Zhu, Jessie Spring 2021 Sunday, April 25, 2021

Assignment 3: Sorting, Putting your affairs in order

Writeup

In this assignment, we implemented three types of sorting methods (four unique sorting functions). For each sorting function, we also keep track of the number of comparisons and moves made for ease of complexity comparison. All of my sorting functions are within the margin of error of the given binary file's comparisons and moves made.

Time complexity:

Bubble sort has a worst case time complexity and average complexity of O(n^2). This means that Bubble sort has a very slow computational time and is not a practical method of sorting. There are many other sorting methods have a same worst case time complexity, but have a much better average time complexity.

Shell sort using the Pratt Sequence has a worst case time complexity of O(nlog^2n) which is also the average time complexity. Compared to other sequences, such as Shell sequence or Knuth sequence, this gap sequence is relatively efficient

Quick sort stack/queue has a worst case time complexity of O(n^2). Although this is not very good, Quicksort's average is O(nlogn) which is the quickest of all sorting methods. This is because of the way Quicksort can be implemented. The pivot can be chosen to avoid the worst case, and the inner loop of the code can be optimized. So despite the bad worst time case, this is a very quick method.

What I learned:

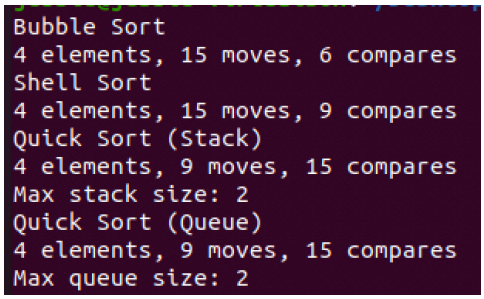
From this assignment, I learned more about how complexity works. With counting moves and comparisons, it took some trial and error to figure out where they are supposed to go; specifically I learned what kinds of things in the function add to the complexity of a sorting functions. I also learned how to implement stack and queues in c code. Previously I had only used a stack in MIPS code so I had never defined functions specific to creating one. This is also the first assignment where we use global variables and it took a few tries to make it work because I didn't get a lot of information on how a global variable should be declared or if something unrelated was

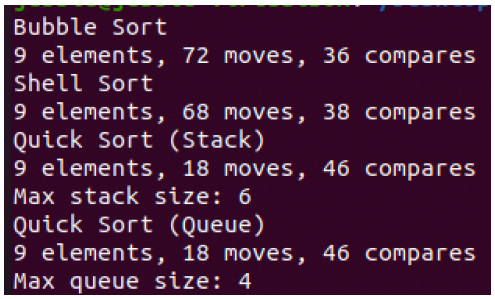
causing the error message. Overall, this assignment was very insightful in learning about how sorting algorithms and data structures work in c code.

Experimenting with sorts:

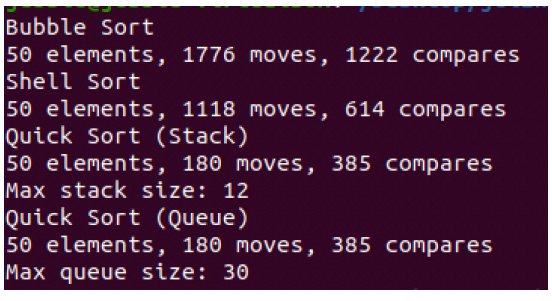
The size of the array plays an important role in a sorting method's time complexity.

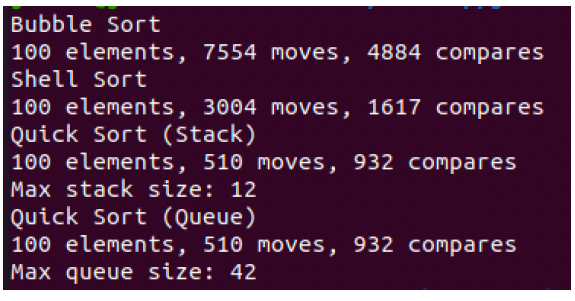
- **Smaller arrays** (less than 10): Bubble sort and shell sort make less comparisons but also have moves than quicksort (stack/queue) when the array is small. Performance between bubble and shell sort are very close.

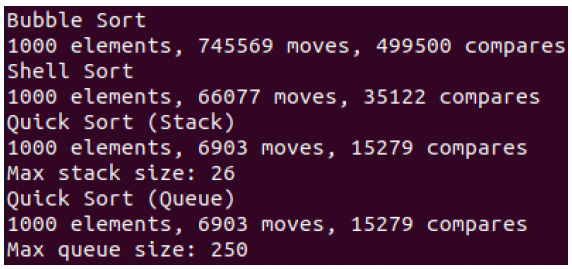




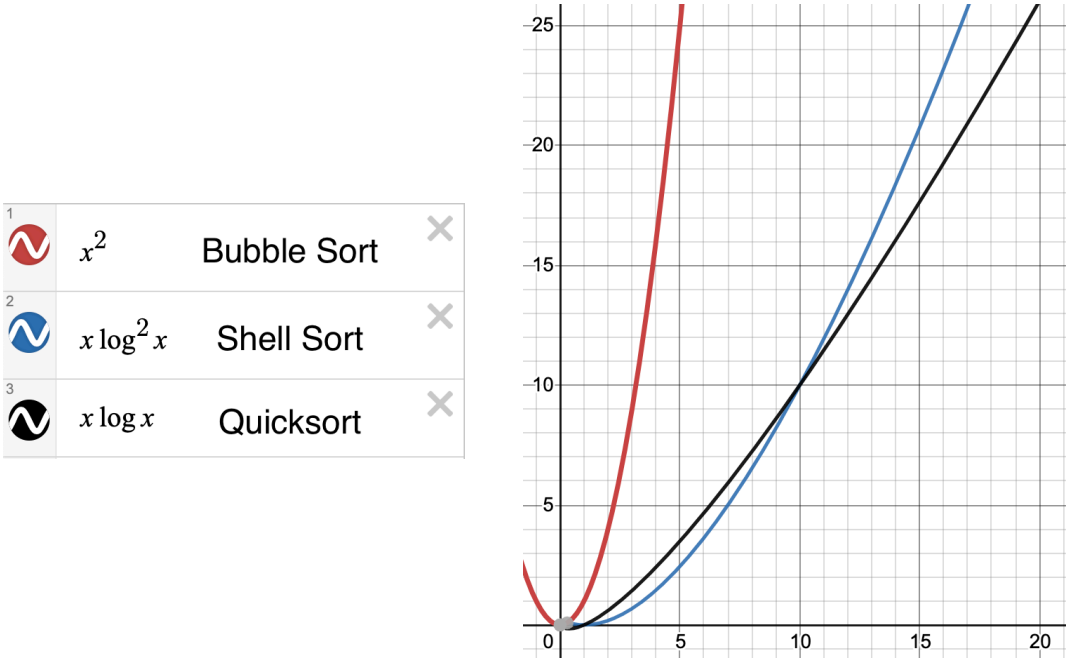
- **Larger arrays** (around 50+): Bubble sort and shell sort have way more comparisons and moves than quicksort does when sorting larger arrays. Quicksort outperforms the other two sorts by a lot. Shell Sort makes significantly less moves and comparisons than bubble sort when sorting a large array.







- Overall, Bubble sort and Shell sort might be better for smaller arrays (for less comparisons) but quicksort greatly outperforms bubble and shell sort in overall efficiency: less moves and less comparisons.
- Bubble sort's complexity increases exponentially so when the array is large, it is the worst performing sorting method by a lot.
- These patterns match the average complexity for all three sorts, as graphed below. Complexity of quick sort becomes comparatively better when n is greater than 10.



(Average time complexity graphs, graphed on Desmos)

Max Stack and Max Queue:

- The stack and queues don't get very big compared to the number of elements in the array. Referring to the images below, max stack and queue size in quick sort are all a lot less than the max number of elements. The larger the size of the array, the less percentage of stack and queue is used. As the size of the array gets larger, the difference between max stack and max queue size gets larger. As you can see from the last photo, max queue size is greatly bigger than max stack size.

```
Bubble Sort
9 elements, 72 moves, 36 compares
Shell Sort
9 elements, 68 moves, 38 compares
Quick Sort (Stack)
9 elements, 18 moves, 46 compares
Max stack size: 6
Quick Sort (Queue)
9 elements, 18 moves, 46 compares
Max queue size: 4
```

```
Bubble Sort
13 elements, 144 moves, 75 compares
Shell Sort
13 elements, 130 moves, 71 compares
Quick Sort (Stack)
13 elements, 30 moves, 71 compares
Max stack size: 6
Quick Sort (Queue)
13 elements, 30 moves, 71 compares
Max queue size: 8
```

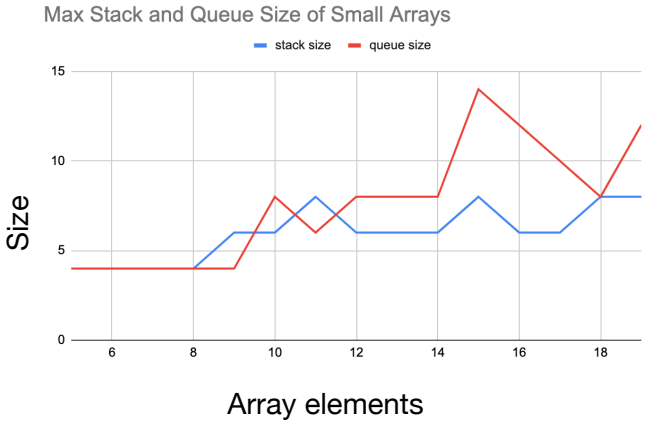
```
Bubble Sort
50 elements, 1776 moves, 1222 compares
Shell Sort
50 elements, 1118 moves, 614 compares
Quick Sort (Stack)
50 elements, 180 moves, 385 compares
Max stack size: 12
Quick Sort (Queue)
50 elements, 180 moves, 385 compares
Max queue size: 30
```

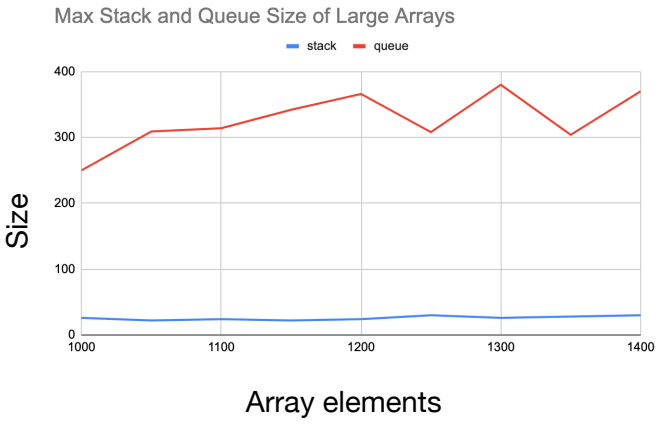
```
Bubble Sort
100 elements, 7554 moves, 4884 compares
Shell Sort
100 elements, 3004 moves, 1617 compares
Quick Sort (Stack)
100 elements, 510 moves, 932 compares
Max stack size: 12
Quick Sort (Queue)
100 elements, 510 moves, 932 compares
Max queue size: 42
```

```
Bubble Sort
1000 elements, 745569 moves, 499500 compares
Shell Sort
1000 elements, 66077 moves, 35122 compares
Quick Sort (Stack)
1000 elements, 6903 moves, 15279 compares
Max stack size: 26
Quick Sort (Queue)
1000 elements, 6903 moves, 15279 compares
Max queue size: 250
```

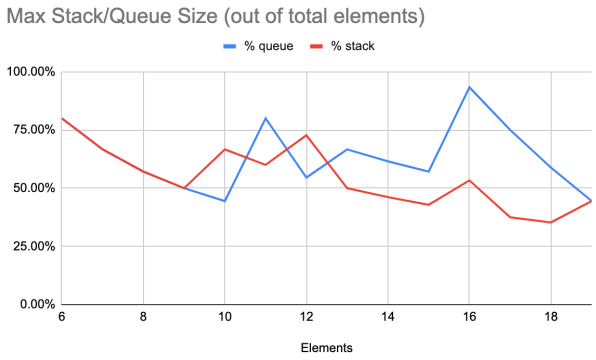
```
Bubble Sort
9000 elements, 60915552 moves, 40480622 compares
Shell Sort
9000 elements, 991788 moves, 523381 compares
Quick Sort (Stack)
9000 elements, 82824 moves, 165114 compares
Max stack size: 34
Quick Sort (Queue)
9000 elements, 82824 moves, 165114 compares
Max queue size: 2002
```

- From the graphs below, you can see the patterns I described above, where all y (max stack size) values are below their x (number of elements) values of the graph

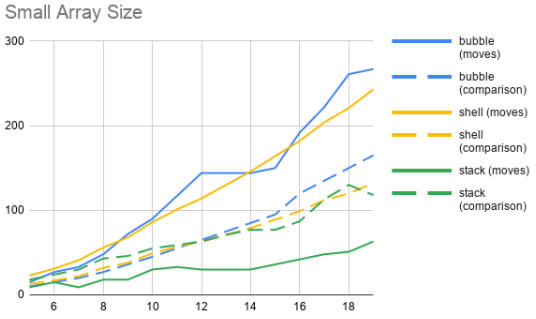


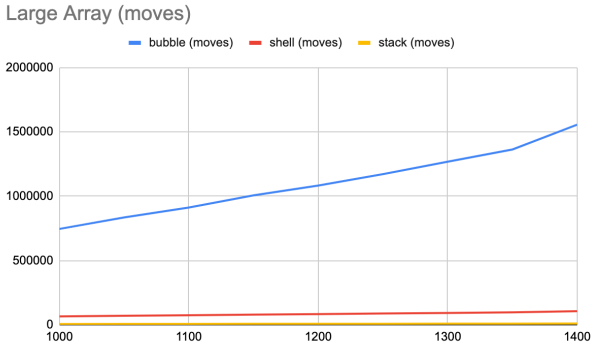


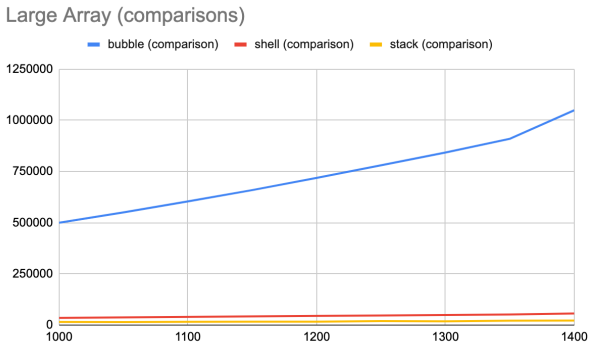
- These graphs below show the percentage of the stack and queue used for when the array is small (left) and when the array is large (right). As you can see, the larger arrays would use a less percentage of their stack and queue.



Graphs and Analysis:







Moves/Comparisons of all sorting methods (Small and Large Arrays)

- In these graphs, I am comparing the complexity of each bubble between sorting small arrays and large arrays. In the first graph, we can notice that all three types of sorting start very close to each other in value.
- As the amount of items in the array increases, we can see that bubble's complexity increases extremely fast, compared to the rest of the sorts.
- Shell sort is the next slowest, but is still many times faster than bubble sort. This is possibly due to the fact that shell sort is better at getting items in their relative areas before sorting. Bubble does the comparative sorting right away which means that if an item is on the wrong side of the array, then this would need many comparisons and swaps to get there. So in a bigger array, shell sort would perform way better
- Quicksort was the fastest of all of them as the array size increases. Similar to shell sort, this sorting method is better when arrays are large because it will split the array based on a pivot value, which makes the worst case complexity very unlikely.