CSE 13S: Assignment 3 Write-Up Document

Johnny Li

February 5, 2022

1 Introduction

Having completed this assignment, I now have a deep appreciation for sorting algorithms and their implementation. This lab allowed me to build and understand each method from scratch, giving me valuable insights into the strengths and weaknesses of each approach.

I now understand why it's crucial to have a solid understanding of different sorting algorithms. It enables us to make informed decisions when choosing the best approach for a particular task, as the efficiency of the algorithm can greatly impact the overall performance of the program. And also, just because sorting resources exist and are easily accessible, it doesn't mean we can ignore the importance of learning about the various algorithms.

Each sorting method has its own unique strengths and weaknesses. For instance, QuickSort is one of the most common and fastest algorithms using comparisons, but it also has a worst case performance of $O(n^2)$. In contrast, its competitors have a strict $O(nlogn)$ worst case performance. This highlights the importance of having a deep understanding of each algorithm and its implementation, so that we can make the most appropriate choice for each task.

By constructing each sorting method from scratch, I was able to see the inner workings of each algorithm and develop a deep understanding of when and how each should be used effectively. I now feel well-equipped to choose the right sorting method for any given task and to implement and optimize it for maximum performance.

2 Sorting Algorithms

One of the key factors that impacts the performance of these sorting algorithms was the size of the input. For smaller inputs, Shell and Batcher performed well. Shell Sort uses incremental insertion, which means that it starts with a larger gap between the elements being compared and gradually reduces the gap size until all elements are sorted. This technique allows the algorithm to make fewer comparisons and exchanges, making it efficient with smaller inputs. Similarly, Batcher Sort also performs well with smaller inputs as it uses a parallel comparison and exchange technique, allowing it to sort the list in a relatively short amount of time. The parallel comparison and exchange technique used in Batcher Sort ensures that multiple elements are compared and swapped at once, reducing the amount of time required to sort the list.

However, as the input size increased, algorithms like Quick and Heap showed better performance in terms of speed and efficiency. Both QuickSort and HeapSort have an average-case time complexity of $O(nlogn)$, making them highly efficient for sorting large data sets. This is because they divide the input into smaller sub-lists, reducing the amount of comparisons and swaps needed. QuickSort uses a divide-and-conquer strategy that divides the input into two sub-lists and repeatedly calls itself on those sub-lists until they are of size one. This leads to a large reduction in the number of comparisons and swaps, making QuickSort highly efficient. Meanwhile HeapSort, on the other hand, uses a binary heap data structure to sort the elements in the input. It first creates a max-heap, where the largest element is at the root, and then repeatedly extracts the maximum element and places it at the end of the list. This leads to a highly efficient sorting algorithm, especially for larger inputs.

Another factor that impacted the performance of these sorts was the type of data being sorted. For the same reasons above, algorithms like Shell and Batcher performed poorly, whereas Quick and Heap performed well with data sets which were already partially sorted or nearly sorted data. On the other hand, for data with few unique values, algorithms like Heap and Quick showed poor performance, whereas Shell and Batcher performed well.

Shell Sort Implementing the Shell Sort algorithm in C was a great learning experience for me. Before starting the implementation, I had only a theoretical understanding of how the algorithm works, but after actually coding it, I gained a much deeper understanding. The process of converting a theoretical algorithm into working code was challenging, but also extremely rewarding.

One of the most interesting aspects of shell sort is that it is a variation of the Insertion Sort algorithm. Instead of sorting the entire list at once, the algorithm first sorts elements that are far apart, and gradually reduces the gap between elements until they are being sorted as a traditional Insertion Sort. This approach allows the algorithm to take advantage of the partially sorted list to sort elements more efficiently.

I encountered several challenges while implementing the shell sort algorithm in C. One of the biggest challenges was finding the optimal gap size for the algorithm. I learned that the choice of gap size has a significant impact on the performance of the algorithm and after researching various methods for choosing the gap size, I found one that worked best for my implementation. Another challenge was debugging the code. The algorithm has several nested loops, and it was difficult to keep track of what was happening at each iteration. I learned that it's important to add print statements and take a step-by-step approach to debug the code.

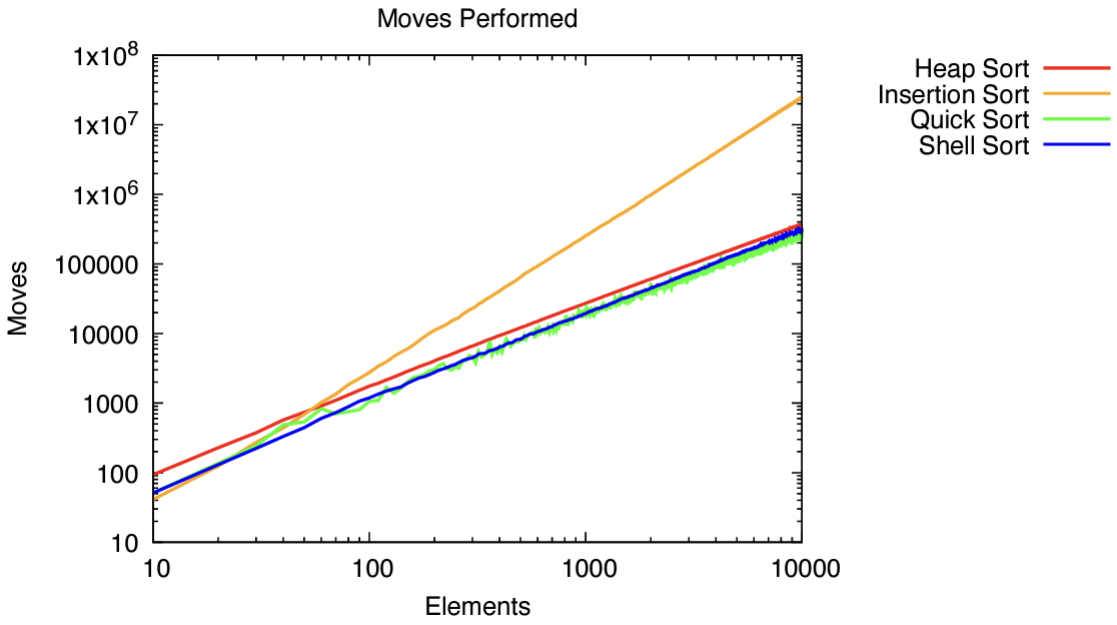
Batcher Sort The implementation process of Batcher Sort was a bit more challenging compared to Shell Sort. This was mostly due to the sorting technique used in Batcher Sort. It uses a parallel comparison and exchange algorithm, which was quite different from the incremental insertion used in Shell Sort. However, the core concept of dividing the list into smaller sub-lists was similar.

Another difference between the two algorithms was the performance. Batcher Sort showed a significant improvement in performance compared to Shell Sort, especially for larger lists. This is can be seen with the notably reduced amount of moves and comparisons in Batcher in relation to Shell.

Heap Sort Heap Sort was unique and I ran into the most issues with this file. This is probably due to the complexity of the Python pseudo code with four distinct functions. Although half of them are quite simple, when debugging – it was still a challenge to find where the errors were since it had the longest code and each one is referenced. This is also the file where I experienced my first Segmentation Fault which threw me for quite a loop. I've heard things about them from people and didn't really understand how much of a pain they were to debug since there's really nothing to go on. Although GDB was able to help to a certain extent, the bug was ironically not where I was expecting (and also not where GDB said it was). This file also has an incorrect number of moves compared to the results of the sorting_arm64 file, the only function with a mismatch although it's quite small (144 vs 147). I've heard we're going to be dealing with a lot of potential Segmentation Faults in Assignment 4 and after this experience, I'm not excited for it at all.

Quick Sort Quick Sort was actually one of the easiest files to code and the one which I ran into the least amount of errors. This is likely due to the fact that the provided Python pseudo-code is quite easy to translate into C and this was also the last file I worked on so I already knew what to do. The partition() function was easy to translate since most of it was simply copy paste without needing to change the syntax while quick_sorter() was even easier in that regard. And does anything need to be said for quick_sort? The only thing that could be incorrect in this function would be the variables and I've already learned my lessons there form the three previous files. Overall, my favorite file to work on since it was the least painful.

3 Sorting Algorithms Comparisons and Analysis



The graph displays the number of moves made per element for each of the four sorting algorithms. Upon closer examination, it is clear that the Insertion Sort algorithm has the highest number of moves per element, which rises steeply and exponentially. Despite the appearance of a linear line in the graph, the y-coordinate, which represents the number of moves, increases at an exponential rate, indicating that Insertion Sort is significantly less efficient compared to its counterparts. On the other hand, the other three sorting algorithms, Quick Sort, Merge Sort, and Bubble Sort, are much more efficient than Insertion Sort and have a much lower number of moves per element. It is worth noting that these three algorithms appear to be relatively similar in terms of their efficiency and it may take a more detailed analysis to differentiate between them.