Sorting Competition Write Up

Nick Antonelli

Josh Thomas

Objective:

To develop a program that will read in words from an input file, sort those words by length and alphabetically, and then output the sorted list.

Algorithms tested:

Over the course of working on the program for the Sorting Competition, we tested and implemented several different algorithms to sort the data. Listed below are the different versions of our sorting algorithm, accompanied by a description of the algorithm and a statement as to how we improved upon it.

Implementation 1:

The first attempt we made to sort the data was to try and create our own version of quick sort to alphabetically sort the words. The words that we would be sorting were stored in a vector of null terminated C-strings. The plan was to use selection sort to sort the words by length into sub groups with all words of on length in that group, and then use the quick sort that we wrote to alphabetize each group of words. Unfortunately, we could not get our quick sort function to work properly, so we moved on to try a slightly different approach.

Implementation 2:

The second sorting algorithm that we devised was to use merge sort in combination with selection sort to sort the data. At this stage, we had the data stored in the form of a vector of null terminated C-strings. We first used selection sort to sort all of the words based on their lengths by implementing the strlen() (string length) function. After the words were all sorted by length, we then used merge sort to sort each of the sub groupings (based on word length) alphabetically.

This implementation was very effective, but was only moderately efficient. We found that the times we were getting to sort all of the words were going to be anywhere near competitive enough to have a chance to win the competition. The program worked fine and didn’t take an abnormally long amount of time to sort, but it was much slower than we wanted. This was the stage the program was in when we submitted it for benchmark testing.

Implementation 3:

Implementation 3 was made up of several, specific things that were changed to improve the functionality of the program. For this implementation of the program, we switched over to using a combination of radix sort and qsort to sort the data. Since we could not get our self written version of quick sort to work, we implemented the standard library implementation of quick sort, more accurately referred to as std::qsort(), to help sort our data.

Before we describe the process that we took to implement those two sorting algorithms, its important to note that we made two significant changes to how we were storing the data that we would be sorting.

The first change that we made was that we switched from using conventional null terminated C-strings, to using length prefixed C-strings as a way to represent the words. This allowed for us to reference the length of each word much faster during the sorting process because it cut out the step of having to repeatedly run the strlen() function to determine the length of each word.

The second change was related to how we stored the words that we were going to sort. Even though we continued to read the words into a vector of C-strings, we decided that instead of copying the words over into a second vector to be sorted, we would copy the words over into a standard char\*\* array. Sorting the char\*\* array was an improvement over sorting a vector because the char\*\* array is a much more basic data structure, and has a lot less overhead related with using it. This improvement helped to increase our sorting time because it reduced the number of function calls and steps that the computer’s processor would need to execute while dealing with the words.

We utilized the length prefixed C-strings by using our radix sort method to sort the words by length, and then using qsort to sort the words alphabetically. We would use the radix sort to re-order the array of words based on length, and then we would pass the whole array to qsort to be sorted alphabetically. We found that this method worked very well and was much faster than Implementation 2, but we did not fully understand why until we took a closer look at our algorithm.

As it turns out, this implementation only worked because we were using length prefixed C-strings. We were sorting the array by length using radix sort (which worked very well and very efficiently), but then we were passing an array of all the words into qsort to be sorted alphabetically. If the words were not stored as length prefixed C-strings, then qsort would have alphabetized the list of words without regard to length. This would have resulted in a final output of words that were alphabetized, but not sorted based on length. This logic error went unnoticed because of the length prefixed C-strings. Because qsort would go through and examine each character of the string, qsort was organizing based on the lengths stored at the front of the string before it organized based on the characters of the word. This resulted in an output that was organized based on length and alphabetical order, but it would have done so regardless of whether or not we had used radix sort prior to qsort. In testing we found that if we removed the radix sort function call entirely (thus sorting the length prefixed strings using only qsort), the program ran faster than before and still generated the correct output.

Implementation 4:

We realized that using a hybrid solution would be faster than using qsort by itself, so we tried to come up with a hybrid solution for our fourth full implementation of the program. What we finally created was an algorithm that used radix sort to separate the words into sub groups based on length, and then used qsort to alphabetize each of those sub groups. This implementation differed from Implementation 3 by no longer using qsort to sort the entire list of words. Instead, this implementation used radix sort to break the data into smaller groups (based on the word length), and then used qsort within each of the groups. The final step was to merge all of the groups back together to create one list of all the words that had been sorted by length and by alphabetical order.

We found that this method produced the fasted sorting time out of all the implementations that we tried, and it is the format of our final submission for the competition.