Sorting Algorithms

Alexander Ragland #1858717

February 5, 2023

Abstract

Given an array of numerical values, the array is either sorted or unsorted. It is sorted if all of the values have a *logical ordering* within the array. If it is not sorted, then it is unsorted; if the array has only one value, it must be sorted. Traditionally, computer scientists assume *logical ordering* to be either ascending or descending by value. Within this project, we have opted to define a sorted array as an array of numerical values in ascending order.

The primary goal of this project is to analyze the efficiency and behavior of various sorting algorithms. The chosen algorithms for this project were Shellsort, Quicksort, Heapsort, and Merge Exchange Sort. Each of these sorting algorithms was tested 10,000 times on the same arrays of pseudo-random numbers. The tests began with sorting an array of 1 element, then an array of 2 elements, and so on until an array of 10,000 elements was sorted. Each time a sort was performed, the number of moves and comparisons performed was recorded.

1 Shellsort

Shellsort is a variation of insertion sort that starts with a chosen number called the *gap*. Elements of the array that are a *gap* number of indices away from each other are compared and then swap positions if necessary. In each iteration these *gaps* shrink, until eventually the array is sorted. Our project utilizes the Pratt Sequence to generate these *gaps*.

In *Figure 1*, you can see Shell Sort remains fairly linear in terms of the number of moves as it sorts arrays of increasing size.

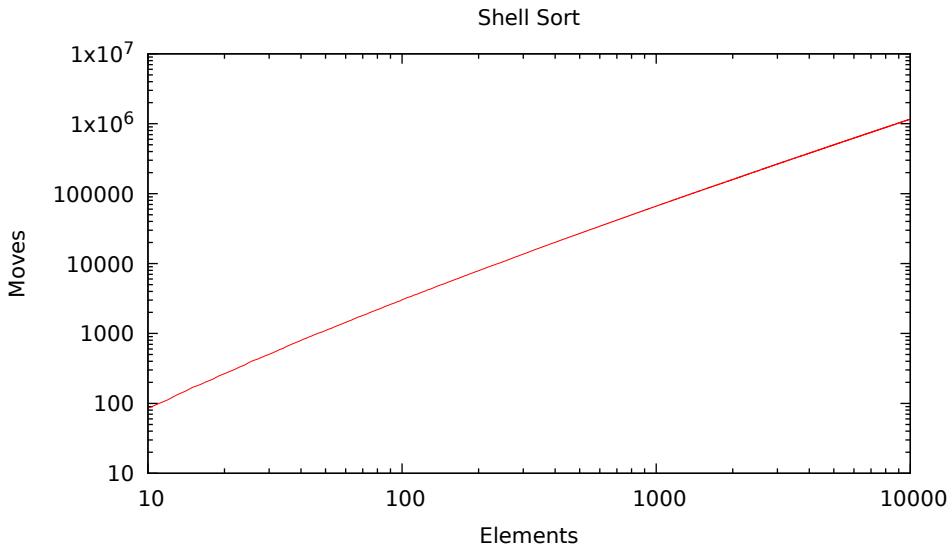

Figure 1: Shell Sort

2 Quicksort

Quicksort is a recursive algorithm that starts with a value within the array called a *pivot* and then splits the sections of the array on either side into sub-arrays. Values less than the *pivot* are placed into the left sub-array, and values greater than the *pivot* are placed into the right sub-array. This process repeats until the array is sorted.

In *Figure 2*, you can see that Quicksort varies unpredictably in the number of moves as it continues sorting larger and larger arrays. This is likely due to the *pivot* selected for each iteration. Some *pivot* values are worse than others; generally, you want to take the median value of an array and select that as your *pivot*. When Quicksort starts with a good *pivot*, the sort takes fewer moves, as opposed to the same sort with a bad *pivot*.

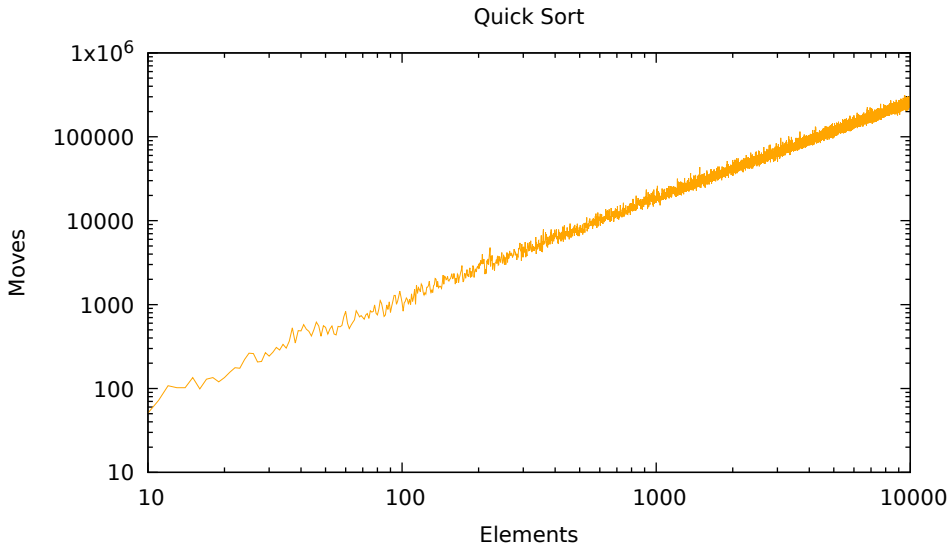

Figure 2: Quicksort

3 Heapsort

Heapsort is an algorithm that makes use of the heap data structure. In our project, we used a *max heap*. The algorithm first arranges the array into a heap, then moves the largest value in the heap to the end of the array. The heap is then re-ordered the match the definition of a *max heap* and the process repeats until you are left with a sorted array.

In *Figure 1*, you can see that Heap Sort remains fairly linear in terms of moves to elements in the array. However, there is a slight variation between iterations. This is likely due to the required "fixing" of the *max heap*.

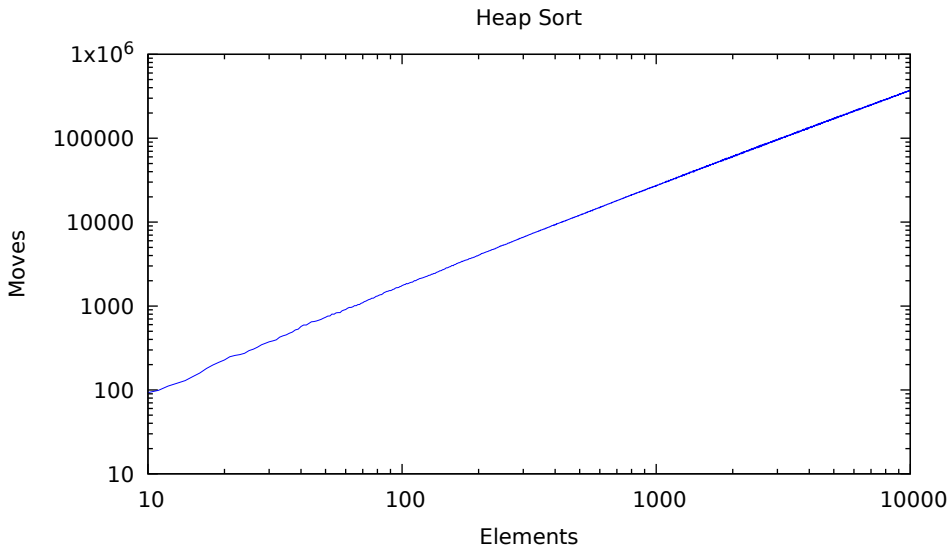

Figure 3: Heap sort

4 Merge Exchange Sort

Merge Exchange Sort is a modification of Batcher's sort, which utilizes a *comparator network* to sort arrays. Batcher's method is traditionally limited to inputs that are powers of 2, but we can apply Knuth's modification to sort any number of inputs. With the modification, it's called Merge Exchange Sort. Merge Exchange sort is similar to Shell Sort, except it *k-sorts* instead of using *gaps*. It also normally runs in parallel, but our implementation runs sequentially.

In *Figure 4*, it's apparent that Batcher's method varies slightly in moves between iterations.

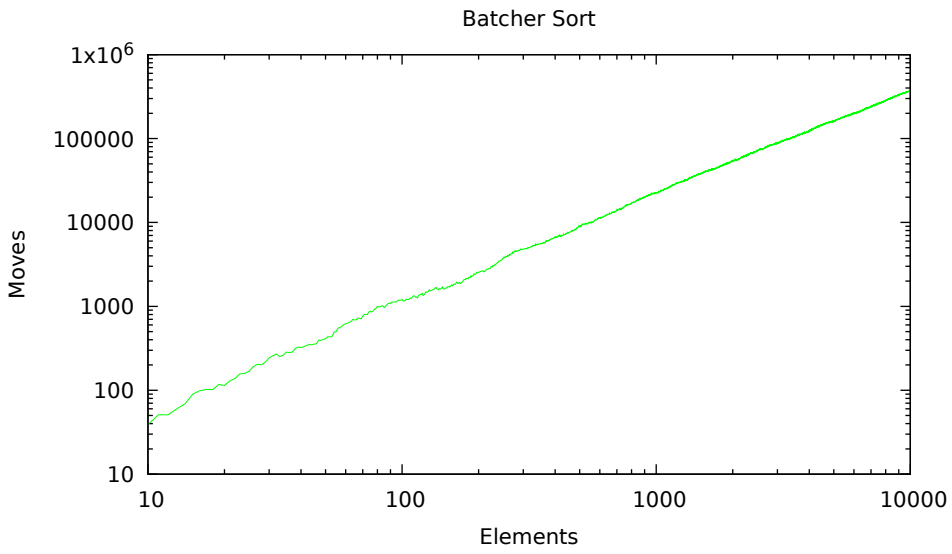

Figure 4: Batcher's Method/Merge Exchange Sort

5 Compare

Figure 5 shows how the sorts compare to one another. As you can see, Shell Sort, although the least varying in moves, is slower than the other sorts, except for heap sort when the number of elements in the array is very small. Batcher's method and Quicksort are very close in terms of efficiency; Up until approximately 80 elements, Quicksort is faster. After 80 elements, Batcher's method bounds Quicksort, sometimes being faster, and sometimes being slower. Eventually, Batcher's method pulls away and remains the most efficient. Interestingly, Heapsort is noticeably slower than Quicksort up until approximately 10,000 elements. At that point, it seems that the lines for Heap Sort and Quicksort cross over, and Heapsort becomes faster.

The first few iterations of sorts have much more variance and overlap between the algorithms' efficiencies because of the constant in their time complexity. A higher constant in an algorithm's time complexity might lead to more efficiency with a smaller data set, but it doesn't mean that the algorithm will continue to be the most efficient compared to others over a longer period.

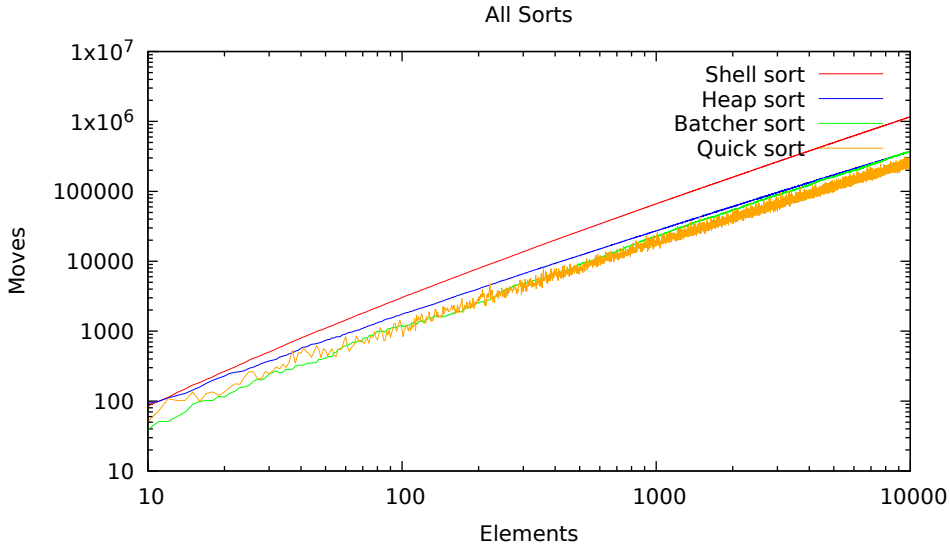

Figure 5: All Sorts

6 Acknowledgements

- **Audrey Ostrom (Tutor):** Syntax for malloc/calloc, set functions, and valgrind
- **Jennie Linn (LSS Tutor):** Set functions, shell sort, quick sort, heap sort
- **GeeksForGeeks (https://www.geeksforgeeks.org/python-program-to-covert-decimal-to-binary-number/):** Python code for a decimal-to-binary function. I converted it into C and changed it so that it calculates bit length instead of the binary value.