



Comperative Analysis of Insertion Sort and Quick Sort

Prepared by-

Taki Tamin (2003002)

Fazle Rabby (2003004)

Moloy Kumar Das (2003012)

Rakibul Hasan Durjoy (2003018)

Md. Rashedul Islam (2003023)

Md. Shefat Hossen Shoikat (2003045)

January 23, 2024



Table of Contents

1 Introduction

► Introduction

► Background Study

Insertion Sort Algorithm

Quick Sort Algorithm

► Results and Analysis

► Conclusion



Introduction

1 Introduction

Sorting algorithms play a crucial role in computer science as they efficiently organize and process data.[1] This paper focuses on the comparative analysis of two sorting algorithms: insertion sort and quick sort, implemented in three popular programming languages - C++, Java, and Python.



Introduction (Contd.)

1 Introduction

- Our main objective is to evaluate the performance of these algorithms by measuring their execution time and identifying factors that influence their efficiency across different programming environments.
- Insertion sort was developed using iterative programming approaches, while quick sort was implemented utilizing recursive programming approaches.
- [1][2] The programming languages were categorized into groups based on their characteristics:
 - Virtual machine-based languages: Java (compiled and interpreted language)
 - Native Languages: C++ (Compiled language)
 - Scripting Languages: Python (Interpreted language)[3]



Introduction (Contd.)

1 Introduction

Key Goals:

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



Introduction (Contd.)

1 Introduction

Anticipated Contributions:

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

[2][3][4]



Table of Contents

2 Background Study

- ▶ Introduction
- ▶ Background Study
 - Insertion Sort Algorithm
 - Quick Sort Algorithm
- ▶ Results and Analysis
- ▶ Conclusion



Insertion Sort Introduction

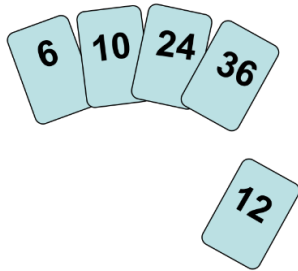
2 Background Study

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.



Insertion Sort - Step 1

2 Background Study



To insert 12, we need to make room for it by moving first 36 and then 24.

Figure: Caption for Step 1



Insertion Sort - Step 2

2 Background Study

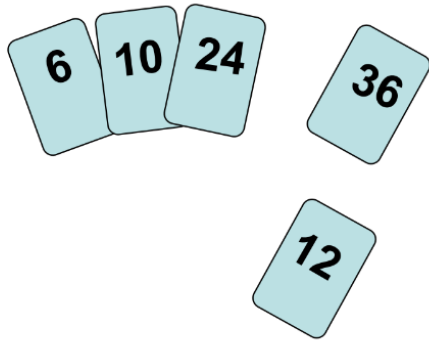


Figure: Caption for Step 2



Insertion Sort - Step 3

2 Background Study

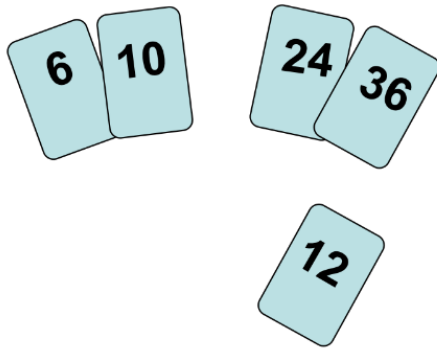


Figure: Caption for Step 3



Insertion Sort Example

2 Background Study

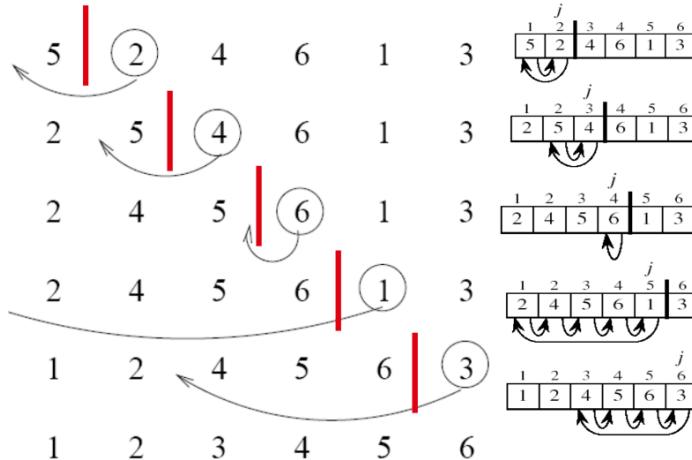


Figure: Caption for Step 4



Insertion Sort Example

2 Background Study

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	5	2	4	6	1	3

key
5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	5	2	4	6	1	3



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	5	2	4	6	1	3



Insertion Sort Example

2 Background Study

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	5	2	4	6	1	3

key
2

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	5	5	4	6	1	3



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	5	4	6	1	3



Insertion Sort Example

2 Background Study

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	5	4	6	1	3

key
4

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	5	5	6	1	3



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	4	5	6	1	3



Insertion Sort Example

2 Background Study

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	4	5	6	1	3

key
6

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	4	5	6	1	3



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	4	5	6	1	3



Insertion Sort Example

2 Background Study

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	4	5	6	1	3

key
1

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	2	2	4	5	6	3



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	1	2	4	5	6	3



Insertion Sort Example

2 Background Study

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	1	2	4	5	6	3

key
3

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	1	2	4	4	5	6



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
$-\infty$	1	2	3	4	5	6



Insertion Sort Algorithm

2 Background Study

Algorithm 1 Insertion Sort

```
1: procedure InsertionSort(arr[], n)
2:   for i  $\leftarrow$  1 to n do
3:     key  $\leftarrow$  arr[i]
4:     j  $\leftarrow$  i - 1
5:     while j  $\geq$  0 and key < arr[j] do
6:       arr[j + 1]  $\leftarrow$  arr[j]
7:       j  $\leftarrow$  j - 1
8:     end while
9:     arr[j + 1]  $\leftarrow$  key
10:  end for
11: end procedure
```



Insertion Sort in C++

2 Background Study

```
1 void insertionSort(int arr[],int n) {  
2     for (int i = 1; i < n; ++i) {  
3         int key = arr[i];  
4         int j = i - 1;  
5  
6         while (j >= 0 && array[j] > key) {  
7             arr[j + 1] = arr[j];  
8             --j;  
9         }  
10  
11         arr[j + 1] = key;  
12     }  
13 }
```



Time and Space Complexity of Insertion Sort

2 Background Study

The time complexity of Insertion Sort varies under different scenarios.

- **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 number of comparisons is limited to n , resulting in linear time complexity.
- **Worst Case:** 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 $O(n^2)$.
- **Average Case:** 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, $n \cdot (n - 1)$ 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 $O(1)$.



Quick Sort Introduction

2 Background Study

Quick Sort is a highly efficient sorting algorithm that uses a divide-and-conquer strategy to sort an array.[5] It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

The key steps in the Quick Sort algorithm include selecting a pivot, partitioning the array, and recursively applying the Quick Sort to the sub-arrays.



Partition Function - Working Principle

2 Background Study

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

Set up pointers: Two pointers, named low and high, are initialized at the beginning and end of the array that is going to be partitioned.

Iterate and swap: The algorithm iterates through the array, comparing elements with the pivot and swapping them based on the following rules:

- 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.

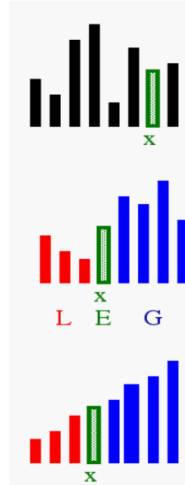
Place pivot in position: When the low and high pointers meet, swap the pivot with the element at high. This places the pivot in its final sorted position, with smaller elements to its left and larger elements to its right. Also, return the index of high as it is needed in the main algorithm to find the particular index.



Idea of Quick Sort Algorithm

2 Background Study

- 1) **Select:** pick an element
- 2) **Divide:** rearrange elements so that x goes to its final position E
- 3) **Recurse and Conquer:** recursively sort





QuickSort Example: Given Array

2 Background Study

- Given array: $a[7] = \{5, 7, 6, 1, 3, 2, 4\}$

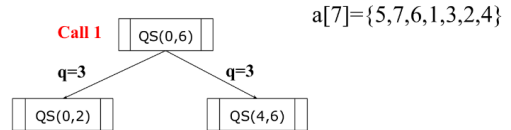


QuickSort: Choosing a Pivot and Partitioning (Steps 1-2)

2 Background Study

Function Calling

- Choose a pivot: $\text{pivot} = 4$ (last element)
- Original array:
 $\{5, 7, 6, 1, 3, 2, 4\}$
- After partitioning:
 $\{1, 3, 2, 4, 7, 6, 5\}$
- Left subarray: $\{1, 3, 2\}$
- Right subarray: $\{7, 6, 5\}$



Range
Left: $p - (q-1)$
Right: $(q+1) - r$

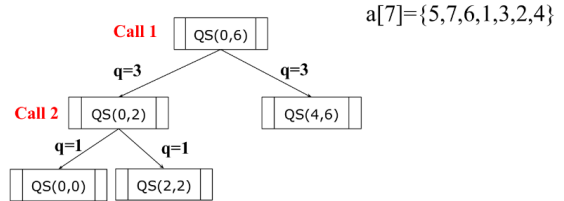


QuickSort: Recursion on Left Subarray (Steps 3-4)

2 Background Study

Function Calling

- Left subarray: $\{1, 3, 2\}$
- Choose pivot: $\text{pivot} = 2$ (last element)
- After partitioning: $\{1, 2, 3\}$



Range
Left: $p - (q-1)$
Right: $(q+1) - r$

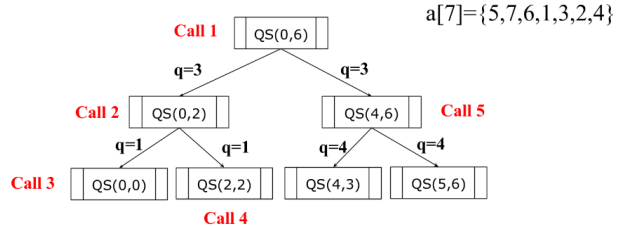


QuickSort: Recursion on Right Subarray (Steps 5-6)

2 Background Study

Function Calling

- Right subarray: $\{7, 6, 5\}$
- Choose pivot: $\text{pivot} = 5$ (last element)
- After partitioning: $\{5, 6, 7\}$



Range

Left: $p - (q-1)$

Right: $(q+1) - r$

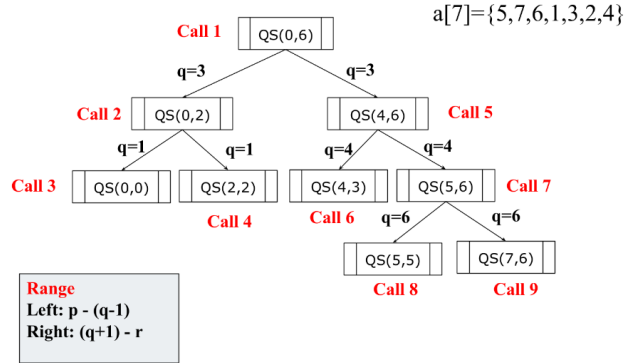


QuickSort: Combine Sorted Subarrays (Step 7)

2 Background Study

Function Calling

- Combine sorted subarrays:
 $\{1, 2, 3, 4, 5, 6, 7\}$





Quick Sort Algorithm

2 Background Study

Algorithm 2 Quick Sort

```
1: procedure QuickSort(arr[], low, high)
2:   if low < high then
3:     pivot  $\leftarrow$  Partition(arr, low, high)
4:     QuickSort(arr, low, pivot - 1)
5:     QuickSort(arr, pivot + 1, high)
6:   end if
7: end procedure
```



Partition Function

2 Background Study

Algorithm 3 Partition

```
1: function Partition(arr[], low, high)
2:   pivot  $\leftarrow$  arr[high]
3:   i  $\leftarrow$  low - 1
4:   for j  $\leftarrow$  low to high - 1 do
5:     if arr[j]  $\leq$  pivot then
6:       Swap arr[i+1] and arr[j]
7:       i  $\leftarrow$  i + 1
8:     end if
9:   end for
10:  Swap arr[i+1] and arr[high]
11:  return i+1
12: end function
```



Quick Sort Function in C++

2 Background Study

```
1 void quickSort(int arr[], int low, int high) {  
2     if (low < high) {  
3         int pivot = partition(arr, low, high);  
4  
5         quickSort(arr, low, pivot - 1);  
6         quickSort(arr, pivot + 1, high);  
7     }  
8 }
```




Partition Function in C++

2 Background Study

```
1 int partition(int arr[], int low, int high) {  
2     int pivot = arr[high];  
3     int i = (low - 1);  
4  
5     for (int j = low; j <= high - 1; ++j) {  
6         if (arr[j] < pivot) {  
7             swap(arr[i+1], arr[j]);  
8             i++;  
9         }  
10    }  
11  
12    swap(arr[i + 1], arr[high]);  
13  
14    return (i + 1);  
15 }
```



Time and Space Complexity of Quick Sort

2 Background Study

Quick Sort has good average-case performance and is often faster in practice compared to other sorting algorithms.

- **Best Case:** The best-case time complexity is $O(n \log n)$ when the partitioning is balanced.
- **Worst Case:** The worst-case time complexity is $O(n^2)$, but this occurs infrequently in practice. Choosing a good pivot strategy helps avoid the worst-case scenario.
- **Average Case:** On average, Quick Sort has a time complexity of $O(n \log n)$. [1][6][7]

Quick Sort typically uses $O(\log n)$ additional space for the recursive call stack, resulting in a space complexity of $O(\log n)$. However, in the worst case, it can use $O(n)$ additional space if the partitioning is highly unbalanced.



Best Case Time Complexity - Mathematical Detail (Part 1)

2 Background Study

Best Case Recurrence Relation:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \cdot \text{constant}$$

Solving the Recurrence:

$$\begin{aligned} T(N) &= 2 \cdot \left(2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2} \cdot \text{constant} \right) + N \cdot \text{constant} \\ &= 4 \cdot T\left(\frac{N}{4}\right) + 2 \cdot \text{constant} \cdot N \end{aligned}$$



Best Case Time Complexity - Mathematical Detail (Part 2)

2 Background Study

Generalizing this pattern, we get:

$$T(N) = 2^k \cdot T\left(\frac{N}{2^k}\right) + k \cdot \text{constant} \cdot N$$

Setting $2^k = N$ and solving for k , we get $k = \log_2 N$.

Substituting this back into the equation, we find:

$$T(N) = N \cdot T(1) + N \cdot \log_2 N \cdot \text{constant}$$

Therefore, the best-case time complexity is $O(N \log N)$.



Average Case Time Complexity - Mathematical Detail (Part 1)

2 Background Study

Average Case Recurrence Relation:

$$T(N) = \frac{2}{N} \sum_{i=1}^{N-1} T(i)$$

Solving the Recurrence:

$$N \cdot T(N) = 2 \sum_{i=1}^{N-1} T(i)$$

$$(N - 1) \cdot T(N - 1) = 2 \sum_{i=1}^{N-2} T(i)$$



Average Case Time Complexity - Mathematical Detail (Part 2)

2 Background Study

Subtracting the second equation from the first:

$$N \cdot T(N) - (N - 1) \cdot T(N - 1) = (N + 1) \cdot T(N - 1) + 2 \cdot N \cdot \text{constant}$$

Simplifying, we get:

$$T(N) = \frac{\log_2 N \cdot (N + 1)}{2}$$

Therefore, the average-case time complexity is $O(N \log N)$.



Worst Case Time Complexity - Mathematical Detail (Part 1)

2 Background Study

Worst Case Recurrence Relation:

$$T(N) = T(N - 1) + N \cdot \text{constant}$$

Solving the Recurrence:

$$\begin{aligned} T(N) &= T(N - 1) + N \cdot \text{constant} \\ &= T(N - 2) + (N - 1) \cdot \text{constant} + N \cdot \text{constant} \\ &= T(N - 3) + 2 \cdot N \cdot \text{constant} - \text{constant} \\ &\vdots \\ &= T(N - k) + k \cdot N \cdot \text{constant} - \frac{k \cdot (k - 1)}{2} \cdot \text{constant} \end{aligned}$$



Worst Case Time Complexity - Mathematical Detail (Part 2)

2 Background Study

If we put $k = N$, then:

$$\begin{aligned} T(N) &= T(0) + N^2 \cdot \text{constant} - \frac{N \cdot (N - 1)}{2} \cdot \text{constant} \\ &= \frac{N^2}{2} + \frac{N}{2} \end{aligned}$$

Therefore, the worst-case time complexity is $O(N^2)$.



Table of Contents

3 Results and Analysis

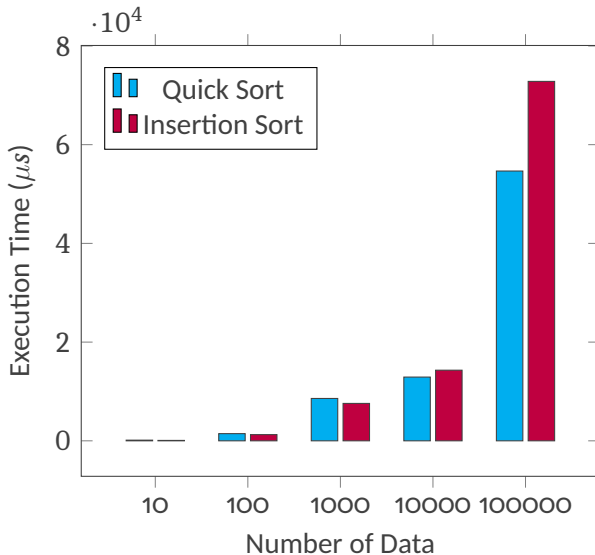
- ▶ Introduction
- ▶ Background Study
 - Insertion Sort Algorithm
 - Quick Sort Algorithm
- ▶ Results and Analysis
- ▶ Conclusion



Comparison of Insertion Sort and Quick Sort

Execution Time for Different Data Sizes

Execution time for
Insertion Sort and Quick
Sort in C++ Language

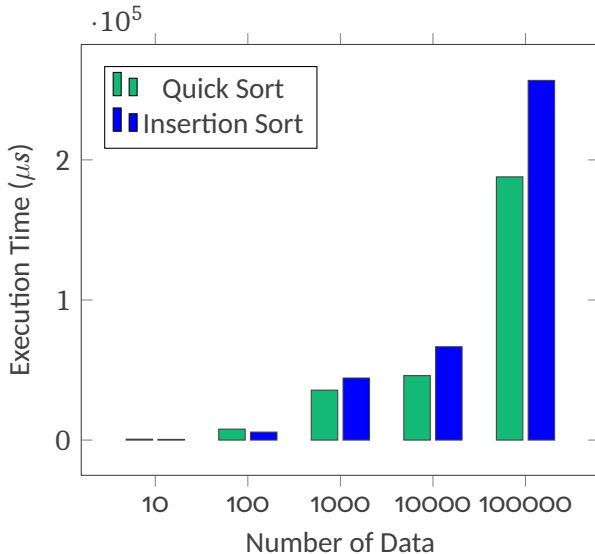




Comparison of Insertion Sort and Quick Sort

Execution Time for Different Data Sizes

Execution time for
Insertion Sort and Quick
Sort in Python Language





Theoretical Background

Time and Space Complexity

Insertion Sort

- Time Complexity:
 - Worst Case: $O(N^2)$
 - Best Case: $O(N)$
- Space Complexity: $O(1)$

Quick Sort

- Time Complexity:
 - Average: $O(N \log N)$
 - Worst Case: $O(N^2)$ (e.g., when the array is sorted in reverse)
- Space Complexity: $O(\log N)$



Practical Observations

Performance on Different Data Sizes

For smaller data (e.g., 100), Insertion Sort outperforms Quick Sort. For larger datasets (e.g., 100,000), Quick Sort performs better.

Sorted Data Set:

- Insertion Sort outperforms Quick Sort.
- True for any dataset size.



Language Dependency

Python, Java, and C++

Python takes more time to implement despite following theoretical aspects. Python's dynamic typing may contribute to this. Java and C++, statically typed languages, show slightly better execution speed.

Recommendation:

- Python is easier to read but slower.
- For efficiency, use C++ or Java.



Table of Contents

4 Conclusion

- ▶ Introduction
- ▶ Background Study
 - Insertion Sort Algorithm
 - Quick Sort Algorithm
- ▶ Results and Analysis
- ▶ Conclusion



Conclusion and Future Work

4 Conclusion

Conclusion:

- Insertion Sort outperforms for small datasets.
- Quick Sort excels for large datasets.
- Language choice affects implementation time.



Good Luck!

4 Conclusion





Any Questions?






Comperative Analysis of Insertion Sort and Quick Sort

Thank you for listening!



-  T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *et al.*, “Introduction to algorithms, chapter 11,” 2001.
-  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.
-  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.
-  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.



-  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.
-  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.
-  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.