CS106X Practice Midterm

Exam Facts:

When: Thursday, October 24th from 7:00 - 8:30 p.m.
Where: Bishop Auditorium

Coverage

The exam is open-book, open-note, closed-electronic-device. We will not be especially picky about syntax or other conceptually shallow ideas. We are simply looking for a clear understanding of core programming concepts. As needed, we will present the prototypes of functions and methods we expect you'll need.

Writing code on paper in a relatively short time period is not quite the same as working with the compiler and a keyboard. We recommend that you practice writing out solutions to these practice problems—starting with a blank sheet of paper—until you're certain you can write code without a computer to guide you.

This practice midterm draws its problems from a few different midterms I've given in past years. Understand that I'm under no obligation to imitate the format of this exam, though. I'm simply presenting this practice midterm to give you a sense of what types of problems have been given on CS106X midterms in previous years.

Problem 1: Word Ladders, Take II

For Assignment 2, you implemented a breadth-first search algorithm that generates the shortest word ladder between two words. The pseudo-code presented in the assignment handout was this:

```
create initial ladder (just start word) and enqueue it
   while queue is not empty
      dequeue first ladder from queue (this is shortest partial ladder)
      if top word of this ladder is the destination word
         return completed ladder
      else for each word in lexicon that differs by one char from top word
           and has not already been used in some other ladder
              create copy of partial ladder
              extend this ladder by pushing new word on top
              enqueue this ladder at end of queue
```

An implementation coded to specification never uses a previously used word to extend a partial ladder. Stated differently, each word—whether or not it ultimately contributes to the word ladder of interest—has a **unique predecessor**.

a. Imagine that you have access to a **Map<string, string>** called **predecessors**. Each key maps to the word preceding it in all partials ever generated during a search. Given references to the **start** word, the **finish** word, and this **predecessors** map, it's possible to reconstruct and return the word ladder connecting **start** to **finish**. Implement the **reconstruct** function, which does exactly that.

```
static Vector<string> reconstruct(const string& start, const string& finish,
                                  const Map<string, string>& predecessors) {
```

Your Assignment 2 implementation made use of a **Queue<Vector<string>>** to maintain a first-in-first-out list of all the partials ever generated during a search. It's possible to reduce the memory footprint of the breadth-first search by relying on a **Queue<string>** (where each **string** is the last word of a partial word ladder), provided you maintain a **predecessors** map along the way as well. Restated, it isn't necessary to (and for this problem you shouldn't) maintain a queue of partial word ladders, since all partial word ladders are implied by their last word and the information in a **predecessors** map.

b. Using the **reconstruct** function from the previous page, implement the **findShortestWordLadder** function, which accepts references to **start** and **finish** (you can assume they're each **string**s of the same length) and returns the shortest word ladder between them (or the empty **Vector<string>** if there isn't one). You should rely on the following function (you may assume it has already been implemented for you):

```
static Vector<string> generateAllNeighbors(const string& word,
                                           const Lexicon& english);
```

which returns all English words that differ from the provided one by exactly one letter.

```
static Vector<string> generateShortestWordLadder(const string& start,
                                                 const string& finish,
                                                 const Lexicon& english) {
```

Problem 2: Autocorrect

We all know that when our big thumbs type out big words on our smart phones, we mistype and spell some words incorrectly. We also know the phone itself presents one or more words it thinks we meant to type. If, for instance, we're texting and type out **"tounf"**, the phone might suggest **"young"**, because it knows that **"tounf"** isn't a word but that **'t'** is right next to **'y'** and **'f'** is right next to **'g'** on the keypad. This particular suggestion required two changes, but there aren't any words in the English language that are one character away from **"tounf"**, so **"young"** is a reasonably good suggestion (as are **"round"** and **"found"**).

Implement the recursive **ls** function (**ls** is short for **listSuggestions**), which given a **string**, lists all of the words in the English language that require no more than a threshold number of substitutions. Your implementation should code to the following prototype:

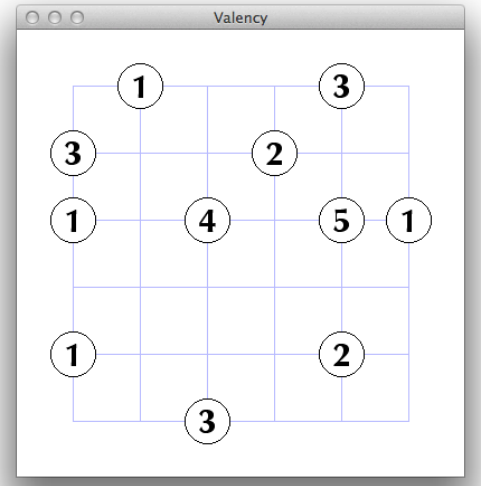
```
static void ls(const string& str, const Lexicon& english,
               const Map<char, string>& alternatives, int maxChanges);
```

str may or may not be a word in the English language, but if it is, it should be printed. Other words in the language should be printed if they require at most **maxChanges** letters to be replaced by their neighbors. **alternatives** has 26 keys—one for each lowercase letter—and each maps to a string of all of the keyboard letters immediately adjacent to it—that is, what we consider reasonable alternatives. For example, **'g'** maps to **"tyfhcvb"**, because those seven letters represent what a big thumb might have intended to hit when it tapped the **'g'**.

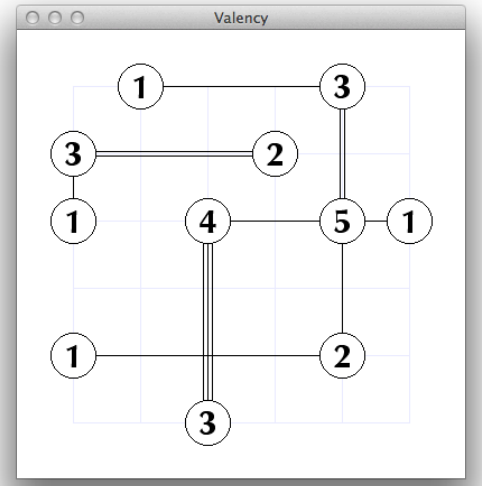
```
static void ls(const string& str, const Lexicon& english,
               const Map<char, string>& alternatives, int maxChanges) {
```

Problem 3: Valency

Valency is a puzzle one solves by repeatedly connecting pairs of circles. Any two circles can be vertically or horizontally linked one or more times, provided there are no other circles in between them. Each circle has an associated **valency** specifying the exact number of connections one must draw between it and other circles. The goal of the puzzle is to connect circles to one another so that all valency constraints are satisfied.



One such puzzle is presented on the left, and one of its solutions is presented on the right. This particularly solution makes use of 1 triple, 2 double, and 6 single connections—a total of 13 in all—to satisfy a combined valency constraint of 26.



The puzzle can be modeled as a **`Grid<int>`**, where a zero reflects the absence of a circle, and a positive value reflects the presence of one.

To help simplify the problem, you should rely on the services of a few data types and helper functions. In particular, you should assume that the following two data types have been defined for you and that operators like **<** have been overloaded so that each may be stored as entries in **`Set`**s and as keys in **`Map`**s.

```
struct coord {
   int row;
   int col;
};
```

```
struct connection {
   coord first;
   coord second;
};
```

You can also assume the following two functions have already been implemented for you:

```
static int computeValencySum(const Grid<int>& valencies);
static Set<coord> getCandidates(const coord& location, const Grid<int>& valencies);
```

`computeValencySum` returns the sum of all of the supplied **`Grid`**'s entries, and **`getCandidates`** returns the **`Set<coord>`** of all other circles with **nonzero** valency that **`location`** could potentially be connected with given the state of the supplied **`Grid`**. (Your implementation is not required to use both of these, but they're there if they help.)

Implement a recursive backtracking **solve** routine that returns **true** if and only if the referenced Valency puzzle—encoded as a **Grid<int>** by the name of **valencies**—can be solved. When **true** is returned, the referenced **connections** should contain all of the connections (mapped to their multiplicity) that solve the puzzle. For the puzzle presented above, **solve** should return **true** and update the referenced **Map** with 9 **connection**s as keys. 6 of the 9 **connection**s should map to 1 (because 6 of the nine pairings are singly connected), 2 of the 9 should map to 2 (because two pairs are doubly connected), and the 9th **connection** should map to a 3. (Be sure to remove any temporarily inserted **connection**s that ultimately map to 0.) If **false** is returned, then the state of the referenced **Map** is irrelevant.

```
static bool solve(Grid<int>& valencies, Map<connection, int>& connections) {
```