EasyChair Preprint

№ 8081

Pointed-Sort

Armaan Garg and Shashi Shekhar Jha

May 24, 2022

Pointed Sort

Armaan Garg, Shashi Shekhar Jha

Department of Computer Science & Engineering

Indian Institute of Technology Ropar

Email: (armaan.19csz0002,shashi)@iitrpr.ac.in

Abstract

Sorting is one of the most studied topics in computer science. It is the technique of arranging any group of entities such as elements of an array in a particular sequence usually in ascending or descending order. In recent times such a technique is widely used to query large database systems, which may be the requirement of an end-user or the data is needed in a sorted manner as an input to another algorithm or a system. In this paper, we propose a new sorting algorithm that performs well in some cases and comparable in others against baseline algorithms that are known to be the benchmarks in this field of study. The aim of the proposed algorithm is to sort an array of n elements. The proposed algorithm is known as Pointed-Sort with best and worst-case time complexity of $\Omega(N)$ and $O(N(LogN)^3)$ respectively.

Keywords: Sorting, Pointers, Time-complexity, Space-complexity

I. Introduction

Every database in the world relies on efficient sorting algorithms for quicker response time. Any time a user queries the database of a computer machine/server, the results are presented in a sorted manner based on a defined preference. Sorting technique is used to make the lookup search efficient and also helps in prepossessing of data in defined order. Opposite of sorting is random-order/shuffling that is how raw data is present, which is of very little use when it comes to extracting some useful information out of it. Sorting/Arranging is of key significance since it streamlines the handiness of information. We can notice a lot of arranging models in our day-to-day existence, for example, we can find required things without much of a look-around in a shopping center or utility store in light of the fact that the things/utilities are displayed in an ordered fashion. Observing something from a word reference is certifiably not a drawn-out task since every one of the words is given in arranged structure. Additionally, observing a phone number, name, or address from a phone index is likewise extremely simple because of the favors of arranging.

The research on the sorting algorithms have become somewhat saturated, but here we propose a new sorting algorithm - Pointed-Sort, with best case time complexity better than Merge-Sort and Quick-Sort. Other popular sorting algorithms like, Bubble-Sort and Insertion-Sort have comparable best case time complexity but with average case time complexity of $\Theta(N^2)$ limits there use only to small number of elements. But, there is no single algorithm that is best in sorting all types of data.

The contribution of this paper lies in achieving better time complexity ($\Omega(N)$) results that Merge-Sort and Quick-sort in simpler scenarios (best case) and performing comparable to Merge-Sort in more complicated cases (average case and worst case) and better than Quick-Sort in worst case.

This paper is organized as follows: section II presents the literature corresponding to various sorting algorithms, section III presents the proposed method (Pointed-Sort) with illustration of its working step-by-step. Section IV highlights the theoretical evaluation of the proposed algorithm in terms of its complexity and section V highlights the empirical performance of the proposed algorithm in comparison with the Merge-Sort and Quick-Sort. Finally, section VI concludes this paper.

II. Related Literature

Quick-Sort is one of the most poplar algorithms upto date [1]. It works based on divide and conquer strategy along with a pivot element. The array elements are partitioned around the pivot element, keeping the smaller ones to the left and bigger elements to the right of the pivot. The partitions are then sorted recursively in the same manner. That all can be done in an in-place fashion and very small extra memory is required. This gives the algorithm the space complexity of the order of $O(LogN)$. The worst case time complexity of Quick-Sort is of the order of $O(N^2)$ and the average case and best case time complexities are of the order $\Omega(NLogN), \Theta(NLogN)$ respectively.

Merge-Sort is another famous sorting algorithm [2], [3] along with Quick-Sort. It uses the same principle of divide and conquer. This algorithm divides the array into N sub-lists each containing single element and then the sub-list are merged in pairwise fashion while making sure the sub-lists are joined to give a sorted list. This process is repeated until we have single complete sorted list/array. Merge-Sort outdoes the Quick-Sort algorithm with its better worst case timer complexity of $O(NLogN)$ and is comparable in the best and average cases to that of Quick-Sort with the complexity of the order of $\Omega(NLogN), \Theta(NLogN)$. It has the drawback in terms of extra memory requirement for the temporary array and gives the space complexity of the order of $O(N)$.

Another algorithm known as Bubble-Sort [4] is famous in sorting domain due to its simplicity. It loops over the entire array and swap the adjacent elements if they are in the wrong order. This looping of the entire array has to be done N times in the worst case giving the worst case time complexity of the order of $O(N^2)$. Time complexity of the order of $\Theta(N^2) and \Omega(N)$ are found in the average and best case scenarios respectively. This algorithms handles the space complexity better than the above two discussed algorithm with the order of $O(1)$.

Fourth most popular algorithm among the ones discussed above is known the Insertion-Sort algorithm [3]. It works by splitting the array into two virtual lists one called the sorted list and the other as unsorted list. Element are picked from the unsorted list one by one and placed into the sorted list at the correct position. Initially, there is one element in the sorted section (As single element is always sorted) and when the algorithm finishes we get all the elements in the correct order in the sorted section. This algorithm has the same time and space complexity of that of Bubble-Sort with best case time complexity of the order of $\Omega(N)$. Time complexity in average and worst case is of the order of $\Theta(N^2) and O(N^2)$ respectively. The space complexity of Insertion-Sort is the order of $O(1)$.

Selection-Sort [4] is another famous sorting algorithm with time complexity of $O(N^2)$ in all the scenarios (best/average/worst) and have the space complexity of $O(1)$. It works by iterating through the entire array and finding the least value element and put it in the beginning in case of sorting in ascending order. This process needs to be repeated N times, hence the time complexity of $O(N^2)$ irrespective of how the original array looks like.

III. Proposed Approach

The proposed algorithm (Pointed-Sort) uses 3 pointers, a front pointer (f), a rear pointer (r) and a shifting pointing (s). At the beginning, pointer f and s points towards the first element in the array and the r pointer points at the second element (see Figure 1).

- Pointer f and r are compared, and if the element pointed by the f pointer is greater than the one pointed by the r pointer, the elements are swapped, given that pointer f and r are pointing to consecutive elements in the array.
- If the element pointed by the f pointer is smaller or equal than the one pointed by the r pointer, both the pointers move one step towards right, given that both are not pointing towards consecutive elements in the array. If so, then only the r pointer is shifted one element to the right in the array.

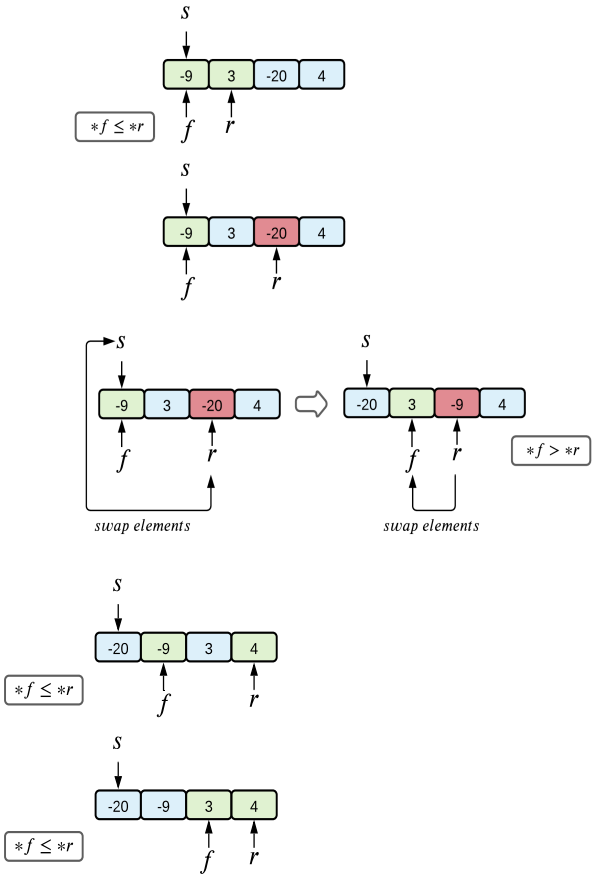

Fig. 1: Example of Pointed-Sorting Algorithm

- If the element pointed by the f pointer is greater than the one pointed by the r pointer and both these pointers are not pointing towards two consecutive elements in the array, then the s pointer comes into use. The element pointed by the

s pointer is compared with the element pointed by the r pointer and the s pointer moves from starting element to the position of the f pointer.

- Pointer s finds the right position for the element pointed by pointer r, such that the list is sorted upto the element pointed by pointer f. After the right position is found, the element pointed by s pointer is remembered using a temporary variable, and the element pointed by pointer r is copied into the position given by pointer s. All the elements are copied into there next cell location one-step towards right upto the location of pointer r.
- After that, the pointer s is reassigned towards the starting of the array (in the next iteration of the while loop), pointing towards the first element and pointer r is moved by one position to the right. The loop goes on until the r pointer reaches at the end of the array and the f pointer moves to the second last element of the array.

IV. Theoretical Evaluation

The best case ($\Omega(N)$) of Pointed-Sort occurs when most of the elements are already present in the right order and very less number of copying and pasting operations are required. Time complexity of Pointed-Sort in such cases is better than Quick-Sort and Merge-Sort and is comparable to that of Bubble-Sort and Insertion-Sort.

When the list is in reverse order (required ascending order, but given descending order), Quick-Sort gives the worst case time complexity of ($O(N^2)$) and Merge-Sort gives time complexity of ($O(NLogN)$). Pointed-Sort in comparison improves on Quick-Sort complexity by reducing it to ($O(N(LogN)^3)$).

The worst case of Pointed-Sort occurs when the shift pointer has to move from beginning of the array to the location of f pointer in every iteration and also all the elements upto the r pointer needs to be copied into the next cells. The average and worst case time complexity of pointed-Sort is the order of $\Theta(N(LogN)^3), O(N(LogN)^3)$ respectively. This complexity is calculated from the Pointed-Sort code using Iterative method.

The worst and average case time complexity of Pointed-Sort are verified empirically. For $N = 10000$, the inner most while loop of Pointed-Sort is called approximately 2,50,36,188 times at most, were the N numbers are in random order. This value increased to 4,99,94,999 approximately, when most of the numbers were in decreasing order or the entire array was in decreasing order. This is of the order of $N(LogN)^3$ (having base of Log as 2), which is proved to be better that N^2 where, $N \geq a$ where, $(a \in \mathbb{N})$. Similar complexity of $N(LogN)^3$ for Pointed-Sort is realised at higher values of N.

Proving worst case time complexity of Pointed-Sort is better than Quick-Sort:
(Log base 2 is considered in the proof)

To prove : $N(LogN)^3 < N^2$ where, $N \geq a$ $(a \in \mathbb{N})$

Put $N = a = 983$

$N^2 = 983 \text{ x } 983 > 983 \text{ x } 982.4 = N(LogN)^3$

Now suppose that, $N(LogN)^3 < N^2$ where, $N \geq 983$

//Removing N from both sides

$\implies (LogN)^3 < N$

//Rewriting the above statement

$= (LogN)^3 < 2^{LogN}$

//Let $u = LogN$

$\implies u^3 < 2^u$

$2^{u+1} = 2^u.2$

$> 2u^3$

$= u^3 + u^3$

$> u^3 + 9u^2$

$= u^3 + 3u^2 + 6u^2$

$> u^3 + 3u^2 + 54u$

$= u^3 + 3u^2 + 3u + 51u$

$> u^3 + 3u^2 + 3u + 1$

$= (u + 1)^3$

Based on the above proof using induction, it can be said that $N(LogN)^3 = O(N^2)$

Similar results are realised from the plots given in Figure 2. When the *s* pointer finds the element larger than the one pointed by *r* pointer relatively quicker, the number of comparisons are less but the number of copy and paste operations are more,

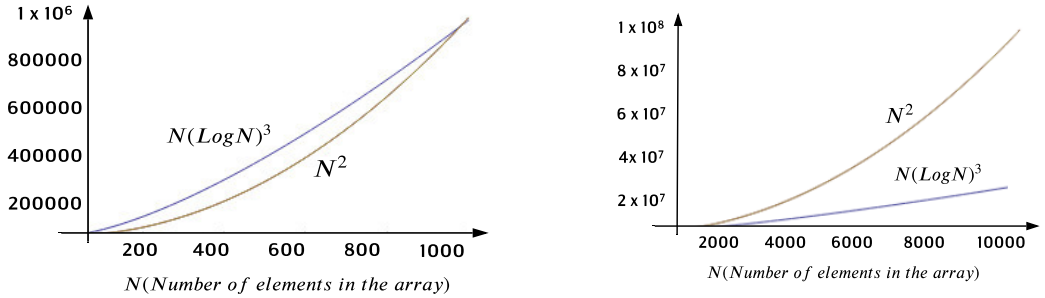

Fig. 2: N(LogN)^3 vs N^2 time complexity comparison plot based on the number of elements.

similarly vice-versa is also true. The copy and paste operations limits the space complexity of Pointed-Sort to $O(1)$ (Just the memory required to store the small number of temporary variables). This outdoes the space complexity requirements of Merge-Sort and Quick-sort as they have the space complexity of the order of $O(N)$ and $O(LogN)$ respectively. The source-code

Algorithm 1 Pointed-Sort

```
int pointed_sort(int [] a, int n)
int *p,*q,*u,*u2,u3,*p1,*p3,*p4,*q1
int *p2,*q2,t,t2,k,k1,m1,m2,k4,t4;
p=a;
q=a+1;
u=a;
while q ≤ a+(n-1) do
  //Loop until the rear pointer reaches at the end of the array.
  if (*p>*q) then
    //If the front pointer element is larger than the one pointed by the rear pointer.
    if ((q-p)==1) then
      //If the front and rear pointer are pointing at consecutive elements, swap the elements.
      p1=p;
      q1=q;
      t=*p1;
      *p1=*q1;
      *q1=t;
    else
      while (u≤p && *u<*q) do
        //If the front and rear pointer are not pointing towards consecutive elements.
        u=u+1;
      end while
      //Move the shift pointer towards the right until an element larger than the one pointed by the rear pointer is found.
      //Copy the element pointed by the rear pointer in the position pointed by the shift pointer.
      q2=q;
      p2=p;
      u2=u;
      k4=*u;
      t4=*(u+1);
      u3=*u2;
      *u2=*q2;
      p3=u;
      while (p3<q2) do
        //Copy the elements into the right side cell.
        p3=p3+1;
        *p3=k4;
        k4=t4;
        t4=*(p3+1);
      end while
    end if
  else
    if ((p+1)!=q) then
      //If the front and rear pointer are not pointing towards consecutive elements.
      p=p+1;
      q=q-1;
    end if
  end if
  q=q+1;
  //Move the rear pointer towards the right by one position.
  u=a;
end while
```

of the proposed approach is given by Algorithm 1. Table I compares the best to worst case time complexities of the discussed algorithms from the literature with the proposed algorithm along with space complexity.

Algorithms	**Time Complexity**			**Space Complexity**
	Best	**Average**	**Worst**	**Worst**
Bubble-Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	O(1)
Insertion-Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	O(1)
Selection-Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	O(1)
Quick-Sort	$\Omega(NlogN)$	$\Theta(NlogN)$	$O(N^2)$	$O(LogN)$
Merge-Sort	$\Omega(NlogN)$	$\Theta(NlogN)$	O(NlogN)	$O(N)$
Pointed-Sort	$\Omega(N)$	$\Theta(N(LogN)^3)$	$ON(LogN)^3$	O(1)

TABLE I: Time and Space complexity comparison of proposed algorithm with various algorithms from literature.

V. Empirical Evaluation

The performance of Pointed-Sort is measured in terms of CPU time using the system clock of the machine, while making sure of minimum background processing. Size of input array is varied from 1K,2K,3K....,10K (where, K means thousand) and comparisons are drawn between proposed algorithm (Pointed-Sort), Merge-Sort and Quick-Sort. Best case and worst case scenarios are considered and the input elements are kept same for all the three algorithms. The input range for the array element is varied from -100000 to 100000. The approximate results can be easily reproduced by giving the same array of elements to all the three algorithms. The array elements for these experiments were generated using online tool. The implementation is run on Dell Precision 7820 Tower Workstation with NVIDIA Quadro RTX 4000 8GB graphics card, 64GB DDR4 RDIMM ECC Memory, Intel Xeon Silver Series processor and primary storage of 512GB and 2TB 7200rpm SATA secondary storage.

A. Implementation Process

1) Generate 1K elements using online number tool.
2) (Optional) Generate nearly sorted list in increasing or decreasing order for best and worst case results respectively.
3) User input this list into the array.
4) Record the time using clock_t typedef that is present in <time.h> library.
5) (Optional) Apply this to Quick-Sort and Merge-Sort for comparison.
6) Increase the number of elements by 1K and repeat steps 2 to 5.

B. Results and Analysis

As can been observed in Figure 3(a), Pointed-Sort significantly improves over the CPU time efficiency of Merge-Sort and Quick-Sort in best case scenario. For average and worst case scenarios, Pointed-Sort is significantly able to improve in terms of time efficiency as compared to CPU time of Quick-Sort as observed in Figure 3(b).

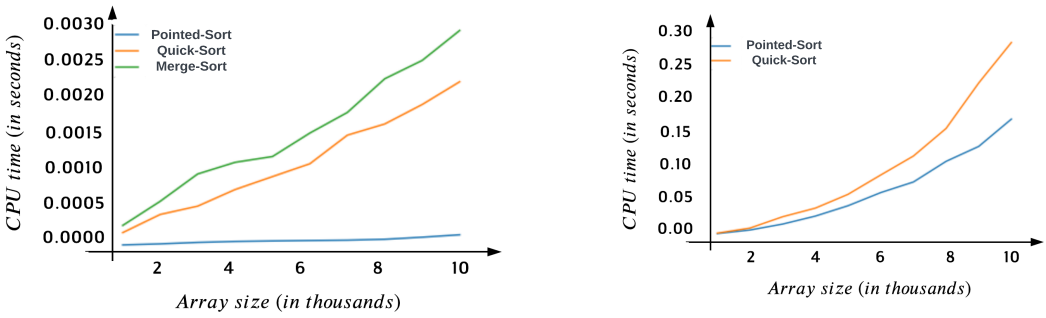

Fig. 3: (a) Best case time complexity comparison between Pointed-Sort, Quick-Sort and merge-Sort (b) Worst case time complexity comparison between Pointed-Sort and Quick-Sort

VI. Conclusion

We introduce a new sorting algorithm (Pointed-Sort) with the worst case complexity of $O(NLogN^3)$. It works based on the mechanism of pointers, where a shift pointer is moved within a pointed range of the array and perform sorting operation. Pointed-Sort performs better than Merge-Sort and Quick-Sort in terms of time efficiency of the order of $\Omega(N)$ in simpler scenarios where much of the elements are sorted. When it comes to complex/worst case scenario, Pointed-Sort performs better than Quick-Sort and comparable to Merge-Sort with the time complexity of $O(NLogN^3)$. Theoretical and empirical evaluation shows that pointed-Sort is better alternative as compared to the major sorting algorithms in simpler scenarios and gives competitive time complexity in case of worst scenarios.

After comprehensive experimental evaluation, we show that Pointed-Sort outperforms Quick-Sort in worst case scenario when the value of $N \geq 983$ and outdoes Merge-Sort in best case scenario. With it's technique Pointed-Sort also performs better than Merge-Sort and Quick-Sort in terms of space complexity that is the of the order of $O(1)$ as compared to that of $O(N)$ $O(LogN)$ for Merge-Sort and Quick-Sort respectively.

VII. Acknowledgments

First author would like to thank TCS for their support under the TCS Research Scholar Program.

References

[1] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 01 1962. [Online]. Available: https://doi.org/10.1093/comjnl/5.1.10

[2] D. E. Knuth, *The art of computer programming*. Addison-Wesley, 1998.

[3] J. Katajainen and J. L. Träff, "A meticulous analysis of mergesort programs," *Lecture Notes in Computer Science Algorithms and ComplexityJyrki Katajainen and Jesper Larsson Traff: A meticulous analysis of mergesort programs*, p. 217–228, 1997.

[4] E. H. Friend, "Sorting on electronic computer systems," *Journal of the ACM*, vol. 3, no. 3, p. 134–168, 1956.