Volume 3, Issue 11, November 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure

Miss. Pooja K. Chhatwani, Miss. Jayashree S. Somani
Lecturer, Information Technology,
Dr. N. P. Hirani Institute of Polytechnic, Pusad,
Maharashtra, India

Abstract— An algorithm is precise specification of a sequence of instruction to be carried out in order to solve a given problem. Sorting is considered as a fundamental operation in computer science as it is used as an intermediate step in many operations. Sorting refers to the process of arranging list of elements in a particular order. The elements are arranged in increasing or decreasing order of their key values. This research paper presents the different types of sorting algorithms of data structure like Bubble Sort, Selection Sort, Insertion Sort, Merge Sort and Quick Sort and also gives their performance analysis with respect to time complexity. These five 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 five 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, Quick Sort, Complexity

I. INTRODUCTION

Algorithm is an unambiguous, step-by-step procedure for solving a problem, which is guaranteed to terminate after a finite number of steps. In other words algorithm is logical representation of the instructions which should be executed to perform meaningful task. For a given problem, there are generally many different algorithms for solving it. Some algorithms are more efficient than others, in that less time or memory is required to execute them. The analysis of algorithms studies time and memory requirements of algorithms and the way those requirements depend on the number of items being processed. Sorting is generally understood to be the process of rearranging a given set of objects 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 depends on one of the comparison-based sorting algorithms: Bubble, Insertion, Selection sort etc. Hence sorting is an almost universally performed and hence, considered as a fundamental activity. The usefulness and significance of sorting is depicted from the day to day application of sorting in real-life objects. For instance, objects are sorted in: Telephone directories, income tax files, tables of contents, libraries, dictionaries.

The methods of sorting can be divided into two categories:

INTERNAL SORTING: If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used.

EXTERNAL SORTING: When the data to be sorted can't be accommodated in the memory at the time and some has to be kept in auxiliary memory (hard disk, floppy, tape etc) , then external sorting method are used.

The complexity of a sorting algorithm measures the running time of function in which 'n' numbers of items are sorted. The choice of which sorting method is suitable for a problem depends on various efficiency considerations for different problem. Three most important of these considerations are:

- The length of time spent by programmer in coding a particular sorting program.
- Amount of machine time necessary for running the program.
- The amount of memory necessary for running program.
- Stability-does the sort preserve the order of keys with equal values.

II. WORKING PROCEDURE OF ALGORITHMS

A. Bubble Sort

The Bubble Sort is the simplest sorting technique, in which smallest data element are moved or 'bubbled up' to the top. In this method, first element is compared with the next element in the array. If the element is first element is larger, then swap or interchange them and move the smaller element to the top position otherwise no swapping or interchange of element is required. After (N-1) comparisons, the largest element among all the element will descends to the bottom of array.

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

BUBBLE (DAT A, 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 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	**23**	**15**	**29**	**11**	**1**
1st Pass	**23**	**15**	**29**	**11**	**1**
	15	**23**	**29**	**11**	**1**
	15	**23**	**29**	**11**	**1**
	15	**23**	**11**	**29**	**1**
	15	**23**	**11**	**1**	**29**
2nd Pass	**15**	**23**	**11**	**1**	**29**
	15	**11**	**23**	**1**	**29**
	15	**11**	**1**	**23**	**29**
	15	**11**	**1**	**23**	**29**
3rd Pass	**15**	**11**	**1**	**23**	**29**
	11	**15**	**1**	**23**	**29**
	11	**1**	**15**	**23**	**29**
4th Pass	**11**	**1**	**15**	**23**	**29**
Sorted	**1**	**11**	**15**	**23**	**29**

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 = O (n^2)

Worst Case= O (n^2)

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. In this, to sort the d ascending order, the 0^{th} element is compare with all other element. If the 0^{th} element is found to be greater than the compared element then they are interchanged. In this way, after first iteration, the smallest element is placed at 0^{th} position. The procedure is repeated for 1^{st} element and so on.

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

SELECTION SORT (DATA, N)

1. Repeat Steps 2 & 3 for K=1 to N-1.
2. Set MIN = DATA[K] and LOC = K
   [Initializes Pointers]
3. Repeat for J= K+1, K+2,… N
4. If MIN > DATA[J] then:
   a) MIN= DATA [J]
   b) LOC = DATA [J]
   c) LOC = J
   [End of Loop]
5. Set TEMP = DATA[K],
   DATA [K] = DATA [LOC]
   DATA [LOC] = TEMP
6. Exit

2) *Example:*

Elements	**0**	**1**	**2**	**3**	**4**
Data	**23**	**15**	**29**	**11**	**1**
1st Pass	**23**	**15**	**29**	**11**	**1**
	15	**23**	**29**	**11**	**1**
	15	**23**	**29**	**11**	**1**
	11	**23**	**29**	**15**	**1**
	1	**23**	**29**	**15**	**11**
2nd Pass	**1**	**23**	**29**	**15**	**11**
	1	**23**	**29**	**15**	**11**
	1	**15**	**29**	**23**	**11**
	1	**11**	**29**	**23**	**15**
3rd Pass	**1**	**11**	**29**	**23**	**15**
	1	**11**	**23**	**29**	**15**
	1	**11**	**15**	**29**	**23**
4th Pass	**1**	**11**	**15**	**29**	**23**
Sorted	**1**	**11**	**15**	**23**	**29**

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, the first iteration starts with comparison of 1^{st} element with 0^{th} element. In the second iteration the element is compared with 0^{th} and 1^{st} element. In general in every iteration an element is compared with all elements. If at same

point it is found the element can be inserted at a position then space is created for it by shifting the other element one position right and inserting the element at the suitable position. This procedure is repeated for all the element in the array.

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

INSERTION (DATA, N)

1. Set A[0] = -∞ [Initializes sentinel elements]
2. Repeat Steps 3 to 5 for K = 2 ,3,…N
3. Set TEMP = DATA[K] and PTR = K-1
4. Repeat while TEMP < DATA[PTR] :
   a) Set DATA[PTR+1] = A[PTR]
      [Moves element forward]
   b) Set PTR =PTR – 1
   [End of loop]
5. Set DATA[PTR+1] = TEMP
   [Inserts the elements in proper place]
   [End of step 2 loop]
6. Exit.

2) *Example*

Elements	**0**	**1**	**2**	**3**	**4**
Data	**23**	**15**	**29**	**11**	**1**
1st Pass	**23**	**15**	**29**	**11**	**1**
2nd Pass	**15**	**23**	**29**	**11**	**1**
3rd Pass	**15**	**23**	**29**	**11**	**1**
4th Pass	**11**	**15**	**23**	**29**	**1**
Sorted	**1**	**11**	**15**	**23**	**29**

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:

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

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 sorts 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

- Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

1) 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:

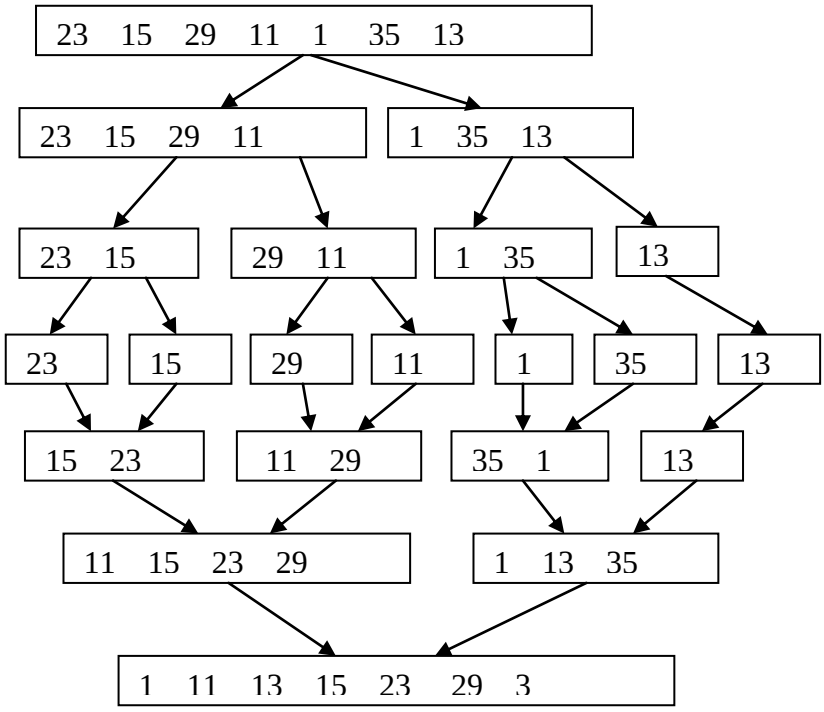

Fig. 4: Working procedure of Merge Sort

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. *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: The basic algorithm to sort an array DATA of n elements is described as follows:

1. If n < =1, then exit.
2. Pick any element P in DATA. This will be called as Pivot.

Rearrange the elements of array by moving all elements X > P to the right of P and all elements X <= P to the left of P. If the place of P after rearrangement is Y, all elements with value less than P, appear in DATA[0], DATA[1]..... DATA[Y-1] and all those value greater than P appear in DATA[Y+1].....DATA[n-1].

3. Apply quick sort recursively to DATA[0].....DATA[Y-1] and DATA[Y+1].....DATA[n-1].

Entire array will thus be sorted by selecting a pivot P.

- Partitioning the array around P
- Recursively sorting left part of the array
- Recursively sorting right part of the array

2) Example:

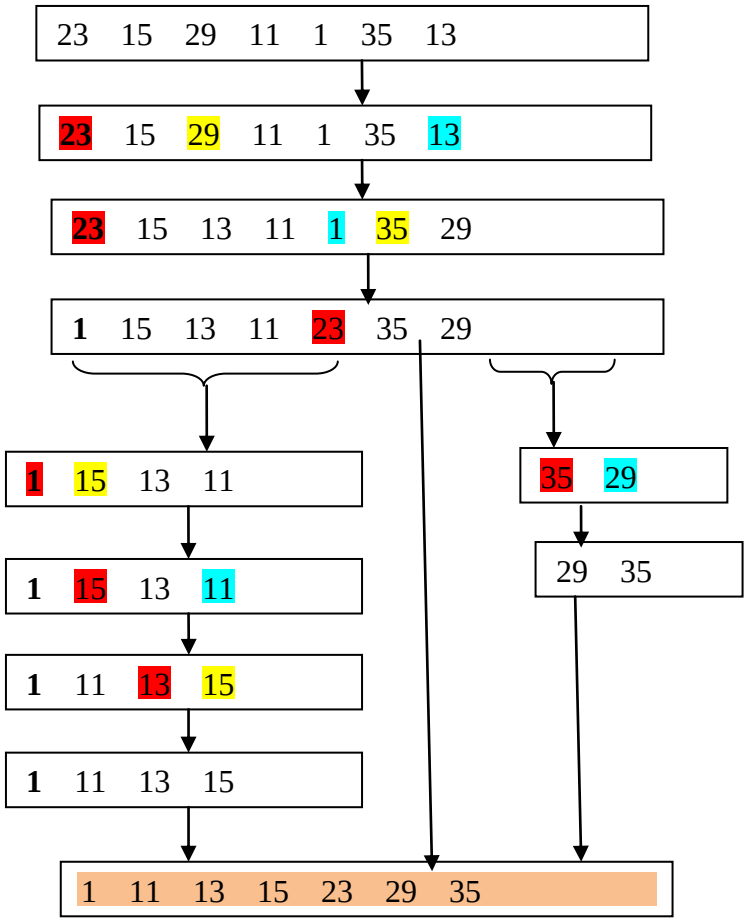

Fig. 5: Working procedure of Quick Sort

3) 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 very 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)

4) 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, especially when it comes to large data sets.
- The worst-case complexity is O(n^2)

III. COMPARITIVE STUDY OF ALL ALGORITHM

TABLE I
COMPLEXITY COMPARISON

Parameter	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Time Complexity 1.Best Case 2.Average Case 3.Worst Case	 O(n) $O(n^2)$ $O(n^2)$	 $O(n^2)$ $O(n^2)$ $O(n^2)$	 O(n) $O(n^2)$ $O(n^2)$	 O(nlogn) O(nlogn) O(nlogn)	 O(nlogn) O(nlogn) $O(n^2)$
Space Complexity	O(1)	O(1)	O(1)	O(n)	O(1)
Method	Exchange	Selection	Insertion	Merging	Partitioning
Stable	Yes	No	Yes	Yes	Depends on Elements
In-Place	Yes	Yes	Yes	No	Yes
Type	Internal	Internal	Internal	Can be both Internal and External	Internal
Strategy	Scan all the elements & bubble up largest element	Scan all the elements & sort the list	Scan all the elements & insert the smallest element before largest	Divides An Array Into Two Separate Lists(Sub Arrays)	Concept of pivot element

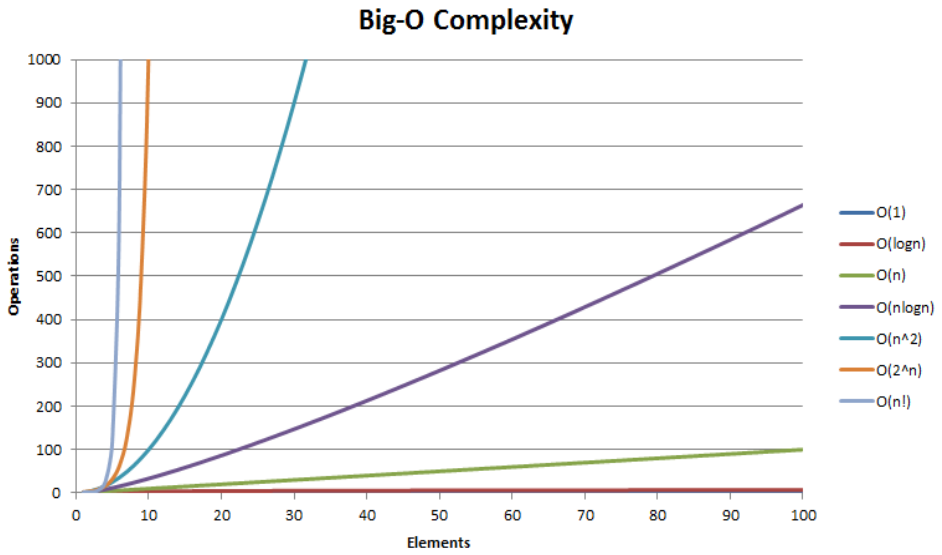

Fig 6: Complexity Statistics

IV. CONCLUSION

From the above analysis it can be said that, Bubble Sort, Selection Sort and Insertion Sort are fairly straightforward, but they are relatively inefficient except for small lists. Merge Sort and Quick Sort are more complicated, but also much faster for large lists. Quick Sort is, on average, the fastest algorithm. Bubble Sort algorithm is the slowest.

ACKNOWLEDGEMENT

We would like to express our gratitude to all those who gave us the possibility to complete this work. We would like to thank Prof. T. N. Boob (Principal of Dr. N. P.Hirani Institute of Polytechnic, Pusad) and Mr. P. B. Shah (H.O.D. Of Information Technology Department) for providing support and material related to the area of this research.

REFERENCES

[1] Seymour Lipschutz and G A Vijayalakshmi Pai , *Data Structures*, (Tata McGraw Hill companies), Indian adapted edition 2006-07 West patel nagar,New Delhi-110063.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest , *Introduction to Algorithms*, Fifth Indian printing (Prentice Hall of India private limited), New Delhi-110001

[3] Eshan Kapur, Parveen Kumar and Sahil Gupta,"Proposal Of A Two Way Sorting Algorithm And Performance Comparison With Existing Algorithms*" International Journal of Computer Science, Engineering and Applications* (IJCSEA) Vol.2, No.3, June 2012.

[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] Ahmed M. Aliyu, Dr. P. B. Zirra, "A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays", *The International Journal Of Engineering And Science* (IJES), volume 2,Issue-7.

[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] G P Raja Sekhar , Lecture Notes :"Design & Analysis of Algorithms", Department of Mathematics I I T Kharagpur.

[8] Rupesh Srivastava, Tarun Tiwari Sweetesh Singh, "Bidirectional Expansion–. Insertion Algorithm for Sorting", 2009, Second *International Conference on Emerging Trend in Engineering and Technology*, ICETET-09.

[9] 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.

