CS 4000

Homework # 2: Circle the Word

due Thursday Feb. 11th, 2016

(50 pts.)

Consider the following problem. Given an $n \times n$ matrix of characters, and a list of words, output the location of the lexicographically first occurrence of each word in the matrix. This is very similar to the somewhat popular "Circle the Word" puzzles that you may have played before. Consider the following matrix of characters.

```
m m y a l c g y e a
s v x g g e f i t k
n y g y r v j x n s
p u b b e f q j p p
j b i r t r l h u t
g l e s s o m m a x
w j j h m g l c k r
m g b s a e s r o h
n w v i h f t y v e
o l p f r f d c a j
```

Can you find the words "horse", "gerbil", "hamster", "fish", "cat", and "frog" in this matrix of characters?

Your task is to write a program that outputs the location of the first character (x, y) of the lexicographically first occurrence of each word from the list in the matrix. (If a word appears twice, at the locations (x_1, y_1) and (x_2, y_2), the lexicographically first of these two is (x_1, y_1) if $x_1 < x_2$ or $x_1 == x_2$ and $y_1 < y_2$.) The natural approach is to search the vector via brute force. (For this problem, the words on the list may not be limited to dictionary words.)

For testing purposes, you will solve this problem by implementing a single class called "CircleTheWordSolver" with at least one public member function

```
vector<pair<int, int> > word_locations(
        vector<vector<char> > &puzzle,
        vector<string> &wordlist);
```

This member function will return the lexicographically first word locations of all the words in the list `wordlist`. If the word does not appear in the puzzle, return $\langle -1, -1 \rangle$ for the location.

1. (10 pts.) Implement `CircleTheWordSolver` without using any parallelism.

   Implement your program in C++ and provide adequate documentation. Test input/output examples will be provided to help you debug your code. Your program should be reasonably efficient.

2. (20 pts.) Use OpenMP to implement a parallel version of the program that you implemented in problem #1. Implement your program using `#pragma omp parallel`, but *do not use* `#pragma omp parallel for`. Hence, you will need to partition the problem space manually.

   Compare your program's running time to your original program. Your program should be at least 75% efficient on a four core machine on input of a 1000 x 1000 puzzle with a word list of 500 words.

   Implement your class in C++ and provide adequate documentation.

3. (20 pts.) Use OpenMP to implement a parallel version of the program that you implemented in problem #2. Implement your program using `#pragma omp parallel for`.

   Compare your program's running time to your original program. Your program should be at least 75% efficient on a four core machine on put of a 1000 x 1000 puzzle with 500 words in the wordlist.

   Implement your class in C++ and provide adequate documentation.