CSE 13S Assignment 4: All Sorts of C Code

Jaren Kawai - jkawai

sorting.c

At the beginning of sorting.c is a function that is used to make copies of the unsorted graph to prevent inaccurate statistics from being returned from the sorting functions if more than one is called. It simply copies the contents of the unsorted array into a “copy” array that is eventually passed to the sorting algorithms.

Sorting.c also initializes new instances of the stats module and creates an empty set that is used to track what sorting algorithms are called by the user.

Getopt() is used next to take in user input and decide how long the array will be, how many elements are to be printed, and what the seed of the random number generator will be. For the cases that represent a sorting algorithm, the value for the sorting algorithm is inserted into the set created earlier. If there are no arguments passed at all, the program is meant to exit with no tests being carried out. If invalid arguments are passed, the program is meant to exit with a non-zero exit code. Otherwise, if it is established that a test is to be run, the absence of certain arguments will be set to default values that are set at the beginning of the file.

Once parameters are processed, the array of random numbers is created, and sorting algorithms are called. Numbers in the array were created using bit masking, hence the inclusion of a bitwise operation when populating the array. Tests are called based on a for loop and the membership function in sets that determines which sorting algorithms are to be used. When a given function is called, a copy of the original array is made using the function from earlier and passed to the function along with the instance of stats, and the length of the array. Once the sorting algorithm is called, the finished sorted array is printed along with the variables that stats tracks (moves and comparisons).

bubble.c

Bubble.c was implemented by using a nested for loop that would help create pairs of array elements that would be compared to each other and swapped if the earlier element in the array was larger than the one that occurred at the next index. This is the main purpose of the inner loop: to compare array elements. With each pass of the outer for loop, the biggest element is picked out each time eventually resulting in a sorted array.

heap.c

Heap.c contains help functions that are used to find the index of the parent and children, and build and fix the heap.

Up heap was created as a first pass ordering of each element in the array to comply with the constraints of min heap. Up heap will end up swapping elements until the lowest value element is at the base of the array. With build_heap, we are creating a heap using the first process of ordering elements by passing in a copy of the original array that will be altered. The array that is used to build the heap eventually becomes a sorted reference that can be simply copied into the

original array which would in turn also sort the original array. The first element of the heap is taken off and added to the original array, and the new first element of the heap is set to the last element based on how many elements have already been sorted to the original array. The parts of the array that haven't been copied are fixed again using down_heap so the array is compliant with min heap constraints again.

quick.c

Quick sort uses a divide and conquer method to sort different parts of the array separately, and bring all the sorted parts back together at the end. Since this function used recursion, the base case was if the length of the passed in array was below a certain length, then it would be sorted using shell sort. To begin, the function needed a pivot value of which to base the sub arrays around. This pivot value is the middle element of the array. The pivot value is then put at the beginning of the array. A for loop then loops over all of the other elements in the array that determine if that element is less than or greater than the pivot value. If an element is less than the pivot, that value is swapped with the pivot. At the end of the loop, the pivot value is placed back in the middle of the array. In the end, values that are less than the pivot are to the left, and the values that are bigger to the right. From there, quick sort is recursively called two more times, one for the left and right side of the pivot. This will result in a sorted left side and a sorted right side, which will collectively result in a sorted array.

shell.c

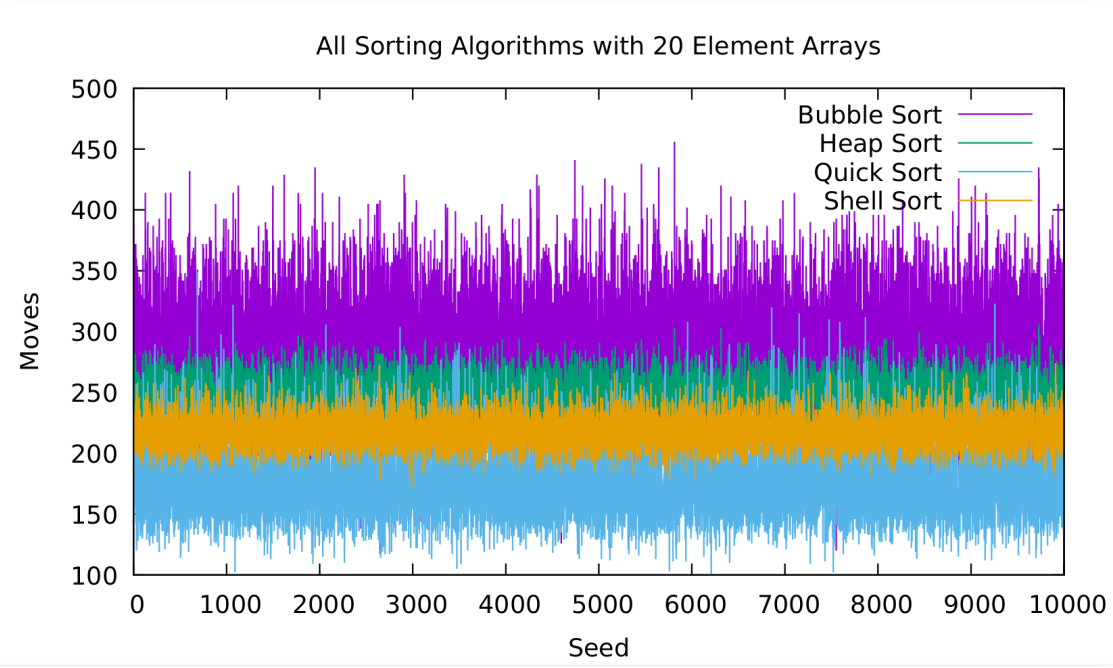
Shell sort arranges elements based on the distance that exists between a pair of elements, with farther apart elements being sorted first. The process continues until a gap of one is obtained. Shell sort was implemented here with the help of a function that would produce the gap length. The first gap would be the gap calculation with n being the length of the array, and would continue as long as the gap is greater than 0. This was facilitated using the outer for loop. One the current gap is calculated, another for loop iterates in the range of gaps to the amount of elements in the array. Inside the inner for loop, is when pairs of elements are compared to each other. The while loop determines if elements are to be sorted or left alone. This loop helps facilitate a version of insertion sort to sort the array within the bounds of the gap. Shell sort, like quicksort, was somewhat implemented in a divide and conquer fashion that involved splitting up the original array into smaller arrays, and sorting those separately.

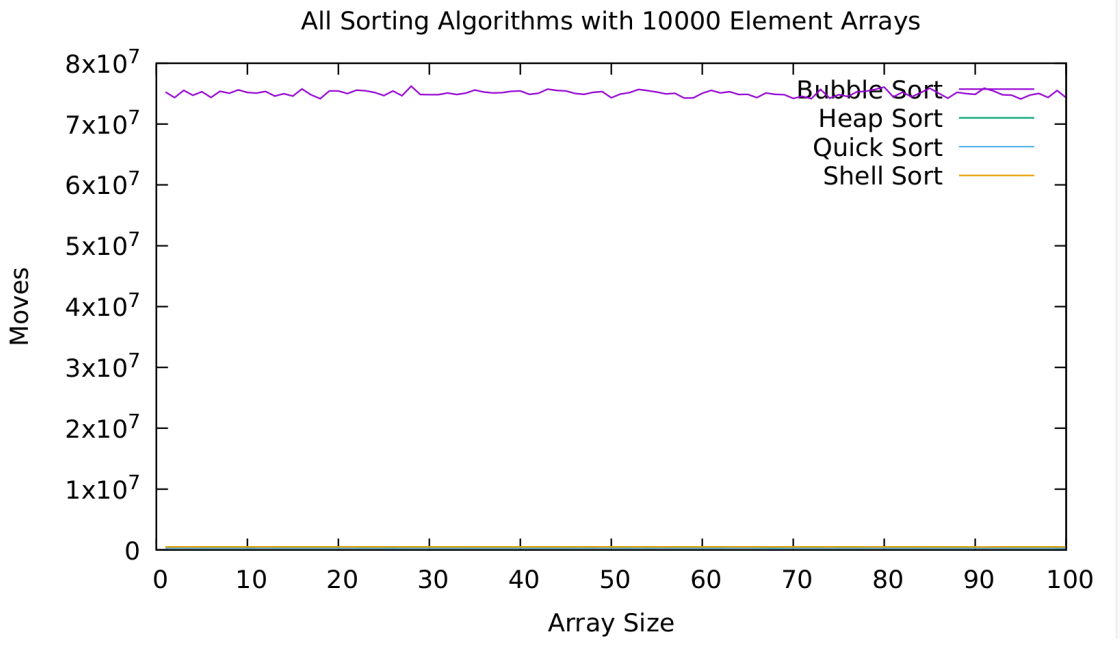
Further Analysis

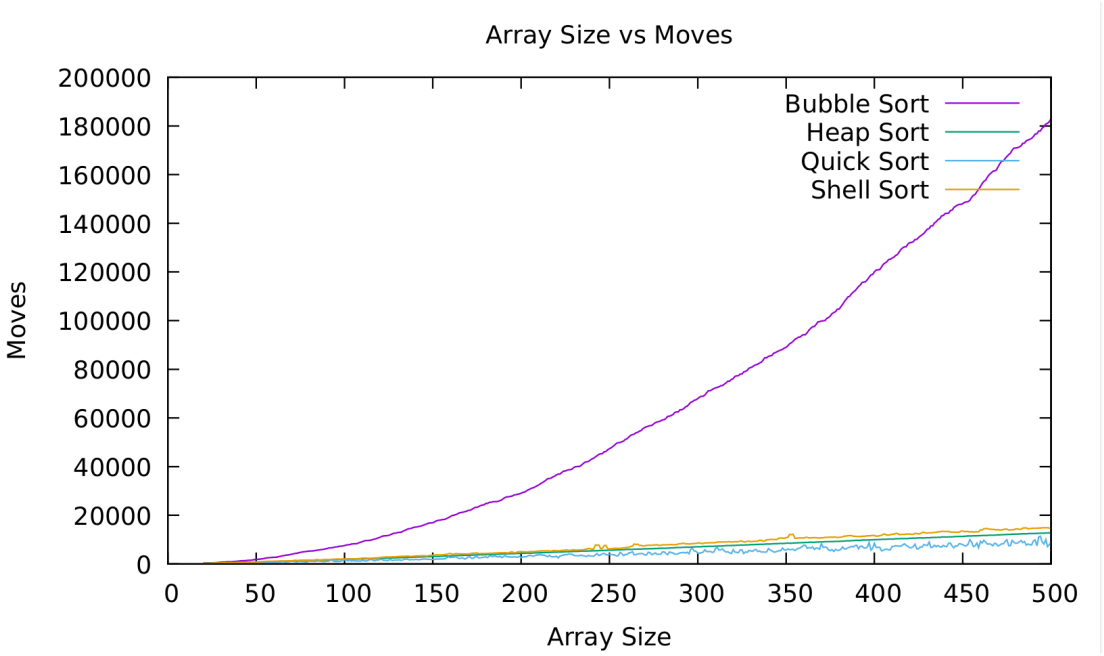
From this assignment, I learned about different sorting algorithms that I have not been previously exposed to. Out of the 4 sorting functions, there were definitely some that were faster than others, and required more moves and comparisons on average. Over the course of completing the assignment, I found that bubble sort was significantly slower than the other sorting functions even though it was the easiest to code.

From the graphs below, I also found out that the size of the array does matter in terms of how long it takes for any algorithm to properly sort all the elements of the original array. Two of the graphs show test results from running each sorting algorithm with different seeds and the same array size. With 20 element arrays, there all sorts took under 500 moves, but with 10,000 elements, functions took longer, especially bubble sort. Even looking at the graphs tested with changing array lengths, the longer the array became, the more moves were required to sort them. For the most part, bubble sort was slower than all the other algorithms with quicksort being the fastest. Heap sort and shell sort were closer together in terms of time complexity, but were still faster than bubble sort.

As shown, none of the graphs were smooth relationships. There were values that may have taken surprisingly long or short for the amount of elements there were. This could be largely in part because of the seed that was used which may have resulted in elements in the original array to be placed in favorable positions that wouldn't require as much rearrangement.







In sum, there are conditions that hinder or help these sorting algorithms perform, with size being one of the most important factors in determining time complexity. Another point to highlight with these graphs is possibly how reliable these algorithms are in more practical situations. Bubble sort becomes progressively longer as array size increases, and although that is true for the other functions as well, the others are more consistent in terms of how long it takes them to sort objects. This highlights the difference in methodology and how finding ways to optimize sorting (such as divide and conquer approaches) can be very beneficial.