Chris Henk
Eric Smith

Sorting Competition

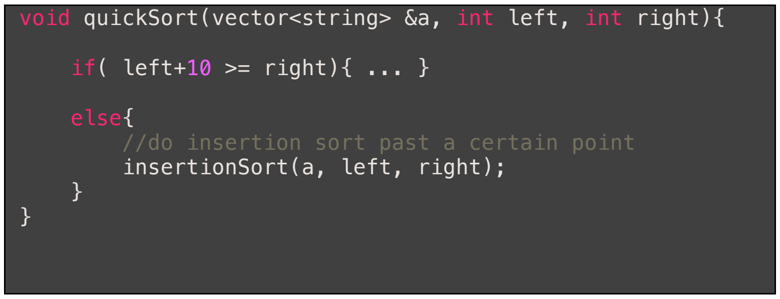
Our solution began with a ton of planning. We spent about a week searching for how to manage and sort data in a way that would maximize efficiency. We finally decided that we'd try two main solutions: one comparison and one noncomparison sort.

Gathering Sort Times

We timed each algorithm using the standard std::chrono on the same computer. We wrote a random word generator that would create strings from size 0 to 99 with random letters. The main method was obviously called a lot, so we tried only writing down the times that reflected major implementations.

sortData()

For our comparison sort, we implemented a version of quicksort that eventually switched to insertion. This "switch point" was a bit hard to find, but here are some different ones we tried with a 44KB file:


```
void quickSort(vector<string> &a, int left, int right){

    if( left+10 >= right){ ... }

    else{
        //do insertion sort past a certain point
        insertionSort(a, left, right);
    }
}
```

Switch Point	Average Time
left+5	~12ms
left+10	~12ms
left+20	~13ms
left+30	~13ms

We chose "left+10" for the first draft, knowing that the first input file would be relatively small. But once we started testing larger files (below are the results of a 162KB file) we decided to switch to our second idea: a noncomparison sort.

Switch Point	Average Time
left+5	~205ms
left+10	~205ms
left+20	~203ms
left+30	~200ms
left+40	~201ms
left+50	~201ms

Our noncomparison sort is a modified version of most significant digit (MSD) radix sort. MSD radix works in lexicographic order, sorting each word in variable-length order (1, 10, 2, 3, 4, 5, 6, 7, 8, 9).

The algorithm takes in 8 characters at time and iterates to see if the word already exists. If it doesn't exist, the word is inserted into the tree. Then that process is repeated within that bucket until you run out of digits.

This was our very first implementation of the radix tree:

Method	File Size	Average Time
Quicksort	50kb	~13ms
Radix tree	50kb	~2ms

Cleaning up some of the code, we were actually able to reduce time by combining get and insert methods for words.

Method	File Size	Average Time
Separate functions	10mb	~21ms
Concise functions	10mb	~16ms

The most difficult improvement was definitely treating each word as a series of numbers, rather than characters.

Method	File Size	Average Time
Chars	40mb	~611ms
Longs	40mb	~105ms

prepareData()

Our original prepareData method was very simple. We chose to prepend each word with a '+' character as this has a lower ASCII value than the character 'A'. From there, quicksort would sort the words of the same length, and eventually switch to a certain point.

When we switched to Radix, however, we prepended the data with a space.

Conclusion

We really took advantage of being able to use a tree in the sortData method. Our solution is unique because we're handling ("comparing") that does 1/8 the depth of a normal search. Because we force each word to be a certain length, the execution time is the same for n number of words. In other words, best and worst case performances are the same. The run time is O log(k*logn).

There are a lot of things we tried, but weren't successful at. Overall, we're just happy we were able to implement the radix tree at all!