Connor Yu

Writeup

Insertion:

When the insertion algorithm is given an array the first step it takes to sort it is designate the first cell at index zero and its corresponding value as sorted. The sorted section should start out as only one cell and value, but as the algorithm runs through the array the sorted section will grow. As the algorithm runs, the sorted section will grow with the highest values being in the highest sorted cells, or to the right near the end of the sorted section. The lowest sorted values, conversely, should be at the start of the sorted section. The sorted section thus borders the unsorted section, with the unsorted cells starting after the end of the sorted section.

The program will look at the first cell in the unsorted section, and compare it to the cell that's next to it, the one that points to the highest value of the sorted section. If that cell has a value higher than the one unsorted cell's value the two cells swap values. The algorithm then repeats this and compares the unsorted value to the next cell and its value, swapping it if the unsorted cell has a lower value. To clarify, the cells are the place where the value is stored, or in python, it's where a reference to the value is stored. This continues on until the value that is being sorted finds a value that's lower than it. If that does happen, the cell that originally contained the unsorted value is officially sorted, the value stays where it is, and the program goes on to the next value in the unsorted section of the array.

This goes on for all subsequent cells and values until the unsorted section no longer exists, with every unsorted value being swapped with all values that are higher than it until it finds its correct place in the array. An exception to this would be if the cell that's being sorted is lower than all of the values in the sorted section, in which case the algorithm would place it at the beginning of the array, though the algorithm would have to compare it to every value in the sorted section, which would take a relatively long amount of time.

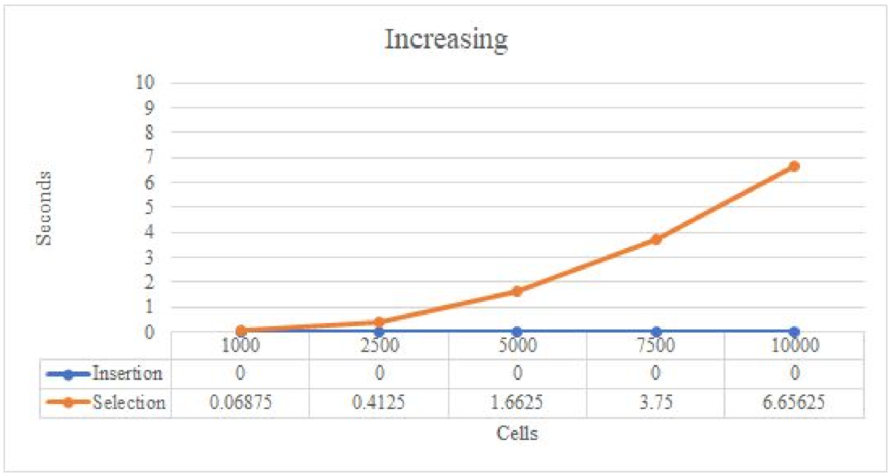
The longer it takes to find a value lower than the value we're sorting, thus stopping the sorting for that value, the longer it takes to sort the entire array. This means there are two things that take up time in the insertion algorithm, comparing the values and swapping the values if the unsorted value is lower. This is, however, the only process in the algorithm that takes up time, so if values don't have to be swapped many times through the array then the algorithm is able to sort the function very fast.

Selection:

The selection algorithm, on the other hand, looks at the cell and figures out which value should be in that cell. Then, once the algorithm finds the cell that contains said value, swaps the values of the two cells. It does this by taking the first value in the array at index 0 and swapping it with the lowest value in the entire array. After that, it goes to the next value at index 1 and swaps it with the smallest value in the array after and including the value at index 1. In other words, it searches from itself to the last value in the array, not including the values that have already been sorted. This continues on for all subsequent cells until it reaches the end of the array. This algorithm took three variables to implement.

- First off, k is the cell that's being sorted, which contains the value that will be swapped if it's not the lowest value in the array we search.
- Next, location will store the number of the cell that has the lowest found value, though initially it will be designated as k, in case k does have the lowest value. So location basically points to the cell with the lowest value that has been found.
- Finally, flipper will be used to make sure we check the value of every cell after k, and if the value at the cell flipper points to is found to be lower than the value at the cell the location variable points to, location will point to flipper's index point instead. Flipper starts out one higher than the value k, as we will check all cells after k. Flipper basically points to the location that we're currently looking at to see if it contains the lowest value.

A while loop will compare the value at k with the next value in the array, with flipper as its index point. If the value is lower than the value at k the location variable will point to that spot instead of k. This will continue on until they've looked through all of the points in the array with an index greater than k. Then, the value of the cell that location points to will swap with the value of the kth cell. The algorithm will then go on to the next unsorted cell in the array, reassign variables k, flipper, and location, and will continue on until all cells are sorted.



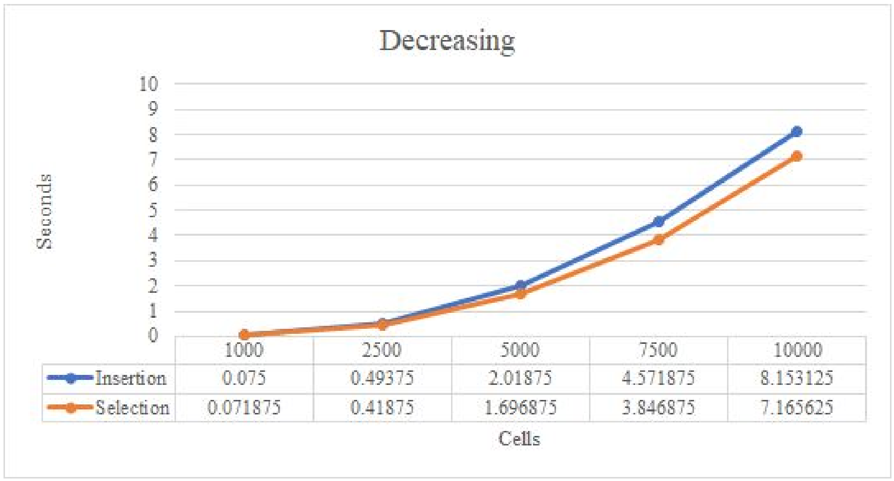
	1000	2500	5000	7500	10000
Insertion	0	0	0	0	0
Selection	0.06875	0.4125	1.6625	3.75	6.65625

Increasing:

An increasing array is sort of the best case scenario for these two arrays, showing the fastest time that these two algorithms can work at. The insertion algorithm is incredibly fast, coming out almost instantaneously. The selection algorithm, on the other, processes the array in around the same amount of time that it takes for it to process the decreasing and random array, taking much longer than the insertion algorithm.

The insertion algorithm is so much faster than the selection algorithm because of the way it processes the unsorted section. It starts out with the lowest value in the array, designating it as sorted, and compares it to the first section of the unsorted array, which is higher than the initial value, so it's already sorted. This goes on and on, only comparing the unsorted cell to the value next to it to make sure it's sorted, and then it can move on.

The selection algorithm, on the other hand, doesn't know for sure if the value it's pointing at is sorted until it has checked through all subsequent values in the array. While it doesn't have to swap values after every time it's found to be unsorted like the insertion algorithm, it has to check it against many more values than the insertion algorithm does. However, this is still a best case scenario for the algorithm, as it doesn't have to swap the values that point to the lowest found value in the array, as when the function runs a cell through the algorithm it's already sorted. Though in the end, this doesn't make too much of a difference.



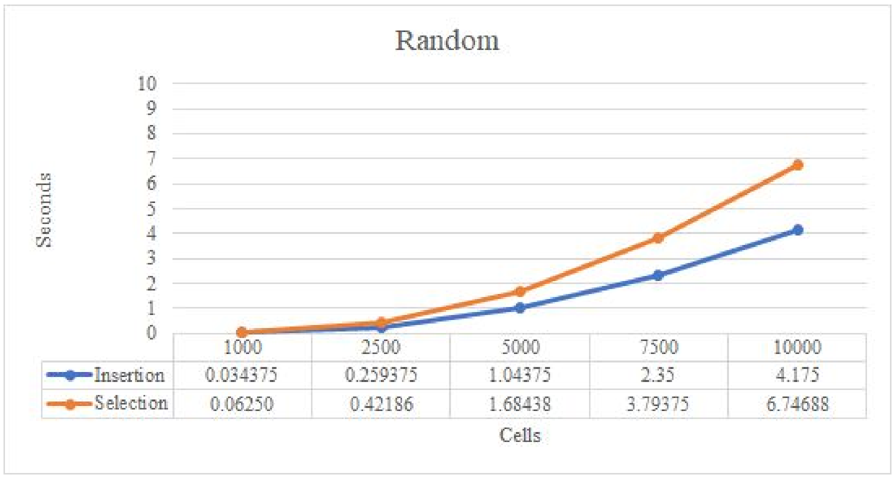
	1000	2500	5000	7500	10000
Insertion	0.075	0.49375	2.01875	4.571875	8.153125
Selection	0.071875	0.41875	1.696875	3.846875	7.165625

Decreasing:

The decreasing array, on the other hand, is sort of a worst case scenario for both algorithms. The selection algorithm is actually faster than the insertion algorithm, though not that much faster. Selection takes around the same amount of time as the increasing algorithm, only taking a little bit more time to sort through the decreasing array. The increase in time most likely has to do with rewriting the location variable so many times, though it doesn't increase the time taken an incredible amount. As the number of cells increase past 10,000 the difference between the time taken by the insertion algorithm and the selection algorithm will only grow, which is evidenced by the increasing difference between the two when comparing the 1,000 cell array to the 10,000 cell array.

The insertion algorithm takes so long because it takes longer for this algorithm to sort the more time it takes for any given cell to be sorted. For the increasing array, the algorithm only had to compare each cell to the one next to it, and the cell that was being sorted would always be higher than the one in the sorted section, so it could move on almost instantly. The decreasing array, however, has to compare and swap every unsorted value through the entire sorted section of the array, making the algorithm overall more inefficient compared to the selection algorithm.

For every array, the selection algorithm always has to check every value after the cell that's being sorted, to look for the lowest value. Compared to the increasing array, the only difference is that the array is always decreasing, meaning that every value after the cell that's being sorted is lower than it. Because of this, the algorithm has to rewrite the location variable to point to a different cell every time its values is compared against another cell's value, as the cell the location variable is pointing to will always be higher than the one it's being compared to. However, this rewriting of the location variable doesn't take too much time and it only takes slightly more time sorting the decreasing array than the increasing array. The selection algorithm takes less time because it looks through the entire array, rewriting the location variable, but the insertion algorithm looks through the entire array and swaps the value it's sorting with every single value in the sorted section of the array.



	1000	2500	5000	7500	10000
Insertion	0.034375	0.259375	1.04375	2.35	4.175
Selection	0.06250	0.42186	1.68438	3.79375	6.74688

Random:

The time it takes for the algorithms to sort the random array falls between each algorithm's time it takes to sort the increasing and decreasing array. This makes sense, as the increasing array is the best case scenario for both of the programs, and the decreasing array is the worst case. The algorithms will generally have to do more work to sort the random arrays than the increasing array and less work to sort the random arrays than the decreasing array. It is possible for the random algorithm to be completely sorted in increasing or decreasing values, but the odds of that are essentially zero, especially as you add more cells.

The insertion algorithm will generally take less time than the selection algorithm because it won't have to iterate over the entire list every single time. An interesting point to note is that despite the array being completely random, there is very little deviance from the mean for each run's timing, which we can see in the data tables in the appendix. This could be because the time it takes for each individual sorted cell will eventually average out, leading to around the same time for each run through of the program.

The selection algorithm will generally take longer than the insertion algorithm. The time the selection algorithm takes is always very similar, the order of the array never makes too much of a difference. The increasing and decreasing array, which should be the most efficient and inefficient arrays, only differ by .418745 seconds for the sorting of the 10,000 cell array. This difference is only caused by the rewriting of the location variable, as the algorithm always flips through the entire array anyway. The random array's timing falls within this range, deviating depending on the number of times we swap the location variable when we find a lower value.

Conclusion:

When the insertion sorting algorithm is passed an array that's already sorted or close to almost sorted it's able to process the array incredibly fast. In the worst case scenarios, however, when the algorithm is passed an array that's ordered the opposite way the algorithm is sorting, it takes longer than the selection algorithm, though not by much. The selection algorithm seems to take around the same amount of time no matter the ordering of the array. Both algorithms appear to be inefficient. When sorting 10,000 unordered random cells it can take a couple of seconds which, when processing longer programs, can add up.

Appendix:

Increasing Array:

Insertion Increasing	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
1k	0	0	0	0	0	0
2.5k	0	0	0	0	0	0
5k	0	0	0	0	0	0
7.5k	0	0	0	0	0	0
10k	0	0	0	0	0	0
Selection Increasing	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**
1k	0.0625	0.0625	0.078125	0.0625	0.078125	0.06875
2.5k	0.421875	0.40625	0.40625	0.40625	0.421875	0.4125
5k	1.671875	1.625	1.640625	1.640625	1.734375	1.6625
7.5k	3.75	3.671875	3.71875	3.71875	3.890625	3.75
10k	6.671875	6.5	6.625	6.625	6.859375	6.65625

Decreasing Array:

Insertion Decreasing	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
1k	0.078125	0.0625	0.078125	0.078125	0.078125	0.075
2.5k	0.484375	0.5	0.5	0.5	0.484375	0.49375
5k	2.015625	2.046875	2	2.015625	2.015625	2.01875
7.5k	4.546875	4.609375	4.578125	4.578125	4.546875	4.571875
10k	8.109375	8.234375	8.140625	8.140625	8.140625	8.153125
Selection Decreasing	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**
1k	0.078125	0.0625	0.078125	0.078125	0.0625	0.071875
2.5k	0.421875	0.421875	0.40625	0.40625	0.4375	0.41875
5k	1.6875	1.671875	1.703125	1.671875	1.75	1.696875
7.5k	3.828125	3.8125	4.03125	3.71875	3.84375	3.846875
10k	6.765625	6.625	8.953125	6.625	6.859375	7.165625

Random Array:

Insertion Random	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
1k	0.03125	0.03125	0.046875	0.03125	0.03125	0.034375
2.5k	0.265625	0.265625	0.25	0.265625	0.25	0.259375
5k	1.015625	1.0625	1.0625	1.03125	1.046875	1.04375
7.5k	2.3125	2.46875	2.359375	2.328125	2.28125	2.35
10k	4.140625	4.296875	4.140625	4.15625	4.140625	4.175
Selection Random	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**
1k	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625
2.5k	0.421785	0.40625	0.421875	0.421875	0.4375	0.421857
5k	1.703125	1.65625	1.671875	1.671875	1.71875	1.684375
7.5k	3.8125	3.75	3.796875	3.75	3.859375	3.79375
10k	6.796875	6.671875	6.75	6.65625	6.859375	6.746875