A UNIQUE SORTING ALGORITHM WITH LINEAR TIME & SPACE COMPLEXITY

Sanjib Palui[1], Somsubhra Gupta[2]

[1]*PLP, CVPR Unit, Indian Statistical Institute, WB, India*
[2]*HOD, Information Technology, JISCE, WB, India*

Abstract

Sorting a list means selection of the particular permutation of the members of that list in which the final permutation contains members in increasing or in decreasing order. Sorted list is prerequisite of some optimized operations such as searching an element from a list, locating or removing an element to/ from a list and merging two sorted list in a database etc. As volume of information is growing up day by day in the world around us and these data are unavoidable to manage for real life situations, the efficient and cost effective sorting algorithms are required. There are several numbers of fundamental and problem oriented sorting algorithms but still now sorting a problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently and effectively despite of its simple and familiar statements.

Algorithms having same efficiency to do a same work using different mechanisms must differ their required time and space. For that reason an algorithm is chosen according to one's need with respect to space complexity and time complexity. Now a day, space (Memory) is available in market comparatively in cheap cost. So, time complexity is a major issue for an algorithm. Here, the presented approach is to sort a list with linear time and space complexity using divide and conquer rule by partitioning a problem into n (input size) number of sub problems then these sub problems are solved recursively. Required time and space for the algorithm is optimized through reducing the height of the recursive tree and reduced height is too small (as compared to the problem size) to evaluate. So, asymptotic efficiency of this algorithm is very high with respect to time and space.

Keywords: *sorting, searching, permutation, divide and conquer algorithm, asymptotic efficiency, space complexity, time complexity, and recursion.*

----------***----------

1. INTRODUCTION

An algorithm [1] [2] [3] [6] [7] is a finite set of instructions, that if followed, accomplish a particular task. Algorithm must satisfy some characteristic like having input, output, definiteness, finiteness, effectiveness.

Sorting [2] [5] [8] a list means a particular permutation of a given sequence in which the elements of the sequence are arranged in increasing/decreasing order. Sorting algorithms used in computer science are often classified by:

- Computational complexity [5] [9] [10] (worst, average and best behavior) of element comparisons in terms of the size of the list.
- Computational complexity of swaps (for "in place" algorithms) are sometimes characterized in terms of the performances that the algorithms yield and the amount of time that the algorithms take.
- Requirements of memory [11] [12] and other computer resources.
- Recursion [13]. Some algorithms are either recursive or non-recursive, while others may be both.
- Stability: stable sorting algorithms maintain the relative order of records with equal keys i.e. values.
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, etc.
- Adaptability: Whether or not the pre-sortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Now a day there are many effective sorting algorithms. Although lots of researchers are working on this, but unlike other field of research, number of proposed new, innovative and cost effective work is very few in the field of sorting algorithm.

We have designed and applied one sorting algorithm to achieve linear time complexity. In this paper, this new algorithm is proposed. As compared to existing algorithm, it gives better result and also it has linear time and space complexity, we named this work of algorithm as --"A Unique Sorting Algorithm with Linear Time & Space Complexity".

Here, our work is organized as follows: PREVIOUS RELATED WORKS is given in section 2, ALGORITHM OF PROPOSED WORK is in section 3, ANALYSIS OF THE ALGORITHM is in section 4, and finally conclusion is given on section 5 and then REFERENCEs.

2. PREVIOUS RELATED WORKS

Some well known sorting algorithm are bubble sort, selection sort, insertion sort ,merge sort, quick sort ,heap sort ,radix sort , cocktail sort ,shell sort etc. All these algorithms can be classified according to their average case and worst case time complexity (asymptotic complexity-big theta notation-θ and big oh notation – O respectively). Time complexity of some algorithms are given in Table 1 along with the stability of these algorithms i.e. they are stable or not. Here (with respect to the Table 1) *n* is the input size of the list to be sorted, *d* is number of digit in the largest number among inputs and k is all possible digit/word (for example- *k*=10 as decimal).

Some comparative study [11] [14] [15] [16] have been carried out in this field and situations of better suitemate for these algorithms (Table 1) are clearly notified. Depend on applications or data structure, which among sorting algorithm or modified version of an existing algorithm has less complexity than the original one or which among these algorithms is the best fitted in a particular circumstances are drawn on some proposed approach [17] [18] [19] [20].

Table -1: Information about Some algorithms

Name of the Algorithm	Time Complexity		Stable?
	Average case	Worst Case	
Bubble sort	$\Theta(n^2)$	$O(n^2)$	Yes
Selection sort	$\Theta(n^2)$	$O(n^2)$	No
Insertion sort	$\Theta(n^2)$	$O(n^2)$	Yes
Merge sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	Yes
Quick sort	$\Theta(n \log_2 n)$	$O(n^2)$	No
Bucket sort	$\Theta(d(n+k))$	$O(n^2)$	Yes
Heap sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	No

Depending on inputs and some predefined conditions, some new algorithms [18] [21] [22] have been projected to achieve better complexity. Some algorithm is designed for linear complexity [23] [24] but they can be executed in the system platform having some special characteristics as these algorithms demand for that. Poll sort, simple pancake sort, bead sort are the example of this type of sorting algorithms. Pancake sorting algorithm is not stable but has linear time complexity. Bead sort algorithm requires special hardware design to execute. Poll sort algorithm takes linear-time to execute but it is an analog algorithm for sorting a sequence of items, requiring O(n) stack space, and the sorting algorithm is stable but n (number of elements in input list) parallel processors are required.

There is an algorithm called Randomized Select [24] [25] algorithm for sorting. The expected running time of this algorithm is θ (n); a linear asymptotic running time where n is the input size of the problem. This algorithm works like Randomized Quick-sort [26] (where pivot element is select randomly from the list).Two main constraint of this algorithm

- All the elements in the input sub-problem are distinct.
- Partition is done based on random selection of an element.

Complexities of algorithms are strictly dependent on input size, caches performance etc., briefly description about complexities of algorithms are given some of the noble work [28] [29] [30].

3. ALGORITHM OF PROPOSED WORK

The pseudo-code of our work, --"A Unique Sorting Algorithm with Linear Time & Space Complexity" is given below

Sort (*A*, *lb*, *ub*)

Here *A* is a 1-D input list of decimal integer with *n* elements in memory where *n*=*ub*-*lb*+1 and *ub*=upper bound, *lb*=lower bound of the list.

```
Begin
 1.  if (lb<ub) then,
 2.    find min and max
 3.    if (min!=max) then,
 4.      set n = ub-lb+1
 5.      create n number of empty List
 6.      set div = (max - min) /n +1
 7.      for i=lb to ub by 1do,
 8.            j= (A[i] - min)/div
 9.            add A[i] to the j^th List
10.     end for
11.     set k = lb
12.     for i =0 to n by 1 do,
13.           set l = k
14.           set  size =number of elements in i^th List
15.           for j =0 to size by 1 do,
16.              A [k]= j^th element of i^th List
17.              set k = k +1
18.           end for
19.           If ((l <k-1) && (! ((l = = lb)&&(k = = ub))))
20.              call sort( A, l, k -1)
21.           end if
22.     end for
23.   end if
24. end if
End
```

Algorithm1.

min and *max* in line number 2 in Algorithm1 are minimum and maximum number from the list respectively. Here we have designed one variable for every "sort (*x*, *y*, *z*)" function call, called *div* which plays major role in the above mentioned pseudo code of the designed sorting algorithm. Every element of the input list forms its own index as it will be in the output list with the help of *div* variable[line number 8 of Algorithm1].Then all the elements of the input list with same indexes (as it is generated in line number 8 of Algorithm1) are solved as sub problems[27]. Elements with same index are treated here as one sub problem. For increasing order output----

- Every sub problem (except left most) as single unit has all smaller elements from the input list in its left.
- Every sub problem (except right most) as single unit has all larger elements from the input list in its right.
- Every sub problem can have 0 to n-1 elements.
- If number of elements in one sub problem is one it takes unit time cost to sort.
- Ideally list with n elements are partitioned in to n sub problems each of which have one element.
- It takes linear time complexity, O (n).
- Relative positions of two elements having same value are not changed, so this algorithm is stable.

All other strategies that are followed to execute this algorithm properly and efficiently-

- If the input list is combination of positive and negative integers, then two list are formed i.e. one list for negative numbers and another list for positive numbers. Then all the elements of the list of negative numbers are made positive. After performing algorithmic operations to make them as sorted, sign of elements are changed again and the list is reversed. Then operations are performed on positive list. At last, these two lists are concatenated.
- If the numbers are real, this algorithm can be applied easily. Problem is divided into sub problems according to their absolute value and if absolute values are same for all elements then partition is done based on their precision value.

4. ANALYSIS OF THE ALGORITHM

From the number statistics if the numbers are in uniform distribution then almost no recursions are happened, otherwise after first partition of the array, it will make greater than or equal to two uniform distributed array where complexity is linearly dependant on *n.* Here, after partition of the array some input elements are taken place in the array like in quick sort one element is fixed in exact position. Here number of fixed elements is 1 to *n* where *n* is the number of elements to be sorted.

4.1 Time Complexity

Asymptotic time complexities [3] [4] of this algorithm in different cases are described here-

4.1.1 Best Case

If the elements are uniform distributed, then ideally problems are divided in to n sub problems. So no recursive call is executed because almost each sub problem has one element.

So, the required time is

$T(n) = c + T(1)+T(1)+T(1)+T(1)\ldots\ldots\ldots\ldots n$ times
$\equiv c + O(n)$ where c=constant time
$\equiv O(n)$ as n is very large

4.1.2 Average Case

If number of elements *n* becomes very high, in first function call, divided sub-problems will be almost uniformly distributed. Let, here recursive call is happened *m* times where ($m<n$) and sub-problems have n_1, $n_2 \ldots$ n_m elements respectively.

So, $n_1+n_2+\ldots..+n_m+p= n$ ---- Eq. (1)

where *p* is the number of elements that are already taken place in the input list as sorted elements. So, time complexity is

$$T(n)=c.n+ T(n_1)+T(n_2)+\ldots\ldots+T(n_m)$$

Where

c=constant time

$=c.\ n+ c.n_1 +c.n_2+ \ldots\ldots+c.n_m$

[From the best case as the sub problems are uniform distributed of $n_1, n_2, \ldots, n_m$ elements, there is rare chance for every sub problem to give average case time complexity]

$= c.\ n+ c.\ (n_1 + n_2 + \ldots\ldots + n_m)$
$=c.\ n+ c.\ n$ [from, Eq. (1)]
$=2cn$
$\equiv O(n)$

4.1.3 Worst Case

The algorithm gives worst case time complexity when,

- Inputs elements are randomly distributed.
- All elements except the largest one of the input list have values less than the value of *div* as it is generated in line number 6 in our algorithm1 and it is happen again and again for every sub problem.
- The input series is one of all possible permutation of the elements of the series-----

  $a, i*(i-1)^{th}$ term $+c \quad \forall\ i=2$ to n - --Eq.(2)

  where a=starting element of this series and having a positive value and c > = 0.

4.1.3.1 Case-I

According to the proposed algorithm, at that situation, time complexity is

$$T(n)=c\ .n+ T(n-1)$$
$$\equiv O(n^2)$$

which is possible theoretically but not in real life because we are talking about asymptotic time complexity. So, number of input elements is very very high. Every real life database collects and stores similar kinds of data and it is a very rare chance that at least some of elements in the sorted output series satisfy Eq. (2).

So, for every real life problem this algorithm gives average case time complexity.

4.1.3.2 Case-II

Depend on that situation of the worst case we have added some more instruction with the original above mentioned algorithm (i.e. Algorithm1) in between line number 3 and line number 4.

- Second largest number is found out along with the largest number.
- Checking is done according to the series of Eq. (2). If match then *max* (variable of the algorithm1) is set to second largest number instead of the largest number.

After this, it is seen that the problem is partitioned at least in to three sub problems. It is empirically designed in such a way that time complexity of the worst case for this algorithm become $n \log_3 n$.

4.2 Space Complexity

Space complexities [4] [8] in different cases are fully dependent on number of times the subroutine (as described in Algorithm1) is called recursively and input size for that sub-problem for which this subroutine is called.

When input size is *n* ,this algorithm needs more memory for the *n* numbers of list which are initially empty and store the elements of same index in the time of inputs being processed (from Algorithm1-line number 5 and line number 9). Now total number of elements in the *n* numbers of list is *n.*

So, required memory unit for every subroutine call is
n (for input elements) +*n* (for index lists)+c
= 2**n* +c

Where c = memory unit for other variables
So, asymptotic space complexity is also linearly dependent on input size as this algorithm is implemented to reduce the depth of recursion tree.

Complexity and stability details of presented algorithm are in Table-2---

Table -2: Information about Presented algorithm

Time Complexity		Space Complexity	Stable?
Best Case	O(*n*)	Same as time complexity	Yes
Average Case	O(*n*)		
Worst Case	• $O(n^2)$ or $n \log_3 n$ Only for Eq.(2) • O(*n*)		

Some experimental results with respect to input sizes and required times of this algorithm are given in Table-3 as compared to the results of some other well known algorithms where computer environment, paradigm, inputs are same.

Table -3: Experiment Results

Input Size	Required Time in Millisecond				
	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Proposed Approach
15000	1248	442	328	7	7
30000	4863	1746	1325	22	13
60000	19904	7056	6883	71	25
90000	43550	15750	14921	150	27
120000	76099	28917	25132	265	36
150000	122416	44925	38081	394	45

5. CONCLUSIONS

This presented algorithm is implemented successfully through a repeatable task of design, carrying out and analyze. The advantages of this algorithm are its speed, requirement of less memory than existing one and stability. Selection of sorting algorithm is application and situation dependent but this algorithm works well in every field. For real life sorting problem, time complexity & space complexity of this algorithm are linear.

There is a broad future scope to experiment the proposed algorithm for finding out short-coming (if any) based on some uncovered real life test suits with solutions.

REFERENCES

[1]. D.E. Kunth, Fundamental Algorithms, The Art of Computer Programming: Vol.1, Addison-Wesley, 1968. Third Edition, 1997.
[2]. D.E. Kunth, Sorting and Searching, The Art of Computer Programming: Vol. 3, Addison-Wesley, 1973.Second Edition, 1998.
[3]. D.E. Kunth, Semi numerical Algorithms, The Art of Computer Programming: Vol.2, Addison-Wesley, 1969. Third Edition, 1997.
[4]. D.E. Kunth, Big omicron and big omega and big theta. SIGACT News,8(2)18-23,1976.
[5]. Cormen T., Leiserson C., Rivest R., and Stein C., Introduction to Algorithm McGraw Hill, 2001.
[6]. A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms.Addison-Wesley,1974.
[7]. A. V. Aho, J. E. Hopcroft and J. D. Ullman Data Structures and Algorithms.Addison-Wesley,1983
[8]. A. Aggarwal and J. Scott Vitter.The input/output Complexity of sorting and related problems. Communication of the ACM, 31(9):1116-1127,1998.
[9]. S. Baase and A. V. Gelder. Computer Algorithm: Introduction to Design and Analysis. Addison-Wesley, Third edition,2000.
[10]. Lavore, Robert. Arrays, Big O Notation and Simple Sorting. Data Structures and Algorithms in Java (2nd Edition). Sams, 978-0-672-32453-6..

[11]. A. Tridgell, Efficient Algorithms for Sorting and Synchronization, Ph.D. Thesis, Dept. of Computer Science, The Australian National University, 1999.
[12]. K. D. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. In Workshop on Memory Systems Performance (MSP '02), Berlin, Germany, June 2002.
[13]. M. Akra and L. Bazzi.On the solution of linear recurrence equation. Computational Optimization and Application,10(2):195-210,1998.
[14]. S. Jadoon, S. F. Solehria and M. Qayum, (2011) "Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study" International Journal of Electrical & Computer Sciences, IJECS-IJENS, Vol: 11 No: 02.
[15]. Y. Yang, P. Yu, Y. Gan, (2011) "Experimental Study on the Five Sort Algorithms", International Conference on Mechanic Automation and Control Engineering (MACE).
[16]. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. ACM Computing Surveys, 24:441–476, 1992.
[17]. W. Min (2010) "Analysis on 2-Element Insertion Sort Algorithm", International Conference on Computer Design And Appliations (ICCDA).
[18]. E. Kapur, P. Kumar and S. Gupta-"Proposal Of A Two Way Sorting Algorithm And Performance Comparison With Existing Algorithms"- International Journal of Computer Science, Engineering and Applications (IJCSEA) Vol.2, No.3, June 2012
[19]. M. Peczarski. New results in minimum-comparison sorting. Algorithmica,40:133–145, 2004
[20]. J.-L. Lambert. Sorting the sums (xi +yj) in o(n2) comparisons. Theoretical Computer Science, 103:137–141, 1992
[21]. S. Z. Iqbal, H. Gull, A. W. Muzaffar, (2009) "A New Friends Sort Algorithm",IEEE Second International Conference on Computer Science and Information Technology.
[22]. J.L. Bentley and R. Sedgewick, Fast Algorithms for Sorting and Searching Strings, ACM-SIAM SODA", pp. 360-369, 1997.
[23]. Box R. and Lacey S., A Fast Sort, Computer Journal of Byte Magazine, vol. 16, no. 4, pp. 315-315, 1991.
[24]. C. A. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). Communications of the ACM, 4(7), pp. 321-322, 1961.
[25]. R. W. Floyd and R.L. Rivest. Expected time bounds for selection. Communications of the ACM, 18(3):pp. 165-172, 1975.
[26]. M. D. McIlroy. A killer adversary for quicksort. Software -Practice and Experience, 29(4):341-344, 1999.
[27]. J. L. Bentley, D. Haken and J. B. Saxe. A general methods for solving divide and conquer recurrence. SIGACT News,12(3):36-44,1980.
[28]. L. Trevisan, Lecture Notes on Computational Complexity, Computer Science Division, U.C. Berkeley, 2002.
[29]. J. Hartmanis and R. E. Stearns, on the computational complexity of algorithms,1963.
[30]. A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. J. Algorithms, 31:66–104, 1999.

BIOGRAPHIES

Sanjib Palui has completed his bachelor degree in Information Technology and master degree in Software Engineering in the year of 2012 and 2014 respectively. Then he has joined Indian Statistical Institute as Project Linked Person in the unit of Computer Vision & Pattern Recognition (CVPR). He is working on Data Structure & algorithm, Image Processing and Optical Character Recognition fields.

Dr. Somsubhra Gupta is presently the Assistant Professor of the Department of Information Technology, JIS College of Engineering (An Autonomous Institution). He is graduated from the University of Calcutta and completed his Masters from Indian Institute of Technology, Kharagpur. He received Ph. D. award entitled "Multiobjective Decision Making in Inexact Environment using Genetic Algorithm: Theory and Applications" form University of Kalyani. His area of teaching is Algorithm and allied domains and research area is Machine Intelligence. In research, he has around 56 papers including Book Chapters so far in National / International Journal / Proceedings and over 40 citations. He is Principal Investigator / Project Coordinator to some Research projects (viz. RPS scheme AICTE). He was the Convener of International Conference on Computation and Communication Advancement (IC3A-2013). He is invited in the Technical Programme Committee of number of Conferences, delivered as an invited speaker, an INS (Elsevier Science) reviewer and attended NAFSA-2013 conference of international educators at St. Louis, Missouri, USA