ADA_Research_Paper_Study

by Sachin Motwani

CO 34563

Design and Analysis of Algorithms

Study of Research Paper on Super Sort algorithm

Utkarsh Shrivastava - 0801CS171090

Sachin Motwani - 0801CS171065

Department of Computer Engineering

Shri G.S. Institute of Technology and Science, Indore

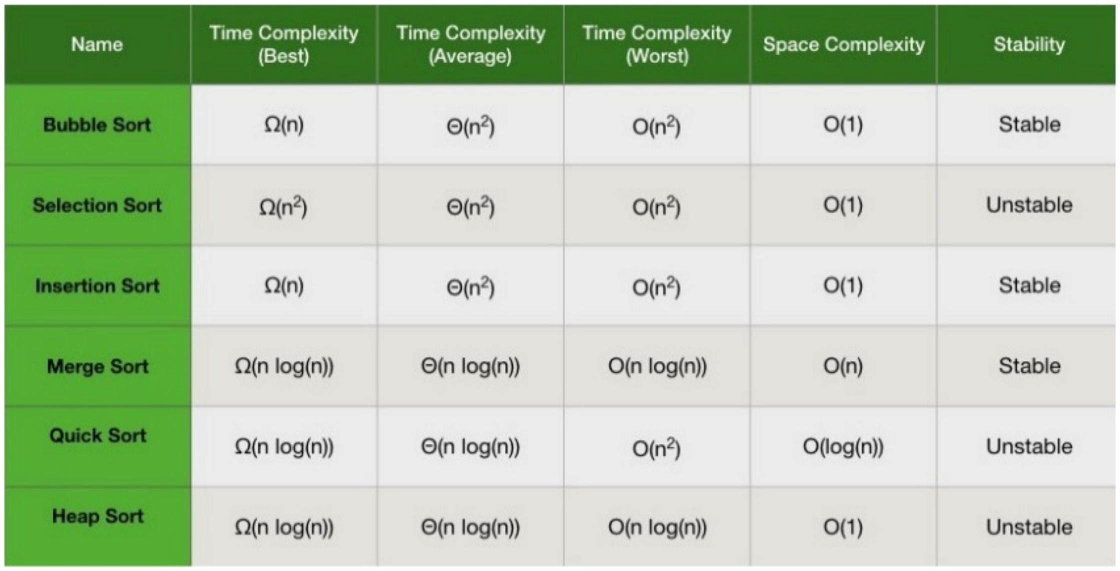
1. PROBLEM STATEMENT

The sorting procedure provided in this research paper aims to sort a sequence of elements in either ascending or descending order by bringing in use the natural sequence of sorted elements in a given array consisting of random numbers so as to reduce the number of iterations/steps for sorting. The algorithm proposed in this paper sorts the given array of elements in O(nlogn) time, n being the total number of elements in the array.

2. LITERATURE SURVEY

Sorting algorithms have become a vast area of research since the beginning of computing because of the great complexities and various methods despite the simple aim of the problem. Researches have evolved with various sorting techniques which differ in performance on the basis of their time complexity, space complexity and stability. These algorithms developed so far may vary in performance depending on the order of elements in the sequence provided, and the number of elements in some cases. Researchers have come up with sorting techniques which solve the problem in similar time or even better than the algorithm this paper has, but there are still many ongoing researches to improve the performance of sorting algorithms.

Below is a comparison between various sorting algorithms which have been researched upon yet and are commonly used.



Name	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Stable
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Unstable
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Stable
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$	Stable
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$	Unstable
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$	Unstable

3. SOLUTION

The algorithm can be divided into the following steps:

1. Forward selection:

Begins with setting ‘current_maximum’ to the first value of ‘unsorted_array’ (i.e. Input array), adding the value to a new ‘sorted_forward’ array and removing this value from the ‘unsorted_array’. Then iterate over the ‘unsorted_array’ while checking if any value is greater than the ‘current_maximum’. If a greater value is found, ‘current_maximum’ is updated by it and; the value is added to the ‘sorted_forward’ array as well as removed from the ‘unsorted_array’. This keeps on iterating until the last element of ‘unsorted_array’ is visited. Hence, *this step removes all the naturally occurring forward sorted sequences into a separate array.*

2. Backward selection:

This sets the ‘current_maximum’ to the last element of ‘unsorted_array’ by adding it to the ‘sorted_backward’ array and removing the same from the ‘unsorted_array’. It again checks in the ‘unsorted_array’ if any element is greater than the ‘current_maximum’ from right to left. If found, it updates ‘current_maximum’ and removes the value from ‘unsorted_array’ as well as adds it to the ‘sorted_backward’ array. Repeat this until the first element of ‘unsorted_array’ is visited. This step is similar to the Forward Selection, just that ‘unsorted_array’ is traversed in the opposite direction. Hence, *this step removes all the naturally occurring backward sorted sequences into a separate array.*

3. Merging forward sorted array and backward sorted array:

In this step, the two sorted arrays; ‘sorted_forward’ and ‘sorted_backward’ are merged using the same method as in Merge Sort. This generates a sorted array; ‘merge_1’. *This step finally sorts all the naturally occurring sorted sequences and gives a resulting array.*

4. Handling residual unsorted array:

There still might be some elements remaining in ‘unsorted_array’. In order to sort these, the ‘unsorted_array’ is divided into two parts; ‘Left_subpart’ and ‘Right_subpart’. Each subpart is recursively sorted using the supersort. This step results in ‘sorted_Left_subpart’ and ‘sorted_Right_subpart’ arrays. Hence, *this step takes into account the residual unsorted array by dividing the problem recursively.*

5. Merging the subparts of unsorted array:

The previously generated sorted arrays are again merged and sorted using the same method in step 3. This generates a ‘merge_2’ array. *This step finally sorts the residual unsorted array that was divided into parts.*

6. Generating the answer:

This is the final step that involves merging and sorting the arrays; ‘merge_1’ (from in step 3) and ‘merge_2’ (from step 5). The result obtained is returned as the final answer. *Therefore, this step finishes the algorithm off by performing the final merge of the two intermediate arrays formed*

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 ⬅ 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))
```

Following images depict the process explained above:

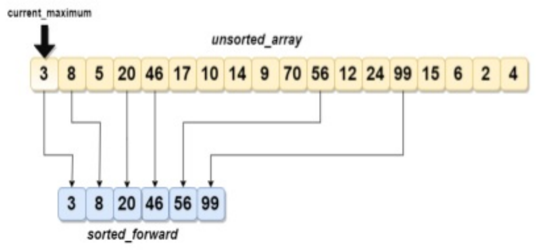


Fig 1. Forward selection

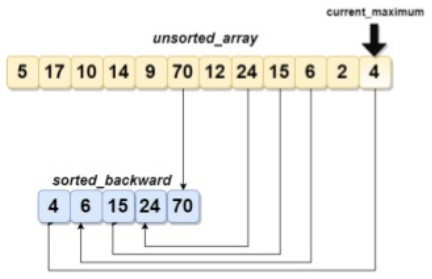


Fig 2. Backward selection

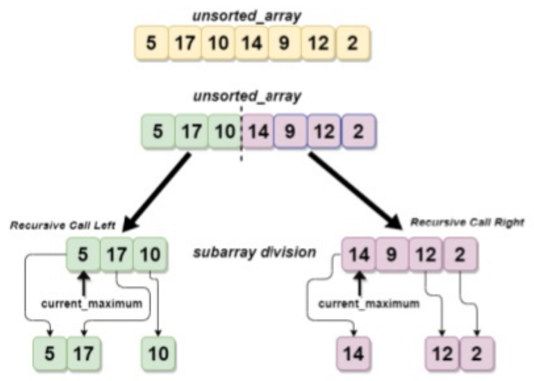


Fig 3. Recursively sorting residual unsorted array

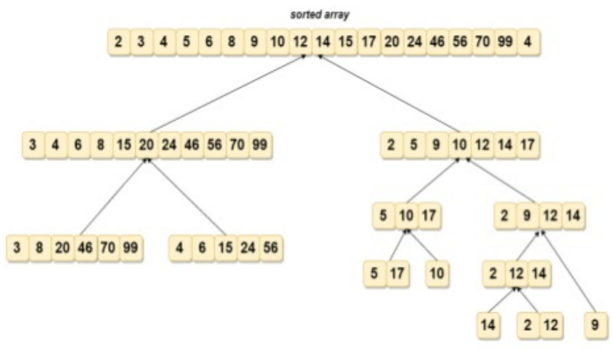


Fig 4. Recursive function merge; to merge the input subarrays in sorted manner

4. JUSTIFICATION

This algorithm performs two major operations in every step or iteration that are forward and backward selection. If in case the input contains elements which are in decreasing order, then forward selection would only extract one element in every pass which is undesirable. But, for the given scenario, in the backward selection process, all the remaining elements would be selected in a single pass thus completing the sorting process in minimum time i.e. 2 passes.

If the input array contains a sequence of numbers like first increasing then decreasing or vice versa, then there might be only one element selected in both the steps which is again undesirable. Therefore, divide and conquer is applied in this algorithm. The remaining list(array) is divided into two sub lists from the middle and the algorithm is applied on both the lists. Hence, after division of the list, the sorting might be completed in 2 or few passes making the algorithm efficient.

Below is the analysis of the algorithm in various cases:

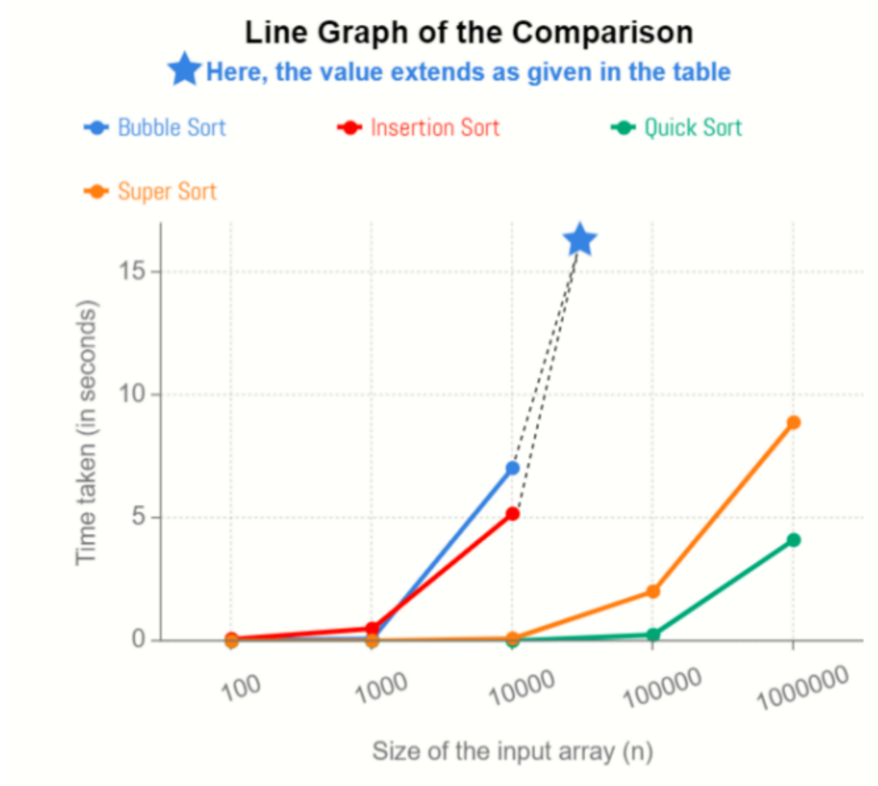
a. Best Case:
The best case will be encountered when the array is already sorted. Here the entire unsorted array would be emptied into the forward sorted array, that is the first pass, and hence it'll take one iteration for the algorithm to sort the array. Therefore, the time complexity attained would be O(n).

b. Worst Case:
The worst is obtained when the array is of decreasing nature at first and then of increasing nature. Say for e.g. [20,19,18, 17, 16, 15, 16, 17, 18, 19, 20]; in this kind of input one element is added to the forward and backward arrays in each pass. Further recursion takes place for that pass making it consume even more time. Therefore, here the time complexity is O(n^2).

c. Average Case:
In an average case where we encounter some arbitrary values at each point in the array; we get an average time complexity of O(nlogn)

TABLE: Time taken by various sorting algorithms in average case

Sorting Algorithm	***100***	***1000***	***10000***	***100000***	***1000000***
Bubble Sort	*0.058*	*0.096*	*7.046*	*890.09*	*14081.73*
Insertion Sort	*0.089*	*0.512*	*0.519*	*781.9*	*12327.57*
Quick Sort	*0.001*	*0.009*	*0.017*	*0.257*	*4.12*
Super Sort	*0.003*	*0.026*	*0.11*	*2.03*	*16.93*



NOTE: The above analysis was done by implementing the algorithm in python programming language on a 64-bit machine having intel core i7 processor (5th gen.) with 3.2 GHz clock frequency and 8 GBs of RAM. The values might vary slightly on a different system with different configuration. For other sorting algorithms, results were obtained from web.

5. IMPROVISATION/SUGGESTIONS

The stated algorithm performs merging (of two different lists) which proves to be a costly operation if the lists are small and the operation is performed upon again and again. Apart from merging, insertion and removal operations on arrays are also tedious. One improvement could be if the algorithm uses linked lists for managing intermediate lists rather than arrays since manipulating pointers is cheaper than manipulating absolute arrays (in terms of operations).

Please find our implementation of the discussed algorithm here:
https://github.com/AchillesWithoutTheHeel/Super-Sorting-Algorithm/blob/master/supersort.py

ADA_Research_Paper_Study

ORIGINALITY REPORT

18%	1%	18%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Yash Gugale. "Super Sort Sorting Algorithm", 2018 3rd International Conference for Convergence in Technology (I2CT), 2018 Publication	17%
2	cse.vnit.ac.in Internet Source	1%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		

ADA_Research_Paper_Study