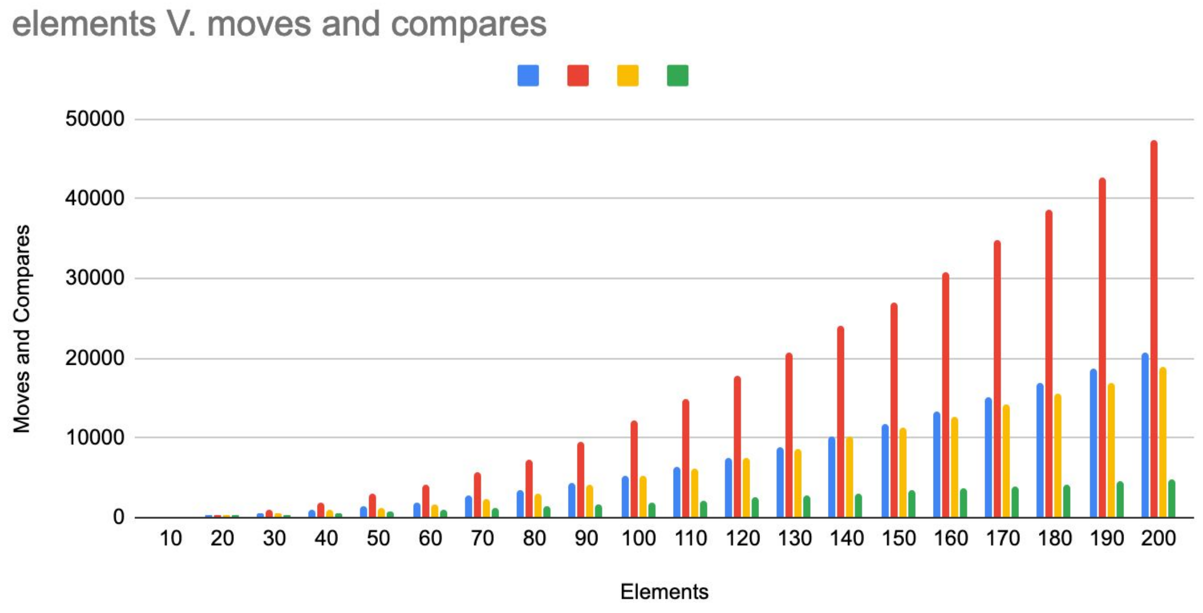
Artyom Martirosyan
11/10/2019
Assignment 6 writeup

For this assignment we took in some user inputs and randomly generated an array of numbers, then depending on the users selection we would send the generated array to a sorting algorithm. For this lab we were given 4 different sorting algorithms: Sort A(selection sort), Sort B(bubble sort), Sort C(insertion sort), and sort D(shell sort). Each of these sorts had relatively different methods for sorting the arrays(except sort C and D) starting off with sort A which would take an array and run through the array finding the smallest value and send it to the front. Though this sort was basic and easy to understand, the user will later realize that this array is extremely inefficient as its runtime is O(n^2) because it is running two different loops. Another fact I learned about the selection sort is that it can be extremely inefficient in for longer lists of numbers as if the list of numbers(n) is larger then say 10,000 it already begins taking a really long time, and that's only for 10,000 what if the number was larger? The next sort: sort B is also known as a bubble sort which simply compares the 2 values which are right next to each other at a time. Immediately after looking at this I can say that this can take a very long time even with smaller numbers as the time constant is also O(n^2) which is very inefficient. Another issue is that, again like selection sort if the number is larger than 10,000 it can take much longer than other sorting algorithms which is both inefficient and a waste of space and time. The next sort: sort C is known as insertion sort; instersion sorts is a concept where the algorithm will compare items in an array with a gap size of one(right next to each other). Unlike sort BY sort C isn't trying to compare everything and know the smallest values out, in theory it's accomplishing both at the same time, but like the other sorting algorithms this takes O(n^2) because in simple terms there are two loops meaning that the algorithm is running for a long time and will be swapping values almost everytime it runs this can be efficient for smaller values, but once you reach a large value such as 100,000 it begins to take a lot of time. While the first three sorting methods take about O(n^2) or n(n+1)/2, the final algorithm known as sort D or shell sort takes O(n^3/2) or O(nlog^2(n)). This method basically does sort C, but rather then only checking with a gap size of 1(right next to each other), it check at larger gaps and works its way down, at first I had assumed that this method would be highly inefficient compared to the insertion sort, that was until I realised the bigger picture. Thought insertion sort is great for a small amount of numbers, once the number gets rather large it becomes highly inefficient, that's where the gaps come in, by using gaps you theoretically cutting the amount of moves down as the smaller the gaps get the more items will be in their right place resulting in less moves the final go through. To show this I ran an experiment in order to find the efficiency threshold of each different method by starting at 50 and going up to 1,000 and having an extra test at 100,000 just for kicks. As the results show below: Note: Blue bar is Selection sort, Red bar is Bubble sort, Yello bar is insertion sort, Green bar is shell sort. As seen below shell sort is even more efficient than I had assumed, this is because the random seed is giving a random scenario meaning that this set of numbers could be more ideal for shell sort then the other sorts, but either way we can see that shell sort is the most efficient followed by insertion, then selection and finally bubble sort this again could be due to the seed(seed being an outlier).


elements V. moves and compares
🟦 🟥 🟨 🟩
Moves and Compares
Elements