**Project 1 – Experimental Analysis of Sorting Algorithms**

Xavier Gschwind

Results

Bubble Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | Random | Descending | Ascending | Runtime Random (ns) | Runtime Descending (ns) | Runtime Ascending  (ns) |
| 1000 | 499500 | 499500 | 499500 | 11,000,900 | 5,997,400 | 998,600 |
| 2000 | 1999000 | 1999000 | 1999000 | 24,000,700 | 21,997,100 | 4,999,600 |
| 4000 | 7998000 | 7998000 | 7998000 | 91,250,800 | 125,996,100 | 20,000,400 |
| 8000 | 31996000 | 31996000 | 31996000 | 396,998,500 | 335,000,300 | 82,999,500 |

The expected performance of Bubble Sort is O(n^2) for worst case, O(n^2) for average case, and O(n) for best case. The data comparisons were the same for each run on a size, but the runtime is similar between random and descending for several cases, descending is the greatest at one point, while the ascending runtime is the smallest.

Insertion Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | Random | Descending | Ascending | Runtime Random (ns) | Runtime Descending (ns) | Runtime Ascending (ns) |
| 1000 | 253367 | 499500 | 999 | 996,500 | 1,990,300 | 0 |
| 2000 | 1039221 | 1999000 | 1999 | 6,003,800 | 5,997,200 | 0 |
| 4000 | 3939393 | 7998000 | 3999 | 25,985,800 | 24,003,300 | 0 |
| 8000 | 16041959 | 31996000 | 7999 | 65,998,600 | 89,995,200 | 0 |

The expected performance of Insertion Sort is O(n^2) for the worst case, O(n^2) for the average case, and O(n) for the best case. The data comparisons are lowest with ascending order, average at random, and highest for descending order. The runtime was larger for descending order compared to random order. Chrono recorded ascending order as either no time or minimal time in every run.

Quick Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | Random | Descending | Ascending | Runtime Random (ns) | Runtime Descending (ns) | Runtime Ascending (ns) |
| 1000 | 15366 | 9984 | 8986 | 1,000,000 | 0 | 0 |
| 2000 | 32931 | 21961 | 19963 | 1,006,000 | 0 | 0 |
| 4000 | 73988 | 47918 | 43923 | 1,007,000 | 0 | 970,600 |
| 8000 | 155980 | 103833 | 95839 | 1,571,100 | 1,995,100 | 524,100 |

The expected performance of Quick Sort is O(n^2) for the worst case, O(nlogn) for the average case, and O(nlogn) for the best case. The data comparisons for quicksort are highest for random, average for descending, and lowest for ascending order. The runtime was similar between runs or recorded as zero by chrono. From what is recorded, the random and descending data had similar runtimes to each other or were similar for any size of the data set.

Heap Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | Random | Descending | Ascending | Runtime Random (ns) | Runtime Descending (ns) | Runtime Ascending (ns) |
| 1000 | 17212 | 16030 | 23461 | 1,009,000 | 1,000,400 | 991,300 |
| 2000 | 38338 | 36099 | 52892 | 993,900 | 998,700 | 1,997,100 |
| 4000 | 84914 | 80152 | 117744 | 3,003,100 | 1,001,000 | 1,993,800 |
| 8000 | 185809 | 176103 | 259432 | 2,996,900 | 1,996,900 | 3,998,600 |

The expected performance of Heap Sort is O(nlog(n)) for the worst, average, and best cases. Heapsort has the most comparisons for ascending, average comparisons for random, and the lowest comparisons for descending order. The runtime was the same or similar for each data set.

3-Way Merge Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size | Random | Descending | Ascending | Runtime Random (ns) | Runtime Descending (ns) | Runtime Ascending (ns) |
| 1000 | 10772 | 6308 | 6187 | 0 | 0 | 0 |
| 2000 | 23942 | 13821 | 13441 | 1,003,400 | 0 | 0 |
| 4000 | 53001 | 30320 | 29673 | 1,013,400 | 991,400 | 992,600 |
| 8000 | 115977 | 65742 | 64237 | 1,995,600 | 1,996,500 | 1,000,600 |

The expected performance of 3-Way Merge Sort is Θ(nlog3(n)) for the average case, O(nlog3(n)) for the worst case, and Ω(nlog3(n)) for the best case. The data comparisons are highest for the random data, average for descending order, and possibly lowest for ascending order. The runtime appears to be higher for random ordered data, and lowest for ascending order. Using chrono, a few runs showed 0 nanoseconds several times.

Analysis

After analyzing all the results, it appears that using chrono for timing some sorting algorithms will not properly give the time. Time was recorded in nanoseconds for each run to ensure that the output would not result in zero, however, the chrono library did not properly time all the sorting algorithms.

The data comparisons on bubble sort happen as they should because every value will get compared. The runtime is as expected for the cases since the average and worst case, being the random and descending order, took longer to compute than the best case of ascending data that doesn’t need to be swapped.

The data comparisons for insertion sort makes less comparisons as the order of the data is already in the correct position as it should, and this best case is represented as O(n) as it should be. The runtime is minimal for ascending and maximized for descending order as it should be between the best the worst cases. The runtime in nanoseconds did not record any time for the ascending cases, which may be due to the fact that it is sorted or the chrono library did not time it right.

Quicksort’s best, average, and worse cases depend on the pivot elements. Initially - for random - n = 1000, pivot=698181; n = 2000, pivot = 16448; n = 4000; pivot = 685587; n = 8000, pivot = 907301. For descending, n = 1000, pivot = 488110; n = 2000, pivot = 502527; n = 4000, pivot = 498479; n = 8000, pivot = 500903. For ascending, n = 1000, pivot = 488024; n = 2000, pivot = 501732; n = 4000; pivot = 499696; n = 8000, pivot = 500736. Looking at n = 1000, the pivot value for random data was the furthest from the median value in the distribution, and this explains the average case. The pivot values for ascending and descending values will tend to be in the middle so those values will normally end up being used for the best cases. The resulting data comparisons match up with what is expected from the given data sets. The runtime was anomalous again when using chrono and several runs yielded 0 runtime for some descending and ascending sorts. The time is similar between runs, judging from the limited data. From what can be gathered, the runtime for a case will be similar between data sets of different sizes.

Heap Sort’s data comparisons were similar for random and descending, and this makes sense since the algorithm runs in nlogn time, however, the ascending data had more comparisons. This may be the result from building the heap with ascending data that caused a higher number of comparisons. The runtime is reflective of the time complexity because the runtimes were all similar in the cases for the inputs.

The 3-way Merge Sort’s data comparisons are similar between descending and ascending ordered data, and the random ordered data will take more comparisons. The random data has more comparisons because the data may take longer to sort through data that isn’t ordered in any way in the subarrays during the merge phase. The descending and ascending ordered data will have a distinct order that will take less comparisons to put it in the right order. Chrono did not give good data for any run of the program so it is difficult to extrapolate any definitive assumptions regarding the runtime. From n = 8000, it can be seen that runtime ascending could possibly will be lower than that of random, which is expected with the data sets after examining the data comparisons.

Based on the n = 8000 data set, Insertion Sort and Quick Sort ran the fastest on already sorted data. Insertion sort yielded 0 ns and Quick Sort yielded 524,100 ns, which are the lowest run times on this data set. Meanwhile, Bubble Sort took the longest with 82,999,500 ns on the sorted data. As for random data, Quick Sort ran the fastest with 1,571,100 ns. With Descending data, Quick Sort, Heap Sort, and 3-Way Merge Sort ran the fastest with similar times near 1,996,000 ns.

Discussion

Chrono allowed for high resolution recording of the execution time with several of the algorithms, however, it failed to properly record data at several points. This resulted in making it more difficult to more definitively analyze and interpret the results. From what can be gathered, Bubble Sort ran as expected in relation to runtime and data comparisons; Insertion Sort ran as expected in relation to runtime and data comparisons; Quick Sort ran as expected in relation to data comparisons, but the runtime was not recorded well; Heap Sort ran as expected with the given data; 3-Way Merge Sort ran as expected for data comparisons, but the runtime was limited to analyze.

Notably, Quick Sort appeared to have similar runtimes for any of the data sets when looking at the random data sets. Additionally, 3-Way Merge Sort’s limited data gives the impression that any form of sorted data will generally have a better run time than that of a random distribution.

When handling pre-sorted data, Insertion Sort or Quick Sort would be best to use, however, at that point there would be no need for the sorting. As for reverse order data, any of Quick Sort, Heap Sort, or 3-Way Merge Sort could be used and yield similar runtimes. If one expects randomly distributed data, then it would be best to use Quick Sort.