ANALYSIS AND COMPARISION OF DIFFERENT SORTING ALGORITHM

Mr. Rajesh Kumar Singh
Computer Science and Engineering
VIT University Vellore
Tamil Nadu, India

Mr. Boominathan P.
Assistant Professor, Department of Software Systems
VIT University, Vellore
Tamil Nadu, India

Abstract

An algorithm or pseudocode in computer science is a set of instructions which are to be done to successfully solve any problem. Sorting is fundamental aspect in computer science which is used in almost every code. In sorting we arrange the data in ascending or descending order or any particular order. This research paper explains about some famous sorting algorithms which are used in today's life like insertion sort, bubble sort, selection sort, heap sort, quick sort, merge sort etc. Here we are focusing on mainly six algorithms and will see how much time on average each algorithm takes to execute successfully. There is a lot of development and more needed to be done in sorting algorithms, that's why we have chosen this as research topic. On the basis of different input values we will try to find out the best algorithm on execution time basis.

Key words—**bubble sort, insertion sort, selection sort, heap sort, merge sort, quick sort**

I. Introduction

Algorithm is a procedure in which steps are there to solve any problem. We can also say algorithm is a logical representation of steps which are required to be executed to solve the problem. Algorithm can be different for same problem for which unique possible way is there. Implementation of algorithm can be done in any language. For a given problem, there are many codes are possible one is simple, other can be tough, one is efficient and other cannot be that much efficient, one may take less time and other will take more time. Hence this apply for sorting also. Sorting can be done in many ways. Sorting is a way to arrange our data in some particular manner. It is introduced in late 1950s and still research and development work is being carried out in this topic. Many more are interested in this topic and still try to make some efficient codes. It is one of the most exciting research topic among researchers. There are many real life, day-to-day examples which uses sorting technique some way. Here in this paper we are focusing on mainly six sorting algorithms namely bubble sort, insertion sort, selection sort, heap sort, quick sort and merge sort.

II.WORKING PROCEDURE OF EACH ALGORITHM

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.used. If we talk about the complexity of this, it has time complexity of $O(n^2)$ in average and worst case.

```
Function bubble(arr,n)
   For i=1 to n-1
      For j=0 to n-1-i
         If (arr[j]>arr[j+1])
            Swap arr[j], arr[j+1]
End function
```

ELEMENTS	0	1	2	3	4
DATA	24	16	30	12	2
1ST PASS	24	16	30	12	2
	16	24	30	12	2
	16	24	30	12	2
	16	24	12	30	2
	16	24	12	2	30
2ND PASS	16	24	12	2	30
	16	24	12	2	30
	16	12	24	2	30
	16	12	2	24	30
3RD PASS	16	12	2	24	30
	12	16	2	24	30
	12	2	16	24	30
4TH PASS	12	2	16	24	30
	2	12	16	24	30

Fig 1: Working of Bubble Sort

B. INSERTION SORT

Insertion sort, just like bubble sort, is a simple sorting algorithm which works better in sorted lists and mostly used for small data list. As its name suggests it picks element and place them in their proper order. It divides the array in two parts one is unsorted and one is sorted. It is seen that it is expensive because of shifting is involved in it, and it shifts one by one. Shell sort is one of the variant of the insertion sort, that is used for larger arrays and is more efficient than given algorithm. The time complexity of insertion sort is same as bubble sort, as it also has time complexity of $O(n^2)$ in average and worst case both.

```
Function insertion(arr, n)
    For i = 1 to n-1
       For j = i to 1
          If  arr[j] < arr[j-1]
              swap arr[j], arr[j-1]
          Else
              End inner loop
End function
```

ELEMENTS	0	1	2	3	4
DATA	24	16	30	12	2
1ST PASS	24	16	30	12	2
	16	24	30	12	2
2ND PASS	16	24	30	12	2
3RD PASS	16	24	30	12	2
	16	24	12	30	2
	16	12	24	30	2
	12	16	24	30	2
4TH PASS	12	16	24	2	30
	12	16	2	24	30
	12	2	16	24	30
	2	12	16	24	30

Fig 2: Working of Insertion Sort

C. SELECTION SORT

The selection sort is considered as the simplest and the easiest sorting method. This is also be called as in place sorting algorithm. In this algorithm, to sort the array in ascending order, it finds the smallest element and then swap it in the 0^{th} place element. After 1^{st} iteration, it searches again for smallest element in remaining array and swaps in 1^{st} place of array. This process until all are in sorted array. Considering its complexity, it has time complexity $O(n^2)$ in all the cases, as best, average and worst case. Thus, is considered not sufficient for large array and is considered worse than the insertion sort algorithm. The reason for usage of this algorithm is that it does maximum of n swaps and hence is better when swaping is very expensive.

```
Function selection(arr,n)
  For  i=0 to n-2
      Init min=a[i]
      loc=i;
      For j=i+1 to n-1
          If(min>a[j])
              Set min=a[j]
              Set loc=j
       Swap arr[i], arr[loc]
End function
```

ELEMENTS	0	1	2	3	4
DATA	24	16	30	12	2
1ST PASS	24	16	30	12	2
	24	16	30	12	2
	24	16	30	12	2
	24	16	30	12	2
	2	16	30	12	24
2ND PASS	2	16	30	12	24
	2	16	30	12	24
	2	16	30	12	24
	2	12	30	16	24
3RD PASS	2	12	30	16	24
	2	12	30	16	24
	2	12	16	30	24
4TH PASS	2	12	16	30	24
	2	12	16	24	30

Fig 3: Working of Selection Sort

D. MERGE SORT

Merge sort comes under Divide and conquer approach in sorting algorithms. It means it divide an array in two equal parts until all elements are broken down to individual parts and then these are merges up in sorted order. In the last merge it gives the sorted order. The advantage of merge sort is that we can use this for large array elements. Its time complexity for best ,average and worst case is O(n log n). But the disadvantage of this method is that it takes high space.

```
Function mergesort(arr,  low, high)
      If  low < high
          Init mid = (low+high)/2
          Call mergesort(arr,low,mid)
          Call mergesort(arr,mid+1,high)
          Call merge(arr,low,high,mid
End function mergesort
```

```
Function merge(arr, low, high, mid)
   Init i,k = low, j = mid + 1
   While  (i <= mid and j <= high)
      If (arr[i] < arr[j])
         Set  c[k] = a[i]
         Inc k, i
      Else
         Set   c[k] = a[j]
         Inc k, j
    While  (i <= mid)
      Set c[k] = a[i]
      Inc k, i
    While  (j <= high)
       Set c[k] = a[j]
       Inc k, j
   For  i = low to k-1
      Set  a[i] = c[i]
End function merge
```

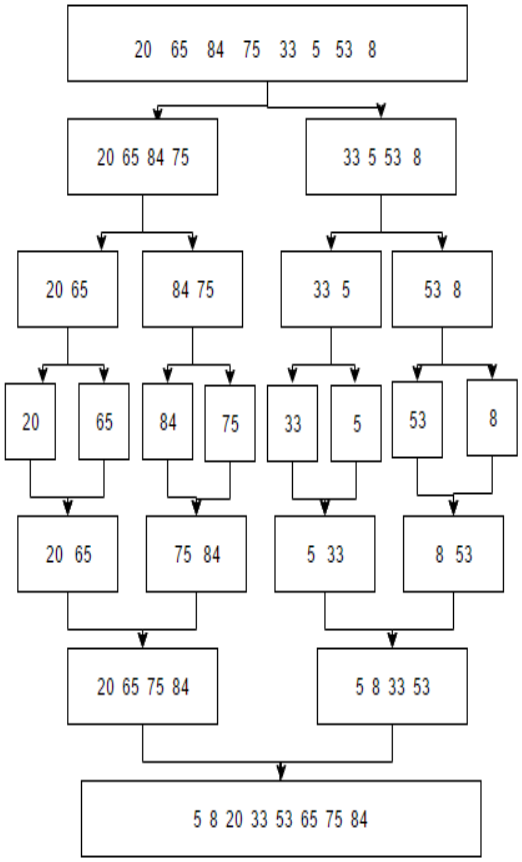

Fig 4: Working of Merge Sort

E. QUICK SORT

Quick sort is another method for to sort algorithms. It works on divide and conquer approach. In this method it partition the array in two parts. Its decides the first element as pivot element and then arrange the array according it pivot. Elements smaller than the pivot comes to left side and larger elements comes to the right side of the pivot elements. Then this is repeated recursively. The time complexity of quick sort in best and average case is O(n log n), but in worst time complexity changes to $O(n^2)$. Quick sort is very efficient algorithm but its implementation is somewhat complex.

```
Function quicksort(A, lo, hi)
    If  lo < hi
        p := Call partition(A, lo, hi)
        Call quicksort(A, lo, p)
        Call quicksort(A, p + 1, hi)
End function quicksort

Function partition(A, lo, hi)
    Set pivot = A[lo], i = lo,  j= hi + 1
    do { do
              i++
          while(A[i]<pivot and i<=hi)

          do
              j--
          while(pivot<A[j])

          if i<j
          swap A[i] , A[j]
       } while(i<j)
   Set a[lo] = a[j]
   Set a[j]=pivot
   Return j
entire array is sorted by calling quicksort(A, 0, length(A))
```

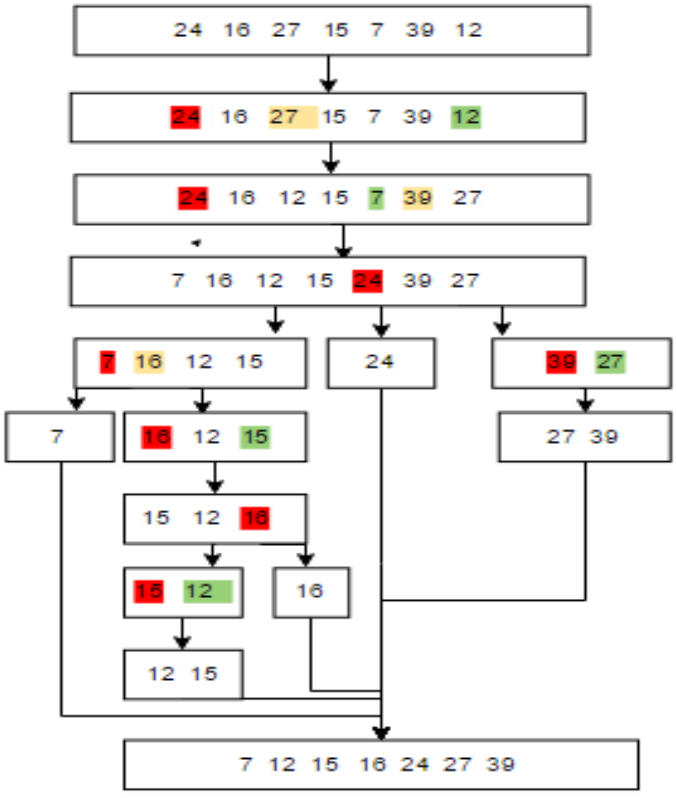

Fig 5: Working of Quick Sort

F. HEAP SORT

Heap sort is a method of sorting array in some order viz ascending or descending order. It is considered more efficient than selection sort. It also works in the same way as selection sort works. It determines the largest or smallest element and then place that element in the last or in beginning as required and this continues until all the elements are sorted. But heap sort uses the data structure heap. Heap is one of the type of binary tree. In this we convert the array into heap, with largest element as the root of the heap. Then root is moved to the last position of the list and the heap is rearranged to place largest element from remaining heap on the top and this go on until heap is empty. The time complexity for heap sort in best case, worst case and average case is $O(n \log(n))$.

```
Function max_heapify(a, i,t n)
  temp = a[i], j = 2*i
  While (j <= n)
     If  (j < n and a[j+1] > a[j])
          j = j+1
     If (temp > a[j])
          end loop
     Else if (temp <= a[j])
          a[j/2] = a[j], j = 2*j
  Set a[j/2] = temp
End function max_heapify

Function heapsort(a, n)
   For  i = n to 2
          Swap a[i],a[1]
          Call max_heapify(a, 1, i - 1)
End function heapsort

Function build_maxheap(a, n)
    For  i = n/2 to  1
          Call max_heapify(a, i, n)
End function build_maxheap
```

Entire array is sorted by calling build_maxheap(a,n), and then by calling heapsort(a, n).

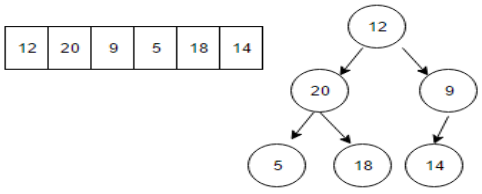

Fig 6: Given list and heap

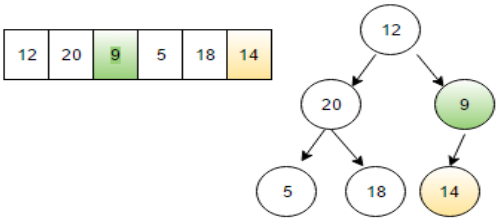

Fig 7 : Compare 9 and 14 and swap

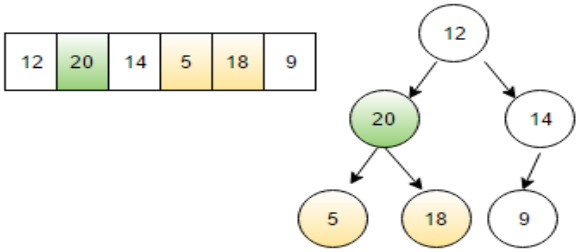

Fig 8: Compare 20,5,18

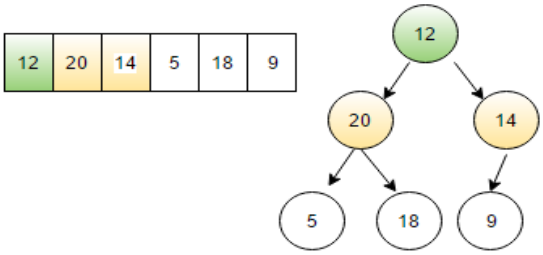

Fig 9: compare 12,20,14 and swap 12 and 20

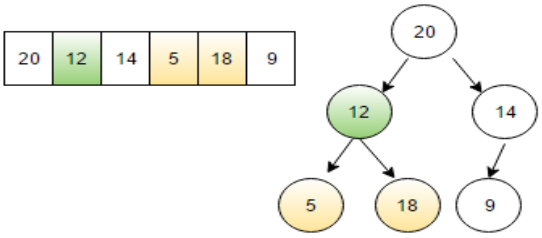

Fig 10: compare 12,5,18 and swap 12 and 18

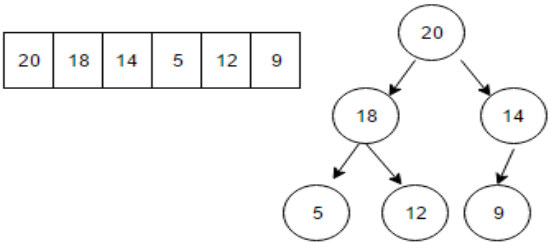

Fig 11: Swap 20 with last element

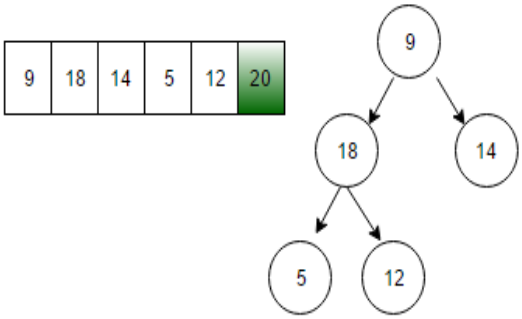

Fig 12 : element 20 is sorted and 20 is removed from the heap.

Now heapify will be called and loop runs again,till whole list is not sorted.

III. COMPARATIVE STUDY AND ANALYSIS

TABLE 1

EXECUTION TIME FOR EACH ALGORITHM(IN NANOSECONDS)

ALGORITHM	TEST CASES	RANDOM	PARTIALLY SORTED	SORTED	REVERSE SORTED	SAME
BUBBLE SORT	10	37613	24644	24170	38672	22157
	50	45773	51800	30107	47878	29193
	100	73643	48300	43416	65372	39044
INSERTION SORT	10	25410	25894	24740	25872	24859
	50	30932	31753	27981	49363	27983
	100	37396	39299	33699	40853	53860
SELECTION SORT	10	24966	25777	25758	24749	23756
	50	36057	31511	29704	44791	30840
	100	43762	41657	38831	65838	38731
MERGE SORT	10	24618	26091	25758	25083	27282
	50	50105	49668	30507	50510	29704
	100	38746	39038	38235	59490	36994
QUICK SORT	10	25153	24598	24485	24778	22002
	50	29524	30813	29736	48037	28716
	100	38639	37358	40089	37494	33047
HEAP SORT	10	38370	26292	30866	24389	25441
	50	47978	48890	49978	39376	33297
	100	60837	61040	64033	59365	55677

TABLE 2

SPACE AND TIME COMPLEXITY

ALGORITHM	**DATA STRUCTURE**	**TIME COMPLEXITY: BEST**	**TIME COMPLEXITY: AVERAGE**	**TIME COMPLEXITY: WORST**	**SPACE COMPLEXITY: WORST**
BUBBLE SORT	ARRAY	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
INSERTION SORT	ARRAY	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
SELECTION SORT	ARRAY	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
MERGE SORT	ARRAY	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	O(n)
QUICK SORT	ARRAY	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	O(n)
HEAP SORT	HEAP	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	O(1)

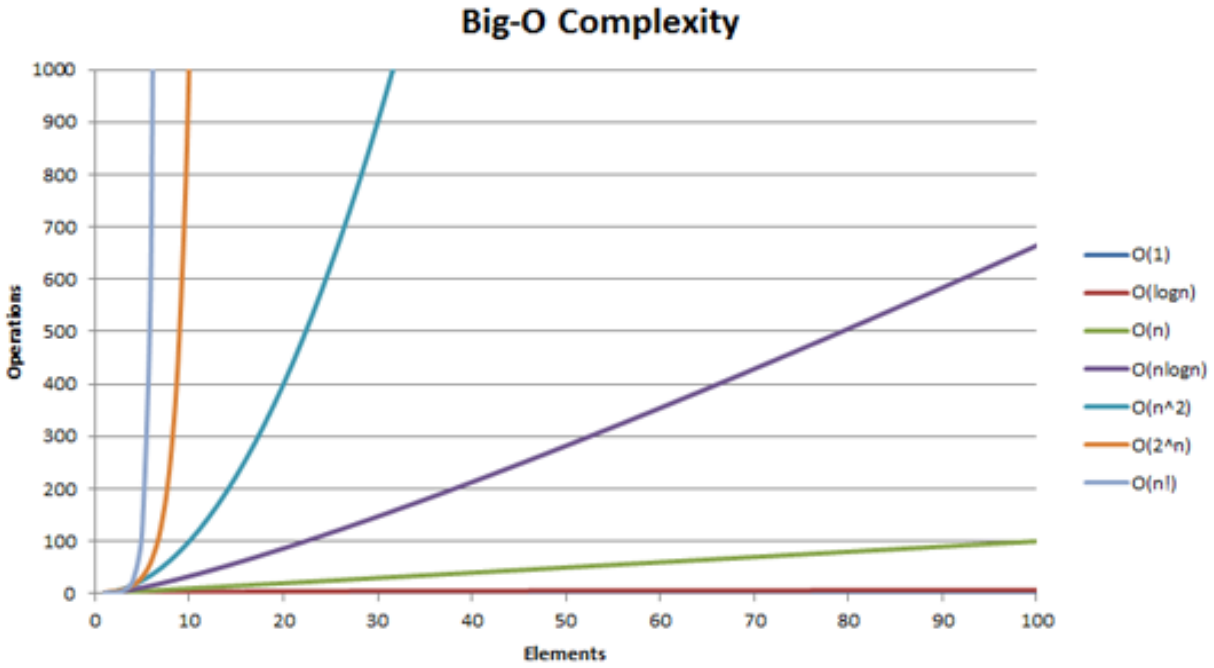

Fig 13 : Complexity Statistics

IV. CONCLUSION

From the table 1 above, it can be said that with increase in size of the array the execution time taken is increasing. Also, it is seen that for small data sets selection sort and insertion sort are effective. When comparing bubble sort with other algorithms it seems to have taken more time than all other. In overall merge sort and quick sort are very effective but are complicated.

V. ACKNOWLEDGEMENT

I would like to thank my Prof. Boominathan P for giving us the required inputs in form of thoughts and speech to understand the subject better and for providing support and material related to the area of this research. Furthermore, I would thank VIT University for being a supporter for the new projects.

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] Let Us C by Yashvant Kanethkar, 8th edition(BPB publications).b-14 Connaught place, New Delhi-110001.

[3] Computer Algorithms by Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Galgotia publications,5 Ansari road, Daryaganj, New Delhi-110002.

[4] Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss

[5] https://www.cs.cmu.edu/~adamchik/15-121/.../Sorting%20Algorithms/sorting.html.

[6] **Sorting algorithm - Wikipedia** https://en.wikipedia.org/wiki/Sorting_algorithm

[7] Debadrita Roy, Arnab Kundu," A Comparative Analysis of Three Different Types of Searching Algorithms in Data Structure" International Journal of Advanced Research in Computer and Communication Engineering) Vol. 3, Issue 5, 2014

[8] ***Data Structures and Algorithms Sorting Techniques - Tutorialspoint*** *https://www.tutorialspoint.com/data_structures algorithms/sorting_algorithms.htm*

1. [9] Ms.NidhiChhajed Assistant Professor C.S.E Dept. PIES,
2. Mr. Imran Uddin Assistant Professor, C.S.E Dept. PIES,
3. Mr.Simarjeet Singh Bhatia Assistant Professor, C.S.E
4. Dept. PIES Indore (M.P), India. A Comparison Based
5. Analysis of Four Different Types of Sorting Algorithms
6. in Data Structures with Their Performances.
7. [10] C.A.R. Hoare, Quicksort, Computer Journal, Vol. 5, 1,
8. 10-15 (1962)
9. [11] Sorting and Searching Algorithms: A Cookbook by Thomas Niemann in Portland, Oregon