Hawk Weisman

Computer Science 112

Laboratory #4 Lab Report

PLEDGE:

Empirical Analysis of Sorting Algorithms

**Methods:**

In this laboratory assignment, our goal was to determine which of five sorting algorithms was the most effective way to sort an array of integers. The algorithms investigated were Bubble Sort, Selection Sort, Insertion Sort, and Quick Sort. The algorithms were implemented in Java by Duane Bailey, and tested using a testing application written by Professor Kapfhammer. Professor Kapfhammer’s sorting experiment application used Clarkware’s Profiler jarfile in order to determine the run time in milliseconds of each algorithm. Professor Kapfhammer’s experiment application tested each sorting algorithm using a randomly generated array of ints, Java’s signed 32-bit integer primitive data type.

Because I was interested in assessing the performance of these algorithms when sorting arrays with different contents, I modified Professor Kapfhammer’s experimental program to allow a command-line argument to control the contents of the arrays. The modified program was capable of generating completely random arrays, nearly sorted arrays, reversed arrays, and arrays containing a few unique values. Each combination of sorting algorithm and array contents was tested using the modified sorting experiment five times for arrays of 5, 10, 100, 1,000, 10,000, and 100,000 integers, and the result of each trial was averaged.

**Results:**

Table 1: Trial 1 Average Results (random array)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 10 | 100 | 1,000 | 10,000 | 100,000 |
| BubbleSort | 0.2 | 0 | 0.6 | 2.8 | 115.8 | 16166.6 |
| MergeSort | 0.2 | 0.2 | 0.2 | 3.4 | 3 | 21.4 |
| QuickSort | 0.6 | 0.4 | 0.8 | 2.2 | 3 | 15 |
| SelectionSort | 0.2 | 0.2 | 0.4 | 2.2 | 72.6 | 6451 |
| InsertionSort | 0 | 0 | 0.2 | 2.6 | 30.8 | 2325.8 |

Table 2: Trial 2 Average Results (few-unique)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 10 | 100 | 1,000 | 10,000 | 100,000 |
| BubbleSort | 0 | 0 | 0.4 | 2.4 | 144.2 | 18080 |
| MergeSort | 0 | 0 | 0.2 | 4.2 | 10.6 | 18.8 |
| QuickSort | 0.4 | 0.6 | 0.4 | 2 | 3 | 18.4 |
| SelectionSort | 0 | 0.2 | 0.4 | 2.8 | 73.6 | 7020.6 |
| InsertionSort | 0 | 0.2 | 0.2 | 2 | 23.6 | 2077 |

Table 3: Trial 3 Average Results (nearly sorted)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 10 | 100 | 1,000 | 10,000 | 100,000 |
| BubbleSort | 0.2 | 0 | 0 | 2.6 | 90.4 | N/A\* |
| MergeSort | 0.2 | 0.2 | 0.2 | 3.8 | 2.6 | N/A\* |
| QuickSort | 0.6 | 1.4 | 1.2 | 9 | 21.2 | N/A\* |
| SelectionSort | 0.2 | 0 | 1.8 | 2.8 | 104.4 | N/A\* |
| InsertionSort | 0.2 | 0.2 | 0.2 | 0.6 | 1.2 | N/A\* |

\*Note: running this test case caused a stack overflow error.

Table 4: Trial 4 Average Results (reversed array)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 10 | 100 | 1,000 | 10,000 | 100,000 |
| BubbleSort | 0.8 | 0.2 | 1.2 | 1.4 | 129.8 | N/A\* |
| MergeSort | 0.6 | 0.2 | 0.2 | 2.8 | 11.4 | N/A\* |
| QuickSort | 1 | 1 | 1 | 2.6 | 37.6 | N/A\* |
| SelectionSort | 1 | 0.2 | 0.4 | 1.6 | 117 | N/A\* |
| InsertionSort | 0.2 | 0 | 0.4 | 2 | 113.2 | N/A\* |

\*Note: running this test case caused a stack overflow error.

**Analysis and Visualization**

Across all tests, the absolute worst performance appears to be exhibited by the BubbleSort algorithm. This is not surprising, as BubbleSort has a reputation in the computer science community as “the generic bad sorting algorithm.” As seen in Figure 1, BubbleSort performs reasonably well on small arrays, but as the array size increases, its performance decreases dramatically. Analysis of the BubbleSort implementation’s source code reveals that BubbleSort contains two nested loops, which allows us to project that its worst-case performance should be O(*n*2). It’s best-case performance is very good, with O(*n*) complexity, but its average case performance is equivalent to its worse-case performance, at O(*n*2). This implies that the only time a BubbleSort implementation should be used is when it is known that all input will be the best case. As seen in Figure 1, BubbleSort’s performance doesn’t seem to vary significantly across different array types.

If BubbleSort is the worst algorithm, QuickSort and MergeSort would probably be considered the best. While QuickSort takes slightly longer to sort very small arrays than some algorithms, as array length increases, performance gains become obvious. As seen in Figure 2, it performs exceptionally well when sorting large arrays. For arrays of length 100,000, it is orders of magnitude better than practically every other algorithm besides Merge Sort. In cases where very small arrays, such as those under 100 elements in length, are known to be the only input, QuickSort might not be ideal, but in cases where large arrays are expected, or array lengths are unknown, QuickSort would be the best choice. QuickSort’s worst-case performance is O(*n*2), just like Bubble Sort, but its best and average-case performance are both O(*n* log *n*). While Bubble Sort’s average- and worst-case performances are equivalent, QuickSort performs much better in average cases. This indicates that QuickSort is a good choice for most situations.

However, it is worth noting that, during this experiment, the QuickSort implementation being tested caused stack overflow errors during the reversed and nearly-sorted array tests. Whether this is an issue unique to Bailey’s implementation of QuickSort or if this is an issue common across all versions of this algorithm is unknown. The Bailey implementation of QuickSort contains only one loop, but the loop contains a great deal of relatively complex code, including a lot of if/then logic.

Merge Sort is also a very strong contender. It performs slightly better than QuickSort on small arrays, and handles the long reversed and nearly sorted arrays better than QuickSort. QuickSort outperforms it on the few unique and random arrays. Merge Sort is said to have O(*n* log *n*) best, worst, and average case performance. Because it has better worst-case performance than QuickSort, it might be a better choice in some cases. Because Merge Sort operates by breaking the array being sorted into separate segments, Bailey’s iterative Merge Sort implementation contains three looping constructs, but they are not nested, and the contents of each looping construct only executes for some of the values being sorted.

Insertion and selection sort performed similarly, both were significantly worse than Merge Sort and QuickSort, but better, often greatly so, than BubbleSort. As seen in Figures 4 and 5, Insertion Sort tended to sort very large arrays faster than Selection Sort, sorting the 100,000-integer randomly ordered and few-unique arrays in 2325.8 and 2077 ms, respectively. This is significantly better than Selection Sort’s 6451 ms and 7020 ms. Performance on the 10,000-unit randomly-generated array was very similar.

However, as seen in Figures 6 and 7, there as a much more significant difference between the performance Insertion and Selection Sort while sorting the nearly-sorted array. Insertion Sort was able to sort the 10,000-element nearly-sorted array in a mere 1.2 seconds, not only faster than Selection Sort but faster than Merge Sort and Quick Sort as well.

Analysis of Bailey’s Insertion Sort and Selection Sort implementations reveals that both contain two nested iteration constructs, meaning that both algorithms should have worst-case complexities of O(*n*2). Additional research reveals that Insertion Sort has an average-case complexity of O(*n*2), and a best-case complexity of O(*n*) compares and O(1) swaps, while Selection Sort has a best-, worst-, and average-case complexity of O(*n*2). Based on this information, I infer that the nearly-sorted array is a best-case scenario for Insertion Sort, explaining its exceptional performance in this case. Selection Sort, therefore, despite being among the slower algorithms in general, might be well suited for applications where nearly sorted inputs are expected.

Based on this analysis of experimental data, we can conclude that in cases where large arrays of unknown contents are to be expected, either QuickSort or Merge Sort is probably the ideal choice. In cases where all input arrays are nearly sorted, Insertion Sort might also be worth considering. For cases where only very short arrays are to be expected, Bubble Sort might be worthy of consideration due to its simplicity and ease of implementation, but in general, it and Selection Sort are probably not good choices.