Jed Jenner Comia

Prof. Radian

CSC-2400-004

27 October 2023

In this assignment I have implemented shell sort, insertion sort, selection sort and bubble sort. These algorithms sorted three different kinds of arrays each kind at different sizes. The arrays were comprised of a randomly generated array, an increasing array, and a decreasing array. Each type of array had five different sizes 100, 1000, 10000, 100000, and 1000000. Here I will look at its empirical performance.

I will attempt to analyze empirical complexity by testing each algorithm with five replicates for each test. That means each kind of array and size have five data points to normalize the data and have more accurate data. All of this testing and empirical analysis is to test each algorithm’s actual performance when faced with smaller set sizes to very large, when their sets are organized ascending, descending, or completely random. I have timed each algorithm in nanoseconds. I did not have access to another device other than my laptop for the amount of time I needed for certain algorithms. Some algorithms also never produced values in microseconds, so I had to have the algorithms timed in nanoseconds. I also considered converting these results into microseconds or even seconds. I personally thought that converting the results into seconds wasn’t as helpful as one would think. Consequently, I kept the data in nanoseconds. To aid in reading the data in nanoseconds, they will be written in scientific notation. Each algorithm also has a table for the averages of that algorithm’s execution time for random arrays, increasing arrays, decreasing arrays, and the average of them all in an overall table.

**Shell Sort**

In Big-O, shell sort at is best performs at O(n(log(n))). At its worst, shell sort performs at O(n(log(n))^2). In shell sort gaps are used to divide the array into smaller subarrays. In text, shell sort is always compared to being better than insertion sort. In all of these shell sorting tables its seen that the factor increase is higher between the first few sizes but then falls off. Especially when comparing the 10^4 to the 10^5 run for all the shell sorting tests it can be seen that there is a significant drop in factor increase. The algorithm seems to follow that same trend of factor increase from then on. This drop is the most extreme in the decreasing array test for shell sort when the factor of increase goes from about 12 to 4. For the beginning of the algorithm, it grows by a factor of around 12-15, towards the end it tapers off anywhere from about 4-9. Considering there are 5 replicates, I wouldn’t think the spread would be as high.

Shell Sort Execution Time (Random)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | 26325.2 |  |
| 10^3 | 411849.8 | |  | | --- | | 15.64 | |
| 10^4 | 5996650 | |  | | --- | | 14.56 | |
| 10^5 | 49914599.8 | |  | | --- | | 8.32 | |
| 10^6 | 407131874.8 | |  | | --- | | 8.16 | |

Shell Sort Execution Time (Increasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 8766.6 | | |  | | --- | |  | |
| 10^3 | |  | | --- | | 125700.2 | | |  | | --- | | 14.33 | |
| 10^4 | |  | | --- | | 1545783.4 | | |  | | --- | | 12.30 | |
| 10^5 | |  | | --- | | 7992400 | |  | | |  | | --- | | 5.17 | |
| 10^6 | |  | | --- | | 78116458.2 | | |  | | --- | | 9.77 | |

Shell Sort Execution Time (Decreasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 8555.4 | |  | |  |
| 10^3 | |  | | --- | | 125624.8 | | 14.68 |
| 10^4 | |  | | --- | | 1553724.8 | | 12.37 |
| 10^5 | 7472216.8 | |  | | --- | | 4.81 | |
| 10^6 | |  | | --- | | 78170500 | | 10.46 |

Shell Sort Execution Time Overall

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 14549.06667 | |  |
| 10^3 | |  | | --- | | 221058.2667 | | |  | | --- | | 15.19 | |
| 10^4 | |  | | --- | | 3032052.733 | | 13.72 |
| 10^5 | |  | | --- | | 21793072.2 | | |  | | --- | | 7.19 | |
| 10^6 | |  | | --- | | 187806277.7 | | |  | | --- | | 8.62 | |

**Insertion Sort**

In Big-O, insertion sort at is best performs at O(n). At its worst, insertion sort performs at O(n^2). Insertion sort works by dividing the list into an unsorted and a sorted list. The algorithm goes through the elements and places it in the correct spot of the sorted list. Insertion sort did horrible with random arrays. The factor of increase stayed around the 80-100 factor of increase range. There were some outliers like in the overall array there was an increase of 75 and in the completely random array to a million there was a factor of increase of 129. On the other hand, the ascending and descending arrays stayed around an increase factor of 8-10. This algorithm struggled with random arrays. In the random arrays and the overall table there was a slight pattern of a smaller factor of increase to a bigger factor of increase and then it would alternate.

Insertion Sort Execution Time (Random)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 10991.4 | |  |
| 10^3 | |  | | --- | | 900808.2 | | |  | | --- | | 81.96 | |
| 10^4 | |  | | --- | | 90250166.6 | | |  | | --- | | 100.19 | |
| 10^5 | |  | | --- | | 7394783692 | | |  | | --- | | 81.94 | |
| 10^6 | |  | | --- | | 9.5497E+11 | | |  | | --- | | 129.14 | |

Insertion Sort Execution Time (Increasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 516.6 | | |  | | --- | |  | |
| 10^3 | 4666.4 | |  | | --- | | 9.03 | |
| 10^4 | |  | | --- | | 45433.2 | | |  | | --- | | 9.74 | |
| 10^5 | |  | | --- | | 393158.4 | | |  | | --- | | 8.65 | |
| 10^6 | |  | | --- | | 4625191.8 | | |  | | --- | | 11.76 | |

Insertion Sort Execution Time (Decreasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 491.8 | | |  | | --- | |  | |
| 10^3 | |  | | --- | | 4316.6 | | 8.78 |
| 10^4 | |  | | --- | | 44966.4 | | 10.42 |
| 10^5 | |  | | --- | | 368766.4 | | |  | | --- | | 8.20 | |
| 10^6 | |  | | --- | | 4398712.6 | | 11.93 |

Insertion Sort Execution Time Overall

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 3999.933333 | |  |
| 10^3 | |  | | --- | | 303263.7333 | | 75.82 |
| 10^4 | |  | | --- | | 30113522.07 | | 99.30 |
| 10^5 | |  | | --- | |  | | 2465181872 | |  | | 81.86 |
| 10^6 | |  | | --- | | 3.18326E+11 | | 129.13 |

**Selection Sort**

In Big-O, selection sort at is best performs at O(n^2). At its worst, shell sort performs at O(n^2). Selection sort works by either picking the maximum or minimum element from an unsorted array and swapping it to its respective position. Usually, the minimum is found by iterating through the array and then swapping until the array is sorted. There is a very particular pattern with these results in factor increase. For a majority of the tests, the factor of increase would increase at a decent amount as the size of the problem increased. But, on the 10^6 it can be seen that the factor increase drops off just slightly. The only test this is not present in is the first random test. This is the first algorithm that has had a pretty consistent factor of increase. This means that the factor of increase is almost basically uniform from size to size no matter the type of array.

Selection Sort Execution Time (Random)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 20083.6 | |  | |  |
| 10^3 | |  | | --- | | 1526316.8 | | |  | | --- | | 76.00 | |
| 10^4 | 144257238.2 | |  | | --- | | 94.51 | |
| 10^5 | 14276789933 | |  | | --- | | 98.97 | |
| 10^6 | |  | | --- | | 1.43022E+12 | | |  | | --- | | 100.18 | |

Selection Sort Execution Time (Increasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 15566.8 | |  |
| 10^3 | |  | | --- | | 1463758.2 | | 94.03 |
| 10^4 | |  | | --- | | 142945025 | | 97.66 |
| 10^5 | |  | | --- | | 14262719500 | | |  | | --- | | 99.78 | |
| 10^6 | |  | | --- | | 1.41393E+12 | | 99.13 |

Selection Sort Execution Time (Decreasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 15533.2 | |  |
| 10^3 | 1443850.2 | 92.95 |
| 10^4 | |  | | --- | | 142924391.8 | | 98.99 |
| 10^5 | 14286351909 | |  | | --- | | 99.96 | |
| 10^6 | 1.40879E+12 | 98.61 |

Selection Sort Execution Time Overall

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 17061.2 | |  |
| 10^3 | 1477975.067 | 86.63 |
| 10^4 | |  | | --- | | 143375551.7 | | 97.01 |
| 10^5 | 14275287114 | 99.57 |
| 10^6 | |  | | --- | | 1.41765E+12 | | |  | | --- | | 99.31 | |

**Bubble Sort**

In Big-O, bubble sort at is best performs at O(n). At its worst, shell sort performs at O(n^2). Bubble sort is very straight forward when it comes to sorting algorithms. It compares elements that are adjacent to each other and swaps them if the first element is larger than the second element. I had expectations for this algorithm to take the longest for random arrays. This algorithm had huge numbers for random array execution time. As for increasing and decreasing arrays they only increased by around a factor of 10 each time. For random arrays, they increased at about factors of 100 each time the size of the problem increased.

Bubble Sort Execution Time (Random)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 33291.4 | |  |
| 10^3 | |  | | --- | | 3151923.8 | | 94.68 |
| 10^4 | 364911722.8 | |  | | --- | | 115.77 | |
| 10^5 | 36979950552 | 101.34 |
| 10^6 | |  | | --- | | 3.66166E+12 | | |  | | --- | | 99.02 | |

Bubble Sort Execution Time (Increasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 292 | |  |
| 10^3 | |  | | --- | | 2739.6 | | |  | | --- | | 9.38 | |
| 10^4 | |  | | --- | | 30065.8 | | |  | | --- | | 10.97 | |
| 10^5 | |  | | --- | | 271308.6 | | |  | | --- | | 9.02 | |
| 10^6 | |  | | --- | | 2725050.2 | | |  | | --- | | 10.04 | |

Bubble Sort Execution Time (Decreasing)

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 368.8 | | |  | | --- | |  | |
| 10^3 | 2916.6 | 7.91 |
| 10^4 | 27401.6 | 9.40 |
| 10^5 | 267796.4 | 9.77 |
| 10^6 | 2897685.4 | 10.82 |

Bubble Sort Execution Time Overall

|  |  |  |
| --- | --- | --- |
| Size of n | Avg Time to complete (nano) | Factor increase (nano) |
| 10^2 | |  | | --- | | 11317.4 | |  |
| 10^3 | 1052526.667 | 93.00 |
| 10^4 | 121656396.7 | 115.59 |
| 10^5 | |  | | --- | | 10565892695 | | 86.85 |
| 10^6 | |  | | --- | | 1.04619E+12 | | 99.02 |

**Graphs**

All of these graphs are based on the overall cahrt for each algorithm. Conclusion will be made based off of the tables and graphs.This graph turned out the best compared to the rest of the graphs. This graph looks very similar to what the shell sorting algorithm looks like if you were to look it up on Google. On average this algorithm did the third best for size 100, the best for sizes 10^3-10^6. It was literally the fastest algorithim at almost all sizes.

This graph was one of the dissapointing ones because there was not enough space to straighten it out. The magnitude was too varying for the graph to see where it really changes. This Is simialr for the next two graphs. This algorithm did the best with size 100 and was the second fastest from sizes 10^3-10^6.

Selection sort was the worst for most arrays across the whole board of sizes. It was the slowest algorithm. It performed the slowest in almost about every category.

Bubble sort was the second fastest for the size 100, was the third best for size 10^3-10^6.

**Conclusion**

Shell sort was the clear victor all around for these sotring algorithms. It was literally the best in almost very single category based on the average results. I think describing the insertion sort as a worse form of shell sort was very accurate because it was always second to shell sort for most of the averages. I also think insertion sort had some more funky numbers because when it came to ascending and descending arrays, the results were always significantly smaller than the random results. Selection was definitely the worst out of all the algorithms. It was just slow compared to them at all levels. Bubble sort definitely stuggled when it came to random arrays at well but it made up for it with increasing and decreasing arrays. Those numbers most likely skewed the results towards bubble sort being faster, even though it was actually slower on average when it came to random of size 10^6 by a large margin. Bubble sort just did very well with ascending ans descending arrays. The mathmatical complexity pretty much reflected each algorithm correctly. Shell sort was the fastest, which matched its mathemtical complexity as stated above. Insertion sort was slower as stated by its mathematical complexity but not as slow as selection sort, which proved to be accurate. The only curious result was bubble sort. I think this was because I presented my data with averages of the random, increasing, and decreasing. I did that to make the data more fair but, it couldve skewed it to be more in favor of bubble sort. In the raw data it was clear to see bubble sort did worse with larger arrays when they were random. I think this program proved that not algorithms are not built equal. It also proves that the topics in class have applications in the real world that we can experience from our computers. The mathematical expression that we read in class like big-o notation is relevant, this is proven through the data. This programming assignment connects real-world computations, algorithms, and analysis of runtime and complexity perfectly.