Sorting Competition

Sorting is the fundamental process of ordering data in a given fashion. As the desired end result is relatively the same, it is the path taken to achieve order that is important. In our quest to find a fast and efficient method for sorting ASCII data, we came across varying algorithms that had varying efficiencies. The size of the data, as well as the type of data, played the biggest role in choosing an applicable sorting method.

We began by writing a basic selection sort as a proof of concept. We soon found that as our data set increased in size, the time required to sort quickly grew as well. This was due to the selection sort being on the order of . This simply would not suffice for lager data sets. After scraping selection sort, it was clear we needed a sort that would prove more elegant in action.

Insertion Sort

One of the first sorts we implemented was an insertion sort. It had a relatively easy implementation and could be adapted to sort both length and ASCII value in one function call. On small sizes of data, 30 to 80 elements, it blew selection sort out of the water. However, just as with selection sort, insertion sort had a quadratic run time as that data set size increased. We believed if the larger data sets could be broken up, insertion sort would prove useful in conjunction.

Shell Sort

We then moved onto implementing a shell sort. It seemed appealing as it utilized an insert type sort, but based on a varying gap size. This means it was able to swap elements that were far from their final sorted location much more efficiently than a selection sort type exchange. We choose a gap size that decreased by a divisor of 2 each pass. This way it would be able to handle larger sets of data more efficiently. Unfortunately due to the nature of the algorithm, sorting by length then ASCII required two separate calls to the shell sort method. This added complexity slightly took away from our efficiency gained by using this method.

Shell Sort with modified gap

In our quest to optimize shell sort for time efficiency we began by streamlining our code. One optimization we found was the way the gap size was chosen. We were using a basic divisor of 2, which in preliminary tests proved effective. However there are many proposed gap sequences available that could reduce the worst-case time complexity. We chose to research the Sedgewick gap sequence. This reduced our sort time slightly.

Quick Sort

Next we went a completely different route and began implementing a quick sort. The divide and conquer behavior of this algorithm is what made it so appealing. It required a basic implementation that proved elegant in the recursive form. It also proved to be faster on larger data than anything we had tried previously. The choice of an efficient pivot was the main focus of optimization. However, we stuck with simply choosing the median element as we tended to sort large amounts of data. The way in which we stored our data also allowed quick sort to sort by length and ASCII value in one efficient function call.

Hybrid(Quick/Insert) Sort

After assessing the sorts we had already tried, we decided that a hybrid of two sorts seemed plausible. We wanted to effectively maximize the strengths offered by two or more sorts. We decided that utilizing the recursive quick sort until a small enough data set was achieved for insert sort to be efficient would take the best of both worlds. After implementing this hybrid sort, we also had to choose at what size of a data set we switch from quick sort to insert sort. Through experimentation we found that at a size of about 9 to 10 elements, sorting with insert sort proved much faster than quick sort alone.

After going through the process of trying all these different sorts, we chose to stick with our hybrid sorting method. We then spent time on figuring out a way to store our data that was effective and would efficiently work with our sorting method. …

Loading, Storing, Preparing, and Sorting

In order to optimize the speed of our quick sort, we needed a way to quickly and effectively remove duplicates during our sortData() routine. In order to accomplish this while also speeding up our comparisons, we chose to initially store our input as short arrays, where [0] contains the length of the word and the following positions contain the ASCII values of the characters of that word. 32 is subtracted from each of the values in the short array for each word, and the new value is stored in two digits of an unsigned integer in an array of unsigned integers corresponding to the word. Each unsigned integer holds up to four characters of the word. Any word containing a character with an ASCII value greater than 128 is stored as an STL string in a vector for later use. Such words are stored in the summary array as unsigned integers, where the high-value character is represented by the placeholder value 95. Also, an array of the indexes of the words in the array of unsigned integer arrays is created. These indexes are to be swapped instead of the words themselves

Before passing data to quicksort, all 1- to 4-character words are stored by incrementing a counter in array1\_4[<value of unsigned integer representing word>]. This allows for an elimination of a great deal of the data to be sorted, while simultaneously removing duplicates. Words of 5 or more characters are moved to the bottom of the array, until the bottom portion of the array contains only words of 5 or more characters. This is the portion of the array to be passed through quick sort and then insertion sort.

When comparing, because arrays of unsigned integers are being compared, four characters are compared at a time. When swapping, instead of swapping the actual value of the word in the array, the indexes to the array containing the actual words are being swapped, increasing efficiency. When it comes time to print, the array of 1- to 4-character words is printed first, the counters at each index are used to determine how many times to print that word. The remaining values are printed from the sorted sumArray beginning at an index recorded during the population of the 1- to 4-character array.