

Title: A Comparative Analysis of Insertion Sort and Quick Sort Algorithms by using C++,Java and Python programming language.

Abstract

In this research paper, we conduct a comparative analysis of two widely used sorting algorithms, insertion sort and quick sort, implemented in C++, Java, and Python. The aim of the study is to evaluate the performance of the algorithm in terms of execution time across different input sizes and programming languages. A consistent implementation of both algorithms serves as the basis for a comprehensive evaluation. The impact of programming language choice and implementation strategy on algorithm efficiency is investigated, and insights into scenarios in which one algorithm may outperform the other are explored. This study provides insight into the relative strengths and weaknesses of insertion sort and quick sort in different programming environments through careful performance analysis. The expected insights are intended to help developers make informed decisions when selecting and implementing sorting algorithms.

Keywords: Sorting Algorithms, Insertion Sort, Quick Sort, Performance Analysis, C++, Java, Python, Programming Languages

Introduction :

Sorting algorithms play a role, in computer science as they help efficiently organize and process data.[1] This paper focuses on comparing two sorting algorithms, namely insertion sort and quick sort implemented in three popular programming languages; C++, Java and Python. Our main objective is to evaluate their performance by measuring execution time and identify factors that affect their efficiency across programming environments.

While Insertion was developed using an iterative programming approaches, Quick Sort was implemented utilising recursive programming approaches. [1][2]The programming language options were divided into groups according to:

***Virtual machine based languages: Java (compiled and interpreted language)**

Native Languages: C++(Compiled language)

Scripting Languages: Python (Interpreted language)

[5]

The key goals of this study are;

- 1. Implementing both algorithms in C++ Java and Python to ensure a basis for comparison.**
- 2. Conducting a performance analysis by measuring execution times for sizes of input arrays.**
- 3. Investigating how the choice of programming language and implementation strategies impact algorithm performance.**
- 4. Identifying scenarios where one algorithm may outperform the other based on data characteristics and programming environments.**
- 5. Providing insights and recommendations for developers to make decisions when selecting and implementing sorting algorithms in their applications.**

We anticipate that this research will;

- 1. Illuminate the strengths and weaknesses of insertion sort and Quick sort across programming languages.**
- 2. Offer guidance to developers on optimizing sorting algorithms based on use cases and performance requirements.**
- 3. Contribute to an understanding of algorithm design and implementation choices, for data management.**

[2][5][6]

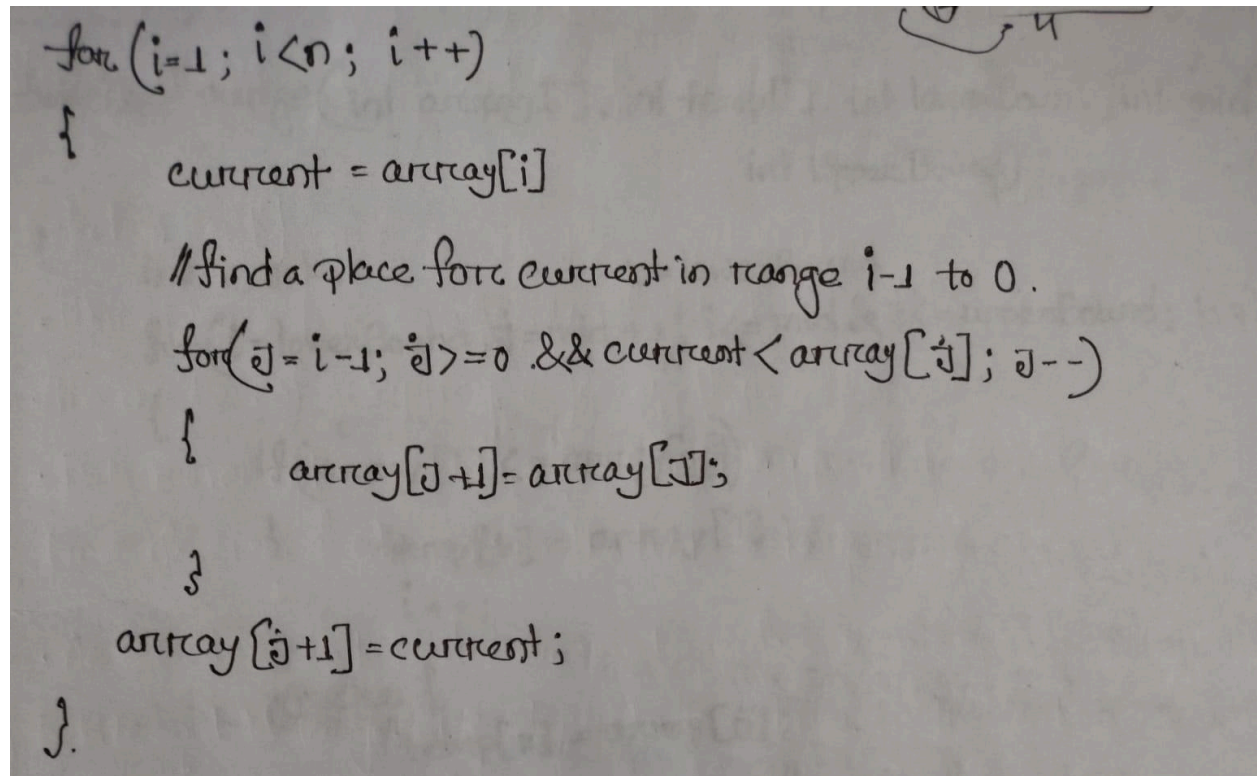
Background Study:

A straightforward and natural sorting method called "insertion sort" builds the sorted array one element at a time. For small data sets, especially those that have been significantly sorted, it is simple to apply and highly effective.

Insertion sort operates by assuming that the initial item in the array is already sorted. It then compares the second item to the first. If the first item is larger, the second is placed ahead of it. These steps ensure that the first two items are sorted. The third item is then

compared to the ones on its left, and it is positioned after the item that is smaller. If there is no smaller item, the third item should be inserted at the beginning of the array. This procedure continues until the entire array is sorted.

Pseudo-code :



```
for (i=1; i<n; i++)  
{  
    current = array[i]  
  
    // find a place for current in range i-1 to 0.  
    for (j = i-1; j >= 0 && current < array[j]; j--)  
    {  
        array[j+1] = array[j];  
    }  
  
    array[j+1] = current;  
}
```

The time complexity of insertion sort varies under different scenarios. In the best case, when the array is already sorted, the algorithm only needs to compare each element with its predecessor, requiring n steps to sort the n -element array. The inner loop does not run at all when the array has already been sorted, but the outside loop continues for n times. Thus, the number of comparisons is limited to n . Complexity is hence linear.

On the other hand, in the worst case, when the array is reverse-sorted, insertion sort has to insert each element at the beginning of the sorted subarray, resulting in a time complexity of $O(n^2)$. In the average case, where the array elements are in random order, the running time is approximately $O(n^2 / 4) = O(n^2)$. Since every element must be compared to every other element, $(n-1)$ comparisons are conducted for every n th element. Consequently, $n*(n-1)$ is the total number of comparisons.

Regarding space complexity, insertion sort uses a constant amount of additional variables besides the input array, resulting in a space complexity of $O(1)$.

Among the quickest and best sorting algorithms are Bubble Sort, Selection Sort, and Insertion Sort. The Quick Sort Algorithm was created by renowned computer scientist C.A.R. Hoare. The Quick Sort algorithm is another excellent illustration of the divide and conquer tactic in action.[3]

The fundamental ideas behind the Quick Sort algorithm are as follows:

- >select the pivot element from the array that has to be sorted.

- >To see that every element in the array with a value smaller than the pivot appears before it, and every element in the array with a value larger than the pivot appears after it, do a partition operation. Following this procedure, the pivot is in the array's final location.

- >The sublist of larger entries and the sublist of smaller elements are sorted recursively.

[3]

The Partition Function of the Quick Sort is as follows:

```
int partition(int array[], int upperBound, int lowerBound)
```

```
{  
    int high = upperBound;
```

```
    int low = lowerBound;
```

```
    pivot = array[lowerBound];
```

```
    if (high > low)
```

```
{
```

```
    while (array[low] <= pivot && high > low)
```

```
    {  
        low ++;
```

```
    }
```

```
    while (array[high] > pivot)
```

```
    {  
        high --;
```

```
    }
```

```
    if (high > low)
```

```
    {
```

```
        swap(array[low], array[high]);
```

```
    }
```

```
}
```

```
array[lowerBound] = array[high];
```

```
array[high] = pivot;
```

```
return high;
```

Here is the working principle of the Partition function:

1. **Choose a pivot:**
 - In General, the first element, a random element, or last element of the sample is selected as the pivot.
2. **Set up pointers:**
 - Two pointers, which are named as low and high, are initialized at the beginning and end of the array that is going to be partitioned.
3. **Iterate and swap:**
 - The algorithm iterates through the array, comparing elements with the pivot and swapping them based on the below :
 - If an element at low is larger than or equal to the pivot, swap it with the element at high, and decrement high.
 - Otherwise, increment low.
4. **Place pivot in position:**
 - When low and high pointers meet, swap the pivot with the element at high. This places the pivot in its final sorted position, with smaller elements to its left and larger elements to its right. And also return the index of High as that gonna need to the main algorithm in which that is used to find the particular index no.

The Main Algorithm of partition function is below:

```

QuickSort(Array[], upperBound, lowerBound)
{
    if (lowerBound >= upperBound)
    {
        return;
    }
    j = partition(Array[], upperBound, lowerBound);
    QuickSort(Array[], j-1, lowerBound);
    QuickSort(Array[], upperBound, j+1);
}

```

The complexity Analysis of the quick sort is given below:

TC \leftarrow QuickSort (Array[], upperBound, LowerBound)

```

{
    if (lowerBound >= upperBound)
    { return;
    }

```

n \leftarrow \bar{v} = partition (Array[], upperBound, LowerBound);

T(n/2) \leftarrow QuickSort (Array[], $\bar{v}-1$, lowerBound);

T(n/2) \leftarrow QuickSort (Array[], upperBound, $\bar{v}+1$); \rightarrow tail recursion

```

}
```

Recurrence relation, $T(n) = 2T(n/2) + n$

The final recurrence Relation becomes,

$$T(n) = \begin{cases} 1, & n=1 \\ 2T(n/2) + n, & n>1 \end{cases}$$

By using the method of induction,

$$T(n) = 2T(n/2) + n$$

$$T(n) = 2 [2T(n/4) + n/2] + n$$

$$= 2^2 T(n/2^2) + 2n$$

$$T(n) = 2^2 [2T(n/2^3) + n/2^2] + n + n$$

$$= 2^3 T(n/2^3) + n + n + n$$

$$= 2^3 T(n/2^3) + 3n.$$

for k iteration,

$$T(n) = 2^k \cdot T(n/2^k) + kn$$

$$\text{Let, } n/2^k = 1$$

$$\Rightarrow n = 2^k$$

$$\Rightarrow k = \log n$$

now substituting the value of k we get

$$\begin{aligned} T(n) &= 2^k T(n/2^k) + kn \\ &= n \cdot T(1) + (\log n \times n) \\ &= n \log n + n. \end{aligned}$$

$$O(n) = (n \log n).$$

The best case & average case time complexity is ~~$O(n \log n)$~~ $O(n \log n)$.

The worst case is $O(n^2)$. This happens when the array is sorted in the reverse order which require n steps in each number of iteration. That's become $O(n^2)$ is the worst case time complexity.

for space complexity,

~~in the In Main Algorithm~~

The QuickSort Algorithm contain a tail recursion. And we know, for tail recursion, operating system does not require to store any additional information into system stack. But for non-tail, it does.

So the space complexity is: $O(\log n)$.

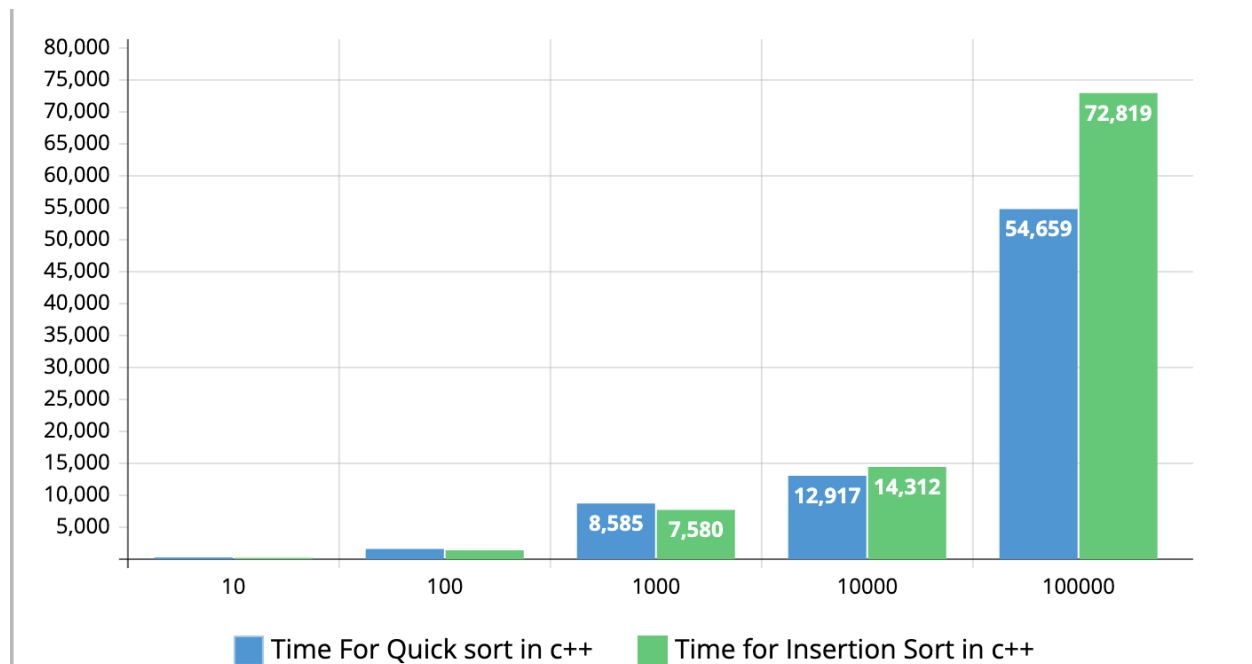
[1][6][9]

Result & Analysis:

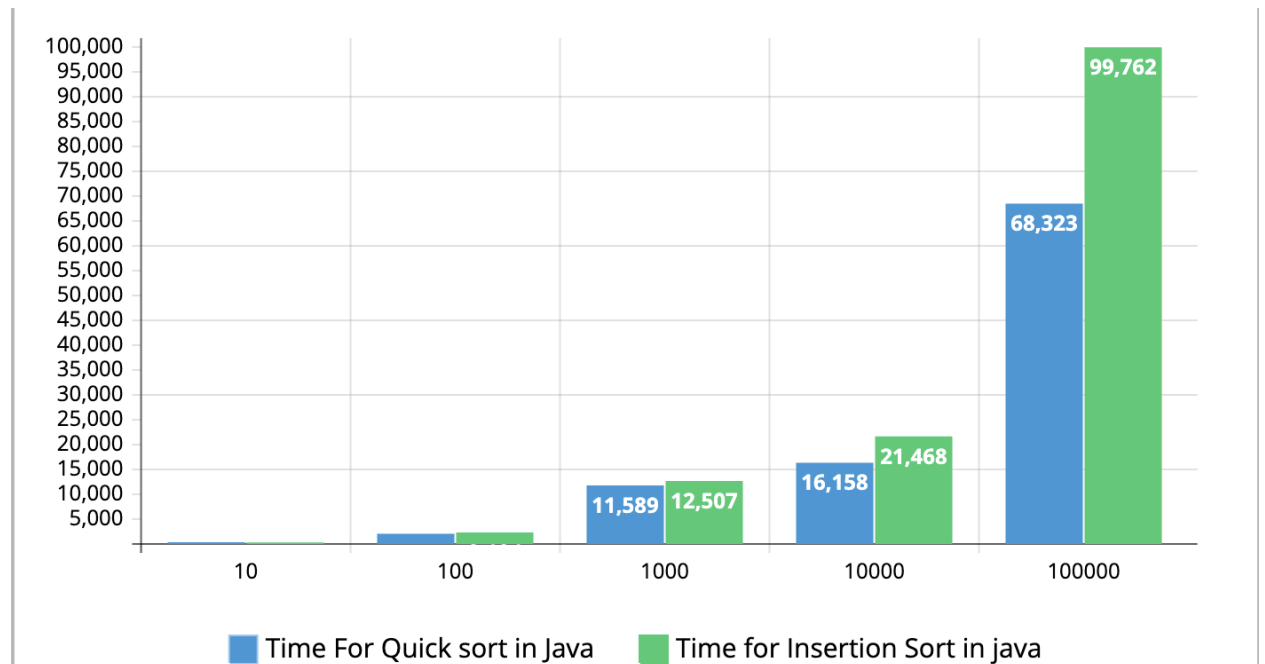
The experiments were carried out on a with the same configuration, running on MacBook Pro. The sorting algorithms were executed on M2 MacBook Pro , which had a 16GB RAM and a M2 chip.

Data Table:

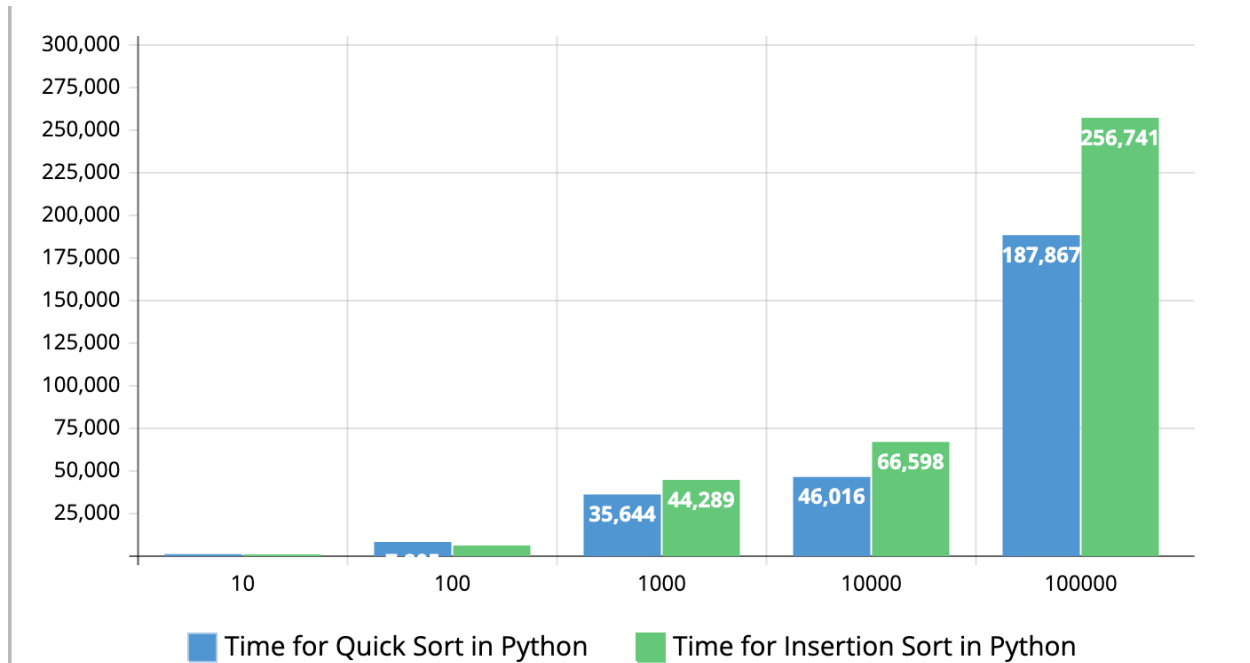
Testing For C++ Language		
Number of Element	Time for Insertion Sort(μ s)	Time for Quick Sort(μ s)
10	85	138
100	1257	1458
1000	7580	8585
10000	14312	12917
100000	72819	54659



Testing For JAVA Programming Language		
Number of Element	Time for Insertion Sort(μ s)	Time for Quick Sort(μ s)
10	146	197
100	2124	1868
1000	12507	11589
10000	21468	16146
100000	99672	68323



Testing For Python Programming Language		
Number of Element	Time for Insertion Sort(μ s)	Time for Quick Sort(μ s)
10	474	614
100	5697	7805
1000	44289	35644
10000	66598	46016
100000	256741	187876



From the theoretical Background we can see that the time complexity of Insertion Sort is $O(N^2)$ in the worst case and for best case it becomes $O(N)$. as far as space complexity concern it always constant $O(1)$.

For the Quick Sort, the average time complexity is $O(N \log N)$ which rises upto $O(N^2)$ in the worst case scenario. This Happens when the array is already sorted in reverse. The Space Complexity of the Quick sort Algorithm $O(\log N)$.

If we try to relate the result from the practical data that we get, we can say it that for a smaller size of data (i.e 100) insertion perform better than quick and for huge level of Data set (i.e .100000) quick sort perform better. For a sorted data set, the insertion sort algorithm outperform the quick sort. This remain true for any amount of Dataset.

Now, if we discuss the dependency of language, we can observe that python take more time to implement even though it follows the theoretical aspect of insertion & quick sort. This may because Java & C++ is statically Typed language. On the other hand, Python is a dynamically typed language.

Python is easier to understand and read with. It also takes less lines of code to perform these implementation with respect to both C++ and Java. But for more computing efficiency using of C++ or Java is recommended as there is a tiny difference between them when it comes to execution speed.

Conclusion:

Conducting this research, we understand that insertion sort outperforms quick sort when the data set is tiny but quick sort performs far better for huge data set (i.e. 100000). Insertion sort is a better choice for a sorted dataset as, in this case, it provides linear time complexity but for quick sort it becomes quadratic complexity. And this rule is followed by any type of programming language even though different programming languages take different spans of time to implement it. One of the major drawbacks of this experiment is that this experiment is done on only a MacBook Pro. It would be better if we could be able to perform this experiment on various hardware configurations and different operating systems (i.e. Windows, Linux). This can be improved not only by using comparatively more lower level languages (like assembly) but also using some modern level languages like JavaScript, RUST. This will create more diversified results. We can also improve it by using more data points which will help us to visualize the time-taking trend by these two algorithms more clearly.

Reference:

- [1] Cormen T, Leiserson C, Rivest R and Stein C. 2001. Introduction to Algorithms, Tata Mc Graw Hill.
- [2] You Yang, Ping Yu and Yan Gan, "Experimental study on the five sort algorithms," 2011 Second International Conference on Mechanic Automation and Control Engineering, Inner Mongolia, China, 2011, pp. 1314-1317, doi: 10.1109/MACE.2011.5987184.
- [3] W. Xiang, "Analysis of the Time Complexity of Quick Sort Algorithm," 2011 International Conference on Information Management, Innovation Management and Industrial Engineering, Shenzhen, China, 2011, pp. 408-410, doi: 10.1109/ICIII.2011.104.
- [4] J. JaJa, "A perspective on Quicksort," in Computing in Science & Engineering, vol. 2, no. 1, pp. 43-49, Jan.-Feb. 2000, doi: 10.1109/5992.814657.
- [5] Ayodele, Oluwakemi Sade, and Bamidele Oluwade. "A Comparative Analysis of Quick, Merge and Insertion Sort Algorithms using Three Programming Languages I: Execution Time Analysis." *African Journal of Management Information System* 1, no. 1 (2019): 1-18.
- [6] Ali, Waqas; Islam, Tahir; Rehman, Habib Ur; Ahmed, Izaz; Khan, Muneeb; Mahmood, Amna (2016) "Comparison of Different Sorting Algorithms" In International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE) volume 5, Issue 7, ISSN: 2277-9043. Pdf [Online] ijarcsee.org/index.php/IJARCSEE/article/view/554/526 . Retrieved on 19/09/2018
- [7] Al-Kharabsheh, Khalid Suleiman, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani, and Nabeel Imhammed Zanoon. "Review on sorting algorithms a comparative study." *International Journal of Computer Science and Security (IJCSS)* 7, no. 3 (2013): 120-126.

[8]Tillison, J. and Shene, C.K., 1995. On generating worst-cases for the insertion sort. *ACM SIGCSE Bulletin*, 27(2), pp.57-58.

[9]Atamazhori S, Hassanvand A, Omidvar S. *On Asymptotic Notation : an Introduction to Analyses of Algorithms*. 2021;(August):0–6.