A Comparative Analysis of Insertion Sort and Quick Sort Algorithms in C++, Java, and Python

Paper Work: Taki Tamim -200302
Fazle Rabbi-200304
Formatting: Moloy Kumar Das-2003012
Rakibul Hasan Durjoy-2003018

Slide maker: Md. Rashedul Islam -2003024
Md. Shefat Hossen Shoikat-2003045

January 23, 2024

1 Abstract

The significance of algorithms in software and programming demands an understanding of the fundamental ideas that underlie them. We often meet issues in our daily lives that require us to start the data sorting process to find solutions. A significant amount of programming logic has been developed for both the case and the generic case. The authors of this study used bubble sort and Tim sort as their two techniques of sorting. C++ is used in the construction of the sorting program. To solve a problem more quickly and effectively while using minimal resources, algorithms are needed.

2 Introduction

Sorting algorithms play a crucial 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 Sort was developed using iterative programming approaches, Quick Sort was implemented utilizing recursive programming approaches.[1] [2]The programming language options were divided into groups according to:

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

1. Virtual machine-based languages: Java (compiled and interpreted language)
2. Native Languages: C++ (Compiled language)
3. Scripting Languages: Python (Interpreted language)[3]

The key goals of this study are:

1. **Implementing both Algorithm:** Implement the sorting algorithms in C++, Java, and Python to ensure a basis for comparison.
2. **Conducting performance analysis:** Measure execution times for various sizes of input arrays to conduct a thorough performance analysis.
3. **Investigating language and strategy impact:** Investigate how the choice of programming language and implementation strategies impact the performance of the algorithms.
4. **Identifying scenarios for performance differences:** Identify scenarios where one algorithm may outperform the other based on data characteristics and programming environments.

5. **Providing insights and recommendations:** Provide insights and recommendations for developers to make informed 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][3][4]

3 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.

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

Algorithm 1 Insertion Sort

```
1: procedure InsertionSort(arr, n)
2:     for i ← 1 to n − 1 do
3:         key ← arr[i]
4:         j ← i − 1
5:         while j ≥ 0 and arr[j] > key do
6:             arr[j + 1] ← arr[j]
7:             j ← j − 1
8:         end while
9:         arr[j + 1] ← key
10:    end for
11: end procedure
```

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 nth element. Consequently, $n \times (n-1)$ 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.[5]

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.[5]

Algorithm 2 Partition Function of Quick Sort

```
1: procedure PlacePivot(arr, low, high)
2:     pivot ← arr[high]
3:     i ← low − 1
4:     for j ← low to high − 1 do
5:         if arr[j] ≤ pivot then
6:             i ← i + 1
7:             Swap arr[i] and arr[j]
8:         end if
9:     end for
10:    Swap arr[i + 1] and arr[high]   ▷ Place pivot in its final position
11:    return i + 1 ▷ Return the index of the pivot
12: end procedure
```

Here is the working principle of the Partition function:

1. **Choose a pivot:** In general, the first element, a random element, or the last element of the sample is selected as the pivot.

2. **Set up pointers:** Two pointers, 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 following:

   - 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. Also, return the index of high as that is needed in the main algorithm, which is used to find the particular index number.

The Main Algorithm of Quick Sort is below:

Algorithm 3 Quick Sort Function

```
1: procedure QuickSort(arr, low, high)
2:     if low < high then
3:         pivotIndex ← Partition(arr, low, high)
4:         QuickSort(arr, low, pivotIndex − 1)   ▷ Sort the left subarray
5:         QuickSort(arr, pivotIndex + 1, high)   ▷ Sort the right subarray
6:     end if
7: end procedure
```

Complexity Analysis of Quick Sort by Iteration

Iteration	Size of Subarray	Work Done
0	n	$O(n)$
1	$\frac{n}{2}$ (on average)	$O(n)$
2	$\frac{n}{4}$ (on average)	$O(n)$
3	$\frac{n}{8}$ (on average)	$O(n)$
$\vdots$	$\vdots$	$\vdots$
k	$\frac{n}{2^k}$ (on average)	$O(n)$

The total work done is the sum of the work done in each iteration:

$$T(n) = O(n) + O(n) + \ldots + O(n) = k \cdot O(n)$$

To find the number of iterations (k), we solve the following equation:

$$\frac{n}{2^k} = 1 \implies k = \log_2 n$$

Therefore, the overall time complexity is:

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

The time complexity of Quick Sort is analyzed in terms of the number of comparisons and swaps.

Best Case:

In the best case, the pivot chosen is always the median element. In each partitioning step, the array is divided into two equal halves. Therefore, the recurrence relation for the best-case time complexity is given by:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

By solving the recurrence relation, we find that the best-case time complexity of Quick Sort is $O(n \log n)$.

Average Case:

On average, if the pivot is chosen randomly, the expected time complexity is also $O(n \log n)$. The recurrence relation remains the same as in the best case.

Worst Case:

In the worst case, the pivot is always the smallest or largest element. This leads to an unbalanced partition, and the recurrence relation becomes:

$$T(n) = T(n-1) + O(n)$$

Solving this recurrence relation yields the worst-case time complexity of $O(n^2)$.

Space Complexity:

The space complexity of Quick Sort is dominated by the recursive call stack. In the worst case, the maximum depth of the call stack is $\log_2 n$, resulting in a space complexity of $O(\log n)$. [1][6][7]

4 Result & Analysis

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

Data Table:

Element	Time for Insertion sort(µs)	Time for Quick sort(µs)
10	85	138
100	1257	1458
1000	7580	8585
10000	14312	12917
100000	72819	54659

Table 1: Testing For C++ Language

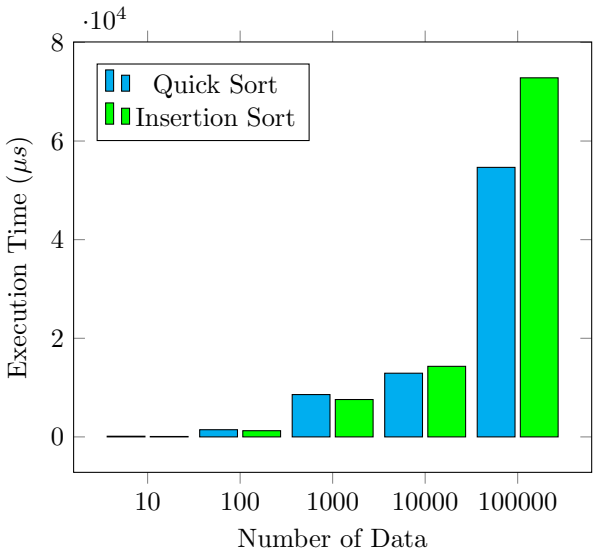

Figure 1: Comparison between Insertion and Quick Sort in C++ Language

Element	Time for Insertion sort(µs)	Time for Quick sort(µs)
10	474	614
100	5697	7805
1000	44289	35644
10000	66598	46016
100000	256741	187876

Table 2: Testing For Python Language

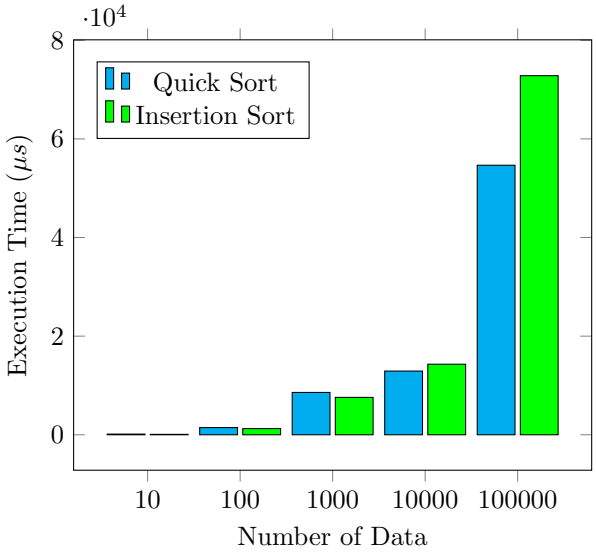

Figure 2: Comparison between Insertion and Quick Sort in Python Language

Theoretical Background and Practical Results

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

For Quick Sort, the average time complexity is $O(N \log N)$, which rises up to $O(N^2)$ in the worst-case scenario. This happens when the array is already sorted in reverse. The space complexity of the Quick Sort algorithm is $O(\log N)$.

Relating the results from practical data, we observe that for a smaller size of data (i.e., 100), Insertion Sort performs better than Quick Sort, while for a larger dataset (i.e., 100,000), Quick Sort performs better. For a sorted dataset, the Insertion Sort algorithm outperforms the Quick Sort. This remains true for any amount of dataset.

Discussing the dependency on language, we observe that Python takes more time to implement even though it follows the theoretical aspects of Insertion and Quick Sort. This may be because Java and C++ are statically typed languages. On the other hand, Python is a dynamically typed language.

Python is easier to understand and read. It also requires fewer lines of code to implement these algorithms compared to both C++ and Java. However, for more computing efficiency, using C++ or Java is recommended, as there is a tiny difference between them when it comes to execution speed.

5 Conclusion

Conducting this research, we understand that insertion sort outperforms quicksort when the dataset is tiny, but quicksort performs far better for a huge dataset (i.e., 100000). Insertion sort is a better choice for a sorted dataset since, in this case, it provides linear time complexity. However, for quicksort, it becomes quadratic complexity. 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 it was conducted only on a MacBook Pro. It would be better if we could perform this experiment on various hardware configurations and different operating systems (i.e., Windows, Linux). This can be improved not only by using a comparatively more lower-level language (like assembly) but also by 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 visualize the time-taking trend by these two algorithms more clearly.

6 References:

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *et al.*, "Introduction to algorithms, chapter 11," 2001.

[2] Y. Yang, P. Yu, and Y. Gan, "Experimental study on the five sort algorithms," in *2011 Second International Conference on Mechanic Automation and Control Engineering*, pp. 1314–1317, IEEE, 2011.

[3] O. S. Ayodele and B. Oluwade, "A comparative analysis of quick, merge and insertion sort algorithms using three programming languages ii: Energy consumption analysis," *Afr J MIS*, vol. 1, no. 2, pp. 44–63, 2019.

[4] W. Ali, T. Islam, H. Rehman, I. Ahmad, M. Khan, and A. Mahmood, "Comparison of different sorting algorithms," *Int. J. Adv. Res. Comput. Sci. Electron. Eng*, vol. 5, no. 7, pp. 63–71, 2016.

[5] W. Xiang, "Analysis of the time complexity of quick sort algorithm," in *2011 international conference on information management, innovation management and industrial engineering*, vol. 1, pp. 408–410, Ieee, 2011.

[6] K. S. Al-Kharabsheh, I. M. AlTurani, A. M. I. AlTurani, and N. I. Zanoon, "Review on sorting algorithms a comparative study," *International Journal of Computer Science and Security (IJCSS)*, vol. 7, no. 3, pp. 120–126, 2013.

[7] F. A. Ardhiyani, E. Sudarmilah, and D. A. P. Putri, "An evaluation of quick sort and insertion sort algorithms for categorizing covid-19 cases in jakarta," in *AIP Conference Proceedings*, vol. 2727, AIP Publishing, 2023.