Assignment 3: “Sorting: Putting your affairs in order” WRITEUP

Vincent Liu

2/5/23

Introduction

Assignment 3, titled “Sorting: Putting your affairs in order”, introduces various sorting methods that are commonly known and often used by those in the field of computer science as well as computers themselves. These sorting methods include: Batcher sort, Shell sort, Heap sort, and Quick sort. Additionally, students will be creating a main() file to be compiled as an executable and run to test the student implemented code. Students are also instructed to implement sets operations for use in the main file. Some files required for this assignment have been provided, such as the ‘.h’ header files for each sorting method, as well as header files such as set.h and stats.h.

Shellsort

Shell sort is a sorting algorithm used on arrays that was first published by Donald Shell in 1959. Shell sort uses a method similar to insertion sort, however its method first pairs elements close to each other before then reducing the gap between all elements. This algorithm is highly dependent on gap sequences, which are partitions of elements in an array. For example, in a 100 element array, the first gap would be 50, then 25, and so on. Given that arrays can be quite large in an actual workspace, Shellsort may not be the most efficient or optimal sorting method. The worst case performances of Shellsort are:

$$O(n * log^2 * n) \text{ and } O(n^2)$$

While the best case performance of Shellsort are:

$$O(n * log * n)\text{and } O(n * log^2 * n)$$

*It must be noted that the latter is also a best case performance because the algorithm highly depends on gap sequence, however most gap sequences result in the former. The average performance of this algorithm is dependent on the gap sequence as well.
The code containing the Shellsort method is ‘shell.c’ and is accompanied by a header file titled ‘shell.h’.

Heapsort

Heapsort is a sorting algorithm used on arrays, developed in 1964 by J. W. J. Williams. It is said to be a comparison-based sorting algorithm. In essence, this means that Heapsort uses comparisons to determine the next element to be sorted. Heap sort uses a sorted region towards the right, and an unsorted region towards the left. What occurs when the algorithm runs is that the highest/largest valued element in the unsorted region is taken and placed to the right, closest to the sorted region. Said element is then considered to be 'sorted', and the algorithm reaches back into the unsorted region to essentially take the next largest element and place it in the sorted region. In doing so, this algorithm works to sort elements from largest to smallest, right to left. The best case and average performance of this algorithm is:

$$O(n * log * n)$$

While the worst case performances of this algorithms are:

$$O(n * log * n)$$
$$O(n)$$

*The former worst case is when there are distinct keys. Interestingly this formula applies to the worst, best and average case performances.
*The latter worst case is if there are equal keys

The code containing the Heapsort method is "heap.c" and is accompanied by a header file titled 'heap.h'.

Batcher's Odd Even Merge Sort

The Batcher Odd-Even merge sort, also known as Batcher's method, was developed by Ken Batcher for sorting networks. This sorting method uses comparators to compare two values and swap the values if necessary. In this assignment it is stated that "sorting networks are typically limited to inputs that are powers of 2", and that this assignment applies "...Knuth's modification to Batcher's method to allow it to sort arbitrary-size inputs", otherwise known as the 'Merge Exchange Sort'. The best case, worst case, and average performance of this algorithm is:

$$O(log^2(n))$$

The code containing the Batcher sort method is "batcher.c" and is accompanied by a header file titled 'batcher.h'.

Quicksort

Quicksort is a commonly used sorting method developed by Tony Hoare in 1959. It is one of the most if not *the fastest* sorting algorithms to be used. It is said to be two to three times faster than Mergesort and Heapsort. This method uses a "divide and conquer" algorithm, in which an element from the array is used as a pivot for sub-arrays that then each recursively organize items in their respective regions (left and right of pivot element). Quicksort is also an *in-place* algorithm, which means it does not require allocation of additional memory for the utilized sub-arrays. Additionally, quicksort is a comparison sort, meaning that it utilizes the "less than" relation between elements to help in sorting. The best case performances of this algorithm is:

$$O(n^2)$$

Which also is the average performance of this algorithm. The worst case performances of this algorithm are:

$$O(n * log * n) \text{ or } O(n)$$

*The former worst case applies when it is a simple partition.

The code containing the Quicksort method is "quick.c" and is accompanied by a header file titled 'quick.h'.

UNIX commands used

- ./sorting: Used to run the main file. Expects a passed argument. Arguments are as follows:
  - -a : Employs all sorting algorithms.
  - -h : Enables Heap Sort.
  - -b : Enables Batcher Sort.
  - -s : Enables Shell Sort.
  - -q : Enables Quicksort.
  - -r seed : Set the random seed to seed. The default seed should be 13371453.
  - -n size : Set the array size to size. The default size should be 100.
  - -p elements: Print out the number of elements from the array. The default number of elements to print out should be 100.
  - -H : Prints out program usage.
- make / make all : Used to create executable files as well as compile the necessary '.c' files into objects files (".o" files)
- make clean : Used to remove the executable main file as well as remove the associated object files
- make format : Used to properly format all '.c' files.

Data of Each Sorting Method

***All data output uses randomized array with seed 13371453**

shell.c

My program's output for the Shell sorting method was 3025 moves and 1575 compares for 100 elements. When testing this sorting method with 1000 elements, Shell sort performed 66,253 moves and 34,407 comparisons.

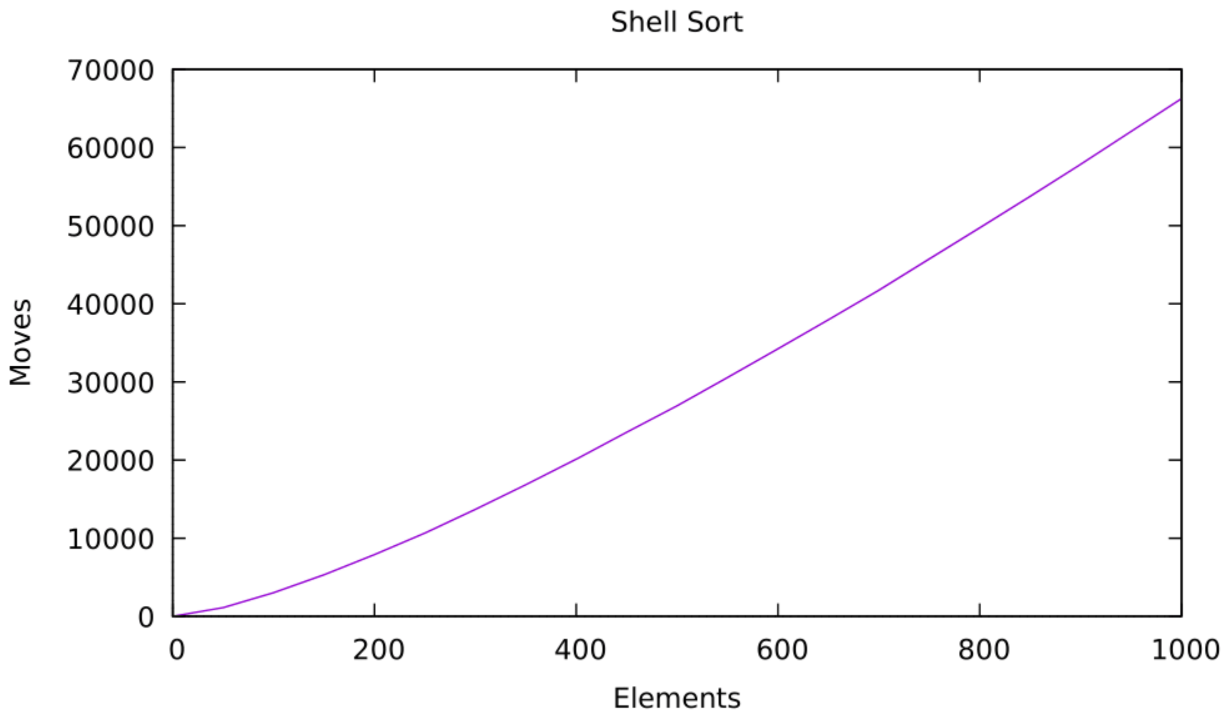

Figure 1

In this graph we are able to visualize the estimated number of moves done by shell sort per element. I say estimated because the points plotted for the line of the graph use +50 element increments (ex. 50 elements = some number of moves, 100 elements = some number of moves, 150, 200, 250, … , 1000). One can easily notice that the line becomes nearly linear beginning from around 400 elements and onwards. We can hypothesize that this is because of the fact that the algorithm's performance is most likely set at $O(n * log^2 * n)$, as the number of elements increases up to 1000. Another thing to note is that the number of moves done here approaches 70,000 (66,253 to be exact) as the algorithm attempts to sort 1000 elements. In short, Shellsort is one of the least efficient sorting algorithms amongst the four presented in this assignment.

batcher.c

My program's output for Batcher's sorting method was 1209 moves and 1077 compares for 100 elements. When testing this sorting method with 1000 elements, Batcher's method performed 22,497 moves and 23,449 comparisons.

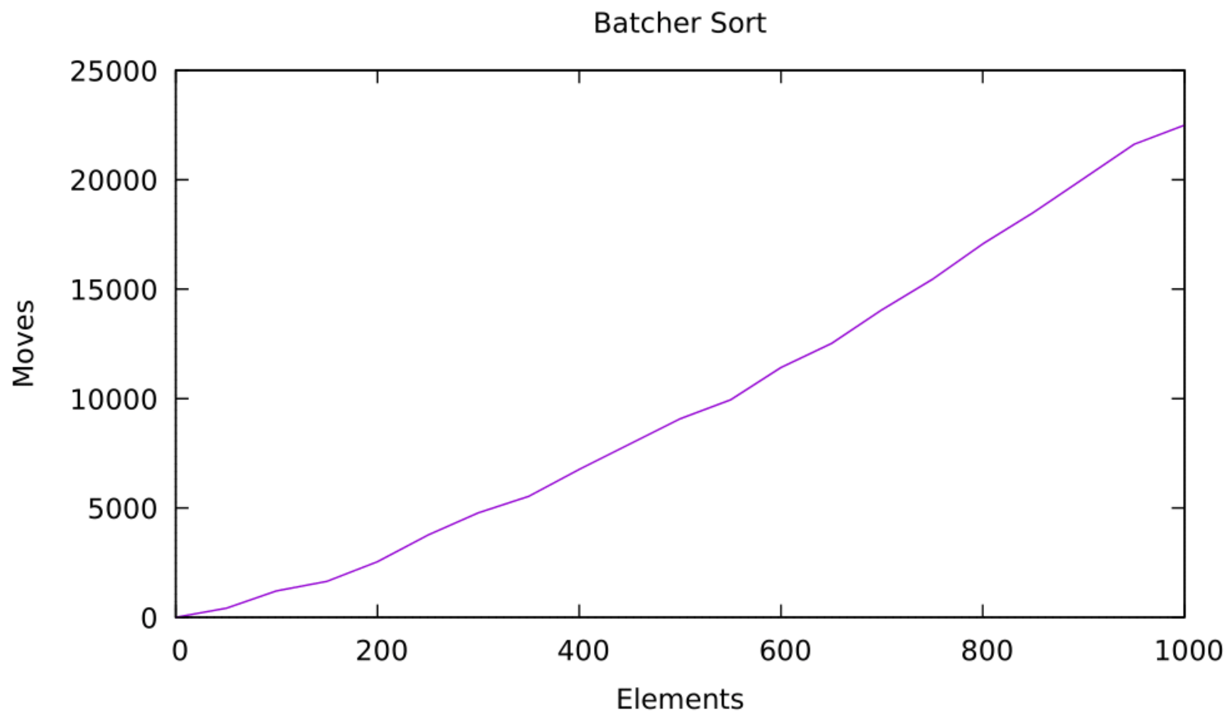

Figure 2

In this graph we are able to visualize the estimated number of moves done by Batcher sort for a set number of elements. For clarity, as with all other graphs on this writeup the points plotted for the line of the graph use +50 element increments (ex. 50 elements = some number of moves, 100 elements = some number of moves, 150, 200, 250, … , 1000). We can notice that the line does not become fully linear, though between some points it does, such as between the 700 and 900 element marks. As the algorithm attempts to sort 1000 elements, we can see that it completes approximately 23,000 moves (22,497 to be exact). According to the data collected, Batcher sort has the third highest number of moves, but the second most number of comparisons amongst the four sorting algorithms/methods in this assignment.

heap.c

My program's output for the shell sorting method was 1755 moves and 1029 compares for 100 elements. When testing this sorting method with 1000 elements, Heapsort performed 27,225 moves and 16,818 comparisons.

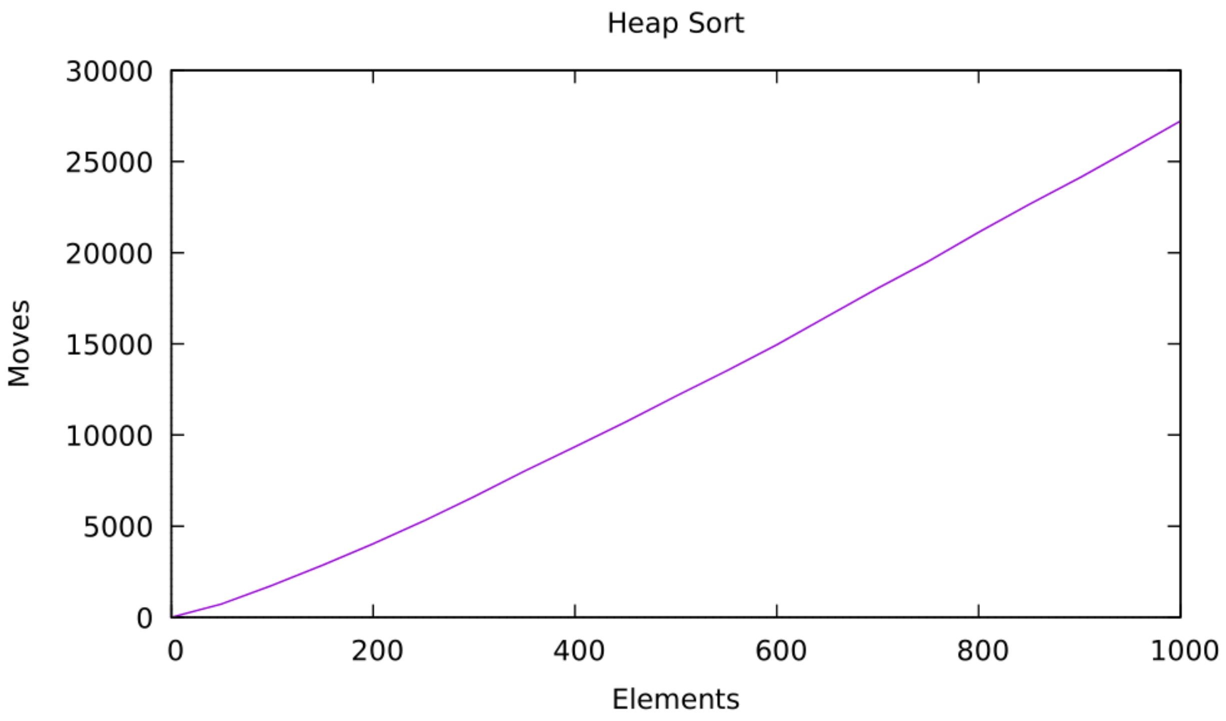

Figure 3

In this graph we are able to visualize the number of moves done by Heapsort for a set number of elements. As with all other graphs on this writeup the points plotted for the lines of the graphs use +50 element increments (ex. 50 elements=some number of moves, 100 elements=some number of moves, 150, 200, 250, … , 1000 elements). We can notice that the line almost does become fully linear when calculating above 200 elements. As the algorithm attempts to sort 1000 elements, we can see that it completes approximately 27,000 moves (27,225 to be exact); the second highest number of moves for 1000 elements amongst the four sorting methods/algorithms presented in this assignment.

quick.c

My program's output for the shell sorting method was 1053 moves and 640 compares for 100 elements. When testing this sorting method with 1000 elements, Quicksort performed 18,642 moves and 10,531 comparisons.

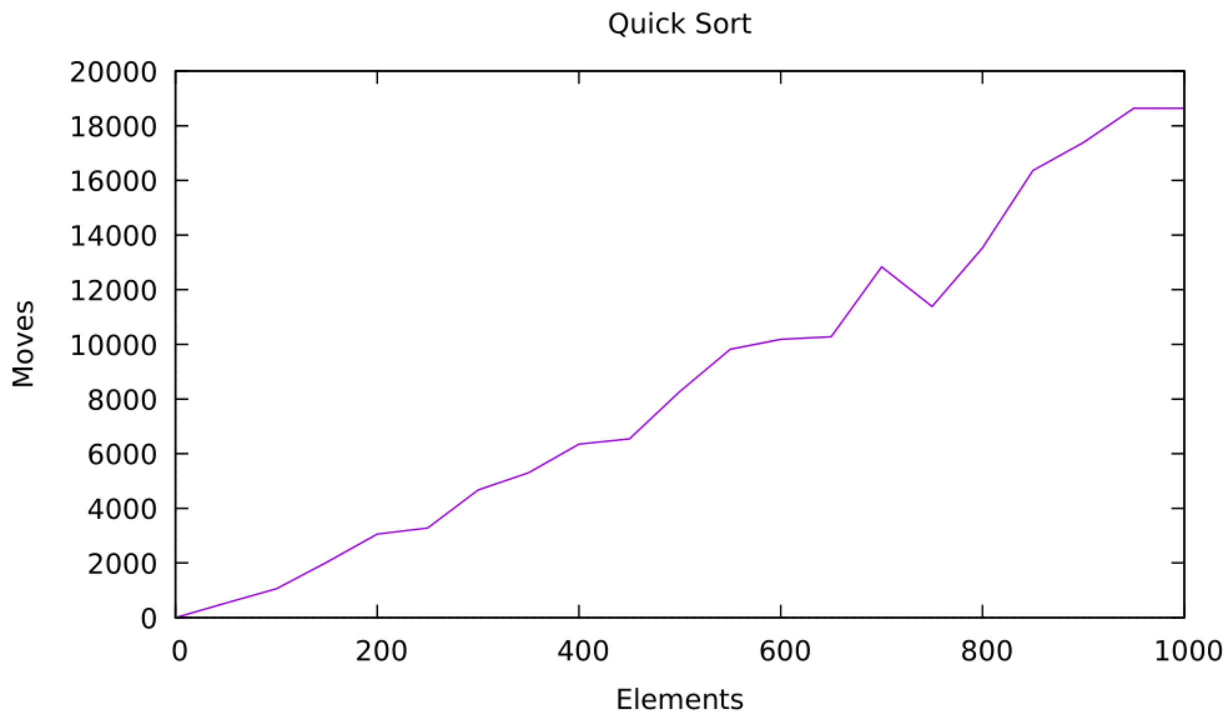

Figure 4
In this graph we are able to visualize the number of moves done by Quicksort for a set number of elements. As with all other graphs on this writeup the points plotted for the lines of the graphs use +50 element increments (ex. 50 elements=some number of moves, 100 elements=some number of moves, 150, 200, 250, … , 1000 elements). We can notice that the line does not become fully linear at any given point, and in fact varies quite a bit between calculating various numbers of elements. Something interesting to note is the fact that the number of moves plateaus when calculating more than 950 elements. As the algorithm attempts to sort 1000 elements, we can see that it completes around 19,000 moves (18,642 to be exact). According to data collected, Quicksort has the lowest number of completed moves as well as the lowest number of comparisons of any given number of elements amongst the four sorting methods/algorithms presented in this assignment.

Number of moves between all four sorting algorithms

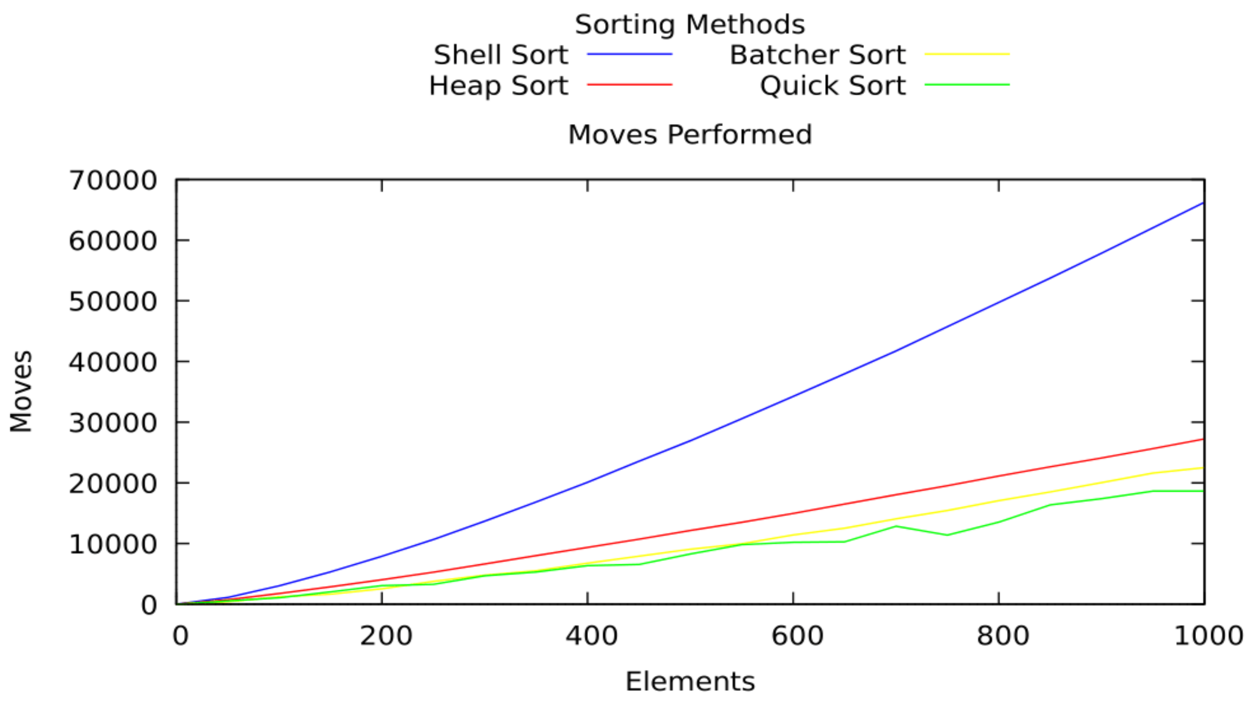

Figure 5
In this graph we are able to see data collected from the previous four graphs of the total number of moves completed, all put into a single visual. For clarity, the lower the number of moves completed the better, as it can be argued that a more efficient sorting method does less moves. As can be seen, Shell sort completes the most amount of moves for any given number of elements, followed by Heapsort, with a large gap between the two. Then there is Batcher sort, as well as Quicksort following closely behind. In case anyone still isn't convinced, this visual shows how inefficient Shell sort is in comparison to its competitors, both in the real world, as well as within this assignment.

Number of comparisons between all four sorting algorithms

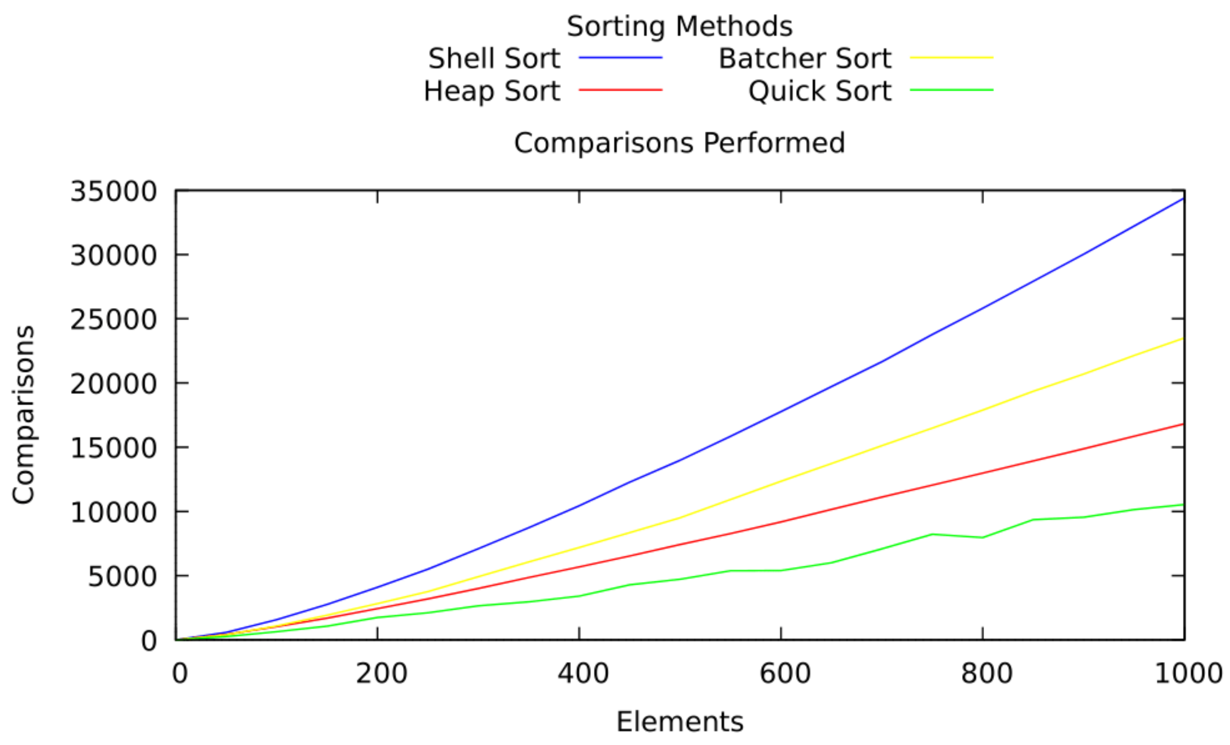

Figure 6
In this graph we are able to see data collected of the total number of comparisons from the four sorting methods. Similar to the number of moves done, the lower the number of comparisons completed, the better. As can be seen, Shell sort does the most number of comparisons for any given number of elements, followed by Batcher sort, Heapsort, and Quicksort. Much like what figure 5 shows, this visual shows how inefficient Shell sort is in comparison to the other sorting methods/algorithms. Not only does shell sort complete the most amount of moves no matter the number of elements, but it also does the most amount of comparisons when sorting, no matter the number of elements.

Other associated files in the directory

quick.h
Header file that specifies the interface to quick.c. Was provided in resources repo

heap.h
Header file that specifies the interface to heap.c. Was provided in resources repo

shell.h
Header file that specifies the interface to shell.c. Was provided in resources repo

batcher.h
Header file that specifies the interface to batcher.c. Was provided in resources repo

stats.c
C program that implements the statistics module. Was provided in resources repo

stats.h
Header file that specifies the interface to the statistics module.Was provided in resources repo

set.c
C program that implements bit-wise Set operations. Used in main file and was student-made

set.h
Header file that implements and specifies the interface for the set ADT. Was provided in resources repo

Makefile
Used to compile all '.c' files into '.o' object files, create an executable file, or remove all object files and executables from the directory.

sorting.c
Is the main file and is compiled into an executable. Its role as a main file is to 'weave' all .c files into a single program that can use all of the following arguments when properly executed from the terminal:

- -a : Employs all sorting algorithms.
- -h : Enables Heap Sort.
- -b : Enables Batcher Sort.
- -s : Enables Shell Sort.

-q : Enables Quicksort.
-r seed : Set the random seed to seed. The default seed should be 13371453.
-n size : Set the array size to size. The default size should be 100.
-p elements: Print out the number of elements from the array. The default number of elements to print out should be 100.
-H : Prints out program usage.

Findings & What I learned

From this assignment, I learned about five sorting methods: Insertion sort, Shell sort, Heap sort, Quick sort, and Batcher's Odd-Even merge sort. The latter four of which were actually programmed for this assignment. Additionally I discovered that out of all four programmed sorting methods, Quicksort was one of the most if not *the* most efficient sorting method. It had the least amount of moves and comparisons when compared to the other sorting algorithms. In terms of total moves made, Batcher Sort came in second, followed by Heapsort, then Shell sort. In terms of comparisons done, Heapsort came in second, followed by Batcher's method, then Shellsort. I found it interesting how Batcher's method had more moves yet fewer comparisons than Heapsort. Though I cannot exactly say whether the number of comparisons done correlates to the efficiency of a sorting algorithm, I can say that it was an interesting find. I also found programming set.c to be quite beneficial to learning and understanding bitwise operations in the C language. Additionally, reading the other files in the directory, especially 'stats.h' and 'stats.c' helped broaden my understanding of the possibility of structures in C.
