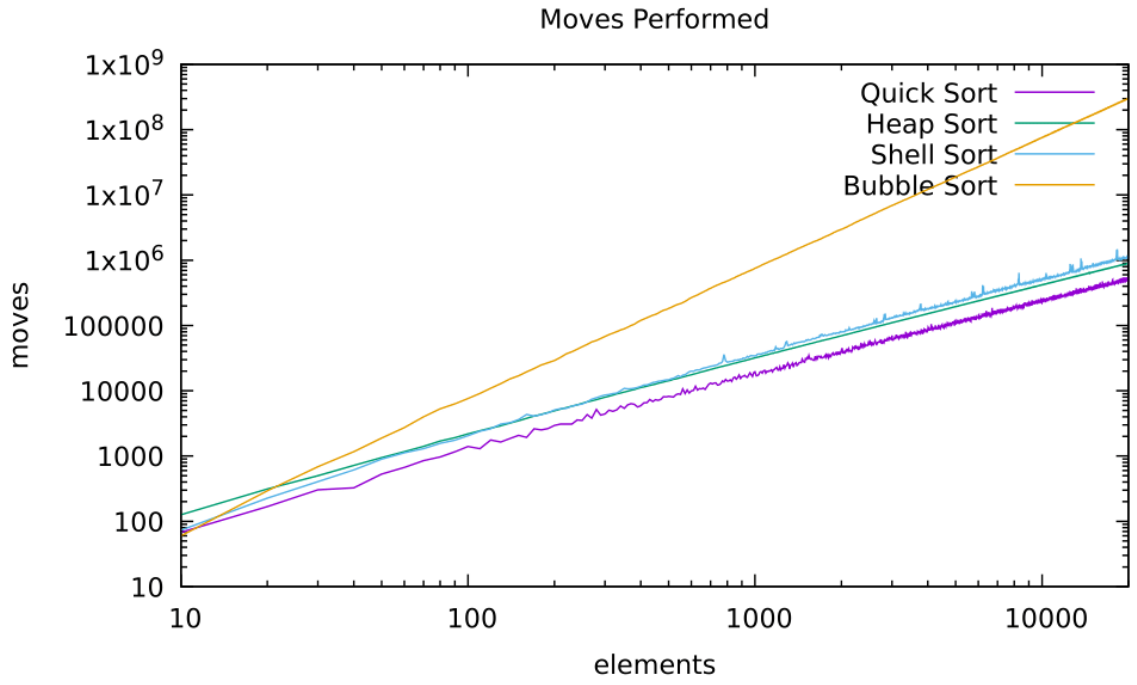
Isaac Flores
Prof. Miller
CSE13S
October 23, 2022

WRITEUP

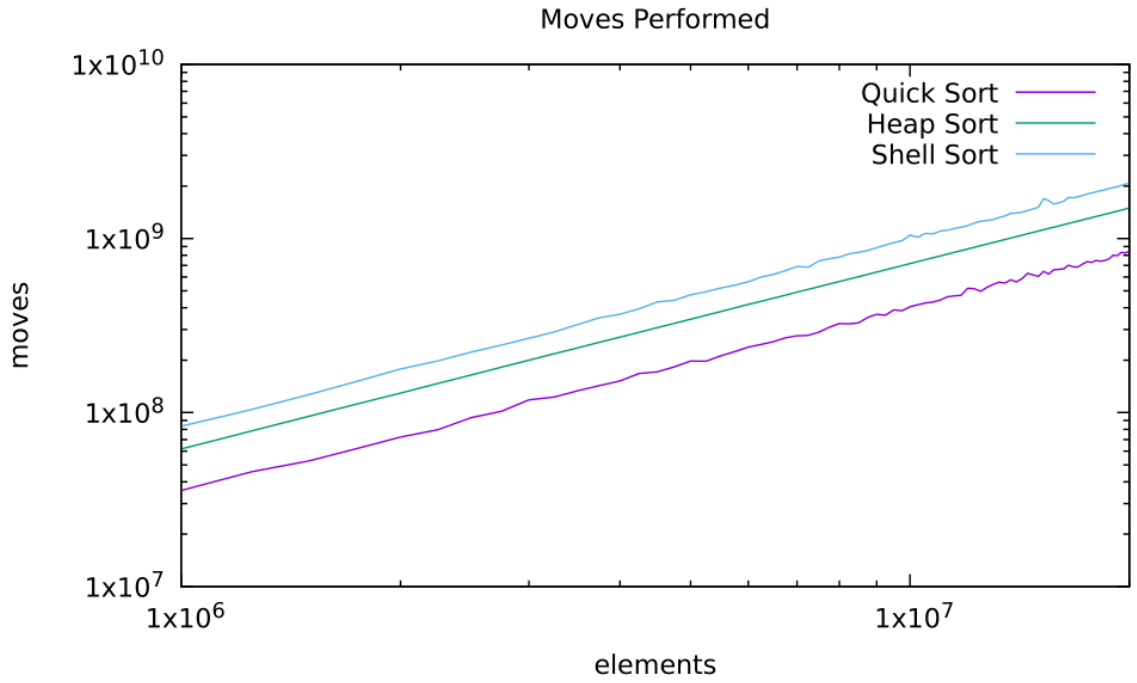
**Graphs:** Each graph is log-scaled and the seed I used for each graph was 13371453.

Small Number of Elements:




Analysis: For this graph, I checked the number of moves performed on each sorting method from 10 to 20,000 elements. From this data, I can tell that quick sort is the fastest method. Heap and shell are closely tied but heap is quicker. Bubble sort has a time complexity of elements$^2$ and it shows since it deviates from the other 3 sorting methods quickly. One feature I noticed is that the graphs of bubble sort and heap sort are smoother than the graphs of quick sort and shell sort. Quick sort may have bumps since I use an approximate middle value for the pivot. This pivot value may not be the actual average value of the elements which can cause more or less moves on different arrays. The bumps in shell sort may be caused by the gap sequence. Since the gap sequence is only dependent on the size of the original array it could cause extra moves than needed. If shell sort sorts the array before reaching the last gap number it will continue to sort. Shell only stops when a gap number of one is reached which could be a reason why the graph of shell sort isn't smooth.

Large Number of Elements:

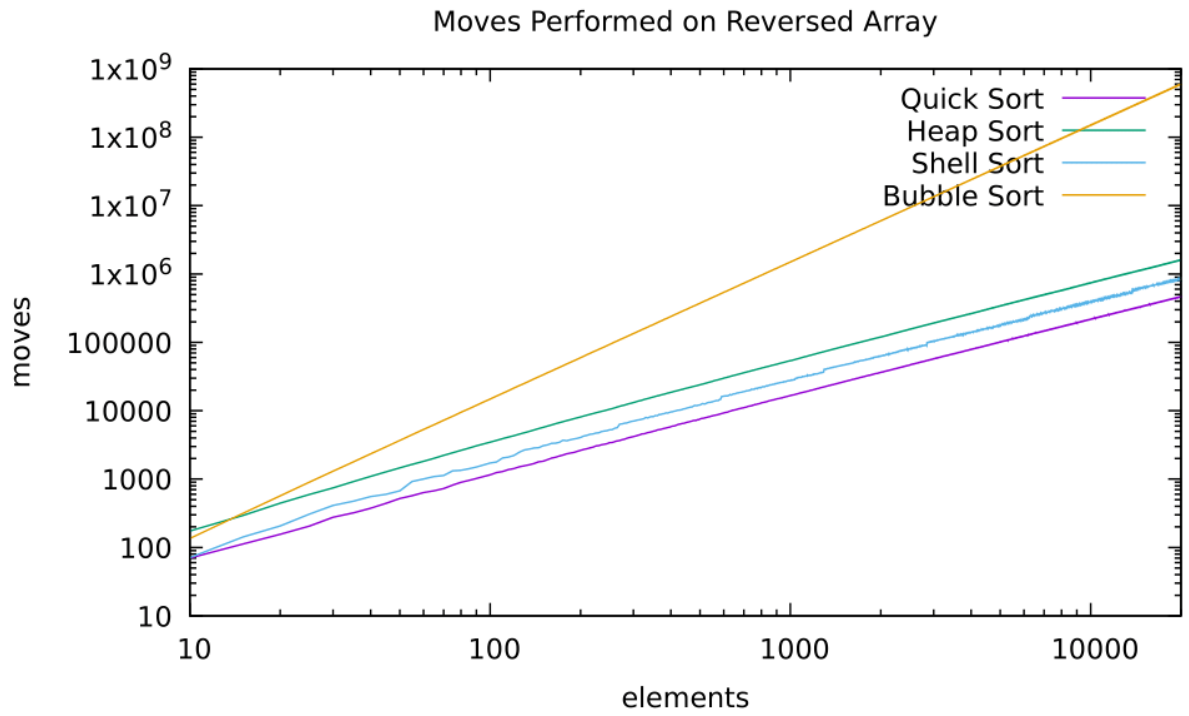



Analysis: For this graph, I check the number of moves for each sorting method from 1 million to 20 million elements. I did not include bubble sort since the number of moves deviated greatly from the other sorting methods. It also took a large amount of time to perform. From this data, I reached the same conclusion as the previous graph. Quick sort is the fastest method, then heap sort, then shell sort, then bubble sort.

Proof for bubble sort: As seen below, bubble sort has about 7 trillion moves for an array of 100,000 elements. For larger arrays bubble sort would take too long to complete which is why I chose to not include it. I attempted to bubble sort 1 million elements and after 2 minutes it still hadn't finished. Attempting to bash script bubble sort on large elements would take too long.

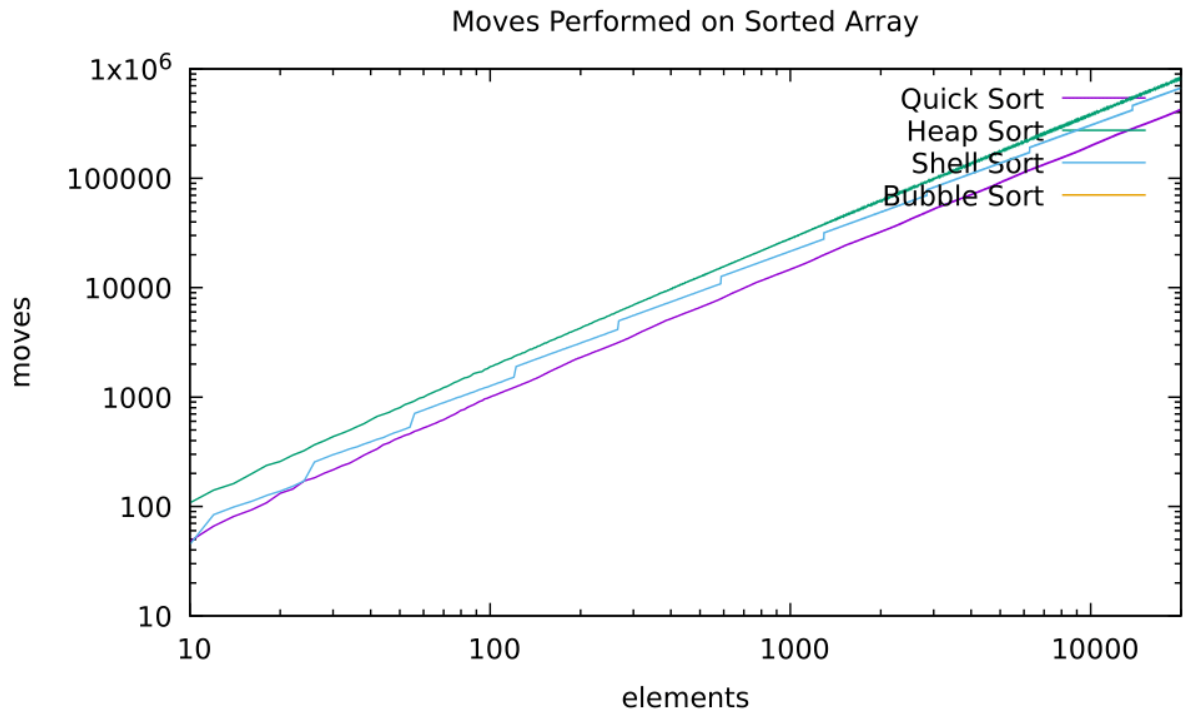
```
isaac@isaac-VirtualBox:~/cse13s/asgn4$ ./sorting -a -n 100000 -p 0
Bubble Sort, 100000 elements, 7486727772 moves, 4999933890 compares
Heap Sort, 100000 elements, 5185368 moves, 3059372 compares
Quick Sort, 100000 elements, 2986851 moves, 1725649 compares
Shell Sort, 100000 elements, 6563746 moves, 2573581 compares
```

Reverse Ordered Elements:




Analysis: For this graph, I sorted an array with each method from 10 to 20,000 elements. I set up the array so that the elements were sorted from greatest to smallest. The data confused me since now shell sort was quicker than heap sort and each graph was smoother. I noticed that shell sort remained about the same speed but heap sort got slower. I think the reason for this is that I used minimum heap sort. This means the top of the heap is the smallest value. Since the array was ordered from greatest to smallest this means that heap sort would have to perform more moves to get the smallest value to the top of the heap. The reason for each graph being smoother might be because there is a consistent fact that for each element, the element to the right of it will be smaller. This consistency may cause the graphs of quick sort and shell sort to be smoother and have fewer bumps.

Sorted Array:




Analysis: For this graph, I sorted an already sorted array with each method from 10 to 20,000 elements. Bubble sort always had 0 moves since bubble sort ends if no values have been sorted. Since the array is already sorted, bubble sort never moves any values. Shell sort is also quicker than heap sort for this case. The reasoning for this may be the fact that heap sort still needs to build the heap. This may be the cause for heap sort being slower than shell sort when sorting already sorted arrays.

**How I Got This Data:**

```
#!/bin/bash

for ((n = 10; n <= 20000; n += 10)); do  ./sorting -b -p 0 -n $n >> bubble.dat ; ./sorting -q -p 0 -n $n >> quick.dat ; ./sorting -h -p 0 -n $n >> heap.dat ; ./sorting -s -p 0 -n $n >> shell.dat ; done

gnuplot <<END
    set terminal pdf
    set output "moves for each sort algorithm.pdf"

    set xlabel "elements"
    set ylabel "moves"
    set logscale xy
    set zeroaxis
    set title "Moves Performed"
    plot "quick.dat" with lines title "Quick Sort", \
         "heap.dat" with lines title "Heap Sort", \
         "shell.dat" with lines title "Shell Sort", \
         "bubble.dat" with lines title "Bubble Sort"

END

rm quick.dat
rm heap.dat
rm shell.dat
rm bubble.dat

echo "Done plotting!"

```

Explanation: To get these graphs I used variations of this bash script by only changing the for loop. For the reverse order and sorted array, I changed my sorting.c file to build an array that was in reverse order or already sorted. The bash script above is what I used for the graph of a small number of elements. For a large number of elements, I started n at 1 million, ended it at 20 million, and incremented by 250,000.

**What I learned from the different sorting algorithms:**

Bubble Sort: From my findings, I can conclude that bubble sort is the worst sorting algorithm out of the four. Bubble sort performs worst on arrays with a large number of elements and can take multiple minutes on million-sized arrays. It only performs best when sorting an already sorted algorithm but that would be pointless. Bubble sort may perform well on arrays that are nearly sorted but it would be smarter to use any of the other three sorting algorithms. Since quick sort is built into the C library, I don't know why you would want to use bubble sort.

Quick Sort: From my findings, I can conclude that quick sort is the fastest sorting algorithm. From all of the cases I checked, quick sort always came out on top. Quick sort doesn't perform worse or less under certain conditions since it is reliable and consistent.

Heap Sort: From my findings, heap sort performs better than shell sort on randomly sorted arrays of varied sizes. On arrays that are in reverse order or already sorted, heap sort does perform worse than shell sort. I would say heap sort is overall better than shell sort since the sorting algorithm that can sort randomly sorted arrays faster is more useful.

Shell Sort: From my findings, shell sort performs worse than heap sort on randomly sorted arrays of varied sizes. Shell sort performed better than heap sort on already sorted arrays and reversed arrays. I would say shell sort is still useful since I used it in quick sort. Other than that, I would say heap sort is the better choice. Shell sort would also be useful for sorting a reversed array.