Abstract—The purpose of this paper is to propose a new sorting algorithm (tentatively titled Gupta Sort). This in-place, recursive, sorting algorithm with a time complexity of $O(n^2)$, is ideal for sorting integers, floats as well as characters, stored in both arrays and doubly linked lists. It greatly reduces the number of elements we have to compare against to find the correct position of each element, and has performed better than selection sort, insertion sort, and bubble sort for a number of cases. Further, this algorithm can be randomised ensuring that there is no worst case.

Keywords—Sorting, Randomisation, Algorithm, Comparison-Based.

I. INTRODUCTION

A sorting algorithm is one that puts the elements of a list in a certain predefined order. Thus, it takes a list of elements as input and produces an output which satisfies the following two conditions:

1. The output is in non-decreasing order. (Each element is no smaller than the previous element), although generally, elements in any predefined order can be considered to be sorted.
2. The output is a permutation of the input.

Some examples of well-known sorting algorithms include Quicksort, Mergesort, Insertion sort, Radix sort etc. Sorting is an extremely useful procedure, often used as a subroutine in various other algorithms (e.g. search and merge algorithms), and therefore of high importance. Thus, it becomes necessary to sort efficiently. Sorting algorithms are further classified on the basis of computational complexity, memory usage, adaptability, stability, whether or not they use comparison between two elements etc.

Gupta sort is an in-place, comparison based sorting algorithm, with a worst-case time complexity of $n^2$. However, it performs far better than most algorithms with the same time complexity.

Given an arbitrary list of elements (could be floats or integers), all methods used to sort them exploit certain properties that determine the elements’ positions relative to each other. Gupta sort uses the fact that if we are aware of the number of elements from the list greater than or smaller than every element from the same list, that becomes sufficient criterion for sorting the list.

This happens because the largest element would have no greater elements, the second largest element would have just one (the largest), and the third largest would have two and so on.

A naïve implementation of the above logic would be using double loops, and finding the number of elements greater than each element, however it would be inefficient, as we would be required to traverse through the complete remaining list for each element.

Gupta Sort is a refined, in-place implementation of the above logic, with a much better performance

II. DESCRIPTION

The algorithm is implemented recursively. Each element in the array is actually a structure which stores the numeric value to be stored at that index along with a pointer to an index (initialised at -1). We pick the first element of the array, and move till we find a greater element. The index field of this greater element corresponds to the index of the first element. Now, we pick this element and move towards right till we find the next greater element. The last element so found is the largest element of the array. However, to find the correct position of the element stored at the index stored in the index field of the greatest element, we only need to look for elements that are positioned after the greatest element in the array. Thus we apply recursion with above process. Thus, effectively we have broken down the set of elements to look for to find the correct position of any given element.

The worst case comes when the elements are in decreasing order, which can be solved by randomisation.

III. ALGORITHM & PSEUDO-CODE

A. Algorithm

1. Assume the list of unsorted numbers stored in an array. Each element of the array consists of the number as well as a pointer (to store the index of another element in the array). Thus, all numbers of the list would contain a
pointer alongside themselves. Assume the list to contain N elements. Initialize all pointers to -1, i.e. they initially point to nothing.

2. Now, we start with the number of the first element of the array. We store that number in another variable, let’s call it X, and its index in Y. It starts traversing the list from left to right. As we move, if we come across an element whose number is greater than X, we update X to that number, and in the pointer field of that greater element, we store Y, and update Y to the index of the greater element. We continue this procedure till we reach the end of the array. (The end is marked by N)

3. The current value of the variable X and Y (after traversing the complete list) are the values of the number of the largest number in the array, and its index respectively.
   Proof: (By Contradiction): Let’s assume that this is not the largest element. Then either the largest element lies either to its right or its left. If it were to its left, we would never have reached the current element, as the values are updated only if it comes across a number larger than itself. And if it were to its right, then we would have updated the current value when we reached that value, as we have traversed the entire list.

4. Now we swap the last element with X, we do so by using Y, and decrement N. This marks the completion of one step, i.e. one element has found its correct position.

5. (Assuming original configuration) Now consider the pointer of the largest element found. This pointer points to the index of an element. Now, to determine its position. No element to its left is greater than this element.
   Proof: (By Contradiction) if any element towards its left was greater than this element, then we would have never reached this element, which is a contradiction, (since indices were stored of only those elements that we did in fact reach).

6. Also, elements greater than this particular element, would lie only toward the right of the index of the largest element.
   Proof: (By Contradiction): Let the element greater than it lie towards its right and towards the left of the largest element. Then while traversing, X would change values at that point, hence the largest element would not point to the current element, which is a contradiction.

7. Now, taking the number pointed by the largest element, we only need to look at elements beginning from the index of the largest element, towards the right (till we reach the modified value of N). Again, if we come across a number larger than this number, we update X and Y, and store the pointer in the pointer field of the larger number. The last element we come across is the second largest number, and we swap it with the second last element (element with index N, as index was decremented) and continue the procedure.

8. Decrement N after a number finds its correct position, to avoid looking at the sorted elements.

9. If we reach the beginning of the list, find the position of the first element (using N) and start the procedure again.

B. Pseudo-Code

procedure SORTARRAY(ar , previdx , last_swapped)
if n = 1 return
if previdx = 0
val ← ar1
idx ← 1
for i ← 2 to n
if ar1 > val
val ← ar1
if aridx > arprevidx
previdx ← idx
parenti ← idx
idx ← i
swap(aridx , arn)
decrement n
SORTARRAY(ar , previdx , idx)
else
if last_swapped >= 1 & last_swapped <= n
if previdx = last_swapped & arprevidx < arparent[last_swapped]
SORTARRAY(ar , parentprevidx , last_swapped)
else
val ← arprevidx
idx ← previdx
if arlast_swapped > arprevidx
val ← arlast_swapped
idx ← last_swapped
for i ← last_swapped to n
if ar1 > val
val ← ar1
parenti ← idx
IV. COMPARISON GRAPH

If we consider Gupta Sort to be a hundred percent efficient, then in the graph above, we have shown other sorting algorithms’ efficiency as compared to Gupta Sort.

V. CONCLUSIONS

Sorting is an extremely important and has found uses in various tasks ranging from simple scheduling to Database management, and hence it becomes imperative to do it efficiently. Gupta sort has been shown to be better than Insertion sort for most test cases. Also, since insertion sort is preferred when sorting less number of elements, Gupta sort can be highly effective in those situations.

REFERENCES