Research Article

SA Sorting: A Novel Sorting Technique for Large-Scale Data

Mohammad Shabaz and Ashok Kumar

1Apex Institute of Technology, Chandigarh University, S.A.S. Nagar, India
2Chitkara University Institute of Engineering and Technology, Chitkara University, Rajpura, India

Correspondence should be addressed to Mohammad Shabaz; bhatsab4@gmail.com

Received 19 August 2018; Accepted 6 December 2018; Published 23 January 2019

Academic Editor: Youyun Xu

Copyright © 2019 Mohammad Shabaz and Ashok Kumar. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Sorting is one of the operations on data structures used in a special situation. Sorting is defined as an arrangement of data or records in a particular logical order. A number of algorithms are developed for sorting the data. The reason behind developing these algorithms is to optimize the efficiency and complexity. The work on creating new sorting approaches is still going on. With the rise in the generation of big data, the concept of big number comes into existence. To sort thousands of records either sorted or unsorted, traditional sorting approaches can be used. In those cases, we can ignore the complexities as very minute difference exists in their execution time. But in case the data are very large, where execution time or processed time of billion or trillion of records is very large, we cannot ignore the complexity at this situation; therefore, an optimized sorting approach is required. Thus, SA sorting is one of the approaches developed to check sorted big numbers as it works better on sorted numbers than quick sort and many others. It can also be used to sort unsorted records as well.

1. Introduction

Sorting big numbers in an optimized way is a challenging task to perform. A number of such sorting algorithms exist which are optimized, but their execution time is still to be optimized. Sometimes, these algorithms take the same amount of time to sort a sorted record as they take to sort an unsorted record. The sorting algorithm should be stable, effective, less complex, and efficient. SA sorting follows most of these parameters. SA sorting is introduced as a new approach to operate on both sorted and unsorted lists or records and shows better execution time on the sorted list.

The following section discusses the existing sorting approaches.

2. Sorting Techniques

2.1. Bubble Sort. It is a stable sorting algorithm in which each element in the list is compared with its next adjacent element, and the process is repeated until the elements are sorted. If we have $n$ elements, then there are $(n-1)$ number of passes and $n(n-1)/2$ number of iterations in totality. Mathematically,

$$F(n) = \frac{n(n-1)}{2} = \frac{(n^2 - n)}{2}$$

and thus,

$$F(n) \leq n^2.$$  

The algorithm for bubble sort is given in Algorithm 1.

In this algorithm, if no swap occurs, it will break the loop and directly go to end, and thus, only one loop executes that determines best-case analysis which becomes $O(n)$. On the contrary, the complexity of average and worst cases would be $O(n^2$). Bubble sort is highly code inefficient, and it is one of the worst sorting approaches, so professionals do not use it.

2.2. Insertion Sort. It is the stable sorting algorithm in which starting from the first number from the record, this first number is compared to every adjacent number to find its sorted position. When the position is found, the number is
inserted there. The algorithm for insertion sort is given in Algorithm 2.

Insertion sort holds good for smaller datasets since it is also code inefficient for large data/lists or big numbers. Insertion sort is twice faster than bubble sort. For the best case, it is \(O(n)\), while for the average or worst case, it is \(O(n^2)\).

2.3. Selection Sort. By nature, selection sort is unstable, but it can be improved to become stable. In this sorting technique, we are supposed to find the smallest number from the record, put this number at the starting position, change the position to next, and then again find the smallest number from the remaining list. This process goes on and on, until the whole list becomes sorted. This algorithm is efficient for smaller records, but for larger records, this technique is again code inefficient. The algorithm is given in Algorithm 3.

Its execution time is better for smaller data/records (up to hundreds of records). The best-, average-, or worst-case complexity is the same as \(O(n^2)\).

2.4. Merge Sort. It is a stable sorting algorithm and is very efficient to handle big numbers. It is based on the following three steps [1]:

1. Divide the given list or record into number of sublists in such a way that every list or sublist is half divided
2. Conquer the sublist by the recursion method
3. Combine the sorted sublist simply by merging

The sorting is actually done in the second step. Merge sort is the only sorting technique which is purely based on the divide-and-conquer technique of solving the algorithm. It requires double the memory as required by other sorting techniques. The algorithm for merge sort is given in Algorithms 4 and 5.

From the algorithm, it is analyzed that merge sort has the following recurrence relation:

\[
T(n) = 2T\left(\frac{n}{2}\right) + cn.
\]  

Using the master method, \(F(n) = cn, a = 2,\) and \(b = 2,\) and thus,

\[
n^{\log_b a} = n^{\log_2 2} = n^1 = n.
\]  

Now,

\[
T(n) = O(n \log n).
\]  

For all three cases, it would be \(O(n \log n).\)

2.5. Quick Sort. It is unstable but efficient and is one of the fast working sorting algorithms. It is based on the divide-and-conquer strategy. While talking about quick sort at an instant, in our mind, there comes the concept of pivot element. The pivot element is one of the randomly chosen members from the list which is under the operation of
2.6. Tree Sort. It is an unstable sorting algorithm built on the binary search tree (BST). The element in tree sort is sorted using in-order traversing operation. Tree sort requires extra memory space, and complexity changes from balanced BST to unbalanced BST. The complexity of the tree sort is \( O(m^2) \) for the worst case and \( O(n \log m) \) for average and best cases.

2.7. Gnome Sort. It is a stable sorting approach. When we think that the list or record is sorted but we are not sure, we need an algorithm which works best on the sorted list; for this purpose, we use gnome sort. It performs well not only on the sorted list but also on the unsorted list. The algorithm is given in Algorithm 8.

From the algorithms, it is clearly seen that if the list is sorted, no interchange of elements is done; hence, it executes linearly. Thus, for the best case, it is \( O(n) \), and for average and worst cases, it is \( O(n^2) \).

2.8. Counting Sort. It is a stable and easily understandable sorting algorithm. As the name depicts, counting sort works by finding the largest element in the given list/record, and then starting from the least element/number, its frequency is counted and at last the sorted list is produced based on the order of its occurrence while sorting. It is useful in those cases where the difference between the numbers is very small and the dataset is also very small. The step-by-step procedure of counting sort is discussed in Algorithm 9.

2.9. Grouping Comparison Sort (GCS). Suleiman with his team of three other members proposed the GCS algorithm. The methodology they have used is to divide the given list/record into groups. Each of these groups contains three elements, and comparison is done in such a way that every element of one group is compared to that in other groups. The main drawback of this algorithm is that the input size must be less than or equal to 25000 records to get the better results. The complexity becomes \( O(n^2) \) for all the three cases.

2.10. Heap Sort. Heap sort is a stable but efficient sorting algorithm which is based on the complete binary tree and follows the heap order. Heap sort contains min heap, in which the root node is having the minimum value, and max heap, in which the root node is having the maximum value.
The procedure of heap sort is explained through Algorithms 10 and 11.

For heap sort, in all the three cases (i.e., best, average, and worst cases), the complexity is the same as $O(n \log n)$, where $n$ is the number of records in the list.

### 2.11. Radix Sort

It is a stable and efficient sorting algorithm when the size of list/record is small. Internally, it acts like counting sort. One of the drawbacks of radix sort is it operates on one number many times as it has the number of significant bits on a digit. Suppose if there is a number 169, the radix sort operates on it three times, sorting from the least significant digit 9 to the most significant digit 1 in the list. Radix sort compares the LSB of all the numbers in a similar way to proceed with further results in a sorting list. The procedure of radix sort is explained in Algorithm 12.

If the size is the longest length $m$, then there are $a$ elements and the complexity would be $O(m \times n)$. If the size varies to a constant number, then the size is ignored and the complexity becomes $O(n)$ for all the three cases.

### 2.12. Cocktail Sort

It is a stable and efficient sorting algorithm as compared to bubble sort as it is the extended version of bubble sort. Cocktail sort works on both sides of the list. During sorting, it puts the largest element to the tail side and smallest element to the head side. The head side and tail side are shown in Figure 1.

Bubble sort puts the biggest element to the tail side after every pass, while cocktail sort puts the smallest element to the head side and the biggest element to the tail side after every pass. The complexity of cocktail sort for worst and average cases is $O(n^2)$, but for the best case, it is $O(n)$.

### 2.13. Comb Sort

It is the stable and another improved version of bubble sort as it changes the gap size from 1 to 1.3 for every iteration. Gap size tells the algorithm to swap. As the gap size increases, the number of swaps decreases; thus, on the average-case scenario, comb sort performs better. But
for the worst case, it remains the same as $O(n^2)$. For the best case, it is $O(n)$ as no swap is done.

2.14. Enhanced Selection Sort. It is the extended version of selection sort which is made stable by decreasing the regular size of the list. This is done in such a way that first the biggest element is found, a swap is made, the size of the list is decreased by 1, and then the sort is again performed in a similar way to get the list sorted. Although the complexity would be the same as that of selection sort but in the best-case scenario, the number of swaps would be zero for enhanced selection sort. The complexity for all the three cases would be $O(n^2)$.

2.15. Shell Sort. It is the unstable and efficient sorting algorithm which is the extended version of insertion sort. Shell sort works well if the given list is partially sorted. This means we are discussing about the average-case scenario. Shell sort uses Knuth’s formula to calculate the interval or spacing. This formula is given as follows:

$$x = x \times 3 + 1, \quad (6)$$

where $x$ has the starting value 1 and is called the interval/spacing. Shell sort divides the given list into sublists by using an increment which is called the gap, compares the elements, and then interchanges them depending upon the order either increasing or decreasing. The algorithm for shell sort is given in Algorithm 13.

For best and worst cases, the complexity is $O(n \log n)$ and $O(n^2)$, respectively. For the average case, it depends upon the gap.

2.16. Bucket Sort. It is a stable sort and consists of bucket. The elements are inserted in the bucket, and then, the sorting is employed on the bucket. Bucket sort does not have comparisons and uses the index to sort. These indexes are not just obtained from any mathematical function but are obtained in such a way that they could satisfy the order of arrangement of numbers inserted in the bucket. The procedure is explained in Algorithm 14.

The complexity for bucket sort would be $O(n)$ for all the three cases.

2.17. Tim Sort. It is a combination of insertion and merge sort. It is a stable and efficient sorting technique in which the list or record is split into blocks called “run.” If the size of list or record is less than run, then it can be sorted using insertion sort. The maximum size of the run would be 64 depending upon the list or record. But if the size of the unsorted list is very large, then both insertion and merge sorting techniques are used. The complexity for best, average, and worst cases is $O(n \log n)$.

2.18. Even-Odd/Brick Sort. It is another extension of bubble sort, in which the BS algorithm is partitioned into two parts: even part and odd part. Even part consists of even numbers, and odd part consists of odd numbers. Both the parts are executed one by one, and at last, the combined result is obtained and the records are sorted. It is a stable algorithm with the same complexity as the bubble sort:

$$\text{best case} \rightarrow O(n),$$

$$\text{worst case} \rightarrow O(n^2),$$

$$\text{average case} \rightarrow O(n^2). \quad (7)$$

2.19. Bitonic Sort. It is introduced through the concept of merge sort. In bitonic sort, we move the list to level $L − 1$ with two parts: left part and right part; the left part is settled in an increasing order, and the right part is settled in a decreasing order [2]. These parts are merged, moved to level $L$, and then sorted to form the sorted sequence. The complexity for bitonic sort is $O(n^2)$ on best, worst, and average cases.
3. Literature Review and Related Work

Ali [3] discussed the number of sorting algorithms in this paper. Evaluation of time complexity, stability, and in-place nature is done. Ali found their running time in the virtual and real environment and suggested where to use a particular algorithm so that the result obtained is efficient, and Ali concluded that quick sort is a better option for sorting in the average-case scenario; counting, bucket, and radix sorts are efficient for smaller size of the list/record and on integer-type data.

Hammad [4] compared three sorting algorithms, namely, bubble, selection, and gnome sorts, based on their average running time in this paper. Hammad took the number of readings to find the running time and concluded that whenever the record list is sorted, gnome sort appears as the fastest sorting algorithm, but when the list or record is unsorted, gnome sort took the same running time as bubble sort or selection sort has in their worst or average case (i.e. \(O(n^2)\)).

Elkahlout and Maghari [5] discussed the two advanced versions of bubble sort, namely, comb and cocktail sort, and one linear-time technique, namely, counting sort, in this paper. Comparing these techniques, they concluded that cocktail sort performs better on average evaluation of their process time.

All these algorithms are graphically implemented in this paper. The main focus is on time complexity.

Jehad and Rami [6] made some changes in the bubble sort and selection sort so as to reduce the number of swaps during sorting operation. In this paper, the author follows the same procedure, compares the enhanced version of bubble sort and selection sort with the original bubble sort and selection sort, and then reduces the execution time. The complexity of enhanced bubble sort is reduced from \(O(n^2)\) to \(O(n \log n)\) and remains the same as that of selection sort.

Pankaj [7] using the C programming language compared five sorting techniques, namely, bubble, selection, quick, merge, and insertion sort, on the basis of average running time. The execution time is calculated in microseconds, and he concluded that quick sort is a better option for sorting the number of elements from 10 to 10000; the paper graphically represents the average running time of each algorithm.

Khalid et al. [8] proposed a new algorithm, namely, grouping comparison sort, which is then compared with the traditional sorting technique. This proposed algorithm has a limitation of having an input size of 25000 elements. As the input size increases, the results become horrible. All these above-discussed papers use the traditional sorting algorithms and compare them to get the average running time and conclude which of them is better to use.

4. SA Sorting

Starting from the left extreme end of the list/record, the first element is obtained as the target. The target is compared with all the other elements until the smaller element is found. The target is swapped with the smaller element and continuously compared till the extreme right end of the list. Again, going back to the target position, a new target element is taken at that position. Similarly, it is processed in the same way. The position is not changed until the targeted element is found as is already operated. When the targeted element is found to be already operated, the position is changed to next by 1. In this way, SA sorting works. The step-by-step process of SA sorting is given in Algorithm 15.

The number of comparisons, \(C\), for SA sorting is given as follows:

\[
C = \left\lfloor \frac{n(n-1)}{2} \right\rfloor + S, \quad \text{(8)}
\]

where \(S\) is the number of swaps. In the best case, \(S = 0\), so

\[
C = \left\lfloor \frac{n(n-1)}{2} \right\rfloor, \quad \text{(9)}
\]

or \(\sum_{i=1}^{n} i - 1\).

Let \(T(n) = \left\lfloor \frac{n(n-1)}{2} \right\rfloor\), \(T(1) = 0\), \(T(2) = 1\), and \(T(3) = 3\). Now creating the recurrence relation,

\[
T(n) = 9T\left(\frac{n}{3}\right) + n. \quad \text{(10)}
\]

Using induction to prove,

At \(n = 3\), \(T(3) = 9T(1) + 3 = 9(1) + 3 = 3\)

At \(n = 6\), \(T(6) = 9T(2) + 3 = 9(2) + 3 = 21\)

At \(n = 9\), \(T(9) = 9T(3) + 3 = 9(3) + 3 = 36\), and so on.

Solving using the master method, 
\(a = 9\), \(b = 3\), \(f(n) = n\), and \(n^{log_b a} = n^{log_3 9} = n^2 > n\); thus, \(T(n) = O(n^2)\)
However, $f(n)$ is a polynomial smaller than $n^{log_a^n}$. SA sorting can be optimized in the future, but in comparison with the optimized quick sort and merge sort, it performs better on the already sorted list. For worst and average cases, complexity is $O(n^2) + S$. Since $S$ is included in $C$, it can be neglected; thus, for all the three cases, it is $O(n^2)$.

### 5. Results

The proposed sorting technique is implemented in C++ and tested with different numbers of elements. The performance of the SA sorting is measured in terms of execution time and memory required for sorting. The comparison of execution time and memory used by existing sorting techniques with SA sorting is shown in Tables 1 and 2. As moving from lower dataset to higher with sorted nature of the dataset, we found that SA sorting improves and performs better. Discussing about memory requirement from lower dataset to higher, only slight change can be seen.

### 6. Conclusion

While implementing all these sorting techniques and comparing them with SA sorting, the following points are concluded:

1. If we increase the space, the time reduces as shell sort and heap sort do
2. The sorting techniques which work well on unsorted records are not very good on sorted records as quick sort and merge sort do
3. In the worst-case scenario, most of the sorting techniques rely on $O(n^2)$ as SA sorting does
4. No sorting technique is universally used, and its usage depends upon their nature and users requirement
5. SA sorting needs to be improved and optimized in the future

### Data Availability

Our article is purely based on algorithm design. We evolve our results from unsorted and sorted data files which include different records, in order to compare the algorithm with already established algorithms. Thus, no such proper data have been used.

### Conflicts of Interest

The authors declare that they have no conflicts of interest.

### References

[1] T. H. Cormen, Introduction to Algorithms, MIT press, Cambridge, MA, USA, 2009.
[2] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Cambridge, MA, USA, 1978.
[3] K. Ali, "A comparative study of well known sorting algorithms," International Journal, vol. 8, no. 1, 2017.
[4] J. Hammad, "A comparative study between various sorting algorithms," International Journal of Computer Science and Network Security (IJCSNS), vol. 15, no. 3, p. 11, 2015.
[5] A. H. Elkahlout and A. Y. A. Maghari, A comparative study of sorting algorithms comb, cocktail and counting sorting, 2017.
[6] A. Jehad and M. Rami, "An enhancement of major sorting algorithms," International Arab Journal of Information Technology, vol. 7, no. 1, 2010.
[7] P. Sareen, "Comparison of sorting algorithms (on the basis of average case)," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no. 3, pp. 522–532, 2013.
[8] K. S. Al-Kharabsheh, I. M. AlTurani, A. M. I. AlTurani, and N. I. Zanoon, "Review on sorting algorithms a comparative study," International Journal of Computer Science and Security (IJCSS), vol. 7, no. 3, pp. 120–126, 2013.

### Table 1: Comparison of execution time in microseconds.

| S. no. | No. of entries | Nature of entry | Quick | Merge | Heap | Insertion | Selection | Shell | SA |
|-------|----------------|----------------|-------|-------|------|-----------|----------|-------|----|
| 1     | 100            | Unsorted       | 67    | 75    | 44   | 65        | 76       | 33    | 109|
| 2     | 500            | Unsorted       | 150   | 165   | 185  | 207       | 495      | 114   | 1078|
| 3     | 1k             | Unsorted       | 232   | 271   | 231  | 553       | 1722     | 373   | 4223|
| 4     | 5k             | Unsorted       | 1207  | 1389  | 2141 | 19335     | 39665    | 1452  | 124973|
| 5     | 10k            | Unsorted       | 2359  | 2826  | 4588 | 67283     | 158382   | 3100  | 506885|
| 6     | 20k            | Unsorted       | 4932  | 5934  | 9747 | 268644    | 627031   | 6710  | 2054339|
| 7     | 1k             | Sorted         | 3492  | 265   | 377  | 144       | 1736     | 156   | 3146|

### Table 2: Comparison of memory used in MB.

| S. no. | No. of entries | Nature of entry | Quick | Merge | Heap | Insertion | Selection | Shell | SA |
|-------|----------------|----------------|-------|-------|------|-----------|----------|-------|----|
| 1     | 1k             | Unsorted       | 1.503 | 1.503 | 3.110| 1.502     | 1.514    | 3.109 | 1.350|
| 2     | 5k             | Unsorted       | 1.561 | 1.570 | 3.242| 1.620     | 1.520    | 3.242 | 1.611|
| 3     | 10k            | Unsorted       | 1.630 | 1.640 | 3.300| 1.600     | 1.678    | 3.343 | 1.615|
| 4     | 20k            | Unsorted       | 1.800 | 1.795 | 3.385| 1.622     | 1.771    | 3.424 | 1.786|
| 5     | 1k             | Sorted         | 1.470 | 1.491 | 3.230| 1.420     | 1.510    | 3.105 | 1.413|
