2018 3rd International Conference for Convergence in Technology (I2CT)
The Gateway Hotel, XION Complex, Wakad Road, Pune, India. Apr 06-08, 2018

Super Sort Sorting Algorithm

Yash Gugale
Department of Computer Engineering
P.E.S. Modern College of Engineering
Pune, India
yashgugale@gmail.com

***Abstract*—sorting algorithms are a major area of research in computer science and engineering where the simple task of sorting the numbers leads to complex algorithms. In recent years, many researchers have proposed several sorting techniques to enhance time and space complexities of the algorithms. The sorting technique presented in this paper makes use of the natural sequence of sorted elements in an array of random numbers so as to reduce the number of steps needed to sort. The complexity of the proposed algorithm is O(nlogn) where n represents the number of elements in the input array.**

***Index Terms*—sorting algorithm; super sort; time complexity; space complexity**

I. Introduction

When given a list of elements in a random order, we want to rearrange them in ascending or descending order to perform useful operations. The sorting techniques can be divided into two categories, comparison based techniques and non-comparison based techniques. Comparison based sorting techniques include bubble sort, insertion sort, selection sort, merge sort, quick sort, etc. [1]. Non-comparison based sorting techniques include bucket sort, radix sort, etc. [2]. The current sorting techniques have certain disadvantages [3]. Sorting algorithms such as bubble sort, selection sort and insertion sort have quadratic time complexities for worst and average cases. Merge sort is faster, but it is not an in-place sort and hence requires a lot of memory. Quick sort has worst case time complexity of $O(n^2)$ for a poor pivot selection or when it tries to sort an already sorted list [4].

The super sort sorting algorithm proposed in this paper is based on the principle of selecting the sequence of already sorted elements in a given unsorted list. These natural sequences are then removed from the original list to form a sublist. The original list is then traced backwards and the same selection is performed. At the end of the forward and backward passes, two sorted lists are obtained, which are merged to form one intermediate list. The algorithm then partitions the original unsorted list (containing the remaining elements after the removal of the two sublists) from the middle, and recursively applies the same sort on each left and right sublist. Each of these recursive calls will then return two more lists, which will be merged to form the second intermediate list. Finally, these two lists will be merged again and will be returned as each recursive call returns till we get one final sorted list. The algorithm uses divide and conquer approach to efficiently sort the elements.

The paper is organized as follows: Section II of the paper gives the related work done in the field of sorting algorithms. Proposed algorithm is presented in section III. Section IV explains the research methodology along with an example. The analysis of the algorithm is given in section V. Comparison with other sorting algorithms is given in section VI. Section VII presents information on the future work to be done to improve the efficiency of the algorithm.

II. Related Work

Research in sorting algorithms began in the early 1950's and is going on even today. With the advances in computing hardware, parallelism came into picture and parallel techniques were developed. However, investing in ideas is more important than investing in hardware. The performance of sorting algorithms can be measured by using basic criteria such as time and space complexities and stability [5][6][7].

The following table (TABLE I) shows the comparison of existing sorting techniques based on these criteria.

TABLE I. Comparison of existing sorting techniques

Sorting algorithm	Time Complexity			Space complexity	Stability
	Average case	*Best case*	*Worst case*		
Bubble Sort [3]	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes
Insertion Sort [3]	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes
Selection Sort [3]	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Merge Sort [3]	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Yes
Quick Sort [3]	O(nlogn)	O(nlogn)	$O(n^2)$	O(nlogn)	No

III. Super Sort Algorithm

The proposed sorting algorithm can sort the elements in ascending or descending order. It goes through the following four steps during each call to the sorting function. The deepest recursive call returns when the boundary conditions are met. At the end of the final execution, we get a completely sorted list. The steps are as follows:

978-1-5386-4273-3/18/$31.00 ©2018 IEEE

A. Forward Selection:

It selects the first element in the input list i.e. element at index 0 as the initial largest element called 'current_highest'. It removes that element from the unsorted list and appends it to an empty 'forward_sorted' list, which contains all the sorted elements in the forward selection pass. Then it compares this 'current_highest' with the next element in the unsorted list. If it is less than the 'current_highest', the next element is checked. If the element is greater than or equal to the 'current_highest', this element is removed from the unsorted list and added to the 'forward_sorted' list. This newly removed element then becomes the 'current_highest' and is used for comparison with the next element in the unsorted list. This selection is done till the end of the unsorted list is reached. Finally, a 'forward_sorted' sublist is obtained which contains elements in ascending order and an unsorted list, which is now smaller than its previous size.

B. Backward Selection:

This step selects the last element as the 'current_highest' element. It removes that element from the unsorted list and appends it to an empty 'backward_sorted' list, which contains all the sorted elements in the backward selection pass. Then it compares this 'current_highest' with the next element in the unsorted list, but in backward direction. If it is less than the 'current_highest', the next element is checked. If the element is greater than or equal to the 'current_highest', it removes this element from the unsorted list, and adds it to the 'backward_sorted' sublist. This newly removed element then becomes the 'current_highest' and is used for comparison with the next element in the unsorted list. This selection is done till the end of the unsorted list is reached. Finally, a 'backward_sorted' sublist is obtained which contains elements in ascending order and an unsorted list, which is now smaller than its previous size.

C. Merging:

The two 'forward_sorted' and 'backward_sorted' sublists are then merged using the technique used for merging in merge sort [8] to get the 'intermediate_sorted_1' list. At the end of this step, two lists are obtained. One is the unsorted list (which is small than its original size) and the second is the 'intermediate_sorted_1' list. This it the first intermediate list obtained by a recursive call on either the left or right sublists after partition. Similarly, another 'intermediate_sorted_2' list is obtained by the recursive call on the other sublist. These two intermediate lists will then be merged and returned by the sorting function. As each recursive call returns, a fully sorted list is obtained at the end of the last recursion.

D. Partition:

After obtaining the 'intermediate_sorted_1' list, the unsorted list is partitioned from the middle to obtain two unsorted left and right sublists. The middle is found by dividing the length of the unsorted list by 2 and taking the floor of the value obtained. Now, the sorting algorithm is called recursively on both the left and right unsorted sublists.

Boundary Conditions:

If the unsorted sublist contains elements less than one, the function returns. As both forward and backward selection is applied which removes the elements from the original list, the forward selection may cause the list to become empty. Hence, the length of the list after forward selection is checked to be at least greater than one, otherwise the backward selection procedure will be useless.

Algorithm:

Algorithm SUPERSORT(L, left, right):

```
//Input : List L(contains n elements), left(lower index), right(higher index)
//Output : List L (ascending order)

if len(L) < 1 then:
    return()
forward_sorted ← empty list
backward_sorted ← empty list

//Forward selection
current_highest ← L[left]
forward_sorted.append(current_highest)
L.remove(current_highest)
For i ← left to right-1 do
    if L[i] >= current_highest then
        current_highest ← L[i]
        forward_sorted.append(current_highest)
        L.remove(current_highest)

//Backward selection
if len(L) > 1 do
    current_highest ← L[len(L)-1]
    backward_sorted.append(current_highest)
    L.remove(current_highest)
    for i ← len(L)-1 to 0 do
        if L[i] >= current_highest then
            current_highest ← L[i]
            backward_sorted.append(current_highest)
            L.remove(current_highest)

intermediate_sorted_1 ← MERGE(forward_sorted, backward_sorted)
mid ← len(L) / 2
mid_list_1 ← SUPERSORT(L, 0, mid)
mid_list_2 ← SUPERSORT(L, mid+1, len(L))

intermediate_sorted_2 ← MERGE(mid_list_1, mid_list_2)

return(merge(intermediate_sorted_1, intermediate_sorted_2))
```

Algorithm MERGE(M, N):
//Input : List M (sorted ascending order), List N (sorted ascending order)
//Output : List merging_list (sorted ascending order)

```
merging_list ← empty list
m ← len(M)
n ← len(N)
i ← 0
j ← 0

while(i+j < m+n) do
      if (i == m) then
            merging_list.append(N[j])
            j ←j+1
      else if (j == n) then
            merging_list.append(M[i])
            i ←i+1
       else if (M[i] <= N[j]) then
            merging_list.append(M[i])
            i ←i+1
       else if (M[i] > N[j]) then
            merging_list.append(N[j])
            j ←j+1
return(merging_list)
```

IV. Research Methodology

The forward selection step is used for picking the elements that may have a sorted order in the forward parsing of the unsorted input list. If the input list contains elements in descending order, then the forward selection step will always return only one number as there will be no number greater than the first i.e the 'current_highest'. For example, if the input list is 10, 9, 8, 7, 6, 5, then, for the forward selection, every time the algorithm will return only one number. Thus, backward selection step is used to overcome this drawback. In backward selection the numbers 5, 6, 7, and so on will be selected in a single pass.

Further, if the input list contains the numbers as 10, 9, 8, 7, 6, 5, 5, 6, 7, 8, 9, 10, i.e. a list that has a decreasing sequence followed by an increasing sequence of numbers, then, for every forward and backward selection, again only one number in obtained in each step. Thus, divide and conquer is used to partition the list from the middle, so that two sublists as 10, 9, 8, 7, 6, 5 and 5, 6, 7, 8, 9, 10 are obtained. Now, the elements in the first sublist will be selected in two passes only. First 10 will be selected in the forward pass and finally 5, 6, 7, 8 and 9 will be selected in the backward pass. For the second sublist, all the elements are already in ascending order. Thus, only one pass will be required. The two sublists are merged to form a bigger sublist and the algorithm returns this sublist. This recurisve merging will yield a fully sorted list after the final merging occurs.

Example:

The figures 1, 2 and 3 show an example of the sorting procedure using super sort.

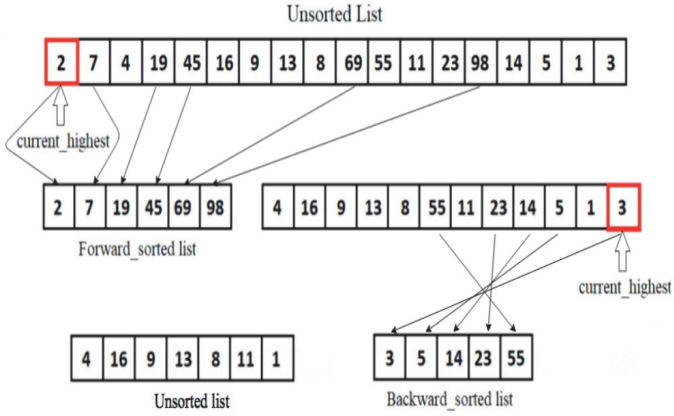

Figure 1. Forward and backward selection

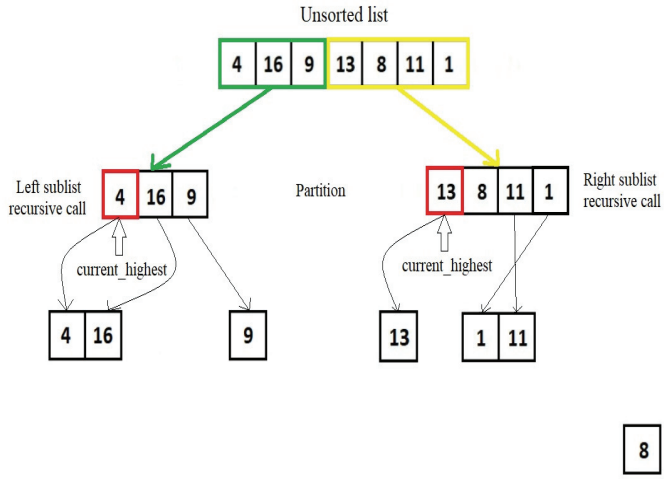

Figure 2. Recursive calls on left and right sublist

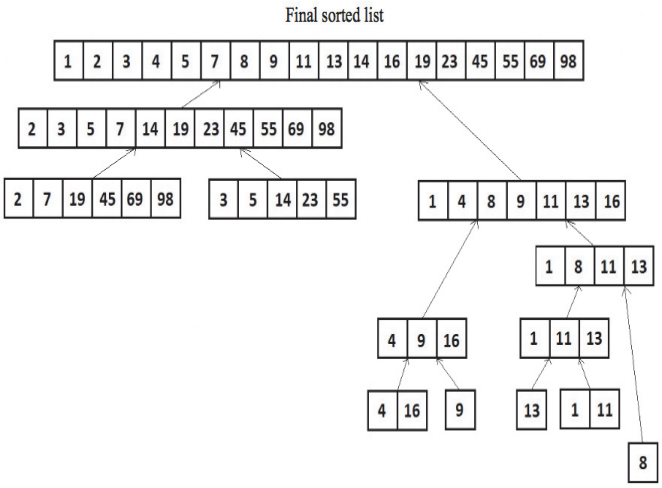

Figure 3. Merging upwards to get final sorted list

V. Analysis Of The Algorithm

A. Best Case:

The super sort algorithm selects the sequence of sorted elements while traversing the input list in forward selection and backward selection. If the elements are already sorted in the correct sequence, it will select the elements in sequential order in the first pass. Hence, the time complexity in this case will be T(n) = O(n) where n is the number of elements.

B. Worst Case:

If on every forward and backward selection, the algorithm appends only one element to the sorted list, it will take n^2 steps to sort all the elements. For example, if the input sequence is 6,4,2,1,2,1,3,4,2,1,2,1,3,5, then in every forward and backward pass only one element is obtained. Due to the divide and conquer strategy, even though elements like 4 and 3 occur more than once in the unsorted list, they will appear only once when separated as left and right sublists. Thus, the worst case time complexity will be $T(n) = O(n^2)$.

C. Average Case:

If the input list is a random sequence of numbers, then the time complexity is T(n) = O(nlogn).

TABLE II. PERFORMANCE IN BEST CASE

Size of input list (n)	Time to sort (milliseconds)
100	0.06
1000	0.69
10000	12
100000	1180

TABLE III. PERFORMANCE IN WORST CASE

Size of input list (n)	Time to sort (milliseconds)
100	0.39
1000	58.46
10000	233.85
100000	2265.18

TABLE IV. PERFORMANCE IN AVERAGE CASE

Size of input list (n)	Time to sort (milliseconds)
100	0.34
1000	5.51
10000	88.01
100000	1311.31

TABLE V. COMPARISON WITH OTHER SORTING ALGORITHMS IN AVERAGE CASE

Sorting Algorithm	Time to sort (seconds) for input of size n				
	100	*1000*	*10000*	*100000*	*1000000*
Bubble Sort	0.06	0.10	7.74	895.7	14098.02
Insertion Sort	0.09	0.56	5.62	788.4	12336.48
Selection Sort	0.03	0.34	3.38	491.8	9317.96
Super Sort	0.005	0.01	0.18	2.39	18.28
Merge Sort	0.004	0.06	0.08	1.00	6.48
Quick Sort	0.001	0.01	0.02	0.31	4.32

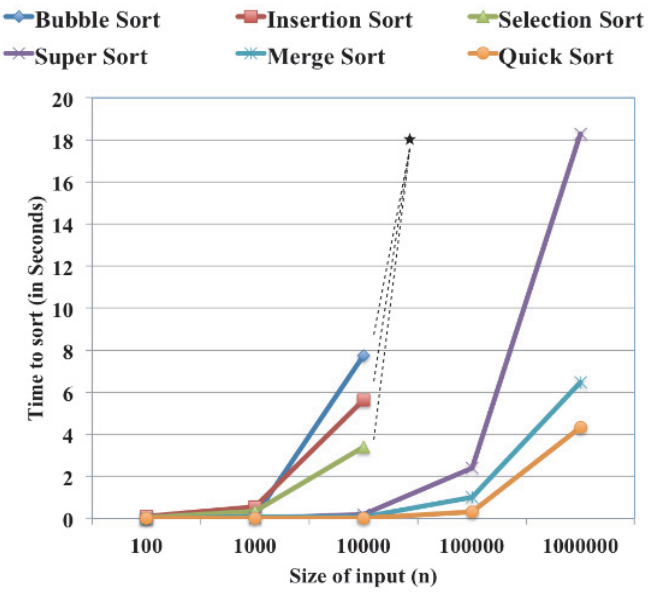

Figure 4. Comparison with other sorting algorithms in average case

VI. Experimental Evaluation

All the algorithms such as bubble sort, insertion sort, selection sort, super sort, merge sort and quick sort were executed on Intel Core i5 processor of 2.5GHz with 12 GB DDR4 RAM and having 64 bit Ubuntu 16.04 LTS as the operating system. The algorithms in the present study were implemented in Python 3.5 [8]. Tables II, III and IV show the time required for best, worst and average case time complexities of super sort algorithm when implemented in Python. The table V. shows the performance comparison between the sorting algorithms. The value of n varies from 100 to 1000000, where n is the number of elements to be sorted. The time required for sorting is measured in seconds. Figure 4 shows the graphical representation of table V. *Due to the large amount of time required for sorting 100000 numbers and above by bubble, insertion and selection sort, they are not plotted on the graph. As we can infer from the graph, the super sort algorithm sorts the elements faster than bubble sort, insertion sort and selection

sort for average cases, but is slower than merge sort and quick sort. The implementation of the super sort algorithm can be found at https://github.com/yashgugale/Super-Sort-Sorting-Algorithm/blob/master/super_sort.py.

VII. Limitations

The super sort algorithm selects the next element based on the value of the current highest element. Thus, if there are two occurrences of the same number, they may not necessarily be selected in the sequence in which they are present in the input list. For a list that contains 5, 2, 6, 3, 2, the first occurrence of 2 will not be selected in the forward pass as 5 is the current highest. However, in the backward pass, the current highest will be the second occurrence of 2. Hence, it will be selected first. Thus, the super sort sorting algorithm is not stable.

Also, the algorithm merges two sorted sublists to form a bigger sorted list. However, if the sublists are very small and contain only a few elements, then the merge will be an expensive operation. Thus, if the number of merging steps is reduced, it will greatly reduce the running time of the algorithm.

VIII. Conclusion And Fututre Work

The algorithm is implemented using the Python programming language for convenience of implementation. However, as Python does not support pointers, every recursive call on the sublist needs to be done by creating new left and right sublists. This is an expensive operation. Also, as lists in Python are mutable, we cannot manipulate the same list, as this, unlike an array, will rearrange the numbers. To enhance the speed of the program, it can be implemented using C or C++ as the programming language of choice. Thus, for merging we would only need to readjust the pointers of the elements so that it points to the next bigger element in the sorted order. Also, for recursion we do not need to create new sublists and instead we can simply pass the linked lists after link readjustment. Thus, the future work would be to reduce some of the intermediate steps and implement the algorithm using C or C++ and then compare the results of time and space complexity with other sorting algorithms.

References

[1] https://en.wikipedia.org/wiki/Comparison_sort

[2] http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html

[3] Karunanithi A., Drewes F., A Survey, Discussion and Comparison of Sorting Algorithms, Ume_a University, June 2014.

[4] C.A.R. Hoare, “Quicksort,” The Computer J, Vol. 5, No. 1, Apr. 1962, pp. 10-15

[5] D.E. Knuth, “The Art Of Computer Programming: Vol. 3, Sorting and Searching”, 3rd Edition Addison-Wesley, 1997.

[6] Horowitz and Sahani, “Fundamentals of Computer Algorithms”, 2nd Edition, University Press, 2008.

[7] Cormen T., Leiserson C., Rivest R., and Stein C., “Introduction to Algorithms”, 3rd Edition, MIT Press, 2009.

[8] Goodrich M., Tamassia R., Goldwasser M., “Data Structures and Algorithms in Python”, John Wiley and Sons, 2013.