Govind Pillai
gopillai@ucsc.edu
4/25/2021

CSE 13s Spring 2021
Assignment 3: Sorting
Lab Writeup

## Time Complexities:

### Bubble Sort:

Worst: O($n^2$)

When the array is the reverse of the sorted array

Best: O(n)

When the array is already sorted

Average: O($n^2$)

### Shell Sort:

Worst: Depends on gap size

Best:O(nlogn)

When the array is already sorted

Average: O(nlogn)

### Quick Sort:

Worst: O($n^2$)

When the pivot selected is on either extreme of the elements (smallest or biggest element)

Best: O(nlogn)

When the middle element is always selected as the pivot

Average: O(nlogn)

## Methods for Data Collection

In order to graph these sorting algorithms, plot points were collected. Three different sorts were tested:

- Bubble Sort
- Shell Sort
- Quick Sort (w/Stack)

Each algorithm was then placed in a loop that iterated 1000 times. In this loop an array of a certain size was produced, sorted using the tested algorithm, and collected moves and comparisons data. These move and comparison values were then used to create average move and comparison values over 1000 iterations. This method was used for 10 sizes for each sort (100 elements, 200 elements, 300 elements,...,1000 elements). This gave 10 exact data points for each sorting algorithm which were then put on a graph.

## Results:

### Sorting Algorithms:

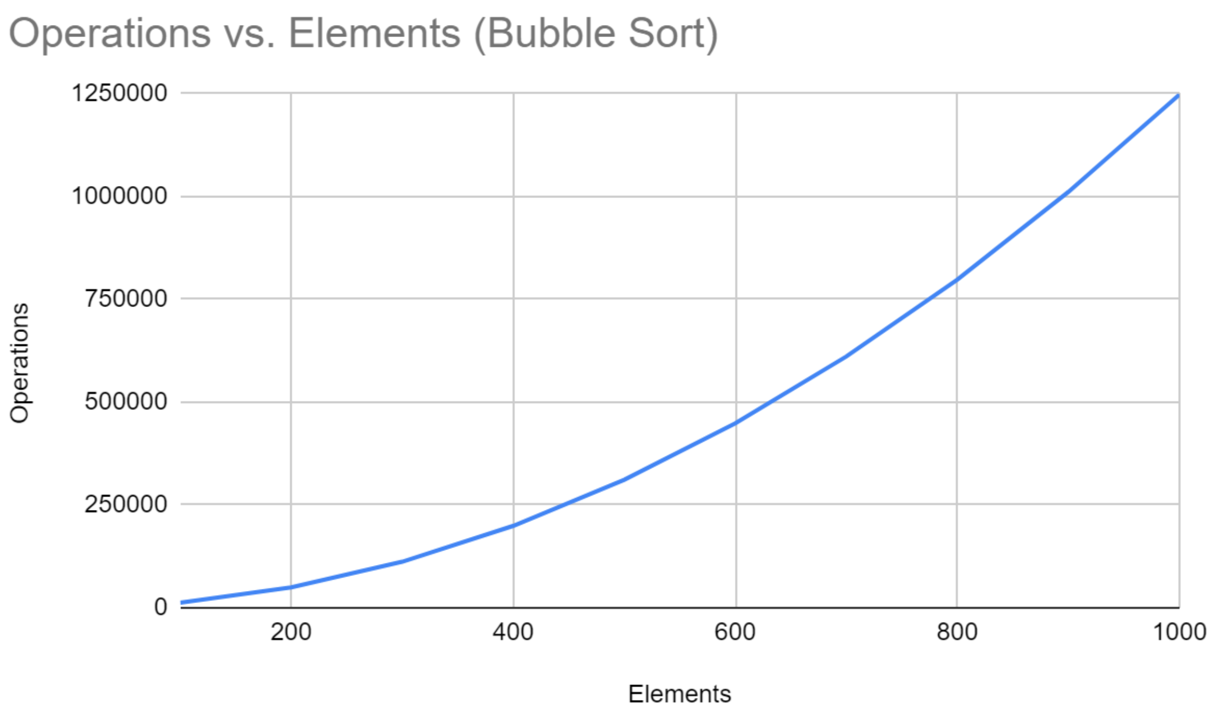
The method above was used to test all three sorting algorithms. Bubble sort was obviously the slowest of the bunch. With an average of O($n^2$) time complexity, the amount of operations were really high when sorting a 1000 element array. The Bubble sort graph below follows an O($n^2$) very similarly. Shell sort gave a drastic improvement in efficiency. Averaging O(nlogn), this sorting algorithm barely went over 100,000 operations for sorting 1000 elements compared to bubble sorts 1.25 million. The shell sort graph seems less curved upwards on the individual graphs, and its efficiency is shown clear on the group graph. Quick Sort was easily the most efficient of the bunch. Even though it experiences a worst case of O($n^2$) (bubble sort's average), selecting good pivot values kept Quick Sort from going into worst case territory. Instead, Quick Sort also experienced an average of O(nlogn). While both Shell and

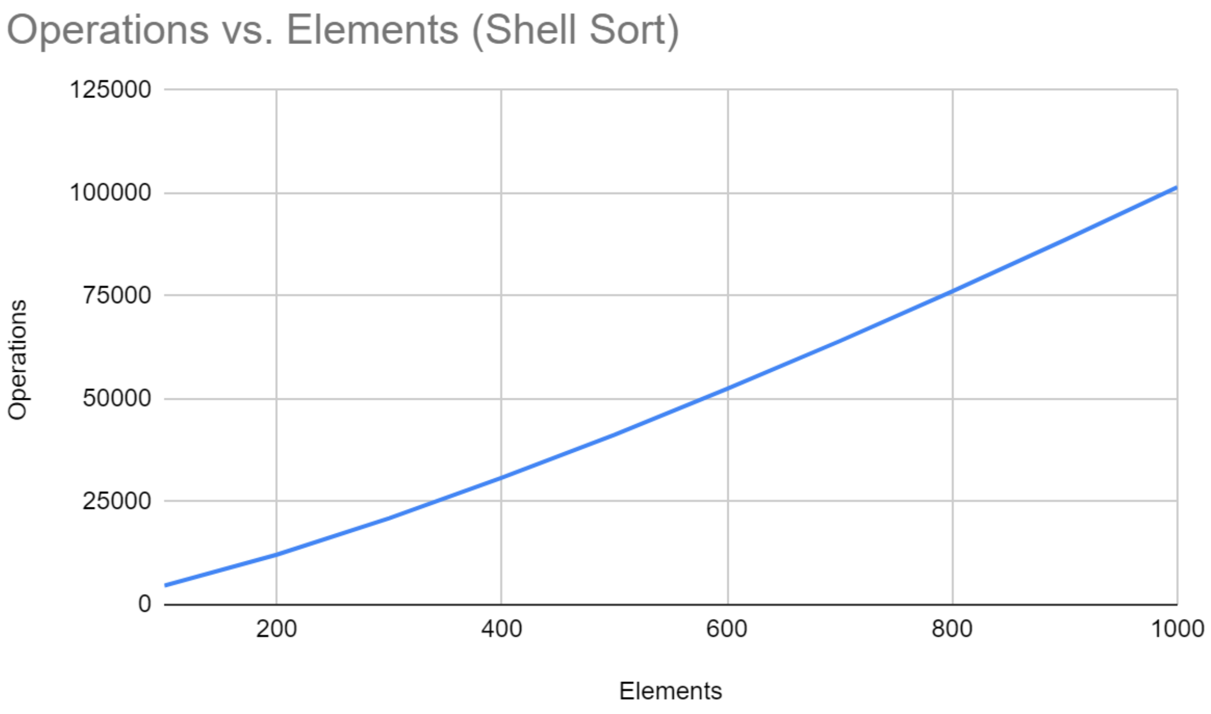
Quick sort featured roughly the same amount of comparisons, Quick used far less moves in order to sort the elements.

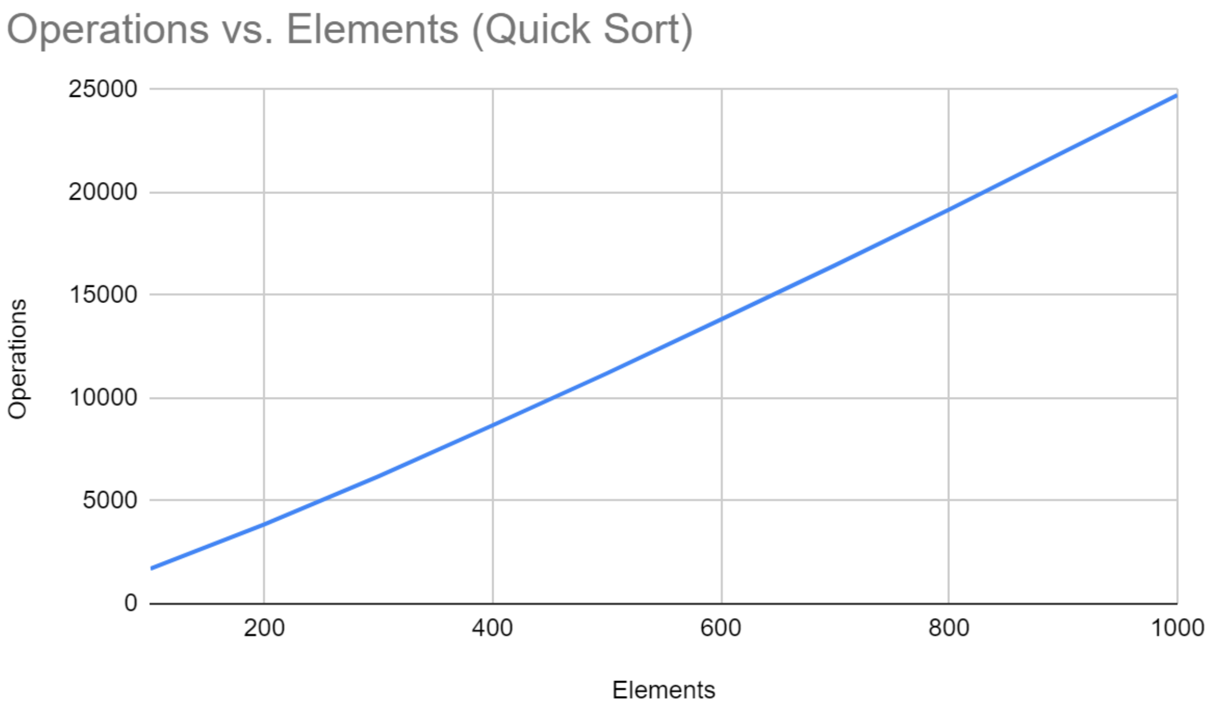
Stack and Queues:

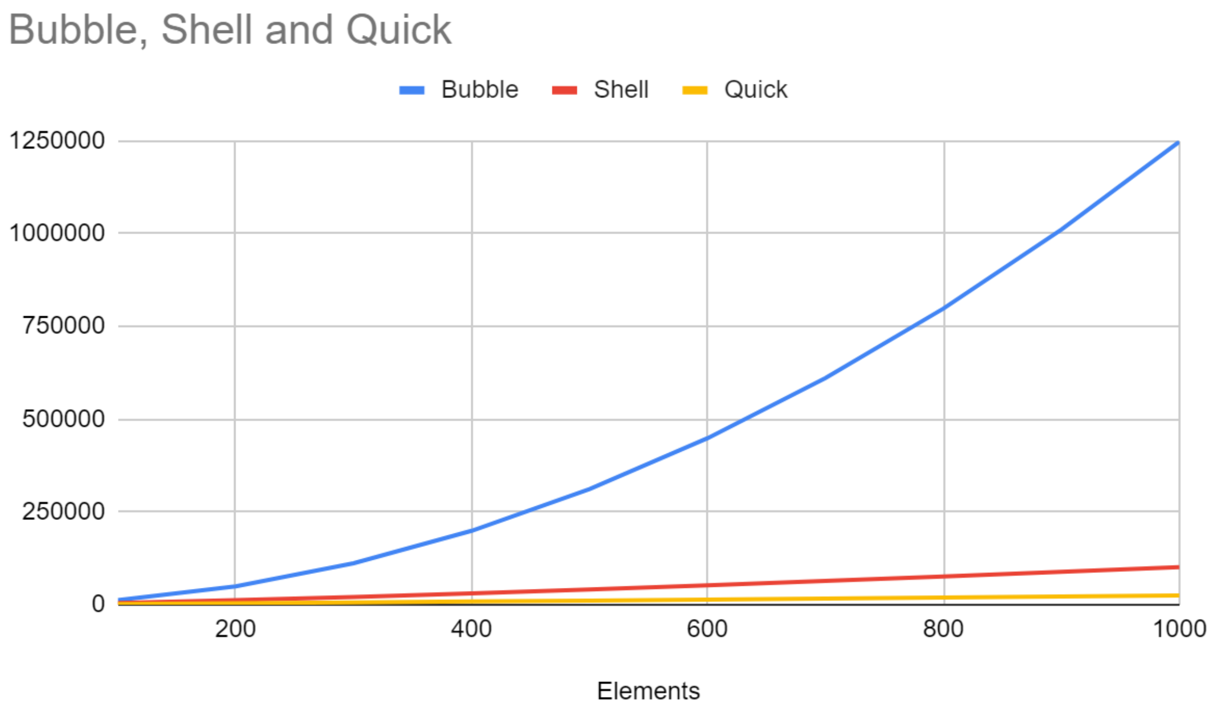
The Quick sort algorithm for Stack and Queues produced identical moves and comparison values when run. However, there is one interesting observation I made when using these two algorithms. The Queue used in the queue function always got way bigger than the stack used in the stack function. I am still not sure why this is but it must have something to do with the hi and lo values pushed onto the stacks and queues. This is because while both values are put on the stack/queue, they leave the stack/queue depending on which function is being used. This is because Stack uses LIFO while Queue uses FIFO.

## GRAPHS:




Operations vs. Elements (Shell Sort)
125000
100000
75000
50000
25000
0
Operations
200
400
600
800
1000
Elements


Operations vs. Elements (Quick Sort)
25000
20000
15000
10000
5000
0
Operations
200
400
600
800
1000
Elements


Bubble, Shell and Quick
Bubble
Shell
Quick
1250000
1000000
750000
500000
250000
0
200
400
600
800
1000
Elements