Assignment 3 - An Assortment of Sorts

Angela Shen

February 2023

1 Abstract

Sorting is an essential tool for computer science, since it makes many processes more efficient and creates new options for searching for items (like binary search). In this assignment, several sorts are implemented and compared to demonstrate the differences in efficiency.

2 Introduction

Assignment 3 contains the following deliverables:

Makefile - Compiles an executable of and creates multiple object files (mathlib-test's dependencies). For a list of these c files, .c below.

README.md - Details the process of building any necessary files, the command line options for any executables, and any errors or bugs.

DESIGN.pdf - Contains the pseudo code and descriptions of each c file.

WRITEUP.pdf - This document. Describes the assignment in its entirety and discusses the results.

sorting.c - The test harness for the other c files. Allows the user to test shell, heap, batcher, and quick sort using command line arguments.

batcher.c - Implements Batcher Sort. Use -b in sorting.c to test this sort.

batcher.h - The header file for batcher.c

gaps.h - Provides an array of gaps to be used by Shell sort.

shell.c - Implements Shell Sort. Use -s in sorting.c to test this sort.

shell.h The header file for shell.c.

heap.c - Implements Heap Sort. Use -h in sorting.c to test this sort.

heap.h - The header file for heap.c.

quick.c - Implements recursive Quicksort. Use -q in sorting.c to test this sort.

quick.h - The header file for quick.c.

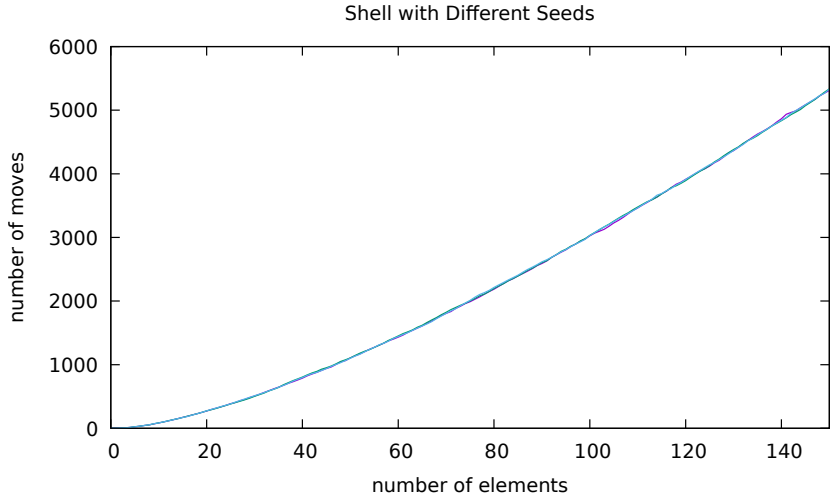
set.c - Implements bit-wise Set operations.

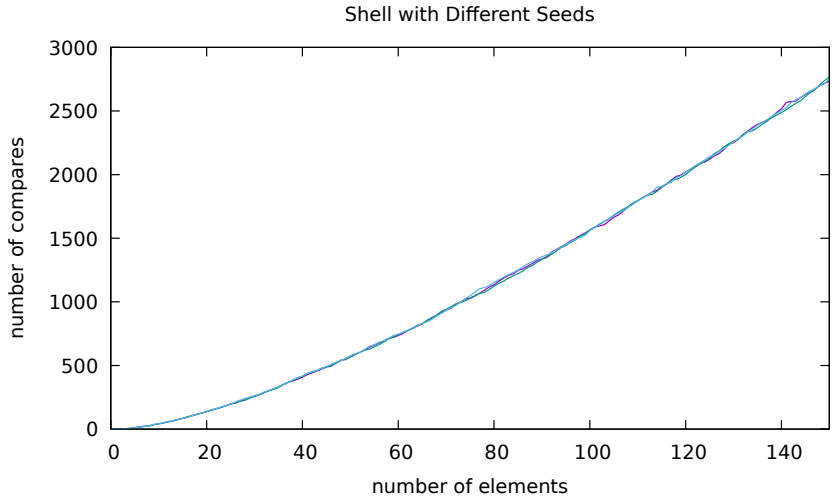
set.h - The header file for set.c.

stats.c - Implements the statistics module.

stats.h - The header file for the statistics module.

3 Shell Sort

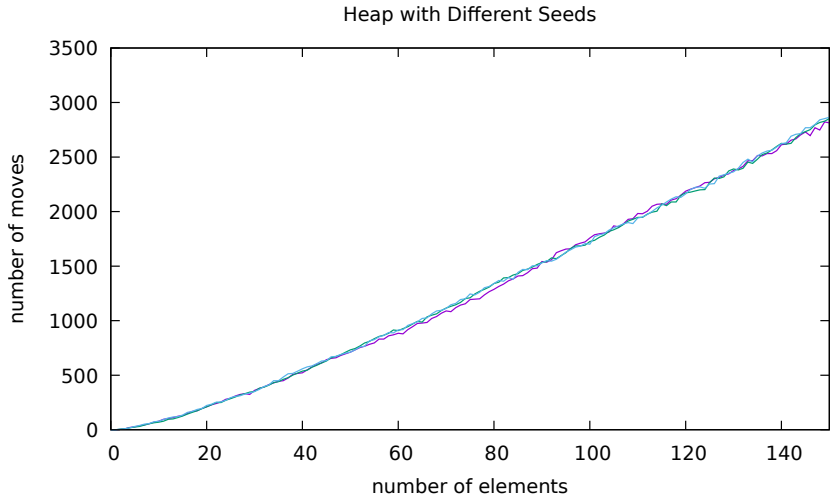


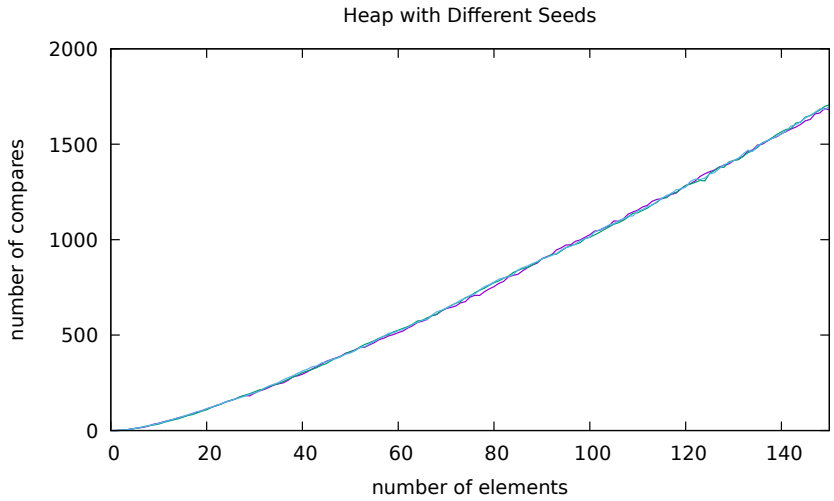


The above graph shows Shell Sort's efficiency with sorting 3 randomized arrays of different sizes (1 - 150 elements).

Shell seems to be the most consistent sort, since even at a relatively low number of array elements it is hard to see the difference between the number of sorts performed.

4 Heap Sort

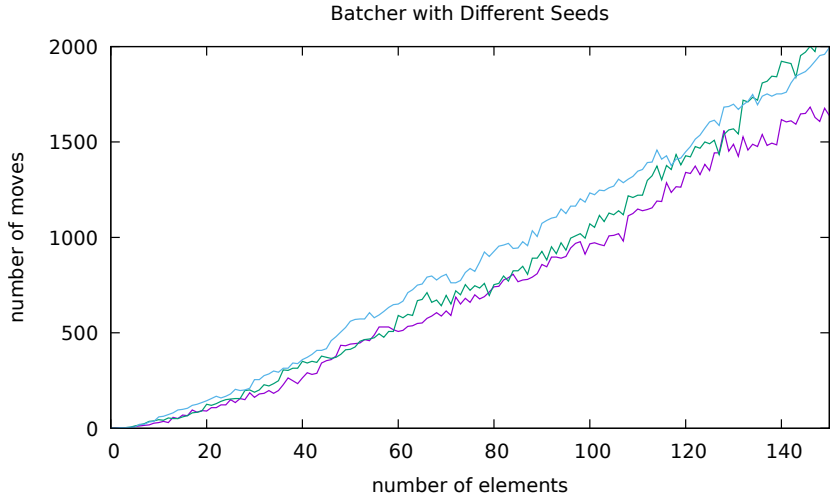


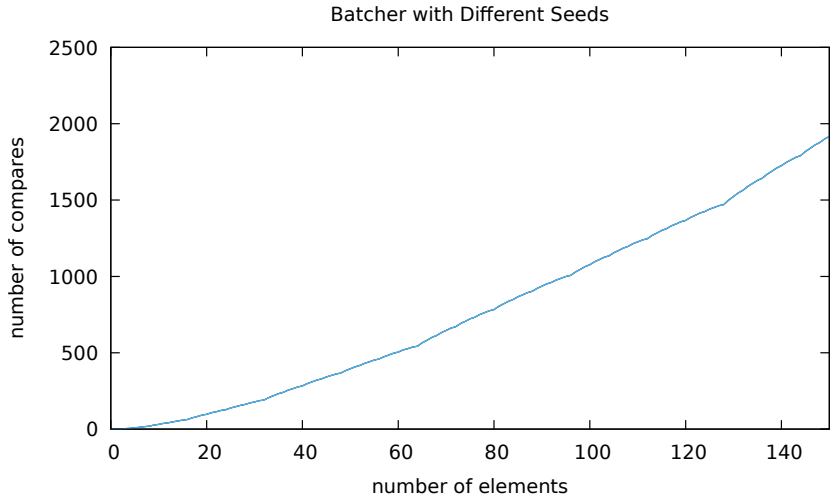


The above graph shows Shell Sort's efficiency with sorting 3 randomized arrays of different sizes (1 - 150 elements).

Heap is also a pretty consistent sort, with very few visible differences between the different seeds.

5 Batcher Sort

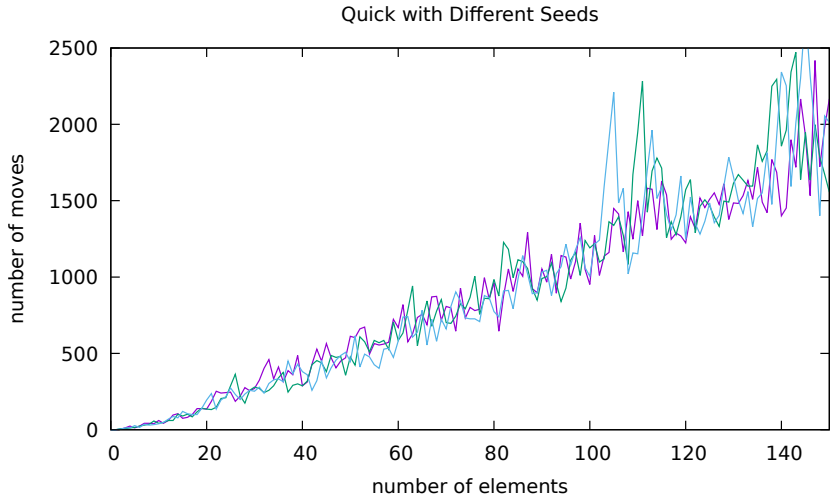


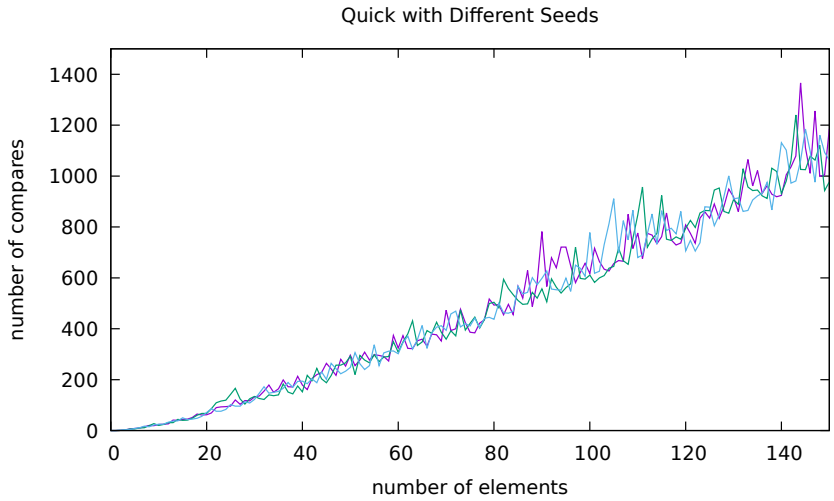


The above graph shows Batcher Sort's efficiency with sorting 3 randomized arrays of different sizes (1 - 150 elements).

The number of moves Batcher does is less consistent than the above sorts, but interestingly, its compares are entirely consistent (which is seen in the provided resources binary). This appears to be because Batcher comparisons are determined by bitwise shifts that use the length of the array, so for every array of a specific length, Batcher sort will always do the same number of comparisons.

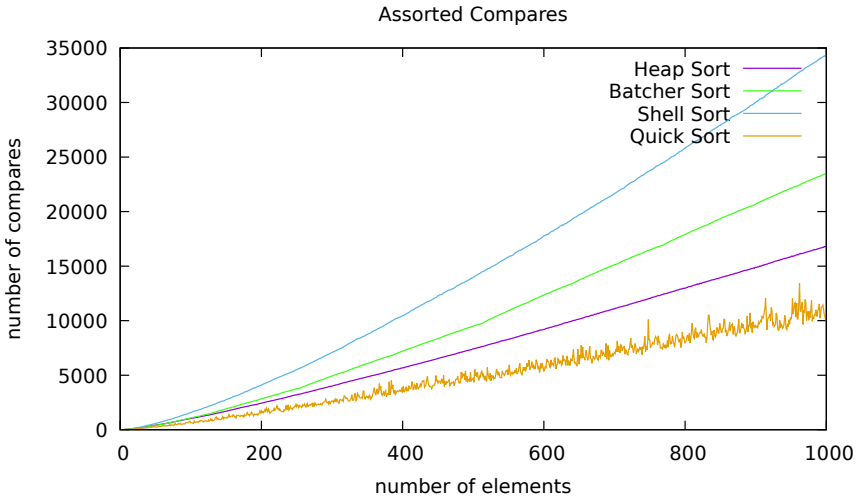
6 Quick Sort

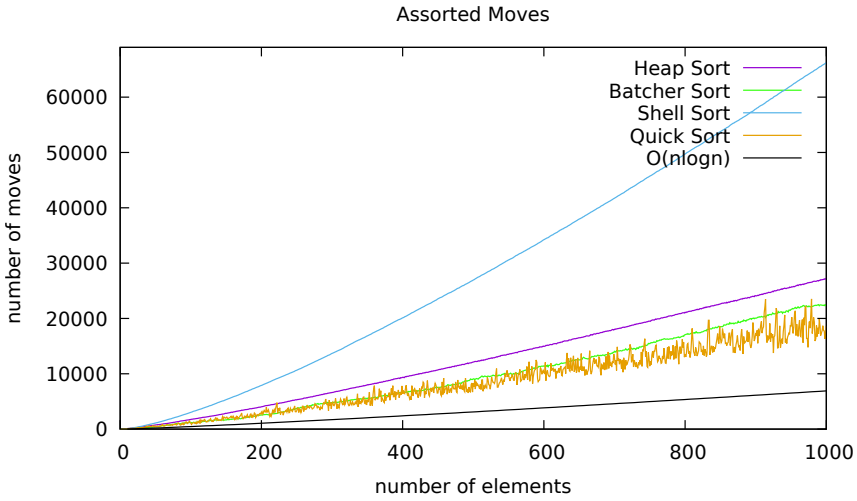




Quick sort is the least consistent of the sorts, and in fact seems to get more inconsistent as the number of elements increases.

7 Comparing Sorts





The above graphs chart the number of moves and compares each sorting algorithm efficiency with the same array of 1000 elements. All of these graphs are O(nlogn), which is charted in the second graph for comparison, but have different constants and therefore look very different.

Quick sort is, like its name, generally the quickest and does the least amount of sorts and compares, but is also relatively inconsistent compared to the other 3 and is the only one that does not appear as a smooth curve. Heap Sort and Batcher are generally slower than quick sort and perform more compares and moves on average. Batcher does more compares, but heap does more moves, which makes sense since heap has to constantly rearrange the heap (which puts it in descending order, but guarantees the largest item is on top). Shell Sort is the slowest by far and performs several more compares and moves than the other algorithms.

8 Conclusion

This assignment was extremely informative as I frantically flipped through the C Programming Book to double check all of my syntax. Prior to this, my knowledge of pointers was limited to CSE12 and my

knowledge of arrays in C was practically nonexistent because they differ quite a lot from arrays in other languages. I also learned the basics of how structures in C are declared and initialized from stats.h and stats.c, especially from reading Chapter 6.7 of the book. This was not in the additional recommended reading, for some reason, but the examples provided did help me with my own code and understanding of structures.

Back to the actual topic, sorting. I learned a lot about the differences between algorithms and the importance of algorithm efficiency. Even most of the sorting methods have an average case of O(nlogn), they have different constants and therefore produce very different looking plots. I especially noticed this when testing sorting.c with very large numbers (greater than 10 million), where the time difference can be felt in full seconds. This showed me how painful very inefficient algorithms can be as I sat there with my head on the desk, hoping this was just normal inefficiency and not another $O(\infty)$ loop.

Overall, I would say I have learned the most about C and general computer science concepts (like sorting and efficiency) from this assignment compared to previous assignments.

9 Credit and Notes

All sorts were based on the python pseudocode provided in the assignment spec. A lot of formatting for filling out and allocating arrays was taken from calloc_example.c in the resources (marked by comments in code).