International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 3 Issue 5, August 2019 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

Analysis and Comparative of Sorting Algorithms

Htwe Htwe Aung

Lecturer, Faculty of Computer Science, University of Computer Studies, Pathein, Myanmar

How to cite this paper: Htwe Htwe Aung "Analysis and Comparative of Sorting Algorithms" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-3 | Issue-5, August 2019, pp.1049-1053, **https://doi.org/10.31142/ijtsrd26575**

IJTSRD26575

ABSTRACT

There are many popular problems in different practical fields of computer sciences, computer networks, database applications and artificial intelligence. One of these basic operations and problems is the sorting algorithm. Sorting is also a fundamental problem in algorithm analysis and designing point of view. Therefore, many computer scientists have much worked on sorting algorithms. Sorting is a key data structure operation, which makes easy arranging, searching, and finding the information. Sorting of elements is an important task in computation that is used frequently in different processes. For accomplish, the task in a reasonable amount of time-efficient algorithm is needed. Different types of sorting algorithms have been devised for the purpose. Which is the best-suited sorting algorithm can only be decided by comparing the available algorithms in different aspects. In this paper, a comparison is made for different sorting algorithms used in the computation.

KEYWORDS: *Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Time Complexity, Stability*

1. INTRODUCTION

Sorting is a process of rearrangement a list of elements to the correct order since handling the elements in a certain order more efficient than handling randomize elements [15].

Sorting is among the most common programming processes, as an example take database applications if you want to maintain the information and ease of retrieval you must keep information in a sensible order, for example, alphabetical order, ascending or descending order and order according to names, ids, years, departments, etc. Each sorting algorithm uses its own technique in execution. It is also possible that a single problem can be solved by using more than one algorithm. Here I will compare between the sorting algorithms based on best case, average case and worst case efficiency that refer to the performance of the number n of elements.

Information growth rapidly in our world leads to an increase in developing sort algorithms. Developing sort algorithms through improved performance and decreasing complexity, because of any effect of sorting algorithm enhancement of the current algorithms or product new algorithms that reflects to optimize other algorithms. A large number of algorithms developed to improve sorting like merge sort, bubble sort, insertion sort, quick sort, selection sort and others, each of them has a different mechanism to reorder elements which increase the performance and efficiency of the practical applications and reduce the time complexity of each one.

When comparing various sorting algorithms, the several factors must be considered such as time complexity, space complexity and stability. The time complexity of an algorithm determined the amount of time that can be taken by an algorithm to run [3][13][14]. The different sorting algorithm compares to another according to the size of data, inefficient sorting algorithm and speed. The time complexity of an algorithm is generally written in form big O (n) notation, where the O represents the complexity of the algorithm and a value n represent the number of elementary operations performed by the algorithm [19]. For typical sorting algorithms, best behavior is O(n log n) and worst behavior is $O(n^2)$.

Space complexity, an algorithm that used recursive techniques need more copies of sorting data that affect memory space [1]. Some algorithms are either recursive or non-recursive while others may be both (e.g., merge sort). Many previous types of research have been suggested to enhance the sorting algorithm to maintain memory and improve efficiency. Most of these algorithms are used comparative operation between the oldest algorithm and the newest one to prove that. In particular, some sorting algorithms are "in place". This means that they need only O (1) memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.

The stability of the algorithm keeps elements with equal values in the same relative order in the output as they were in the input [20]. Some sorting algorithms are stable by its nature such as bubble sort, insertion sort and merge sort, etc., while some sorting algorithms are not, such as selection sort, quick sort, heap sort, etc. Any given sorting algorithm which is not stable can be modified to be stable [13]. Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).

2. SORTING ALGORITHMS

Sorting algorithms are an important part of managing data. Most sorting algorithms work by comparing the data being sorted. In some cases, it may be desirable to sort a large

volume of data based on only a portion of that data. The piece of data actually used to determine the sorted order is called the key. Sorting algorithms are usually judged by their efficiency [15]. Sorting is the process of arranging data in a specific order which benefits searching and locating the information in an easy and efficient way. Sorting algorithms are developed to arrange data in various ways; for instance, an array of integers may be sorted from lower to highest or from highest to lower or array of string elements may sort in alphabetical order.

This paper describes a comparative study of bubble sort, selection sort, insertion sort, merge sort, quick sort and heap sort. Compares six algorithms of their best case, average case and worst-case time complexity and also discuss their stability. It is a machine-independent analysis, which is a good approach.

Bubble Sort

Bubble sort is a simple sorting algorithm. Let A be a list of N numbers. Sorting A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that

$$A[1]<A[2]<A[3]<\ldots<A[N]$$

The step in the bubble sort algorithm works as follows:

- Compare A[1] and A[2], if A[1] > A[2] then it swaps them.
- It continues doing this for each pair of adjacent elements until to reach the right end.
- After N-1 comparisons and swaps them if needed, then A[N] will contain the largest element.
- It then starts again with the first two elements, repeating until no swaps have occurred on the last pass [1][5][18].

Pseudo-code:

```
func bubblesort(var a as array)
for j from 2 to N
   swaps = 0
   for k from 0 to N-2
      if (a[k] > a[k+1])
         swap (a[k], a[k+1])
         swaps = swaps + 1
      if swaps = 0
      break
end func
```

The sorted array as input or almost all elements are in the proper place, bubble sort has O(n) as the best case performance since it passes over the items one time and $O(n^2)$ as the worst-case performance and average-case performance because it requires at least two passes through the data. 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 increase that is needed to be sorted. Although bubble sort is quite simple and easy to implement it is inefficient in coding reference. It is in place sorting algorithm and it can be implemented as a stable sort.

Selection Sort

Selection sorts the simplest of sorting techniques. The main idea of the selection sort algorithm is given by

- Find the smallest element in the data list.
- Put this element at first position of the list.
- Find the next smallest element in the list.
- Place at the second position of the list and continue until the whole data items are sorted [11].

Pseudo-code:

```
for j ← 1 to n-1
   smallest ← j;
   for k ← j+1 to n
      if (a[k] < a[min])
         smallest ← k;
         Exchange A[k] and A[smallest]
end func
```

Selection sort is work very well for small files, also it's has a quite important application because each item is actually moved at most once [17]. It has the best case and worst case time complexity is $O(n^2)$, making it inefficient on large lists. Selection sort has one advantage over other sort techniques [16][2]. Although it does many comparisons, it does the number of swaps reduced. That means, if input data is small keys but large data area, then selection sorting may be the quickest [19]. Selection sort is in-place sorting algorithm and it can't be implemented as a stable sort.

Insertion Sort

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e. the position to which it belongs in a sorted array. Insertion sort works as below:

- It compares the current element with the largest value in the sorted array.
- If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position.
- This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position of the front [10].

Pseudo-code:

```
for k ← 1 to n-1
   key ← A[k]
   i ← k
   while i > 0 and A[i-1] > key
      A[i] ← A[i-1]
      i ← i - 1
      A[i] ← key
end func
```

The comparisons and copies of this algorithm required: on the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of N-1 comparisons on the last pass. This is

$$1 + 2 + 3 + \ldots + N-1 = N*(N-1)/2$$

However, because on each pass an average of only half of the maximum numbers of items are actually compared before the insertion point is found, can divide by 2, which gives

$$N*(N-1)/2$$

The number of copies is approximately the same as the number of comparisons. However, a copy isn't as time-consuming as a swap, so for random data, this algorithm runs faster than bubble sort and selection sort. The insertion sort runs in $O(n^2)$ time for random data as the worst-case and average-case complexity. Best case complexity of O(n)

while the array is already sorted. It is much less efficient on large lists than more advanced algorithms such as quicksort and merges sort. However, insertion sort provides several advantages simple implementation and efficient for small data sets [6][4]. It is in place sorting algorithm and stable sort.

Merge Sort

Merge sort is based on divide and conquer strategy technique which is a popular problem-solving technique. The merge sort algorithm is work as under:

- Split array A(x1, x2, x3, …, xn) from middle into two parts of length n/2 (x1, x2, x3, …, xn/2 and x(n/2)+1, …, xn).
- Sorts each part calling Sort algorithm recursively.
- Merge the two sorting parts into a single sorted list [21].

Pseudo-code:

```
MERGE-SORT(A, left, right)
  if left < right
    mid = (l+(r-l)/2)
    MERGE-SORT(A, left, mid)
    MERGE-SORT (A, mid+1, right)
    MERGE(A, left, mid, right)
end func
MERGE(A, l, h, ub)
  j ← 0
  lb ← l
  mid ← h-1
  n ← ub-lb+1
  while (l <= mid && h <= ub)
    if(theArray[l] < theArray[h])
      A[j++] ← theArray[l++]
    else
      A[j++] ← theArray[h++]
  while(l <= mid)
    A[j++] ← theArray[l++]
  while(h <= ub)
    A[j++] ← theArray[h++]
  for(j=0; j<n; j++)
    theArray[lb+j]← A[j]
end func
```

Merge sort can be easily applied to lists and arrays because it needs sequential access rather than random access. It can handle very large lists due to its worst case, best case and average case running time are O(n log n). The O(n) additional space complexity and involvement of huge amount of copies in simple implementation made it a little inefficient. It is stable sort, parallelizes better and is more efficient at handling slow-to-access sequential media but not in place. Merge sort is often the best choice for sorting a linked list [7][12].

Quick Sort

Quicksort 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. All elements smaller than the pivots are moved before it and all greater elements are moved after it. The lesser and greater sub-lists are then recursively sorted. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex but are among the fastest sorting algorithms in practice [8].

Pseudo-code:

```
QUICKSORT( array A, int j, int k )
  if(k > j)
    then i← a random index from [j..k]
    swap A[i] and A[j]
    p ← PARTITION(A, j, k)
    QUICKSORT(A, j, p − 1)
    QUICKSORT(A, p + 1, k)
end func
```

The partition algorithm works as follows

- A[j] = x is the pivot value.
- A [j…p - 1] contains elements less than x.
- A [p + 1…r - 1] contains the elements which are greater than or equal to x.
- A[r...k] contains elements which are currently unknown.

```
PARTITION( array A, int j, int k)
  x ← A[j]
  p ← j
  for r ← j + 1 to k do
    if (A[r] < x) then p ← p + 1
      swap A[p] and A[r]
      swap A[j] and A[p]
      return p
end func
```

Quicksort is one of the fastest sorting algorithms which is the part of many sorting libraries. The running time of Quick Sort depends upon heavily on choosing the pivot element. Since the selection of pivot element is randomly, therefore average case and best-case running time is O(n log n). However, worst-case running time is $O(n^2)$ but it happens rarely. Quicksort is not stable but is an in-place [20].

Heap Sort

Heapsort is a comparison-based sorting algorithm. It is the most efficient version of selection sort. It divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a heap data structure rather than a linear-time search to find the maximum [22]. Heaps can be used in sorting an array. In max-heaps, the maximum element will always be at the root. Heapsort uses this property of heap to sort the array.

Consider an array A which is to be sorted using Heap sort.

- Initially build a max heap of elements in array A.
- The root element, that is A[1], will contain a maximum element of A. After that, swap this element with the last element of array A and Heapify the max heap excluding the last element which is already in its correct position and then decrease the length of the heap by one.
- Repeat the above step, until all the elements are in their correct position.

Pseudo-code:

```
Heapsort(A)
  BuildHeap(A)
  for i ← length(A) step -1 until 2
    interchange A[1] and A[i]
  Heapify(A, 1)
end func
BuildHeap(A)
```

```
  heapsize ← length(A)
  for i ← floor( length/2 ) step -1 until 1
  Heapify(A, i)
end func

Heapify(A, i)
  l ← left(i)
  r ← right(i)
  if (l <= heapsize) and (A[l] > A[i])
    largest ← l
  else
    largest ← i
  if (r <= heapsize) and (A[r] > A[largest])
    largest ← r
  if (largest != i) {
    interchange A[i] and A[largest]
  Heapify(A, largest)
end func
```

Heap sort has O(n log n) time complexities for all the cases (best case, average case and worst case). Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime. Heapsort is an in-place algorithm, but it is not a stable sort. [22].

3. COMPARATIVE STUDY AND DISCUSSION

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

O(n^2):

- Bubble Sort
- Selection Sort
- Insertion Sort
- O(n log n)
- Merge Sort
- Quick Sort
- Heap Sort

Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is abysmal, while the insertion sort and selection sorts also have complexities; they are significantly more efficient than bubble sort.

Heapsort is the slowest of the sorting algorithms but unlike merge and quicksort, it does not require massive recursion or multiple arrays to work. The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array. The quicksort is massively recursive sort. It can be said as the faster version of the merge sort.

In the following figures is the efficiency of different algorithms according to the above-stated criteria.

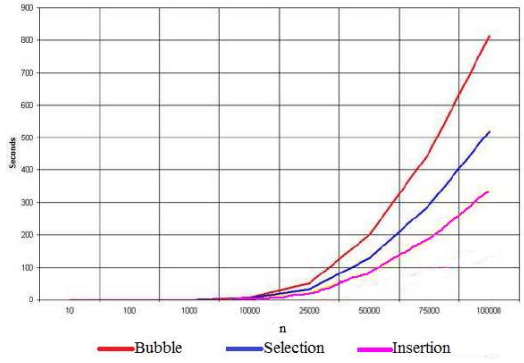

Figure.1 Efficiency for O(n^2) Sorts [9]

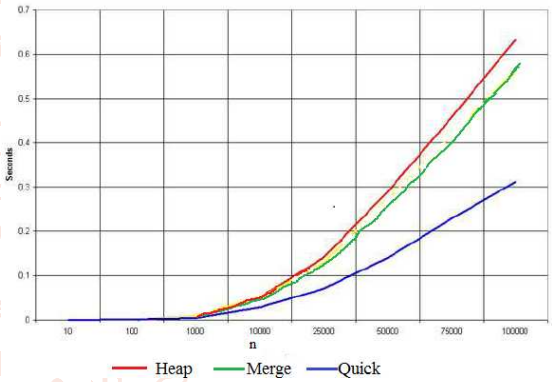

Figure.2 Efficiency for O(n log n) Sorts[9]

This table gives the comparison of time complexity or running time of different sorting algorithms in a short and precise manner given as under.

Table 1: Comparison of sorting algorithms

Sort	Time			Stable	In place
	Avg	Best	Worst		
Bubble sort	O(n^2)	O(n)	O(n^2)	Yes	Yes
Selection sort	O(n^2)	O(n^2)	O(n^2)	No	Yes
Insertion sort	O(n^2)	O(n)	O(n^2)	Yes	Yes
Merge sort	O(n logn)	O(n logn)	O(n logn)	Yes	No
Quick sort	O(n logn)	O(n logn)	O(n^2)	No	Yes
Heap sort	O(n logn)	O(n logn)	O(n logn)	No	Yes

4. CONCLUSIONS

This paper discusses well-known sorting algorithms, their pseudo-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 tried to analyze as an abstract point of view. This paper describes six well-known sorting algorithms and their running time which is given in the above table. To determine the good sorting algorithm, the time complexity is the main consideration but other factors include handling various data type, consistency of performance and complexity of code, etc. From the above discussion, every sorting algorithm has some advantages and disadvantages and the programmer must choose according to his or her requirement of sorting algorithms.

References

[1] Amity Dev Mistral & Deepak Garg. (2008, DEC). "Selection of Best Sorting Algorithm", International Journal of intelligent information Processing, pp.363-368.

[2] A.Levitin, "Introduction to the Design & Analysis of Algorithms", Addison–Wesley Longman, 2007, pp.98-100.

[3] C.Cook, D.Kim. "Best sorting algorithm for nearly sorted lists". Commun. ACM, 23(11), pp.620-624.

[4] http://corewar.co.uk/assembly/insertion.htm

[5] https:// en.wikipedia.org/wiki/Seymour_Lipschutz

[6] http://en.wikipedia.org/wiki/Insertion_sort

[7] http://en.wikipedia.org/wiki/Merge_sort

[8] https://en.wikipedia.org/wiki/quick_sort

[9] http://linux.wku.edu/~lamonml/algor/sort/sort.html

[10] http://www.hackerearth.com/sorting

[11] Kazim Ali. (2017 FEB). "A Comparative Study of Well-Known Sorting Algorithms", International Journal of Advanced Research in Computer Science, pp.5-6.

[12] Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field", Soviet Mathematics - Doklady (10), pp.744.

[13] M. Goodrich and R. Tamassia, "Data Structures and Algorithms in Java", Johnwiley& sons 4th edition, 2010, pp.241-243.

[14] M. Sipser, "Introduction to the Theory of Computation", Thomson, 1996, pp.177-190.

[15] P. Adhikari, Review on Sorting Algorithms, "A comparative study on two sorting algorithms", Mississippi state university, 2007.

[16] R. Sedgewick, "Algorithms in C++", Addison-Wesley Longman, 1998, pp.273-274.

[17] R. Sedgewick and K. Wayne, "Algorithms", Pearson Education, 4th Edition, 2011, pp.248-249.

[18] SCHAUM LIPSCHUTZ. "Data Structures", pp.73-74.

[19] S. Jadoon, S. Solehria, S. Rehman and H. Jan.(2011, FEB). "Design and Analysis of Optimized Selection Sort Algorithm", pp.16-21.

[20] 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.

[21] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", 1976, pp.66

[22] Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM, 7(6), pp.347-348.