605.202: Introduction to Data Structures

Thomas Muamar

Project 4: Sorting

Due Date: Aug 17,2021

Dated Turned In: Aug 16, 2021

Project 4: Sorts

Justification for data structures/reasons for choices:

To start off with this project I decided to use my own set of files by creating a class called input files. The way I decided to make the input files in a .dat file and have it use a number random generate in order to create a different set of numbers. That first set of numbers would be the random file. Then in order to reverse order them I had to create a method that would go through a for loop in order to reverse the files. Once that was done in order to read each text file, I use a scanner sequence to read each value and input it into an array. Using an array sequence seemed to be the most optimal due to just having the random-access availability. Since we had a set length for each of the arrays it makes it super easy to implement an array for each of the files. In order to get all the values on the console I limited the values that print on the console to only the first 50 sorted values in order to see that each of the file was performing in the correct manner. It also allowed less of a load time when checking each run is working.

With the quick sort I started by using a code from Geeks for geeks that is called a advanced quick sort algorithm which uses a quick sort with a insertion sort. Made a few different modifications to the partition selection and so I don't have to end up making multiple quick sorts for the different k values that we needed to test. Used the average of the low and high value with then the file and then divided them by two in order to get the middle partition to allow for the quick sort to be divide into two and then be sorted. With the middle pivot being compared to the K-value each time we had it be greater than the K-value it would end up using insertion sort to finish off the sorting of the quick sort.

In the shell sort algorithm, I had to make four different classes for each of the different increments. Starting with the first which was Knuth's sequence and in order to do that I used the equation from lecture in order to calculate the interval/gap for that sequence. Basically equation used was having the interval start at 1 then use the equation of (interval*3)+1. Then the next one was just adding two instead of 1. For the third sort I couldn't figure out the exact equation, so I made it look through the increment array and then find the first gap that was greater than the array length. Once it has been found then it ends and moves the increment back two to get the starting value.

In order to produce the time complexity of each of the sorts in the code I created two classes one called shell sort comp and other quick sort comp. In these classes I used System.nanoTime() method in order to get the time for each sort. I took the start time and then when the array was sorted, I then took the end time and subtract both to get the exact time it took to sort the integers.

What I Learned:

While doing this project I definitely learned a lot more about shell and quick sort algorithms. Looking through the different quick sort algorithms that were online and how they had different ways of implementing the way they get their pivots. With shell sort got a better understanding of how different increments work and how it can end up effecting the amount of time it takes for the items to sort.

What I also learned is the way the sorts are implemented might end up effecting the time complexity of each of these items.

What I might do differently next time/if:

With this project something that could be done differently next time is to create a recursive solution for both of the sorts and compare the difference with the iterative solution that I decided to do for this project.

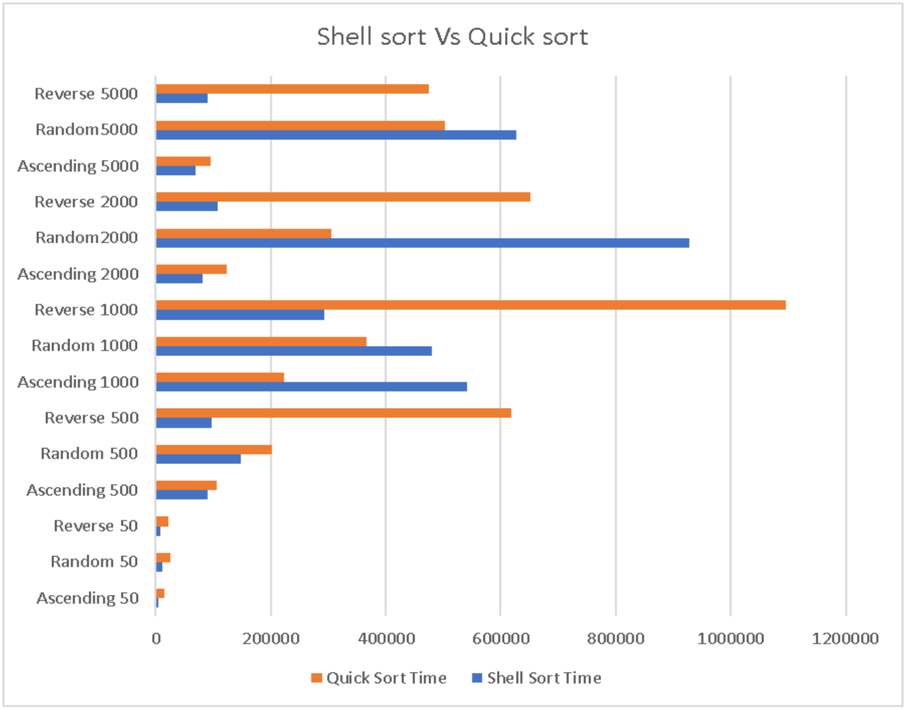
Something I also could have done next time is trying to figure out a way to calculate the average run time when the quick sort and shell sort is running a certain amount of time instead of doing it manually to get a decent chart for the two sorts.

Another thing that could also be interesting to see instead of using arrays to sort the algorithms here is to try and use a linked list to store the values and then see what the time complexity be with sorting these items with using a linked list.

Shell Sort Vs Quick Sort:

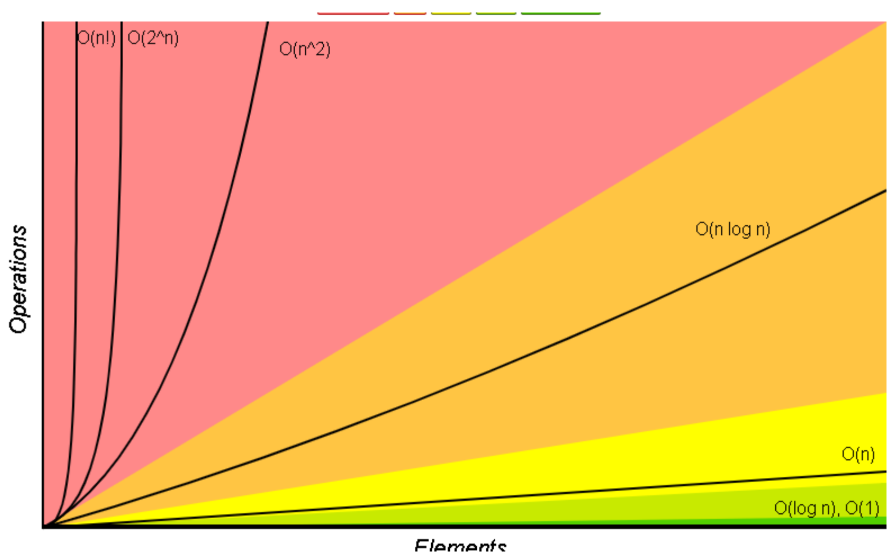
Comparison of Knuth's shell sort with a quick sort:

file names	Shell Sort Time	Quick Sort Time
Ascending 50	4400	14400
Random 50	11200	26200
Reverse 50	7600	22600
Ascending 500	90900	106200
Random 500	148700	201900
Reverse 500	97300	617600
Ascending 1000	541400	222400
Random 1000	481000	366500
Reverse 1000	293900	1095200
Ascending 2000	81300	123300
Random2000	928500	306200
Reverse 2000	107600	651800
Ascending 5000	69300	96300
Random5000	628000	502500
Reverse 5000	89900	475000



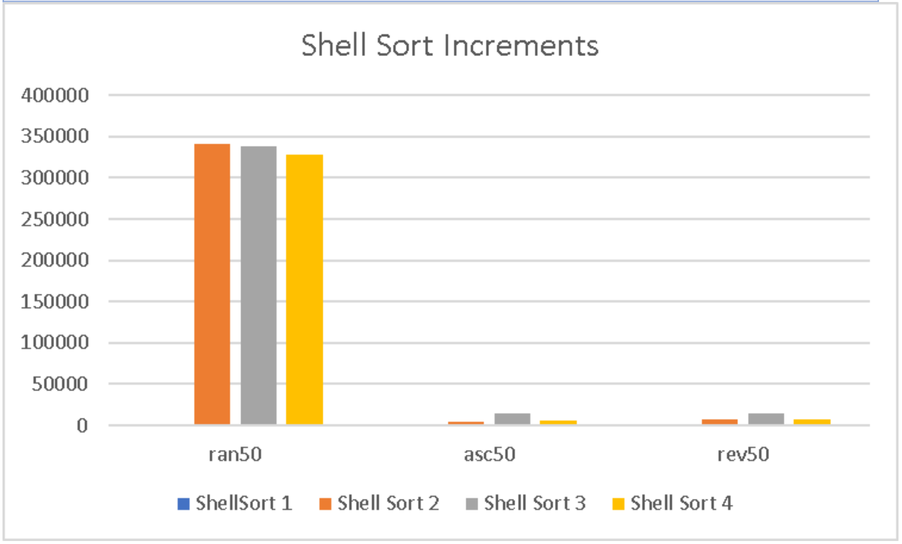
Shell sort when looking at the different runs produced seems out perform a quick sort when the items with in a file are in reverse order. The reverse order for the shell sort is out perfoming the quick sort better and better as the file gets larger. The reason we see this being a key point in the data is due to quick sort have a worst time complexity of O(n^2). Since the reverse values would need to interchange every time within the quick sort and is implemented with an insertion sort which could end up causing the average cause to become just as bad as the worst case scenario. As the files end up getting larger it seems to be that the random value sort starts to become an issue with the shell sort and starts to hit the worst case more often. Unlike the quicksort were the time complexity seems to be worse for it when it is sorting the values in reverse order. This could be due to the pivot selection of the quick sort causing it to have the worst case possible of not spliting the data equally from the pivot selection.

The space complexity of these two sorts don’t seem to be much of an issue. The reason for that is with a shell sort you have a constant of o(1) for the space complexity and the quick sort would be o(log(n)). When graphing these two space complexities it shows that they are both at the same location.



Comparison of the different Shell Sort increments:

file	ShellSort 1	Shell Sort 2	Shell Sort 3	Shell Sort 4
ran50	1600	341400	338500	328000
asc50	700	4100	14000	5700
rev50	1300	7600	13800	7700

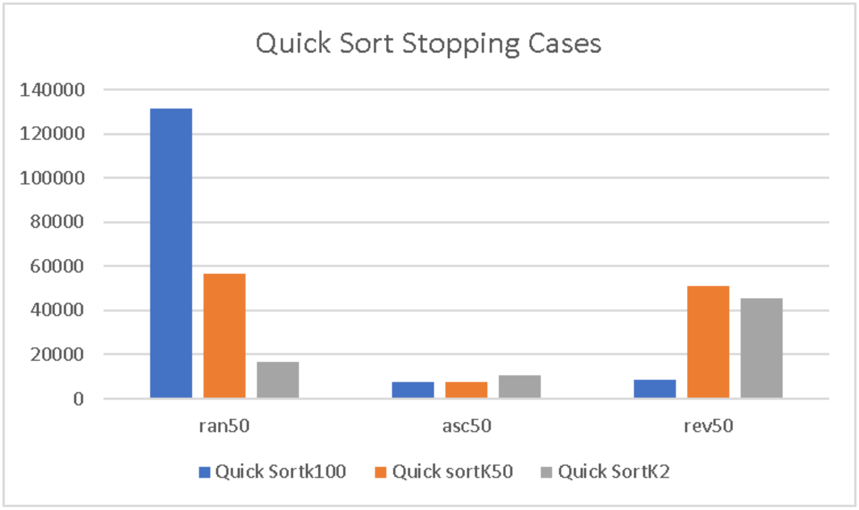


When comparing the different shell sort increments it seems like the knuth sequence has the best time complexity in comparison to the 3 other shell sort increments. Shell sort 2 performs the increment of (1, 5, 17, 53, 149, 373, 1123, 3371, 10111, 30341). Shell Sort 3 performs the increment of (1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160). Shell sort 4 performs the increment of n/2,n/4,n/8 and so on. Depending on the increment that the shell sort uses it will have different time complexities for each of them. For example the worst case scenario for a time complexity that uses the Knuth sequence would be o(n^3/2). While some other increments might end up having a complexity of o(n^4/3). As the gap sequence seems to be getting worse the worse the time complexity becomes for the sorting of a random set of values.

Quicksort with different K values:

file	Quick Sortk100	Quick sortK50	Quick SortK2

ran50	131600	56500	16900
asc50	7400	7800	10600
rev50	8600	51300	45300



Here in the quick sort the most optimal all around seems to be the k value at 2. Due to the random and reverse not performing all to will on the random sort having the worst-case complexity. If you were to have a sorted set of data, the most optimal would be using a k value of 100. K value of 100 would also work for reverse while the other k values end up producing the worst case on the run time.

In the end both shell sort and quick sort can have an issue with their time complexity dependent on the scenario. The shell sorts efficiency are effected by the different increments that are implemented and from the above short it shows that the Knuth's sequence seems to be the best. The sort that I would most likely use in these runs on both quick sort and shell sort would most likely be a shell sort that uses the increment with Knuth sequence.

With most of these sorts a sorted data set with low amounts of values needed to be sorted it will have a less of a time complexity. While the number of items in the file increases the sort will take longer just because it has to check that each value is sorted but doesn't have interchange each of the values, so it becomes the best case. While in the reverse order it has to end up doing a lot of interchanges with each value and as each interchange goes by it compares with the next value until that value is in the right spot. While random order could end up being the same as reverse but sometimes the value of that item won't need to make as many swaps in comparison to a reverse order were u know the item needs to be swapped over and over again.

References:

1. ShellSort. GeeksforGeeks. (2021, July 20). https://www.geeksforgeeks.org/shellsort/.
2. Advanced quick sort (hybrid algorithm). GeeksforGeeks. (2021, August 4). https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm/.