# REVIEW ON PERFORMANCE OF INSERTION SORT ALGORITHM

Department of Computer Science and Engineering

University B.D.T. College of Engineering Davanagere-577004,Karnataka

Dr.Mohammed Rafi

Professor and HOD

DOS in Computer Science and Engineering

Mdrafi2km@yahoo.com

Bheemambhikeshwari.Katti

Computer Science and Engineering

bheemakatti173@gmail.com

Sukanya N

Computer Science and Engineering

sukanyanpoojar@gmail.com

## Abstract:

Sorting is an important of computer science field. Many sorting algorithms available,insertion sort decrease and conquer technique.Knuth(TAOCP 3,p.82) writes that the variant of using binary insertion "was mentioned by John Mauchly as early as 1946,in the first published discussion of computer sorting". Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

## Introduction:

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

Insertion sort with the efficiency of O(n2) is popular and known best for its performance among the other O(n2) sorting algorithm. It is adaptive to the presence of ordering among the elements when elements are in required order and shows a linear complexity (i.e. best case O(n)). However, when the ordering among the elements is not in required order, insertion sort shows its worst case behavior which is O(n2) quadratic in nature [1,2].

## Literature Survey :

Insertion sort, like many classical algorithms, doesn't have a single definitive inventor or author. However, its conceptual roots can be traced back to early work in the field of computer science and numerical analysis. Here are a few key figures and resources that have contributed to the understanding and popularization of the insertion sort:

1. Donald Knuth: Knuth's seminal work, "The Art of Computer Programming," is one of the most comprehensive resources on algorithms and data structures. In Volume 3, "Sorting and Searching," Knuth provides a detailed description and analysis of insertion sort, among other algorithms.

2. C.A.R. Hoare: Known for developing the quicksort algorithm, Hoare also contributed to the broader study of sorting algorithms, including discussions on insertion sort in the context of comparison-based sorting methods.

3.Robert Sedgewick: In his books and research, Sedgewick has explored a wide range of algorithms, including insertion sort. His works often include practical considerations and efficient implementations.

4. Jon Bentley: Bentley's book "Programming Pearls" contains practical advice and discussions on sorting techniques, including insertion sort, focusing on real-world applications and performance tuning.

5.Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: The authors of "Introduction to Algorithms," commonly known as "CLRS," provide a comprehensive introduction to insertion sort, along with a formal analysis of its performance and properties.

These authors and their works are foundational in the study of algorithms and provide a deep understanding of insertion sort, both from theoretical and practical perspectives.

## Methodology:

At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it.

Insertion sort works by iterating through a list of items, comparing each element to the ones that come before it, and inserting it into the correct position in the list. This process is repeated until the list is fully sorted. To illustrate this process, let's use the example of a list of numbers that we want to sort in ascending order:

### Step by step algorithm:

1. Initialize:

Start with an array A of n elements.

Assume the first element A[0] is already sorted.

2. Iterate Through the Array:

For each element A[i] from the second element to the last (i.e., from i = 1 to n - 1):

3. Set Key and Start Comparison:

Set key = A[i]. This key will be inserted into the sorted portion of the array.

Initialize j = i - 1. This represents the last index of the sorted portion.

4. Shift Elements to Make Space for the Key:

While j >= 0 and A[j] > key, do the following:

Shift A[j] to the right by setting A[j + 1] = A[j].

Decrement j by 1 (i.e., j = j - 1).

5. Insert the Key:

After exiting the while loop, insert the key into the correct position in the sorted portion by setting A[j + 1] = key.

6. Repeat:

Repeat steps 3 to 5 for the next element in the array.

7. End:

The array A is now sorted.

### Pseudocode:

```
function insertionSort(A):
    n = length(A)
    for i from 1 to n - 1 do
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
    return A
```

Example:

Given the array A = [5, 2, 9, 1, 5, 6]:

1. Initial Array: [5, 2, 9, 1, 5, 6]

2. Iteration 1 (i = 1):

key = 2

Compare and shift: 5 > 2, so A = [5, 5, 9, 1, 5, 6]

Insert key: A = [2, 5, 9, 1, 5, 6]

3. Iteration 2 (i = 2):

key = 9

9 is greater than 5, no shifting needed.

4. iteration 3(i = 3):

key = 1

Compare and shift: 9 > 1, 5 > 1, 2 > 1, so A = [2, 5, 9, 9, 5, 6], then A = [2, 5, 5, 9, 5, 6], then A = [2, 2, 5, 9, 5, 6]

Insert key: A = [1, 2, 5, 9, 5, 6]

5. Iteration 4 (i = 4):

key = 5

Compare and shift: 9 > 5, so A = [1, 2, 5, 9, 9, 6]

Insert key: A = [1, 2, 5, 5, 9, 6]

6. Iteration 5 (i = 5):

key = 6

Compare and shift: 9 > 6, so `A = [1, 2, 5,

**Example 1:**

[22, 6, 15, 48, 1].

Compare the first element, 22, to the second element, 6. Since 15 is smaller than 22, we swap them, resulting in the list [6, 22, 15, 48, 1].

Compare the second element, 22, to the third element, 15. Since 15 is smaller than 22, we swap them, resulting in the list [6, 15, 22, 48, 1].

Compare the third element, 22, to the fourth element, 48. Since 48 is larger than 22, no swap is necessary.

Compare the fourth element, 48, to the fifth element, 1. Since 1 is smaller than 48, we swap them, resulting in the list [6, 15, 22, 1, 48].

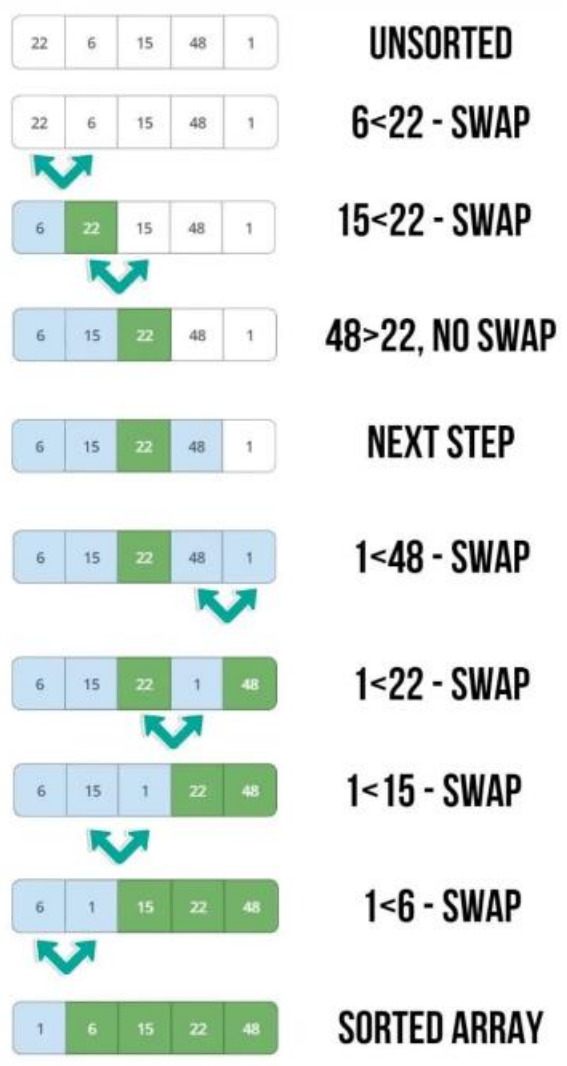
Compare the third element, 22, to the fourth element, 1. Since 1 is smaller than 22, we swap them, resulting in the list [6, 15, 1, 22, 9].

Compare the second element, 15, to the third element, 1. Since 1 is smaller than 15, we swap them, resulting in the list [6, 1, 15, 22, 48].

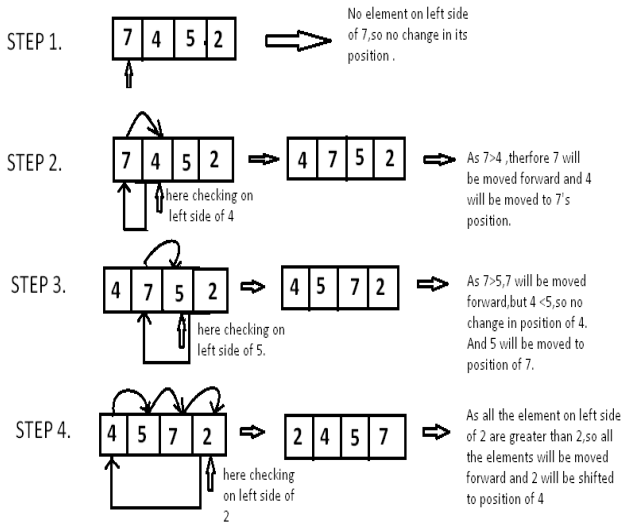
Compare the first element, 6, to the second element, 1. Since 1 is smaller than 6, we swap them, resulting in the final sorted list of [1, 6, 15, 22, 48].

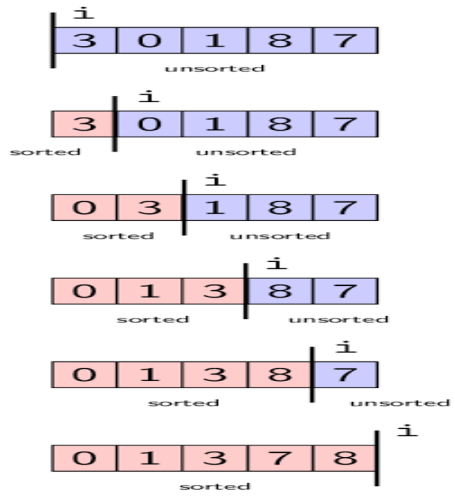
Here is an illustration for you to have a better understanding of the sorting method.

Illustration for Insertion sort technique




**Example 2 :**






As we have seen, the amount of comparisons insertion sort needs to perform between the item to insert and the items in the sorted sublist isn't always the same; it depends on the input data. This is why it is useful to look at the best- and worst-case scenarios.

## Best-case scenario

- The best-case scenario is when the algorithm performs the smallest possible number of comparisons. For insertion sort, this happens when the items are already in order. Then the algorithm performs only one comparison during each pass since the item to insert is already in the correct position and does not need to be moved. The algorithm will perform $n-1$ passes (where $n$ is the number of items) because the first item is already in the sorted sublist. Therefore, one comparison per pass for $n-1$ passes will result in $n-1$ comparisons.

For example, suppose that you have five playing cards as shown in **Figure 11**. You want to sort the cards into ascending order. The first card is in the sorted group and the remaining cards are in the unsorted group. Each time a card from the unsorted group is compared to the previous card (the last card in the sorted group), it is already in the correct position and is now part of the sorted group. This is the best-case scenario, where each of the four unsorted cards only need to be compared once to the previous card and none of the cards are moved.

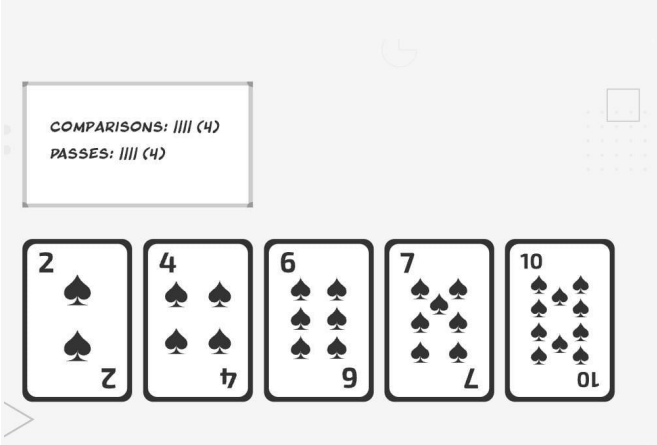



**Figure 11:** The cards are already sorted into ascending order

## Worst-case scenario

- The worst-case scenario takes place when the list of items you are sorting results in the greatest possible number of comparisons. For insertion sort, this happens when the items are the most unordered they can be; for example, the original list of items is in descending order and the algorithm is sorting the items into ascending order. Then the algorithm has to move every item in the sorted sublist along one place every single pass to make room for the item to insert at the start of the list.

For example, suppose that you have five playing cards arranged in descending order, as shown in **Figure 12**. The insertion sort algorithm you are using orders the cards into ascending order.

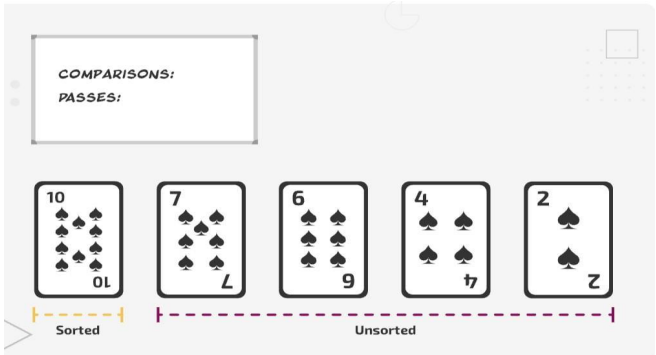



**Figure 12:** The starting order of the worst-case scenario

In the first pass, the card in the sorted group is moved up one place and the card to insert is placed at the start of the group (see **Figure 13**).

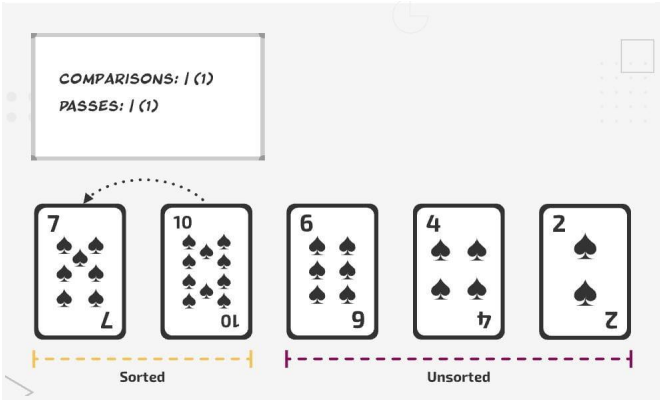



**Figure 13:** The order of the cards after the first pass

In the second pass, the two cards in the sorted group are each moved up one place and the card to insert is placed at the start of the group. In the third pass, there are three cards in the sorted sublist to move, and in the fourth pass there are four cards to move.

This is the worst-case scenario, where you will need to move every card in the sorted group up one place every single pass (see **Figure 14**). As the sorted group grows in size, the number of comparisons also increases at the same rate.

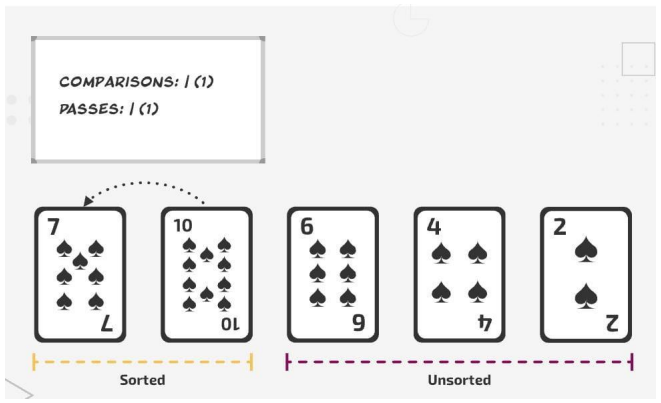



**Figure 14:** The order of the cards after the final pass

Advanced

## Insertion sort – complexity

### Time complexity

If you are asked only to state the "time complexity" of the insertion sort algorithm, you should give the worst case, which is $O(n^2)$. This is called polynomial time efficiency.

*How the time complexity is calculated*

- The outer for loop is repeated $n-1$ times, because it starts with the second item.
- The inner while loop is repeated a variable number of times because it checks and moves all the previous items that need to be shifted to free up space for the value to insert.

*Best, average, and worst-case time complexity*

- In the **best case**, when the items are already in order, no items will need to be moved. The outer for loop will iterate $n-1$ times. The condition of the inner while loop will evaluate to False and so the loop will not be executed. Therefore, the best-case time complexity is $O(n)$.
- In the **worst case**, the values of the original list are in reverse order (in this case ordered high to low). Therefore, to order a list of $n$ items, you need to move every item_to_insert you examine to the very front of the list:

You will make one comparison to move the second item to the front of the list
You will make two comparisons to move the third item to the front of the list
...
You will make $n-1$ comparisons to move the $n$th item to the front of the list

The total number of comparisons you will make is $1+2+3+...+n-1$. This evaluates to $2n(n-1)=2n^2-n$. However, in Big O notation, you express this time complexity as $O(n^2)$ because the less dominant terms (e.g. $n$) and the constants (e.g. 21) become insignificant as the size of the input grows.

- The average case, requires a good understanding of statistics, but it will clearly be more efficient than the worst case. However, in Big O notation, this will also be expressed as $O(n^2)$ because the less dominant terms (e.g. $n$) and the constants (e.g. 21) become insignificant as the size of the input grows.

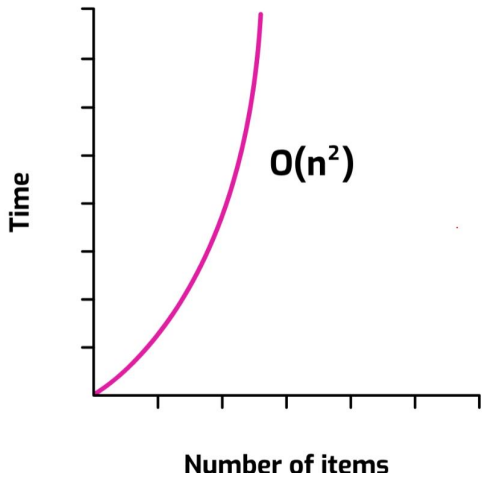
Insertion sort time complexity

### Space complexity

The insertion sort is very efficient in terms of memory requirement, as the sorting is done in the **same space** as the original data (i.e. the items array in the pseudocode). The only extra space that is needed is to hold a copy of the item

to be being inserted, to free up space for any other items to be moved so that the value can be inserted into the correct place.

Therefore, the **space complexity** of the algorithm is $O(1)$.

- 


## Comparison of insertion sort with other sorting algorithms

Insertion sort is similar to selection sort; the primary difference is that the *ith* iteration in insertion sort gives the sorted subarray from the *i* elements input to it, whereas, in selection sort, the *ith* iteration gives the *i* smallest elements in the entire array.

Although bubble sort has the same time complexity, i.e., O(n2), it is far less efficient than both insertion and selection sort. While some divide-and-conquer algorithms, such as Quick sort and merge sort outperform insertion sort for larger arrays, non-recursive sorting methods such as insertion sort or selection sort are often quicker for small arrays.

## Advantages of Insertion Sort:

- Simple and easy to understand and implement.
- Efficient for small data sets or nearly sorted data.
- In-place sorting algorithm, meaning it doesn't require extra memory.
- Stable sorting algorithm, meaning it maintains the relative order of equal elements in the input array.

## limitations of Insertion Sort:

1. Inefficient for large datasets: Insertion sort has a time complexity of O(n^2), making it less efficient for large datasets.

2. Slow performance: Insertion sort is slower compared to other sorting algorithms like Quick Sort, Merge Sort, and Heap Sort.

3. Not suitable for real-time data: Insertion sort is not suitable for real-time data as it can take a long time to sort large amounts of data.

4. Not efficient for reverse-sorted data: Insertion sort performs poorly on reverse-sorted data, with a time complexity of O(n^2).

5. Not a stable sort: Insertion sort is not a stable sort, meaning that equal elements may not keep their original order.

6. Not efficient for data with many duplicates: Insertion sort can be slow when dealing with data that has many duplicate elements.

7. Not suitable for parallel processing: Insertion sort is not suitable for parallel processing as it is a sequential algorithm.

8. Not efficient for data with varying sizes: Insertion sort can be slow when dealing with data that has varying sizes.

Overall, while Insertion Sort is simple to implement and understand, its limitations make it less suitable for large-scale data sorting and real-world application.

## real-world scenarios where Insertion Sort can be helpful:

- Sorting a small list of numbers.
- Organizing cards in a card game.
- Sorting a list of students by their grades.
- Maintaining a sorted list of stock prices or exchange rates for real-time trading applications.

# Result and Discussion:

Insertion sort is a simple sorting algorithm that works by dividing the input into a sorted and an unsorted region. Each subsequent element from the unsorted region is inserted into the sorted region in its correct position.

## Result:

- Insertion sort has a time complexity of O(n^2) in the worst case, making it less efficient for large datasets.
- It has a best-case time complexity of O(n) when the input is already sorted.

- Insertion sort has a space complexity of O(1) since it only requires a single additional memory space for the temporary variable.

### Discussion:

- Insertion sort is simple to implement and understand, making it a good choice for small datasets or educational purposes.
- It is a stable sorting algorithm, meaning that the order of equal elements is preserved.
- Insertion sort is adaptive, meaning that it performs well on partially sorted data.
- However, its poor performance on large datasets makes it less suitable for real-world applications.
- Insertion sort can be improved by using techniques like binary insertion sort or shell sort.

Overall, insertion sort is a basic sorting algorithm that is easy to understand and implement but has limitations in terms of efficiency for large datasets.

## Conclusion:

Insertion Sort in Ada: A Simple yet Effective Sorting Algorithm

Insertion Sort is a straightforward and intuitive sorting algorithm that has been implemented in various programming languages, including Ada. The algorithm's simplicity and ease of understanding make it an excellent choice for small datasets and educational purposes.

### Key Takeaways:

Insertion Sort has a time complexity of O(n^2), making it less efficient for large datasets.

The algorithm is simple to implement and understand, making it a great choice for beginners.

Insertion Sort is a stable sorting algorithm, preserving the order of equal elements.

It is not suitable for real-time data, large datasets, or parallel processing.

### Ada Implementation:

The Ada implementation of Insertion Sort is concise and efficient, utilizing the language's strong typing and control structures. The algorithm's simplicity translates well to Ada's syntax, making it easy to read and maintain.

### Future Improvements:

While Insertion Sort is not the most efficient sorting algorithm, it can be improved by:

Using binary insertion sort to reduce the number of comparisons.

Implementing a hybrid sorting algorithm that combines Insertion Sort with other algorithms for larger datasets.

In conclusion, Insertion Sort in Ada is a simple yet effective sorting algorithm that is well-suited for small datasets and educational purposes. Its limitations make it less suitable for large-scale data sorting, but its simplicity and ease of understanding make it a great choice for beginners.

## References:

[1] Lipschutz, Data Structures With C. McGraw-Hill Education (India) Pvt

Limited, 2011.

[Online]Available: http://books.google.co.in/books?id=YJQIOLgFnnYC

[2] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, Introduction

to algorithms. The MIT press, 2001.

[3] Tiwari, T.; Singh, S.; Srivastava, R.; Kumar, N., "A bi-partitioned

insertion algorithm for sorting," Computer Science and Information

Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference

[4] Encode.su

[5] Geeks For Geeks website

[6] Wikipedia.org

[7]Meta AI

[8]C.hatGpt