NK-SORTING ALGORITHM

Assist .Prof. Dr. NIDHAL K. EL ABBADI, ZAID YAHYA A. KAREEM
University of Kufa, Iraq

nidhalka@it.kuiraq.com zaidak@it.kuiraq.com

ABSTRACT

Sorting has been a profound area for the algorithmic researchers and many resources are invested to suggest more works for sorting algorithms. For this purpose, many existing sorting algorithms were observed in terms of the efficiency of the algorithmic complexity.

Many algorithms are very well known for sorting the unordered lists.

In this proposed algorithm, we suggested a new algorithm for sorting integers number depending on dividing the input array to many sub-arrays (which represents a vector or array with one dimension), according to the number of digits in each integer number, the relation between sub-array elements is determined, and this relation used to determines the right location of each element in sub-arrays.

Collision may happen, which is solved by moving elements in sub-array to next location. Finally, all ordered sub-arrays will be merged together to rebuild the origin array. The proposed algorithm compared with many famous algorithms gives promising results.

Key Words: Sorting, Time complexity, Integers, Comparison, Time analysis, Space analysis.

1. Introduction

Sorting is the process of rearranging a sequence of objects so as to put them in some logical order, such as an alphabetic or numeric order. In the early days of computing, the common wisdom was that up to thirty percent of all computing cycles were spent in sorting. If that fraction is lower today, one likely reason is that sorting algorithms are relatively efficient; not that sorting has been diminished in relative importance. Indeed, the ubiquity of computer usage has put us awash in data, and the first step to organize data is often to sort it. All computer systems have implementations of sorting algorithms, for use by the system and by the users.

Sorting has been a profound area for the algorithmic researchers and many resources are invested to suggest more working sorting algorithms. For this purpose, many existing sorting algorithms were observed in terms of the efficiency of the algorithmic complexity [7]. Sorting plays a major role in commercial data processing and in modern scientific computing. Applications abound in transaction processing, combinatorial optimization, astrophysics, molecular dynamics, linguistics, genomics, weather prediction, and many other fields. As stated in [1], sorting has been considered as a fundamental problem in the study of algorithms, that due to many reasons:

- The need to sort information is inherent in many applications.
- Algorithms often use sorting as a key subroutine.

In algorithm design, there are many essential techniques represented in the body of sorting algorithms.

Many algorithms are very well known for sorting the unordered lists. Most important of them are:

Bubble sort [5], the idea is to make repeated passes up the array; "bubbling" the light ("light" means "large" or "small") key values to the top. On each pass the next lightest value will appear in the proper place. Assuming the array is indexed [0...n-1], we require (n − 1) passes to guarantee that the array is sorted. The bubbling process compares adjacent values and insures that the larger of the two is on top.

Quick sort [8], fundamentally, is based on a simple idea: Pick some key. Put all the records that have a smaller key than the selected key at the beginning of the array, and put all the records with larger keys at the end, then apply the same procedure recursively to each group of records, continuing until you get down to groups of size zero or one. However, organizing all this and doing it efficiently requires some cleverness.

Insertion sort [4], The algorithm that people often use to sort bridge hands is to consider the cards, one at a time, inserting each into its proper place among those which are already considered (keeping them sorted). In a computer implementation, we need to make space for the element being inserted by moving larger elements one position to the right, and then inserting the element into the vacated position

Selection sort [7], one of the simplest sorting algorithms works as follows: First, find the smallest element in the array, and exchange it with the element in the first position, then find the next smallest element and exchange it with the element in the second position. Continuing in this way until the entire array is sorted.

Efficient sorting is important to optimize the use of other algorithms that require sorted lists to work correctly. It is also often in producing human-readable output [6]. Formally, the output should satisfy two major conditions: The output is in non-decreasing order, and it is a permutation or reordering of the input.

In this paper, a new sorting algorithm is presented, called NK-Sort (the author's first name). The study shows that the proposed algorithm is more efficient and faster compared with many standard sorting algorithms when dealing with a large size (*n*) of the input integer array.

Section 2 presents the proposed algorithm, its concept, and steps. Section 3 introduces the detailed time and space analysis of the proposed algorithm. It also presents a comparison between the proposed algorithm and other sorting algorithms. Finally, conclusions were presented in section 4.

2. Proposed Algorithm

The main concept of the proposed algorithm is distributing the elements of the input array (unordered list of integers) on many additional temporary sub-arrays according to a number of digits in each number. The size of each of these sub-arrays are decided depending on a number of elements with the same number of digits in the input array.

2.1 The steps of proposed algorithm

Algorithm consists of many steps to accomplish the sorting of all integers in input array as follow:

A. Step 1:

The first step is to build many temporary sub arrays from input array, the input array scan and distribute their elements on sub arrays according to a number of digits in each element. Therefore, the integers with one digit (0..9) puts in sub array (1), and integers with two digits (10..99) puts in sub array (2), and the integers with three digits (100..999) puts in sub array (3), and so on. The number of sub arrays depends on number of elements with different number of digits. The size of each sub array depends on the number of elements with the same number of digits.

Let (A) set (input array) of (N) integer numbers, so

$$A = \bigvee_{i=1}^{j} A_i j = \text{number of sub-array } (A_i) \quad .(1)$$

Where A_i is a subset (sub-array) of A, such that each element (integer number) in (A_i) consists of (i) digit. And size of (A_i) (number of elements in A_i) equals (m_i), where

$$N = \sum_{i}^{j} m_i \quad \text{.....} \quad (2)$$

B. Step2:

For each (A_i), find a maximum and minimum element value and a number of elements in sub array (i).

C. Step3:

Imagine the sub-array (A_i) as two dimension graphs as shown in figure (1):

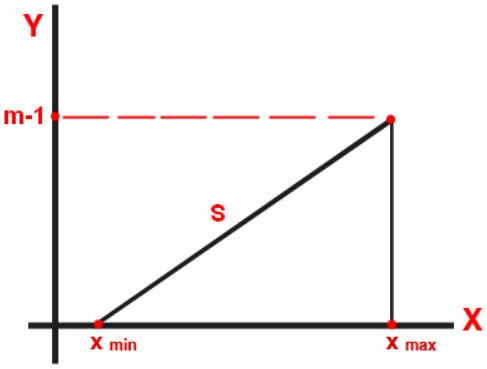

Figure (1): relation between array values and its location in sub array

Where: X-coordinate, represent the elements (integer) values.

Y-coordinate, represent the locations in sub-array start from location (0) to location (m_i-1) OR (m_i) if we start with location (1),

From figure 1, find the slope (S) which will be use to find the relation between elements value and its locations in the sub array .In this case, each element between (X_{max}, and X_{min}) can project on graph (figure 1) with assistance of (S) to find the corresponding location in the sub array.

$$S = DY / DX \quad \text{.....} \quad (3)$$

Where: DY = m-1 And

DX = X_{max} - X_{min}

Now for each sub-array find (S_i).

(*Note*: always check Dx not equal zero, when Dx equal zero that mean all the sub array elements are equal (or may be one element) and not need to sort.).

Step4:

For each sub-array, its elements are ordered by finding the locations of elements in the sub array according to their values, with assistance of equation:

location = element value * S ..... (4)

E. Step5:

If the determined location for an element, was occupied or assigned to other element, the location would increase or decrease by one (according to sorting way ascending, or descending), and check the new location if it was assigned or not .If it is empty it will be assigned to an element. Otherwise, it should compare the values to decide which one appropriate to this location, and the other element move to the next location, and so on.

D. Step6:

When each sub array now is sorted, the final step is to append all sub arrays with each other according to the number of digits for sub array elements (i.e. sub array with one digit element at first, and sub array with two digit elements append to it, and so on).

$A = A_1 \ Y \ A_2 \ Y \ A_3 Y \ldots\ldots Y \ A_j$ ..... (5)

Let's take an example to clarify the above steps:

Example:

As example: let take set A (42) of integer's numbers as follow:

A = 816, 657, 243, 453, 76, 98, 4567, 123, 25, 3, 7, 1, 3456, 157, 23, 0, 2345, 678, 4, 12, 7, 1, 90, 111, 1234, 678, 2345, 10, 23, 45, 456, 123, 9870, 345, 1, 6, 4, 15, 234, 3456, 3111, 98.

Step1: Build sub-arrays (OR sub-sets) from set (A) according to the number of digits:

- *Sub-array with one digit…*

A_1 = {3, 7, 1, 0, 4, 7, 1, 1, 6, 4} (10 elements)

- *Sub-array with two digit…*

A_2 = {76, 98, 25, 23, 12, 90, 10, 23, 45, 15, 98} (11 elements)

- *Sub-array with three digit …*

A_3 = {816, 657, 243, 453, 123, 157, 678, 111, 678, 456, 123, 345, 234} (13 elements)

- *Sub-array with four digit …*

A_4 = {4567, 3456, 2345, 1234, 2345, 9870, 3456, 3111} (8 elements)

➢ *We will process the first sub-array (A_1) to sort its elements, and the rest sub-arrays follow the same way*

The first sub-array consists of (10 element, each with one digit). Then,

Max number in the A_1 = 7

Min number in the A_1 = 0

Therefore, **DY** =10 – 1 = 9, and **DX** = 7 – 0 = 7

S = DY / DX = 9 / 7 = 1.285

Let us check locations of elements in A_1 (location= S × value of element):

Locations	0	1	2	3	4	5	6	7	8	9
Elements										

Location of element (3) in A_1 =

$4 \approx 1.285 \times 3 = 3.8$

Locations	0	1	2	3	4	5	6	7	8	9
Elements					3					

Location of second element (7) in A_1 = 1.285× 7= 9

Locations	0	1	2	3	4	5	6	7	8	9
Elements					3					7

Location of third element (1) in A_1 =

$1.285\times 1 = 1.285 \approx 1$

Locations	0	1	2	3	4	5	6	7	8	9
Elements		1			3					7

Location of fourth element (0) in $A_1 = 1.285\times 0 = 0$

Locations	0	1	2	3	4	5	6	7	8	9
Elements	0	1			3					7

Location of fifth element (4) in $A_1 = 1.285\times 4 = 5.142 \approx 5$

Locations	0	1	2	3	4	5	6	7	8	9
Elements	0	1			3	4				7

Location of sixth element (7) in $A_1 = 1.285\times 7 = 9$

In this case, the location (9) is occupied with element (7), the new element (7) is compared with the element in location (9), in this case they are equal,

Locations	0	1	2	3	4	5	6	7	8	9
Element	0	1			3	4			7	7

and so program put it in the next location (8)

The location of the seventh element (1) in A_1 = $1.285\times 1 = 1.285 \approx 1$

In this case, the location is occupied with element (1) and it is compared with the new element, they also are equal, and so program put it in the next location, here, location (2)

Locations	0	1	2	3	4	5	6	7	8	9
Elements	0	1	1		3	4			7	7

The location of the eighth element (1) in A_1 = $1.285\times 1 = 1.285 \approx 1$

The same case as above so the element moves to next location (2), it is also occupied and the same thing is done, compare and move it to next location (3).

Locations	0	1	2	3	4	5	6	7	8	9
Elements	0	1	1	1	3	4			7	7

The location of the ninth element (6) in $A_1 = 1.285\times 6 = 7.714 \approx 8$

Location (8) is occupied with element (7), it compared with new element (6), it is found that the new element is less than element in location (8), and due to ascending order, then, the small element should be on left side of the largest one ,then, the location is (7).

Locations	0	1	2	3	4	5	6	7	8	9
Elements	0	1	1	1	3	4		6	7	7

The location of last element (4) in $A_1 = 1.285\times 4 = 5.142 \approx 5$

The location (5) is occupied and at the same way its new location is (6)

Locations	0	1	2	3	4	5	6	7	8	9
Elements	0	1	1	1	3	4	4	6	7	7

By the same way all the other sub-arrays (A_2, A_3, and A_4) are sorted to get

A_2

L	0	1	2	3	4	5	6	7	8	9	10
E	10	12	15	23	23	25	45	76	90	98	98

A_3

L	0	1	2	3	4	5	6	7	8	9	10	11	12
E	111	123	123	157	234	243	345	453	456	657	678	678	816

A_4

L	0	1	2	3	4	5	6	7
E	1234	2345	2345	3111	3456	3456	4567	9870

Now all sub-arrays are connected with each other according to the number of digits in each sub-array that begin with the smallest number of the digit: A_1+A_2+ A_3+ A_4

The final result becomes:

A = 0, 1, 1, 1, 3, 4, 4, 6, 7, 7, 10, 12, 15, 23, 23, 25, 45, 76, 90, 98, 98, 111, 123, 123, 157, 234, 234, 345, 453, 456, 657, 678, 678, 816, 1234, 2345, 2345, 3111, 3456, 3456, 4567, 9870.

3. Analysis of Proposed Algorithm

Comparisons are the heart of sorting, we could ask: "How many comparisons does this algorithm make in the process of sorting?" We could then suggest that the algorithm that required fewer comparisons was the fastest. There are some factors that make it only a rough estimate:

• We did not include relocation overhead—somehow items must be repositioned to obtain the sorted list.

• We did not include miscellaneous overhead such as initialization and subroutine calls. We will ignore such problems and just look at the number of comparisons. Even so, there are problems:

• Using the parallel processing capabilities of supercomputers or special purposed devices will throw time estimates off because more than one comparison can be done at a time. The amount of parallelism that is possible can vary from one algorithm to another.

• The number of comparisons needed may vary greatly, depending on the order of the items in the unsorted list. Some studies ignore these factors in the discussion, except for parallelism in sorting networks, where it is of a major importance.

Besides all these problems with estimating running time, there is another problem: Running time is not the only standard that can be used to decide how good an algorithm is. Other important questions include

• How long will it take to get an error free program running?

• How much storage space will the algorithm require?

All studies ignore these issues and focus on running time.

3.1. Time Analysis

The goal of proposed program is to get the sub-arrays, maximum and minimum value and the number of elements in each sub-array needed to sort it. This requires scanning the array and reaching each element one

time in a single pass in the worst case; this takes **O (n)** time complexity, see table 1.

Table (1): Comparing Time Complexity of proposed algorithm with many standard sort algorithms.

The Algorithms	Time Complexity	
	Worst Case	Best Case
Bubble Sort	$O(n^2)$	O(n)
Insertion Sort	$O(n^2)$	O(n)
Selection Sort	$O(n^2)$	$O(n^2)$
Shell Sort	depends on gap sequence	O(n)
Merge Sort	O(n log n)	O(n log n) typical, O(n) natural variant
Quick Sort	O(n2)	O(n log n)
Heap Sort	O(n log n)	O(n log n)
Proposed Algorithm	O(n)	O(n)

3.2 Comparison of proposed algorithm with other algorithms

The proposed algorithm compared with many standard algorithms, like (Bubble sort, Selection sort, Insertion sort, Quick sort, Shell sort, Merge sort, and Heap sort). *Note: all algorithms execute at the same computer (fixed environments for all),* the proposed algorithm gave good time better than the other sort algorithms, the difference in time between algorithms increased with increasing the number of integers sorted. Figure 2, and table 2 show the difference in time for sort algorithms.

Table 2: Time elapsed for sorting integers for different sort algorithms

Number of Integers Sorted / *The Algorithms*	80	200	400	600	800	1000	2000	4000	8000	10
Proposed Algorithm	317	347	737	1312	1491	1536	3058	6109	12270	18
Bubble Sort	327	449	991	1497	1768	1889	3713	7244	14522	19
Selection	600	761	1470	1979	2324	3103	4802	9690	17300	24
Insertion	560	843	1182	1339	1576	1862	3690	7205	14411	19
Quick Sort	409	497	776	1102	1700	2579	3642	6310	12628	20
Shell Sort	390	482	756	1090	1689	2545	3512	6300	12732	20
Merge Sort	427	524	742	1231	1688	2625	3825	7180	13280	20
Heap Sort	485	520	771	1126	1721	2598	3904	6625	12800	20

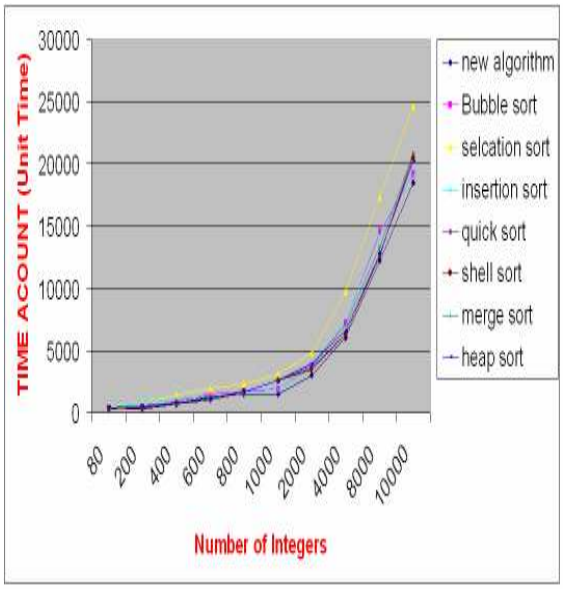

Figure (2): Graph shows the time elapsed for sorting different sets of integers, for different algorithms

3.3 Space Analysis of the proposed algorithm

The algorithm creates many sub-arrays from the original array; the size of all these sub-arrays is equal to the size of the original array. When creating sub-arrays the original array is not needed any more space. In all cases, the

algorithm does not need the space more than space of the original array. So, in any case the algorithm needs O (n) space. And may need (3 additional spaces for max, min, and S for each sub-array), which can neglect it with increasing number of integers in array. Additional feature, all the sub-arrays can be stored in the secondary storage and only one of the sub-arrays can be put in an active memory in one time to sort its elements, and after that the sort can be swapped with other one (in case of very large input array).

3.4 Number of comparisons

The proposed algorithm is compared with the other algorithms to see number of element comparisons in execution time, figure 3 showed the result.

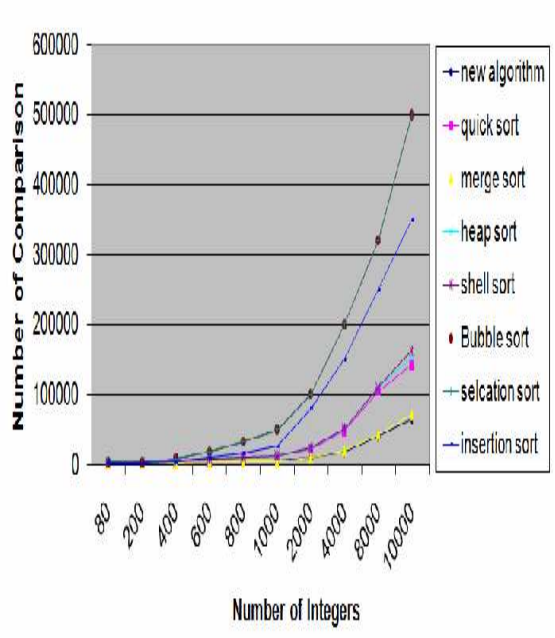

Figure (3): Number of comparisons for many sort algorithms.

Note new algorithm and merge sort are almost had the same number of comparison.

4. Conclusion

In this paper a new sorting algorithm is presented, called the NK-algorithm, the algorithm divided the input array to many sub-arrays, that will reduce the comparison process (number of comparisons of its elements) see figure 3, (merge sort typically makes almost equal number of comparisons with proposed algorithm, but merge sort requires more writing because the inner loop can require shifting large sections of the sorted portion of the array, and that caused increase in the elapsed time for sorting. In general, merge sort will write to the array O (*n logn*) times [10], whereas proposed Algorithm will write only O (*n*) times).

Proposed algorithm compared with many algorithms (Bubble sort, Insertion sort, Selection sort, Quick sort, Shell sort, Merge sort, and Heap sort), it was more efficient and faster. The more array size increases, the more execution time increases with a specific number of unit times for all algorithms, but this amount of time decreases with the increasing size more than 600 elements for the Proposed algorithm, see figure 2, this mean the Proposed algorithm more efficient with large array size (n) of the input array.

Algorithm is very suited for external sorting, because it divides the array to many sub arrays which can be saved externally, and sort one of them each time, that means one partition each time in memory. It is possible to process all the sub arrays in parallel, and that will reduce sorting time in the significant amount (sort time in this case depends on the size of larger sub-array).

The maximum and minimum value for each sub-array can be relocated initially in the first and last locations of the sub-array without determining its locations .This may decrease the sort time a little bit.

References

[1] Aho A., Hopcroft J., and Ullman J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Box R. and Lacey S., "*A fast, easy sort*," Byte Magazine, vol.16, no.4, pp.315-ff, April 1991.

[3] Cormen T., Leiserson C., Rivest R., and Stein C., *Introduction to Algorithms*, 2nd edition, McGraw-Hill Book, 2001.

[4] Donald Knuth. *The Art of Computer Programming*, Volume 3: Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 5.2.1: Sorting by Insertion, pp.80–105

[5] Kruse R., and Ryba A., *Data Structures and Program Design in C++*, International Edition, Prentice Hall, 1999.

[6] Levitin A., *Introduction to the Design & Analysis of Algorithms*, 2nd Edition, Section 3.1: Selection Sort, pp 98-100, 2007.

[7] Moller F., *Analysis of Quicksort*, Department of Computer Science, University of Wales Swansea, 2001.

[8] Nyhoff L., *An introduction to Data Structures*, 2nd edition, pp: 581-585.

[9] Thomas H. , et at. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 2.1: Insertion sort, pp.15–21.

[10] Weiss M., *Data Structures and Problem Solving using Java*, Addison-Wesley, 2002.