Benjamin Byerly

CSC 2400-004

Cristina Radian

10/20/23

Analysis of Program 2

Below is a table and graph for each algorithm, it has the size of n, the time to complete the algorithm, and the factor increase between each size of n. One thing to be noted is that for some of the algorithms the size of n will be different. This is because on my computer the sorting algorithm for some algorithms completed to fast before the clock was able to record time. So, I increased the size of the arrays to provide accurate data the be able to study and record. The algorithms that will start at a size of n at 4 will be shell sort. Selection sort, insertion sort, bubble sort will remain at a size of n at 2-6­­.

This paper will be outlined like this. With the graph at the beginning of each algorithm. The tables will be under each array type with their explanation and analysis under each table.

1. Shell sort
   1. Random array
      1. Analysis
   2. Increasing array
      1. Analysis
   3. Decreasing array
      1. Analysis
2. Etc.

**For context, I ran each algorithm on its own to get the most accurate results. I ran into some issues with running all the algorithms back to back. I decided to break it up and go from there.**

Shell Sort **RANDOM** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 10,000 | .001333 |  |
| 100,000 | .011111 | 8.34 |
| 1,000,000 | .158667 | 14.28 |
| 10,000,000 | 3.37133 | 21.25 |
| 100,000,000 | 88.0893 | 26.13 |

**Random array:**

For the first table and the blue line on the graph, the rate of increase in time is not entirely consistent, but it becomes more noticable as n becomes larger. This follows the behavior of an average-case or best-case time complexity of Θ (nlog(n))) or O(n^2). This suggests that the time taken can grow quadratically with with input size. However, since shell sort is an adaptive sorting algorithm, meaning that this algorithm takes advantage of existing order in the data to optimize its performance. Which is why this graph seems to follow a more linear path at the beginning. What we can infer from this graph is that the random array was at an average case scenario for some of the arrays, meaning that it some of the values were already partially sorted. And since the array with 10^8 took around 90 seconds to complete we can assume that these arrays are at a worst-case scenario. So this graph follows a linear path at the earlier arrays because the average-case: Θ (n log(n)). For last iteration of the array, it is safe to say that it is close to its worst-case: O(n^2).

Shell Sort **INCREASING** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 10,000 | .002333 |  |
| 100,000 | .017333 | 5.71 |
| 1,000,000 | .139 | 8.02 |
| 10,000,000 | 1.663 | 11.96 |
| 100,000,000 | 28.9403 | 17.72 |

**Increasing array:**

For this table and orange line on the graph, I recorded the completion time of an incresaing array of size n increasing to 10^8. Since these arrays are already sorted we are at a best-case scenario. As you can see the graph tends to follow a more linear path then it climbs up when it gets to the n value of 10^8. This is because the time complexity of the shell sort algorithm at base-case is Ω(nlog(n)). Since we are at a base-case I expected the growth of the graph to be quadratic, where the time to sort the algorithm increases quadratically as the input size grows.

Shell Sort **DECREASING** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 10,000 | .00067 |  |
| 100,000 | .006 | 8.96 |
| 1,000,000 | .11233 | 12.05 |
| 10,000,000 | 1.706 | 15.187 |
| 100,000,000 | 31.0813 | 18.21 |

**Decreasing array:**

This table and graph can be confusing when looking at the decreasing graph, the random graph, and their completion times. On the largest value of n the random array took much longer to sort than the decreasing array’s larget value of n. This could be for a number of reasons such as internet connection, memory space issues, and the gap sequence with the shell sort, etc. However, even though we are working with a decreasing array, this may not mean we are at a worst-case scenario. This is because the way that shell sort works. It compares values that are far away from each other and sorts those. So in a decreasing array the values are farther apart from each other the bigger the array is. Which could allow the array to get sorted faster than if those same values were right next to each other. What we can conclude from this is that we are at a very similar time complexity to the increasing array. By definition the decreasing array should be an average-case or worst-case. It will never be a best-case because we are swapping values. There is a low possibility of it being worst-case but it is still possible. However, for this graph, it follows the trend line of the best-case the closest. So it is safe to say that it has a average time complexity of nlog(n).

**Insertion Sort:**

Below is a graph and tables for the **Insertion** sort algorithm. For this algorithm, I had to start at an n size of 2 in order for my program to finish within the decade. The time to complete the algorithm starts off slow then it jumps up significantly when it gets to an n size of 6.

Insertion sort **RANDOM** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.00333 | 0 |
| 10,000 | 1.22167 | 366.87 |
| 100,000 | 8.59033 | 7.03 |
| 1,000,000 | 155.377 | 18.09 |

**Random array:**

The results that I got from the insertion sort algorithm demonstrate its mathematical time complexity of Θ (n^2). For the random array, with an initial size of 100, the algorithm finished in an insignificant amount of time. As the array increased to 1,000, the time it took to complete is still small which indicated a linear increase in time. Keep in mind that these are different arrays each time, so we will get different time complexities for each 10^n value. However, the most noticable jump by the time facotr is between the 1,000 and 10,000 values. This instance reflects the quadratic nature of the insertion algorithm. This would fall under an average-case scenario which is Θ(n^2). As the array increases from 100,000 to 1,000,000 the time quadratic time complexity becomes much more evident from looking at the graph. From looking at the graph we can confidently assume that we are working with an average case scenario here. If you compare them to the increasing and decreasing lines, we are pretty much in the center of them. In summary, my analysis of this algorithm concludes that the results align with the expected mathematical time complexity of the insertion sort algorithm.

Insertion sort **INCRESASING** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0 | 0 |
| 10,000 | 0.000067 | 0.000067 |
| 100,000 | 0.000967 | 14.432 |
| 1,000,000 | 0.026667 | 27.58 |

**Increasing array:**

The results that I got from using the insertion sort algorithm on an increasing array are to be expected. It correlates well with its mathematical time complexity. Since this algorithm is already sorted we are at a best-case scenario. For the first two iteration the time to complete was at an insignicant amount of time to be able to record. When the array grew to a size of 10,000 we are finally able to record the runtime of the algorithm. At the array size of 10,000 we are at a runtime of .000067 seconds. Which this solidified my analysis that the increasing array has a best-case time complexity Ω(n). This is because the exectution time of the algorithm scales linearly with the input size. My analysis is further supported by looking at the time completion at the values of 100,000 and 1,000,000. The time to complete the algorithm stayed small, this shows that even with large inputs thanks to its linear time complexity it maintains efficienct when searching the algorithm.

Insertion sort **DECREASING** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.03333 | 0.000333 |
| 10,000 | 2.715 | 81.46 |
| 100,000 | 25.09433 | 9.24 |
| 1,000,000 | 315.955 | 12.6 |

**Decreasing array:**

The results that I got from analyzing the insertion sort algorithm when working with a decreasing array is to be expected when comparing it to its expected mathematical time complexity. At the lower values or 100 and 1,000 the time to complete the algorithm remained low. However, once you look at change from the array size of 1,000 to 10,000 we can see that the execution time drastically jumps to a much larger factor. This indicated that although insertion sort is less effiecient with a decreasing array compared to an increasing array, it is relatively effiecient when working with smaller arrays. Now if we look at the large jump from in completeion time from the 100,000 size array to the 1,000,000, we can solidifiy our assumption from the graph that when faced with a decreasing array. The graph shows much worse performance on the 1,000,000 size array than a increasing array or a random array did. The results from my data and analysis shows that we are working with worst-case scenarios when using decreasing arrays. In summary, the results align with the mathematical time complexity of the insertion sort algorithm. The time complexity of this algorithm when working with a decreasing array is O(n^2).

**Selection Sort:**

Below is a graph and tables for selection sort algorithm­. For this algorithm, I had to start at an n size of two so that my algorithm would finish in a relatively similar time to the shell sort and insertion sort. As you can see, the time to sort/search the array drastically jumps up when it goes up to the array size of 1,000,000.

Selection sort **RANDOM** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.001 | 0.001 |
| 10,000 | 0.053 | 53 |
| 100,000 | 5.199 | 98.09 |
| 1,000,000 | 666.329 | 128.16 |

**Random array:**

The selection sort section of these loops will be relatively short. This is because in the selection sort algorithm no matter what you are sorting, you will always have a time complexity of n^2. The results that I got from this analysis align well with its expected mathematical time complexity which is Θ (n^2). For the array element size of 100 all the way up to 10,000, the time to complete the sorting algorithm was below one second. And the factor of each of the iterations grow quadratically. The reason each of the graphs are so close to each other on time completion is because of the way the algorithm is written. In a selection sort algorithm the algorithm will search the entire array for the next smallest value, which is why the time complexity is always n^2, for the best-case, worst-case, and average-case scenarios. Inside these arrays there will be a random number of swaps which is why the graphs line is in between the other two arrays lines. These random arrays would be classified as an average-case scenario. This is because it is between the decreasing array and the increasing array. The time complexity of these array would be at Θ (n^2).

Selection sort **INCREASING** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.001 | 0.001 |
| 10,000 | 0.051 | 51 |
| 100,000 | 5.907 | 115.82 |
| 1,000,000 | 662.264 | 112.16 |

**Increasing array:**

The results from this analysis of sorting an increasing array using seleciton sort are to be expected. For this analysis we are looking at the orange line in the graph, which is just below the blue line(the random array). Since the orange line is below the gray line and the blue line we can conlcude that we are working with a best-caes scenario. Even though it is still a time complexity of n^2, there will be no swaps within the sorting process. This will save time and cause the increasing array to be just below the random array which is an average case. Our time complexity for these arrays would be Ω(n^2).

Selection sort **DECREASING** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.001 | 0.001 |
| 10,000 | 0.054 | 54 |
| 100,000 | 7.051 | 130.57 |
| 1,000,000 | 700.434 | 99.34 |

**Decreasing array:**

Just like with the other two sets of arrays, my analysis shows that these arrays have a time complexity of n^2. The graph shows this decreasing array’s line (gray line) is above the other two lines. Since we are working with a decreasing array using a selection sort algorithm we will be at a worst-case scenario of O(n^2). Each time we compare values in the array we are also making a swap. This causes the time of completion to increase, even though the most time consuming part about this algorithm are the comparisons in each iteration. In summary, the results from myy analysis closely align with the mathematical time complexity of the selection sort algorithm.

**Bubble Sort:**

Below is a graph and tables for the bubble sort algorithm. For this algorithm I had to start at an n size of 2 and increase it up to an n size of 6. I did this because the bubble sort algorithm has an average case of n^2, so I knew that the algorithm would run for a very long time with higher values.

Bubble sort **Random** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.001 | 0.001 |
| 10,000 | 0.2 | 53 |
| 100,000 | 29.363 | 98.09 |
| 1,000,000 | 3367.41 | 128.16 |

**Random array:**

When I first looked at this grpah I thought that there was some sort of error. But then I studied the algorithm and remembered that bubble sort compares two values at every iteration and if the left one is greater than the right one than it swaps them. I then came to the conlcusion that this graph was correct and it follows the expected mathematical time complexity of the bubble sort algorithm. In this case, a randomized array will have several swaps because very few elements will be close to their desired position. But if you compare that to the decreasing array there are quiete a few elements that are towards the center of the array that are close to their position that don’t need that many swaps. However, there is a small chance that the random array can have a better time completion than the decreasing array. Which is why this conclusion is to be expected when you study the graph and its data. The time complexity for these arrays would be between a worst-case(O(n^2)) scenario and an average-case (Θ (n^2)) scenario.

Bubble sort **Increasing** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0 | 0.001 |
| 10,000 | 0 | 51 |
| 100,000 | 0 | 115.82 |
| 1,000,000 | 0.005 | 112.16 |

**Increasing array:**

The results from this data are to be expected for a best-case scenario. It correlates perfeclty well with the mathematical time complexity of a best-case scenario bubble sort array. Since the increasing array is already sorted the array just compares the values and does no actualy swapping. The execution time of the algorithm scales linearly with the input size. Which is shown by looking at the runtime of the algorithm and arrays. The algorithms runs so fast when searching for the values of n from 2-5 that it cant record an actual runtime until the array size is 1,000,000. The time to complete the algorithm stayed really even with larger inputs, this is because of its linearity.

Bubble sort **Decreasing** array:

|  |  |  |
| --- | --- | --- |
| Size of n | Time to complete in (s) | The factor increase in time |
| 100 | 0 |  |
| 1,000 | 0.002 | 0.001 |
| 10,000 | 0.224 | 54 |
| 100,000 | 29.615 | 130.57 |
| 1,000,000 | 2959.12 | 99.61 |

**Decreasing arry:**

The decresaing array for this algorithm has an average time complexity of Θ(n^2). As you can see from the graphs, the decreasing array is in between the increasing array and the random array. The decreasing array is at an average case, because of how the algorithm sorts its values. When sorting the decreasing array, there are several values inside the arrays that don’t need to be moved very far to get to their desired position. Since they are doing less swaps, this makes the run time lower than the random arrays. And since the array with 10^8 took around 3,000 seconds to complete, we can assume that these arrays are at a average-case scenario. The growth of the time to complete sorting these arrays are exponential. In summary, the algorithm exhibits the expected quadratic growth in execution time.

In conclusion, the choice of a sorting algorithm should be based on the characteristics of the input data and the desired performance. For small or nearly sorted datasets, bubble and insertion sort can be suitable due to their linear time complexity in the best case. However, for general-purpose sorting, shell sort offers a good balance of adaptability and efficiency. Other advanced sorting algorithms, such as merge sort, quicksort, and heap sort, should also be considered for optimal performance, especially with larger datasets. The empirical results presented here provide valuable insights into how these algorithms behave under different conditions and can guide the selection of the most appropriate sorting method for a given application.

**BONUS : For the bonus, I used the new updated code the outputs were very similar to graphs in the original analysis**

**For these new datasets, I created new arrays for every single algorithm.**

**Graph Details for Array Size 10^4, Random Array: Below is a graph of the 4 runtimes of the random array and the source code I used to make the graph. The data points are the exact runtime I got for the new updated code**

**A screenshot of a computer

Description automatically generated**

1. **Shell Sort:**

* From the given graph of the random array of size 10^4, Shell sort seems to work the best.
* The average time of the shell sort algorithm was 0.0166667
* Which this indicates that shell sort’s ability to handle the random array of size 10^4 very efficiently.

2. **Insertion Sort:**

* The insertion sort algorithm signifies a much larger jump in the time to comlete sorting the array.
* Insertion sort is a more simple sorting algorithm, insertion sort tends to be better at sorting smaller datasets.
* The average time of the insertion sort algorithm was 1.094667.

3. **Selection Sort:**

* Selection sort’s time complexity is O(n^2) making it less suitable for larger data sets.
* Selection sort has the same time complextiy of insertion sort but there is the possibility that the array for selection sort was more out of order than insertion sort.
* The time to complete sorting this array was 0.590667

4. **Bubble Sort:**

* Bubble sort had the second best time to complete sorting the random array.
* This was not expected to be this different from the other three algorithms, but since I created a new array for the algorithm to sort each time it makes sense that my datasets would be more sporadic.

In conclusion, deciding between choosing a sorting algorithm should be chosen based off of the of the input data. While bubble sort and selection sort seem to be in the middle of with a more average time to complete, shell sort comes out on top for the clear winner. The worst algorithm to choose for this data size would be insertion sort since it had the worst average time to compelte than the other three algorithms.