftIndex <= middleIndex)
{
aux[auxIndex] = array[leftIndex++];
auxIndex++;
}
while (rightIndex <= right)
{
aux[auxIndex] = array[rightIndex++];
auxIndex++;
}
}
/******* MERGE SORT DEFINITION END
*******/


/****** QUICK SORT DEFINITION START
*******/
public static void QuickSort(int[] array, int left, int
right)
{
if (left < right)
{
int limit = left;
for (int i = left + 1; i < right; i++)
{
if (array[i] > array[left])
{
QuickSwap(array, i, ++limit);
}
}
QuickSwap(array, left, limit);
QuickSort(array, left, limit);
QuickSort(array, limit + 1, right);
}
}

private static void QuickSwap(int[] array, int left, int
right)
{
int tmp = array[right];
array[right] = array[left];
array[left] = tmp;
}
/***** QUCIK SORT DEFINITION END *****/

}
}
```

BIOGRAPHY

Mr. Deepak Kumar Pathak is working as an Assistant Professor in Future Group of Institutes, Bareilly, U.P. His academic qualification is M.Tech.(C.S.E.), MCA. He has more than three year experience in teaching field. His research areas are Operating System, Data Mining.

Mr. Vivek Kumar Sharma is working as an Assistant Professor in Department of Computer Science & Engineering, N.I.E.T. Greater Noida. His academic qualification is M. Tech. (C.S.E.), B.Tech. (C.S.E.). He has five year experience in teaching field. His research areas are Parallel Computing, Software Engineering.

Ms. Mansi Rastogi is working as an Assistant Professor in Future Group of Institutions Bareilly, U.P. His academic qualification is MCA. She has two year experience in teaching field. His research areas are Data Base, Software Engineering.