Communications of ACM SIGPALN, Vol.31, No.3, March, 1996, ACM, pp.28-36

A New Sort Algorithm: Self-Indexed Sort

Yingxu Wang

Computing Laboratory, Oxford University
Oxford OX1 3QD, UK
y.wang@comlab.ox.ac.uk

ABSTRACT: This paper presents a new sort algorithm, self-indexed sort *(SIS),* on an approach of non compare-based sorting. Results on time complexity *O(n)* and space complexity *O(n+m)* are achieved, where *n* is the size of data being sorted and *m* is the size of the sorting space.

Two versions of *SIS* sort algorithm are implemented with one for the sort of structured records and the other for pure data file. The performance of *SIS* sort is tested and analysed under various conditions.

Index terms: *Computing, software, algorithm, sort, performance analysis*

1. Introduction

Sort algorithm is one of the fundamental techniques in computer science because of the following reasons. First, it is the basis of many other algorithms such as searching, pattern matching, digital filters etc, and many applications have been found in database systems, data statistics and processing, data communications and pattern matching[1]. Second, it plays an important role in the teaching of algorithm design and analysis, data structure and programming. Finally and especially, it is such a challenging problem that has been thoroughly studied[1−6], the performance is dramatically improved[7−10] and considered the lower-bound of complexity has been reached[1,2,11,12].

A formal description of sort can be defined based on *partial order*. As preparation the definition of partial order is given as following.

Def. 1. Suppose *R* is a relation on a set *S*. For *a, b, c* in *S,* if *R* is:

a) Reflexive, i.e. aRa for every a in S;
b) Transitive, i.e. $aRb \wedge bRc \Rightarrow aRc$; and
c) Antisymmetric, i.e. $aRb \wedge bRa \Rightarrow a = b$,

then, *R* is a *partial order* on set *S*.

Sort is generally defined as arranging a list of random data by their key or themselves into a partial order *R*, where *R* implies ≤ particularly.

Def. 2. For *n* elements $a_1, a_2, ..., a_n$ in set *S*, sort is an rearrangement of the elements in order to obtain a partial order $a_{s_i} R\, a_{s_{i+1}}$ for $\forall s_i,\ 1 \le s_i < n$. Generally, *R* is defined as ≤ in sort, so that the partial order is

$$a_{s_1} \le a_{s_2} \le , ..., \le a_{s_i} \le ,..., \le a_{s_n} \qquad (1)$$

Conventional internal sort algorithms can be divided into two categories. They are compare-based sort and radix-based sort.

Compare-based sort

Compare-based sort can be further categorised as basic sort and advanced sort.

***a. Basic sorting*:** The basic compare-based sort algorithms are such as the *selection* sort, *insertion* sort, *exchange* (or *bubbl*e) sort and etc. They are easy to implement and the complexities are in the order of $O(n^2)$.

The simple sort algorithms are still found their applications in some cases such as for small size of data, no care of sorting time and expecting easier to implementation, or under some particular cases or any partiality of users.

***b. Advanced sorting*:** The advanced sort algorithms based on comparison are *Quicksort* $(O (n \log_2 n))$[3], *heap* sort $(O (n \log_2 n))$[13], *Mergesort* $(O (n \log_2 n))$[1], *Shell* Sort $(O (n (\log_2 n)^2))$[1] and etc.

In this category, *Quicksort* is more advanced which is initiated by C.A.R. Hoare in 1961[3]. It is widely regarded as the best sort algorithm for decades[1,2,5,11], since it is up to twice as fast as its nearest competitors[2]. Many variations in implementation[2,9] have been developed so far for Quicksort in order to overcome its defects in seeking the reasonable partition strategy, in the performance stability under various cases and in the cost of recursive-calling in space and time.

Radix-based sort

Radix-based sort is another type of sort algorithm that carried out sorting cyclically based on the individual digits in the keys of data.

Suppose m be the radix of the keys, k be the number of digits or characters in the key, and n be the size of the data, then the time complexity of radix sorting is $O(k* (n+m))$ and the space complexity *is* $O(n * (m+1))$. In addition, heavy overhead could be required by the algorithm to extract the *i*th index[1,2].

Besides the above mentioned categories, a third type of sort algorithm, self-indexed sort (*SIS*), is developed in this paper by taking the key-value of the data as their address in the sorting space.

2. Self-indexed sort (SIS) algorithm

The *SIS* sort carries out sorting by directly mapping the element into a relative offset based on the value of its key. It is a new type of key-value-based sort. To implement *SIS* algorithm the conflict of identical keys in the data should be well considered and efficient post-processing of the scattered data has to be solved.

In this section, two versions of *SIS* algorithm are created and their mechanisms are illustrated. Then, the time and space complexities are analysed.

2.1 Algorithm of SIS

A general version of *SIS* algorithm is shown in the following for sorting a linked list of records. Sort is carried out in three steps by *SIS* algorithm: a) Initialisation of the sorting space; b) Self-indexed arrangement witch is a linear mapping based on the value of the keys in the sorted data; and c) Order-preserved compression during dumping the sorted but scatteredly located data elements into its original space.

Algorithm 1. *Self-Indexed Sorting*

```
Input:          Random list  X^ = (x_0 , x_1 , x_2 , ..., x_{n-1});
Output:         Sorted list  X^ = (x_{s_0} , x_{s_1} , x_{s_2} , ..., x_{s_{n-1}} );
Sorting space:  SS^ = array [r_0 .. r_{m-1}] of record
                                      key:      integer;
                                      field_{1-k}: anytype;
                                      link:     RecordPtr;
                                    end;
begin
  a) Initialisation
      new(SS);
      for j:=r_0  to  r_{m-1} do
        SS^[j].key := 0;

  b) Self-indexed arrangement
     for i:=0 to n-1 do
        if SS^[X^[i].key].key=0
          then begin
                 SS^[X^[i].key]:=X^[i];
                 SS^[X^[i].key].key:=1;
               end
          else begin
                 new ( node_i );        // inserting new node for the redundant records
                 SS^[X^[i].key].link:= node_i ;
                 node_i ^:= X^[i];
                 node_i ^.link := nil;
                 inc(SS^[X^[i].key].key);
               end;

  c) Order-preserved compression
     i:=0;
     for j:=r_0  to  r_{m-1} do    // r_i  is the relative offset address in SS^
        begin
          node_j ^:= SS^[j];
          while node_j ^[j].key > 0  do
             begin
               X^[i] := node_j ^;
               X^[i].key := j;
               node_j := node_j ^.link;
             end;
        end;
end.
```

2.2 Simplified version of SIS algorithm

A simplified version of *SIS* algorithm for helping to illustrate the concept of the method is derived in the following. It is also useful in the cases that the sorted data are a list of integers. When the data elements are characters, there are ways to transform them into integers by mapping or hashing. In other rare case, if the keys are real numbers, a key field of integer could be created according to the regular forms in database theories.

Algorithm 2. *Simplified Self-Indexed Sorting*

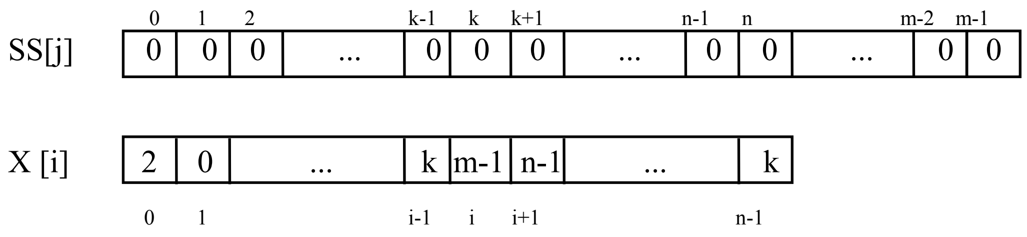
Input: Random list $X = (x_0, x_1, x_2, ..., x_{n-1})$;
Output: Sorted list $X' = (x_{s_0}, x_{s_1}, x_{s_2}, ..., x_{s_{n-1}})$;
Sorting space: $SS = (s_{r_0}, s_{r_1}, s_{r_2}, ..., s_{r_{m-1}})$, $m = max(x_i)$;

```
begin
  a) Initialisation

     for $j := r_0$ to $r_{m-1}$ do
       $SS[j] := 0$;

  b) Self-indexed arrangement

     for $i := 0$ to $n-1$ do
       inc($SS[X[i]]$);

  c) Order-preserved compression

     $i := 0$;
     for $j := r_0$ to $r_{m-1}$ do    // $r_j$ is the relative offset address in $SS$
       while $SS[j] > 0$ do
         begin
           $X[i] := j$;
           dec($SS[j]$);
           inc($i$);
         end;
end;
```

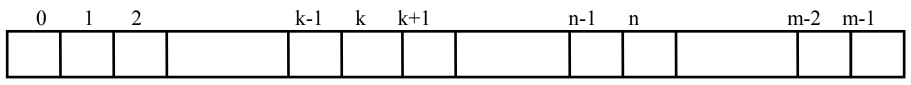
2.3 Illustration of SIS algorithm

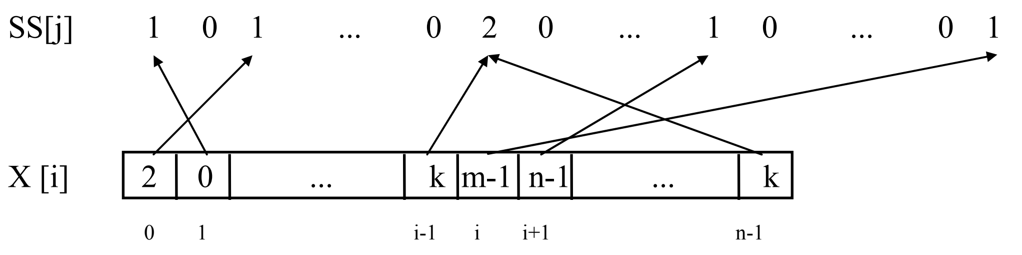
The algorithm of *SIS* sort employs three phases to implement the sort: initialisation, self-indexed arrangement and order-preserved compression.

For example, suppose an integer key list $X = (2, 0, ..., k, m-1, n-1, ..., k)$, $k<n$, $n<m-1$, it need to be sorted in the order of $X' = (0, 2, ..., k, k, n-1, ..., m-1)$. The procedures to sort it by *SIS* sorting are shown in Fig. 1.

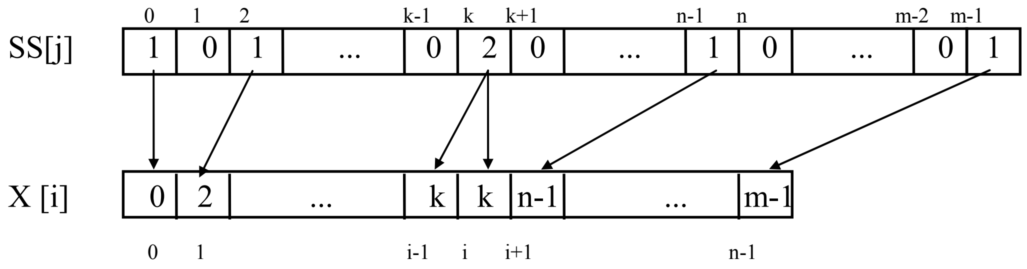


(a) *Initialisation*





(b) *Self-indexed arrangement*



(c) *Order-kept compression*

Fig.1 Illustration of SIS algorithm

The trick in step (b) is to increase the contents of the self-indexed elements by one rather than directly moving the *X[i]* into them. This method will show its advantages when there are redundant elements that have the same key values.

2.4 Analysis of SIS algorithm

2.4.1 Time complexity of SIS

The time cost of the algorithm is *2n* that consists of *n* times self-indexed allocation plus *n* times order-preserved compression. Therefore the time complexity *T(n)* can be obtained

$$T(n) = O(n) \qquad (2)$$

2.4.2. Space complexity of SIS

The space complexity of the algorithm is *n* plus *m*. So the space complexity *S(n)* of the algorithm is

$$S(n) = O(n+m) \qquad (3)$$

where m, $m=max(x_i)$, is the preallocated sorting space, and n is in proportion to the size of the sorted data.

2.4.3. Robustness of the algorithm

The complexity of *SIS* algorithm is quite stable under the conditions of average, worst and best cases. So the upper bounds of the algorithm is identical to the average expectation. These features will be shown in the next section.

3. Performance testing and analysis of SIS algorithm

3.1 Complexity comparison between typical sort algorithms

The comparison of complexity between *SIS* and conventional sort algorithms are listed in Table 1. It is observed that the lower-bound of sorts based on key compare is $O(n \log_2 n)$ and that Quicksort has reached the bound. But by adopting the key-value based non-comparing sorting as in *SIS* algorithm, the cost can be linear.

Table 1. Time complexity of typical sorting algorithms

Algorithm	Average case	Worst case
SIS sort	$O(n)$	$O(n)$
Quicksort	$O(n \log_2 n)$	$O(n^2)$
Shell sort	$O(n (\log_2 n)^2)$	$O(n^{3/2})$
Bubble sort	$O(n^2)$	$O(n^2)$

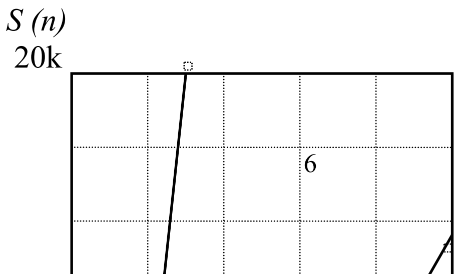
The other things need to be observed is the stability of the algorithms. In Table 1 one can find that some of the algorithms such as Quicksort and shell sort have non-stable performance in different cases. That means if you testing the performance of the algorithms you will find that the cost is dynamic at different execution, especially when the test case varying from average case to worst case. It can be shown from the algorithm 1 and 2 that the performance of *SIS* algorithm is stable based on its case free nature, and it also can be proved by the following test results.

3.2 Performance in average case

The performances of *SIS* and a set of typical sort algorithms are comparatively tested under average cases by using random test data from size *100* to *1000*. The result obtained is given in Table 2 and the curves are shown in Fig. 2.

Table 2. Testing result in average case

size (n)	SIS	Quick	Shell	Bubble
100	200	525	880	2,555
200	400	1,154	2,192	10,466
300	600	2,090	3,518	20,635
400	800	2,770	5,317	39,628
500	1,000	3,889	6,259	59,731
600	1,200	4,545	8,279	89,741
700	1,400	5,712	9,392	121,304
800	1,600	6,504	12,693	160,985
900	1,800	7,870	14,315	192,551
1,000	2,000	8,095	15,373	242,014



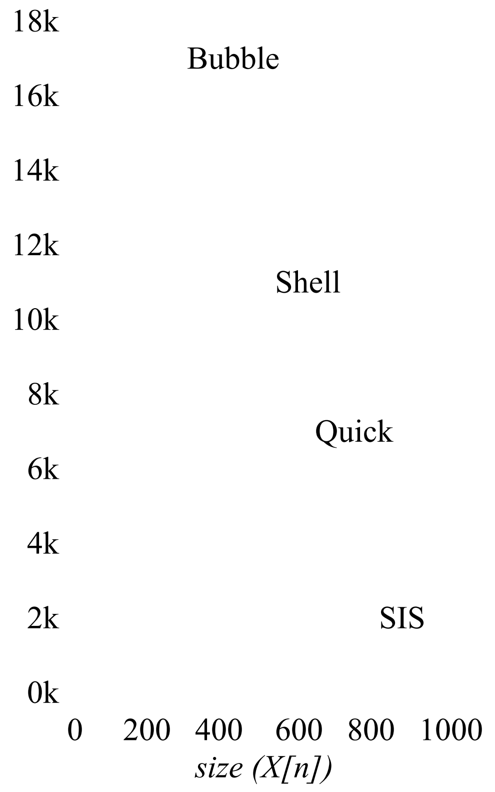


Fig.2 Performance comparison in average case

It is obvious in the test results that a linear and stable performance has been reached by *SIS* algorithm. As the size of data is *1000*, the speed of *SIS* sort is approximately *4* times fast than Quicksort, *7.5* times than shell sort and *121* times than bubble sort.

3.3 Performance in worst case

When the distribution of the data being sorted is in the worst case, i.e. it is just an inverse order against the expected partial order, the performances of the algorithms tested under this case are shown in Table 3 and illustrated in Fig. 3.

Table 3. Testing result in worst case

size (n)	SIS	Quick	Shell	Bubble
100	200	1,732	928	9,900
200	400	3,465	2,248	19,800
300	600	5,727	3,812	24,164
400	800	12,461	5,288	94,064
500	1,000	20,549	7,216	118,428
600	1,200	28,738	8,812	162,792
700	1,400	39,751	10,795	227,156
800	1,600	52,016	12,168	311,520
900	1,800	72,704	14,673	415,884
1,000	2,000	89,075	16,416	474,712

S (n)

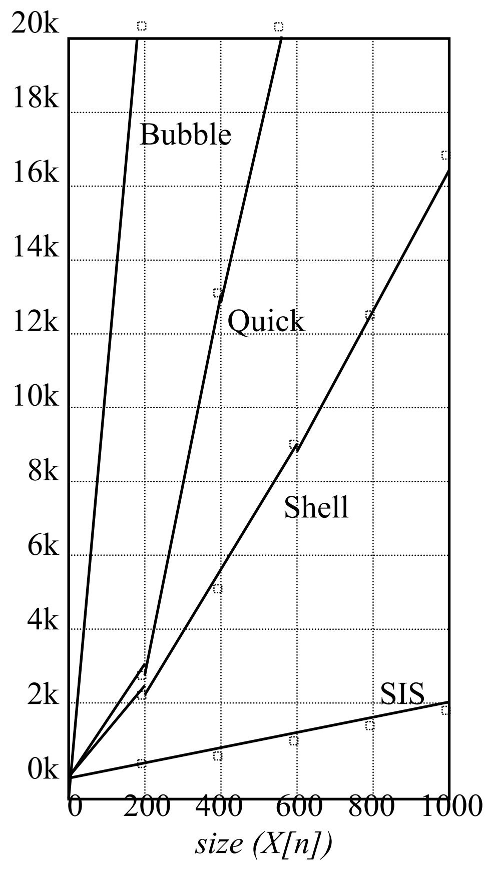


Fig. 3 Performance comparison in worst case

It is observed that *SIS* keeps the same speed under the worst case; but the other algorithms become worse in the case. In the worst condition, the speed of *SIS* sort, at *n=1000,* reaches 44.6 times fast than Quicksort, 8.3 times than shell sort and 237.4 times than bubble sort approximately. Comparing Table 2 and 3, it is also note-worthy that the stability of *SIS* algorithm.

4. Conclusions

The time complexity *O(n)* and space complexity *O(n+m)* are obtained by *SIS* algorithm. The advantage of the algorithm is speed, although a space of size *m*, $m = max(x_i)$, is required additionally so that the algorithm is more suitable to sort data with coherent integer keys.

It is well recognised that the choice of proper sort algorithm for specific application according to the features of data distribution, language and running environment[1,2,11]. Therefore many applications of *SIS* sort algorithm can be found in general purpose sorting and especially in the application of real-time database or information systems which concern more about speed.

Acknowledgements

This work is carried out at Oxford University Computing Laboratory. The author would like to thank Professor C.A.R. Hoare for his comments and encouragement, and to thank my colleagues Dr. P. Lin and Michael N. Zhu for their helpful discussions.

References

[1] D.E. Kunth, The Art of Computer Programming: Vol. 3, Sorting and Searching, 2nd printing, Addison-Wesley, Reading, MA, 1975.
[2] R. Sedgewick, Algorithms, 2nd ed., Addison-Wesley Series in Computer Science, pp. 111-113, 1988.
[3] C.A.R. Hoare, "Algorithm 64: Quicksort," Communications of the ACM, Vol. 4 , pp. 321, 1961.
[4] C.A.R. Hoare, "Quicksort," BCS Computer Journal, Vol. 5, No. 1, pp. 10-15, 1962.
[5] C.A.R. Hoare, Essays in Computer Science, Prentice-Hall, 1992.
[6] R. Sedgewick. Stanford University Ph.D. Dissertation, 1975.
[7] R. Sedgewick, "Implementing Quicksort Programs," Communications of the ACM, Vol.21, No. 10, pp. 847-856, 1978.
[8] R. Sedgewick. Quicksort, Garland, New York, 1980.
[9] B.C. Huang and D.E. Kunth, "A one-way, stackless quicksort algorithm," BIT 26, pp.127-130, 1986.
[10] J.P. Linderman. "Theory and Practice in the Construction of a Working Sort Routine." Bell System Technical Journal 63, pp.1827-1843, 1984.
[11] D.F. Stubbs and N.W. Webre, Data Structures with Abstract Data Type and Ada, PWS-KENT Publishing Company, pp.301-341, 1993.
[12] S. Baase, Computer Algorithms: Introduction to Design and Analysis, Addison-Wesley Publishing Co., pp. 58-132, 1978.
[13] J.W.J. Williams. "Algorithm 232: Heapsort," Communication of the ACM, No.7, pp.347-348, 1964.
[14] Yingxu Wang, On the Design of Self-Indexed Sorting Algorithm, Research Report 95-06-02, Oxford University Computing Laboratory, 1995.