Rachel A Schifano

IT 279 - 3/24/14

Assignment 4 Paper: Which is the best sort of them all?

## Description

Four sorting algorithms were implemented in C++. Array assignments and comparisons were measured in order to determine the efficiency of each algorithm when handling array elements (n) - varying from small (up to 100) or to large (up to 100,000) elements. Comparisons were defined exclusively as any comparison between two array elements and not anything else (such as i < n). Assignments were defined exclusively as an assignment to or from an array element, and not other assignments (such as i = 0). The following sections for **Comparisons**, **Assignments**, and **Cost** analyze the collected data graphs in order to draw conclusions.

**Comparisons**

Figure 1.0 – “Comparisons Up to 10” to analyze performance of small n values

Figure 1.1 – “Comparisons Up to 100,000” to analyze performance of large n value

*Analysis*

* Insertion Sort makes the least amount of comparisons for smaller values up to 10, following by Heap Sort. Insertion and Heap sort may be more efficient at handling smaller values quicker as it does fewer comparisons.
* Quick Sort and then Merge Sort perform the most comparisons, nearly doubling or tripling the amount compared to Insertion and Heap Sort.
* There exists an upward trend in comparisons as the values of n increase, indicating a nearly exponential curve.
* Insertion Sort and Heap Sort perform with nearly identical comparisons with very small value of n = 5. It would appear that the true gain in performance from either algorithm might not be achieved until larger values.
* Quick Sort and Merge Sort perform in parallel to each other for very small value of n = 5, doubling comparison performance respective to Insertion Sort and Heap Sort. Again for these algorithms, their gain in performance may not be apparent until larger values of n.
* At larger values of n = 50,000, Quick Sort and Heap Sort perform nearly identical this time. They make fewer comparisons than the other sorts. At larger values n, Heap Sort appears to outperform Insertion Sort this time.
* Merge Sort and Insertion Sort appear to make far more comparisons than Heap Sort and Quick Sort, which contrast to their greater performance for smaller values.
* Merge Sort and Insertion Sort also continue trending exponentially as large values of n increase. In contrast, Quick Sort and Heap Sort do not increase comparisons as quickly as large values of n increase.

**Assignments**

Figure 1.2 – “Assignments Up to 10” to analyze performance of small n values

Figure 1.3 – “Assignments Up to 100,000” to analyze performance of large n values

*Analysis*

* Heap Sort and Merge Sort performed similarly in the amount of assignments made, in contrast to Heap Sort and Insertion Sort with Comparisons. They make more assignments at smaller values than previously with comparisons. This could mean that where Heap Sort succeeded in gain for small values and comparisons is also where gain is potentially lost.
* Insertion Sort performs fewer assignments with small values than previously with comparisons. Insertion Sort may demonstrate gains in sorting small values due to this.
* Insertion Sort overall demonstrates greater efficiency than the other algorithms in assignments even as values of n increase.
* There is a similar exponential trend of assignments as the elements of n grow for every sorting algorithm.
* All algorithms appear to continue trending for smaller values of n, maintaining a similar efficiency. Unlike the comparisons graphs, none of the algorithms outperform or underperform another algorithm.
* The predictable trending in performance for small values of n demonstrates an order of best to worst performance where Insertion Sort > Quick Sort > Heap Sort > Merge Sort.
* Larger values of n demonstrate a change in performance. In smaller values of n, Insertion Sort had outperformed Quick Sort. However, Quick Sort has now switched in efficiency and appears to make much fewer assignments than Insertion Sort.
* Heap Sort performs with fewer assignments similar to Quick Sort (transparency on the Quick Sort plots enabled a view of Heap Sort plots which are roughly identical in performance).
* Heap Sort does not perform as well with smaller values of n, and its gain can be seen with larger values.
* All sorts at larger values of n still increase exponentially. While other sorting algorithms increase gain as values of n increase, Merge Sort continues to make the most comparisons and assignments overall, thus indicating that Merge Sort may be the worst sorting algorithm among the four.
* Overall, Quick Sort performs with significantly fewest assignments. These fewer assignments coupled with the fewer comparisons made amongst the algorithms would make Quick Sort the most efficient algorithm for sorting.

**Cost**

Figure 1.4 – “Cost – Comparisons – C++”

Figure 1.5 – “Cost – Assignments – C++”

Figure 1.6 – “Cost – Comparisons – Java” where assignments are more expensive than comparisons

Figure 1.7 – “Cost – Assignments – C++” where assignments are more expensive than comparisons

*Analysis – Which Strategy Would be most Useful? C++ vs. Java*

* In C++ comparison are cheaper than object assignments. In Java – comparisons are more expensive than assignments.
* In sorting array of elements that are of large size n, cheaper comparisons may be most suited since, in order to sort many elements, you cannot predict the large amount of sub arrays that may be needed or values that will need switching and thus comparisons to.
* Arrays that are reversed or far from being sorted could benefit from having cheaper comparisons since it would, once again, require more comparisons in order to place them in the appropriate spot.
* Arrays that are reversed or far from being sorted would also need to benefit from having cheaper assignments since having more comparisons will equal to having more assignments necessary to switch places.
* The graphs show Insertion Sort, Merge Sort and Quick sort to trend exponentially with larger values of elements n.
* Between how C++ and Java work, they seem to perform similarly for larger values of n according to the graph.
* Sorting algorithms like Merge Sort or Quick Sort that require breaking down a given array into multiple smaller sub arrays could benefit from the cheaper assignments that are used in Java. It is possible that there are some algorithms that work best in either C++ or Java.

**Observed Data Tables**

*Comparison Data*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elements (n)** | **Heap** | **Insert** | **Merge** | **Quick** |
| 5 | 7 | 8 | 48 | 28 |
| 10 | 32 | 26 | 140 | 97 |
| 50 | 319 | 592 | 1968 | 2054 |
| 100 | 767 | 2172 | 6618 | 7161 |
| 500 | 5546 | 63360 | 132362 | 50844 |
| 1000 | 12588 | 256808 | 520340 | 105698 |
| 5000 | 79938 | 6183250 | 12644900 | 629022 |
| 10000 | 175234 | 24748400 | 5030550 | 1516450 |
| 50000 | 1049960 | 621218000 | 1251740000 | 17612900 |
| 100000 | 2233910 | 2465300000 | 5003630000 | 60174600 |

*Assignment Data*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elements (n)** | **Heap** | **Insert** | **Merge** | **Quick** |
| 5 | 39 | 16 | 40 | 22 |
| 10 | 90 | 44 | 120 | 64 |
| 50 | 551 | 690 | 1808 | 1105 |
| 100 | 107 | 51 | 146 | 85 |
| 500 | 7122 | 64358 | 132466 | 25484 |
| 1000 | 15216 | 258806 | 516432 | 51755 |
| 5000 | 875533 | 6193250 | 12622300 | 259413 |
| 10000 | 185239 | 24768400 | 50259600 | 523625 |
| 50000 | 1040080 | 621318000 | 1251510000 | 2632290 |
| 100000 | 2171360 | 2465500000 | 5003170000 | 5232030 |

*Cost - Comparison C++*

|  |  |  |  |
| --- | --- | --- | --- |
| **Elements (n)** | **Insert** | **Merge** | **Quick** |
| 5 | 8 | 48 | 28 |
| 10 | 26 | 140 | 97 |
| 50 | 592 | 1968 | 2054 |
| 100 | 2172 | 6618 | 7161 |
| 500 | 63360 | 132362 | 50844 |
| 1000 | 256808 | 520340 | 105698 |
| 5000 | 6183250 | 12644900 | 629022 |
| 10000 | 24748400 | 5030550 | 1516450 |
| 50000 | 621218000 | 1251740000 | 17612900 |
| 100000 | 2465300000 | 5003630000 | 60174600 |

*Cost – Assignment C++*

|  |  |  |  |
| --- | --- | --- | --- |
| **Elements (n)** | **Insert** | **Merge** | **Quick** |
| 5 | 32 | 80 | 44 |
| 10 | 88 | 240 | 128 |
| 50 | 1380 | 3616 | 2210 |
| 100 | 102 | 292 | 170 |
| 500 | 129076 | 264932 | 50968 |
| 1000 | 517612 | 1032864 | 103510 |
| 5000 | 12386500 | 25244600 | 518826 |
| 10000 | 49536800 | 100519200 | 104724 |
| 50000 | 124263600 | 2503020000 | 5264580 |
| 100000 | 4931000000 | 10006340000 | 10464060 |

*Cost – Comparison Java*

|  |  |  |  |
| --- | --- | --- | --- |
| **Elements (n)** | **Insert** | **Merge** | **Quick** |
| 5 | 16 | 96 | 56 |
| 10 | 52 | 280 | 194 |
| 50 | 1184 | 3936 | 4108 |
| 100 | 4344 | 13236 | 14322 |
| 500 | 126720 | 264724 | 101688 |
| 1000 | 513616 | 10400680 | 211396 |
| 5000 | 12366500 | 25289800 | 1258044 |
| 10000 | 49496800 | 10061100 | 3032900 |
| 50000 | 1242436000 | 2503480000 | 35225800 |
| 100000 | 4930600000 | 10007260000 | 120349200 |

*Cost – Assignment Java*

|  |  |  |  |
| --- | --- | --- | --- |
| **Elements (n)** | **Insert** | **Merge** | **Quick** |
| 5 | 16 | 40 | 22 |
| 10 | 44 | 120 | 64 |
| 50 | 690 | 1808 | 1105 |
| 100 | 51 | 146 | 85 |
| 500 | 64358 | 132466 | 25484 |
| 1000 | 258806 | 516432 | 51755 |
| 5000 | 6193250 | 12622300 | 259413 |
| 10000 | 24768400 | 50259600 | 523625 |
| 50000 | 621318000 | 1251510000 | 2632290 |
| 100000 | 2465500000 | 5003170000 | 5232030 |

Note: Randomly generated text files containing number of elements 5 through 100,000 were used to compare sorting algorithms. The same file corresponding to element values was used to fairly analyze algorithms.