Introduction:

This study evaluates the performance of four distinct sorting algorithms -- Insertion Sort, Heap Sort, Quick Sort, and Merge Sort -- across seven different input array sizes. The objective is to empirically assess and compare the efficiency of each algorithm when handling large datasets of random integers.

Methods:

*Array Sizes:*

Seven different array sizes were selected for testing:

|  |
| --- |
| n = 300,000 |
| n = 400,000 |
| n = 500,000 |
| n = 600,000 |
| n = 750,000 |
| n = 900,000 |

(Minimum array sizes were chosen for runtimes >= 10ms)

*Data Population:*

For each specified array size, an empty array variable is created and every element was assigned a random integer value between X and Y using Java's Random class, which produces values with a uniform distribution across the entire array. The array generation process is as follows:

A computer screen with text

Description automatically generated

To ensure a consistent comparison, a copy of the original input array was created for each algorithm, guaranteeing that each sorting algorithm operated on identical and unsorted data.

A screen shot of a computer code

Description automatically generated

*Runtime Measurement:*

The measureRuntime(); method captured the start and stop times at the sorting function's invocation using Java's System.currentTimeMillis(); function. After sorting, each algorithm's output array was verified for correct ordering using the isSorted(); method, which simply iterates trough the array to ensure that it is arranged in increasing order.

A screen shot of a computer code

Description automatically generated

Validation function:

A computer screen with colorful text

Description automatically generated

*Execution and Output Logging:*

The runtime and validation results were printed to the console on each function invocation. For each array size, the program executed each sorting algorithm once.

A screen shot of a computer code

Description automatically generated

Results:

The main program ran 10 times to gather 10 data points for each algorithm in order to minimize unrelated artifacts.

All operations were performed on the same machine hardware and in the same execution environment to minimize external variables.

The runtime data for each sorting algorithm across the specified input sizes are summarized in Table 1.

***Table 1: Empirical Runtime (ms) of Sorting Algorithms***

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Average-Case Runtime Complexity** | **Worst-Case Runtime Complexity** | **Best-Case Runtime Complexity** |
| *Insertion Sort* |  |  |  |
| *Heapsort* |  |  |  |
| *Quicksort* |  |  |  |
| *Merge Sort* |  |  |  |

Table 2 presents the empirical average runtime values (in milliseconds) for each sorting algorithm based on varying input sizes.

***Table 2: Empirical Runtime (ms) of Sorting Algorithms***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms)** | | | | |
| **Input Array Size (# elements)** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge Sort** |
| 300000 | 6320.5 | 35.6 | 23.3 | 41.8 |
| 400000 | 11542.8 | 44.4 | 26.5 | 46.0 |
| 500000 | 16971.7 | 56.9 | 33.8 | 55.6 |
| 600000 | 24538.3 | 70.2 | 41.5 | 70.3 |
| 750000 | 38858.9 | 87.8 | 51.9 | 85.7 |
| 900000 | 56195.9 | 107.3 | 63.3 | 100.9 |
| 1000000 | 69863.7 | 123.4 | 70.9 | 117.1 |

* **Figure 1**: **Clustered Column Chart** displaying each algorithm’s runtime for given input sizes.
* **Figure 2**: **Line Chart** illustrating the relationship between input size (n) and total runtime, mirroring the trends observed in the clustered column chart.
* **Figure 3** and **Figure 4**: **Separated Line Charts** focusing on **O(n log n)** algorithms (Heapsort, Quicksort, Merge Sort) and **O(n²)** algorithm (Insertion Sort) respectively, providing a more precise view of their individual performance trends.

***Figure 1:*** shows a clustered column chart of each algorithm’s runtime for a given input size

***Figure 2:*** shows a line chart of input size (n) vs. the total run time, displaying the same trend as the clustered column chart

***Figure 3 and Figure 4:*** show the same as the previous line chart, however, the algorithms were separated from the insertion sort data to show a more precise view of the trends.

Discussion:

*Key Observations:*

* **1. Alignment with Theoretical Expectations**
* **The empirical results align closely with the theoretical time complexities outlined in Table 1:**
* **Insertion Sort: Exhibits a quadratic increase in runtime with increasing input sizes, which is consistent with its O(n²) time complexity. This steep growth renders it inefficient for large datasets, validating theoretical predictions.**
* **Heapsort, Quicksort, and Merge Sort: These algorithms, characterized by O(n log n) time complexities, display a much slower and more manageable increase in runtime as input sizes grow. This alignment underscores the theoretical advantage of these algorithms over Insertion Sort for large-scale data.**
* **2. Comparative Performance Analysis**
* **Quicksort vs. Heapsort and Merge Sort: Quicksort consistently outperforms both Heapsort and Merge Sort across all tested input sizes. This superior performance can be attributed to Quicksort's efficient in-place sorting and cache-friendly access patterns, which reduce overhead and enhance speed. Additionally, the choice of pivot selection strategies in Quicksort can significantly impact its practical performance, often mitigating the theoretical O(n²) worst-case scenario.**
* **Heapsort vs. Merge Sort: Both Heapsort and Merge Sort demonstrate steady and comparable runtimes. However, Merge Sort may incur additional overhead due to its need for auxiliary space during the merging process, whereas Heapsort operates in-place, potentially offering slight performance advantages in memory-constrained environments.**
* **3. Practical Implications**
* **Suitability for Large Datasets: Given the exponential growth in runtime, Insertion Sort is unsuitable for large datasets, reaffirming its theoretical limitations. In contrast, Quicksort, Heapsort, and Merge Sort are appropriate choices for handling large-scale data efficiently, with Quicksort emerging as the most practical due to its superior average-case performance.**
* **Choice of Sorting Algorithm: The empirical superiority of Quicksort suggests its preference in scenarios where average-case performance is prioritized. However, Merge Sort remains a viable option when stability is required, as it preserves the relative order of equal elements—a characteristic not inherently provided by Quicksort and Heapsort.**
* **4. Analysis of Anomalies and Observed Differences**
* **Quicksort's Superior Performance: The consistently lower runtimes of Quicksort may stem from its divide-and-conquer approach, which efficiently partitions the dataset. Additionally, optimized implementations that utilize randomized pivot selection can prevent the algorithm from degrading to its worst-case O(n²) runtime, enhancing its practical performance.**
* **Insertion Sort's Quadratic Growth: While Insertion Sort is theoretically efficient for nearly sorted data with a best-case O(n) runtime, the provided data likely reflects scenarios with random or reverse-sorted inputs, exacerbating its inefficiency and highlighting the importance of input data characteristics in algorithm performance.**
* **5. Similarities and Differences Between Empirical and Theoretical Results**
* **Consistent Scaling: The empirical data faithfully represent the expected scaling behaviors dictated by their respective Big-O notations. O(n log n) algorithms show a logarithmic increase in runtime, whereas O(n²) algorithms like Insertion Sort exhibit a quadratic increase.**
* **Performance Margins: Beyond asymptotic behavior, the empirical results reveal actual runtime differences influenced by constant factors and lower-order terms not captured by Big-O notation. For instance, the significant runtime disparity between Insertion Sort and the other algorithms underscores the practical implications of algorithmic inefficiencies beyond theoretical analysis.**
* **6. Limitations and Considerations**
* **Implementation Variations: The observed runtimes are contingent on specific implementations of each algorithm. Factors such as recursion depth in Quicksort, heap construction methods in Heapsort, and merge strategies in Merge Sort can impact performance. Optimizations in one implementation may not be present in another, affecting comparability.**
* **Hardware and Environment: The tests were conducted in a specific hardware and software environment, which may affect the generalizability of the results. Variations in CPU speed, memory hierarchy, and compiler optimizations can influence runtime measurements, potentially skewing comparisons.**
* **Data Characteristics: The input data's distribution (e.g., randomness, pre-sortedness) plays a pivotal role in algorithm performance. While the data provided does not specify these characteristics, they are critical factors in interpreting the results. For example, Quicksort can perform optimally on randomly ordered data but may suffer on already sorted or reverse-sorted datasets without proper pivot selection strategies.**
* **7. Recommendations for Future Research**
* **Exploration of Diverse Data Distributions: Investigating how different data distributions (e.g., already sorted, reverse-sorted, nearly sorted) affect each algorithm's performance can provide deeper insights into their practical applicability and robustness.**
* **Optimization Techniques: Implementing and analyzing optimization strategies, such as randomized pivot selection in Quicksort or in-place merging in Merge Sort, can enhance performance and mitigate worst-case scenarios, offering a more comprehensive understanding of algorithmic efficiencies.**
* **Comparative Analysis with Additional Algorithms: Extending the study to include other sorting algorithms like Radix Sort, Timsort, or Bucket Sort can offer a more holistic performance landscape, enabling more informed algorithm selection based on specific use-case requirements.**
* **Memory Usage Evaluation: Assessing memory consumption alongside runtime can present a balanced view of each algorithm's efficiency, especially critical in memory-constrained environments where space complexity plays a significant role in algorithm selection.**

Reference Data: