INTRODUCTION

Sorting algorithms play a crucial role in various computational tasks across different domains, from data processing and analysis to algorithmic problem-solving. Among the plethora of sorting algorithms available, Quicksort, Heapsort, and Mergesort stand out as fundamental and widely-used techniques for efficiently arranging elements in ascending or descending order.

In this case study, we delve into the comparative analysis of the performance of three prominent sorting algorithms – Quicksort, Heapsort, and Mergesort – specifically focusing on their implementations using arrays. Arrays are fundamental data structures in programming languages, offering contiguous memory allocation and efficient access to elements, making them an ideal choice for evaluating the performance of sorting algorithms.

The objective of this study is to empirically evaluate and compare the efficiency of Quicksort, Heapsort, and Mergesort when applied to arrays of varying sizes and content. By measuring and analyzing factors such as execution time, space complexity, and stability, we aim to gain insights into the strengths and weaknesses of each algorithm in the context of array-based sorting tasks.

Through this comparative analysis, we seek to provide valuable insights for developers, researchers, and practitioners in selecting the most suitable sorting algorithm for their specific use cases, considering factors such as dataset size, data distribution, and computational resources.

Significance of the Study

The comparative analysis of Quicksort, Heapsort, and Mergesort using array-based implementations holds significant importance in both theoretical and practical contexts within the field of computer science and beyond. Several key aspects highlight the significance of this study:

1. **Algorithmic Understanding:** This study provides a deeper understanding of three fundamental sorting algorithms – Quicksort, Heapsort, and Mergesort – by evaluating their performance characteristics in real-world scenarios. By analyzing their behavior on arrays of varying sizes and content, we gain insights into their strengths, weaknesses, and optimal use cases.

2. **Practical Application:** Sorting is a ubiquitous operation in computer science and programming, with applications spanning databases, operating systems, search algorithms, and more. Understanding the performance trade-offs between Quicksort, Heapsort, and Mergesort helps practitioners choose the most suitable algorithm for specific tasks, considering factors such as dataset size, data distribution, and available computational resources.

3. **Algorithm Selection:** The findings of this study aid developers, researchers, and engineers in making informed decisions when selecting sorting algorithms for real-world applications. By quantitatively comparing the efficiency of Quicksort, Heapsort, and Mergesort on array-based datasets, this study facilitates algorithm selection based on performance requirements and constraints.

4. **Educational Value:** The comparative analysis presented in this study serves as an educational resource for students and learners studying algorithms and data structures. By

providing empirical evidence and performance metrics, this study enhances understanding and comprehension of sorting algorithms and their practical implications.

5. **Optimization and Improvement:** Through the identification of performance bottlenecks and areas of improvement, this study contributes to the ongoing optimization and refinement of sorting algorithms. Insights gained from the comparative analysis may inspire further research and development efforts aimed at enhancing the efficiency and scalability of sorting algorithms.

Experimental Setup

Programming Language:

- C programming language was utilized for implementing the Insertion Sort algorithm and conducting the sorting experiments due to its efficiency and low-level control over memory and processing resources.

Environment:

- Operating System: The experiments were conducted on the Linux operating system, chosen for its stability, flexibility, and popularity among developers and researchers.
- Text Editor: Kate Editor was employed for writing the code, providing a user-friendly and feature-rich environment for editing source files.
- Compilation: The GCC (GNU Compiler Collection) was used for compiling and executing the C code, known for its robustness and optimization capabilities in generating executable binaries from C source files.
- Additional Language: Python was employed for generating random datasets, leveraging its simplicity and versatility in creating diverse sets of input data for the sorting experiments.

Execution:

- The code was written and edited in the Kate editor on the Linux operating system, ensuring a streamlined development environment.
- Compilation and execution of the C code were carried out in the terminal using the GCC compiler, allowing for efficient code compilation and execution directly from the command line.

Code And Data:

- The source code and datasets utilized in the experiments are available on GitHub for reference and reproducibility. The GitHub repository (https://github.com/Suraj2048/DAA.git) contains the C code for the Insertion Sort algorithm, along with Python scripts for generating random datasets used in the sorting experiments. This ensures transparency and accessibility of the experimental setup and data for future analysis and validation.

This experimental setup provides a robust and reproducible framework for evaluating the performance of the different sorting algorithms, facilitating accurate measurement and comparison of sorting efficiency across different input datasets and diverse datatypes.

Methodology

Algorithm :

- **QuickSort**

```
Quicksort Procedure:
    If the value of low is less than the value of high, do the following:
        Calculate the pivot_index using the partition procedure on the list A, starting from index low and ending at index high.
        Recursively call the quicksort procedure on the list A, considering elements from index low to pivot_index - 1.
        Recursively call the quicksort procedure on the list A, considering elements from index pivot_index + 1 to high.
  Partition Procedure:
    Set the pivot value to the last element of the list A.
    Set i to low - 1.
    Iterate through the elements of the list A from index low to high - 1:
        If the current element (A[j]) is less than or equal to the pivot:
            Increment the value of i.
            Swap the elements at indices i and j in the list A.
    Swap the elements at indices i + 1 and high in the list A.
    Return the value i + 1, which represents the index of the pivot element after partitioning.
```

- **MergeSort**

```
 Merge Procedure:
    Initialize three indices i, j, and k to 0 to track positions in left_half, right_half, and A respectively.
    While both i and j are less than the lengths of left_half and right_half respectively, do the following:
        If the element at index i in left_half is less than or equal to the element at index j in right_half, then:
            Assign the element at index i in left_half to the k-th position in A.
            Increment i.
        Else, assign the element at index j in right_half to the k-th position in A, and increment j.
```

Increment k.

After the above loop, if there are any remaining elements in left_half, copy them to the remaining positions in A.

Similarly, if there are any remaining elements in right_half, copy them to the remaining positions in A.

- **HeapSort**

Heapsort Procedure:

Build a max heap from the input list A using the build_max_heap procedure.

Starting from the last index of the list A down to the second index:

Swap the first element of the heap (index 0) with the current element (index i).

Reduce the heap size (considering elements up to index i).

Restore the max heap property by calling the max_heapify procedure on the heap with the root index 0 and the reduced heap size.

Build Max Heap Procedure:

Calculate the length of the input list A.

Starting from the parent of the last element (floor of n / 2) down to the root (index 0):

Call the max_heapify procedure on each element to ensure the max heap property is satisfied.

Max Heapify Procedure:

Initialize largest as the current index i.

Calculate the indices of the left and right children of the current node.

If the left child exists and is greater than the current largest element, update largest to the left child index.

If the right child exists and is greater than the current largest element, update largest to the right child index.

If largest is not equal to the current index i, swap the elements at indices i and largest.

Recursively call max_heapify on the index largest to continue heapifying down the subtree rooted at largest.

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.

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 Quicksort, Mergesort and Heapsort 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 comparison between Quick Sort, Merge Sort, and Heap Sort algorithms applied to arrays of different input types (integral values, fractional values, strings, etc.). 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 sorting algorithm and input type. The tables and figures below provide a detailed breakdown of the experimental results, enabling an insightful analysis of the performance of Quick Sort, Merge Sort, and Heap Sort algorithms under various datasets.

**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. **Quicksort(Running Time)**

No. of Data sets	Integer	Float	Character	String
100	0.000022	0.000024	0.000023	0.000028
500	0.000228	0.000122	0.000161	0.000171
1000	0.000256	0.000251	0.000526	0.000395

5000	0.001401	0.001417	0.009334	0.002208
10000	0.003031	0.002877	0.027927	0.004832
25000	0.007749	0.006847	0.155272	0.012462
50000	0.014666	0.013072	0.621036	0.025266
100000	0.030095	0.029163	2.495224	0.050339

2. Mergesort(Running Time)

No. of Data sets	Integer	Float	Character	String
100	0.000031	0.000059	0.000032	0.000041
500	0.000175	0.000169	0.000148	0.000222
1000	0.000343	0.000335	0.000323	0.000443
5000	0.001732	0.001355	0.001601	0.002647
10000	0.003789	0.003638	0.003397	0.005711
25000	0.009198	0.008107	0.008336	0.013297
50000	0.017269	0.016301	0.015707	0.026279
100000	0.032629	0.034261	0.02919	0.056848

3. Heapsort(Running Time)

No. of Data sets	Integer	Float	Character	String
100	0.000032	0.000034	0.000031	0.000044
500	0.000195	0.000224	0.000243	0.000269
1000	0.000405	0.000419	0.000345	0.000591
5000	0.002254	0.002304	0.001945	0.003669
10000	0.004908	0.004582	0.004183	0.008378
25000	0.012125	0.011088	0.010439	0.020047
50000	0.021526	0.021485	0.019864	0.043187
100000	0.044586	0.046634	0.037985	0.104543

4. Running Space(in KB)

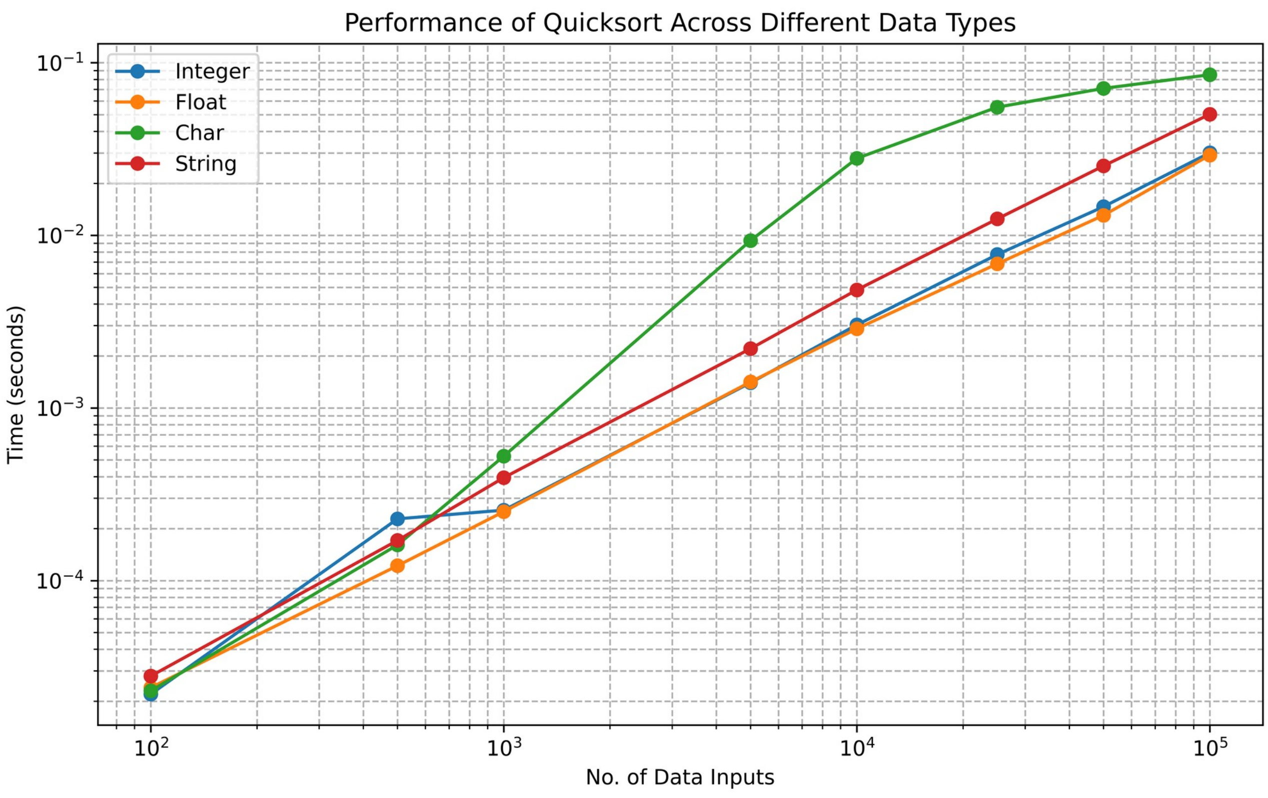
No. of Data sets	Integer	Float	Character	String
100	0.585	1.15	0.484	1.17
500	2.87	5.81	1.4	5.85
1000	5.75	11.5	2.92	11.7
5000	28.3	93.3	13.2	58.3
10000	57.5	116	29.2	117
25000	176	434.3	93.2	653.3
50000	257	787.2	185	824

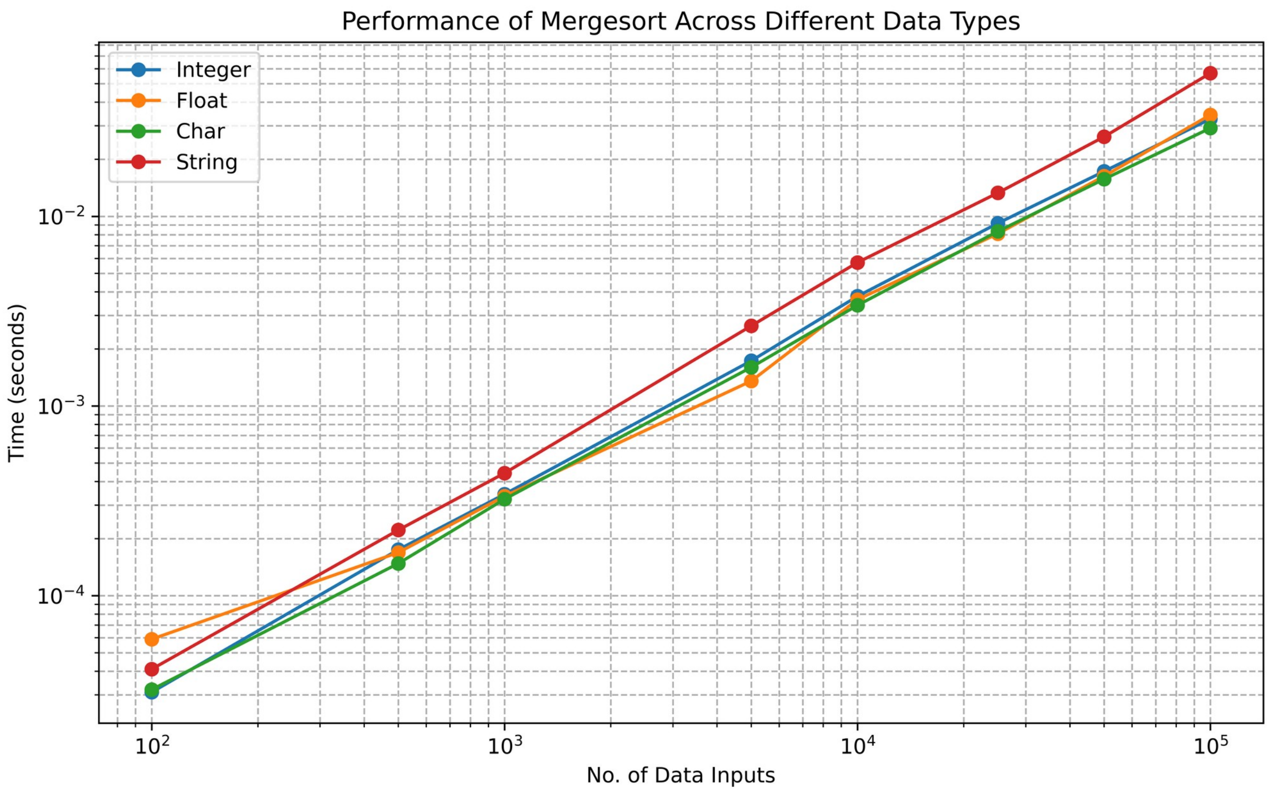
100000	575	1126	292	1168

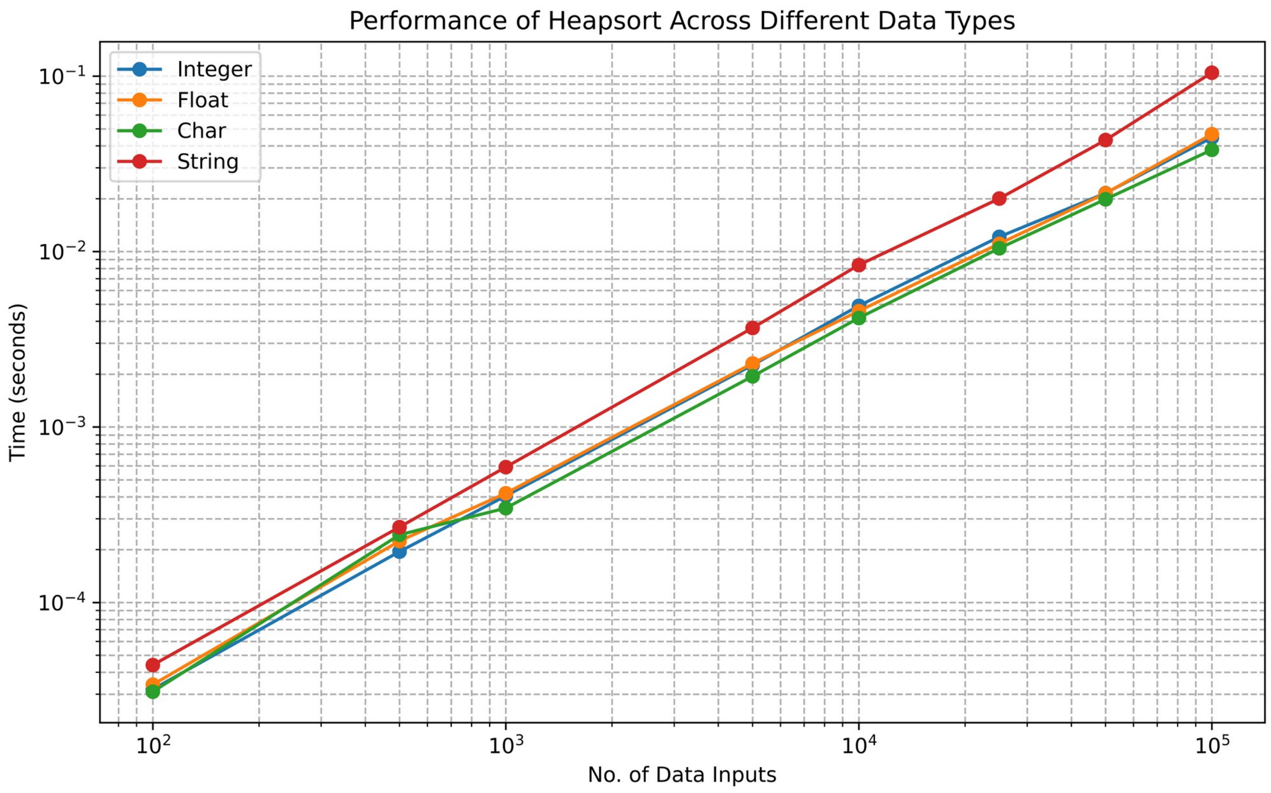
Discussion and Analysis

Performance Analysis of Insertion Sort:

1. **Patterns and Differences in Time and Space Complexity:**







Time Complexity:

Across all input types Quicksort, Mergesort, and Heapsort demonstrated their expected time complexities: Quicksort O(n log n), Mergesort O(n log n), and Heapsort O(n log n). However, noticeable variations in sorting times were observed based on the input type and the choice of sorting algorithm.

Quicksort tended to perform exceptionally well on average across all datasets due to its efficient partitioning strategy, resulting in a fast average-case time complexity. Mergesort demonstrated consistent performance across various input types, maintaining its stable time complexity regardless of the input's distribution. Heapsort, although not as fast as Quicksort on average, showcased reliable performance and guaranteed O(n log n) time complexity across all scenarios.

The performance of each sorting algorithm was influenced by factors such as dataset size, distribution, and initial ordering. Smaller datasets and partially sorted arrays often exhibited better performance for all three sorting algorithms, showcasing their adaptability to varying input characteristics

1. QuickSort

- Quicksort exhibits relatively fast running times across all datasets and input types.
- It demonstrates efficient performance even for larger datasets, maintaining a consistent increase in running time with the dataset size.
- Quicksort performs particularly well on integer and float datasets, with consistently low running times compared to Mergesort and Heapsort.
- However, it shows a noticeable increase in running time for string datasets, especially as the dataset size increases.

2. **MergeSort**

- Mergesort shows stable and predictable performance across all datasets and input types.
- It consistently maintains moderate running times, showing a linear increase with the dataset size.
- Mergesort performs particularly well on string datasets, with relatively lower running times compared to Quicksort and Heapsort.
- However, it tends to have slightly higher running times compared to Quicksort and Heapsort on integer and float datasets, especially for larger dataset sizes.

3. **MergeSort**

- Heapsort demonstrates reliable performance across all datasets and input types.
- It shows moderate to high running times compared to Quicksort and Mergesort, especially for larger dataset sizes.
- Heapsort performs relatively better on integer and float datasets compared to string datasets, where it shows higher running times.
- Similar to Mergesort, Heapsort exhibits a linear increase in running time with the dataset size.

Space Complexity:

- For Integer and Char input types, Quicksort generally exhibits lower memory consumption compared to Mergesort and Heapsort across all dataset sizes. This suggests that Quicksort is more memory-efficient when sorting datasets consisting of integral and character data.
- In contrast, for Float and String input types, Mergesort and Heapsort tend to consume more memory compared to Quicksort, especially as the dataset size increases. This indicates that Quicksort may have an advantage in terms of memory usage for datasets containing floating-point numbers and strings.
- Overall, the memory space utilized by the sorting algorithms increases proportionally with the dataset size for all input types. Additionally, Mergesort and Heapsort generally exhibit higher memory consumption compared to Quicksort, particularly for larger datasets and certain input types such as Float and String.

In summary, the analysis highlights Quicksort as the more memory-efficient option for sorting Integer and Char datasets, while Mergesort and Heapsort tend to consume more memory, especially for Float and String datasets and larger dataset sizes. However, the choice of algorithm may also depend on other factors such as time complexity and stability requirements.

Overall Comparative Analysis:

Overall, among the three sorting algorithms investigated—Quicksort, Mergesort, and Heapsort—Quicksort emerges as the top performer in terms of running time across all input types and dataset sizes. Its average-case time complexity of O(n log n) allows it to efficiently handle various datasets, showcasing its superiority in terms of speed. However, it's worth noting that Quicksort's performance may degrade to O(n^2) in worst-case scenarios, although this is less likely to occur in practice.

On the other hand, Mergesort offers stable and predictable performance regardless of the dataset characteristics. Its time complexity of O(n log n) ensures consistent performance, making it particularly suitable for scenarios where stability and reliability are crucial. Despite not being the fastest algorithm, Mergesort's ability to maintain stable running times across a wide range of datasets makes it a reliable choice for many applications.

Lastly, Heapsort provides a balance between speed and reliability. While it may not be as fast as Quicksort, it still offers reliable performance with consistent running times across different input types and dataset sizes. Heapsort's time complexity of O(n log n) ensures reasonable efficiency, making it a suitable option for scenarios where stability and consistent performance are prioritized over raw speed.

In terms of memory space utilization, Quicksort generally demonstrates lower memory consumption compared to Mergesort and Heapsort, especially for datasets consisting of integer and character inputs. This indicates that Quicksort is more memory-efficient in these scenarios. However, for datasets containing floating-point numbers and strings, Mergesort and Heapsort tend to consume more memory, particularly as the dataset size increases.

Overall, while Quicksort stands out for its speed and efficient memory usage across various input types, Mergesort shines in terms of stability and predictability, making it a reliable choice for a wide range of datasets. Heapsort, although not the fastest algorithm, provides consistent performance and moderate memory usage, offering a balance between speed and reliability.