Assignment 3 Write Up: Sorting

Aniket Pratap

January 25, 2022

Abstract

The main goal of this assignment was to code heap, quick, insertion, and batcher sort using provided python pseudo-code. The user input something like this: ./sorting -q -n 15 -r 232 -p 3. The -q represents quick sort, while -h, -i, and -b represent heap, insertion, and batcher sort. -n represents the size of the array, 15, -r represents a seed that the random function uses to set an array of random numbers. Finally, -p represents how many numbers of the array to print using PRIu32's.

1 Insertion Sort

This sort was the first sort I did due to the ease of understanding. The sort was implemented using the provided python code. However, due to python's quality of life improvements, some extra code needed to be done. Insertion sort took in the stats variable to count the moves and the array and length of the array. If the value before the current array index is greater, one must switch the values. Every time I needed to store a variable in temp, I used to move. This increased the move counter by 1. Insertion sort also has one comparison, temp < A[j -1] which checked if the value before it was greater than it. It then loops until it was less than all the values before it–which I then moved into the index of that place.

2 Heap Sort

This sorting method was a little harder to understand. The array starts at 1, and leaves are represented as 2k - left leaf, and 2k + 1 - right leaf. The first function I made was buildHeap. This function built a max heap array ordering. Since this function called fix heap, I had to have Stats *stats as an argument. The loop started at the middle of the heap, and checked the children while moving up the heap to the max value. The nect function I wrote was fixHeap. This function removed the max element from the heap, or the first item in the list, and ignored during the next runs. Then, the function would rebuild the maxHeap and do the process over again. This was done by comparing the mother element to the maximum child and swapping if necesssary. Once the heap was fixed into a max heap, the variable found, which was previously false, is now true. fixHeap. used the function maxChild, which return the biggest number based on the left and right leaf. These function calling eaechother is what makes the heap sort.

3 Quick Sort

This was next because it was the next easiest to understand. The goal is that there is a pivot, and you want all the numbers that are less than the pivot on the left and all the numbers greater than the pivot on the right. This process repeats as the main array is broken into sub-arrays. In this case, the partioning is done

by the partition function and takes in a lo and hi value–along with the stats and array of course. If the value is greater than the pivot, place it right, otherwise keep it left. The quickSorter function recursively calls itself so that we can break down the main array into two sub-arrays. This process repeats until sorted.

4 Batch Sort

This was the hardest to understand since it was so new. Since this was a sorting network, this meant that it had a fixed number of comparisons due to the fixed number of comparators. The algorithm works in parallel because it sorts the even indices of an array and the odd indices–which then get merged. This idea of even and odd indices is represented by bit shifting the variable p by the bit length of the array. By doing 1 « (count -1) I am choosing how many k-sorts I should do. The k is how far 2 numbers are away from each other. These numbers are then compared in the comparator function, which swaps then if one is less than the other. Since this program is run using k-sorts, no elements overlap when being compared–since each comparison is k elements away. Now that the number swaps, the process is repeated except the p variable, which determined the k sort, is bit shifted by one. Since k is bit shifted by one, the new run will have a k-sort of $frack2$. The algorithm continues this process until it 1 sorts the array.

5 Sets

Sets were used for most of the error handling, and this is quite similar to how I did error handling on the last assignment. Previously, I used an array of boolean values, but now we are using bits to represent if the user input something. I decided to create two separate enumerations: one for the sorts, and one for the filters. I thought this would be more organized. I then created two empty sets, one for each enumeration, and I inserted a 1 value if the user input a filter or sort into the set. I created a loop inside the main function to check if the user inputted an algorithm. I did this using a for loop checking if the algorithm was a member in the set. If the final bool value of check was false, then an error would be printed saying that no algorithm was inputted. This concept was also implemented for the other filters. I used sets and if-statements to check if the input was a member of the set, then I operated.

6 Calling the Sorts

There isn't much to say about this section except that I had to use "malloc" to allocate memory of the size that the user wanted–multiplied by the size of a uint32. This was done to get the size of memory that the user wanted for his array. I then randomized the array by using a default seed and bit masking it with a 30-bit number before I added it to the list. Finally, "free" was called so that I could deallocate the memory and create an identical randomized list. This was done because the sorting algorithms accessed the memory directly, rather than creating a copy of it and changing the copy while keeping the original intact.

7 Graphs

The first graph shows insertion sort with the highest amount of moves. This is natural because the algorithm needs to iterate through the array multiple times. For 10,000 elements? That's going to be a while. The line for the graph is mainly linear. The interesting part is that insertion sort is faster up to the 70 elements mark. Quicksort, as the name implies is supposed to be quick–but it loses to bathcer and insertion due to smaller elements. Why is this the case? It could be due to the recursion that may seem tedious for smaller elements. But why is it zig zag? This is because of how quicksort works. It's the fastest if the partition is in the middle, but has a worst-case if the partition is either the first or last element. The reason why the line is continuous is because of the cases in between. The partition could be second to last or the third element in the array. As the elements increase, it can be seen that quicksort is faster than the rest. Heapsort is the worst up till the 70 element mark. This could be because of how it rearranges the heap and swaps elements. But as the elements increases, heap sort starts to rival batcher quick. The next sort is batcher, and since it has a fixed number of comparisons it appears to have fewer moves, but as the elements increase batcher starts to increase. Why? According to the second graph, it seems that batcher sort has a bigger C constant. Over time, this C constant starts affecting the moves that batcher sort does. Why is quick sort the fastest as the elements get bigger? It has the smallest C constant. And by dividing the comparisons by $nlog(n)$ gives us this constant.

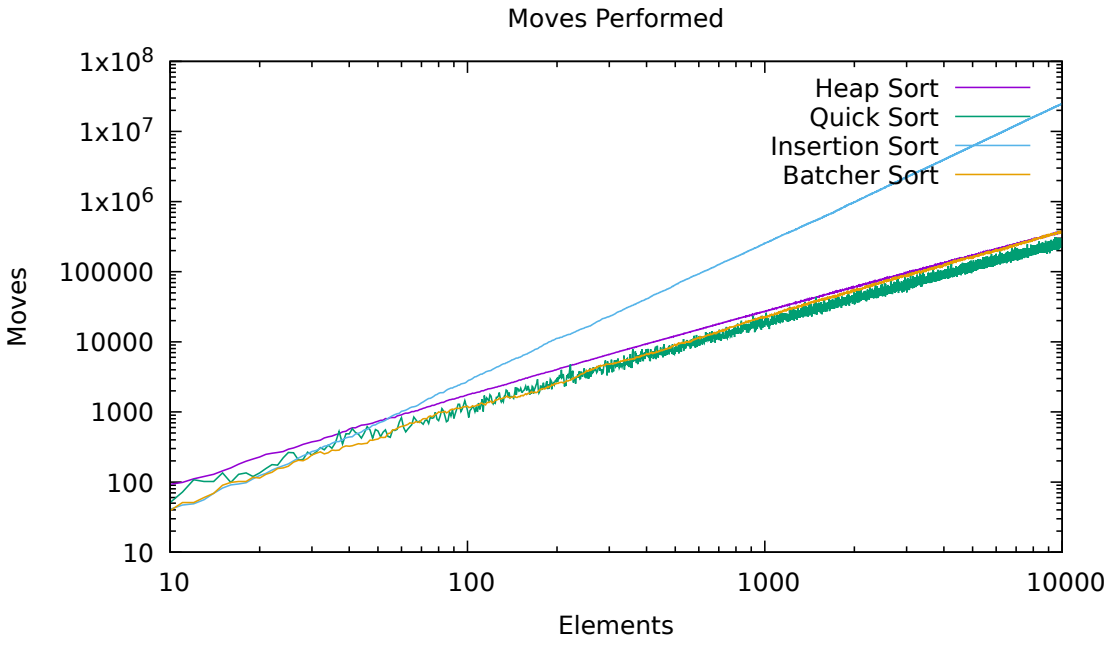

Figure 1: Elements vs Moves

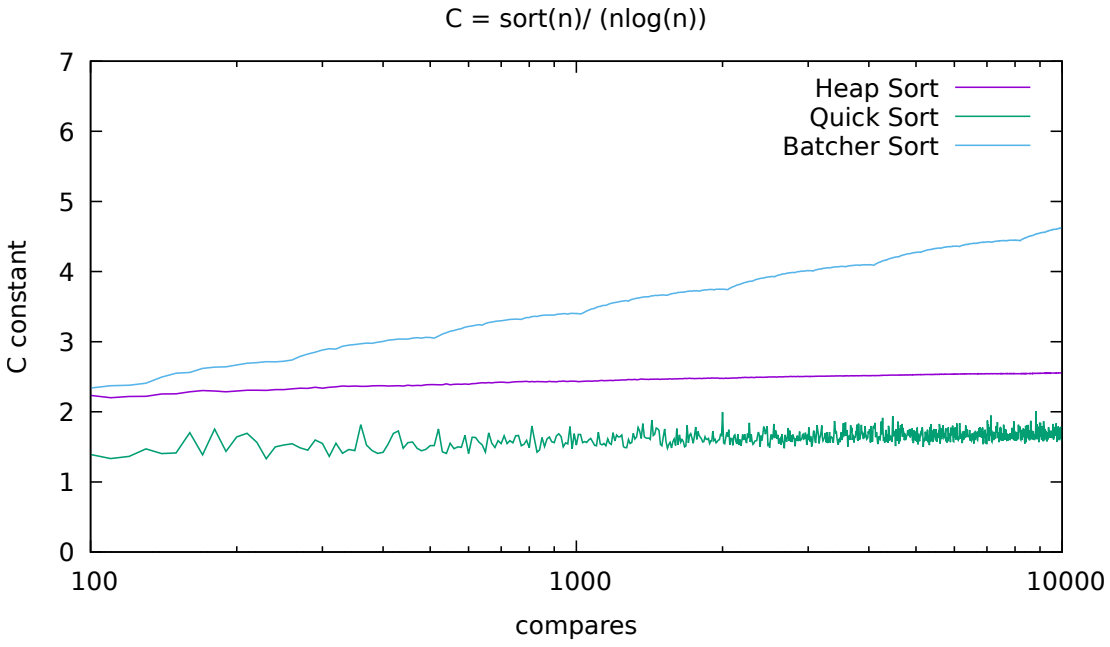

Figure 2: Comparisons vs C