Different Sorting Algorithms comparison based upon the Time Complexity

Yash Chauhan[#1], Anuj Duggal[*2]

[#]*Parul Institute of Engineering and Technology, Parul University*
Po.Limda, Vadodara

[*] *Intel Technology India Pvt. Ltd.*
Deverabeesanahalli, Bangalore.

Abstract : Sorting is a huge demand research area in computer science and one of the most basic research fields in computer science. The sorting algorithms problem has attracted a great deal of study in computer science. The main aim of using sorting algorithms is to make the record easier to search, insert, and delete. We're analysing a total of five sorting algorithms: bubble sort, selecting sort, insertion sort, merge sort and quick sort, the time and space complexity were summarized. Moreover from the aspects of the input sequence, some results were obtained based on the experiments. So we analysed that when the size of data is small, insertion sort or selection sort performs well and when the sequence is in the ordered form, insertion sort or bubble sort performs well. In this paper, we present a general result of the analysis of sorting algorithms and their properties. In this paper a comparison is made for different sorting algorithms.

***IndexTerms* – Algorithm, Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Time Complexity, Stability.**

I. Introduction

Sorting is an algorithm that arranges the elements either in an ascending or descending order. Sorting Algorithms can decrease the complexity of a problem.

Mainly there are two types of Sorting,

- Internal Sorting
- External Sorting.

Sorting algorithms can be classified based on the following parameters:

A. Stability

A sorting algorithm is stable [1] if it preserves the form of duplicate keys, or stability means those similar elements retain their relativistic positions, after sorting.

B. Adaptivity

A sorting algorithm is adaptive [2] if it sorts the sequences that are close to sorted faster than random sequences. With a few sorting algorithms, the complexity changes based on pre-sortedness [Quick sort]: pre-sortedness of the input affects the running time.

C. Time complexity

The time complexity of an algorithm signifies the total time required by the program to run to completion. Algorithms have different cases of complexity which are the best case, the average case, and the worst case. The time complexity of an algorithm is represented using the asymptotic notations [3]. Asymptotic notations provide the lower bound and upper bound of an algorithm.

D. Space complexity

The space complexity of any algorithm is also important, and it is the number of memory cells which an algorithm needs. Space complexity calculated by both auxiliary space and space used by the input [4].

II. Five Sorting Algorithms

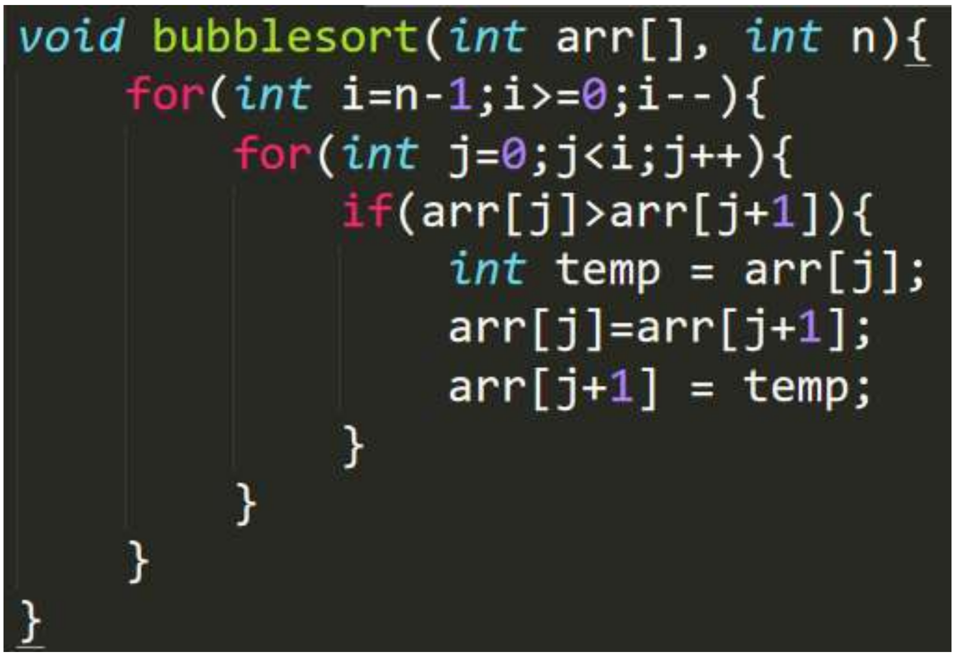
In this paper we have discussed five sorting algorithms with their complexity and stability.

1. Bubble sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

A. Bubble Sort

The sorted array as input or almost all elements are in the proper place, bubble sort has O(n) as the best case performance and O(n*n) as the worst-case performance. Bubble sort has to perform a large number comparison when there are more elements in the list and it increases as the number of items increases that need to be sorted. Although bubble sort is quite easy to implement and it's inefficient while comparing to all other sorting algorithms. It is the simplest sorting algorithm that works by frequently swaps the neighboring elements if they are in the wrong order.

Bubble sort algorithm [5] used in the experiments below was described by C++ language as shown in fig 1:



```
void bubblesort(int arr[], int n){
    for(int i=n-1;i>=0;i--){
        for(int j=0;j<i;j++){
            if(arr[j]>arr[j+1]){
                int temp = arr[j];
                arr[j]=arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Fig. 1 Bubble Sort Code

The above code shown in the Fig 1 takes O(n*n) even in the best case. This code can be optimized by introducing an extra variable swapped.

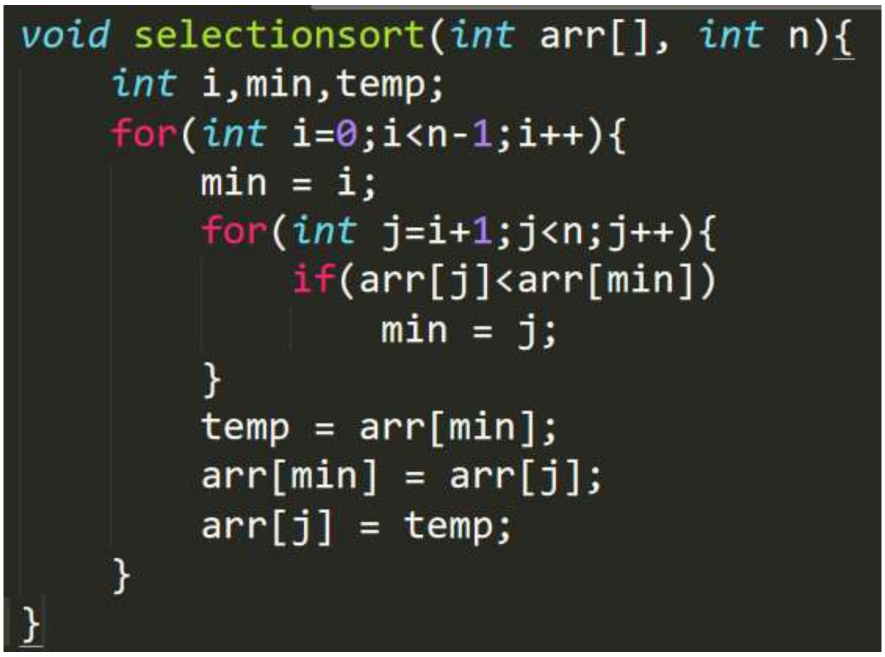
Example: First Pass:
(55 11 44 22 88) –> (11 55 44 22 88),
The first two elements compared by algorithm,
until swaps 55 > 11.
(11 55 44 22 88) –> (11 44 55 22 88),
Swaps until 55 > 44
(11 44 55 22 88) –> (11 44 22 55 88),
Swaps until 55 > 22
(11 44 22 55 88) –> (11 44 22 55 88),

Second Pass:
(11 44 22 55 88) –> (11 44 22 55 88),
(11 44 22 55 88) –> (11 22 44 55 88),
Swaps until 44 > 22
(11 22 44 55 88) –> (11 22 44 55 88)
Now, the array is sorted but the algorithm wants one whole pass without any swap to verify.

Third Pass:
(11 22 44 55 88) –> (11 22 44 55 88)
(11 22 44 55 88) –> (11 22 44 55 88)
(11 22 44 55 88) –> (11 22 44 55 88)
(11 22 44 55 88) –> (11 22 44 55 88)

B. Selection Sort

Selection sort is the most simplistic algorithm but it’s inefficient. In selection sort, we have to find the smallest element doing the linear scan and move it to the front (swapping it with the front element). Then, we have to find the second smallest element and move it from its place, again doing a linear scan. Continue doing this until all the elements are at their place. In the selection sort algorithm, it maintains two sub-arrays in a given array. The first sub-array which is already sorted and the second remaining sub-array which is unsorted. Every Iteration of selection sort, the slightest element from the unsorted secondary the array is picked and moved to the sorted secondary array. Selection sort algorithm [6] used in the experiments below was described by C++ language as shown in fig 2:



```
void selectionsort(int arr[], int n){
    int i,min,temp;
    for(int i=0;i<n-1;i++){
        min = i;
        for(int j=i+1;j<n;j++){
            if(arr[j]<arr[min])
                min = j;
        }
        temp = arr[min];
        arr[min] = arr[j];
        arr[j] = temp;
    }
}
```

Fig. 2 Selection Sort Code

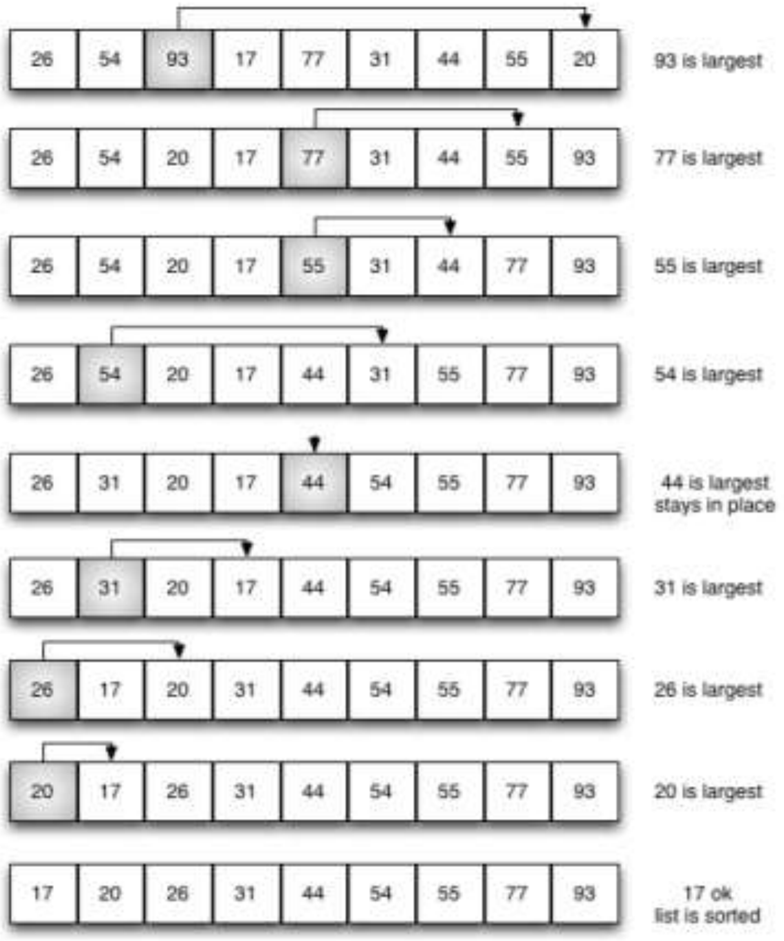
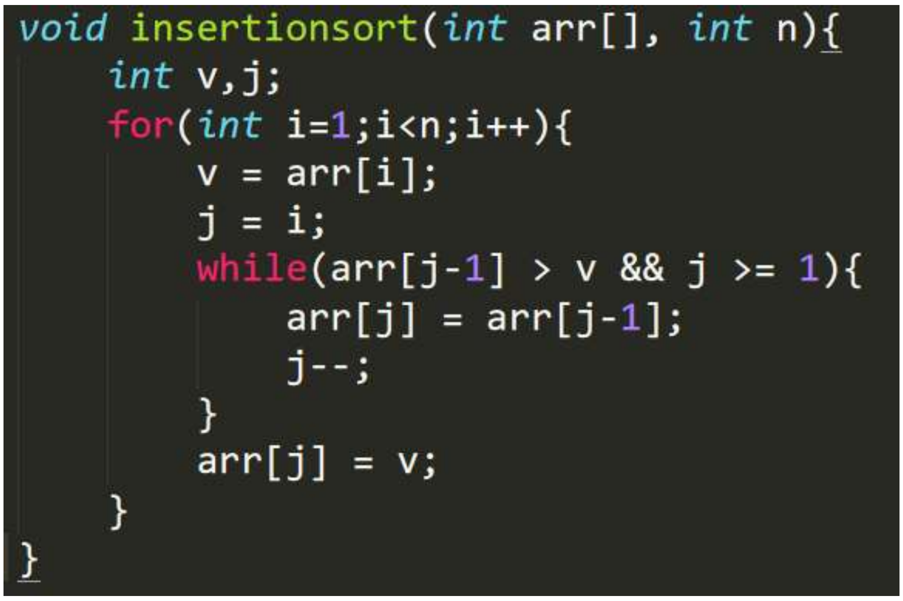


Fig. 2 Example of Selection Sort [10]

C. *Insertion Sort*

Insertion sort algorithm picks components individually and places it to the right position any place it has a place in the sorted lists of elements. Insertion sort is a simple yet efficient comparison sort for small data. Insertion Sort is reduced to its total number of steps if given a partially sorted list, increased efficiency. It's practically more efficient than selection sort and a bubble sort, even though all of them have O(n*n) worst case complexity. Numbers are divided into sorted and unsorted. The unsorted values are put into their suitable positions in the sorted secondary array one by one. Insertion sort is well-organized for minor data sets but very incompetent for larger lists. It reduces its total number of steps if given a partially sorted list, increased efficiency.

Insertion sort algorithm [7] used in the experiments below was described by C++ language as shown in fig 3:



```
void insertionsort(int arr[], int n){
    int v,j;
    for(int i=1;i<n;i++){
        v = arr[i];
        j = i;
        while(arr[j-1] > v && j >= 1){
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = v;
    }
}
```

Fig. 3 Insertion Sort Code

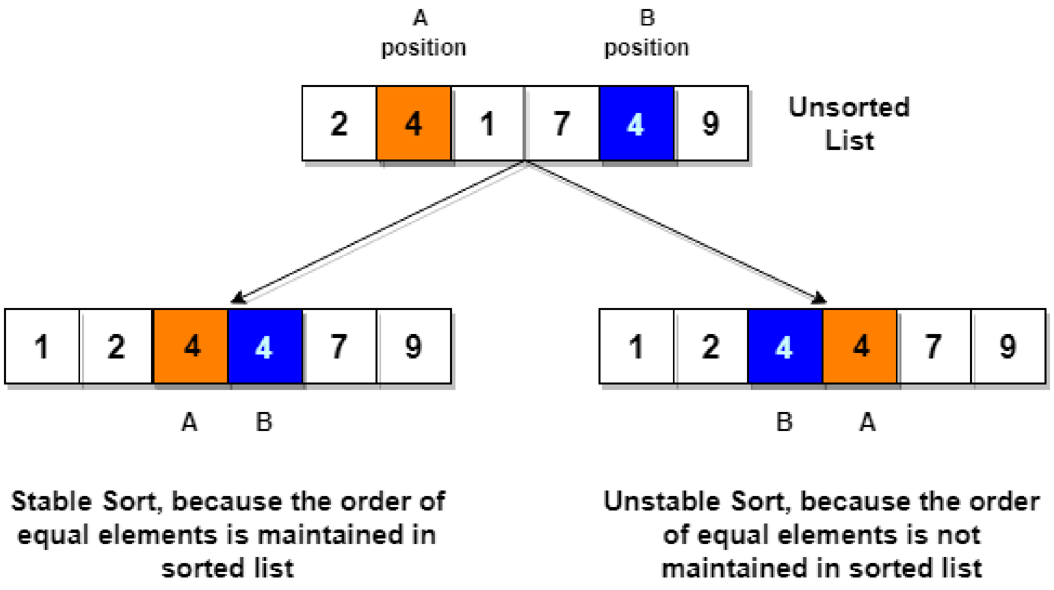

A position
B position
2 4 1 7 4 9
Unsorted List
1 2 4 4 7 9
A B
1 2 4 4 7 9
B A
Stable Sort, because the order of equal elements is maintained in sorted list
Unstable Sort, because the order of equal elements is not maintained in sorted list

Fig. 3.1 Example of Insertion Sort [11]

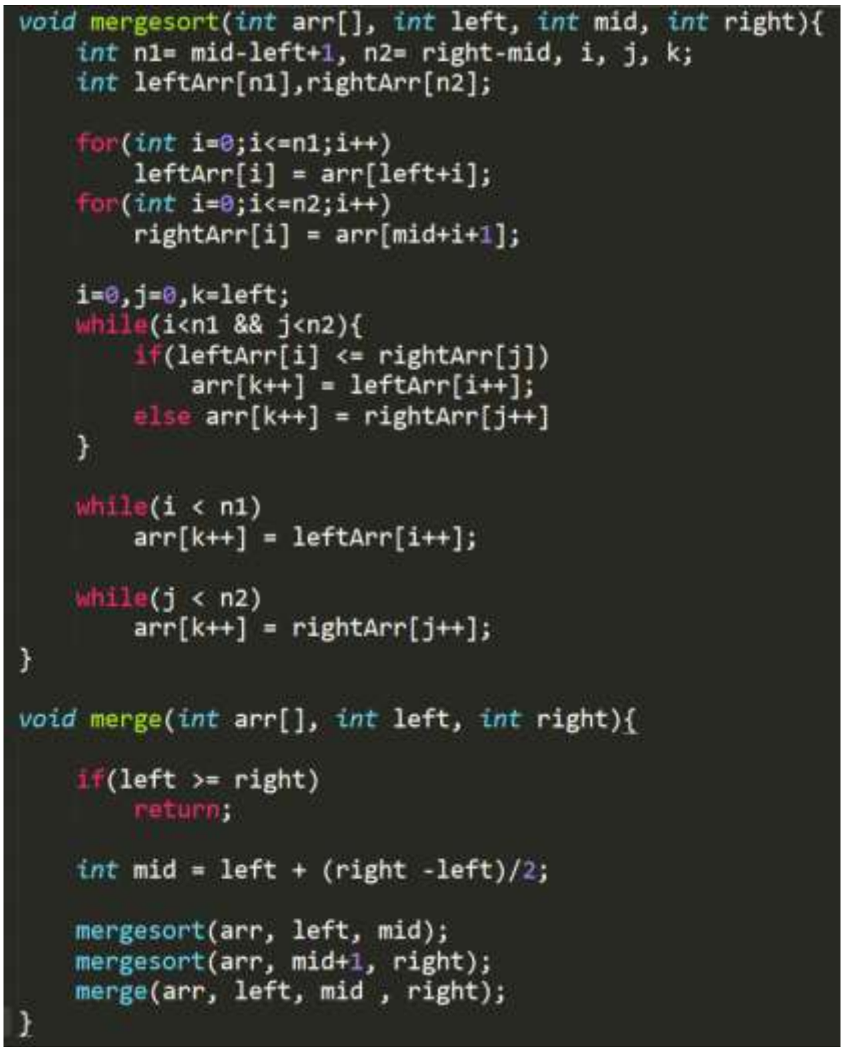
D. Merge Sort

Merge Sort is worked with the basis of Divide and Conquer algorithm. It splits the input array into two bisects, calls itself for the two bisects and merges two sorted bisects.

1. Divide Step: If a given array Arr has zero or one element, simply return; if previously sorted. Otherwise, split Arr [x ... z] into two secondary arrays Arr [x ... z] and A [y + 1 ... z], each containing about parts of the elements of Arr [x ... z]. That is, y is the intermediate point of Arr [x ... z].

2. Conquer Step: Now Sorting the two small arrays Arr [x ... z] and Arr [y + 1 ... z] recursively by using conquer

3. Combine Step: After all, we have to Merge the back of the element in Arr [x ... z] by merging the two sorted small arrays Arr [x ... z] and Arr [y + 1 ... z] into a sorted series. To achieve this pace, will identify a procedure MERGE (Arr, x, y, z). Merge sort algorithm [8][9] used in the experiments below was described by C++ language as shown in fig 4:



```
void mergesort(int arr[], int left, int mid, int right){
    int n1= mid-left+1, n2= right-mid, i, j, k;
    int leftArr[n1],rightArr[n2];

    for(int i=0;i<=n1;i++)
        leftArr[i] = arr[left+i];
    for(int i=0;i<=n2;i++)
        rightArr[i] = arr[mid+i+1];

    i=0,j=0,k=left;
    while(i<n1 && j<n2){
        if(leftArr[i] <= rightArr[j])
            arr[k++] = leftArr[i++];
        else arr[k++] = rightArr[j++]
    }

    while(i < n1)
        arr[k++] = leftArr[i++];

    while(j < n2)
        arr[k++] = rightArr[j++];
}

void merge(int arr[], int left, int right){

    if(left >= right)
        return;

    int mid = left + (right -left)/2;

    mergesort(arr, left, mid);
    mergesort(arr, mid+1, right);
    merge(arr, left, mid , right);
}
```

Fig. 4 Merge Sort Code

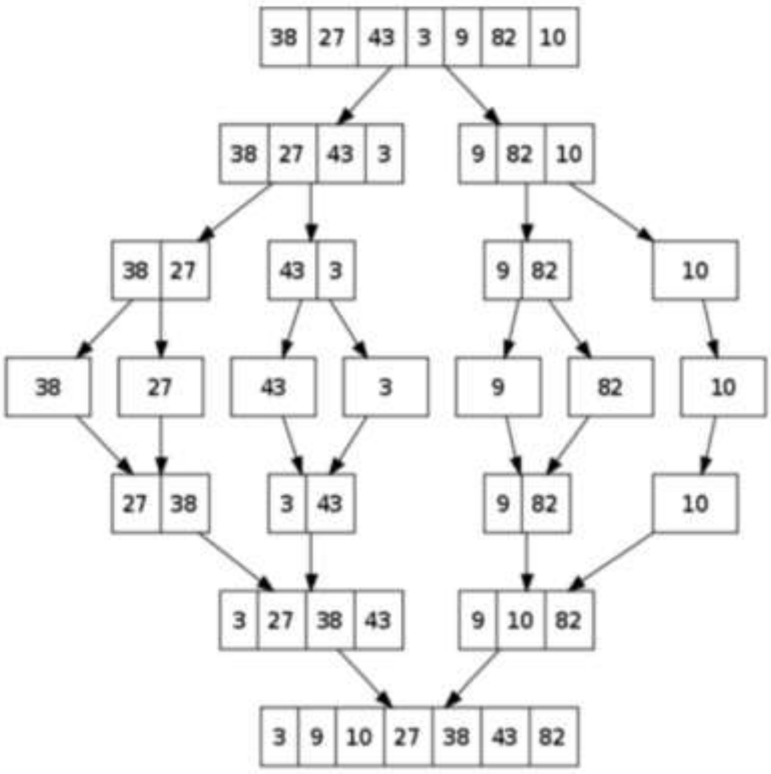


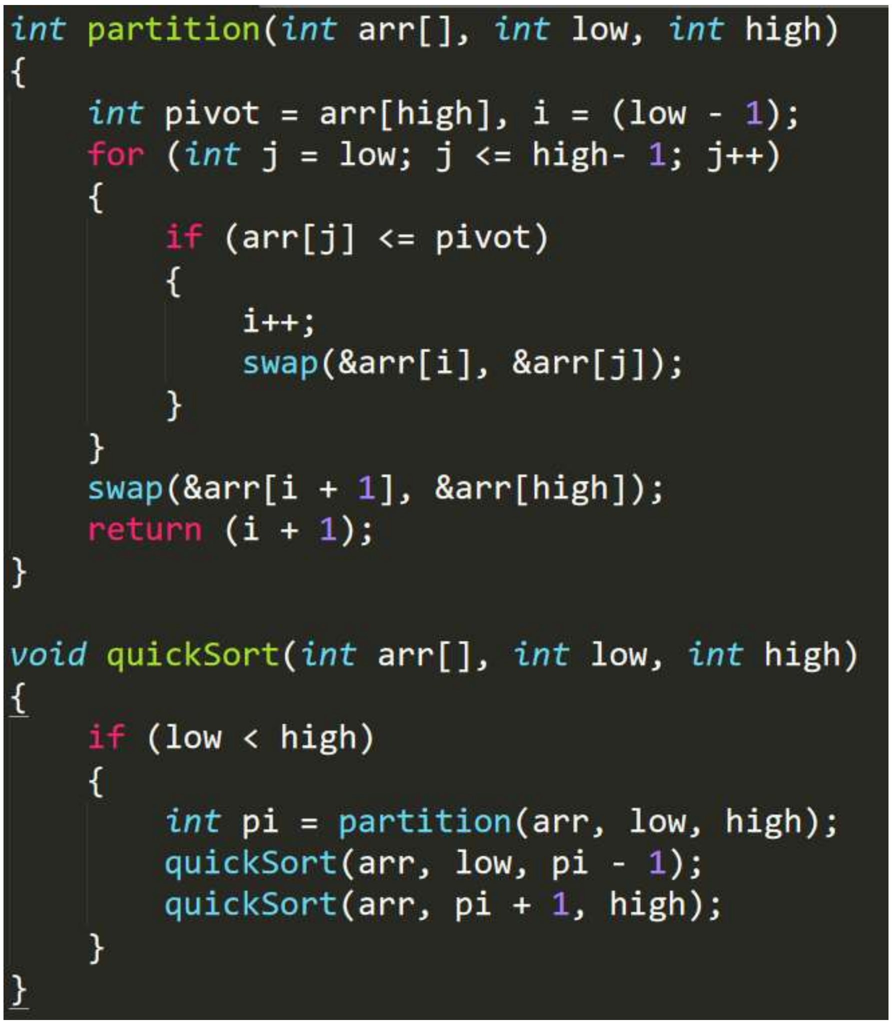
Fig. 4.1 Example of Merge Sort [12]

E. Quick Sort

Quick sort is one of the fastest sorting algorithms which is the part of many sorting libraries. The running time of Quick Sort depends heavily on choosing the pivot element. Quick sort also belongs to the divide and conquer category of algorithms. It depends on the operation of the partition. To partition an array of an element called a pivot is selected. The Quick sort algorithm works as follows:

- Arr [j] = x is the pivot value.
- Arr [j…p - 1] contains elements less than x.
- Arr [p + 1…r - 1] contains the elements which are larger than or equivalent to x.
- Arr[r...k] contains elements which are currently unexplored.

Since the selection of pivot elements is random, therefore average case and best case running time is O(n log n). Moreover, the worst case time complexity is O(n*n). Quick sort is not a stable sorting algorithm. Quick sort algorithm [13] used in the experiments below was described by C++ language as shown in fig 5:



```
int partition(int arr[], int low, int high)
{
    int pivot = arr[high], i = (low - 1);
    for (int j = low; j <= high- 1; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Fig. 5 Quick Sort Code

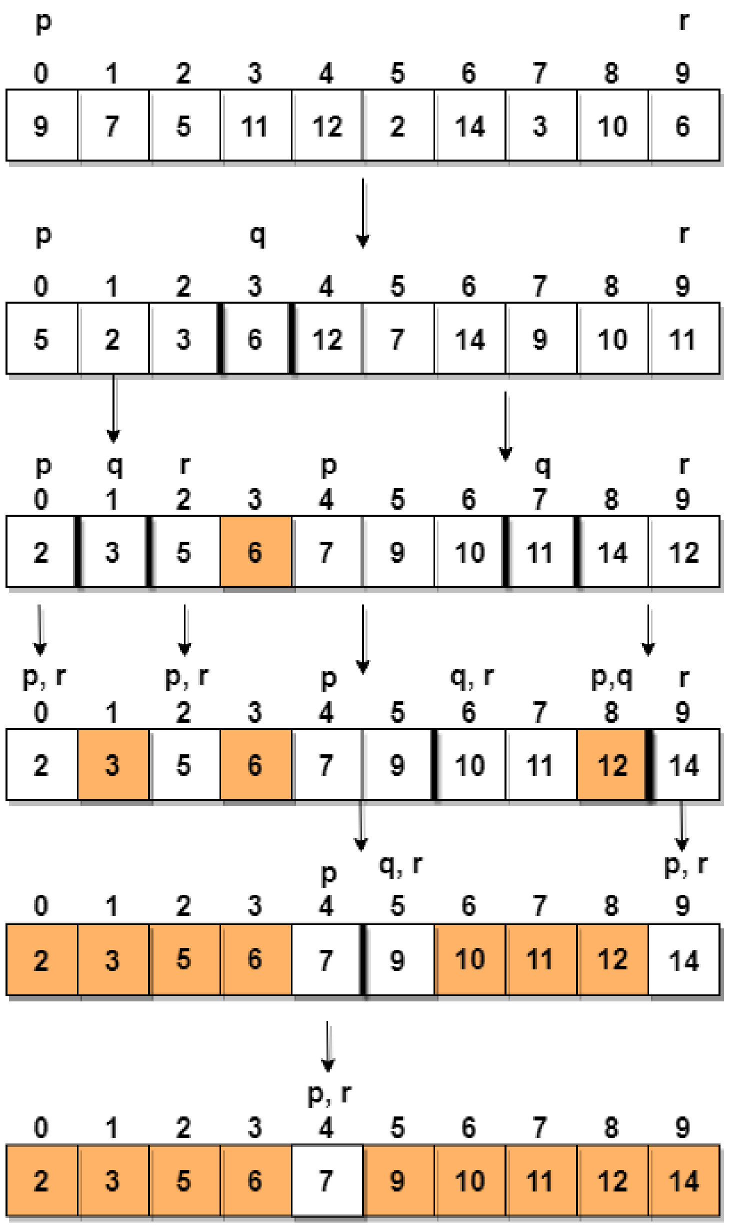


Fig. 5.1 Example of Quick Sort [14]

III. COMPARISON ON THE BASIS OF COMPLEXITY AND OTHER FACTOR

In this paper, there are two classes of Sorting Algorithms:

1. *O(n*n):*
   a. Bubble Sort
   b. Selection Sort
   c. Insertion Sort
2. *O(n log n)*
   a. Merge Sort
   b. Quick Sort

In best-case conditions (if the list is already in sorted order), the bubble sort can approach a constant O(n) level of complexity, while the insertion sort and selection sorts also have complexities; they are significantly more efficient than bubble sort. The quick sort is a massively recursive sort. It can be said as merge sort is the faster version of quick sort.

This paper describes five well known sorting algorithms and their running time and stability which is given in the below Table 1. This table gives the comparison of time complexity or running time of different sorting algorithms and the stability in the precise manner.

TABLE I
COMPARISONS ON BASIS OF COMPLEXITY AND OTHER FACTORS

No	Sorting Algorithm	Best Case	Average Case	Worst Case	Stable
1	Bubble Sort	O(n)	O(n*n)	O(n*n)	Yes
2	Selection Sort	O(n*n)	O(n*n)	O(n*n)	No
3	Insertion Sort	O(n)	O(n*n)	O(n*n)	Yes
4	Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	Yes
5	Quick Sort	O(nlogn)	O(nlogn)	O(n*n)	No

IV. CONCLUSIONS

This paper discusses well-known sorting algorithms, their code and running time. In the previous work section, people have done a comparative study of sorting algorithms. Nowadays, some of them compared the running time of algorithms on real computers on a different number of inputs which is not much use because the diversity of computing devices is very high. This paper compares the running time of their algorithms as a mathematical entity and tries to analyse as an abstract point of view. This paper describes five well-known sorting algorithms and their running time complexity and their stability. To determine the good sorting algorithm, the time complexity is the main consideration but other factors include handling various data types, consistency of performance, the complexity of code and the stability. From the above discussion, we can conclude every sorting algorithm has some advantages and disadvantages of their usage and the programmer must choose according to his or her requirement of sorting algorithms.

V. REFERENCES

[1] Elisseeff, Andre, Theodoros Evgeniou, and Massimiliano Pontil. "Stability of randomized learning algorithms." Journal of Machine Learning Research. 2005.
[2] Estivill-Castro, Vladmir, and Derick Wood. “A survey of adaptive sorting algorithms." ACM Computing Surveys (CSUR) 24.4, 1992: 441-476.
[3] Cormen, Thomas H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to algorithms. MIT press, 2009.
[4] T. H. Cormen, C. E. Lieserson, R. L. Rivest and S. Clifford, "Introduction to Algorithms", 3rd ed., the MIT Press Cambridge, Massachusetts London, England 2009.
[5] Debosiewicz W. An Efficient Variation of Bubble Sort. Information Processing Letters. 1980, 11(1): 5-6.
[6] Iraj H., Afsari M. H. S., Hassanzadeh S. A New External Sorting Algorithm with Selecting the Record List Location. USEAS Transactions on Communications. 2006, 5(5):909-913.
[7] Kumari A., Chakraborty S. Software Complexity: A Statistical Case Study through Insertion Sort. Applied Mathematics and Computation. 2007, 190(1): 40-50.
[8] http://en.wikipedia.org/wiki/Merge_sort
[9] Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field", Soviet Mathematics - Doklady (10), pp.744.
[10] https://images.app.goo.gl/t9vU5mqdwsb9UuDJ6
[11] https://images.app.goo.gl/w9iMuC5u5FtarsJV6
[12] https://images.app.goo.gl/rwycUTeSUwyHye9k7
[13] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", 1976, pp.66
[14] https://images.app.goo.gl/VesQXHy5hADJptCV7