International Journal of Mechanical Engineering and Technology (IJMET)
Volume 8, Issue 8, August 2017, pp. 488–498, Article ID: IJMET_08_08_055
Available online at http://iaeme.com/Home/issue/IJMET?Volume=8&Issue=8
ISSN Print: 0976-6340 and ISSN Online: 0976-6359

AN EMPIRICAL STUDY AND ANALYSIS ON SORTING ALGORITHMS

R Gangadhar Reddy
Assistant Professor, Dept. of ECE, Institute of Aeronautical Engineering, Hyderabad, india

P R Anisha, C Kishor Kumar Reddy
Assistant Professor, Department of CSE,
Stanley College of Engineering. & Tech. for Women, Hyderabad

M. Srinivasa Reddy,
Associate Prof, Dept. of ECE, MLR Institute of Technology, Hyderabad, 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. Sorting involves rearranging information into either ascending or descending order. Sorting is considered as a fundamental operation in computer science as it is used as an intermediate step in many operations. The goal of this paper is to review on various different sorting algorithms and compares the various performance factors among them.

Keywords: Bubble Sort; Selection Sort; Insertion Sort; Quick Sort; Merge Sort; Heap Sort; Cocktail Sort.

Cite this Article: R Gangadhar Reddy, P R Anisha, C Kishor Kumar Reddy and M. Srinivasa Reddy, An Empirical Study And Analysis On Sorting Algorithms, International Journal of Mechanical Engineering and Technology 8(8), 2017, pp. 488–498.
http://iaeme.com/Home/issue/IJMET?Volume=8&Issue=8

1. INTRODUCTION

Sorting is the rearrangement of a list into an order defined by a monotonically increasing or decreasing sequence of sort keys, where each sort key is a single-valued function of the corresponding element of the list. Sorting reorders a list into a sequence suitable for further

processing or searching. Often the sorted output is intended for people to read; sorting makes it much easier to understand the data and to find a datum.

Sorting is used in many types of programs and on all kinds of data. It is such a common, resource-consuming operation that sorting algorithms and the creation of optimal implementations constitute an important branch of computer science.

2. CRITERIA FOR SORTING:

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)).

3. CLASSIFICATION OF SORTING ALGORITHMS:

Computational complexity (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n) and bad behavior is O(n2). Ideal behavior for a serial sort is O(n), but this is not possible in the average case. Optimal parallel sorting is O (log n). Comparison-based sorting algorithms, need at least O(n log n) comparisons for most inputs.

- **Memory usage** (and use of other computer resources): In particular, some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in-place".
- **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).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- **General method:** insertion, exchange, selection, merging, etc.
- **Exchange sorts** include bubble sort and quicksort.
- **Selection sorts** include heap sort. Also whether the algorithm is serial or parallel.
- **Adaptability:** Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

4. VARIOUS SORTING ALGORITHMS

4.1. Bubble sort

A bubble sort compares the adjacent elements in an array and exchanges or swaps with it if they are out of order. This occurs in the process of passes. In this way, smaller values “bubble” to the top and larger to the end. This process continues till the end where the array is sorted completely from small to large.

Figure 1 depicts, where

- first it checks all the elements in given array and compares the first two numbers.
- The number 54 is checked with each number. If 1st number is greater than 2nd number then it swaps, Same case occurred in the below example for first pass hence exchange takes place. (54>26)
- Similarly, 54 is compared with 93 (54<93) hence no exchange takes place.
- This process continues till all values are in order.

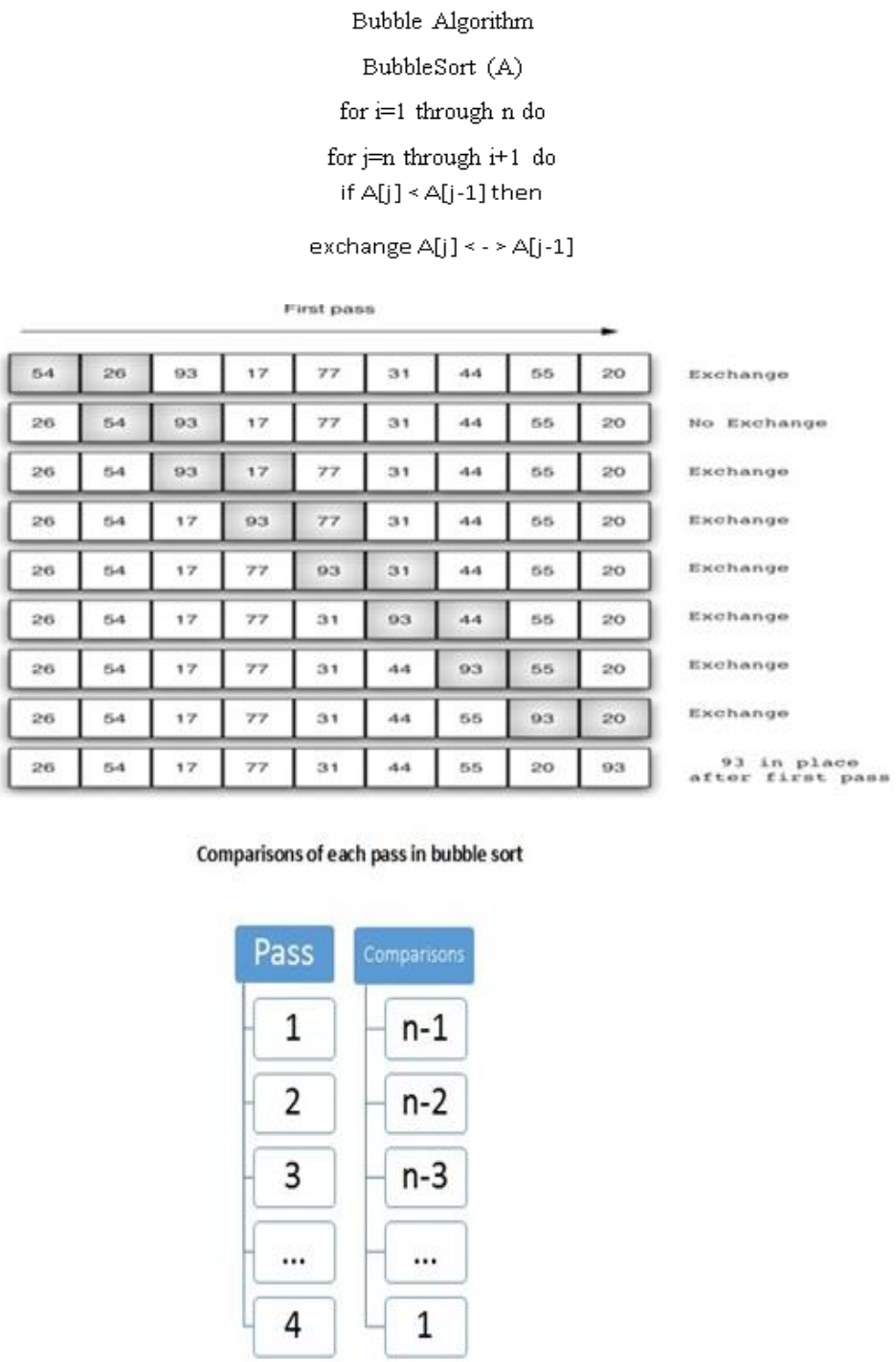

Figure 1 Explanation of Bubble Sort

4.2. SELECTION SORT

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place.

After the second pass, the next largest is in place. This process continues and requires n−1 passes to sort n items, since the final item must be in place after the (n−1) pass.

Figure 2 depicts, on each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on

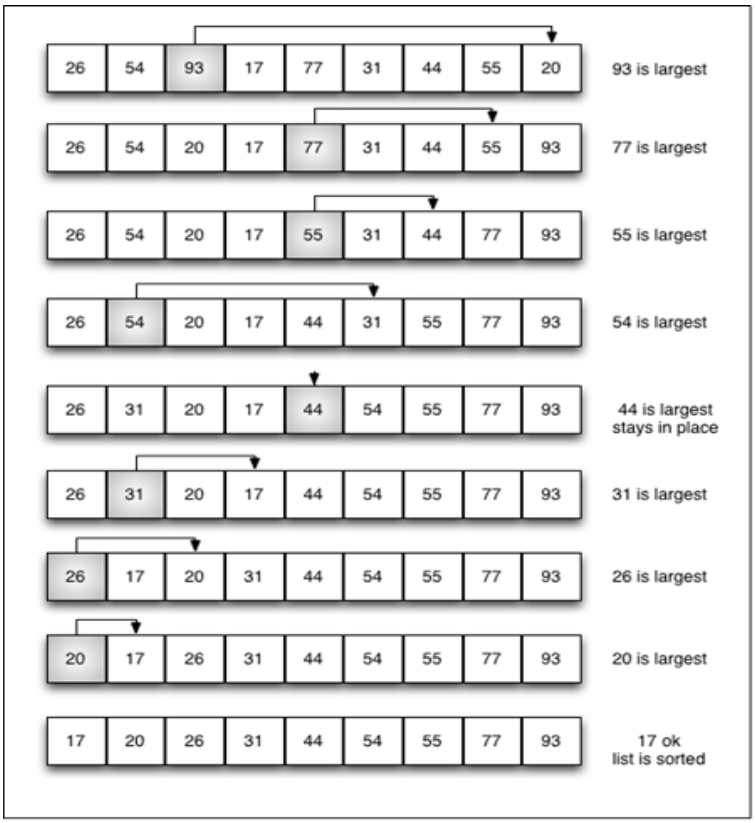

Figure 2 Explanation of Selection Sort

```
for i ← 0 to N − 1 do
    Min ← i
    for j ← 0 to N − 1 do
        if A[j] ≤ A[Min] then
            Min ← j
        end if
    end for
    if i ≠ Min then
        Swap(A[i],A[Min])
    end if
end for
```

4.3. Insertion Sort

The insertion sort, although still O(n2), works in a slightly different way. It always maintains a sorted sub list in the lower positions of the list. Each new item is then "inserted" back into the previous sub list such that the sorted sub list is one item larger. We begin by assuming that a list with one item (position 00) is already sorted. On each pass, one for each item 1 through n−1, the current item is checked against those in the already sorted sub list. As we look back into the already sorted sub list, we shift those items that are greater to the right. When we reach a smaller item or the end of the sub list, the current item can be inserted.

Figure 3 depicts, at this point in the algorithm, a sorted sub list of five items consisting of 17, 26, 54, 77, and 93 exists. We want to insert 31 back into the already sorted items. The first comparison against 93 causes 93 to be shifted to the right. 77 and 54 are also shifted. When the item 26 is encountered, the shifting process stops and 31 is placed in the open position. Now we have a sorted sub list of six items.

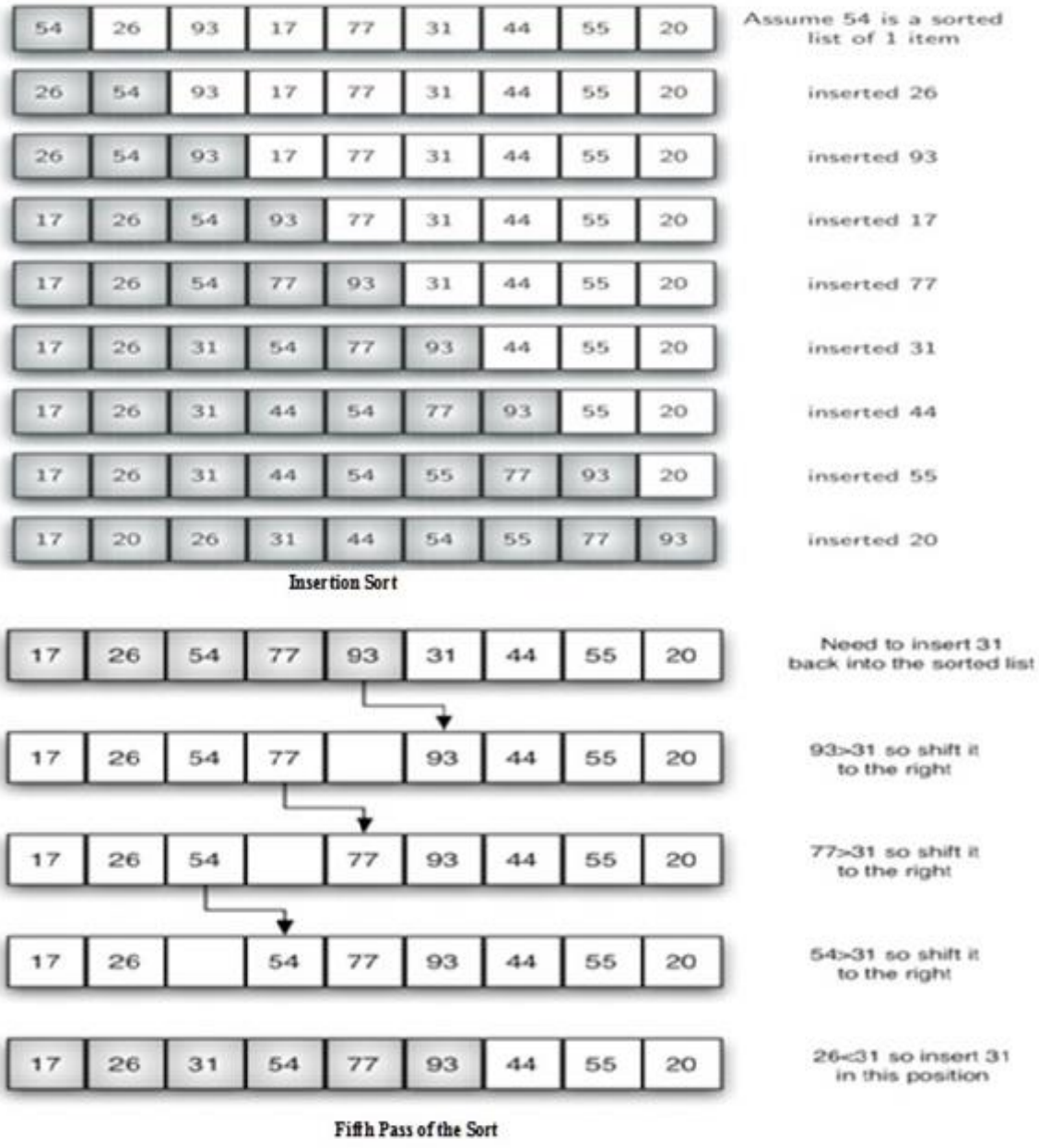

Figure 3 Explanation of Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

4.4. Quick Sort

The quick sort uses divide and conquer. A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

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

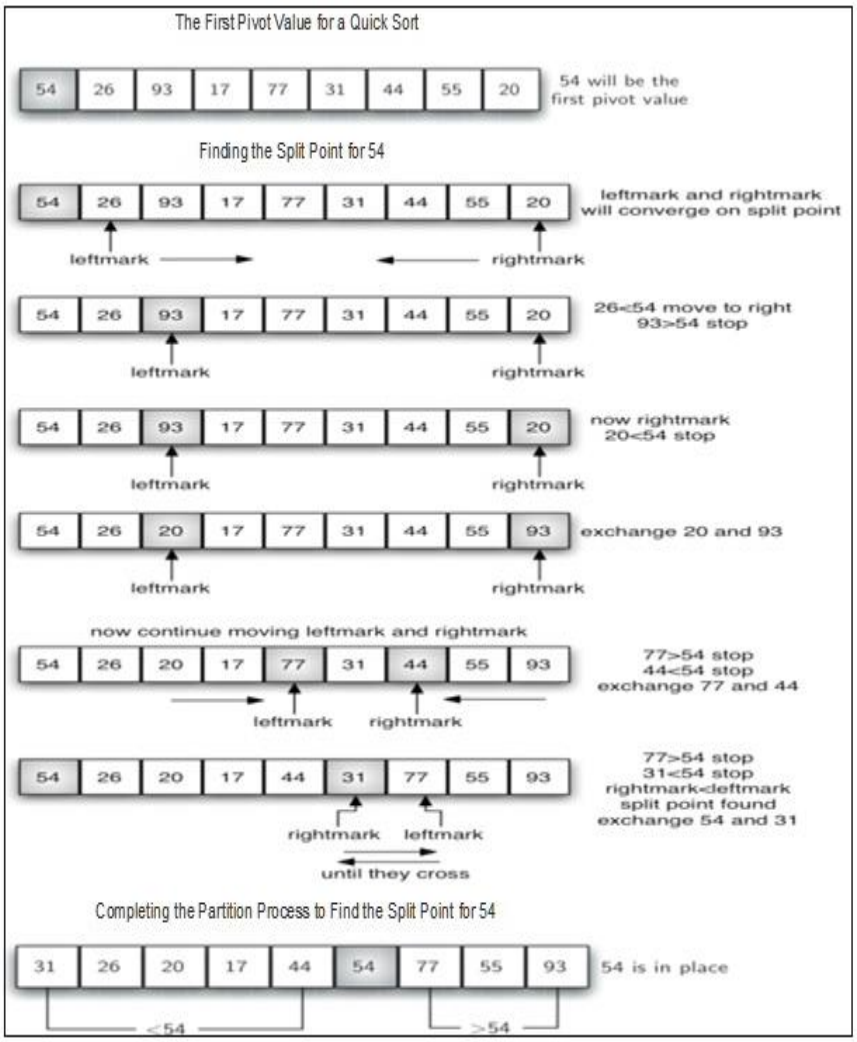

Figure 4 Explanation of Quick Sort

Figure 4 shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

4.5. Merge Sort

Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Figure 5 depicts algorithm of merge sort

Figure 6 shows our familiar example list as it is being split by Merge Sort. The simple lists, now sorted, as they are merged back together.

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Figure 5 Merge Sort Algorithm

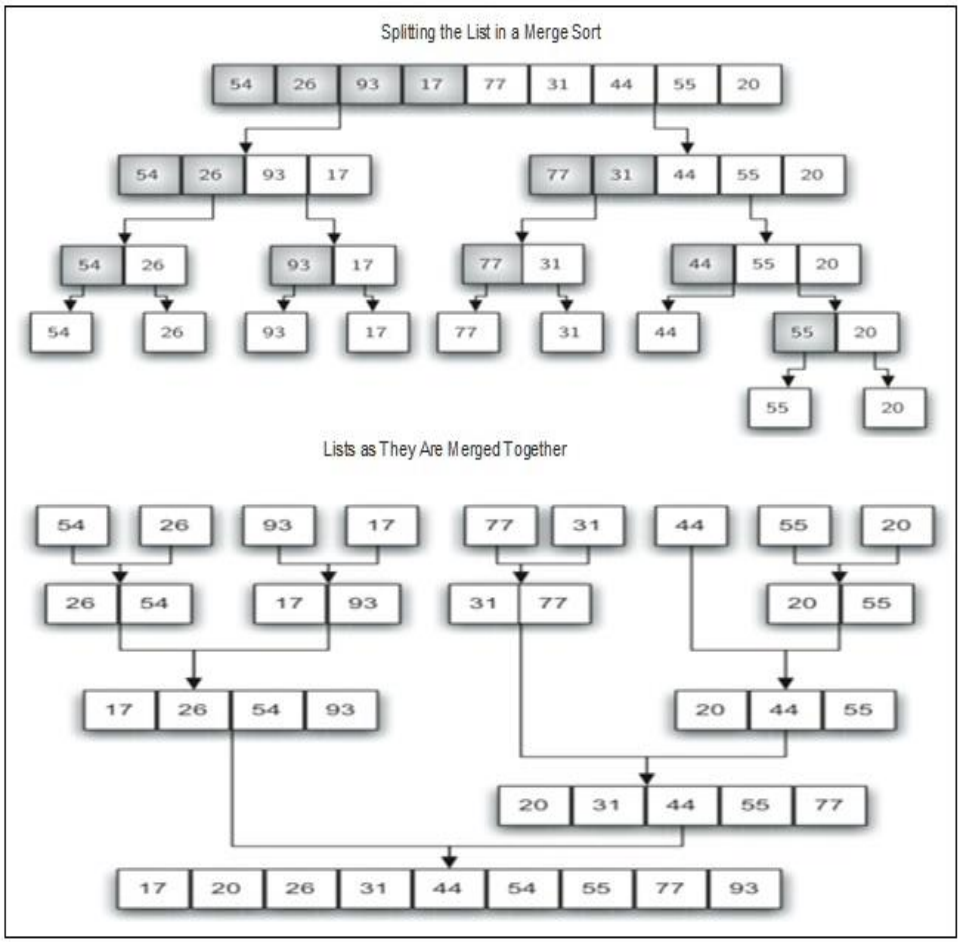

Figure 6 Explanation of Merge Sort

4.6. Heap Sort

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree.

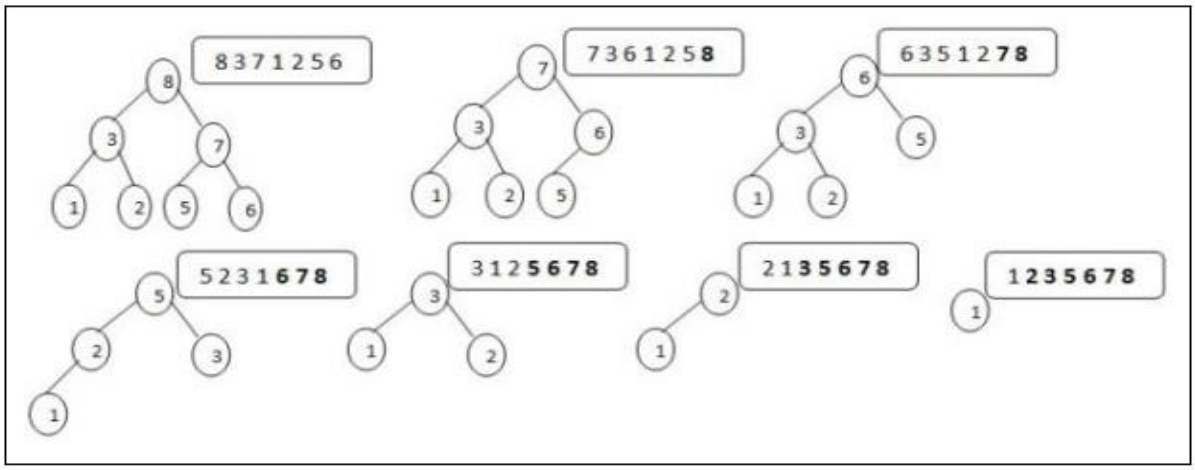

Figure 7 Explanation of Heap Sort

4.7. Cocktail Shaker Sort

The cocktail shaker sort is an improvement on the Bubble Sort. The improvement is basically that values "bubble" both directions through the array, because on each iteration the cocktail shaker sort bubble sorts once forward and once backward. In simple words, the difference between cocktail sort and bubble sort is that instead of repeatedly passing through the list from bottom to top, it passes alternately from bottom to top and then from top to bottom.

Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O (log n) time, instead of O (n) for a linear scan as in simple selection sort. This allows Heap sort to run in O (n log n) time, and this is also the worst-case complexity

The result is that it has a slightly better performance than bubble sort, because it sorts in both directions. (Bubble sort can only move items backwards one step per iteration.) Normally cocktail sort or shaker sort pass (one time in both directions) is counted as two bubble sort passes. In a typical implementation the cocktail sort is less than two times faster than a bubble sort implementation. Because you have to implement a loop in both directions that is changing each pass it is slightly more difficult to implement. Figure 8 depicts the explanation.

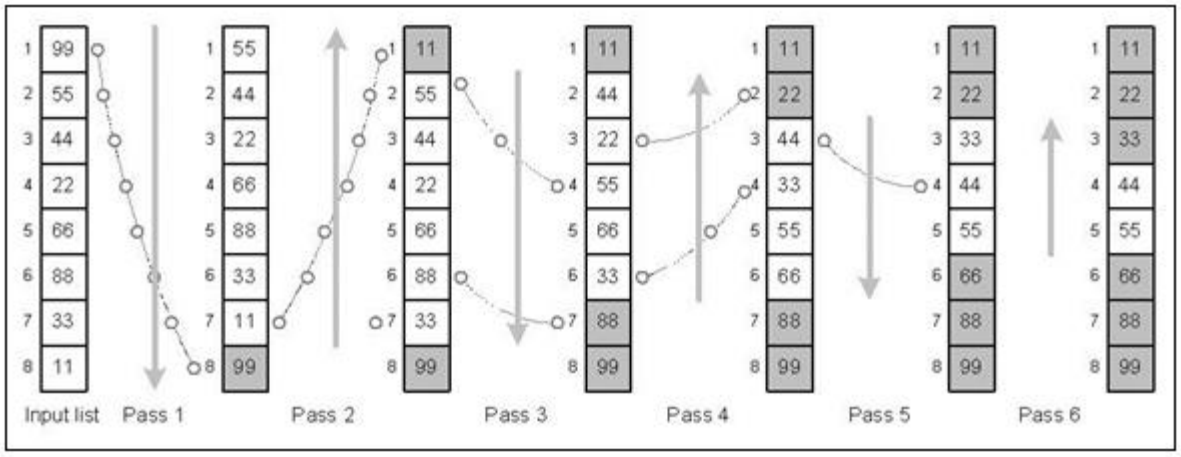

Figure 8 Explanation of Cocktail Shaker Sort

5. ANALYSIS OF SORTING ALGORITHMS

Figure 9 depicts the comparison of various sorting algorithms, illustrates time complexity in terms of best, average and worse, space complexity and stability complexity. As per the analysis cocktail shaker algorithm is outperforming.

		TIME COMPLEXITY					
		BEST	*WORST*	*AVERAGE*	*SPACE*	*STABLE*	*COMMENT*
COMPARISION SORT	BUBBLE SORT	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	For each pairing of indices,swap the elements if they are out of order
	SELECTION SORT	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	Swap happens only once in a single pass
	INSERTION SORT	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Very small constant factor even if the complexity is O(n^2).Array which is sorted is best case and sorted in reverse order is worst-case
	MERGE SORT	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	Best to sort linked list.Best for very large number of elements which cannot fit in memory
	QUICK SORT	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	When pivot divide in two equal halves it is best-case and array already sorted -1/(n-1)
	HEAP SORT	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	-
	COCKTAIL SHAKER SORT	O(n)	O(n^2)	O(n^2)	O(1)	Yes	-

Figure 9 Comparison of various sorting algorithms

6. CONCLUSION

Sorting is used in many types of programs and on all kinds of data. It is such a common, resource-consuming operation that sorting algorithms and the creation of optimal implementations constitute an important branch of computer science. As per the survey, cocktail sort is an improved version of bubble sort, where we can perform backward and forward sorting simultaneously, which increases the efficiency.

REFERENCES

[1] Pradip Dey and Manas Ghosh, “Programming in C”, Oxford Higher education, 2007(1st edition) 2011(2nd edition).

[2] Yashavant Kanetkar, “Working with C”, BPB Publications, 1994

[3] Brian W.Kernighan and Dennis M.Ritchie, ”The C programming language”, AT&T Bell Laboratories

[4] Jehad Alnihoud and Rami Mansi, “An Enhancement of Major Sorting Algorithms”, http://ccis2k.org/iajit/PDF/vol.7,no.1/9.pdf , 2010

[5] Robert Sedgewick, “Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms. Bundle of Algorithms in Java”, AddisonWesley Professional, Third Edition.

[6] V. Mansotra and Kr. Sourabh, “Implementing Bubble Sort Using a New Approach”, http://www.bvicam.ac.in/news/INDIACom%202011/86.pdf , 2011

[7] Pankaj Sareen, “Comparison of Sorting Algorithms (On the Basis of Average Case)”, https://www.ijarcsse.com/docs/papers/Volume_3/3_March2013/V3I3-0319.pdf , 2013

[8] Ahmed M. Aliyu and Dr. P. B. Zirra, “A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays”, http://www.theijes.com/papers/v2-i7/Part.3/D0273025030.pdf, 2013.

[9] You Yang, Ping Yu and Yan Gan, “Experimental Study on the Five Sort Algorithms”, http://home.iitk.ac.in/~depak/cs300/pres/Experimental-Study-on-the-Five-Sort-Algorithms.pptx

[10] Eshan Kapur, Parveen Kumar and Sahil Gupta, “Proposal of a two way sorting algorithm and performance with existing algorithms”, airccse.org/journal/ijcsea/papers/23112ijcsea06.pdf 2012

[11] https://en.wikipedia.org/wiki/Sorting_algorithm

[12] https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm

[13] Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani & Nabeel Imhammed Zanoon , "Review on Sorting Algorithms A Comparative Study",

[14] http://www.cscjournals.org/manuscript/Journals/IJCSS/Volume7/Issue3/IJCSS-877.pdf

[15] D.Garg," Selection O. Best Sorting Algorithm", International Journal of Intelligent Information Processing, http://www.academia.edu/1976253/Selection_of_Best_Sorting_Algorithm, 2008

[16] Malika Dawra and Priti, "Parallel Implementation of Sorting Algorithms", http://ijcsi.org/papers/IJCSI-9-4-3-164-169.pdf, 2012

[17] Gaurav Kocher and Nikita Agrawal, "Analysis and Review of Sorting Algorithms", http://www.ijser.in/archives/v2i3/SjIwMTMxODE=.pdf, 2014

[18] Mahfooz Alam and Ayush Chugh, "Sorting Algorithm:An Empirical Analysis", http://www.ijesit.com/Volume%203/Issue%202/IJESIT201402_16.pdf , 2014.

[19] https://www.toptal.com/developers/sorting-algorithms

[20] http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html

[21] http://nptel.ac.in/courses/Webcoursecontents/IIT%20Guwahati/data_str_algo/Module_5/binder1.pdf

[22] Asokan M, VISUALIZATION OF SORTING ALGORITHMS USING FLASH, International Journal of Graphics and Multimedia (IJGM), Volume 5, Issue 1, January - April 2014, pp. 01-15

[23] Ms Geeta and Ms. AnubhootiPapola, AN EFFICIENT SORTING ALGORITHM: INCRECOMPARISION SORT, International Journal of Advanced Research in Engineering and Technology (IJARET), Volume 5, Issue 9, September (2014), pp. 10-16