CS590 Assignment 1

1. Abstract

This experiment explores the growth rate of function run time in respect to input size for several different sorting functions. The results show that, by pre-calculating vector lengths, the improved insertion sort algorithm is an order of magnitude faster than the naïve insertion sort. Additionally, merge sort outperforms both insertion sort algorithms when inputs are random or reverse sorted. When sorting pre-sorted inputs, insertion sort outperforms merge sort; even the naïve version is better than merge sort in this scenario. These performance differences match the theory and is examined in more depth in the analysis section.

1. Results

The three sorting algorithms’ run times are measured with 72 different input combinations, with varying numbers of input vectors ‘m’, vector dimension ‘n’, and with three input vector conditions: random value, reverse sorted, and sorted. For each of the 72 different input combinations, the sorting algorithm is run through 10 iterations in order to reduce the impact of noise. With all three sorting algorithms, this gives us 2160 data points, averaged to 216 data points. The results of each individual sort algorithm is tabulated below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **vect\_size** | | **10** | | | **25** | | | **50** | | |
| **data\_type** | | **RANDOM** | **REVERSE** | **SORTED** | **RANDOM** | **REVERSE** | **SORTED** | **RANDOM** | **REVERSE** | **SORTED** |
| **num vects** | **1000** | 20.0 | 40.0 | 0.0 | 48.0 | 99.0 | 0.0 | 98.0 | 200.0 | 0.0 |
| **2500** | 133.0 | 274.0 | 0.0 | 361.0 | 678.0 | 0.0 | 757.0 | 1426.0 | 0.0 |
| **5000** | 661.0 | 1252.0 | 0.0 | 1674.0 | 3089.0 | 0.0 | 3515.0 | 6383.0 | 0.0 |
| **10000** | 3066.0 | 5590.0 | 0.0 | 7357.0 | 14823.0 | 0.0 | 16256.0 | 31365.0 | 10.0 |
| **25000** | 24187.0 | 42165.0 | 2.0 | 55826.0 | 95559.0 | 10.0 | 113385.0 | 199266.0 | 13.0 |
| **50000** | 112474.0 | 177178.0 | 0.0 | 253150.0 | 406606.0 | 20.0 | 489211.0 | 836286.0 | 21.0 |
| **100000** | 517687.0 | 683763.0 | 20.0 | 1020830.0 | 1548484.0 | 30.0 | 1853612.0 | 3120282.0 | 50.0 |
| **250000** | 3393640.0 | 4343627.0 | 27.0 | 6583880.0 | 11162846.0 | 64.0 | 13045987.0 | 21366234.0 | 118.0 |

Table 1. Insertion sort time (ms) with varying input conditions.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **vect\_size** | | **10** | | | **25** | | | **50** | | |
| **data\_type** | | **RANDOM** | **REVERSE** | **SORTED** | **RANDOM** | **REVERSE** | **SORTED** | **RANDOM** | **REVERSE** | **SORTED** |
| **num vects** | **1000** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **2500** | 10.0 | 11.0 | 0.0 | 10.0 | 18.0 | 0.0 | 10.0 | 19.0 | 0.0 |
| **5000** | 35.0 | 71.0 | 0.0 | 35.0 | 72.0 | 0.0 | 40.0 | 73.0 | 0.0 |
| **10000** | 151.0 | 334.0 | 0.0 | 158.0 | 337.0 | 0.0 | 189.0 | 400.0 | 0.0 |
| **25000** | 1202.0 | 2540.0 | 0.0 | 1327.0 | 2683.0 | 10.0 | 1374.0 | 2570.0 | 5.0 |
| **50000** | 5253.0 | 10872.0 | 0.0 | 5629.0 | 10906.0 | 10.0 | 4153.0 | 8601.0 | 17.0 |
| **100000** | 21552.0 | 44927.0 | 10.0 | 20790.0 | 42225.0 | 20.0 | 20148.0 | 41169.0 | 30.0 |
| **250000** | 129273.0 | 260834.0 | 10.0 | 129393.0 | 266645.0 | 31.0 | 141707.0 | 287695.0 | 58.0 |

Table 2. Improved insertion sort time (ms) with varying input conditions.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **vect\_size** | | **10** | | | **25** | | | **50** | | |
| **data\_type** | | **RANDOM** | **REVERSE** | **SORTED** | **RANDOM** | **REVERSE** | **SORTED** | **RANDOM** | **REVERSE** | **SORTED** |
| **num vects** | **1000** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **2500** | 0.0 | 0.0 | 0.0 | 7.0 | 3.0 | 0.0 | 11.0 | 10.0 | 10.0 |
| **5000** | 8.0 | 1.0 | 1.0 | 18.0 | 11.0 | 10.0 | 35.0 | 30.0 | 30.0 |
| **10000** | 20.0 | 13.0 | 16.0 | 40.0 | 39.0 | 40.0 | 88.0 | 90.0 | 91.0 |
| **25000** | 55.0 | 50.0 | 50.0 | 135.0 | 129.0 | 132.0 | 237.0 | 248.0 | 237.0 |
| **50000** | 128.0 | 120.0 | 118.0 | 288.0 | 277.0 | 280.0 | 466.0 | 563.0 | 555.0 |
| **100000** | 444.0 | 400.0 | 446.0 | 761.0 | 748.0 | 753.0 | 1023.0 | 1014.0 | 1018.0 |
| **250000** | 1161.0 | 1160.0 | 1137.0 | 1781.0 | 1764.0 | 1746.0 | 3239.0 | 3191.0 | 3183.0 |

Table 3. Merge sort time (ms) with varying input conditions.

In summary, improved insertion sort is faster than the naive insertion sort across the board. Merge sort is distinctly faster than the other two sort algorithms for random and reverse sorted inputs, but not sorted vectors.

1. Analysis

Overall, the results are in agreement with expectations. Insertion sort is expected to be O(n2) for average/worst case. However, in the naïve approach, the cost of the while loop condition check is significantly more expensive, costing 2\*(vector\_length) every single time two values are compared. A plot of the sort times for all three algorithms sorting random length 10 vectors can be seen below (left). A second plot is also shown without the naïve insertion sort, to better compare the scale of improved insertion sort and merge sort (right).

|  |  |
| --- | --- |
|  |  |

Figure 1 (left) Sort time vs varying input lengths for all three algorithms; Figure 1 (right) sort time comparison of merge sort and improved insertion sort.

For random inputs, the merge sort algorithm is exponentially faster. The insertion sort algorithm is shown to be behaving as O(n2) using a log-log plot. This is not a proof of the bound, rather a check to show that our data congruent with the theory. At the same time, we can also show that under best case conditions, the insertion sort algorithm is O(n).

|  |  |
| --- | --- |
| Chart, line chart  Description automatically generated |  |

Figure 2 (left) Log-log plot of improved insertion sort with random inputs, bounded exponentially; Figure 2 (right) Improved insertion sort time of pre-sorted input vectors, bounded linearly.

Next, the values of m (number of vectors) and n (dimension of vectors) and their impact on the run time of the algorithms is examined. First, looking at the naïve insertion sort, the relationship between dimension ‘n’ and run time appears to be linear. For instance, for a fixed m=50,000, increasing dimension n=10 to n=25, or 2.5x, yields a time increase of 2.25x. Increasing again from n=25 to n=15, or 2x, yields a time increase of 1.9x. This highlights the major advantage of the improved insertion sort. While holding ‘m’ constant, increasing the dimension n *yields no appreciable change in runtime* for the improved insertion sort. This linear relationship is shown in Figure 3 below. In this figure, a ‘baseline’ runtime is selected as vector dimension=10. Then for a fixed size number of inputs, the change in runtime is calculated for vector dimension=25 and vector dimension=50. While it is clear that this relationship is linear for the insertion sort algorithms, it is worth noting that the improved insertion sort has negated this effect.  
Chart

Description automatically generated with medium confidence

Figure 3. Insertion sort runtime dependency on vector dimension for naïve sort (red) and improved sort (blue).

Finally, taking a look at merge sort, we see a time dependence on ‘n’ similar to that of the naïve insertion sort. For all three sorting algorithms, the number of inputs ‘m’ makes a significantly larger impact on runtime.

Finally, it is worth noting the dependency on the input data. For pre-sorted inputs, insertion sort significantly outperforms merge sort. This was shown visually above, in Figure 2 (left). With pre-sorted inputs, insertion sort is O(n), while merge sort is 𝛩(nlgn). Thus, there is a crossover point where insertion sort is better than merge sort, depending on ‘how sorted’ the input vectors are. Depending on the use case, insertion sort may be preferable to merge sort. That is, the performance of merge sort is negligibly impacted by the data itself, as it is tightly bound (upper and lower)

There are a number of ways we can improve the performance of the sorting algorithms. A proposed improvement of the insertion algorithm uses a dynamic threshold to reduce the number of comparisons made in every loop (Elshqeirat, Altarawneh, & Aloqaily). Their approach uses a moving reference index, which is equal to the floor of the loop iteration divided by three. In each iteration, the value being inserted is first compared to the threshold value. If it is smaller than the threshold value, it is inserted to the left of the threshold, inserted to the right otherwise.

1. Conclusion

This experiment was successful, in that it gave several key insights into the behavior of these sorting algorithms. First and foremost, the data reflects the theory. The naïve insertion sort and improved insertion sort both, for random inputs, significantly underperformed against merge sort. This reflects the fact that insertion sort is O(n2), while merge sort is 𝛩(nlgn). On top of this, by pre-calculating vector length, the improved insertion sort is an order of magnitude faster than the naïve sort. Additionally, it is shown that the best case scenario of insertion sort, pre-sorted input, significantly outperforms merge sort, as it is O(n). This gives several key takeaways. First, the sorting algorithm selected should be done in context of the data. For instance, data that is expected to be pre-sorted or close to pre-sorted might benefit from an algorithm like insertion sort, even though the average run-time is significantly worse. More generally, and perhaps more importantly, it was shown that an algorithm tightly bounded by a smaller 𝛩 can perform worse than an algorithm that has a larger, upper bound big-O, depending on input data size/shape/composition.

Citations:

1. Elshqeirat, B., Altarawneh, M., & Aloqaily, A. Enhanced Insertion Sort by Threshold Swapping.