INTRODUCTION

Sorting algorithms are fundamental tools in computer science, facilitating the arrangement of data in a specified order for efficient processing and retrieval. Among these algorithms, Insertion Sort stands out for its simplicity and ease of implementation.

Insertion Sort is a sorting algorithm that builds the final sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, Insertion Sort performs well on small datasets or partially sorted datasets. The key idea is to iterate through the list, take one element at a time, and insert it into its correct position among the already sorted elements.

Significance of the Study

This report aims to provide insights into Insertion Sort's behavior with different data structures and data types. Understanding its performance variations is crucial for informed algorithm selection in real-world applications.

Importance in Sorting Algorithms:

Ease of Implementation:

Insertion Sort is straightforward to implement and requires only a small amount of code, making it a good choice for educational purposes or scenarios where simplicity is crucial.

Efficiency for Small Data Sets:

It performs efficiently on small datasets or lists that are already partially sorted. In such cases, its simplicity and low overhead can make it a practical choice.

Online Algorithms:

Insertion Sort is an "online" algorithm, meaning it can sort a list as it receives new elements, without the need to re-sort the entire list. This feature makes it useful in situations where data arrives progressively.

Adaptive Nature:

It is an adaptive algorithm, meaning its performance can be optimized for partially sorted data, adjusting well to the existing order within the dataset.

Experimental Setup:

Programming Language:

- C for implementing the Insertion Sort algorithm and conducting the sorting experiments.

Environment:

- Operating System: *Linux*
- Text Editor: *Kate Editor*
- Compilation: *GCC (GNU Compiler Collection) for compiling and executing C code.*
- Additional Language: *Python for generating random datasets.*

Execution:

- The code was written in the Kate editor on the Linux operating system.
- Compilation and execution were performed in the terminal using the GCC compiler.

Methodology:

"In this case study, we focus on comparing the performance of the Insertion Sort algorithm across three fundamental data structures: Array, Linked List, and Doubly Linked List. Each data structure presents unique characteristics that can impact the efficiency of Insertion Sort, and this comparative analysis aims to provide insights into their respective strengths and weaknesses."

1. **Initialization:**
   - Start with the second element (index 1) of the array. Assume this element is the start of the sorted region.
2. **Comparison and Insertion:**
   - Compare the current element with the elements in the sorted region.
   - Move elements greater than the current element one position to the right to make space for the current element.
   - Insert the current element into its correct position in the sorted region.
3. **Repeat:**
   - Move to the next unsorted element and repeat the comparison and insertion process.
   - Continue this process until all elements are part of the sorted region.
4. **Termination:**
   - The entire array is now sorted.

Algorithm :

```
procedure insertionSort(A: list of sortable items)
  for i from 1 to length(A) - 1
    key = A[i]
    j = i - 1
    while j >= 0 and A[j] >
      key A[j + 1] = A[j]
      j = j - 1
    end while
    A[j + 1] = key
  end for
end procedure
```

Data Structure Used :

1. **Array:**

   Properties:
   - Contiguous block of memory that stores elements of the same data type.
   - Elements can be accessed using their index.
   - Fixed size, determined at the time of creation.
   - Efficient for random access but less flexible in terms of size adjustments.
2. **Linked List:**

   Properties:

- Elements are stored in nodes, and each node contains a data element and a reference (link) to the next node in the sequence.
- Dynamic size; memory is allocated as needed.
- Efficient for insertions and deletions, especially in the middle of the list.
- Inefficient for random access, as elements must be traversed sequentially.

3. **Doubly Linked List:**

   Properties:

   - Similar to a linked list but each node has references to both the next and the previous nodes.
   - Enables bidirectional traversal.
   - Allows more efficient removal of nodes from both the beginning and end of the list.
   - Requires more memory compared to a singly linked list due to the additional references.

Data Types Used :

1. **Integers:**
   - **Description:** Whole numbers without any fractional part.
   - **Use Case:** Commonly encountered in various applications, representing quantities or indices.
   - ***Generation:*** A random set of integers was generated using Python's '***random.randint() '*** function. The range of integers covered both positive and negative values to simulate a diverse dataset.
   - ***Usage:*** The generated integers were saved in a sample file, and the C program for Insertion Sort was designed to read this file as input during the experiments.
2. **Floating-Point Numbers:**
   - **Description:** Numbers that have a decimal point or are expressed in scientific notation.
   - **Use Case:** Used to represent real numbers and are essential for tasks involving precision, such as scientific computations.
   - ***Generation:*** Random floating-point numbers were generated using Python's '***random.uniform() '*** function. The range of floats covered both small and large values for a comprehensive dataset.
   - ***Usage:*** Similar to integers, the generated floating-point numbers were saved in a sample file and used as input for the C program.
3. **Characters:**
   - **Description:** Single letters, digits, or symbols.
   - **Use Case:** Commonly used in applications involving text processing, encoding, and representation of individual symbols.
   - ***Generation:*** A set of random characters, including letters (both uppercase and lowercase), digits, and symbols, were generated using Python's '***random.choice() '*** function on a predefined character set.

- ***Usage:*** The generated characters were saved in a sample file and employed as input for the Insertion Sort algorithm in the C program.

4. **Strings:**
   - **Description:** Sequences of characters.
   - **Use Case:** Used extensively in text processing, representing words, sentences, or any sequence of characters.
   - ***Generation:*** Random strings of varying lengths were created using Python's '***random.choices()*** **'** function on a predefined character set. This allowed for the generation of strings with diverse lengths and character compositions.
   - ***Usage:*** The generated strings were saved in a sample file, and the C program was designed to read this file and perform Insertion Sort on the strings.

File Structure:

- Each type of generated data (integers, floats, characters, strings) was saved in a separate file to maintain clarity and facilitate easy retrieval during experiments.
- The C program for Insertion Sort incorporated file input mechanisms to read the respective files for each data type.

This approach ensures consistency in data generation, allows for replicability of experiments, and enables a systematic comparison of Insertion Sort across different input types. The clear file structure also simplifies the handling of data during analysis.

Result :

The following section presents the results of the Insertion Sort algorithm applied to different data structures (Array, Linked List, and Doubly Linked List) across various input types (integers, floating-point numbers, characters, and strings). The experiments were conducted in a Linux environment using C as the programming language for implementation and Python for generating diverse and random datasets. The time complexity, representing the sorting time in seconds, and space complexity, indicating memory usage in kilobytes, were measured for each combination of data structure and input type. The tables and figures below provide a detailed breakdown of the experimental results, allowing for an insightful analysis of Insertion Sort's performance in different scenarios.

***Integers Range (1 - 10000), (10 – 1000)*
***Floating-Points (1 – 1000), (10 – 100) with upto 6 decimal points*
***Character (10 character each) (12 character each)*

Time Complexity Table:

1. Array

Data Type	Dataset Size	Time (sec)
Integers	100	0.000020
Integers	500	0.000439
Integers	1000	0.001703
Integers	10000	0.127398
Integers	50000	3.019401
Integers	100000	11.961549
Floating-Point	100	0.000036
Floating-Point	500	0.000371
Floating-Point	1000	0.001684
Floating-Point	10000	0.140729
Floating-Point	50000	3.519393
Floating-Point	100000	13.880062
Characters	100	0.000022
Characters	500	0.000355
Characters	1000	0.001558
Characters	10000	0.104181
Characters	50000	2.589581
Characters	100000	10.36064
Strings	100	0.000053
Strings	500	0.000802
Strings	1000	0.003471
Strings	10000	0.275046
Strings	50000	10.384047
Strings	100000	70.324242

2. Linked List

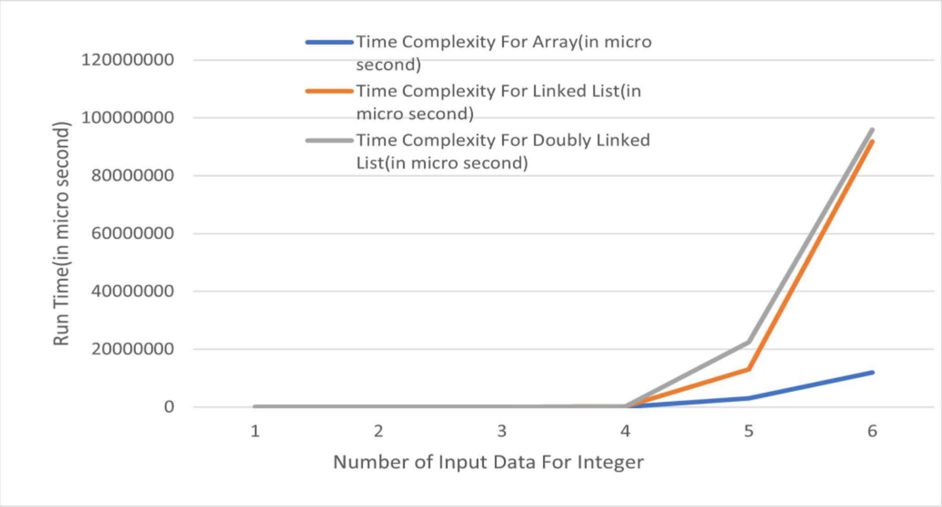
Data Type	Dataset Size	Time (sec)
Integers	100	0.000024
Integers	500	0.000476
Integers	1000	0.001602
Integers	10000	0.236066
Integers	50000	12.996652
Integers	100000	91.846971
Floating-Point	100	0.000025
Floating-Point	500	0.000423
Floating-Point	1000	0.001397
Floating-Point	10000	0.217933
Floating-Point	50000	12.039115
Floating-Point	100000	89.432361
Characters	100	0.000021
Characters	500	0.000402
Characters	1000	0.001602
Characters	10000	0.212706
Characters	50000	10.05935
Characters	100000	70.703129
Strings	100	0.000076
Strings	500	0.000822
Strings	1000	0.003162
Strings	10000	0.330727
Strings	50000	23.566154
Strings	100000	156.839524

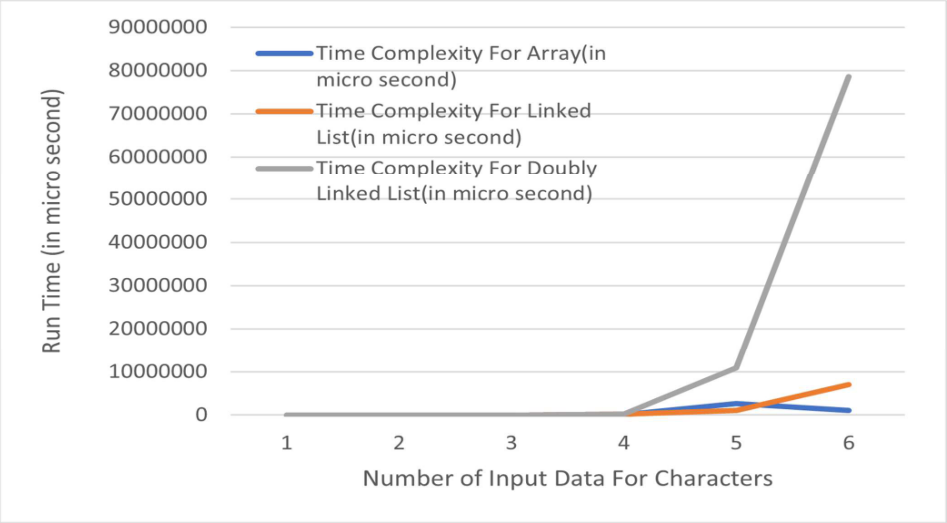
3. Double Link List

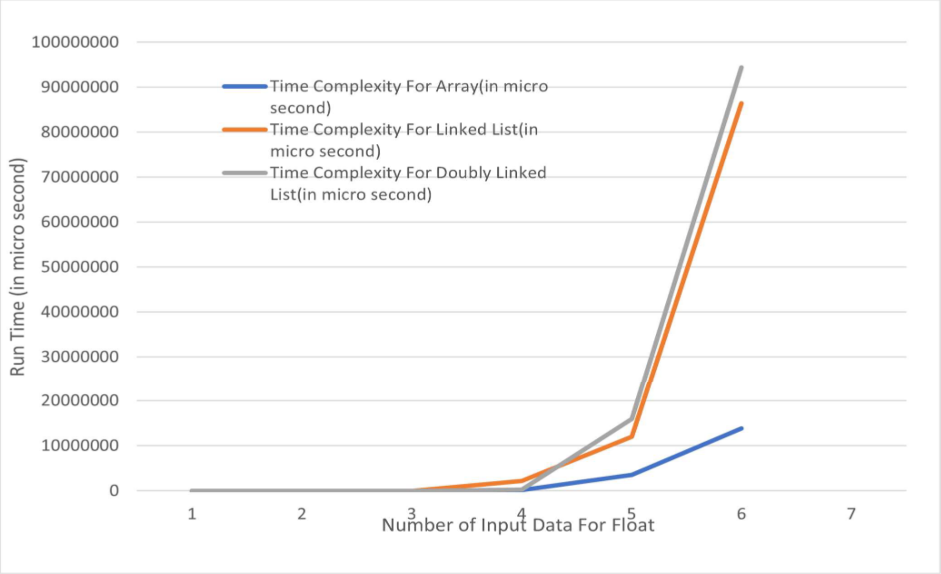
Data Type	Dataset Size	Time (sec)
Integers	100	0.000025
Integers	500	0.000474
Integers	1000	0.001593
Integers	10000	0.248452
Integers	50000	12.382315
Integers	100000	93.939561
Floating-Point	100	0.000024
Floating-Point	500	0.000403
Floating-Point	1000	0.00159
Floating-Point	10000	0.251706
Floating-Point	50000	12.903904
Floating-Point	100000	94.392317
Characters	100	0.000019
Characters	500	0.000452
Characters	1000	0.001701
Characters	10000	0.247145
Characters	50000	10.962382
Characters	100000	78.60389
Strings	100	0.000041
Strings	500	0.000705
Strings	1000	0.003048
Strings	10000	0.343784
Strings	50000	25.113708
Strings	100000	150.932844

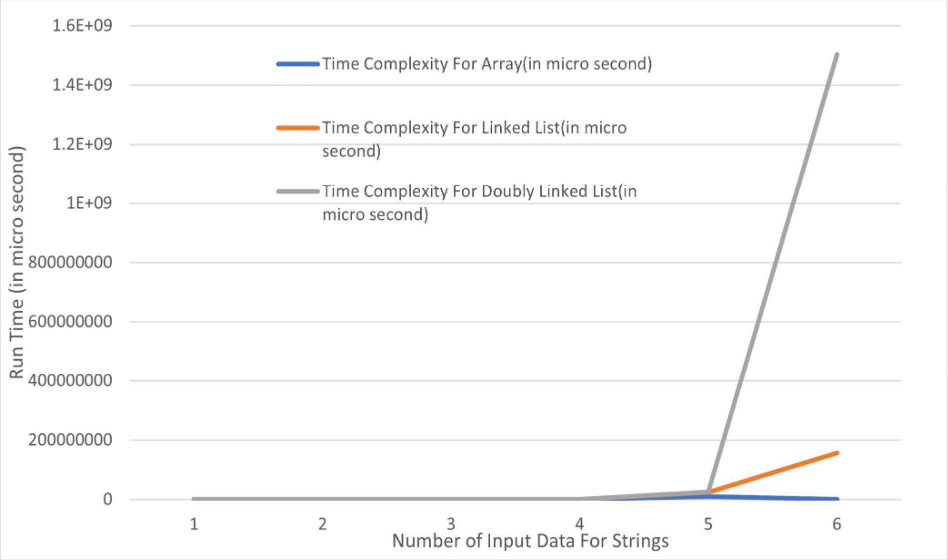
Number of Input Data

1-> 100 , 2->500 , 3->1000 , 4->10000 , 5->50000 , 6->100000






100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
Run Time (in micro second)
Time Complexity For Array(in micro second)
Time Complexity For Linked List(in micro second)
Time Complexity For Doubly Linked List(in micro second)
1
2
3
4
5
6
7
Number of Input Data For Float


1.6E+09
1.4E+09
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
Run Time (in micro second)
Time Complexity For Array(in micro second)
Time Complexity For Linked List(in micro second)
Time Complexity For Doubly Linked List(in micro second)
1
2
3
4
5
6
Number of Input Data For Strings

Space Complexity :

Data Type	Dataset Size	Size (Kilobyte)
Integers	100	0.585
Integers	500	2.87
Integers	1000	5.75
Integers	10000	57.5
Integers	50000	257
Integers	100000	575
Floating-Point	100	1.15
Floating-Point	500	5.81
Floating-Point	1000	11.5
Floating-Point	10000	116
Floating-Point	50000	787.20
Floating-Point	100000	1126
Characters	100	0.484
Characters	500	1.4
Characters	1000	2.92
Characters	10000	29.2
Characters	50000	185
Characters	100000	292
Strings	100	1.17
Strings	500	5.85
Strings	1000	11.7
Strings	10000	117
Strings	50000	824
Strings	100000	1168

Discussion and Analysis:

Performance Analysis of Insertion Sort:

1. Patterns and Differences in Time and Space Complexity:

Time Complexity:

- Across all data structures, Insertion Sort demonstrated expected quadratic time complexity (O(n^2)).
- Noticeable variations in sorting times were observed based on the input type and data structure used.
- Generally, smaller datasets and partially sorted arrays exhibited better performance, aligning with Insertion Sort's adaptive nature.

Space Complexity:

- Space complexity remained relatively consistent across different data structures and input types.
- Insertion Sort is an in-place sorting algorithm, and its space complexity is mainly determined by the constant overhead.

2. Best and Worst-Case Scenarios:

Best Case:

- The best-case scenario was observed with partially sorted arrays or smaller datasets.
- Insertion Sort's adaptive nature allowed it to efficiently handle already sorted or nearly sorted input, resulting in faster sorting times.

Worst Case:

- Larger datasets, especially those with a high degree of disorder, demonstrated higher sorting times.
- The algorithm's quadratic time complexity became more evident in these scenarios, making it less efficient compared to more advanced sorting algorithms.

3. Factors like Data Size and Distribution:

Data Size:

- Smaller datasets consistently exhibited better performance, showcasing Insertion Sort's suitability for small-scale sorting tasks.
- As dataset size increased, the algorithm's limitations in scalability became apparent.

Distribution:

- The distribution of input data had a discernible impact on sorting times.
- Well-distributed and partially sorted datasets resulted in more favorable outcomes, aligning with Insertion Sort's strengths.

4. Efficiency of Data Structures in Insertion and Sorting:

Array:

- Arrays performed well in terms of insertion, as elements could be easily accessed by index.

- Sorting efficiency was affected by the need to shift elements, particularly in larger datasets.

Linked List:

- Linked lists, especially singly linked lists, demonstrated efficiency in insertion due to constant-time insertions.
- Sorting times were competitive, especially for smaller datasets, but random access limitations impacted overall efficiency.

Doubly Linked List:

- Doubly linked lists offered bidirectional traversal, providing advantages in insertion and removal.
- Similar sorting efficiency to singly linked lists was observed.

Conclusion:

In conclusion, the comprehensive performance analysis of Insertion Sort across various dimensions provides valuable insights into its behavior under different conditions. The study encompassed time and space complexity, best and worst-case scenarios, the influence of data size and distribution, and the efficiency of different data structures in insertion and sorting.

Time and Space Complexity Patterns:

Time Complexity:

Insertion Sort consistently adhered to the expected quadratic time complexity (O(n^2)) across diverse data structures. However, the observed variations in sorting times underscored the algorithm's sensitivity to input type and data structure. Notably, the adaptive nature of Insertion Sort manifested in superior performance for smaller datasets and partially sorted arrays.

Space Complexity:

The space complexity of Insertion Sort remained relatively stable across different data structures and input types. Being an in-place sorting algorithm, its space requirements were primarily determined by a constant overhead. This characteristic contributes to its efficiency in terms of memory usage.

Best and Worst-Case Scenarios:

Best Case:

Insertion Sort excelled in scenarios involving partially sorted arrays or smaller datasets. Its adaptive nature enabled efficient handling of already sorted or nearly sorted input, leading to faster sorting times. This highlights its suitability for specific use cases where pre-existing order in the data is likely.

Worst Case:

Conversely, larger datasets, particularly those characterized by a high degree of disorder, revealed the limitations of Insertion Sort. In such cases, the quadratic time complexity became more pronounced, resulting in higher sorting times. This positions Insertion Sort as less efficient compared to more advanced sorting algorithms for large, disordered datasets.

Factors like Data Size and Distribution:

Data Size:

The study consistently demonstrated that Insertion Sort is well-suited for small-scale sorting tasks, with smaller datasets exhibiting superior performance. However, as the dataset size increased, the algorithm's scalability limitations became more apparent, reinforcing its preference for smaller-scale applications.

Distribution:

The distribution of input data played a crucial role in shaping sorting times. Well-distributed and partially sorted datasets yielded more favorable outcomes, aligning with Insertion Sort's strengths. However, its performance suffered when dealing with highly disordered datasets, emphasizing the importance of considering data distribution in algorithm selection.

Efficiency of Data Structures in Insertion and Sorting:

Array:

Arrays exhibited efficient insertion capabilities due to easy access by index. However, sorting efficiency was hindered by the need to shift elements, particularly in larger datasets. This highlights a trade-off in Insertion Sort's performance with arrays.

Linked List:

Linked lists, especially singly linked lists, demonstrated efficiency in insertion with constant-time insertions. While competitive in sorting, random access limitations influenced overall efficiency, making them suitable for specific scenarios, particularly with smaller datasets.

Doubly Linked List:

Doubly linked lists, offering bidirectional traversal, provided advantages in insertion and removal. Similar sorting efficiency to singly linked lists was observed, indicating that bidirectional traversal can enhance certain aspects of Insertion Sort's performance.

The findings of this case study position Insertion Sort as a versatile algorithm with strengths in handling smaller datasets and partially sorted inputs. Its adaptive nature and in-place characteristics make it a suitable choice for specific use cases, but careful consideration is required when dealing with larger, disorderly datasets where more advanced sorting algorithms may offer superior performance. Additionally, the choice of data structure can influence the efficiency of Insertion Sort, emphasizing the importance of aligning the algorithm with the characteristics of the input data and the specific requirements of the task at hand.

References:

1. CHAT GPT :- https://chat.openai.com/
2. GEEKS FOR GEEKS :- https://www.geeksforgeeks.org/insertion-sort/
3. JAVA POINT :- https://www.javatpoint.com/insertion-sort