Future Generation Computer Systems 71 (2017) 102–112

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort

Adnan Saher Mohammed [a,*], Şahin Emrah Amrahov [b], Fatih V. Çelebi [c]

[a] Ankara Yıldırım Beyazıt University, Graduate School of Natural Sciences, Computer Engineering Department, Ankara, Turkey
[b] Ankara University, Faculty of Engineering, Computer Engineering Department, Ankara, Turkey
[c] Ankara Yıldırım Beyazıt University, Faculty of Engineering and Natural Sciences, Computer Engineering Department, Ankara, Turkey

HIGHLIGHTS

- We propose a new efficient sorting algorithm Bidirectional Conditional Insertion Sort (BCIS).
- We compare BCIS with insertion sort and quicksort.
- We prove that the average time complexity of BCIS very close to $O(n^{1.5})$ for normally or uniformly distributed data.

ARTICLE INFO

Article history:
Received 8 August 2016
Received in revised form 5 December 2016
Accepted 30 January 2017
Available online 31 January 2017

Keywords:
Insertion sort
Sorting
Quicksort
Bidirectional insertion sort
BCIS

ABSTRACT

In this paper, we proposed a new efficient sorting algorithm based on insertion sort concept. The proposed algorithm is called Bidirectional Conditional Insertion Sort (BCIS). It is in-place sorting algorithm and it has remarkably efficient average case time complexity when compared with classical insertion sort (IS). By comparing our new proposed algorithm with the Quicksort algorithm, BCIS indicated faster average case time for relatively small size arrays up to 1500 elements. Furthermore, BCIS was observed to be faster than Quicksort within high rate of duplicated elements even for large size array.

1. Introduction

Algorithms have an important role in development process of computer science and mathematics. Sorting is a fundamental process in computer science which is commonly used for canonicalizing data. In addition to the main job of sorting algorithms, many algorithms use different techniques to sort lists as a prerequisite step to reduce their execution time [1]. The idea behind using sorting algorithms by other algorithm is commonly known as reduction process. A reduction is a method for transforming one problem to another easier than the first problem [2]. Consequently, the need for developing efficient sorting algorithms that invest the remarkable development in computer architecture has increased.

Sorting is generally considered to be the procedure of repositioning a known set of objects in ascending or descending order according to specified key values belonging to these objects. Sorting is guaranteed to finish in finite sequence of steps [3].

Among a large number of sorting algorithms, the choice of which is the best for an application depends on several factors like size, data type and the distribution of the elements in a data set. Additionally, there are several dynamic influences on the performance of the sorting algorithm which can be briefed as the number of comparisons (for comparison sorting), number of swaps (for in-place sorting), memory usage and recursion [4].

Generally, the performance of algorithms is measured by the standard Big $O(n)$ notation which is used to describe the complexity of an algorithm. Commonly, sorting algorithms has been classified into two groups according to their time complexity. The first group is $O(n^2)$ which contains the insertion sort, selection sort, bubble sort etc. The second group is $O(n \log n)$, which is faster than the first group, includes Quicksort, merge sort and heap sort [5]. The insertion sort algorithm can be considered as one of the best algorithms in its family ($O(n^2)$ group) due to its performance, stable algorithm, in-place, and simplicity [6]. Moreover, it is the fastest algorithm for small size array up to 28–30 elements compared to the Quicksort algorithm. That is why it has been used in conjugate with Quicksort [7–10].

* Corresponding author.
E-mail addresses: adnshr@gmail.com (A.S. Mohammed), emrah@eng.ankara.edu.tr (Ş.E. Amrahov), fvcelebi@ybu.edu.tr (F.V. Çelebi).

http://dx.doi.org/10.1016/j.future.2017.01.034
0167-739X/

Several improvements on major sorting algorithms have been presented in the literature [11–13]. Chern and Hwang [14] give an analysis of the transitional behaviors of the average cost from insertion sort to quicksort with median-of-three. Fouz et al. [15] provide a smoothed analysis of Hoare's algorithm who has found the quicksort. Recently, we meet some investigations of the dual-pivot quicksort which is the modification of the classical quicksort algorithm. In the partitioning step of the dual-pivot quicksort two pivots are used to split the sequence into three segments recursively. This can be done in different ways. Most efficient algorithm for the selection of the dual-pivot is developed due to Yaroslavskiy question [16]. Nebel, Wild and Martinez [17] explain the success of Yaroslavskiy's new dual-pivot Quicksort algorithm in practice. Wild and Nebel [18] analyze this algorithm and show that on average it uses $1.9n\,ln\,n+O(n)$ comparisons to sort an input of size n, beating standard quicksort, which uses $2n\,ln\,n\,+\,O(n)$ comparisons. Aumüller and Dietzfelbinger [19] propose a model that includes all dual-pivot algorithms, provide a unified analysis, and identify new dual-pivot algorithms for the minimization of the average number of key comparisons among all possible algorithms. This minimum is $1.8n\,ln\,n\,+\,O(n)$. Fredman [20] presents a new and very simple argument for bounding the expected running time of Quicksort algorithm. Hadjicostas and Lakshmanan [21] analyze the recursive merge sort algorithm and quantify the deviation of the output from the correct sorted order if the outcomes of one or more comparisons are in error. Bindjeme and Fill [22] obtain an exact formula for the L2-distance of the (normalized) number of comparisons of Quicksort under the uniform model to its limit. Neininger [23] proves a central limit theorem for the error and obtain the asymptotics of the L_3*–distance*. Fuchs [24] uses the moment transfer approach to re-prove Neininger's result and obtains the asymptotics of the L_p*–distance* for all $1 \leq p < \infty$.

Grabowski and Strzalka [25] investigate the dynamic behavior of simple insertion sort algorithm and the impact of long-term dependencies in data structure on sort efficiency. Biernacki and Jacques [26] propose a generative model for rank data based on insertion sort algorithm. The work that presented in [27] is called library sort or gapped insertion sort which is trading-off between the extra space used and the insertion time, so it is not in-place sorting algorithm. The enhanced insertion sort algorithm that presented in [28] is use approach similar to binary insertion sort in [29], whereas both algorithms reduced the number of comparisons and kept the number of assignments (shifting operations) equal to that in standard insertion sort $O(n^2)$. Bidirectional insertion sort approaches presented in [3,30]. They try to make the list semi sorted in Pre-processing step by swapping the elements at analogous positions (position 1 with n, position 2 with $(n\,-\,1)$ and so on). Then they apply the standard insertion sort on the whole list. The main goal of this work is only to improve worst case performance of IS [30]. On other hand, authors in [6] presented a bidirectional insertion sort, firstly exchange elements using the same way in [3,30], then starts from the middle of the array and inserts elements from the left and the right side to the sorted portion of the main array. This method improves the performance of the algorithm to be efficient for small arrays typically of size lying from 10–50 elements [6]. Finally, the main idea of the work that presented in [31], is based on inserting the first two elements of the unordered part into the ordered part during each iteration. This idea earned slightly time efficient but the complexity of the algorithm still $O(n^2)$ [31]. However, all the cited previous works have shown a good enhancement in insertion sort algorithm either in worst case, in large array size or in very small array size. In spite of this enhancement, a Quicksort algorithm indicates faster results even for relatively small size array.

In this paper, a developed in-place unstable algorithm is presented that shows fast performance in both relatively small size array and for high rate duplicated elements array. The proposed algorithm Bidirectional Conditional Insertion Sort (BCIS) is well analyzed for best, worst and average cases. Then it is compared with well-known algorithms which are classical Insertion Sort (IS) and Quicksort. Generally, BCIS has average time complexity very close to $O(n^{1.5})$ for normally or uniformly distributed data. In other word, BCIS has faster average case than IS for both relatively small and large size array. Additionally, when it is compared with Quicksort, the experimental results for BCIS indicates less time complexity up to 70%–10% within the data size range of 32–1500. Besides, our BCIS illustrates faster performance in high rate duplicated elements array compared to the Quicksort even for large size arrays. Up to 10%–50% is achieved within the range of elements of 28-more than 3000,000. The other pros of BCIS that it can sort equal elements array or remain equal part of an array in $O(n)$.

This paper is organized as follows: Section 2 presents the proposed algorithm and pseudo code, Section 3 executes the proposed algorithm on a simple example array, Section 4 illustrates the detailed complexity analysis of the algorithm, Section 5 discusses the obtained empirical results and compares them with other well-known algorithms, Section 6 provides conclusions. Finally, you will find the important references.

2. The proposed algorithm BCIS

The classical insertion sort explained in [31–33] has one sorted part in the array located either on left or right side. For each iteration, IS inserts only one item from unsorted part into proper place among elements in the sorted part. This process repeated until all the elements sorted.

Our proposed algorithm minimizes the shifting operations caused by insertion processes using new technique. This new technique supposes that there are two sorted parts located at the left and the right side of the array whereas the unsorted part located between these two sorted parts. If the algorithm sorts ascendingly, the small elements should be inserted into the left part and the large elements should be inserted into the right part. Logically, when the algorithm sorts in descending order, insertion operations will be in reverse direction. This is the idea behind the word 'bidirectional' in the name of the algorithm.

Unlike classical insertion sort, insertion items into two sorted parts helped BCIS to be cost effective in terms of memory read/write operations. That benefit happened because the length of the sorted part in IS is distributed to the two sorted parts in BCIS. The other advantage of BCIS algorithm over classical insertion sort is the ability to insert more than one item in their final correct positions in one sort trip (internal loop iteration).

Additionally, the inserted items will not suffer from shifting operations in later sort trips. Alongside, insertion into both sorted sides can be run in parallel in order to increase the algorithm performance (parallel work is out of scope of this paper).

In case of ascending sort, BCIS initially assumes that the most left item at *array*[1] is the left comparator (LC) where is the left sorted part begin. Then inserts each element into the left sorted part if that element less than or equal to the LC. Correspondingly, the algorithm assumes the right most item at *array*[n] is the right comparator (RC) which must be greater than LC. Then BCIS inserts each element greater than or equal to the RC into the right sorted part. However, the elements that have values between LC and RC are left in their positions during the whole sort trip. This conditional insertion operation is repeated until all elements inserted in their correct positions.

If the LC and RC already in their correct position, there are no insertion operations occur during the whole sort trip. Hence, the algorithm at least places two items in their final correct position for each iteration.