Devan O’Boyle
CSE13s
Assignment 3: Sorting - Putting Your Affairs in Order

Write Up

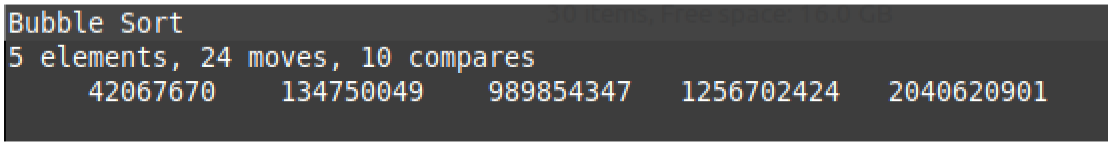
Each sorting algorithm runs with a different complexity depending on the way that the algorithm handles looping.

In bubble sort, two nested loops are used to continuously loop through the array and swap adjacent elements with one another. Since the sort continuously iterates through the array as many times as it needs to until the array is sorted, the worst case scenario in bubble sort is a complexity of O(n^2) with n being the size of the array.

In shell sort, two nested loops are used similar to bubble sort. However, rather than comparing adjacent elements, shell sort compares elements that are gap indices away from one another. Since the value of gap is given from an array which changes the value of gap after each iteration of the array, the complexity of shell sort is dependent on the pattern of gap values. Since nested loops are used, the worst case complexity for shell sort is O(n^2) or better.

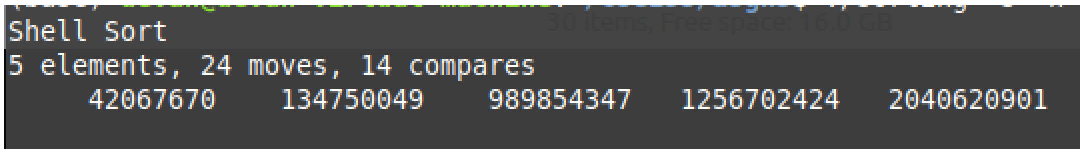
In quick sort, a nested loop is also used which makes its worst case complexity O(n^2). However, because it continuously divides the number of values that need to be compared, it is often able to tackle sorting much more efficiently since it only has to worry about a smaller number of values before it goes on to sort the rest which means that it doesn’t have to continuously iterate through larger values that it won’t sort until the end. This is what allows quick sort to have an average complexity of O(nlog(n)).

I experimented a bit with bubble sort and noticed how much faster it works when there are less elements.

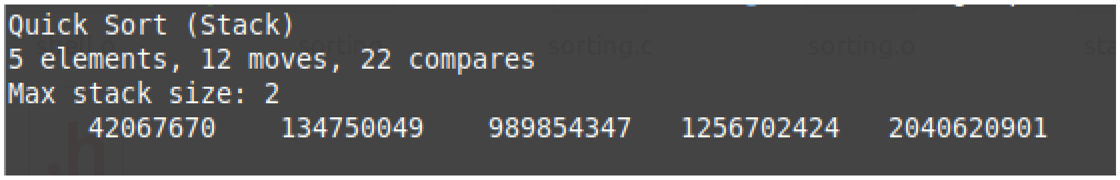


```
Bubble Sort
5 elements, 24 moves, 10 compares
     42067670    134750049    989854347   1256702424   2040620901
```

Now granted, of course if there are less values it will perform better, but it was roughly able to work just as well, if not better, than a shell and quick sort.



```
Shell Sort
5 elements, 24 moves, 14 compares
     42067670    134750049    989854347   1256702424   2040620901
```

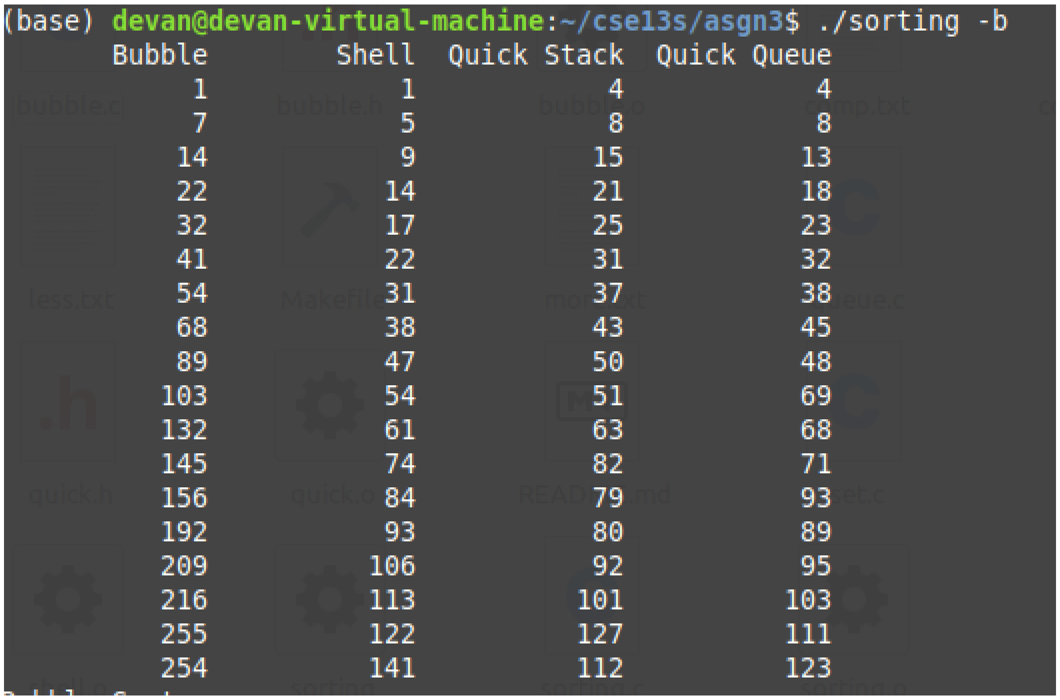


```
Quick Sort (Stack)
5 elements, 12 moves, 22 compares
Max stack size: 2
     42067670    134750049    989854347   1256702424   2040620901
```

As we can see, bubble sort had the same number of moves as shell sort with four less compares, and although bubble sort had twice the moves of quick sort, it had half the compares to make up for it.

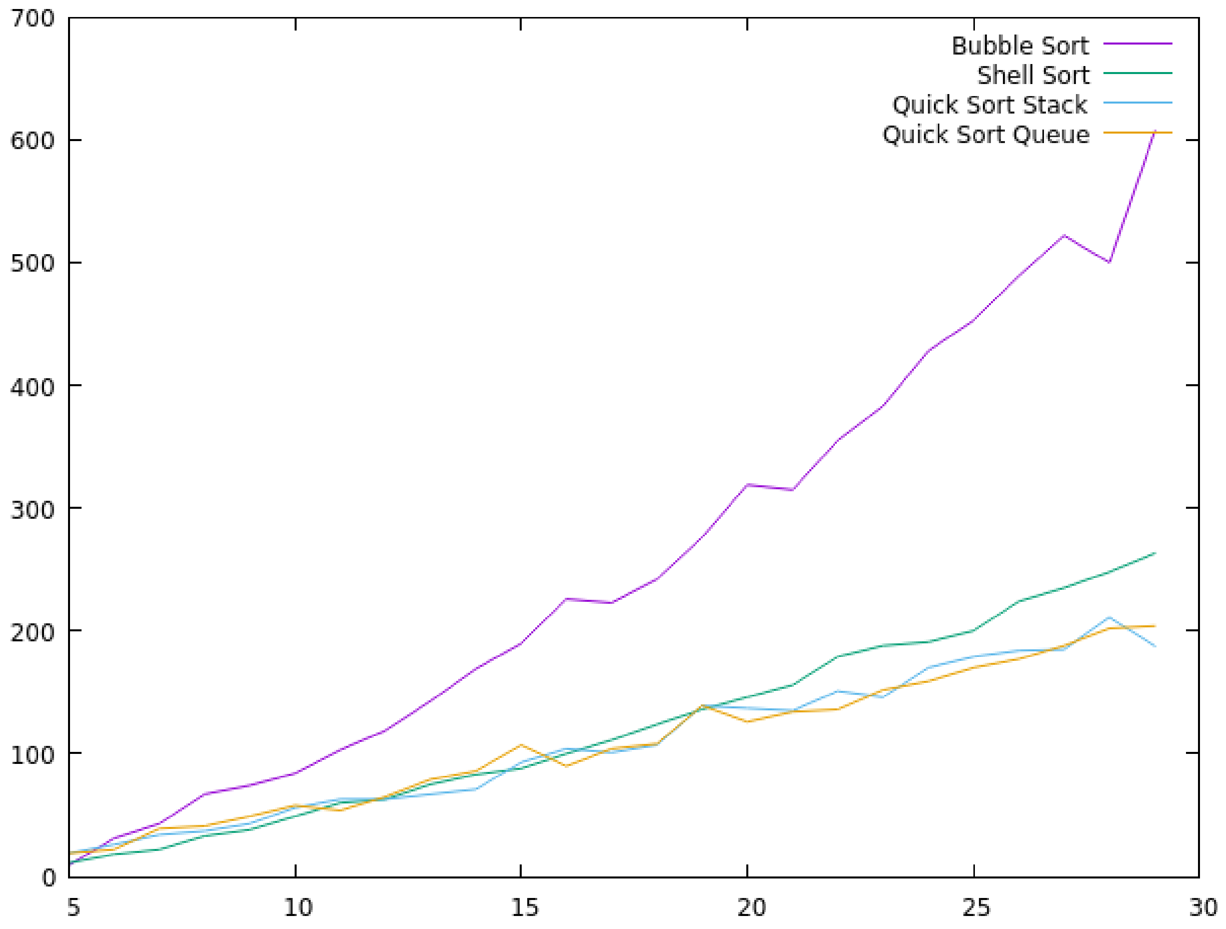
So clearly, bubble sort is a viable sort at least for smaller values. However, as we try to sort more elements things get a bit more complicated.

By taking a look at the following output table of comparisons, we can see that as the size of the arrays gets larger, bubble sort ends up making far more comparisons than the other sorts.



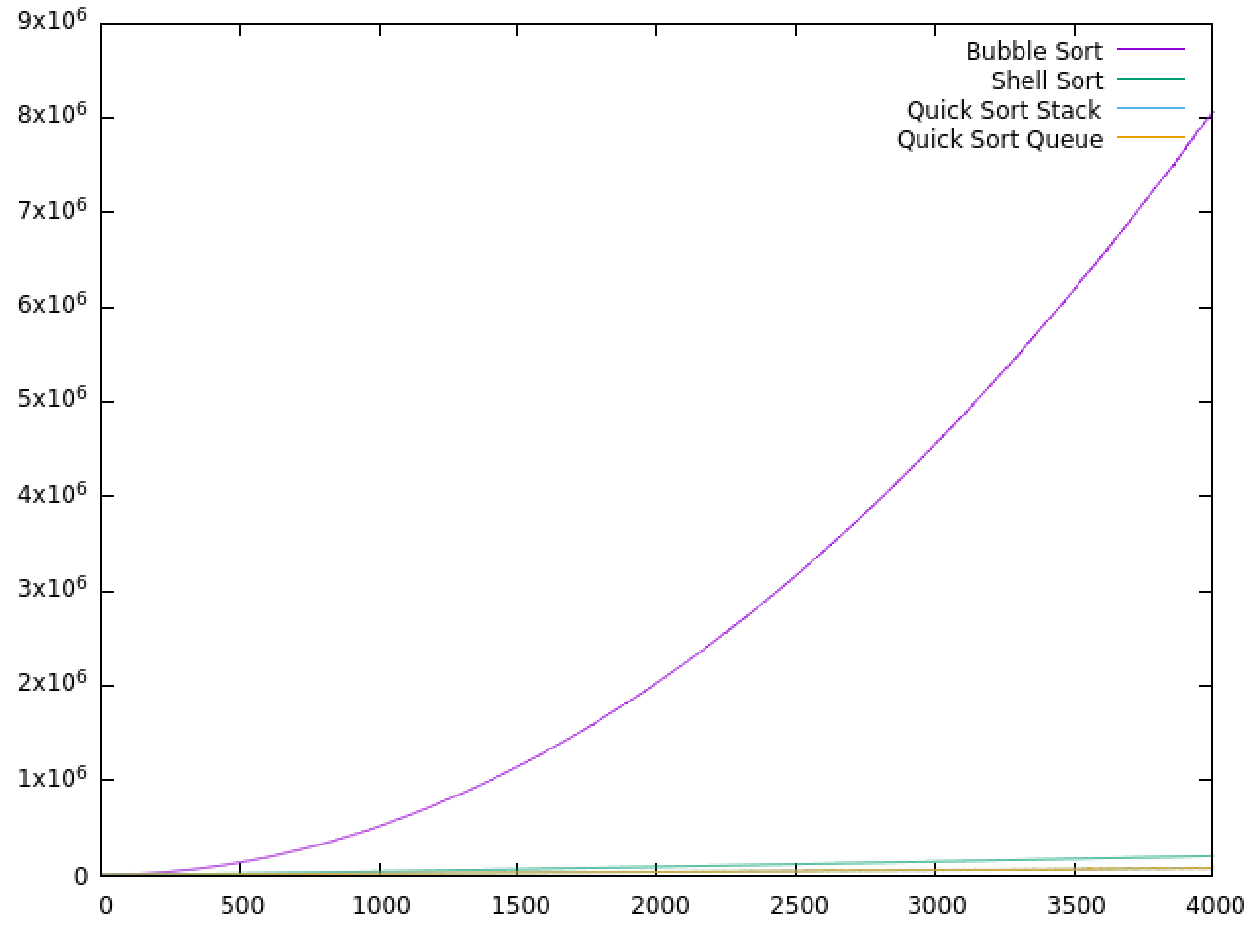
```
(base) devan@devan-virtual-machine:~/cse13s/asgn3$ ./sorting -b
      Bubble        Shell  Quick Stack  Quick Queue
           1            1            4            4
           7            5            8            8
          14            9           15           13
          22           14           21           18
          32           17           25           23
          41           22           31           32
          54           31           37           38
          68           38           43           45
          89           47           50           48
         103           54           51           69
         132           61           63           68
         145           74           82           71
         156           84           79           93
         192           93           80           89
         209          106           92           95
         216          113          101          103
         255          122          127          111
         254          141          112          123
```

This is better seen in the following graph where the size of the random array is on the x-axis and the number of comparisons that each sort made for that corresponding size is on the y-axis.



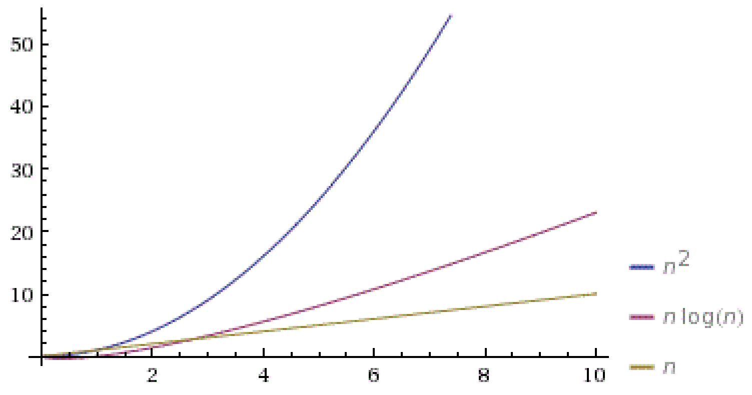
Although bubble sort starts out being the most efficient sort at smaller values, but it’s slope quickly gets steeper and steeper. Therefore, bubble sort becomes obsolete as we try to sort random arrays with sizes greater than around 5 to 10. This is because its worst case complexity, O(n^2), is its average case. On the other hand, shell sort, despite having the same worst case complexity, is able to do better than bubble sort in the long run because the gap sequence allows certain elements to be sorted earlier than in bubble sort, making it so that less comparisons need to be made overall.

A similar situation takes place with quick sort, but this is better shown in the graph below.



Although it is a bit harder to see, there is a distinct difference between the complexity of quick sort and shell sort as the arrays get bigger since quick sort has a noticeably smaller rate of change than shell sort. This is because of quick sort's average complexity of O(nlog(n)). While shell sort's complexity varies on its gap sequence, quick sort tends to be much more consistent since it always splits up the array in the same way.

Plot:



To further reaffirm just how different the complexity of O(n^2), take a look at the graph above. As we can see, by the time the graph of nlog(n) reaches around a y value of 6, the graph of n^2 has already reached a y value of 50. This just goes to show how much worse it is for bubble sort to have its average case be its worse case of O(n^2), while quick sort is able to average a complexity of nlog(n) and only in its occasional worse case does it reach O(n^2).

Overall, I learned in this lab that different sorts have various different complexities and a lot of their variencies come down to how they handle looping and comparisons in their algorithms. In particular, I found that sorts like shell sort and quick sort that don’t simply iterate and compare adjacent values tend to be much quicker than one that does such as bubble sort.