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.

In the pseudo code (part 1 & 2), the BCIS algorithm is presented in a format uses functions to increase the clarity and traceability of the algorithm. However, in statements (6 & 7) the algorithm sets two indexes, SL indicates on the beginning of the Sorted Left side and SR indicates on the beginning of the Sorted Right side.

Algorithm BCIS Part 1 (Main Body)

```
1:  procedure BCIS( array, left, right )
2:      array is the array that required to sort
3:      left is the index of left most element in array
4:      right is the index of right most element in array
5:      array[i] is in-process element
6:      SL ← left                                  ▷ Sorted Left part index
7:      SR ← right                                 ▷ Sorted Right part index
8:      while SL < SR do
9:          SWAP(array, SR, SL + (SR−SL)/2)
10:         if array[SL] = array[SR] then          ▷ equality check
11:             if ISEQUAL(array, SL, SR)=-1 then
12:                 return
13:             end if
14:         end if
15:         if array[SL] > array[SR] then
16:             SWAP (array, SL , SR)              ▷ swap if LC > RC
17:         end if
18:         if (SR − SL) ≥ 100 then
19:             for i ← SL + 1 to (SR − SL)^0.5 do
20:                 if array[SR] < array[i] then
21:                     SWAP (array, SR, i)
22:                 else if array[SL] > array[i] then
23:                     SWAP (array, SL, i)
24:                 end if
25:             end for
26:         else
27:             i ← SL + 1
28:         end if
29:         LC ← array[SL]
30:         RC ← array[SR]
31:         while i < SR do
32:             CurrItem ← array[i]
33:             if CurrItem ≥ RC then
34:                 array[i] ← array[SR − 1]
35:                 INSRIGHT(array, CurrItem, SR, right)
36:                 SR ← SR − 1
37:             else if CurrItem ≤ LC then
38:                 array[i] ← array[SL + 1]
39:                 INSLEFT(array, CurrItem, SL, left)
40:                 SL ← SL + 1
41:                 i ← i + 1
42:             else
43:                 i ← i + 1
44:             end if
45:         end while
46:         SL ← SL + 1
47:         SR ← SR − 1
48:     end while
49: end procedure
```

Initially, SL and SR set to indicate on the most left element and the most right element respectively. The main loop starts at statements (8) and stops when the left sorted part index (SL) reaches the right sorted part index (SR).

The selection of LC and RC is processed by the statements (9–30). In order to ensure the correctness of the insertion operations LC must be less than RC, this condition processed in statement (15). In case of LC equal to RC, the statement (11), using ISEQUAL function, tries to find an item not equal to LC and replace it with

Algorithm BCIS Part 2 (Functions)

```
1:  function ISEQUAL(array, SL, SR)
2:      for k ← SL + 1 to SR − 1 do
3:          if array[k]! = array[SL] then
4:              SWAP(array, k, SL)
5:              return k
6:          end if
7:      end for
8:      return − 1
9:          ▷ End the algorithm because all scanned items are equal
10: end function
```

```
1:  function InsRight(array, CurrItem, SR, right)
2:      j ← SR
3:      while j ≤ right and CurrItem > array[j] do
4:          array[j − 1] ← array[j]
5:          j ← j + 1
6:      end while
7:      Array[j − 1] ← CurrItem
8:  end function
```

```
1:  function InsLeft(array, CurrItem, SL, left)
2:      j ← SL
3:      while j ≥ left and CurrItem < array[j] do
4:          array[j + 1] ← array[j]
5:          j ← j − 1
6:      end while
7:      Array[j + 1] ← CurrItem
8:  end function
```

```
1:  function SWAP(array,i,j)
2:      Temp ← array[i]
3:      array[i] ← array[j]
4:      array[j] ← Temp
5:  end function
```

LC. Otherwise, (if not found) all remaining elements in the unsorted part are equal. Thus, the algorithm should terminate at the statement (12). Furthermore, this technique allows equal elements array to sort in only $O(n)$ time complexity. Statements (9 & 18–25) do not have an effect on the correctness of the algorithm, these statements are added to enhance the performance of the algorithm. The advantage of these techniques will be discussed in the analysis section (Section 4).

The **while** statement in (31) is the beginning of the sort trip, as mentioned previously, conditional insertions occur inside this loop depending on the value of current item (*CurrItem*) in comparison with the values of LC and RC. Insertion operations are implemented by calling the functions INSRIGHT and INSLEFT.

3. Example

The behavior of the proposed algorithm on an array of 15 elements generated randomly by computer is explained in Fig. 1. In order to increase the simplicity of this example, we assumed the statements (9 & 18–25) do not exist in the algorithm. For all examples in this paper we assumed as follows: Items in red color mean these items are currently in process. Bolded items represent LC and RC for current sort trip. Gray background means the position of these items may change during the current sort trip. Finally, items with green background mean these items are in their final correct positions.

22	17	56	57	52	59	80	78	73	19	53	28	65	72	67	Check LC and RC, no swap

First sort trip starts here

22	17	56	57	52	59	80	78	73	19	53	28	65	72	67	Insert into the left
17	22	56	57	52	59	80	78	73	19	53	28	65	72	67	No insertion
17	22	56	57	52	59	80	78	73	19	53	28	65	72	67	No insertion
17	22	56	57	52	59	80	78	73	19	53	28	65	72	67	No insertion
17	22	56	57	52	59	80	78	73	19	53	28	65	72	67	No insertion
17	22	56	57	52	59	80	78	73	19	53	28	65	72	67	Insert into the right
17	22	56	57	52	59	72	78	73	19	53	28	65	67	80	Insert into the right
17	22	56	57	52	59	65	78	73	19	53	28	67	72	80	No insertion
17	22	56	57	52	59	65	78	73	19	53	28	67	72	80	Insert into the right
17	22	56	57	52	59	65	28	73	19	53	67	72	78	80	No insertion
17	22	56	57	52	59	65	28	73	19	53	67	72	78	80	Insert into the right
17	22	56	57	52	59	65	28	53	19	67	72	73	78	80	No insertion
17	22	56	57	52	59	65	28	53	19	67	72	73	78	80	Insert into the left

End of first sort trip.

17	19	22	57	52	59	65	28	53	56	67	72	73	78	80	Check LC and RC, swap

Second sort trip starts here

17	19	22	56	52	59	65	28	53	57	67	72	73	78	80	Insert into the left
17	19	22	52	56	59	65	28	53	57	67	72	73	78	80	Insert into the right
17	19	22	52	56	53	65	28	57	59	67	72	73	78	80	Insert into the left
17	19	22	52	53	56	65	28	57	59	67	72	73	78	80	Insert into the right
17	19	22	52	53	56	28	57	59	65	67	72	73	78	80	Insert into the left

End of second sort trip, all items has been sorted

17	19	22	28	52	53	56	57	59	65	67	72	73	78	80

Fig. 1. BCIS example.

4. Analysis of the proposed algorithm

The complexity of the proposed algorithm mainly depends on the complexity of insertion functions which is in turn depends on the number of inserted elements in each function during each sorting trip. To explain how the performance of BCIS depends on the number of inserted element per each sort trip, several assumptions are presented which revealed theoretical analysis very close to experiential results that we obtained.

In order to simplify the analysis, we will concentrate on the main parts of the algorithm. Assume that during each sort trip (k) elements are inserted into both sides, each side get $k/2$. Whereas insertion functions work exactly like standard insertion sort. Consequently, time complexity of each sort trip equal to the sum of the left and right insertion function cost which is equal to $T_{is}(k/2)$ for each function, in addition to the cost of scanning of the remaining elements (not inserted elements). We can express this idea as follows:-

$$\begin{aligned} T(n) = {} & T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) + 2(n-k) \\ & + T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) + 2(n-2k) \\ & + T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) + 2(n-3k) \\ & + \cdots + T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) + 2(n-ik). \end{aligned}$$

BCIS stops when $n - ik = 0 \Longrightarrow i = \frac{n}{k}$

$$T(n) = \frac{n}{k}\left[T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right)\right] + \sum_{i=1}^{\frac{n}{k}}(n-ik) \quad (1)$$

$$= \frac{n}{k}\left[T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right)\right] + \frac{n^2}{k} - n$$

$$= \frac{n}{k}\left[T_{is}\left(\frac{k}{2}\right) + T_{is}\left(\frac{k}{2}\right) + n\right] - n. \quad (2)$$

Eq. (2) represents a general form of growth function, it shows that the complexity of the proposed algorithm mainly depends on the value of k and the complexity of insertion functions.

4.1. Average case analysis

The average case of classical insertion sort $T_{is}(n)$ that appeared in Eq. (2) has been well analyzed in terms of comparisons in [34,35]

and in [35] for assignments. However, authors of the cited works presented the following equations which represent the average case analysis for classical insertion sort for comparisons and assignments respectively.

$$T_{isc}(n) = \frac{n^2}{4} + \frac{3n}{4} - 1 \tag{3}$$

$$T_{isa}(n) = \frac{n^2}{4} + \frac{7n}{4} + 3. \tag{4}$$

Eqs. (3) and (4) show that the insertion sort has approximately equal number of comparisons ($T_{isc}(n)$) and assignments ($T_{isa}(n)$). However, for BCIS, it is assumed that in each sort trip k elements are inserted into both side. Therefore, the main **while** loop executes n/k times that represent the number of sort trips. Suppose each insertion function get $k/2$ elements where $2 \leq k \leq n$. Since both insertion functions (INSLEFT, INSRIGHT) exactly work as a standard insertion sort, so the comparisons average case for each function during each sort trip is.

$$\begin{aligned} Comp.\#/SortTrip/Function &= T_{isc}\left(\frac{k}{2}\right) \\ &= \frac{k^2}{16} + \frac{3k}{8} - 1. \end{aligned} \tag{5}$$

BCIS performs one extra assignment operation to move the element that neighbored to the sorted section before calling each insertion function. Considering this cost we obtained as follows: -

$$\begin{aligned} Assig.\#/SortTrip/Function &= T_{isa}\left(\frac{k}{2}\right) + \frac{k}{2} \\ &= \frac{k^2}{16} + \frac{11k}{8} + 3. \end{aligned} \tag{6}$$

In order to compute BCIS comparisons complexity, we substituted Eq. (5) in Eq. (2) and we obtained as follows:-

$$T_c(n) = \frac{n}{k}\left[\frac{k^2}{8} + \frac{3k}{4} - 2 + n\right] - n. \tag{7}$$

Eq. (7) shows that when k gets small value the algorithm performance goes close to $O(n^2)$. For $k = 2$ the growth function is shown below.

$$\begin{aligned} T_c(n) &= \frac{n}{2}\left[\frac{4}{8} + \frac{3}{2} - 2 + n\right] - n \\ &= \frac{n^2}{2} - n. \end{aligned} \tag{8}$$

When k gets large value also the complexity of BCIS goes close to $O(n^2)$. For k=n the complexity is:-

$$\begin{aligned} T_c(n) &= \frac{n}{n}\left[\frac{n^2}{8} + \frac{3n}{4} - 2 + n\right] - n \\ &= \frac{n^2}{8} + \frac{3n}{4} - 2. \end{aligned} \tag{9}$$

Hence, the approximate best performance of the average case for BCIS that could be obtained when $k = n^{0.5}$ as follows:-

$$\begin{aligned} T_c(n) &= \frac{n}{n^{0.5}}\left[\frac{n}{8} + \frac{3n^{0.5}}{4} - 2 + n\right] - n \\ &= \frac{9n^{1.5}}{8} - \frac{n}{4} - 2n^{0.5}. \end{aligned} \tag{10}$$

Eq. (10) computes the number of comparisons of BCIS when runs in the best performance of the average case. On other hand, to compute the number of assignments for BCIS in case of best performance of average case. Since assignments operations occur only in insertions functions, Eq. (6) is multiplied by two because there are two insertion functions, then the result is multiplied by the number of sort trip $\frac{n}{k}$. When $k = n^{0.5}$ we got as follows:

$$\begin{aligned} T_a(n) &= \frac{n}{n^{0.5}}\left[\frac{n}{8} + \frac{11n^{0.5}}{4} + 6\right] \\ &= \frac{n^{1.5}}{8} + \frac{11n}{4} + 6n^{0.5}. \end{aligned} \tag{11}$$

The comparison of Eq. (10) with Eq. (11) proves that the number of assignments less than the number of comparisons in BCIS. As we mentioned previously in Eqs. (3) and (4), IS has approximately equal number of comparisons and assignments. This property makes BCIS runs faster than IS even when they have close number of comparisons.

Hence, we wrote the code section in statements (18)–(25) to optimize the performance of BCIS by keeping k close to $n^{0.5}$ as possible. This code segment is based on the idea that ensures at least a set of length $(SR-SL)^{0.5}$ not to be inserted during the current sort trip (where sort trip size $= SR - SL$). This idea realized by scanning this set looking for the minimum and maximum element and replace them with LC and RC respectively. However, this code does not add extra cost for the performance of the algorithm because the current sort trip will start where the loop in statement (19) has finished (sort trip will start at $(SR-SL)^{0.5}+1$). Theoretical results have been compared with experimental results in Section 5 and BCIS showed performance close to the best performance of average case that explained above.

In the rest part of this section, instruction level analysis of BCIS is presented. We re-analyze the algorithm for average case by applying above assumption to get more detailed analysis. However, the cost of each instruction is demonstrated as comment in the pseudo code of the algorithm, we do not explicitly calculate the cost of the loop in statement (19), because it is implicitly calculated with the cost of not inserted elements inside the loop started in statement (30). Code segment within statements (18–25) activates for sort trip size greater than 100 elements only. Otherwise, sort trip index i starts from the element next to SL (statement 27).

The total number of comparisons for each insertion function is calculated by Eq. (5) multiplied by the number of sort trip (n/k) as following:-

$$\frac{n}{k}\left(\frac{k^2}{16} + \frac{3k}{8} - 1\right) = \frac{nk}{16} + \frac{3n}{8} - \frac{n}{k}. \tag{12}$$

The Complexity of the check equality function ISEQUAL is neglected because **if** statement at (10) rarely gets true. The total complexity of BCIS is calculated as following:-

$$\begin{aligned} T(n) = {} & C1 + C2 + C3 \\ & + (C3 + C4 + C5 + C6 + C7 + C8 + C21 + C22)\frac{n}{k} \\ & + (C10 + C11 + C12 + C16 + C20)\sum_{i=1}^{\frac{n}{k}}(n - ik) \\ & + (C13 + C15 - C16 + C17 + C18 + C19)\frac{n}{k} \\ & + C14 * 2\left(\frac{nk}{16} + \frac{3n}{8} - \frac{n}{k}\right) - C20n \end{aligned}$$

$$a = (C3 + C4 + C5 + C6 + C7 + C8 + C21 + C22)$$

$$b = C14$$

$$c = (C10 + C11 + C12 + C16 + C20)$$

$$d = (C13 + C15 - C16 + C17 + C18 + C19)$$

$$e = C20$$

$$f = (C1 + C2 + C3)$$

Algorithm BCIS Average case analysis Part 1

```
1:  procedure BCIS(array, left, right)
2:      array is the array that required to sort
3:      left is the index of left most element in array
4:      right is the index of right most element in array
5:      SL ← left                                              ▷ C1
6:      SR ← right                                             ▷ C2
7:      while SL < SR do                                       ▷ C3(n/k + 1)
8:          SWAP(array, SR, SL + (SR−SL)/2)                    ▷ C4(n/k)
9:          if array[SL]= array[SR] then                       ▷ C5(n/k)
10:             if ISDUP(array, SL, SR)=-1 then
11:                 return
12:             end if
13:         end if
14:         if array[SL] > array[SR] then                      ▷ C6(n/k)
15:             SWAP (array, SL , SR)
16:         end if
17:         if (SR − SL) ≥ 100 then                            ▷ C7(n/k)
18:             for i ← SL + 1 to (SR − SL)^0.5 do
19:                 if array[SR] < array[i] then
20:                     SWAP (array, SR, i)
21:                 else if array[SL] > array[i] then
22:                     SWAP (array, SL, i)
23:                 end if
24:             end for
25:         else
26:             i ← SL + 1
27:         end if
28:         LC ← array[SL]                                     ▷ C8(n/k)
29:         RC ← array[SR]                                     ▷ C9(n/k)
30:         while i < SR do                                    ▷ C10 Σ_{i=1}^{n/k}(n − ik)
31:             CurrItem ← array[i]                            ▷ C11 Σ_{i=1}^{n/k}(n − ik)
32:             if CurrItem ≥ RC then                          ▷ C12 Σ_{i=1}^{n/k}(n − ik)
33:                 array[i] ← array[SR − 1]                   ▷ C13(n/2)
34:                 INSRIGHT(array, CurrItem, SR, right)       ▷ C14
    (nk/16 + 3n/8 − n/k)
35:                 SR ← SR − 1                                ▷ C15(n/2)
36:             else if CurrItem ≤ LC then ▷ C16 (Σ_{i=1}^{n/k}(n − ik) − n/2)
37:                 array[i] ← array[SL + 1]                   ▷ C17(n/2)
38:                 INSLEFT(array, CurrItem, SL, left)         ▷ C14
    (nk/16 + 3n/8 − n/k)
39:                 SL ← SL + 1                                ▷ C18(n/2)
40:                 i ← i + 1                                  ▷ C19(n/2)
41:             else
42:                 i ← i + 1                  ▷ C20(Σ_{i=1}^{n/k}(n − ik) − n)
43:             end if
44:         end while
45:         SL ← SL + 1                                        ▷ C19(n/k)
46:         SR ← SR − 1                                        ▷ C19(n/k)
47:     end while
48: end procedure
```

$$T(n) = a\frac{n}{k} + b\left(\frac{nk}{8} + \frac{3n}{4} - \frac{2n}{k}\right) + c\sum_{i=1}^{\frac{n}{k}}(n - ik) + d\frac{n}{2} - en + f$$

14	1	2	3	4	5	6	15	8	9	10	11	12	13	7

Fig. 2. Best case example for array less than 100 elements.

$$\begin{aligned} &= a\frac{n}{k} + b\left(\frac{nk}{8} + \frac{3n}{4} - \frac{2n}{k}\right) + c\left(\frac{n^2}{2k} - \frac{n}{2}\right) \\ &\quad + d\frac{n}{2} - en + f \\ &= \frac{n}{k}\left[a + b\left(\frac{k^2}{8} + \frac{3k}{4} - 2\right) + c\frac{n}{2}\right] - c\frac{n}{2} \\ &\quad + d\frac{n}{2} - en + f. \end{aligned} \tag{13}$$

We notice that Eq. (13) is similar to Eq. (7) when constants represent instructions cost.

4.2. Best case analysis

The best case occurs in case of every element is placed in its final correct position consuming a limited and constant number of comparisons and shift operations at one sort trip. These conditions are available once the first sort trip starts while the RC and LC are holding the largest and second largest item in the array respectively, and all other elements are already sorted. The following example in Fig. 2 explains this best case (the element 15 will replaced with 7 by the statement 9).

For this best case, we note that all insertions will be in the left side only with one shifting operation per each insertion. That means the cost of insertion each item is $O(1)$. Therefor, the total cost of the left insertion function is $T_{is}(n) = n$. Also all elements will inserted in one sort trip so that $k = n$. These values is substituted in Eq. (1) as follows:-

$$\begin{aligned} T(n) &= \frac{n}{k}\left[T_{is}(n)\right] + \sum_{i=1}^{\frac{n}{k}}(n - ik) \\ &\quad \text{where } k = n \\ T(n) &= n. \end{aligned} \tag{14}$$

Hence, the best case of BCIS is O (n) for $n < 100$. Otherwise, (for $n \geq 100$) the loop started in statement (19) always prevents this best case occurred because it only put LC and RC in their correct position and disallow insertions during all sort trips. As result, the loop in statement (19) forces the algorithm running very slow on already sorted or revers sorted array.

Generally, already sorted and reverse sorted arrays are more common in practice if compared with the above best case example. Therefore, statement (9) has been added to enhance the performance of best case and worst case when BCIS run on sorted and revers sorted arrays. In case of already sorted array, this statement makes the BCIS, during each sort trip, inserts half of $(SR - SL)$ in least cost.

The example in Fig. 3 explains how BCIS runs on already sorted array. For simplicity not inserted elements are not represented in each sort trip during the demonstration of this example.

For already sorted array, BCIS scans the first half consuming two comparisons per item (no insertions), then inserts the second half of each sort trip consuming two comparisons per item too. Because the sort trip size is repeatedly halved. Hence, it can represent as following.

$$T(n) = \left(2\frac{n}{2} + 2\frac{n}{2}\right) + \left(2\frac{n}{4} + 2\frac{n}{4}\right) + \left(2\frac{n}{8} + 2\frac{n}{8}\right) + \cdots + \left(2\frac{n}{2^i} + 2\frac{n}{2^i}\right)$$

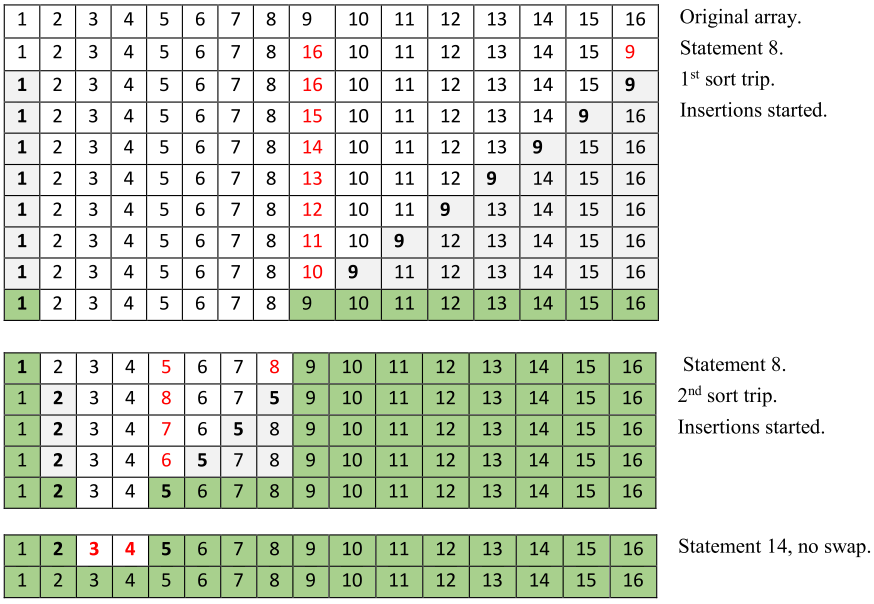

Fig. 3. Example of running BCIS on already sorted array.

14	13	12	11	10	9	8	15	6	5	4	3	2	1	7

Fig. 4. Worst case example for array less than 100 elements.

$$\text{stop when } 2^i = n \Longrightarrow i = \log n$$

$$T(n) = \sum_{i=1}^{\log n} \left(2\frac{n}{2^i} + 2\frac{n}{2^i}\right) = 4n. \tag{15}$$

Eqs. (14) and (15) represent the best case growth functions of BCIS when run on array size less than 100 and greater than and equal to 100 respectively.

4.3. Worst case analysis

The worst case happens only if all elements are inserted in one side in reverse manner during the first sort trip. This condition provided when the RC and LC are the largest and second largest numbers in the array respectively, and all other items are sorted in reverse order. The insertion will be in the left side only. The following example in Fig. 4 explains this worst case when $n < 100$.

Since each element in the this example inserted reversely, the complexity of left insertion function for each sort trip equal to $T_{is}(n) = \frac{n(n-1)}{2}$. Also there is one sort trip so $k = n$, by substitute these values in Eq. (1) as follows:-

$$T(n) = \frac{n}{k}\left[T_{is}(n)\right] + \sum_{i=1}^{\frac{n}{k}}(n - ik)$$

$$\text{where } k = n$$

$$T(n) = \frac{n(n-1)}{2}. \tag{16}$$

Hence, the worst case of BCIS is $O(n^2)$ for $n < 100$. Likewise the situation in the best case, the loop in statement (19) prevents the worst case happen because LC will not take the second largest item in the array. Consequently, the worst case of BCIS would be when it runs on reversely sorted array for $n \geq 100$. The example in Fig. 5 explains the behavior of the BCIS on such arrays even the size of array less than 100.

In case of reversely sorted array, BCIS does not insert the first half of the scanned elements, cost two comparisons per each element, then insert the second half reversely for each sort trip approximately. Considering the cost of reverse insertion (for each sort trip) is $T_{is}(k) = \frac{k(k-1)}{2}$ where k halved repeatedly. Like already sorted array analysis, the complexity of BCIS can be represented as follows.

$$T(n) = \left(2\frac{n}{2} + \frac{(\frac{n}{2})^2 - \frac{n}{2}}{2}\right) + \left(2\frac{n}{4} + \frac{(\frac{n}{4})^2 - \frac{n}{4}}{2}\right) + \left(2\frac{n}{8} + \frac{(\frac{n}{8})^2 - \frac{n}{8}}{2}\right) + \cdots + \left(2\frac{n}{2^i} + \frac{(\frac{n}{2^i})^2 - \frac{n}{2^i}}{2}\right)$$

$$\text{stop when } 2^i = n \Longrightarrow i = \log n$$

$$T(n) = \sum_{i=1}^{i=\log n}\left(2\frac{n}{2^i} + \frac{(\frac{n}{2^i})^2 - \frac{n}{2^i}}{2}\right) = \frac{n^2}{6} + \frac{3n}{2}. \tag{17}$$

Eqs. (16) and (17) represent the worst case growth functions of BCIS when run on array size less than 100 and greater than and equal to 100 respectively.

5. Results and comparison with other algorithms

The proposed algorithm is implemented by C++ using NetBeans 8.0.2 IDE based on Cygwin compiler. The measurements are taken on a 2.1 GHz Intel Core i7 processor with 6 GB 1067 MHz DDR3 memory machine with windows platform.

Experimental test has been done on an empirical data (numbers) that generated randomly using a C++ library. This library classes generate specified ranged of random numbers with different distributions [36]. The real-world data that has been used in this study downloaded from [37]. All results comparisons use Uniform distribution except these shown in Fig. 9.

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	**1**	Original array
16	15	14	13	12	11	10	9	1	7	6	5	4	3	2	8	Statement 8
16	15	14	13	12	11	10	9	1	7	6	5	4	3	2	8	Statement 14, swap
8	15	14	13	12	11	10	9	1	7	6	5	4	3	2	**16**	1st sort trip insertions
1	**8**	14	13	12	11	10	9	15	7	6	5	4	3	2	**16**	started
1	7	**8**	13	12	11	10	9	15	14	6	5	4	3	2	**16**	
1	6	7	**8**	12	11	10	9	15	14	13	5	4	3	2	**16**	
1	5	6	7	**8**	11	10	9	15	14	13	12	4	3	2	**16**	
1	4	5	6	7	**8**	10	9	15	14	13	12	11	3	2	**16**	
1	3	4	5	6	7	**8**	9	15	14	13	12	11	10	2	**16**	
1	2	3	4	5	6	7	**8**	15	14	13	12	11	10	9	**16**	

1	2	3	4	5	6	7	8	15	14	13	9	11	10	12	16	Statement 8
1	2	3	4	5	6	7	8	12	14	13	9	11	10	15	16	Statement 14, swap
1	2	3	4	5	6	7	8	**12**	14	13	9	11	10	**15**	16	2nd sort trip
1	2	3	4	5	6	7	8	9	**12**	13	14	11	10	**15**	16	insertions started
1	2	3	4	5	6	7	8	9	11	**12**	14	13	10	**15**	16	
1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	**15**	16	

1	2	3	4	5	6	7	8	9	10	11	12	**13**	**14**	15	16	Statement 14, no
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	swap

Fig. 5. Example of running BCIS on reversely sorted array.

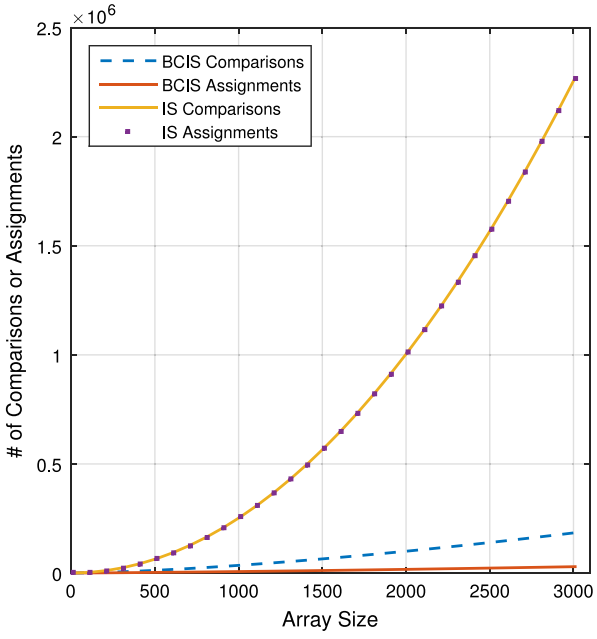

Fig. 6. No. of comparisons and assignments for BCIS and IS.

5.1. BCIS and classical insertion sort

Fig. 6 explains the average number of comparisons and assignments (Y axis) for BCIS and IS with different list size (X axis). The figure has been plotted using Eqs. (3), (4), (10) and (11). This figure explains that the number of comparisons and assignments of IS are approximately equal. In contrast, in BCIS the number of assignments are less than the number of comparisons. This feature show the better performance of BCIS and support our claim that BCIS has less memory read/write operations when compared with IS.

Though the theoretical average analysis for BCIS and IS is calculated in term of the number of comparisons and assignments separately in Eqs. (3), (4), (10) and (11). In order to compare the results of these equations with experimental results of BCIS and IS which are measured by execution elapsed time, we represent these quantities as a theoretical and experimental ratio of $\frac{BCIS}{IS}$. In theoretical ratio we assumed that the cost of an operation of comparison and assignment is equal in both algorithms. Therefore, Eq. (3) has been added to Eq. (4) to compute the total cost of IS (IS_{total}). Similarly, the total cost of BCIS ($BCIS_{total}$) is the result of add Eq. (10) to Eq. (11).

Fig. 7 illustrates a comparison in performance of the proposed algorithm BCIS and IS. This comparison has been represented in terms of the ratio BCIS/IS (Y axis) that required to sort a list of random data for some list sizes (X axis). Theoretically, this ratio is equal to($\frac{BCIS_{total}}{IS_{total}}$). In opposition, the experimental ratio computed by divide $BCIS_{time}$ over IS_{time}, when these parameters represent experimental elapsed running time of BCIS and IS respectively.

In the experimental $\frac{BCIS_{time}}{IS_{time}}$ ratio, we noticed that the proposed algorithm has roughly equal performance when compared to classical insertion sort for list size less than 50 ($\frac{BCIS_{time}}{IS_{time}} = 1$). However, the performance of BCIS increased for larger list size noticeably. The time required to sort the same size of list using BCIS begin in 70% then inclined to 4% of that consumed by classical insertion sort for list size up to 6000. Fig. 8 explains the same ratio for $n > 6000$. This figure shows that $\frac{BCIS_time}{IS_time}$ is decreased when the list size increased. For instance for the size 3,643,076 the experimental ratio equal to 0.00128 that means BCIS 781 times faster than IS.

In conclusion, Figs. 7 and 8 show that the theoretical and experimental ratio are very close especially for large size lists. This means BCIS go close to the best performance of average case for large size lists.

The experimental BCIS/IS for several distributions has been tested and compared with real-world data in Fig. 9. This figure show that BCIS approximately has the same performance on the Uniform, Normal and Poisson distributions as well as real-world data. However, Binomial distribution class in [36] has been configured to generate more duplicated numbers compared with other distributions to show the effect of moderate rate of duplicate elements. For this reason, Binomial distribution shows slightly better performance compared with other distributions. That happened because duplicated elements reduce the comparisons and assignments (shifting operations) inside inserting functions in BCIS. Concisely, BCIS shows approximately equal performance for the all tested distributions and real-world data with low or no duplicated

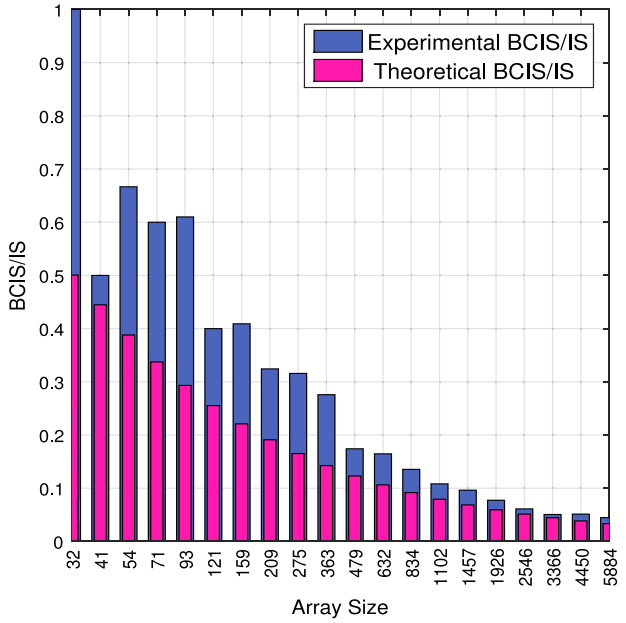

Fig. 7. BCIS/IS ratio for $n < 6000$.

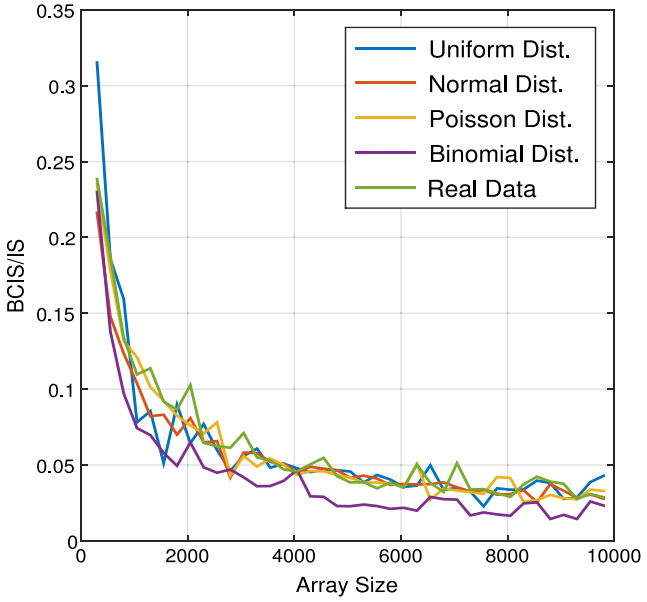

Fig. 9. BCIS/IS ratio with different data Distributions.

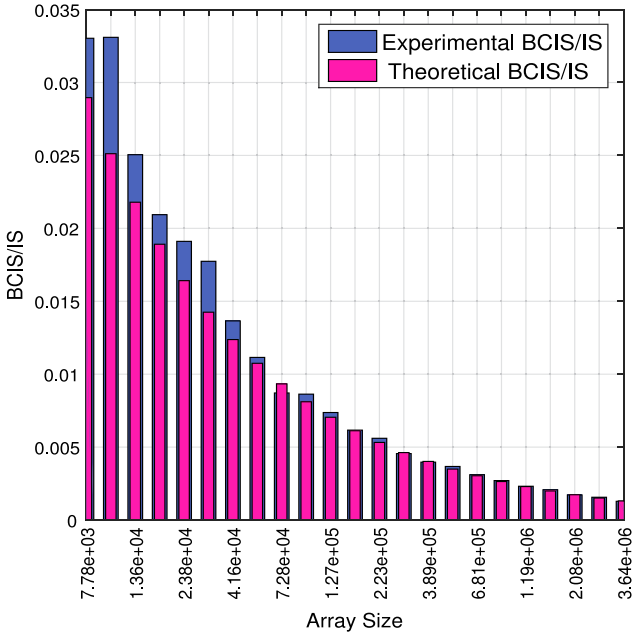

Fig. 8. BCIS/IS ratio for $n > 6000$.

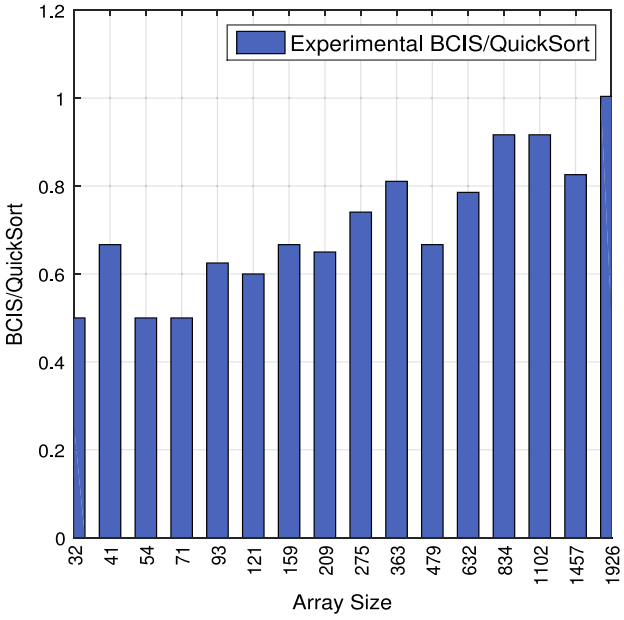

Fig. 10. BCIS and Quick Sort performance $n < 2000$.

elements, while the performance of BCIS increases when duplicated elements increase. For high rate of duplicated elements BCIS compared with QuickSort in Section 5.2.2.

5.2. BCIS and Quicksort

5.2.1. BCIS and Quicksort comparison for no duplicated-elements data set

Fig. 10 explains a comparison in experimental performance of the proposed algorithm BCIS and Quicksort for small size lists. Widely used enhanced version of Quicksort (median-of-three pivots) is used, which is proposed by [7]. This comparison has been represented in terms of the experimental ratio $\frac{BCIS_{time}}{QuickSort_{time}}$ (Y axis) that required to sort a list with random data for some list sizes (X axis). We remarked that BCIS is faster than Quicksort for list size less than 1500 for most cases. The time required to sort the same size of list using BCIS ranged between 30% and 90% of that consumed by Quicksort when list size less than 1500.

Although theoretical analysis in all previous cited works that have been explained in literature (Section 1) show that QuickSort has more efficient comparison complexity than BCIS. But experimentally BCIS defeats QuickSort for relatively small array size for some reasons. First the low number of assignment operations in BCIS, second an assignment operation has lower cost if compared with swap operation that used in QuickSort, whereas each swap operation requires three assignments to done. Finally, due to the nature of the cache memory architecture [38], BCIS uses cache memory efficiently because shifting operations only access to the

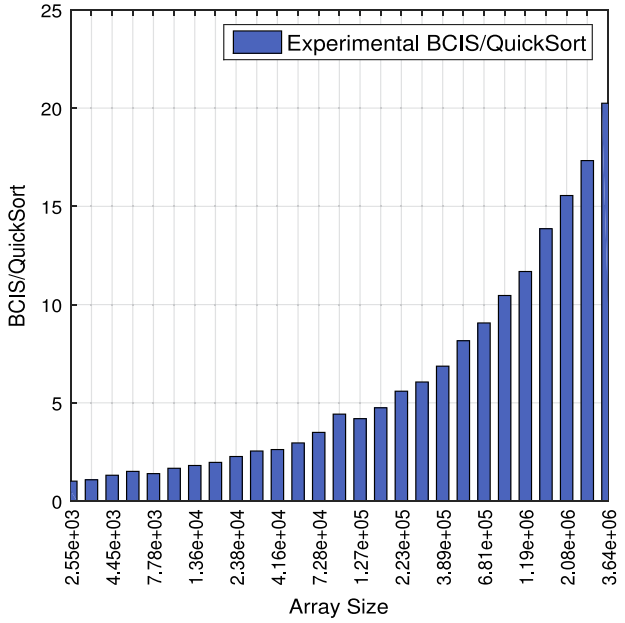

Fig. 11. BCIS/Quicksort for $n > 2000$.

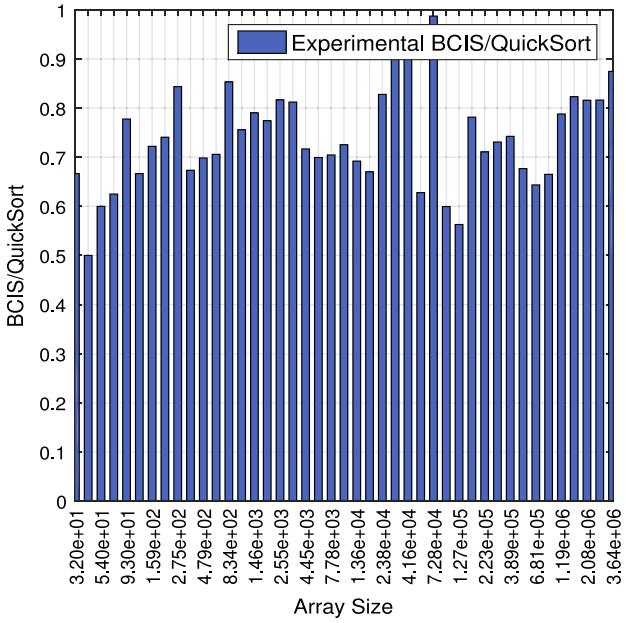

Fig. 12. Experimental BCIS/Quicksort ratio for 50 duplicated elements.

adjacent memory locations while swap operations in QuickSort access memory randomly. Therefore, QuickSort cannot efficiently invest the remarkable speed gain that is provided by the cache memory. However, Fig. 11 shows experimental $\frac{BCIS_{time}}{QuickSort_{time}}$ ratio for array size greater than 2000 and less than 3640,000.

5.2.2. BCIS and Quicksort comparison for high rate of duplicated elements data set

Experimental test showed that BCIS faster than Quicksort when running on data set has high rate of duplicated elements even for large list size. Fig. 12 explains the experimental ratio $\frac{BCIS}{Quicksort}$ when the used array has only 50 different elements. The computer randomly duplicates the same 50 elements for arrays that have size greater than 50. This figure shows that BCIS consumes only 50% to 90% of the time consumed by Quicksort when run on high rate of duplicated elements array. This variation in ratio is due to the random selection of LC and RC during each sort trip. The main factor that makes BCIS faster than Quicksort for such type of array is that there a small number of assignments and comparisons operations if there are many numbers equal to LC or RC in each sort trip. This case can occur with high probability when there is high rate of duplicated elements in array.

6. Conclusions

In this paper we have proposed a new bidirectional conditional insertion sort. The performance of the proposed algorithm has significant enhancement over classical insertion sort. As above shown results prove. BCIS has average case about $n^{1.5}$. Also BCIS keeps the fast performance of the best case of classical insertion sort when runs on already sorted array. Since BCIS time complexity is (4n) over that array. Moreover, the worst case of BCIS is better than IS whereas BCIS consumes only $\frac{n^2}{6}$ comparisons with reverse sorted array.

The other advantage of BCIS is that algorithm is faster than Quicksort for relatively small size arrays (up to 1500). This feature does not make BCIS the best solution for relatively small size arrays only. But it makes BCIS powerful interested algorithm to use in conjugate with quick sort. The performance of the sorting process for large size array could be increased using hybrid algorithms approach by using Quicksort and BCIS. Additionally, above results shown that BCIS is faster than quick sort for arrays that have high rate of duplicated elements even for large size arrays. Moreover, for fully duplicated elements array, BCIS indicates fast performance as it can sort such array in only $O(n)$.

Acknowledgments

This research is partially supported by administration of governorate of Salahaddin and Tikrit university — Iraq (2245-6-2-2013). We thank Thamer Fahad Al-Mashhadani for comments that greatly improved the manuscript. We thank the editor and the anonymous reviewers for their constructive comments which helped us to improve this paper.

References

[1] Abdel latif Abu Dalhoum, Thaer Kobbay, Azzam Sleit, Enhancing QuickSort algorithm using a dynamic pivot selection technique, Wulfenia 19 (10) (2012) 543–552.
[2] R. Sedgewick, K. Wayne, Algorithms, 4th ed., Addison-Wesley Professional, U.S.A., 2011. ISBN: 032157351X 9780321573513.
[3] Pooja K. Chhatwani, Insertion sort with its enhancement, Int. J. Comput. Sci. Mobile Comput. 3 (3) (2014) 801–806.
[4] Jehad Alnihoud, Rami Mansi, An enhancement of major sorting algorithms, Int. Arab J. Inform. Technol. (ISSN: 16833198) 7 (1) (2010) 55–62.
[5] Sahil Gupta Eshan Kapur, Parveen Kumar, Proposal of a two way sorting algorithm and performance comparison with existing algorithms, Int. J. Comput. Sci. Eng. Appl. 2 (3) (2012) 61–78.
[6] Rupesh Srivastava, Tarun Tiwari, Sweetesh Singh, Bidirectional expansion - Insertion algorithm for sorting, in: Second International Conference on Emerging Trends in Engineering and Technology, ICETET, ISBN: 9780769538846, 2009, pp. 59–62. http://dx.doi.org/10.1109/ICETET.2009.48.
[7] Robert Sedgewick, The analysis of Quicksort programs, Acta Inform. (ISSN: 0001-5903) 7 (4) (1977) 327–355. Available on http://dx.doi.org/10.1007/BF00289467.
[8] Robert Sedgewick, Implementing Quicksort programs, Commun. ACM (ISSN: 00010782) 21 (10) (1978) 847–857. http://dx.doi.org/10.1145/359619.359631.
[9] Sebastian Wild, Markus E. Nebel, Ralph Neininger, Average case and distributional analysis of dual-pivot quicksort, ACM Trans. Algorithms 11 (3) (2015) 22.

[10] Jl Bentley, M. Douglas McIlroy, Engineering a sort function, Software Practice Exp. (ISSN: 00380644) 23 (November) (1993) 1249–1265. Available on http://dx.doi.org/10.1002/spe.4380231105.
[11] W. Min, Analysis on bubble sort algorithm optimization, in: Information Technology and Applications (IFITA), 2010 International Forum on, vol. 1, 2010, pp. 208–211. http://dx.doi.org/10.1109/IFITA.2010.9.
[12] Coding Unit Programming Tutorials, Cocktail sort algorithm or Shaker sort algorithm, 2016. http://www.codingunit.com/cocktail-sort-algorithm-or-shaker-sort-algorithm. (Accessed 16 January 2016).
[13] Debasis Samanta, Classic Data Structures, 2nd, PHI Learning Pvt. Ltd. ISBN: 812033731X, 2009.
[14] H.H. Chern, H.K. Hwang, Transitional behaviors of the average cost of quicksort with median-of-$(2t+1)$, Algorithmica (ISSN: 01784617) 29 (1–2) (2001) 44–69. http://dx.doi.org/10.1007/s004530010054.
[15] Mahmoud Fouz, Manfred Kufleitner, Bodo Manthey, Nima Zeini Jahromi, On smoothed analysis of Quicksort and Hoare's find, Algorithmica (ISSN: 01784617) 62 (3–4) (2012) 879–905. http://dx.doi.org/10.1007/s00453-011-9490-9. arXiv:arXiv:0904.3898v2.
[16] V. Yaroslavskiy, Question on sorting, 2010.
[17] Markus E. Nebel, Sebastian Wild, Conrado Martínez, Analysis of pivot sampling in dual-pivot Quicksort: A holistic analysis of Yaroslavskiy's partitioning scheme, Algorithmica (ISSN: 0178-4617) 75 (4) (2016) 632–683. http://dx.doi.org/10.1007/s00453-015-0041-7.
[18] Sebastian Wild, Markus E. Nebel, Average case analysis of Java 7 ' s dual pivot Quicksort, in: European Symposium on Algorithms, Springer, 2012, pp. 825–836. http://dx.doi.org/10.1007/978-3-642-33090-2_71. arXiv:1310.7409.
[19] Martin Aumüller, Martin Dietzfelbinger, Optimal partitioning for dual-pivot Quicksort, ACM Trans. Algorithms (ISSN: 15496325) 12 (2) (2016) 18. http://dx.doi.org/10.1145/2743020. arXiv:1303.5217.
[20] Michael L. Fredman, An intuitive and simple bounding argument for Quicksort, Inform. Process. Lett. (ISSN: 0020-0190) 114 (3) (2014) 137– 139. http://dx.doi.org/10.1016/j.ipl.2013.10.010. http://www.sciencedirect.com/science/article/pii/S0020019013002676.
[21] Petros Hadjicostas, K.B. Lakshmanan, Recursive merge sort with erroneous comparisons, Discrete Appl. Math. (ISSN: 0166218X) 159 (14) (2011) 1398–1417. http://dx.doi.org/10.1016/j.dam.2011.05.010.
[22] Patrick Bindjeme, James Allen Fill, Exact Lˆ2 -distance from the limit for quicksort key comparisons (extended abstract), ArXiv Preprint arXiv:1201.6445v1, 2012, pp. 1–9.
[23] Ralph Neininger, Refined Quicksort asymptotics, Random Structures Algorithms 46 (2) (2015) 346–361. http://arxiv.org/abs/1207.4556.
[24] Michael Fuchs, A note on the quicksort asymptotics, Random Structures Algorithms (ISSN: 10982418) 46 (4) (2015) 677–687. http://dx.doi.org/10.1002/rsa.20524.
[25] Franciszek Grabowski, Dominik Strzalka, Dynamic behavior of simple insertion sort algorithm, Fund. Inform. 72 (1–3) (2006) 155–165.
[26] Christophe Biernacki, Julien Jacques, A generative model for rank data based on insertion sort algorithm, Comput. Statist. Data Anal. (ISSN: 01679473) 58 (1) (2013) 162–176. http://dx.doi.org/10.1016/j.csda.2012.08.008.
[27] Michael A Bender, Martin Farach-Colton, Miguel A. Mosteiro, Insertion sort is $O(n \log n)$, Theory Comput. Syst. 397 (2006) 391–397.
[28] Tarundeep Singh Sodhi, Enhanced insertion sort algorithm, Int. J. Comput. Appl. 64 (21) (2013) 35–39.
[29] Bruno R. Preiss, Data Structures and Algorithms with Object-oriented Design Patterns in C++, John Wiley & Sons, 2008.
[30] Partha Sarathi Dutta, An approach to improve the performance of insertion sort algorithm, Int. J. Comput. Sci. Eng. Technol. 4 (05) (2013) 503–505.
[31] Wang Min, Analysis on 2-element insertion sort algorithm, in: 2010 International Conference on Computer Design and Applications, ICCDA 2010, ICCDA, vol. 1, 2010, pp. 143–146. http://dx.doi.org/10.1109/ICCDA.2010.5541165.
[32] Md. Khairullah, Enhancing worst sorting algorithms, Int. J. Adv. Sci. Technol. 56 (2013) 13–26.
[33] Kamlesh Nenwani, Vanita Mane, Smita Bharne, Enhancing adaptability of insertion sort through 2-way expansion, in: Confluence the Next Generation Information Technology Summit (Confluence), 2014 5th International Conference, ISBN: 9781479942367, 2014, pp. 843–847.
[34] Kenneth H. Rosen, Discrete Mathematics and its Applications, Seventh ed., McGraw-Hill, ISBN: 1584884347, 2012 arXiv:1439880182.
[35] McQuain, Data structure and algorithms, 2009. http://courses.cs.vt.edu/~cs3114/Fall09/wmcquain/Notes/T14.SortingAnalysis.pdf.
[36] Network, The C++ resources. C/C++ reference, 2016. http://www.cplusplus.com/reference/random/. (Accessed 5 November 2016).
[37] Spatialkey.com. Sample CSV Data SpatialKey Support, 2016. https://support.spatialkey.com/spatialkey-sample-csv-data/. (Accessed 5 November 2016).
[38] Jim Handy, The Cache Memory Book, Morgan Kaufmann, 1998.

Adnan Saher Mohammed received B.Sc. degree in 1999 in computer engineering technology from College of Technology, Mosul, Iraq. In 2012 he obtained M.Sc degree in communication and computer network engineering from UNITEN University, Kuala Lampur, Malasyia. He is currently a Ph.D student at graduate school of natural sciences, Ankara Yıldırım Beyazıt University, Ankara, Turkey. His research interests include Computer Network and computer algorithms.

Şahin Emrah Amrahov received B.Sc. and Ph.D. degrees in applied mathematics in 1984 and 1989, respectively, from Lomonosov Moscow State University, Russia. He works as Associate Professor at Computer Engineering department, Ankara University, Ankara, Turkey. His research interests include the areas of mathematical modeling, algorithms, artificial intelligence, fuzzy sets and systems, optimal control, theory of stability and numerical methods in differential equations.

Fatih Vehbi Çelebi obtained his B.Sc. degree in electrical and electronics engineering in 1988, M.Sc. degree in electrical and electronics engineering in 1996, and Ph.D. degree in electronics engineering in 2002 from Middle East Technical University, Gaziantep University and Erciyes University respectively. He is currently head of the Computer Engineering department and vice president of Ankara Yıldırım Beyazıt University, Ankara-Turkey. His research interests include Semiconductor Lasers, Automatic Control, Algorithms and Artificial Intelligence.