Natalie Valett, nvalett
CSE13S
Prof. Darrell Long
April 17, 2021

Asgn3 Writeup:

Identify the respective time complexity for each sort and include what you have to say about the constant.

Sort	**Time Complexity**	**Notes**
Bubble	O(n^2)	While O(n^2) is the worst time complexity, it can be shorter to the extent that the former part of the array is already in order. However, as a general case it will tend towards O(n^2) because the algorithm loops through n elements in the array approximately n times, resulting in n * n scans and comparisons, or n^2. This sort has the worst time complexity of the 3, a difference that's proportional to the size of the constant n. This means that for smaller constants, this algorithm has can be more efficient than quick sort and closer in efficiency to shell sort, but as n grows it becomes less efficient than the other 2.
Shell	O(n^5/3)	Shell sort is a unique case where the time complexity is mostly dependent on the given gap sequence. The time complexity is proportional to both the number of gaps given and the magnitude of each gap (particularly if the array is large enough to qualify implementing such large gaps). For this reason, shell sort can potentially be extremely efficient if the number and magnitude of the gaps is conservative and efficient, but the bigger the array and gap sizes, and number of gaps to implement, the

		less time-efficient this algorithm will be. As with most algorithms, the constant n being small lowers the time complexity, and a large constant n increases it.
Quick	Worst Case: O(n^2) Average Case: O(nlog(n))	Because quicksort breaks the larger array into smaller sub-arrays, it's likely that many of these sub-arrays will happen to be sorted or at least have sorted sections. The worst case scenario of time complexity O(n^2) would only occur if every random sub-array were independently sorted in the opposite manner from the function's goal (ascending vs descending), which is extremely unlikely given the random nature of each sub-array's boundaries. In most cases, the time complexity will be O(nlog(n)) because the maximum possible amount of moves and comparisons are rarely needed.

What you learned from the different sorting algorithms.

I learned that there are many more ways of sorting an array than I anticipated. Bubble sort was the one that came most naturally to me through intuition, but having to understand and implement shell and quick sort was more challenging because of the nuance in them. I had an especially challenging time understanding quicksort. The idea of utilizing stack and queue data structures to track partition indexes to iteratively compute on was a very roundabout way to sort an array, but it proved to be very efficient. I was surprised to learn the different time complexities of each sort method, and to identify what cases are best to use each one.

How you experimented with the sorts.

For the most part I tried my best to implement the psuedocode given in the assignment document, so I didn't diverge too much from the methods given there. The most notable experimentation is use of the do-while vs a while loop in the partition() function for quicksort. As the assignment document suggested, I used a do-while loop to compare A[i] and A[j] to the pivot. However, this implementation caused the sort to stop too soon, resulting in a returned array composed of sorted sub-arrays, but not being fully sorted itself. This issue took some time to decipher, but eventually I decided to change the do-while to a while loop and it worked.

The other way in which I augmented the code was by making a less_than() function that both compares 2 elements and iterate the compares external variable. This ensures that every time an array comparison is made that the variable is iterated, in order to avoid under-counting or over-counting.

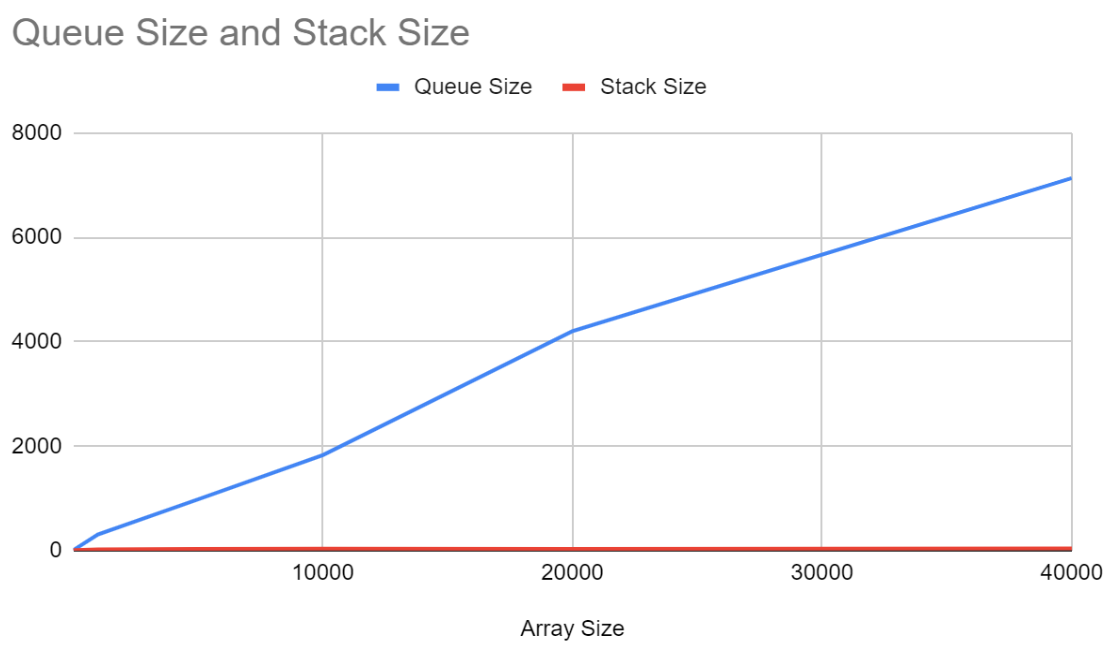
How big do the stack and queue get? Does the size relate to the input? How?

At a very small array size, the queue and stack size are roughly the same, but it doesn't take much growth for the queue size to grow rapidly larger than the stack. The stack seems to stay relatively small (under 100) for up to and beyond 1,000,000 array elements, while the queue alternatively grows into the hundreds of thousands.

This divergence in size can be explained by the 2 datatypes different methods of recursing over all the array subsets. The stack implementation is depth-first, meaning that once it has a new sub-array returned to it, it immediately dives into it and segments all the sub-arrays within it before returning back to the main branch and exploring the other branches. This leaves the stack with less nodes to keep track of (to keep on the stack at one time), because it handles each step immediately as it comes, as is implied by "depth-first". Alternatively, the queue is "breadth-first", meaning that instead of going "down" or following each node as it appears, it instead collects all those nodes in the queue and ignores them for the time being, intending to go back to them once the breadth of nodes on that level have been enqueued. This approach results in many more elements getting saved to the data structure, as it tries to collect them all before going "down" into them iteratively.

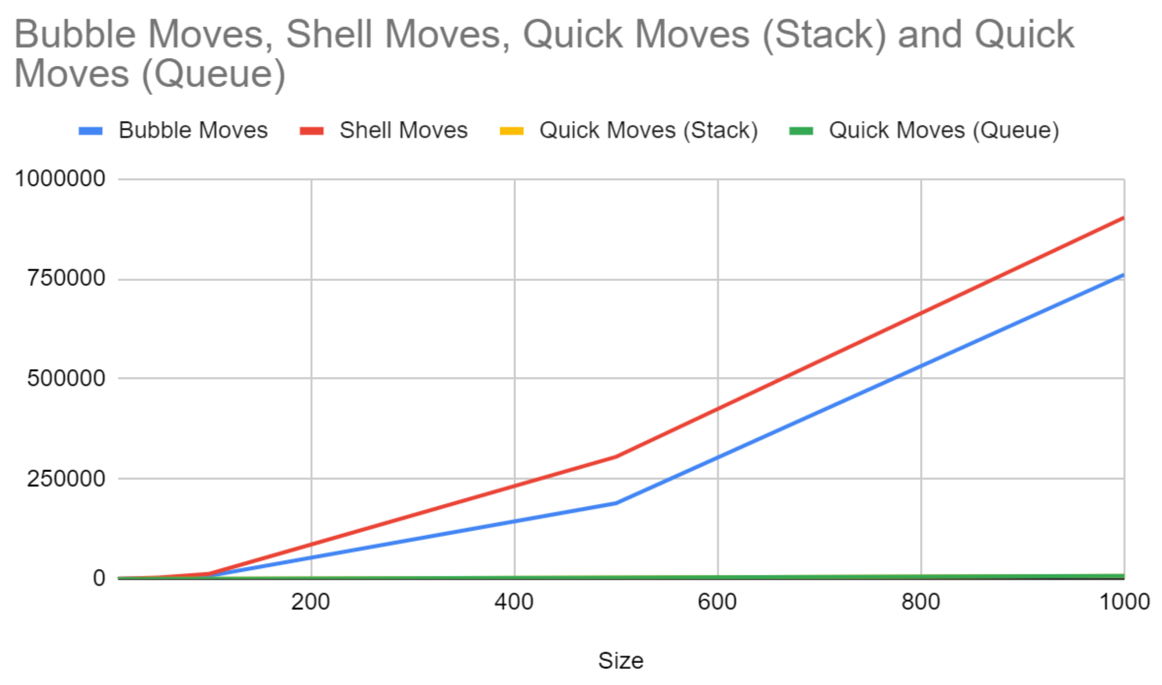
Array Size	**Queue Size**	**Stack Size**
10	8	6
100	36	12
1000	312	24
5000	982	32
10,000	1830	38
20,000	4204	34
40,000	7134	42
80,000	16,382	42
100,000	17,492	50
1,000,000	159,380	64

Visual representation of queue and stack size in comparison to array size:

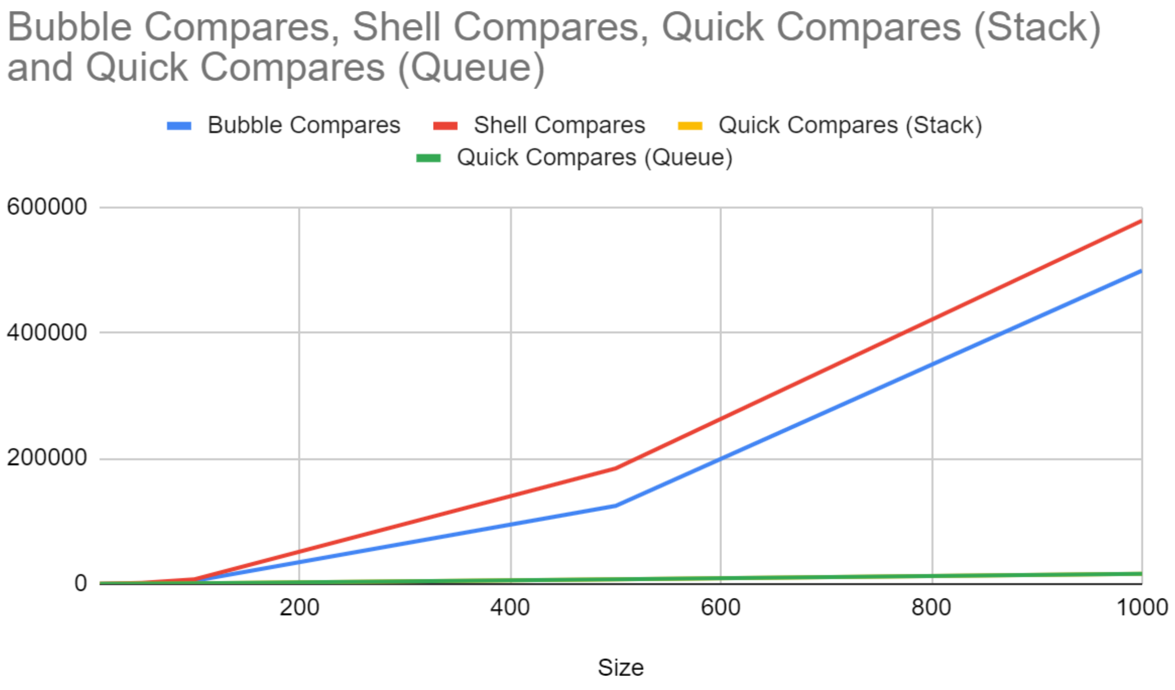


Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements.

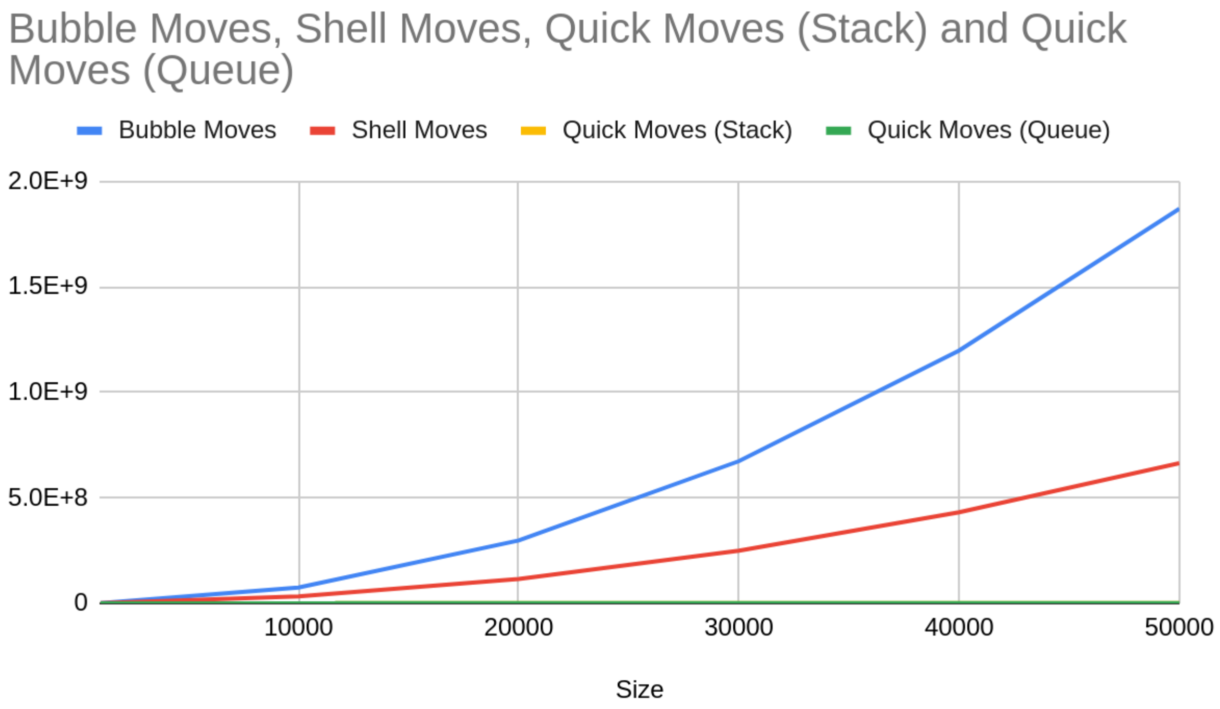
<u>Moves on smaller arrays (up to 1000 items):</u>



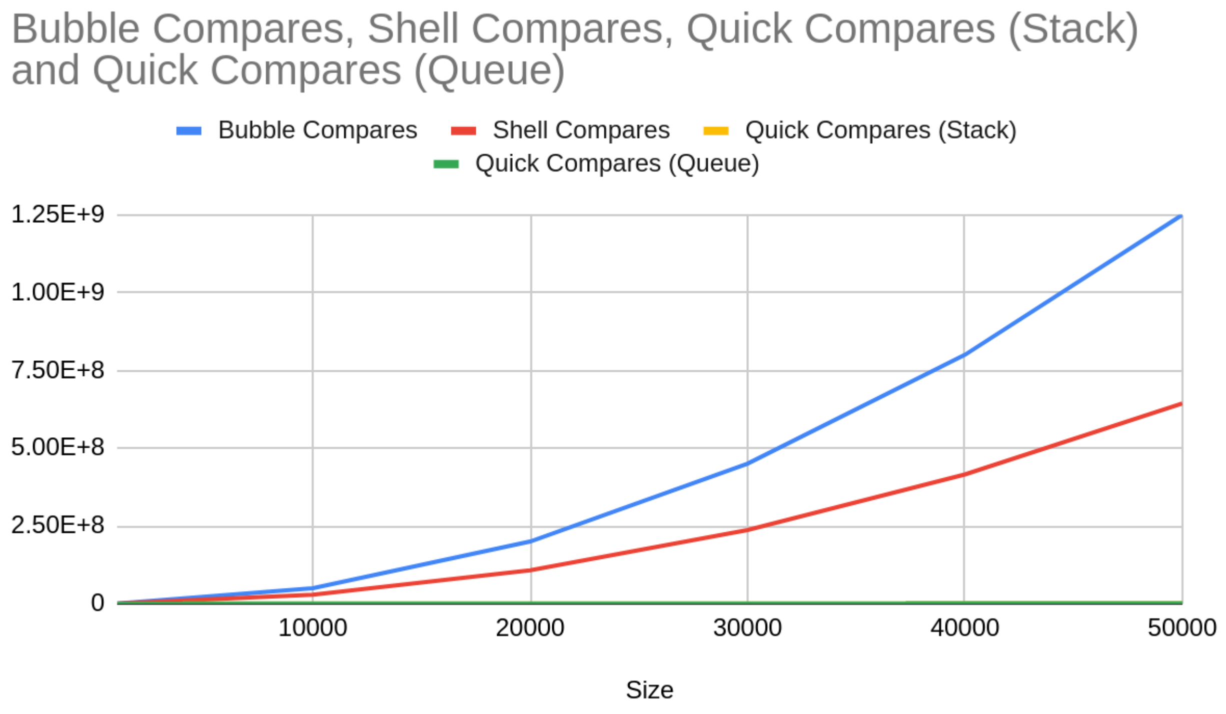
Compares on smaller arrays (up to 1000 items):



Moves on bigger arrays (up to 100,000 elements)



Compares on bigger arrays (up to 100,000 elements)



Analysis of the graphs you produce.
As seen in all 4 graphs, for both large and small array sizes, quicksort consistently outperforms bubble and shell sort in efficiency by a huge margin. Whereas bubble and shell sort have what looks to be exponential time complexity, quicksort appears to be linear, but it’s graphical representation can barely be deciphered in comparison to bubble and shell sorts.
The major difference between the graphical representation of performing these sorts with small and large arrays is that for both moves and comparisons, bubble sort outperforms shell sort for arrays of smaller sizes. However, for arrays larger than 1000 elements, shell sort becomes increasingly more efficient than bubble sort. This shows us that for sorting smaller arrays bubble sort is more efficient, and for larger arrays shell sort is more efficient. However, quicksort seems to always outperform both of the others.