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.

***Github Repo:*** [***https://github.com/reidroberts24/CPSC-6109-Algorithms-Analysis-and-Design/tree/main/Assignment\_02\_Big\_Oh\_Exploration***](https://github.com/reidroberts24/CPSC-6109-Algorithms-Analysis-and-Design/tree/main/Assignment_02_Big_Oh_Exploration)

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:*

The empirical results confirm the theoretical complexities: Insertion Sort’s runtime grows quickly (O(n²)) and is thus unsuitable for large inputs, while Heapsort, Quicksort, and Merge Sort scale more efficiently (O(n log n)) with Quicksort often outperforming the others due to in-place sorting and cache-friendly behavior. Minor performance differences also arise from implementation details and data characteristics, but overall, the observed timings closely match their expected growth rates.

Reference Data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Average Runtime (ms)** | | | | |
| **Input 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 |
| 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 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 300,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 6342 | 34 | 23 | 40 |
| 2 | 6289 | 36 | 25 | 56 |
| 3 | 6322 | 35 | 22 | 40 |
| 4 | 6356 | 35 | 22 | 38 |
| 5 | 6278 | 37 | 24 | 43 |
| 6 | 6317 | 36 | 25 | 40 |
| 7 | 6318 | 35 | 23 | 39 |
| 8 | 6308 | 35 | 22 | 41 |
| 9 | 6360 | 37 | 23 | 41 |
| 10 | 6315 | 36 | 24 | 40 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 400,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 11483 | 43 | 25 | 46 |
| 2 | 11478 | 44 | 28 | 46 |
| 3 | 11486 | 47 | 27 | 46 |
| 4 | 11553 | 45 | 26 | 48 |
| 5 | 11437 | 45 | 26 | 44 |
| 6 | 11475 | 42 | 26 | 46 |
| 7 | 11480 | 43 | 26 | 46 |
| 8 | 11430 | 43 | 27 | 45 |
| 9 | 12133 | 45 | 27 | 46 |
| 10 | 11473 | 47 | 27 | 47 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 500,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 17257 | 54 | 34 | 55 |
| 2 | 16857 | 56 | 34 | 54 |
| 3 | 16940 | 56 | 35 | 56 |
| 4 | 16793 | 57 | 33 | 55 |
| 5 | 16831 | 58 | 32 | 56 |
| 6 | 16894 | 56 | 34 | 54 |
| 7 | 16807 | 56 | 34 | 62 |
| 8 | 16801 | 57 | 33 | 54 |
| 9 | 17715 | 55 | 33 | 55 |
| 10 | 16822 | 64 | 36 | 55 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 600,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 24905 | 69 | 41 | 73 |
| 2 | 24672 | 67 | 42 | 68 |
| 3 | 24576 | 68 | 41 | 69 |
| 4 | 24342 | 74 | 40 | 69 |
| 5 | 24409 | 69 | 42 | 71 |
| 6 | 24506 | 68 | 43 | 68 |
| 7 | 24393 | 78 | 42 | 78 |
| 8 | 24614 | 67 | 41 | 69 |
| 9 | 24605 | 68 | 41 | 69 |
| 10 | 24361 | 74 | 42 | 69 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 750,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 39684 | 85 | 52 | 85 |
| 2 | 38457 | 88 | 53 | 85 |
| 3 | 38933 | 91 | 51 | 85 |
| 4 | 38405 | 86 | 51 | 87 |
| 5 | 39869 | 91 | 52 | 84 |
| 6 | 38509 | 87 | 52 | 86 |
| 7 | 38904 | 88 | 52 | 89 |
| 8 | 38591 | 88 | 52 | 83 |
| 9 | 38897 | 88 | 51 | 87 |
| 10 | 38340 | 86 | 53 | 86 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 900,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 56010 | 105 | 61 | 98 |
| 2 | 55950 | 105 | 64 | 98 |
| 3 | 57030 | 106 | 62 | 98 |
| 4 | 55817 | 112 | 63 | 117 |
| 5 | 57219 | 109 | 62 | 100 |
| 6 | 56142 | 106 | 65 | 99 |
| 7 | 55779 | 106 | 63 | 98 |
| 8 | 56013 | 110 | 63 | 99 |
| 9 | 56057 | 106 | 63 | 103 |
| 10 | 55942 | 108 | 67 | 99 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Runtime (ms): n = 1,000,000** | | | | |
| **Test #** | **Insertion Sort** | **Heapsort** | **Quicksort** | **Merge sort** |
| 1 | 69607 | 126 | 72 | 118 |
| 2 | 69324 | 119 | 71 | 116 |
| 3 | 69180 | 119 | 70 | 116 |
| 4 | 69316 | 119 | 69 | 116 |
| 5 | 70369 | 122 | 70 | 120 |
| 6 | 69882 | 125 | 75 | 116 |
| 7 | 70186 | 123 | 71 | 119 |
| 8 | 71362 | 120 | 69 | 116 |
| 9 | 69777 | 118 | 70 | 116 |
| 10 | 69634 | 143 | 72 | 118 |