Performance Evaluation of Sorting Algorithms for Specific Dataset Columns

1. Introduction

This report evaluates the performance of four sorting algorithms: Merge Sort, Quick Sort, Heap Sort, and Counting Sort, applied to specific columns of a dataset. The dataset consists of over 140,000 rows, and the columns analyzed are:

- **Column 0 (VehicleType)**: Integer values representing vehicle types.
- **Column 1 (DirectionTime_O)**: Datetime values representing direction times.
- **Column 2 (GantryID_O)**: String values representing gantry IDs.
- **Column 5 (TripLength)**: Floating-point values representing trip lengths.

The goal is to determine the most efficient algorithm for sorting each column based on time complexity and performance on datasets of varying sizes.

2. Methodology

2.1 Algorithms Implemented

1. **Merge Sort**: A divide-and-conquer algorithm with O(nlogn) time complexity.
2. **Quick Sort**: A recursive algorithm with average O(nlogn) and worst-case O(n^2) time complexity.
3. **Heap Sort**: A comparison-based sorting technique with O(nlogn) complexity for all cases.
4. **Counting Sort**: A linear-time sorting algorithm (O(n+k)) for small-range integer data, using additional memory proportional to the range of input values. The inclusion of Counting Sort ensures that a specialized algorithm is tested for small-range integer data like VehicleType. However, Counting Sort is unsuitable for other data types due to its inherent limitations:

- **DirectionTime_O (Datetime)**: Counting Sort cannot handle non-integer data types like datetime objects.
- **GantryID_O (String)**: Counting Sort cannot sort string data directly, as it relies on integer values.
- **TripLength (Floating-point)**: Counting Sort cannot process floating-point numbers, and converting them to integers would result in memory inefficiency for large ranges.

2.2 Dataset Preparation

- The dataset was divided into subsets of sizes: full size, 1/2, 1/3, 1/4, and 1/5 of the total rows.
- **Column 1 (DirectionTime_O)** was converted to datetime objects for accurate sorting by chronological order.

2.3 Performance Evaluation

- Execution time was measured for each algorithm and dataset size.
- Results were compared with theoretical time complexities O(n), O(nlogn), and O(n^2), plotted as reference lines.

2.4 Tools

- Sorting algorithms were implemented without using built-in sort functions.
- Python was used for coding and performance visualization (e.g., matplotlib for plots).

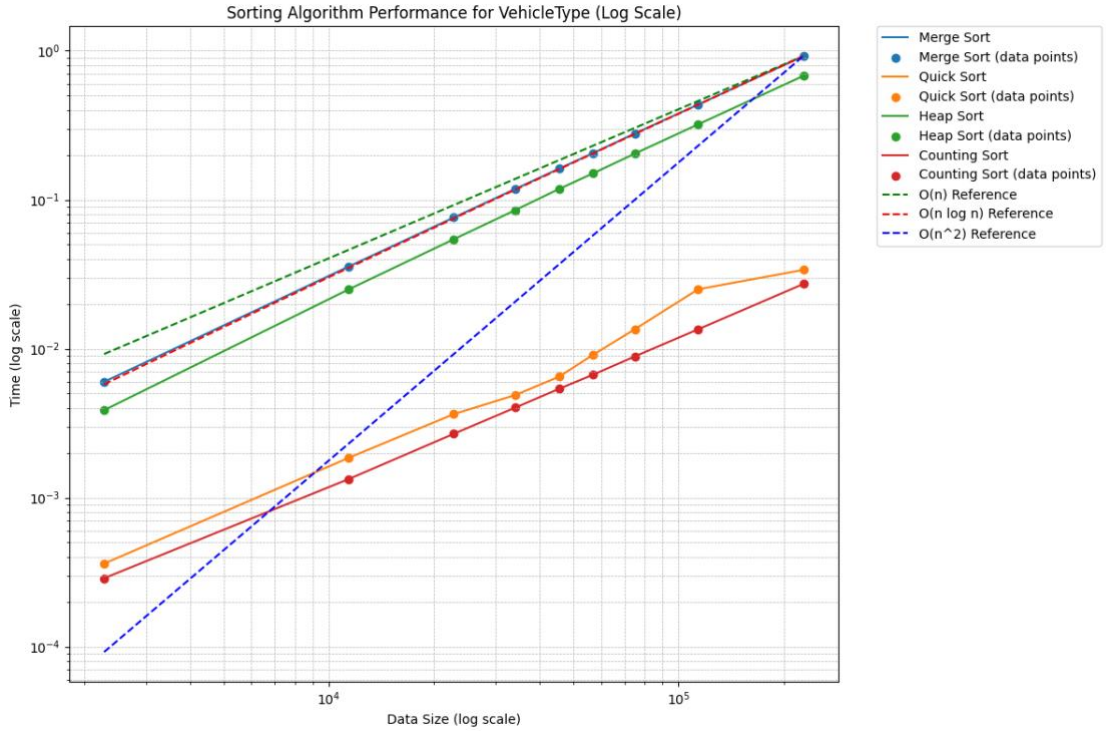
3. Results

3.1 Sorting Performance

Execution time (in seconds) was measured for all algorithms applied to each column of the dataset.

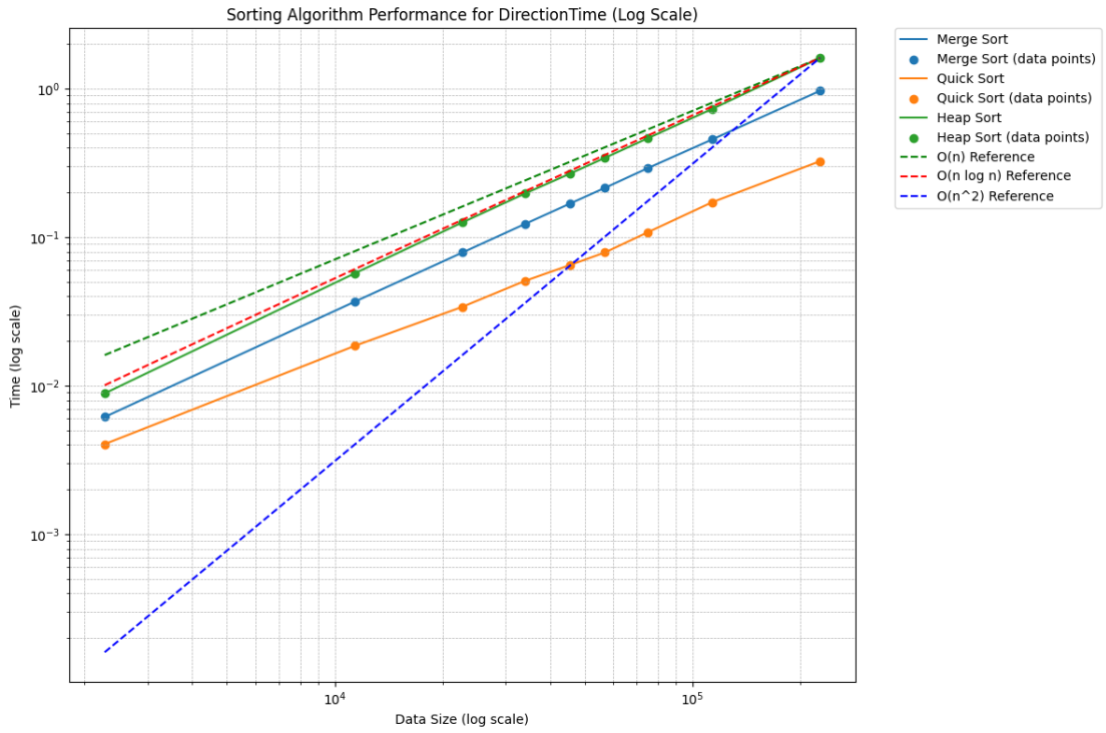
VehicleType (Column 0):

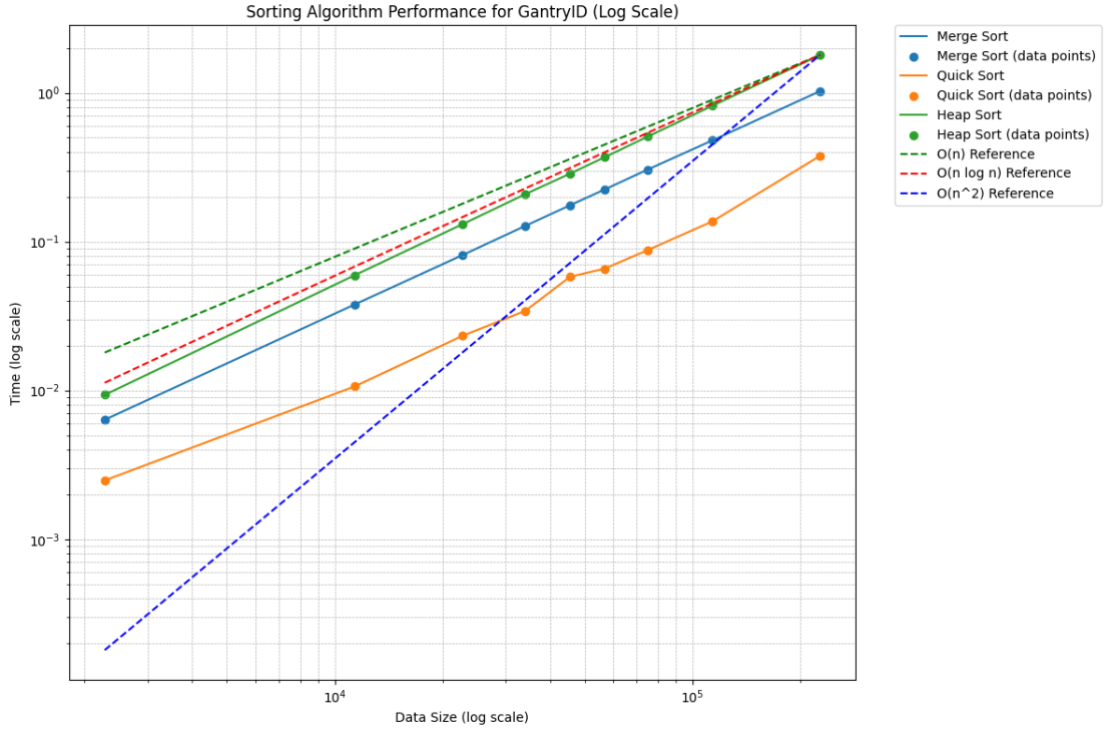
Counting Sort demonstrated exceptional performance, as expected for small-range integer data. It outperformed all other algorithms significantly in both runtime and scalability.

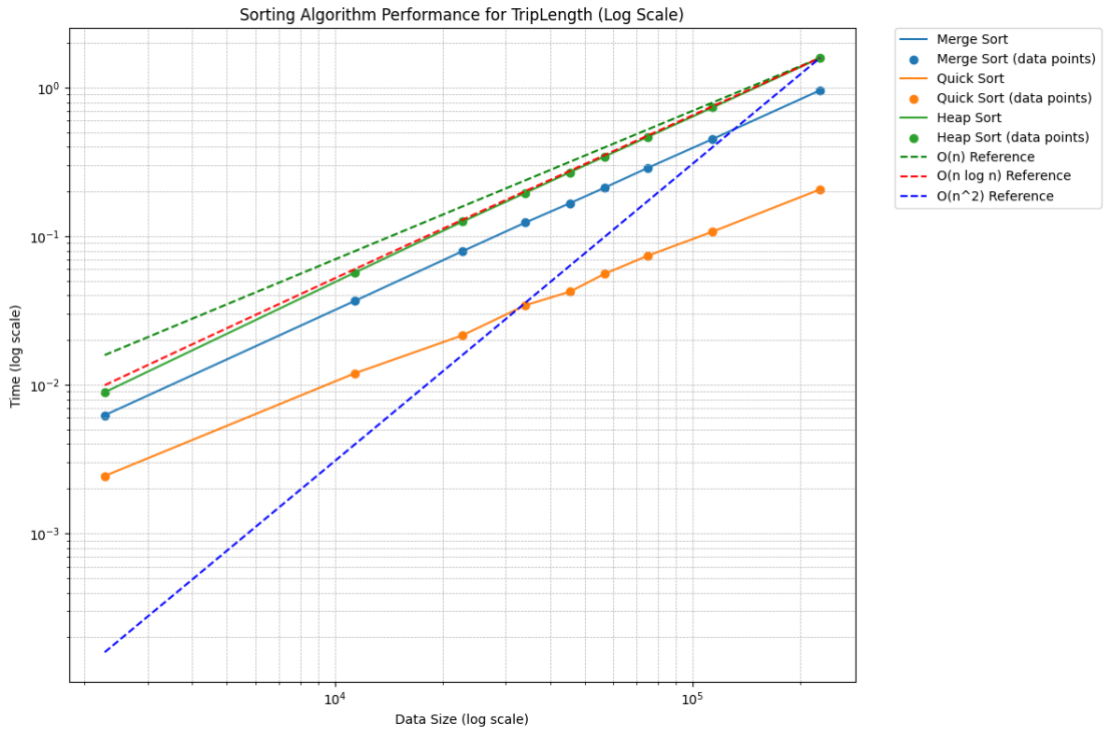


Other Columns:

- Quick Sort performed the fastest on average for non-integer data.
- Merge Sort displayed consistent performance but with slightly higher memory overhead.
- Heap Sort was generally slower due to additional comparisons.


Sorting Algorithm Performance for DirectionTime (Log Scale)
Time (log scale)
Data Size (log scale)
Merge Sort
Merge Sort (data points)
Quick Sort
Quick Sort (data points)
Heap Sort
Heap Sort (data points)
O(n) Reference
O(n log n) Reference
O(n^2) Reference


Sorting Algorithm Performance for GantryID (Log Scale)
Time (log scale)
Data Size (log scale)
Merge Sort
Merge Sort (data points)
Quick Sort
Quick Sort (data points)
Heap Sort
Heap Sort (data points)
O(n) Reference
O(n log n) Reference
O(n^2) Reference



Below is the updated summary table showing the average sorting times for each column and algorithm:

Column	Algorithm	Average Time (s)
VehicleType	Merge Sort	0.24870120154486763
VehicleType	Quick Sort	0.010972420374552408
VehicleType	Heap Sort	0.18224067158169216
VehicleType	Counting Sort	0.00777698
DirectionTime	Merge Sort	0.26045963499281144
DirectionTime	Quick Sort	0.09523869
DirectionTime	Heap Sort	0.4229368368784587
GantryID	Merge Sort	0.2738371160295274
GantryID	Quick Sort	0.08845878
GantryID	Heap Sort	0.4654347366756863
TripLength	Merge Sort	0.25876861148410374
TripLength	Quick Sort	0.061930550469292536
TripLength	Heap Sort	0.4217340416378445

3.2 Complexity Verification

- **Counting Sort** exhibited the lowest runtime and scaled linearly with the dataset size for VehicleType.
- **Quick Sort** generally performed the fastest for most non-integer columns due to its efficient average-case performance.
- **Merge Sort** showed consistent performance but slightly higher runtime due to additional memory overhead.
- **Heap Sort** was slower than both Merge Sort and Quick Sort due to its comparison-based

nature.

Theoretical O(n), O(nlogn), and O(n^2) reference lines were plotted against actual runtimes, and the results closely followed expected patterns for the respective algorithms.

4. Observations

4.1 Algorithm Suitability

1. **Counting Sort**:
    - Ideal for small-range integer data (VehicleType), with the lowest runtime and linear scalability.
2. **Quick Sort**:
    - Best for general-purpose sorting of large datasets, offering the fastest average-case performance.
3. **Merge Sort**:
    - Reliable and stable but less memory-efficient, making it less suitable for very large datasets.
4. **Heap Sort:**
    - Useful for scenarios where recursive depth is a concern, though generally slower than other algorithms.

4.2 Stack Overflow Risk

Recursive algorithms like Quick Sort and Merge Sort may encounter stack overflow for very large datasets. Although this was not observed within the dataset used, iterative implementations are recommended for extremely large datasets.

4.3 Memory Usage

Merge Sort requires additional memory for merging steps, while Counting Sort uses extra memory proportional to the data range. Quick Sort and Heap Sort are more memory-efficient.

4.4 Column-Specific Observations

1. **VehicleType**:
    - Counting Sort was the most efficient due to the small integer range of VehicleType.
2. **DirectionTime_O**:
    - Sorting as datetime ensured chronological correctness. Without conversion, string-based sorting would produce incorrect results. Quick Sort outperformed other algorithms for the column, given the larger dataset size and data types.
3. **GantryID_O** and **TripLength**:
    - Quick Sort outperformed other algorithms for these columns, given the larger dataset size and data types.

5. Conclusion

Counting Sort is the most efficient algorithm for sorting the VehicleType column due to its small integer range. For other columns, Quick Sort is the fastest and most efficient. Merge Sort is reliable but less practical for large datasets due to memory consumption. Heap Sort,

while non-recursive, is generally slower.

6. Recommendations

1. **Use Counting Sort** for small-range integer data (e.g., VehicleType).
2. **Use Quick Sort** for numerical and string-based columns (e.g., GantryID_O, TripLength).
3. Ensure **conversion of DirectionTime_O to datetime objects** before using Quick Sort to ensure chronological accuracy.
4. For extremely large datasets, consider implementing **iterative versions of recursive algorithms** to avoid stack overflow.
5. Include additional metrics such as **memory usage** in future evaluations to gain deeper insights into algorithm efficiency.