Volume 5, Issue 5, May 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

An Analytical Comparison of Different Sorting Algorithms in Data Structure

Miraj Gul, Noorul Amin, M. Suliman
Deptt of IT, Hazara University
Mansehra, Pakistan

Abstract— Sorting is considered as a very basic operation in computer science. Sorting is used as an intermediate step in many operations. Sorting refers to the process of arranging list of elements in a particular order either ascending or descending using a key value. There are a lot of sorting algorithms have been developed so far. This research paper presents the different types of sorting algorithms of data structure like Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Heap Sort and Quick Sort and also gives their performance analysis with respect to time complexity. These six algorithms are important and have been an area of focus for a long time but still the question remains the same of "which to use when?" which is the main reason to perform this research. Each algorithm solves the sorting problem in a different way. This research provides a detailed study of how all the six algorithms work and then compares them on the basis of various parameters apart from time complexity to reach our conclusion.

Keywords— Algorithm, Sorting, Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Complexity

I. NTRODUCTION

Algorithm [12] is a clearly written step-by-step procedure for solving a problem, which is guaranteed to terminate after a finite number of steps. For a given problem, there are generally many different algorithms [7] for solving it. Some algorithms are more efficient than others [15] i.e. less time or memory is required to execute them. The analysis of algorithms studies time and memory requirements of algorithms and the inter-relationship among those requirements and the number of items being processed. Sorting is generally understood to be the process of rearranging a given set of data elements in a specific order and therefore, the analysis and design of useful sorting algorithms has remained one of the most important research areas in the field. Despite the fact that, several new sorting algorithms being introduced, the large number of programmers in the field depend on one of the comparison-based sorting algorithms: Bubble sort [2], Insertion sort [3][6], Selection sort[12] etc. Hence sorting is an almost universally performed and hence, considered as a fundamental operation in computer processing [12]. Farther more sorting enhances the efficiency of other operations such as searching, deletion insertion etc. The usefulness and significance of sorting is depicted from the daily life e.g. students data, telephone directories, dictionaries etc.

The sorting algorithms are also classified on the basis of different characteristics of these algorithms.

- **Computational Complexity**

On the basis of computational complexities [5], sorting algorithms are classified as $O(n \log n)$,$O(\log^2 n)$ and $O(n^2)$ where n is the size of data.

- **Comparison based and Non-Comparison based Sorting**

Comparison-based [5] algorithm sorts a data set by comparing the data set values. For example, quick sort, merge sort, heap sort, bubble sort, insertion sort etc.
The non-comparison based algorithms sort data without pair-wise comparison of data elements. Bucket sort, radix sort are examples of non comparison based sorting algorithms.
Comparison based algorithm sorts a data set by doing comparison of the values. Examples of such algorithms are quick sort [12], merge sort [12], heap sort [13], bubble sort [2], insertion sort etc.
A non-comparison based algorithm sorts' data without pair wise comparison of data elements. Bucket sort, radix sort are examples of non comparison based sorting algorithms.

- **Memory Usage**

There are two classes of sorting algorithms on the basis of memory usage [9]:
1) The internal sorting uses primary memory only (i.e. the data set is small). For example, Selection sort, Bubble sort, Insertion sort etc.
2) The external sorting uses primary memory as well as secondary memory (i.e. the data set is large). For example Merge sort, Radix Sort etc.

- **Stability**

On the basis of stability [8], a sorting algorithm may be classified into stable or unstable. Stable sorting algorithms maintain the relative order of record with equal keys while unstable sorting algorithm does not maintain the relative order of record with equal keys.

The choice of which sorting method is suitable for a problem depends on various efficiency considerations [4][11]-[12] [14]for different problem. Three most important of these considerations are:

- The size of data set to be sorted [4].
- The nature of the data set [1].
- Stability-dose the sort preserves the order of keys with equal values [8].
- The time and memory limitations [9].

II. WORKING PROCEDURE OF ALGORITHMS

A. Bubble Sort:

The Bubble Sort is the simplest internal and comparison based sorting technique, in which. It is also known as a sinking sort (because the smallest items "sink" to the bottom of the array). In bubble sort, instead of searching an array as a whole, the bubble sort works by comparing adjacent pairs of data elements in the array. If the data elements are not in the correct ordered, they are swapped so that the larger of the two moves up. This process continues until the largest of the objects, eventually "bubbles" up to the highest position in the array. After this occurs, the search for the next largest data element begins. The swapping continues until the whole array is in the correct order.

1) Algorithm: Here DATA is an array with N elements. This algorithm sorts the element in DATA

```
BUBBLE (DATA, N)
1.  Repeat Steps 2 and 3 for K=1 to N-1
2.    Set PTR = 1 [Initializes the pass pointer PTR]
3.    Repeat while PTR<= N-K [Execute pass]
      a)  If DATA[PTR] > DATA[PTR+1] then
            i.   Interchange DATA[PTR] and DATA[PTR+1]
          [End of If structure]
      b)  Set PTR = PTR+1
      [End of inner loop]
    [End of step 1 outer loop]
4.  Exit
```

2) Example:

Elements	0	1	2	3	4
Data	25	11	16	19	9
1st Pass	25	11	16	19	9
	11	25	16	19	9
	11	16	25	19	9
	11	16	19	25	9
	11	16	19	9	25
2nd Pass	11	16	19	9	25
	11	16	19	9	25
	11	16	19	9	25
	11	16	9	19	25
3rd Pass	11	16	9	19	25
	11	16	9	19	25
	11	9	16	19	25
4th Pass	11	9	16	19	25
Sorted	9	11	16	19	25

Fig 1: Working of Bubble Sort

3) Analysis: Bubble sort is data sensitive. The number of iterations required may be between 1 and (N-1). The base case for bubble sort is when only one iteration is required. The number of comparisons required is (N-1). The worst case arises when the given array is sorted in reverse order.

Best Case = O (n)

Average Case =(On2)

Worst Case= O (n2)

4) Pros and Cons:

Pros:

- Simplicity and ease of implementation.
- No additional temporary storage is required

Cons:

- Very inefficient for large list of element.
- General complexity is O (n^2).

B. Selection Sort.

The selection sort is the easiest method of sorting. The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its proper position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

1) Algorithm: Here DATA is an array with N elements. This algorithm sorts the element in DATA

SELECTION SORT (DATA, N)

```
1.  Repeat for K=1 to N-1.
2.   Set Max = DATA[K] and LOC=K
3.     Repeat For J=1 to N-K
      i.  If (MAX< DATA[J+1]) then:
              a) MAX=DATA[J+1]
              b) LOC= J+1
       [End of If in step i]
     [End of For loop in step 3]
4.  Set TEMP=DATA[J+1]
5.  Set DATA [j+1]=MAX
6.  Set DATA[LOC] = TEMP
       [end of For loop in step 1]
7.  Exit
```

2) Example:

Elements	0	1	2	3	4
Data	25	17	14	16	9
1st Pass	25	17	14	16	9
	25	17	14	16	9
	25	17	14	16	9
	25	17	14	16	9
	9	17	14	16	25
2nd Pass	9	17	14	16	25
	9	17	14	16	25
	9	17	14	16	25
	9	16	14	17	25
3rd Pass	9	16	14	17	25
	9	16	14	17	25
	9	14	16	17	25
4th Pass	9	14	16	17	25
Sorted	9	14	16	17	25

Fig 2: Working of Selection Sort

3) Analysis: Selecting the smallest element requires scanning all n elements, so this takes $n - 1$ comparisons and then Swapping or interchanging it into the first position. Finding the next lowest element requires scanning the remaining (n − 1) elements and so on, for $(n - 1) + (n - 2) + ... + 2 + 1 = n (n - 1) / 2 = O(n^2)$

Best Case = $O(n^2)$

Average Case= $O(n^2)$

Worst Case= $O(n^2)$

4) Pros and Cons:

Pros:

- Selection Sort is simple method.
- No additional temporary storage is required.

Cons:

- Suitable for small list of elements.
- General complexity is O (n^2).

C. Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. In Insertion sort, assume that at some stage of the sorting process the first *i* entries in a table are in sorted order. The basic idea of insertion sort is to insert the *i*+1th entry into its correct place in the first *i* entries. This increases the length of the sorted section to *i*+1. Initially *i* is set to 1, obviously the first number on its own constitutes a sorted list. After *n*-1 iterations of this process the table is sorted.

1) Algorithm: The algorithm for insertion sort having DATA as an array with N elements is as follows

```
INSERTION (DATA, N)
1.      FOR I=2 to N
2.       Set key= DATA[I]
3.      Set J=I-1
4.      While(J>=1 && key<DATA[J])
a)      DATA[J+1]=DATA[J]
b)      Set J=J-1
        [End of While loop in step 4]
5.      DATA[J+1]=key
  [End of FOR loop in step 1]
6.      Exit
```

2) *Example*

Elements	0	1	2	3	4	5
Data	5	1	6	2	4	3
1st Pass	5	1	6	2	4	3
2nd Pass	1	5	6	2	4	3
3rd Pass	1	5	6	2	4	3
4th Pass	1	2	5	6	4	3
5th Pass	1	2	4	5	6	3
Sorted	1	2	3	4	5	6

Fig 3: Working of Insertion Sort

3) Analysis: The implementation of insertion Sort shows that there are (n−1) passes to sort *n* . The iteration starts at position 1 and moves through position (n−1), as these are the elements that need to be inserted back into the sorted sublists. The maximum number of comparisons for an insertion sort is (n−1) .Total numbers of comparisons are:

$(n-1) + (n-2) + ... + 2 + 1 = n(n-1)/2 = O(n^2)$

Best Case = $O(n^2)$

Average Case= $O(n^2)$

Worst Case= $O(n^2)$

4) Pros and Cons:

Pros:

- Insertion sort exhibits a good performance when dealing with a small list.
- The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

Cons:

- Insertion sort is useful only when sorting a list of few elements.
- The insertion sort repeatedly scans the list of elements each time inserting the elements in the unordered sequence into its correct position.

D. Merge Sort

This sorting method is an example of the Divide-And-Conquer paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. Conceptually, a merge sort works as follows

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.
3. *Algorithm:* To sort the entire sequence DATA[1 .. n], make the initial call to the procedure MERGE-SORT*(A*,1,*n).*

Input**:** Array *DATA* and indices *p, q, r* such that $p \le q \le r$ and sub array *DATA* [*p* ... *q*] is sorted and sub array *DATA* [*q* + 1 ... *r*] is sorted. By restrictions on *p, q, r*, neither sub array is empty.
Output**:** The two sub arrays are merged into a single sorted Sub array in *DATA*[*p* .. *r*].

```
MERGE-SORT (DATA, p, r)
1.   If p < r
[Check for base case]
2.   Then q = FLOOR [(p + r)/2] [Divide step]
3.   MERGE (DATA, p, q)
[Conquer step.]
4.   MERGE (DATA, q + 1, r) [Conquer step.]
5.   MERGE(DATA, p,q,r) [Conquer step.]
```

2) Example:

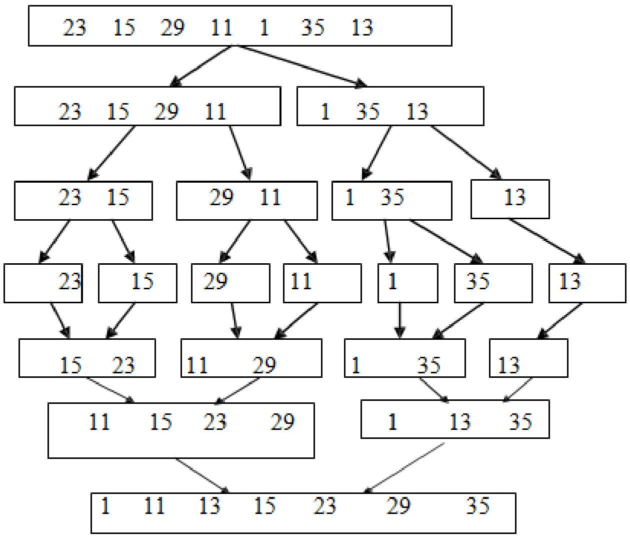


Fig. 4: Working procedure of Merge Sort

3) Analysis: In order to analyze the Merge Sort function, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We divide a list in half log*n* times where *n* is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So the merge operation which results in a list of size *n* requires *n* operations. The result of this analysis is that log*n* splits, each of which costs n for a total of n (log n) operations.
Best Case = O (n logn)
Average Case= O (n logn)
Worst Case= O (n logn)

4) Pros and Cons:
Pros:
- Time Complexity is O(nlogn).
- It can be used for both internal and external sorting

Cons:
- At least twice the memory requirements of the other sorts because it is recursive.
- Space complexity is very high

E. Heap Sort:

Heapsort is an in-place sorting algorithm with worst case and average complexity of O(*n*log*n*).
The basic idea is to turn the array into a binary heap structure, which has the property that it allows efficient retrieval and removal of the maximal element. We repeatedly "remove" the maximal element from the heap, thus building the sorted list from back to front. Heapsort requires random access, so can only be used on an array-like data structure.

1) Algorithm:

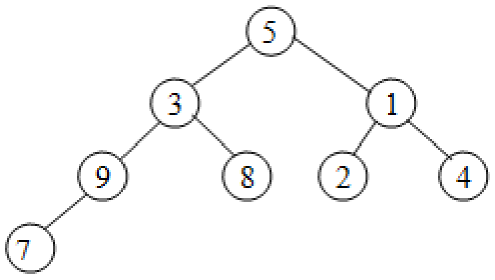
```
heapSort(DATA, N)
  1. heapify(DATA, N)
  2. Set end = N - 1
  3. while end > 0 do
       a) swap(DATA[end], DATA[0])
       b) Set end = end – 1
       c) siftDown(DATA, 0, end)
[End of while in Step 3]
4. Exit
```

```
FUNCTION heapify(DATA,N)
1.  start := (N - 2) / 2
 2. while start ≥ 0 do
      a) siftDown(DATA, start, N-1)
      b) start := start – 1
[End of While in step 2]
```

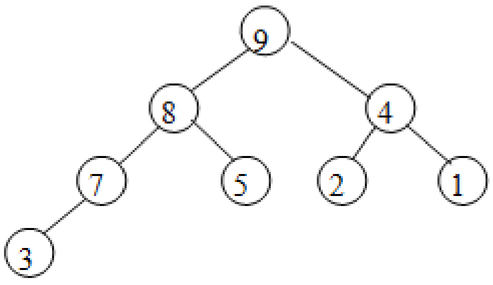
```
FUNCTION siftDown(DATA, start, end)
1.    Set root = start
2.    while root * 2 + 1 ≤ end do
a.    Set child = root * 2 + 1
b.     if child + 1 ≤ end and DATA[child] < DATA[child + 1] then
i.     Set child = child + 1
            [End of if in step b]
c.    if DATA[root] < DATA[child] then
i.    swap(DATA[root], DATA[child])
ii.   Set root = child
             else
i.    Return
          [End of if in step c]
      [End of while in Step 2]
```

2. Example:

5	3	1	9	8	2	4	7



After the shifting the tree will become



Now deleting the root node and adjusting the remaining tree and continuing the process until the whole tree is deleted, we will get the array as

1	2	3	4	5	7	8	9

3. Analysis

Now we turn to the runtime analysis of Heapsort. We begin with Heapify function, which is where all the data comparisons are done. Heapify is recursive, so the number of comparisons it makes on n items is

$H(n)= H(m)+2$ where m is the number of items.

The worst case will occur when that subheap is as large as possible. How many items **m** can a subheap of a heap with n items have? It is easy to see that the largest m can be is 2n/3

We then have the recurrence **$H(n)=H(2n/3)+2$.**

So by Master theorem we have **$H(n)=2\log_{2/3}n=O(\log n)$**

On a closer analysis the constant turns out to be 2, i.e. **$H(n)\sim 2n\log n$.**

Now we turn to the runtime for Buildheap. Since the only comparisons Heapify does are in its n calls to Heapify, then it does at most **O(nlogn)** comparisons. In fact, since most of those calls involve only subheaps with fewer then **n** elements, one can get a tighter upper bound, namely **O(n);** but we will not need this fact. Finally, the data comparisons done by Heapsort consist of those done by its one call to Buildheap (with **O(nlogn)** comparisons), together with those done by Heapsort's own **n-1**calls to Heapify (each of which makes **O(logn)** comparisons), for a total of **O(nlogn+(n-1)logn)=O(nlogn).** So

Worst Case Time Complexity : O(n log n)

Best Case Time Complexity : O(n log n)

Average Time Complexity : O(n log n)

Space Complexity : O(n)

4) Pros and Cons:

Pros:

- Time Complexity is O(nlogn).
- Very fast and widly used.

Cons:

- It is not stable.
- Space complexity is very high due to recursive nature.

F. Quick Sort:

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

- Pick an element, called a pivot, from the list.
- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

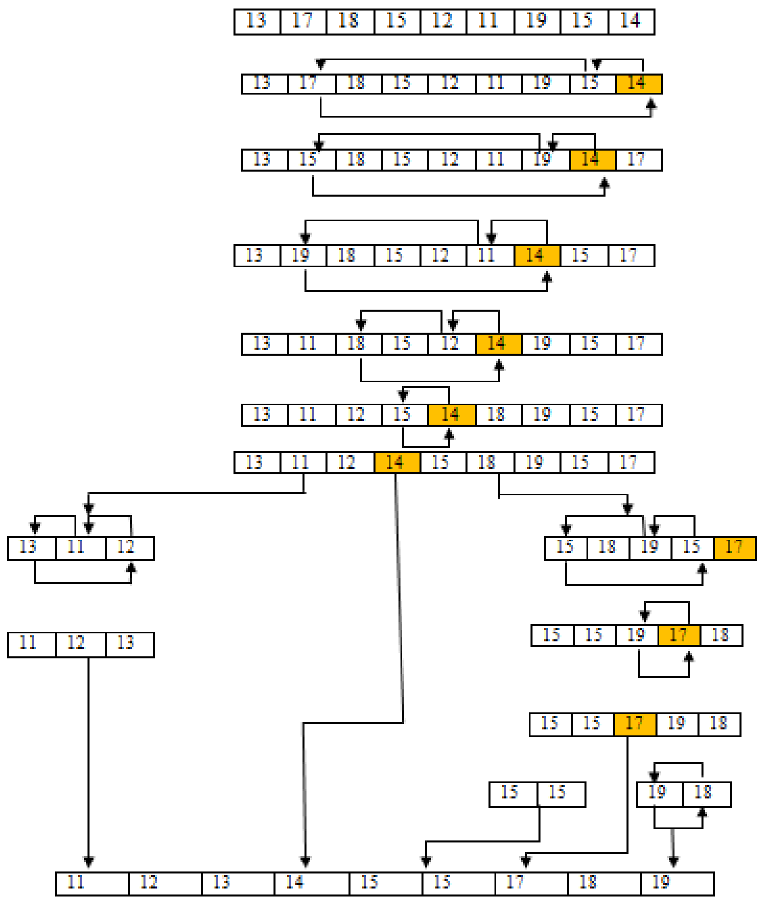
1) Algorithm:

```
1.    quicksort(DATA, start, end):
2.    if (start < end)
a.    Set  p = partition(DATA, start, end)
b.    quicksort(DATA, start, p - 1)
c.    quicksort(DATA, p + 1, end)
       [End of if in step 2]
3.    Exit
```

Function partition(DATA, start, end)

```
1.    Set pivotIndex = choosePivot(DATA, start, end)
2.    Set pivotValue = DATA[pivotIndex]
3.    Set swap DATA[pivotIndex] and DATA[end]
4.    Set storeIndex = start
5.    for  i = start to end−1
a)    if DATA[i] <= pivotValue
i.    swap DATA[i] and DATA[storeIndex]
ii.   storeIndex = storeIndex + 1
              [End of if in step a]
          [End of for in step 5]
6.    swap DATA[storeIndex] and DATA[end]
7.    return storeIndex
```

1. *Example*



2. *Analysis:* To analyze the Quick Sort algorithm, note that for a list of length *n*, if the partition always occurs in the middle of the list, there will again be **log*n*** divisions. In order to find the split point, each of the *n* items needs to be checked against the pivot value. The result is ***n*log*n***. In the worst case, the split points may not be in the middle and can be much skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of *n* items divides into sorting a list of 0 items and a list of ***n*−1** items. Then sorting a list of **n−1**, divides into a list of size 0 and list of size **n−2**, and so on. The result is an ***O*(*n*2)** sort with all of the overhead that recursion requires.

Best Case = **O(n log n)**

Average Case=**O (n log n)**

Worst Case= **O (n^2)**

3. *Pros and Cons:*

Pros:

- One of the fastest algorithms on average.
- One of the fastest algorithms on average.
- The list is being traversed sequentially, which produces very good locality of reference and cache behavior for arrays.

Cons:

- Space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly[10], especially when it comes to large data sets.
- The worst-case complexity is O(n^2)

III. COMPARITIVE STUDY OF ALL ALGORITHMS

We summaries the different analytical aspects of all six sorting algorithms in the table given below.

Table 1: Analytical Comparison

Name	Best	Average	Worst	Memory	Stable	Method
Bubble sort					Yes	Exchanging
Selection sort					No	Selection

Name	Best	Average	Worst	Memory	Stable	Method
Insertion sort					Yes	Insertion
Merge sort	-	-	-	worst case	Yes	Merging
Heap sort	-	-	-		No	Selection
Quick sort	-	-		- on average, worst case is ;	typical in-place sort is not stable	Partitioning

We implemented the all six sorting algorithms in C# for N=10. 100, 1000 and 10000, and calculated the run tine by using System.Diagnostics.Stopwatch Class. The resultant run time is given in the table given below.

Table 2: Run Time

	Run(Time in Microseconds)					
N(Number of elements)	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Heap sort	Quick sort
10	202	251	224	538	511	302
100	297	296	242	612	688	321
1000	8382	3689	3277	3398	3412	612
10000	808922	347522	263499	91991	9312	3299

The graph constructed for the above table is given below in figure 1.

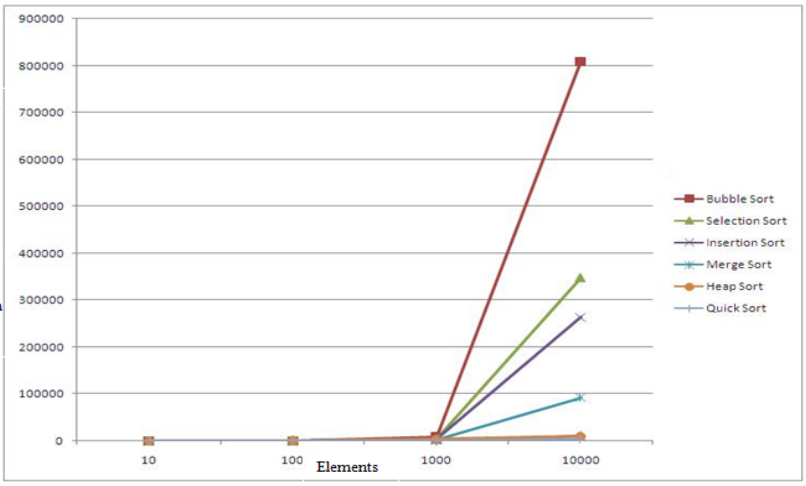


Fig 1. Running Time of Different Sorting Algorithms on Different size of Data

IV. CONCLUSION

From the above analysis it can be said that, Bubble Sort, Selection Sort and Insertion Sort are fairly straightforward, and easy to implement. Merge Sort, Heap Sort and Quick Sort are more complicated, but also much faster for large lists. Quick Sort is, on average, the fastest algorithm but it needs enough memory. Bubble Sort algorithm is the slowest but needs no extra memory. Quick sort is more often used as external sorting. On the above comparison and the resultant analysis, it is clear to use Bubble Sort, Selection Sort and Insertion Sort for small data set whereas Merge Sort, Heap Sort and Quick Sort for large data sets.

REFERENCES

[1] Ahmed M. Aliyu and Dr. P. B. Zirra. 2013. A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays. The International Journal Of Engineering And Science (IJES).Volume.2. Issue. 7.Pages: 25-30.

[2] Astrachan, Owen. 2003 Bubble Sort: An Archaeological Algorithmic Analysis. SIGCSE, ACM.

[3] Bender *et al.*, (2006) Insertion Sort is O(n log n). Theory of Computing Systems **39** (3): 391.

[4] Sultanullah Jadoon, Salman Faiz Solehria, Mubashir Qayum, "Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study*",International Journal of Electrical & Computer Sciences* IJECS-IJENS Vol: 11 No: 02.

[5] D.T.V Dharmajee Rao and B.Ramesh. 2012 . International Journal of Modern Engineering Research (IJMER) Vol.2, Issue.4, pp-2908-2912.

[6] Tarundeep Singh Sodhi, Surmeet Kaur, Snehdeep Kaur, "Enhanced Insertion Sort Algorithm ",*International Journal of Computer Applications* (0975 – 8887) Volume 64– No.21, February 2013.

[7] Deependra Kr. Dwivedi. 2011. Comparison Analysis of Best Sorting Algorithms. VSRD-IJCSIT, Vol. 1 (4), 2011, 261-267.

[8] Franceschini, Gianni, 2007. Sorting Stably, in Place, with O(n log n) Comparisons and O(n) Moves. Theory of Computing Systems **40** (4): 327–353.

[9] J. S. Vitter. 2006. Algorithms and Data Structures for External Memory, Foundation and Trends in Theoretical Computer Science, vol 2, no 4, pp 305–474. Lacey.Stephen and Box. Richard.1991. A Fast Easy Sort. ACM. Volume 16 Issue 4:315

[10] Mishra.D.A, 2009. Selection of Best Sorting Algorithm for a Particular Problem. Computer Science And Engineering Department Thapar University

[11] Oded Goldreich. 2008. Computational complexity: a conceptual perspective. Newsletter ACM SIGACT News archive Volume 39 Issue 3, Pages 35-39.

[12] T. H. Cormen, et al.2001. Introduction to Algorithms. MIT Press, Cambridge, MA, 2nd edition.

[13] Aayush Agarwal ,Vikas Pardesi , Namita Agarwal, "A New Approach To Sorting: Min-Max Sorting Algorithm" *International Journal of Engineering Research & Technology* (IJERT) Vol. 2 Issue 5, May – 2013.

[14] Mahfooz Alam, Ayush Chugh, "*Sorting Algorithm: An Empirical Analysis"*, International Journal of Engineering Science and Innovative Technology (IJESIT) Volume 3, Issue 2, March 2014.

[15] Miss. Pooja K. Chhatwani, Miss. Jayashree S. Somani, "*Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure*", International Journal of Advanced Research in Computer Science and Software Engineering., Volume 3, Issue 11, November 2013