# Accepted Manuscript

Brownian motus and Clustered Binary Insertion Sort methods: An efficient progress over traditional methods

Shubham Goel, Ravinder Kumar

PII: S0167-739X(17)31842-3
DOI: https://doi.org/10.1016/j.future.2018.04.038
Reference: FUTURE 4116

To appear in: *Future Generation Computer Systems*

Received date : 15 August 2017
Revised date : 24 January 2018
Accepted date : 11 April 2018

Please cite this article as: S. Goel, R. Kumar, Brownian motus and Clustered Binary Insertion Sort methods: An efficient progress over traditional methods, *Future Generation Computer Systems* (2018), https://doi.org/10.1016/j.future.2018.04.038

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

ACCEPTED MANUSCRIPT

# Brownian Motus and Clustered Binary Insertion Sort Methods: An Efficient Progress over Traditional Methods

Shubham Goel[1], Ravinder Kumar[1]

*Computer Science and Engineering Department, Thapar Institute of Engineering and Technology, Patiala, Punjab 147004, India*

**Abstract**

Sorting is the basic operation in every application of computer science. The paper proposes two novel sorting algorithms based on the concept of traditional Insertion Sort (IS). Firstly, Brownian Motus Insertion Sort (BMIS) based on IS is proposed. It is followed by Clustered Binary Insertion Sort (CBIS) based on the principles of Binary Insertion Sort (BIS). BIS is a binary search enhancement of IS which is a quite famous variant of it. Average case time complexity of BMIS is $O(\sqrt[0.54]{n})$; and that of CBIS is $O(n \log n)$. The scenario which results into the worst case of IS is with the complexity of $O(n^2)$; and BIS with $O(n \log n)$ is the best case scenario for BMIS and CBIS with complexity of $O(n)$. The probability of getting a worst case scenario for BMIS and CBIS is approximately zero. Comparison of proposed algorithms with IS and BIS has been performed at $25\%, 50\%, 75\%$ and $100\%$ level of randomness in the initial dataset. These results lead to prove our claim of devising efficient enhancements of IS. The results further reveal that performance of BMIS and CBIS will increase with a decrease in randomness level of the dataset in comparison to its counterparts. The number of comparisons required by BMIS and CBIS will approach to $O(n)$ with randomness level approach to zero. So, for nearly sorted datasets, our proposed BMIS and CBIS are the best choice. Both BMIS and CBIS are *in-place*, *stable* and *online sorting* algorithm.

*Keywords:* Brownian Motus Insertion Sort, Clustered Binary Insertion Sort, Insertion Sort, Binary Insertion Sort, Sorting Algorithm, Online Algorithm.

## 1. Introduction

With the tremendous use of computers in various field, data in large quantity is required to be accumulated to solve our numerous problems. In order to enhance the efficiency of mechanism, the researchers have designed certain algorithms over the years to overcome the emerging problems. Basically, an algorithm is a step-wise procedure to efficiently solve a problem with pencil and paper in a finite number of steps [1].

*Corresponding author
*Email address:* ravinder@thapar.edu (Ravinder Kumar)

ACCEPTED MANUSCRIPT

Algorithms have a crucial role in solving problems of mathematics and computer science. Data search is the basic problem almost in every field where computers are being used [2]. Traversing each element of a list from beginning to end in order to find the desired element is a complete wastage of time and computation cycles. To avoid this wastage, techniques of indexing and sorting came into light, but sorting proved to be more beneficial.

Sorting is one of the fundamental operations in computer science that re-arranges elements of a list into an ascending or descending order. Elements can be numeric, alphabetic or any object having specific key value [3]. There are a large number of sorting algorithms being used in industry and academia [4]. Factors affecting the selection of suitable sorting algorithm for an application can be divided into two groups; direct factors and indirect factors [5]. The size of a list, distribution scenario of the elements, percentage of already ordered elements in the list [6], time and space complexity of an algorithm come under direct factors [7]. Indirect factors include programming effort, data type of elements, processor speed, size of primary and secondary memory [8].

The performance of an algorithm can be computed on the basis of time and space complexity, which is usually represented by asymptotic notations [4]. The time complexity of a sorting algorithm is, generally, calculated in terms of number of comparisons or shifting operations made. Total working storage required by an algorithm in worst case scenario will amount to space complexity. On the basis of worst case time complexity, sorting algorithm can be divided into two categories. First category is $O(n^2)$; and second is $O(n \log n)$ which is faster than the first. Selection sort, Insertion sort, Bubble sort, etc. come under $O(n^2)$ category and Binary insertion sort, Heap sort, Merge sort, etc. appear under $O(n \log n)$ category. Due to stability, performance, simplicity, in-place, online nature; insertion sort is considered from the first category [7] and binary insertion sort from second as the best among peers.

Sorting is often used as a building block in designing of many important algorithms as once a list gets sorted, various other problems are reduced. A reduction is a process of transforming complex computational problems into easier ones. Some of the major applications where sorting is a prerequisite process are searching, element uniqueness, closest pair determination, selection, frequency distribution, convex hulls, etc. Usage of sorting in a large number of applications has increased the demand for developing more efficient, scalable and shorter running time sorting algorithms whose performance does not deteriorate with an increase in dataset size. This research paper proposes Brownian Motus Insertion Sort (BMIS) and Clustered Binary Insertion Sort (CBIS) sorting algorithms to decrease the time complexity of traditional Insertion Sort (IS) and Binary Insertion Sort(BIS) respectively. Time complexity has been measured in terms of a number of comparisons required by sorting algorithm. The time complexity has been decreased by eliminating many useless comparison operations through a novel approach for location identification. BMIS is a variant of IS; and CBIS is a variant of BIS. Both BMIS and CBIS belong to the *comparison*, *in-place*, *stable* and *online sort* class of sorting algorithms. Performance and scalability of BMIS and CBIS

ACCEPTED MANUSCRIPT

is much better than their traditional counterparts.

Further, the paper has been organized into various sections. Section 2 describes the application of Insertion Sort; and also highlights major improvements in the field. Section 3 presents the data used for the experimental work. Section 4 explains the proposed algorithms and pseudo-code, while Section 5 shows the execution illustration of BMIS and CBIS. Proposed algorithms have been theoretically analyzed; and complexity is computed in section 6. The results of this research work have been discussed in section 7. A comparison of proposed algorithm with traditional IS and BIS has been also made in the section. Section 8 summarizes and concludes the research.

## 2. Literature Review

Enhancement of sorting algorithms has always been a matter of keen interest for the researchers from the fields of computer science and mathematics. This section presents a brief review of published studies on Insertion Sort and its variants. The study has been classified into two sub-sections. The first sub-section reviews those studies which have used Insertion Sort as helping module. The second sub-section focuses on major variants of Insertion Sort.

### *2.1. Insertion Sort as Application*

Mishra et al.[8], Yang et al. [9] and Beniwal et al. [10] analyzed and compared different sorting algorithms in order to help users to find appropriate algorithm as per their need. Mishra et al. [8] divided the sorting algorithms into comparison-based and non-comparison sort. Comparison in a respective category is performed on the basis of average, worst case complexity, and pros and cons of each algorithm. Suitable algorithms for different sequence scenarios have also been clearly stated. An experimental study by Yang et al.[9] revealed that for smaller sized list, Insertion and Selection sort performs well, but for a larger sized list, merge and quicksort are good performers. Dynamic sorting of modules in physical memory [11] uses Insertion Sort algorithm to sort free spaces according to their size in fragmented memory. The ultimate motive to perform sorting is to aggregate free spaces at one location and make room for new modules.

### *2.2. Improvements in Insertion Sort*

Many improvements in traditional Insertion Sort (IS) algorithm were done by various researchers in the past. Some of the major improvements are presented here for a consideration. A new sorting algorithm for a nearly sorted list of elements was devised [12] from the combination of IS and quickersort algorithm. An auxiliary list is used to remove the unordered pairs from the main list; and sorting is done using IS or quickersort algorithm. Results of both the lists are then merged to give a final ordered list. Fun-Sort [13] use repeated binary search approach on an unsorted array for IS and acquired $\Theta(n^2 \log n)$ time complexity to sort an array of $n$ elements. When IS is executed by people in physical world, gaps are left by them to

ACCEPTED MANUSCRIPT

accelerate insertions. This idea of gapped insertion sort is studied [14] to devise an algorithm with $O(n \log n)$ time complexity with high probability. For each insertion, $O(\log n)$ time is required. Gapped insertion sort is also known as library sort algorithm as a similar concept is used by a librarian to arrange books on a shelf. Due to the usage of binary search, it is also known as Binary Insertion Sort (BIS). The impact of gap value and re-balancing factor on the execution time of an algorithm is studied in [15], which reveals the need for optimization of considered parameters in order to get better performance. An approach similar to the one followed by Bender et al.[14], to find the correct location of an element from an unsorted list.

An improvement for IS is also proposed by Min [16] as 2-element insertion sort where two elements from unsorted part of a list are inserted into sorted part in contrast to IS, only one element is inserted at one time. So, theoretically, $n/2$ number of iterations are required where $n$ is a number of element in a list. Time complexity is improved in 2-element insertion sort, but space complexity is increased as more auxiliary space is required. Dutta et al.[17] designed an approach for data which is in opposite order. Findings reveal that number of comparison operations are much less, i.e., nearly equal to $(3n-2)/2$ in comparison to IS. Khairullah [5] used another approach to enhance IS to give EISA algorithm. Initially, the first element from an unsorted array is copied to the middle of an additional temporary array. The size of this additional array is double the size of an input array. In further iterations, remaining elements from the unsorted array is copied to either left or right of middle elements depending on the result of comparison with them. The expansion is done from the center towards edges. Experimental results reveal that number of shift operations are highly reduced in comparison to IS, but at the cost of increase in space complexity by $O(2n)$. Number of comparison operations remain equal to those in IS.

Rotated library sort [18], number of operations per iteration is $O(\sqrt{n} \log n)$; and for worst case sorting time is $O(n^{1.5} \log n)$. The $O(w)$ auxiliary bits are used in the algorithm where $w$ is the number of bits for each element. Based on 2-way expansion, an Adaptive Insertion Sort (AIS) is proposed [19] to reduce the number of comparisons and shift operations. Doubly Inserted Sort [20], an approach similar to the max-min sorting algorithm is used to scan list from both ends. In each iteration, two elements from both ends are selected and compared with each other. Depending on comparison result swapping is performed to insert the elements in correct location into a sorted list. Mohammed et al. [7] proposed Bidirectional Insertion Sort (BCIS) based on the concept of IS, left and right comparators. As compared to IS a number of comparison and shift operations are less in BCIS. Average case and best case complexity of BCIS is $O(n^{1.5})$ and $O(4n)$ receptively.

The literature also provides that there is a lot of scope for improvement in the performance of IS. Most of the variants present in literature are using almost the same type of concept to enhance IS. Some have claimed to manage slightly better performance in comparison to IS, but that too at the cost of increase in space complexity or time complexity. There is hardly any variant of IS which has claimed to maintain *online sorting* feature of IS. Even some had also deviated from *in-place* and *stable sort* feature of IS to give better

ACCEPTED MANUSCRIPT

performance. Most of them failed to provide any practical proof of their claim.

The current paper proposes Brownian Motus Insertion Sort (BMIS), a variant of Insertion Sort (IS) and Clustered Binary Insertion Sort (CBIS) based on the Binary Insertion Sort (BIS). Both BMIS and CBIS are *in-place*, *online sort* as well as *stable sorting* algorithms. A detailed analysis of best, average, and worst case complexity of BMIS and CBIS has been done. Comparison of BMIS with IS and CBIS with BIS has been done on an extensive dataset with a different number of elements in each trial. Different levels of randomness has also been introduced in the initial dataset to perform a comparison of algorithms under consideration. Results of the comparison are shown with the help of 2D graphs.

## 3. Data acquisition

An extensive dataset has been used for the experiment to show the effectiveness of proposed algorithms over Insertion Sort (IS) algorithm and its variants. The entire dataset has been divided into three groups; and each group contains five sets of elements. A number of elements in each set of every group are shown in Table 1.

**Table 1**
Number of elements in each set w.r.t. group

| Sets | Group 1 | Group 2 | Group 3 |
|---|---|---|---|
| Set 1 | 100 | 2000 | 20000 |
| Set 2 | 500 | 4000 | 40000 |
| Set 3 | 1000 | 6000 | 60000 |
| Set 4 | 1500 | 8000 | 80000 |
| Set 5 | 2000 | 10000 | 100000 |

The four different versions of the entire dataset are considered for the purpose of experimental analysis of this research as follows:

1. 100% Random data
2. Partially Random data
3. Sorted data
4. Reverse order data

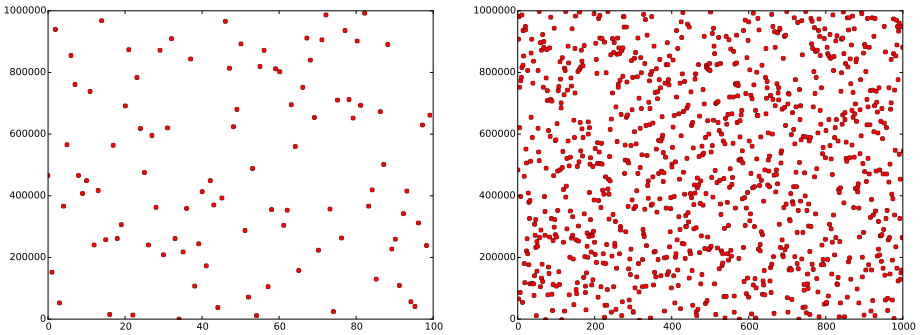
Initial data set is the 100% random data. In order to analyze the change in performance of proposed algorithms with change in randomness level, partially random version of data has been taken i.e. 25%, 50% and 75% level of randomness. In third version, completely sorted dataset has been taken in-order in which sorting is desired. In fourth version of data set the reverse ordered data has been taken as input. Performance of proposed algorithms have been measured in terms of number of comparisons and execution time required.

For each set, five samples have been collected. Each sample has been collected from a list of random numbers generated using Uniform Distribution, also known as rectangular distribution. For the dataset of

ACCEPTED MANUSCRIPT

this research work, distribution has been applied over the range of first one million Natural numbers $\mathbb{N}$. In Uniform Distribution [21, 22], the probability $p$ of selecting a variable randomly from an interval [$a$,$b$] remains the same for every variable in the interval. $p$ can be calculated as $(1/(b-a))$.

The cumulative total of probabilities of every random variable in the interval [$a$,$b$] must be equal to 1. Entire range of one million Natural numbers $\mathbb{N}$ has been divided into five sub-interval ranges, i.e., $200K$, $400K$, $600K$, $800K$ and $1000K$. The dataset taken up for this research work has random numbers from each sub-interval range without being biased for a particular range. From each group of the dataset, some of the representative sets shown in Fig. 1 strengthen our claim of being unbiased in random number selection. The final value of x-axis in every subfigure is representing a number of elements in the selected set. Sub-interval ranges are represented by y-axis. Each red circle in every subfigure of Fig. 1 is representing one random number.




**(a)** **(b)**

**Fig. 1.** Visualization of Dataset of (a) 100 and (b) 1000 random numbers

## 4. Proposed Algorithms

Traditional Insertion Sort (IS) is one of the well-known sorting algorithms, among the class of online sorting. It is usually preferred for the sorting of smaller sized list as its performance gets worse on a larger sized list due to the presence large number of comparisons required for sorting the elements.

The proposed algorithms of this research work remove useless comparison operations while performing a search on sorted part of a list to find a location for an element from unsorted part of a list with help of a novel methodology. Initially, the entire list was divided into two parts, i.e., sorted and unsorted just like IS by the proposed algorithms. The sorted part contains the elements which are already sorted in previous iterations of algorithm and unsorted part comprises of elements that are yet to be sorted. Two types of pointers have been used in the proposed algorithms; Current Pointer ($COP$) indicates towards a location of the first element of the unsorted part, while Position Pointer ($POP$) points towards the location at which

ACCEPTED MANUSCRIPT

current element, i.e., element at $COP$ is inserted into sorted part in the previous step. $POP$ will help the proposed algorithms to further divide the sorted part into two subparts, i.e., left and right subparts. Left subpart contains elements lesser than element at $POP$; and right subpart consists of elements greater than element at $POP$.

Unlike IS, elements present either in left or right subpart only are a candidate for comparison, whereas in IS each element in sorted part is a candidate. The main benefit of it is that number of comparisons required by the proposed algorithms get highly reduced in comparison to IS and its available variants. The current research work proposes two algorithms, one is *Brownian Motus Insertion Sort*(BMIS), and the other is *Clustered Binary Insertion Sort*(CBIS).

Brownian Motus Insertion Sort (BMIS) algorithm is an improvement over IS. In BMIS, firstly, a comparison of the element at $COP$ is made with the element at $POP$ and then with the elements either in a left or right subpart of a sorted part. The decision of subpart depends on the comparative results of elements at $COP$ and $POP$. Resultant is right subpart, if the element at $COP$ is greater than element at $POP$; and is left subpart, if it is lesser than element at $POP$. After the selection of subpart, a linear search is performed on selected subpart starting with the element at $POP$. Two functions, *viz.* $place\_finder\_left$ and $place\_finder\_right$ are used to perform linear search and location identification to insert an element at $COP$ into left or right subparts respectively. After the identification of location, $POP$ is updated to the latest location value, insertion and shifting logic is handled by the *place_inserter* function. Final outcome after completion of all iterations is a sorted list of elements. The proposed algorithm derived its name from a famous theory of the movement of particles, i.e., Brownian Motion given by a renowned botanist Robert Brown. The theory explained that particles move in a zigzag pattern. So, if $POP$ is considered as a particle and a geometric figure is drawn by joining $POP$ locations in each step of sorting process in a step by step manner, then, a zigzag pattern is obtained as shown in Fig. 2(b). A total number of comparisons required by both IS and BMIS for the entire sorting process is directly proportional to total distance traveled ($D$) which is equal to the sum of length of each line segment given by Eq. (1). Where, $l_n$ is the number of line segments in Fig. 2 (a)-(b) respectively and it can be clearly noticed that there is a huge difference between the values of $D$ for both IS and BMIS. For step by step explanation of BMIS, refer Algorithm 1; and for *place_inserter* function, refer Algorithm 3.

$$D = \sum_{i=1}^{l_n} len_i \tag{1}$$

Clustered Binary Insertion Sort (CBIS) algorithm is an improvement over IS and its famous variant Binary Insertion Sort (BIS) algorithms. In CBIS, firstly, a comparison of the element at $COP$ is made with the element at $POP$, and then, with elements either in left or right subparts. Here, after the decision of subpart, a binary search is performed on elements in the selected subpart. Function $binary\_loc\_finder$ is used to

ACCEPTED MANUSCRIPT

INPUT SEQUENCE: [83, 65, 31, 34, 47, 50, 76, 75, 40, 39, 71, 72, 79, 45, 23, 33]

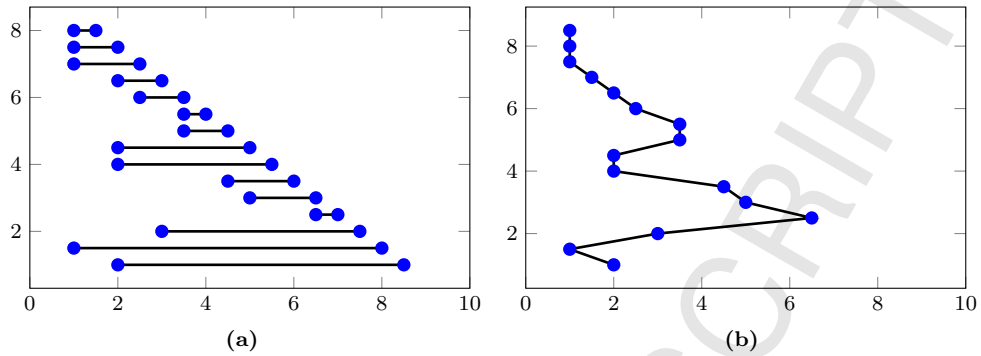



**Fig. 2.** Comparison traces of (a) IS and (b) BMIS

**Algorithm 1:** Brownian Motus Insertion Sort

**Input** : A list of uniformly distributed elements *a_list*
**Required** : A Sorted list
**Initialize** : $POP \leftarrow 0$ ▷ *POP is position pointer*
**for** $i \leftarrow 1$ *to* $length(a\_list) - 1$ **do** ▷ *a_list start at index 0*
  $COP \leftarrow i$ ▷ *COP is current pointer*
  $key \leftarrow a\_list[COP]$
  **if** $key \geq a\_list[POP]$ **then** ▷ *left or right movement decision*
    $place \leftarrow place\_finder\_right(a\_list, POP, COP - 1, key)$
  **else**
    $place \leftarrow place\_finder\_left(a\_list, 0, POP, key)$
  $position \leftarrow place$ ▷ *POP is updated*
  $a\_list \leftarrow place\_inserter(a\_list, place, current)$ ▷ *Insert COP in sorted list*
  $i \leftarrow i + 1$
**Function** *place_finder_right*$(a\_list, start, end, key)$**:**
  **for** $j \leftarrow start$ *to* $end$ **do**
    **if** $key \geq a\_list[j]$ ***and*** $key \leq a\_list[j + 1]$ **then**
      $loc \leftarrow j + 1$ ▷ *location to insert COP right of POP*
      **return** loc
    $j \leftarrow j + 1$
**Function** *place_finder_left*$(a\_list, start, end, key)$**:**
  **for** $j \leftarrow end$ *to* $start$ **do**
    **if** $j \neq 0$ **then**
      **if** $key \leq a\_list[j]$ ***and*** $key \geq a\_list[j - 1]$ **then**
        $loc \leftarrow j$ ▷ *location to insert COP left of POP*
        **return** loc
    **else**
      **if** $key \leq a\_list[0]$ **then**
        $loc \leftarrow 0$ ▷ *location to insert COP left of POP*
        **return** loc
    $j \leftarrow j - 1$

ACCEPTED MANUSCRIPT

perform binary search operation and location identification to insert an element at $COP$ into either left or right selected subpart. $POP$ is updated to the latest location value. Insertion and shifting operation is handled by *place_inserter* function. Final outcome after completion of all iterations is a sorted list of elements. This name for the proposed algorithm CBIS has been chosen for the reason that after the decision of subpart, a cluster of elements is formed ranging $[0, POP-1]$ for the left subpart and $[POP+1, COP-1]$ for the right subpart. Therefore, due to the use of binary search logic on a cluster of elements, it is named as Clustered Binary Insertion Sort. CBIS is much more efficient than IS, BIS and BMIS in terms of a number of comparisons required to identify correct location for the element at $COP$. For step by step explanation of CBIS, refer Algorithm 2; and for *place_inserter* function, refer Algorithm 3.

**Algorithm 2:** Clustered Binary Insertion Sort

```
Input     : A list of uniformly distributed elements a_list
Required  : A Sorted list
Initialize : POP ← 0                                   ▷ POP is position pointer
for i ← 1 to length(a_list) − 1 do                     ▷ a_list start at index 0
    COP ← i                                            ▷ COP is current pointer
    key ← a_list[COP]
    if key ≥ a_list[POP] then                          ▷ left or right movement decision
        place ←binary_loc_finder(a_list, POP + 1, COP − 1, key)   ▷ right movement
    else
        place ←binary_loc_finder(a_list, 0, POP − 1, key)         ▷ left movement
    position ← place                                   ▷ POP is updated
    a_list ← place_inserter(a_list, place, current)    ▷ Insert COP in sorted list
    i ← i + 1
Function binary_loc_finder(a_list, start, end, key):
    if start == end then
        if a_list[start] > key then
            loc ← start
            return loc
        else
            loc ← start + 1
            return loc
    if start > end then
        loc ← start
        return loc
    else
        middle ← ⌊(start+low)/2⌋
        if a_list[middle] < key then
            return binary_loc_finder(a_list, middle + 1, end, key)
        else if a_list[middle] > key then
            return binary_loc_finder(a_list, start, middle − 1, key)
        else
            return middle
```

In both BMIS and CBIS, it is assumed that the first element of a list is at index zero; and the last

ACCEPTED MANUSCRIPT

**Algorithm 3:** Shifter

```
Function place_inserter(a_list, start, end):
    temp ← a_list[end]
    for k ← end to start do
        a_list[k] ← a_list[k − 1]
        k ← k − 1
    a_list[start] ← temp
    return a_list
```

element is at index one less than the length of an input list. *a_list* is the initial input list to the proposed algorithms of this work. Initially, the element at index zero of *a_list* will be the only element in the sorted part of a list. While initializing, $POP$ will be at index zero; and $COP$ at index one of input *a_list*. The trace of comparisons required by elements of INPUT SEQUENCE and process of location identification for the entire sorting is shown in Fig. 2.

## 5. Example

Working of the BMIS and CBIS is illustrated on a list of 16 random elements. The entire list of random elements is generated using uniform distribution on an interval of two digit numbers. In this paper, following color coding has been used to show the example: blue color denotes $POP$, red color $COP$ and ashgrey background represents sorted list of elements. After each step in the example a number of comparisons ($C_N$) have been required; and elements with which comparison has been made by the element at $COP$ is separately mentioned for both BMIS and CBIS. The order in which comparisons are mentioned is exactly the same order in which they are made by element at $COP$. Insertion will be performed at identified location in the next step; and $POP$ is updated to new index value, i.e., an identified location. The total number of comparisons ($C_T$) get their mention after the last step. Here, only two-digit numbers have been taken for sorting process just for the sake of representational simplicity in this paper, otherwise, any number of digits can be taken.

Step by step working of BMIS and CBIS is illustrated in Table 2. Linear search is performed by BMIS on selected subpart starting with $POP$ to find the correct location to insert the element at $COP$ whereas binary search for CBIS. Boldly-styled elements are used to show a list of elements on which binary search is performed.

**Table 2**
Example of BMIS and CBIS

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 46 | 24 | 25 | 71 | 72 | 84 | 60 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Initialize* |
| BMIS | | 24 compared with 46 | | | | | | | | | | $C_N = 1$ | | | | |
| CBIS | | 24 compared with 46 | | | | | | | | | | $C_N = 1$ | | | | |

ACCEPTED MANUSCRIPT

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 24 | **46** | 25 | 71 | 72 | 84 | 60 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-1* |
| BMIS | 25 compared with 24,46 | | | | | | | | | | | $C_N = 2$ | | | | |
| CBIS | 25 compared with 24,46 | | | | | | | | | | | $C_N = 2$ | | | | |
| 24 | 25 | **46** | 71 | 72 | 84 | 60 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-2* |
| BMIS | 71 compared with 25,46 | | | | | | | | | | | $C_N = 2$ | | | | |
| CBIS | 71 compared with 25,46 | | | | | | | | | | | $C_N = 2$ | | | | |
| 24 | 25 | 46 | 71 | 72 | 84 | 60 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-3* |
| BMIS | 72 compared with 71 | | | | | | | | | | | $C_N = 1$ | | | | |
| CBIS | 72 compared with 71 | | | | | | | | | | | $C_N = 1$ | | | | |
| 24 | 25 | 46 | 71 | 72 | 84 | 60 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-4* |
| BMIS | 84 compared with 72 | | | | | | | | | | | $C_N = 1$ | | | | |
| CBIS | 84 compared with 72 | | | | | | | | | | | $C_N = 1$ | | | | |
| **24** | **25** | **46** | **71** | **72** | 84 | 60 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-5* |
| BMIS | 60 compared with 84,72,71,46 | | | | | | | | | | | $C_N = 4$ | | | | |
| CBIS | 60 compared with 84,46,71 | | | | | | | | | | | $C_N = 3$ | | | | |
| 24 | 25 | 46 | 60 | **71** | **72** | **84** | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-6* |
| BMIS | 87 compared with 60,71,72,84 | | | | | | | | | | | $C_N = 4$ | | | | |
| CBIS | 87 compared with 60,72,84 | | | | | | | | | | | $C_N = 3$ | | | | |
| 24 | 25 | 46 | 60 | 71 | 72 | 84 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-7* |
| BMIS | 91 compared with 87 | | | | | | | | | | | $C_N = 1$ | | | | |
| CBIS | 91 compared with 87 | | | | | | | | | | | $C_N = 1$ | | | | |
| 24 | 25 | 46 | 60 | 71 | 72 | 84 | 87 | 91 | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-8* |
| BMIS | 96 compared with 91 | | | | | | | | | | | $C_N = 1$ | | | | |
| CBIS | 96 compared with 91 | | | | | | | | | | | $C_N = 1$ | | | | |
| **24** | **25** | **46** | **60** | **71** | **72** | **84** | **87** | **91** | 96 | 45 | 20 | 61 | 48 | 22 | 21 | *Step-9* |
| BMIS | 45 compared with 96,91,87,84,72,71,60,46,25 | | | | | | | | | | | $C_N = 9$ | | | | |
| CBIS | 45 compared with 96,71,25,46 | | | | | | | | | | | $C_N = 4$ | | | | |
| **24** | **25** | 45 | 46 | 60 | 71 | 72 | 84 | 87 | 91 | 96 | 20 | 61 | 48 | 22 | 21 | *Step-10* |
| BMIS | 20 compared with 45,25,24 | | | | | | | | | | | $C_N = 3$ | | | | |
| CBIS | 20 compared with 45,24 | | | | | | | | | | | $C_N = 2$ | | | | |
| 20 | **24** | **25** | **45** | **46** | **60** | **71** | **72** | **84** | **87** | **91** | **96** | 61 | 48 | 22 | 21 | *Step-11* |
| BMIS | 61 compared with 20,24,25,45,46,60,71 | | | | | | | | | | | $C_N = 7$ | | | | |
| CBIS | 61 compared with 20,71,45,46,60 | | | | | | | | | | | $C_N = 5$ | | | | |
| **20** | **24** | **25** | **45** | **46** | **60** | 61 | 71 | 72 | 84 | 87 | 91 | 96 | 48 | 22 | 21 | *Step-12* |
| BMIS | 48 compared with 61,60,46 | | | | | | | | | | | $C_N = 3$ | | | | |
| CBIS | 48 compared with 61,25,46,60 | | | | | | | | | | | $C_N = 4$ | | | | |
| **20** | **24** | **25** | **45** | **46** | 48 | 60 | 61 | 71 | 72 | 84 | 87 | 91 | 96 | 22 | 21 | *Step-13* |

ACCEPTED MANUSCRIPT

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BMIS | 22 compared with 48,46,45,25,24,20 | | | | | | | | | | | $C_N = 6$ | | | |
| CBIS | 22 compared with 48,25,20,24 | | | | | | | | | | | $C_N = 4$ | | | |
| **20** | 22 | 24 | 25 | 45 | 46 | 48 | 60 | 61 | 71 | 72 | 84 | 87 | 91 | 96 | 21 | *Step-14* |
| BMIS | 21 compared with 22,20 | | | | | | | | | | | $C_N = 2$ | | | |
| CBIS | 21 compared with 22,20 | | | | | | | | | | | $C_N = 2$ | | | |
| 20 | 21 | 22 | 24 | 25 | 45 | 46 | 48 | 60 | 61 | 71 | 72 | 84 | 87 | 91 | 96 | *Sorted* |

The input list as exhibited in the table given above is also sorted using IS and BIS; and counted total number of comparisons required is 78 and 46 respectively. But in comparison to highly popular IS and its variant BIS, the proposed BMIS and CBIS has less number of comparisons, i.e., $C_T = 47$ and $C_T = 36$ respectively.

## 6. Theoretical Analysis

The time complexity of the proposed algorithms chiefly depends on the process of location identification. For BMIS, location identification is handled by $place\_finder\_left$ and $place\_finder\_right$ functions, whereas in CBIS, $binary\_loc\_finder$ function is used to handle it. Further, complexity of location identifying functions depend on the number of comparisons required to insert an element from unsorted part of a list into sorted part. The time complexity of an algorithm is also known as running time of an algorithm. The complexity analysis aims at estimating the effect of size and type of input sequence scenario on running of an algorithm. This section provides a theoretical analysis showing the complexity of BMIS and CBIS algorithms which has been confirmed by experimental evaluations in the next section.

The running time of an algorithm has been calculated by adding together the running of time for each executed instruction. Execution cost of an $i^{th}$instruction is represented by $co_i$. If an instruction executes $n$ time, then, running time this instruction is contributing to total running of an algorithm is $n * co_i$. For the sake of simplicity, $co_i$ for each instruction is the same and constant. Total running time of the proposed algorithms is represented by $T(n)$ in which major contribution is of instructions executing location identifying function as shown by $t_l$ in Eq. (2).

$$T(n) = co_1(1)+co_2(n)+co_3(n-1)+co_4(n-1)+co_5(n-1)+(co_6 \text{ or } co_7)\sum_{l=1}^{n-1} t_l+co_8(n-1)+co_9(n-1)+co_{10}(n-1) \tag{2}$$

The same equation has been used for BMIS and CBIS, but in the case of BMIS, $t_l$ is a linear function, while it is a logarithmic function for CBIS. For more details of cost and time taken by each instruction, refer Algorithms 4 and 5.

ACCEPTED MANUSCRIPT

**Algorithm 4:** Complexity analysis of BMIS

**Input :** A list of uniformly distributed elements *a_list*
**Required :** A Sorted list
**Initialize :** $POP \leftarrow 0$ ▷ $co_1(1)$
**for** $i \leftarrow 1$ *to* $length(a\_list) - 1$ **do** ▷ $co_2(n)$, *where* $n = length(a\_list)$
  $COP \leftarrow i$ ▷ $co_3(n-1)$
  $key \leftarrow a\_list[COP]$ ▷ $co_4(n-1)$
  **if** $key \geq a\_list[POP]$ **then** ▷ $co_5(n-1)$
    $place \leftarrow place\_finder\_right(a\_list, POP, COP-1, key)$ ▷ $co_6 \sum_{l=1}^{n-1} t_l$
  **else**
    $place \leftarrow place\_finder\_left(a\_list, 0, POP, key)$ ▷ $co_7 \sum_{l=1}^{n-1} t_l$
  $position \leftarrow place$ ▷ $co_8(n-1)$
  $a\_list \leftarrow place\_inserter(a\_list, place, current)$ ▷ $co_9(n-1)$
  $i \leftarrow i+1$ ▷ $co_{10}(n-1)$
**Function** *place_finder_right*$(a\_list, start, end, key)$**:**
  **for** $j \leftarrow start$ *to* $end$ **do** ▷ $co_{11}(l)$, *where* $l = end - start + 1$
    **if** $key \geq a\_list[j]$ ***and*** $key \leq a\_list[j+1]$ **then** ▷ $co_{12}(l-1)$
      $loc \leftarrow j+1$ ▷ $co_{13}(1)$
      **return** loc ▷ $co_{14}(1)$
    $j \leftarrow j+1$ ▷ $co_{15}(l-1)$
**Function** *place_finder_left*$(a\_list, start, end, key)$**:**
  **for** $j \leftarrow end$ *to* $start$ **do** ▷ $co_{16}(l)$
    **if** $j \neq 0$ **then** ▷ $co_{17}(l-1)$
      **if** $key \leq a\_list[j]$ ***and*** $key \geq a\_list[j-1]$ **then** ▷ $co_{18}(l-1)$
        $loc \leftarrow j$ ▷ $co_{19}(1)$
        **return** loc ▷ $co_{20}(1)$
    **else**
      **if** $key \leq a\_list[0]$ **then** ▷ $co_{21}(1)$
        $loc \leftarrow 0$ ▷ $co_{22}(1)$
        **return** loc ▷ $co_{23}(1)$
    $j \leftarrow j-1$ ▷ $co_{24}(l-1)$

### *6.1. Best case analysis*

In the best case analysis, lower bound of the running time of an algorithm is calculated. Running time of an algorithm is considered in terms of a number of comparisons required. Two sequence scenarios have been considered for the best case analysis. In the first scenario, input list is already sorted in an ascending order; and for the second, input list is already sorted in a descending order. Our task is to calculate a number of comparisons required to sort the input list in both the scenarios into an ascending order. The best case complexity for both BMIS and CBIS is the same. Following is the proof for BMIS.

**Scenario 1 : List already sorted in an ascending order**

In order to sort a list, an element at $COP$ is compared with the element at $POP$, and then with the element in right subpart because the element at $COP$ is greater than element at $POP$. But in this case, right subpart will be empty in each iteration as $POP$ is at a maximum index of sorted part. So, there will be

ACCEPTED MANUSCRIPT

only one comparison per iteration resulting $t_l$ equals to one in Eq. (2). When taken together for the entire sorting process, there are $O(n)$ comparisons. For the present scenario, $co_7 = 0$ because $place\_finder\_left$ function will not execute in any iteration.

$$\begin{aligned}
T(n) &= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + co_6\sum_{l=1}^{n-1} 1 + co_8(n-1) + co_9(n-1) \\
&\quad + co_{10}(n-1) \\
&= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + co_6(n-1) + co_8(n-1) + co_9(n-1) \\
&\quad + co_{10}(n-1) \\
&= n(co_2 + co_3 + co_4 + co_5 + co_6 + co_8 + co_9 + co_{10}) + co_1 - (co_2 + co_3 + co_4 + co_5 + co_6 + co_8 \\
&\quad + co_9 + co_{10}) \\
&= an + b \\
&= O(n)
\end{aligned}$$

$a = co_2 + co_3 + co_4 + co_5 + co_6 + co_8 + co_9 + co_{10})$

$b = co_1 - (co_2 + co_3 + co_4 + co_5 + co_6 + co_8 + co_9 + co_{10})$

**Scenario 2 : List already sorted in a descending order**

In order to sort a list, the element at $COP$ is compared with the element at $POP$, and then with an element in the left subpart because the element at $COP$ is less than the element at $POP$. But in this case, left subpart will be empty in each iteration as $POP$ is at a minimum index of sorted part. So, there will be only a single comparison per iteration resulting $t_l$ equals to one in Eq. (2). When taken together for the entire sorting process, there are $O(n)$ comparisons. For the present scenario, $co_6 = 0$ because $place\_finder\_right$ function will not execute in any iteration.

$$\begin{aligned}
T(n) &= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + co_7\sum_{l=1}^{n-1} 1 + co_8(n-1) + co_9(n-1) \\
&\quad + co_{10}(n-1) \\
&= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + co_7(n-1) + co_8(n-1) + co_9(n-1) \\
&\quad + co_{10}(n-1) \\
&= n(co_2 + co_3 + co_4 + co_5 + co_7 + co_8 + co_9 + co_{10}) + co_1 - (co_2 + co_3 + co_4 + co_5 + co_7 + co_8 \\
&\quad + co_9 + co_{10}) \\
&= an + b \\
&= O(n)
\end{aligned}$$

$a = co_2 + co_3 + co_4 + co_5 + co_7 + co_8 + co_9 + co_{10})$

$b = co_1 - (co_2 + co_3 + co_4 + co_5 + co_7 + co_8 + co_9 + co_{10})$

ACCEPTED MANUSCRIPT

**Algorithm 5:** Complexity analysis of CBIS

**Input** : A list of uniformly distributed elements *a_list*
**Required** : A Sorted list
**Initialize** : $POP \leftarrow 0$ ▷ $co_1(1)$
**for** $i \leftarrow 1$ *to* $length(a\_list) - 1$ **do** ▷ $co_2(n)$, *where* $n = length(a\_list)$
  $COP \leftarrow i$ ▷ $co_3(n-1)$
  $key \leftarrow a\_list[COP]$ ▷ $co_4(n-1)$
  **if** $key \geq a\_list[POP]$ **then** ▷ $co_5(n-1)$
    $place \leftarrow binary\_loc\_finder(a\_list, POP+1, COP-1, key)$ ▷ $co_6 \sum_{l=1}^{n-1} t_l$
  **else**
    $place \leftarrow binary\_loc\_finder(a\_list, 0, POP-1, key)$ ▷ $co_7 \sum_{l=1}^{n-1} t_l$
  $position \leftarrow place$ ▷ $co_8(n-1)$
  $a\_list \leftarrow place\_inserter(a\_list, place, current)$ ▷ $co_9(n-1)$
  $i \leftarrow i+1$ ▷ $co_{10}(n-1)$

There is similarity in best case complexity analysis of CBIS and BMIS except only one difference, i.e., for BMIS value of $t_l = 1$ and for CBIS $t_l = \log 0 = 1$ in the initial step. The $t_l$ is a logarithmic function in CBIS; and number of elements applied on log is zero. Unlike Traditional Insertion Sort (IS) and Binary Insertion Sort (BIS), the scenario 2 is the best case scenario for BMIS and CBIS with $O(n)$ comparisons, but for IS and BIS, it is their worst case scenario with $O(n^2)$ comparisons. So, it proves that the proposed algorithms are far better than the existing ones.

### *6.2. Average case analysis*

In average case analysis, generally, a running time of the algorithm is nearly half of the worst case. Input sequence will be of the pattern in which element at $COP$ can be inserted at a location for which on an average half of the elements, either in left or right subpart, are compared starting with $POP$. If the element at $COP$ is less than element at $POP$, then, elements of left subpart are compared otherwise elements in right subpart are compared. Most probably, $POP$ will be at a location closer to mid of sorted part. In each iteration, $POP$ is updated to new location value. So, our task is to calculate number of comparisons required to sort such input sequence in an ascending order.

If a list is sorted using BMIS, then, to search correct location for the element at $COP$, a linear search is performed on selected subpart starting with the element at $POP$. A number of comparisons required will be on an average equal to half of length of selected subpart. In other words, only one-fourth of elements in sorted part are compared. So, to sort $(i+1)^{th}$ element number of comparisons required is $i/4$, resulting $t_l$ equals to $l/4$ in Eq. (2). The total number of comparisons required is $O(\sqrt[0.54]{n})$ for the whole sorting process. Mathematical proof for BMIS is as follows:

ACCEPTED MANUSCRIPT

$$T(n) = co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7)\sum_{l=1}^{n-1}\frac{l}{4} + co_8(n-1) + co_9(n-1) + co_{10}(n-1)$$
$$= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7)\Big(\frac{n(n-1)}{8}\Big) + co_8(n-1) + co_9(n-1) + co_{10}(n-1)$$
$$= \frac{n^2}{8}(co_6 \text{ or } co_7) + n(co_2 + co_3 + co_4 + co_5 - \frac{(co_6 \text{ or } co_7)}{8} + co_8 + co_9 + co_{10}) + co_1 - (co_3 + co_4 + co_5 + co_8 + co_9 + co_{10})$$
$$= a\frac{n^2}{8} + bn + c$$
$$= O(\frac{n^2}{8}) \subseteq O(\sqrt[0.54]{n})$$

$a = co_6$ or $co_7$

$b = co_2 + co_3 + co_4 + co_5 + (co_6 \text{ or } co_7)/8 + co_8 + co_9 + co_{10}$

$c = co_1 - (co_3 + co_4 + co_5 + co_8 + co_9 + co_{10})$

If the list is sorted using CBIS, then, to search a correct location for the element at $COP$ a binary search is performed on selected subpart. To sort $(i+1)^{th}$ element, number of comparisons required is $\log(i/2)$ because most probably $POP$ will lie in a location closer to middle of sorted part. So, there will be about 50% elements in selected subpart resulting $t_l$ equals to $\log(l/2)$ in Eq. (2). The total number of comparisons required is $O(n\log(n))$ for the whole sorting process. Mathematical proof for CBIS is as follows:

$$T(n) = co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7)\sum_{l=1}^{n-1}\log(\frac{l}{2}) + co_8(n-1) + co_9(n-1) + co_{10}(n-1) \quad \Big[\sum_{l=1}^{n-1}\log(\frac{l}{2}) = (n-1)\log(n-1)\ refer\ Eqs.\ (3) to\ (8)\ for\ detail\Big]$$
$$= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7)((n-1)\log(n-1)) + co_8(n-1) + co_9(n-1) + co_{10}(n-1)$$
$$= ((n-1)\log(n-1))(co_6 \text{ or } co_7) + n(co_2 + co_3 + co_4 + co_5 + co_8 + co_9 + co_{10}) + co_1 - (co_3 + co_4 + co_5 + co_8 + co_9 + co_{10})$$
$$= a((n-1)\log(n-1)) + bn + c$$
$$= O((n-1)\log(n-1)) \approx O(n\log(n))$$

$a = co_6$ or $co_7$

$b = co_2 + co_3 + co_4 + co_5 + co_8 + co_9 + co_{10}$

$c = co_1 - (co_3 + co_4 + co_5 + co_8 + co_9 + co_{10})$

ACCEPTED MANUSCRIPT

$$\sum_{l=1}^{n-1} \log(\frac{l}{2}) = \log(\frac{1}{2}) + \log(\frac{2}{2}) + \log(\frac{3}{2}) + ..... + \log(\frac{n-1}{2}) \tag{3}$$

$$= \log(\frac{1}{2} * \frac{2}{2} * \frac{3}{2} * ... * \frac{n-1}{2}) \tag{4}$$

$$= \log\left(\frac{(n-1)!}{2n}\right) \tag{5}$$

$$= \log(n-1)! - \log(2n) \tag{6}$$

$$= \log(n-1)! \tag{7}$$

$$= \log(n-1)! \subseteq (n-1)\log(n-1) \tag{8}$$

Contribution of $\log(2n)$ computed in Eq. (6) is neglected in further calculations as its impact is negligible as compared to $\log(n-1)!$. The average case complexity of BMIS is $O(\sqrt[0.54]{n})$, while it is $O(n^2)$ for IS. The asymptotic complexity of BIS and CBIS is equal, but practically the number of comparisons required by CBIS is quite less in comparison to BIS.

*The average case scenario of IS:* The sequence scenario for which each element at the start index of an unsorted part is less than half of the elements in a sorted part. It is an average case scenario of insertion sort for which $(i+1)^{th}$ element makes $i/2$ comparisons. For the whole sorting process, number of comparisons is $O(n*(n-1)/4)$. If BMIS is used to sort this sequence, then, to sort $i^{th}$ element only 2 comparisons are required; one with $POP$ itself to decide which subpart to select, and another is with the element at location $[POP-1]$ for the left subpart or at $[POP+1]$ for the right subpart. For the whole sorting process, the number of comparisons is $\Theta(2n)$. So, there is a huge difference between a number of comparisons required by BMIS and IS.

### *6.3. Worst case analysis*

In the worst case analysis, upper bound of running time of an algorithm is calculated. A sequence like input sequence as shown in Fig. 3 is the only sequence scenario that will result into the worst case complexity. Our task is to calculate number of comparisons to sort input sequence shown in Fig. 3 in an ascending order.

INPUT SEQUENCE : $\{x, x+i, x-i, x+2i, x-2i, x+3i, x-3i\}$

$x$

$x, x+i$

$x-i, x, x+i$

$x-i, x, x+i, x+2i$

$x-2i, x-i, x, x+i, x+2i$

$x-2i, x-i, x, x+i, x+2i, x+3i$

$x-3i, x-2i, x-i, x, x+i, x+2i, x+3i$

**Fig. 3.** Worst case pattern pyramid

In order to sort a list, the element at $COP$ is compared with the element at $POP$, and then with

ACCEPTED MANUSCRIPT

elements either in the left or right subpart. In each iteration, different subparts will be selected starting with the right subpart for input sequence as shown in Fig. 3 or starting with the left subpart, if sequence is $x, x-i, x+i, x-2i, x+2i, x-3i, x+3i$. Stick to the input sequence as shown in Fig. 3, if the right subpart is selected, then, element at $COP$ is inserted at index one plus a maximum index of sorted part and $POP$ updated to a new location. If the left subpart is selected, then, element at $COP$ is inserted at zero of a sorted part and $POP$ updated to index zero. After completion of all iterations, a pyramid-like structure as shown in Fig. 3 is received as output, if only the behavior of sorted part is considered in each iteration. Each red colored element in the pyramid represents a $POP$ location for each iteration. A number of comparisons required to sort above list will be different for both BMIS and CBIS.

If a list is sorted using BMIS, then, a linear search is performed on the selected subpart starting with $POP$ to search a correct location. It can be easily observed from Fig. 3 that in a case of linear search to insert an element at $COP$ in the sorted part, the number of comparisons required is equal to the index of an element. As $i^{th}$ element is at index $i-1$, so to sort $i^{th}$ element $i-1$ comparisons are required resulting $t_l$ equals to $l$ in Eq. (2). For the whole sorting process, number of comparisons required $O(n^2)$. But the probability of getting a sequence scenario as shown in Fig. 3 is approximately zero. Mathematical proof for BMIS is as follows:

$$
\begin{aligned}
T(n) &= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7)\sum_{l=1}^{n-1} l + co_8(n-1) + co_9(n-1) \\
&\quad + co_{10}(n-1) \\
&= co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7)\Big(\frac{n(n-1)}{2}\Big) + co_8(n-1) \\
&\quad + co_9(n-1) + co_{10}(n-1) \\
&= \frac{n^2}{2}(co_6 \text{ or } co_7) + n(co_2 + co_3 + co_4 + co_5 - \frac{(co_6 \text{ or } co_7)}{2} + co_8 + co_9 + co_{10}) + co_1 - (co_3 + co_4 + co_5 \\
&\quad + co_8 + co_9 + co_{10}) \\
&= a\frac{n^2}{2} + bn + c \\
&= O(n^2)
\end{aligned}
$$

$a = co_6 \text{ or } co_7$

$b = co_2 + co_3 + co_4 + co_5 + (co_6 \text{ or } co_7)/2 + co_8 + co_9 + co_{10}$

$c = co_1 - (co_3 + co_4 + co_5 + co_8 + co_9 + co_{10})$

If a list is sorted using CBIS, then, a binary search is performed on the selected subpart to search a correct location. In a case of binary search, the number of comparisons required is equal to logarithmic of a number of elements in selected subpart. Therefore, to sort $n^{th}$ element, $\log(n-1)$ comparisons are required as there are $n-1$ elements in the selected subpart as shown in Fig. 3 resulting $t_l$ equals to $\log l$ in Eq. (2).

ACCEPTED MANUSCRIPT

For the whole sorting process, number of comparisons required is $O(n \log n)$. But the probability of getting a sequence as shown in Fig. 3 is approximately zero. Mathematical proof for CBIS is as follows:

$$T(n) = co_1(1) + co_2(n) + co_3(n-1) + co_4(n-1) + co_5(n-1) + (co_6 \text{ or } co_7) \sum_{l=1}^{n-1} \log l + co_8(n-1) + co_9(n-1) + co_{10}(n-1)$$

Rest of the proof has remained the same as the average case of CBIS. The worst case complexity of BMIS is $O(n^2)$ and that of CBIS is $O(n \log n)$, but the probability of getting the worst case sequence as highlighted in Fig. 3 is approximately zero. The worst case complexity of IS is $O(n^2)$; and that of BIS is $O(n \log n)$, but the sequence scenario which results into their worst case complexity is the best case scenario for both BMIS and CBIS.

## 7. Results and Discussions

The proposed algorithms have been implemented using Python 2.7.10 Shell. The results have been obtained on a Dell Workstation T5600- with 2.6GHz Intel Xenon e5 2650 CPU and 8GB 1600MHz DDR3 RAM, running windows platform. Experiments have been performed on elements randomly generated using uniform distribution. For complete visualization of the dataset used for experiments, refer to Section 3.

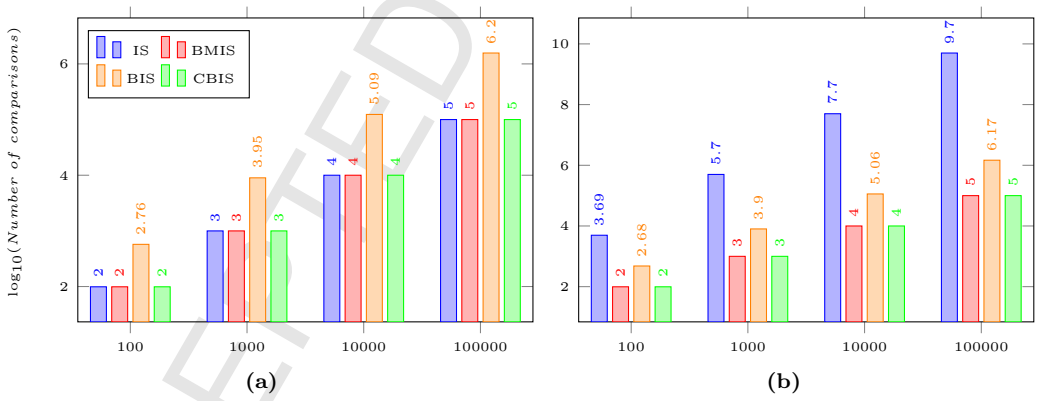



**Fig. 4.** Comparative analysis of IS, BMIS, BIS and CBIS for dataset (a) Sorted and (b) Reverse Sorted

Fig. 4 clearly illustrated logarithmic of a number of comparisons (Y-axis) required by IS and BIS sorting algorithms in comparison to the proposed BMIS and CBIS sorting algorithms on a sorted dataset. Both scenarios of sorted data, i.e., sorted in an increasing order and reverse order have been considered for the experimental evaluation as shown in Fig. 4 (a) and (b) respectively. The experiment has been performed with a different number of elements in a list (X-axis) for the trials. Fig. 4 (a) evidently explains that a number of comparisons required by IS, BMIS, and CBIS algorithms are equal i.e. $O(n)$ while sorting the

ACCEPTED MANUSCRIPT

elements already ascending order. But for the same dataset, a number of comparisons required by BIS are very high in comparison to other three algorithms. Dataset sorted in an increasing order is the best case scenario for most of the existing sorting algorithms. Both BMIS and CBIS are also giving the same or better performance for dataset sorted in an increasing order in comparison to the existing sorting algorithms. Fig. 4 (b) highlights that a number of comparisons required by IS and BIS algorithms are very high in comparison to the proposed BMIS and CBIS for sorting elements already in a reverse order into an ascending order. For IS and BIS number of comparison are $O(n^2)$ and $O(n\log(n))$ respectively. Contrary to it, BMIS and CBIS still have a number of comparisons $O(n)$. Dataset sorted in a reverse order is the worst case scenario for most of the existing sorting algorithms, but for BMIS and CBIS, it is their best case. The minimum number of comparisons which can be achieved by any sorting algorithm is $O(n)$. So, for both cases of a sorted dataset, the performance of proposed BMIS and CBIS is far better.

### *7.1. IS and BMIS*

The performance of IS is compared with BMIS algorithm in terms of a number of comparisons and execution time required to sort the elements into an ascending order. Extensive experiments have been performed to do the same with different percentages of randomness in the initial dataset. The entire dataset already described in Section 3 has been divided into three groups. For each group, the experiment has been repeated with four different levels of randomness, i.e., $100\%, 75\%, 50\%$, and $25\%$. For example, $x\%$ level of randomness means that $(100 - x)\%$ of elements in a set of an initial dataset is sorted; and remaining have been left without any sorting. Sorting of $(100 - x)\%$ of a number of elements is not done as a single chunk but in multiple chunks with some in the increasing order, while the others in reverse order. The only reason to take a different level of randomness is to get a new dataset for our experiment and analyze the impact of reduction in randomness level on number of comparisons required by sorting algorithms under consideration.

In Fig. 5, bar-plots clearly show the difference in a number of comparisons required by both IS and BMIS algorithms to sort the elements in a set. For each subplot, Y-axis represents the number of comparisons required, while X-axis indicates the a number of elements in a set. Case of 100% randomness in the dataset is exhibited in Fig. 5(a)-(c), where the number of elements in a set for the first group appears as [100,500,1000,1500,2000] in subsequent trials. For the second and third groups, the number of elements in a set in subsequent trials are [2000,4000,6000,8000,10000] and [20000,40000,60000,80000,100000] respectively. The number of comparisons required by both algorithms are mentioned over the respective color bars for each subsequent trial in a group. It can be seen from the bar plots that BMIS require only a limited number of comparisons to sort elements as compared to IS. The difference between the number of comparisons required by IS and BMIS has increased with further repetition of the experiment on a dataset with decreased levels of randomness as observed from Fig. 5.

ACCEPTED MANUSCRIPT

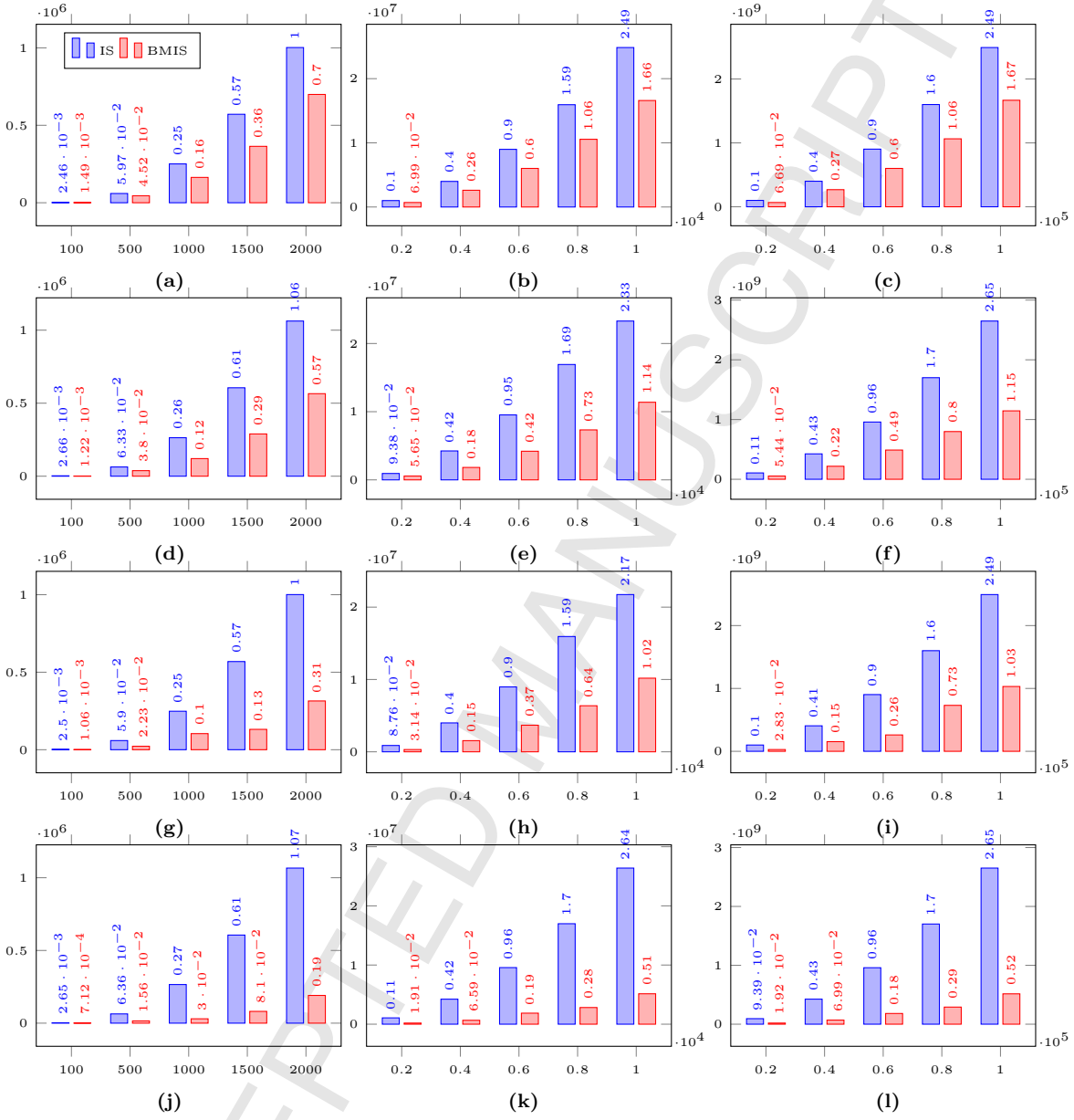



**Fig. 5.** Comparative analysis of number of comparisons required by IS vs BMIS for (a,b,c) 100% random data, (d,e,f) 75% random data, (g,h,i) 50% random data, (j,k,l) 25% random data

Experiment to evaluate the performance of proposed BMIS over traditional IS on the basis of execution time in *nanoseconds* is also performed. The same dataset which was used to measure the number of comparisons required while sorting the elements has been again used for the experiment. The results are depicted in Fig. 6, where respective colored markers represent the execution time (Y-axis) taken by both algorithms to sort a different number of elements in a subplot. It can be noticed from the pattern of colored lines that execution time requirement of proposed BMIS is far much less than the traditional IS sorting

ACCEPTED MANUSCRIPT

algorithm. To measure the execution time for every set of elements average over five iterations is considered. For the sorting of one hundred thousand random numbers with 100% randomness case shown in Fig. 6 (c) the execution time requirement of BMIS is approximately 35% less as compare to IS. As we further move down the column, with the decrease in randomness of data elements the execution time requirement of BMIS will also decreases as shown in Fig. 6 (l). Execution time for sorting by BMIS is approximately 66.8% less than IS.

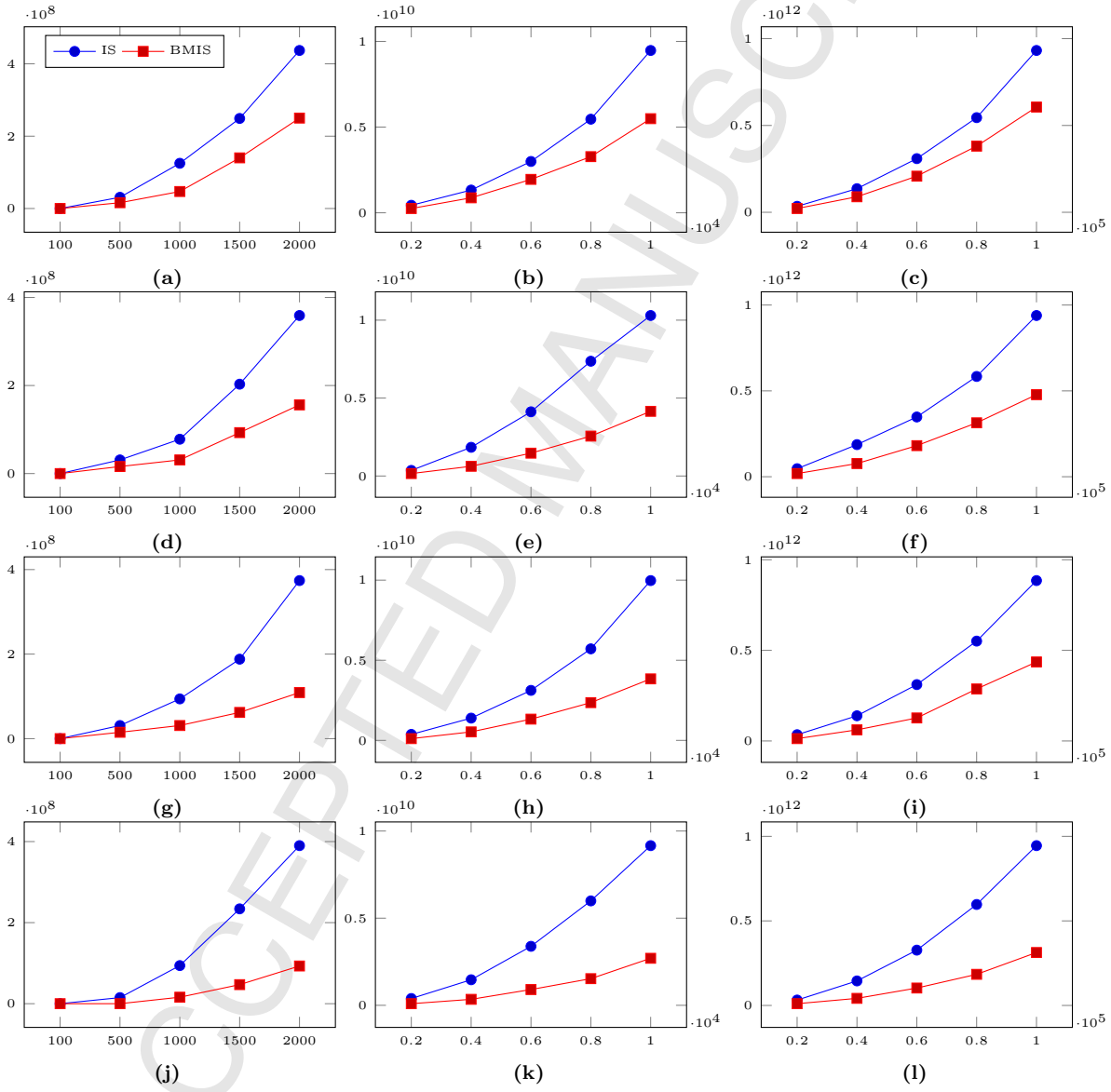



**Fig. 6.** Comparative analysis of execution time of IS vs BMIS for (a,b,c) 100% random data, (d,e,f) 75% random data, (g,h,i) 50% random data, (j,k,l) 25% random data

For each single percentage decrease in the randomness of dataset there is an increasing trend in the

ACCEPTED MANUSCRIPT

number of comparisons and execution time requirements for IS, but for BMIS reversed trends are observed from Figs. 5 and 6. If the reduction in randomness level continues, then it will result in a number of comparisons for IS to approach $O(n^2)$ and BMIS to $O(n)$. Thus, for nearly sorted datasets, proposed BMIS is the best choice. Some real life scenarios which result in a nearly sorted dataset are a case of error log files, which are nearly sorted due to cache and log engines. In a case of animated *gifs*, data is nearly sorted. This supports the claim of this research work that BMIS is far better than IS.

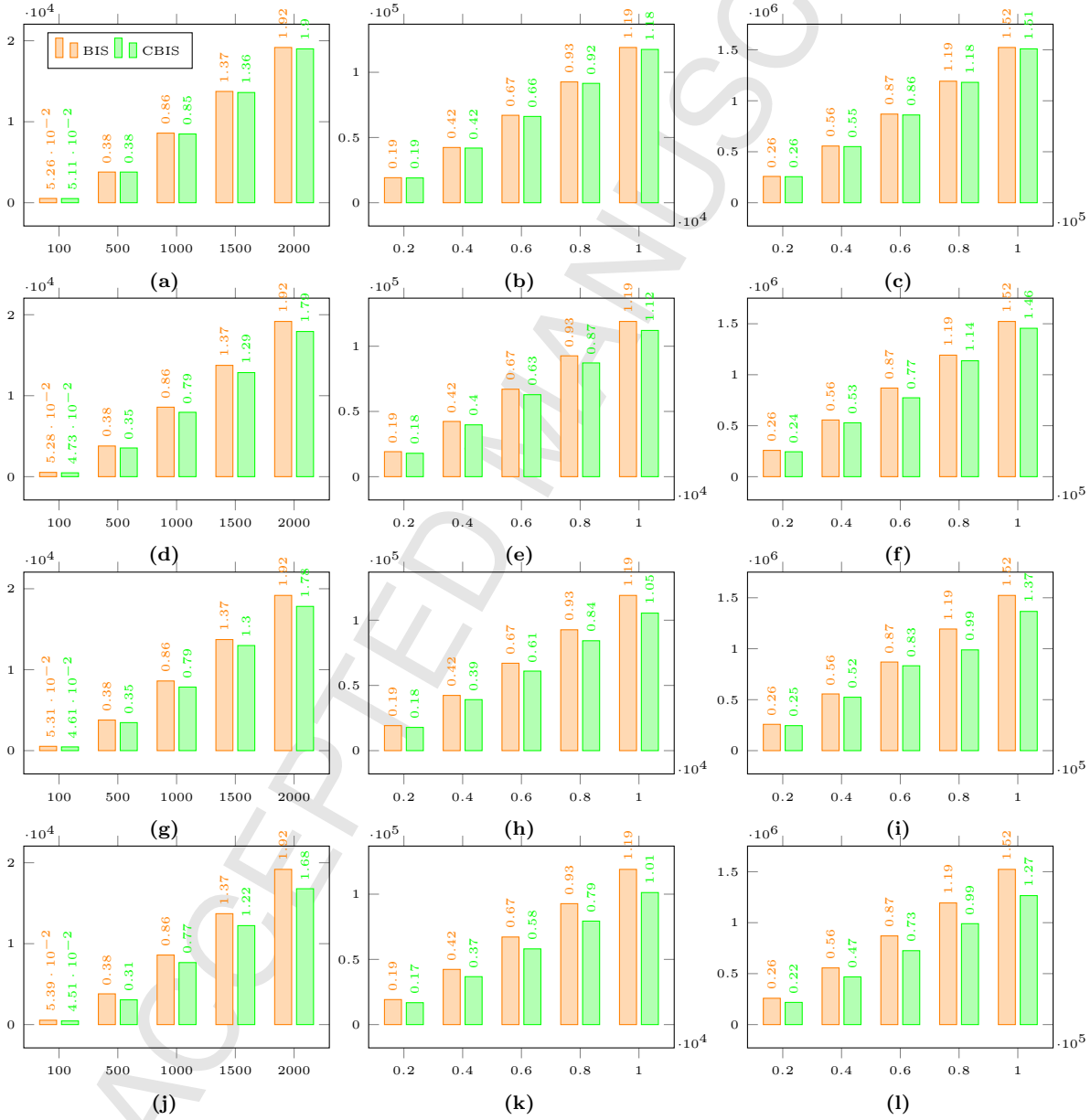



**Fig. 7.** Comparative analysis of number of comparisons required by BIS vs CBIS for (a,b,c) 100% random data, (d,e,f) 75% random data, (g,h,i) 50% random data, (j,k,l) 25% random data

ACCEPTED MANUSCRIPT

### *7.2. BIS and CBIS*

The performance of BIS has been compared with that of CBIS algorithm on the basis of a number of comparisons and execution time required to sort the elements of a dataset into an ascending order. Every experiment conducted to compare IS and BMIS in Sub Section 7.1 has been repeated to get the evaluation results for BIS and CBIS algorithm on the same datasets. The bar-plots shown in Fig. 7 clearly depicts that number of comparisons required by CBIS are much less than BIS to sort elements of a dataset.

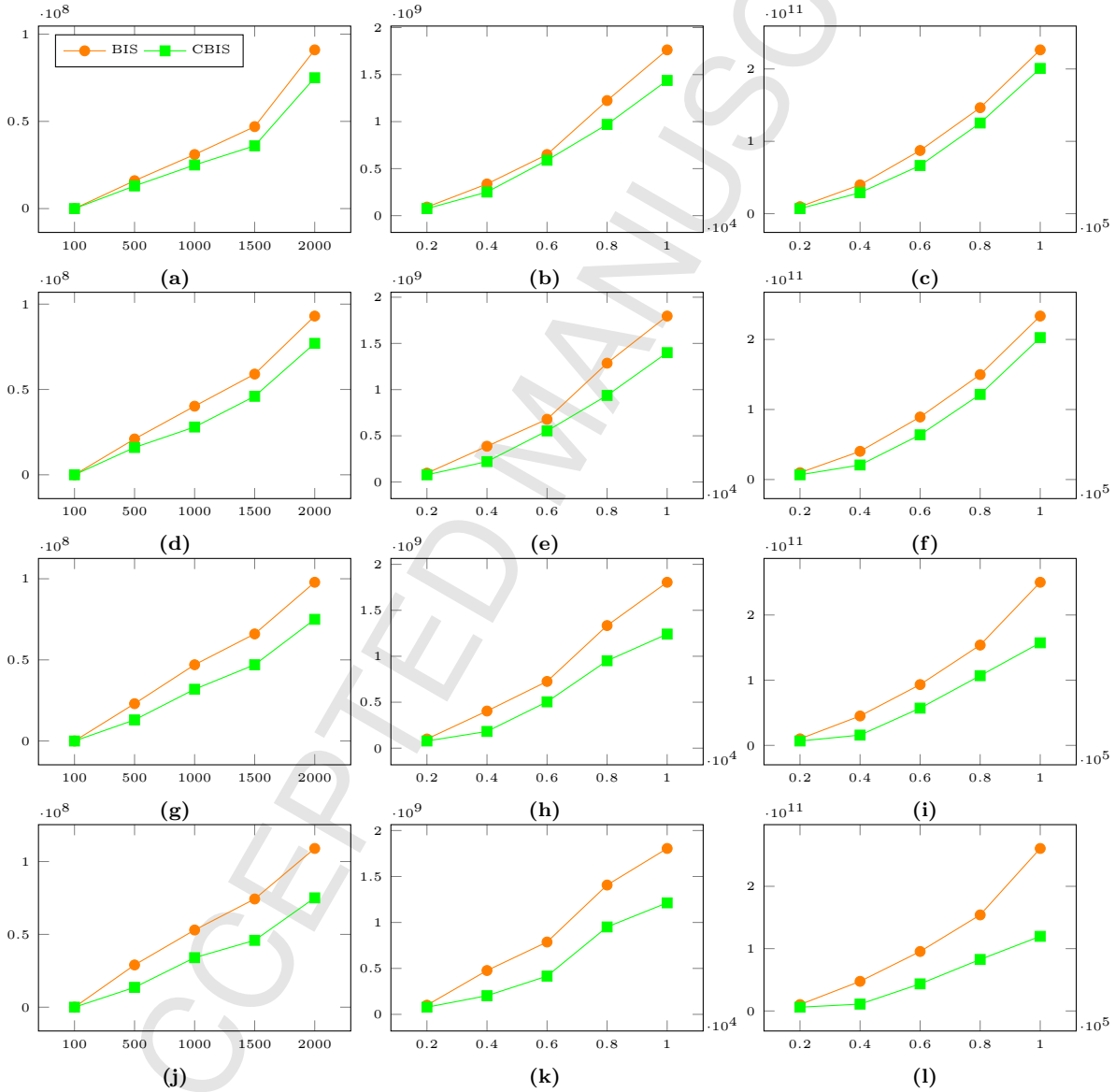



**Fig. 8.** Comparative analysis of execution time of BIS vs CBIS for (a,b,c) 100% random data, (d,e,f) 75% random data, (g,h,i) 50% random data, (j,k,l) 25% random data

A reduction in the level of randomness leads to increase the difference in a number of comparison

ACCEPTED MANUSCRIPT

operations required by BIS and CBIS. BIS tends to be $O(n \log n)$; and CBIS tends to be $O(n)$. The execution time required for sorting of the dataset by BIS and proposed CBIS is compared in Fig. 8. The trends represented by respective colored line plots are of similar type to the one for IS and BMIS i.e. increasing for BIS and decreasing for CBIS.

If the randomness is further decreased beyond 25%, then the nature of trends for both BMIS and CBIS will start changing from curve to flat lines i.e. $O(n)$, a slight picture of the scene can be seen in Figs. 6 and 8 (j)-(l). But the rate of change is different for BMIS and CBIS i.e. high for BMIS and low for CBIS, which can be clearly observed from the behavior of corresponding subplots in Figs. 6 and 8. The difference in execution times taken by BIS and CBIS measured from distance between respective line plots, which is not much high as compared to the one observed in case of IS and BMIS. The number of comparisons and execution time required for sorting of dataset in every case of randomness for every set of elements by CBIS is far much less than IS, BIS and BMIS according to results shown in Figs. 5 to 8.

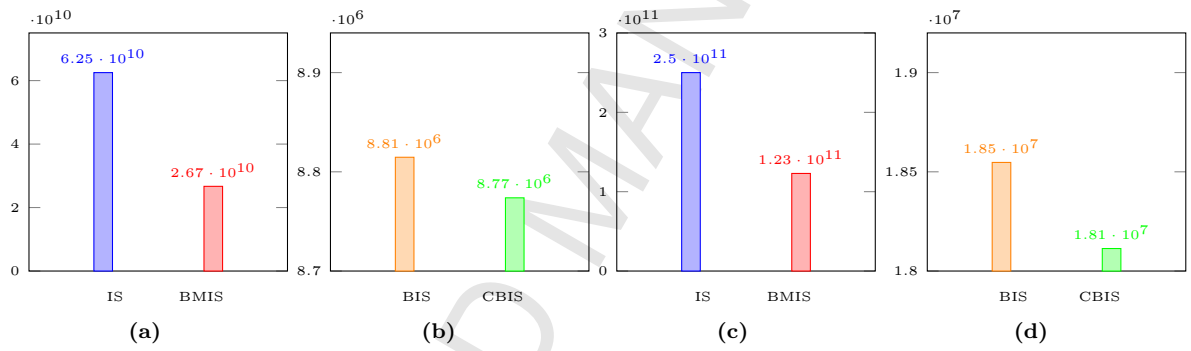



**Fig. 9.** Comparative analysis of number of comparisons required by IS vs BMIS and BIS vs CBIS to sort five hundred thousand (a)-(b) and one million (c)-(d) random number.

To show the effectiveness of proposed algorithms over traditional IS and its variant, the experiment has been also conducted on a much bigger dataset generated similarly to the one discussed in Section 3. The dataset of five hundred thousand and one million random numbers is used to count the number of comparisons (Y-axis) required by sorting algorithms under consideration to sort the elements into ascending order. But here only the case of 100% randomness has been considered. The obtained results depicted in Fig. 9 confirmed our claim of developing more efficient online sorting algorithm i.e. BMIS and CBIS, whose performance does not deteriorate even on the larger sized list, in comparison to its counterparts.

## 8. Conclusion

This paper proposes two new sorting algorithms, *viz.* Brownian Motus Insertion Sort (BMIS) and Clustered Binary Insertion Sort (CBIS). BMIS is a variant of traditional Insertion Sort (IS), while CBIS a variant of Binary Insertion Sort (BIS). The performance of both the proposed algorithms has shown a

ACCEPTED MANUSCRIPT

significant improvement over their traditional methods. BMIS has been compared with IS, while CBIS with BIS. Measuring parameter for performance is time complexity which, in turn, is measured on the basis of a number of comparisons required by algorithms under consideration. The sorting made for each algorithm under consideration has been in the ascending order.

Unnecessary comparisons have been avoided in the proposed algorithms by way of using a novel methodology for location identification. An extensive theoretical analysis has been performed to calculate the best, average and worst case complexity of IS, BIS, BMIS, and CBIS. It has been further confirmed by experimental evaluations. The results of analysis have revealed that the average case time complexity of BMIS is asymptotically equal to $O(\sqrt[0.54]{n})$, while that of CBIS is equal to $O(n \log n)$. The space complexity of both the proposed algorithms remains equal to $O(n)$. In the case of BMIS, the worst case time complexity is $O(n^2)$, whereas in CBIS it is $O(n \log n)$, but the probability of getting the worst case sequence as already discussed in section 6 is approximately zero. Reverse sorted list is the worst case scenario for IS and BIS with time complexity of $O(n^2)$ and $O(n \log n)$ respectively, but for BMIS and CBIS, it is their best case scenario with the complexity of only $O(n)$. Unlike IS and BIS, the performance of BMIS and CBIS is not affected by an increase in the size of a list. Both BMIS and CBIS have properties of *stable*, *comparison*, *in-place*, and *online sorting* algorithm.

Apart from the above, there are some other advantages of using BMIS and CBIS. Extensive experiments have been performed to compare IS, BIS, BMIS and CBIS on various sized (100, 500, 1K, 1.5K, 2K, 4K, 6K, 8K, 10K, 20K, 40K, 60K, 80K, 100K, 500K, 1000K) lists with different randomness levels, i.e., $100\%, 75\%, 50\%$ and $25\%$. Experimental results reveal that performance of BMIS and CBIS has increased with a decrease in randomness level of a dataset. This performance is far better than that of traditional methods such as IS and BIS. The number of comparisons and execution time required by BMIS and CBIS approach to $O(n)$ with a reduction in randomness level. Thus, for nearly sorted datasets the proposed BMIS and CBIS are the best choice.

## References


[1] T. H. Cormen, Introduction to algorithms, MIT Press, 2009.

[2] S. S. Skiena, The algorithm design manual: Text, Vol. 1, Springer Science & Business Media, 1998.

[3] Ö. Ergül, Sorting, in: Guide to Programming and Algorithms Using R, Springer London, 2013, pp. 99–115. doi:10.1007/978-1-4471-5328-3_6.

[4] R. Sedgewick, Algorithms, Pearson Education India, 1988.

[5] M. Khairullah, Enhancing worst sorting algorithms, International Journal of Advanced Science and Technology 56 (2013) 13–26.

[6] H. Mannila, Measures of presortedness and optimal sorting algorithms, IEEE Transactions on Computers 100 (4) (1985) 318–325. doi:10.1109/TC.1985.5009382.

[7] A. S. Mohammed, Ş. E. Amrahov, F. V. Çelebi, Bidirectional conditional insertion sort algorithm; An efficient progress on the classical insertion sort, Future Generation Computer Systems 71 (2017) 102–112. doi:10.1016/j.future.2017.01.034.

ACCEPTED MANUSCRIPT

[8] A. D. Mishra, D. Garg, Selection of best sorting algorithm, International Journal of Intelligent Information Processing 2 (2) (2008) 363–368.

[9] Y. Yang, P. Yu, Y. Gan, Experimental study on the five sort algorithms, in: Proceedings of 2nd International Conference on Mechanic Automation and Control Engineering (MACE), IEEE, 2011, pp. 1314–1317. doi:10.1109/MACE.2011.5987184.

[10] S. Beniwal, D. Grover, Comparison of various sorting algorithms: A review, International Journal Emerging Research in Management & Technology 2 (2013) 83–86.

[11] M. A. Bender, S. P. Fekete, T. Kamphans, N. Schweer, Maintaining arrays of contiguous objects, in: Proceedings of International Symposium on Fundamentals of Computation Theory, Springer, 2009, pp. 14–25. doi:10.1007/978-3-642-03409-1_3.

[12] C. R. Cook, D. J. Kim, Best sorting algorithm for nearly sorted lists, Communications of the ACM 23 (11) (1980) 620–624. doi:10.1145/359024.359026.

[13] T. Biedl, T. Chan, E. D. Demaine, R. Fleischer, M. Golin, J. A. King, J. I. Munro, Fun-sortor the chaos of unordered binary search, Discrete Applied Mathematics 144 (3) (2004) 231–236. doi:10.1016/j.dam.2004.01.003.

[14] M. A. Bender, M. Farach-Colton, M. A. Mosteiro, Insertion sort is o (n log n), Theory of Computing Systems 39 (3) (2006) 391–397. doi:10.1007/s00224-005-1237-z.

[15] N. Faujdar, S. P. Ghrera, A detailed experimental analysis of library sort algorithm, in: Proceedings of Annual IEEE India Conference (INDICON), IEEE, 2015, pp. 1–6. doi:10.1109/INDICON.2015.7443165.

[16] W. Min, Analysis on 2-element insertion sort algorithm, in: Proceedings of International Conference on Computer Design and Applications (ICCDA), Vol. 1, IEEE, 2010, pp. 143–146. doi:10.1109/ICCDA.2010.5541165.

[17] P. S. Dutta, An approach to improve the performance of insertion sort algorithm, International Journal of Computer Science & Engineering Technology 4 (2013) 503–505.

[18] F. Lam, R. K. Wong, Rotated library sort, in: Proceedings of the 19th Computing: The Australasian Theory Symposium-Volume 141, Australian Computer Society, Inc., 2013, pp. 21–26.

[19] K. Nenwani, V. Mane, S. Bharne, Enhancing adaptability of insertion sort through 2-way expansion, in: Proceedings of 5th International Conference on Confluence The Next Generation Information Technology Summit (Confluence), IEEE, 2014, pp. 843–847. doi:10.1109/CONFLUENCE.2014.6949294.

[20] S. Paira, A. Agarwal, S. S. Alam, S. Chandra, Doubly inserted sort: A partially insertion based dual scanned sorting algorithm, in: Emerging Research in Computing, Information, Communication and Applications, Springer, 2015, pp. 11–19. doi:10.1007/978-81-322-2550-8_2.

[21] L. Kuipers, H. Niederreiter, Uniform distribution of sequences, Courier Corporation, 2012.

[22] I. Niven, Uniform distribution of sequences of integers, Transactions of the American Mathematical Society 98 (1) (1961) 52–61. doi:10.2307/1993512.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

**Author's Biography**

**Shubham Goel** received B. Tech. degree in 2011 in Computer Science and Engineering from Maharishi Markandeshwar University, Mullana, Haryana, India. In 2015 he received M.E. degree in Computer Science and Engineering from Thapar Institute of Engineering & Technology (TIET), Patiala, India. He is currently a PhD research scholar at TIET, Patiala. His research interest includes machine learning, data mining, social network analysis and computer algorithms.

**Ravinder Kumar** received his PhD degree from the Thapar Institute of Engineering & Technology in 2015 in Computer Science. He joined TIET, Patiala, India, as an assistant professor in 2006. He has about fourteen years of teaching experience. He has already developed a complete working project on speech recognition and online handwritten recognition system for Indian regional language (Punjabi). His current research interests include theoretical and practical aspects of **algorithms**, **combinatorial optimization**, and **machine learning.** He is guiding many students pursuing ME and PhD in the area of algorithm designing.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Shubham Goel

Ravinder Kumar

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

**Highlights**

- We propose two new efficient sorting algorithms Brownian Motus Insertion Sort (BMIS) and Clustered Binary Insertion Sort (CBIS) over traditional insertion sort algorithm.
- We compare BMIS with traditional Insertion Sort and CBIS with Binary Insertion Sort.
- We prove mathematically and experimentally that the average time complexity of BMIS is $O\left(\sqrt[0.54]{n}\right)$ and CBIS is $O(n \log n)$.
- We also prove that for reverse sorted data complexity of both BMIS and CBIS is $O(n)$.
- Both BMIS and CBIS are *in-place*, *stable* and *online sorting* algorithm.

ACCEPTED MANUSCRIPT