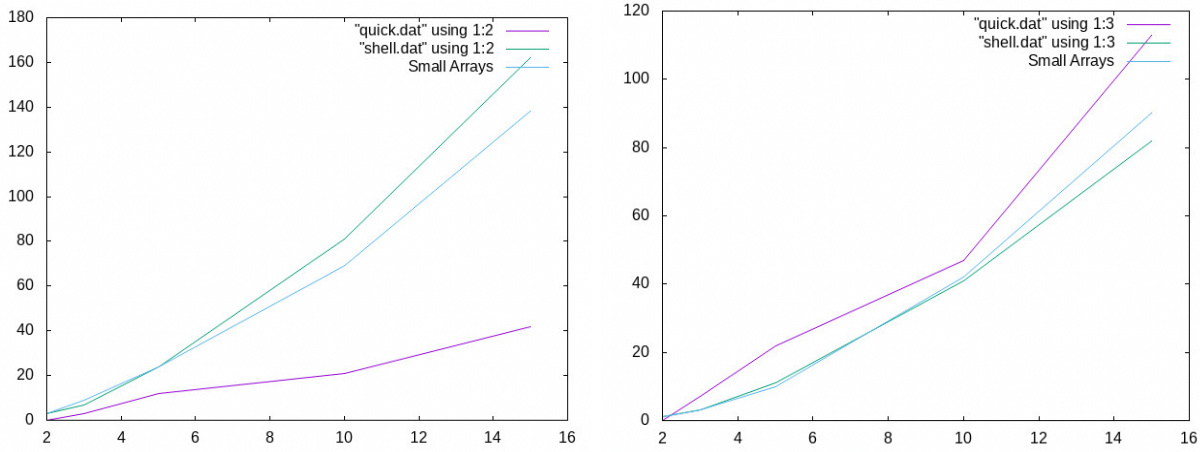
Fabrice Kurmann
fkurmann@ucsc.edu
CSE 13s Spring 2021

Writeup for Assignment 3, Sorting

In this assignment I learned to implement four different sorting algorithms and understand their various benefits and shortcomings. Additionally, I created a stack and a queue as an abstract data type for quicksort and was able to see which abstract data type is more suited to the task. Below I discuss my findings in a few different sections.

Overhead Optimization and constant values:

How long does it take for the increased overhead of quicksort or shellsort to be overturned by their better growth models? Here's what I found.



Left: Moves made by all three sorts. Right: Comparisons made by all three sorts.

The takeaway: overhead is not really a factor: Already when there are only 4 elements, the number of moves made by quicksort is considerably smaller than shellsort and bubblesort. Perhaps more surprising is that in moves, quicksort sets itself apart far more quickly than in comparisons, where it stays on track with the slower 2 algorithms right up through 15 elements.

Complexity Analysis:

Comparison of Sorting Algorithms's Move and Comparison Values:

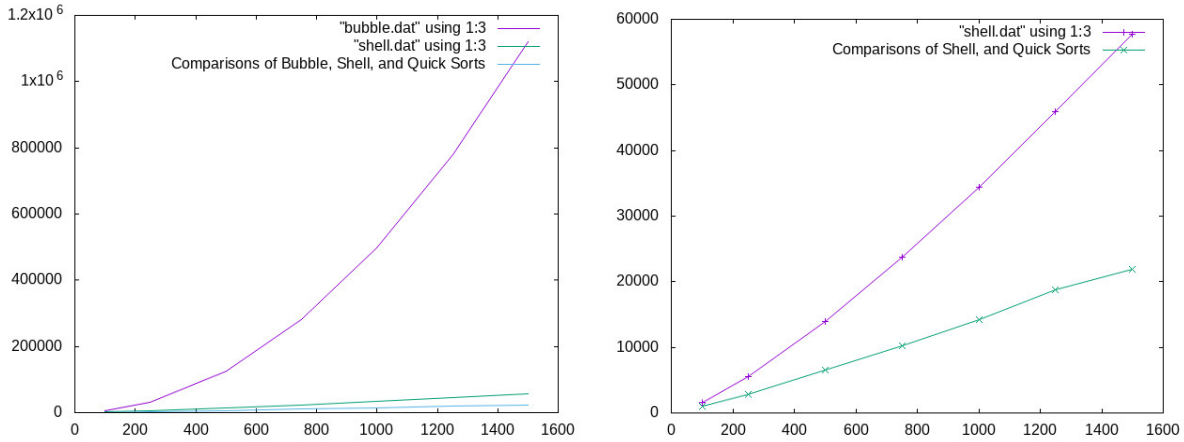
To compare these four sorting algorithms, I decided to run each of them with the same seed multiple times, changing only the size of the input array. My input arrays ranged in size from 100

elements to 1500, enough to see a considerable divergence in the number of comparisons and moves the different algorithms made.

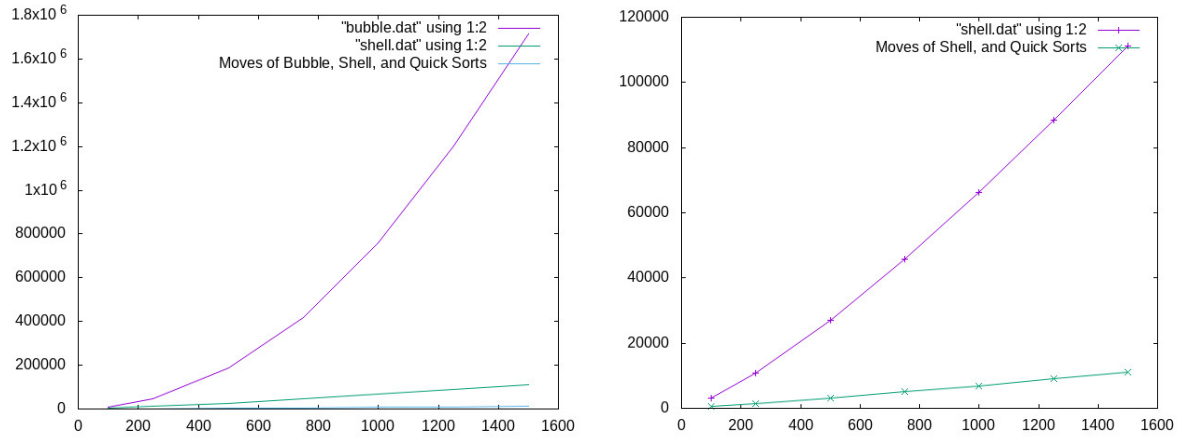
As it turns out, graphing bubblesort on the same graph as shell and quicksort is not a great idea. The number of comparisons and moves it makes simply dwarfs the other two sorting algorithms. Thus, I have provided two graphs for each of the comparisons and two for the moves - the second of each excluding bubblesort to allow a better understanding of shell and quicksort.

Comparisons of **bubblesort follow a very clear O(n^2) trajectory**, with growth speeding up as size of the array increases.

Bubblesort reaches 1.2 million comparisons and 1.8 million moves for 1500 items.

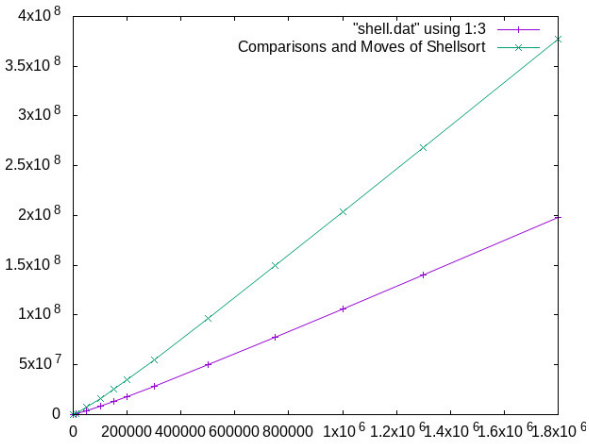


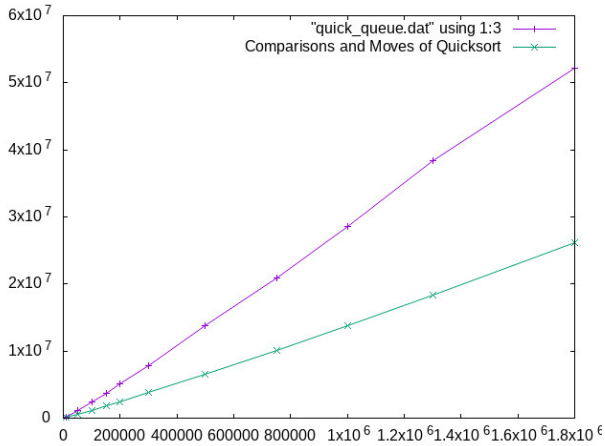
Left: Comparisons made by all three sorts. Right: Comparisons made by Shell/Quicksorts.



Left: Moves made by all three sorts. Right: Moves made by Shell/Quicksorts.

Below, I have also expanded the range of the shell and quicksort graphs since, unlike with bubblesort, those two algorithms require far more than 1500 to see a clear complexity trend. In fact, both the moves and comparisons graphs for both shell and quicksort look almost linear in this limited range. Let's see if by adding elements to the array this can be changed.





Left: Comparisons and moves for shellsort. Right: Comparisons and moves for quicksort.

As these above two graphs, which show the complexity behavior of quick and shellsort up to an array length of 1.8 million, both sorting algorithms continue to grow at a linear trend. Thus, assuming this particular seed is not an exceptional outlier, **the tested average computation complexity for both shellsort and quicksort is O(n).**

That said, same big O value does not mean same values:

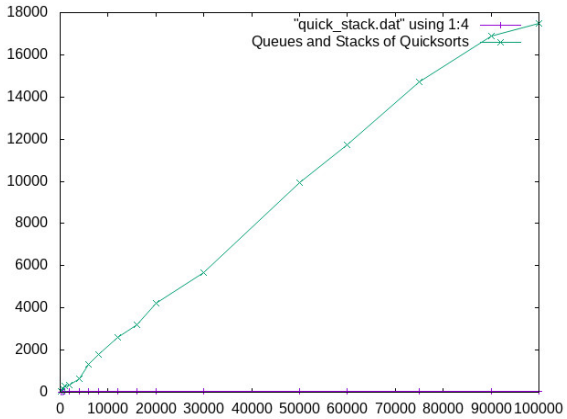
> **Shellsort** reaches 400 million moves and 200 million comparisons for 1.8 million items.
> **Quicksort** reaches 50 million comparisons and 30 million moves for 1.8 million items.

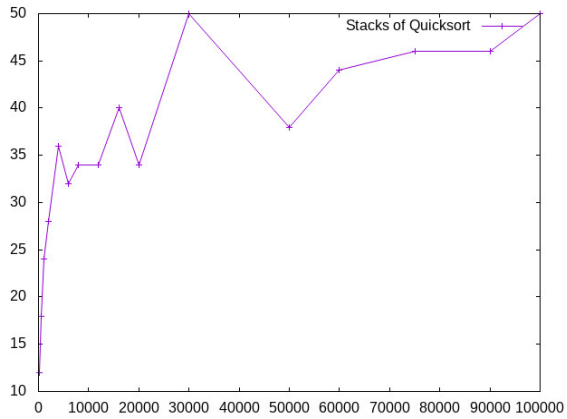
Quicksort - Stack vs Queue:

Next, I focused on comparing the maximum sizes of the stacks and queues allocated by my two quicksort algorithms. Graphing these two curves on the same graph showed just how much and how quickly stack and queue size diverges.

For an array of 100,000 elements, queue size grows to about 17,000 maximum elements, meanwhile the stack size never exceeds 50.

The difference between how large a stack and a queue gets is vast. However, it is somewhat easy to understand when analyzing how the quicksort function works. Initially it pushes items to the ADT, then it pops items right away. Based on this algorithm, it is no surprise that the LIFO method of the stack is more intuitive and thus requires less memory allocation.



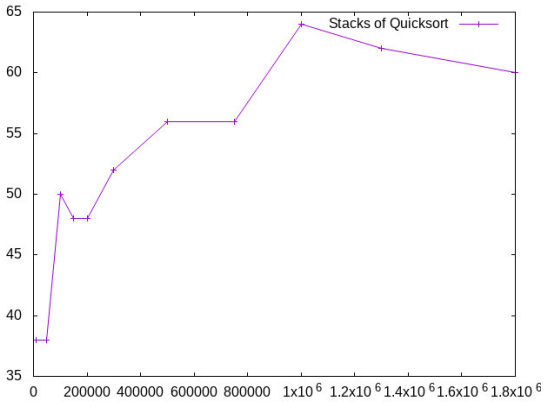


In fact, perhaps even more surprising than the differences in stack and queue sizes are the ways they grow as a function of array size:

Queue size grows in a near linear fashion, reaching 18,000 elements when there are 100,000 array elements. **Therefore, the growth of queue size also follows a complexity of O(n).**

Meanwhile, stack size very quickly increases from 12 elements for 100 items to 35 elements at 2000 items. However, from there onwards, it's graph becomes nearly horizontal, at times increasing then decreasing again, never exceeding 50 all the way through 100,000 elements.

Going further, increasing array size up to 1.8 million only raises stack size to a maximum of 64, which interestingly is not even reached when the input array is the longest. I am tempted to call such a growth model constant, however, am not sure what the ceiling constant is or indeed if there is a true ceiling. At 50 million items in quicksort, I am getting a maximum stack size of 76, and 74 for 2 different random seeds.



Experimentation with Time:

In addition to observing how the number of comparisons grows as array size increases, I wanted to see if the execution time follows the same growth pattern.

As I have already highlighted in the above section, for the shell and quicksort algorithms, the complexity seems to grow in a linear fashion. Using this I used shellsort to sort two massive arrays: one of length 3 million, 30 million and finally of length 300 million.

The reason I used shellsort was because it does not allocate a stack or queue, which I figured may interfere with the time to comparisons relationship since obviously allocating a large queue or stack also takes time that is not counted in the comparison or moves counters.

If my hypothesis about computational complexity is correct and applies to literal time as it does to comparisons and moves, then the 300 million array should take 10 times as long to sort as the 30 million array.

The results are as follows:

Shell Sort with huge arrays	Moves:	Comparisons:	Runtime on my MacBook Pro:
3 million	639182681 600 million	337182149 300 million	0m2.396s 2.3 sec
30 million	6947134236 7 billion	3889681036 4 billion	0m25.056s 25 sec
300 million	106000268780 106 billion	75401366366 75 billion	8m55.512s 535 sec

Interesting: My hypothesis about time and computations growing holds very well from 3 to 30 million, with all values increasing by a factor of 10. However, at 300 million, things diverge somewhat: comparisons increased 15 fold instead of 10 fold, and computation time increased 20 fold.

Given that a constant runtime O(n) is smaller than the upper limit of shellsort, which is quoted at O(n * log n^2), it is entirely possible that the curve in the graph where linear runtime is exceeded is simply not reached until above 30 million elements.