CS106X Practice Midterm

Exam Facts:

When: Thursday, February 7th from 7:00 - 8:30 p.m.
Where: Building 370, Room 370

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: Regular Expressions and String Matching

A regular expression—or regex, for short—is a **`string`** used to pattern match words in the English language. The simplest regular expressions consist of just lowercase letters, but they're also allowed to contain one or more **character sets** like **`[a-z]`**, and the presence of **`[a-z]`** in a regular expression matches any lowercase letter. Here're a few examples of regular expressions and the English words that match them:

regex	*matches*
`and`	and
`[a-z]lur`	blur, slur
`wil[a-z]`	wild, wile, wili, will, wily, wily
`m[a-z][a-z]m`	maim, malm, marm, mumm
`x[a-z][a-z][a-z]x`	xerox
`[a-z]x[a-z]`	axe, exo, oxo, oxy

The notion of a character set can be generalized to specify one or more smaller ranges to represent sets of lowercase letters, as with:

character set	*possible characters*
`[a-g]`	**`abcdefg`**
`[c-gmw-z]`	**`cdefgmwxyz`**
`[aeiou]`	**`aeiou`**
`[x-za-bp]`	**`abpxyz`**

Note that isolated characters can sit among zero or more ranges to compactly express a small set of characters, as I do with the three of the four sample character sets above. This notation allows us to match a more constrained set of English words:

regex	*matches*
`m[aeiou][x-z]`	max, may, mix, miz, moy, moz, mux
`z[a-cor-z][a-gkn-p]`	zag, zap, zoa, zoo
`[a-c][d-g][h-m][n-q][r-z]`	adios, agios, aglow, below

Finally, an asterisk (i.e., one of these things: **`'*'`**) can follow any character or character set as an instruction that the single character or character set preceding it can be skipped and go unmatched, be matched exactly once, or be matched an arbitrarily large number of times.

What can regexes look like now, and what strings do they match? Here are some examples:

- **aa[a-z]*** matches all those words that begin with aa, including *aa*, *aah*, *aahed*, *aardvark*, *aardvarks*, *aarti*, and *aasvogel*. The **[a-z]*** portion of **aa[a-z]*** can match the empty string, a single letter, or an arbitrary string of length 2 or more.
- **[a-z]*zz[a-z]*** matches all of those words that contain a zz somewhere, including *buzz*, *jacuzzi*, *pizzelle*, *sizzle*, *spazzing*, *zyzzyvas*, and *zzzs*.
- **[a-z]*zz[a-z]*zz[a-z]*** matches all of those words containing two independent double z's. This list is pretty small, but it's nonempty! It matches exactly 11 words, and *bezzazz*, *pizzazz*, and *razzamatazz* are among them.
- **[a-g]*** matches all those words that can be formed using just the first seven letters of the alphabet, including *begged*, *cabbage*, *deface*, *defaced*, *feedbag*, and *gaffed*. Musicians love these words, because they can be formed using just the notes of a C major scale.
- **[aeiou][aeiou][aeiou][aeiou]*** matches all of the English words of length 3 or more that contain only the five principal vowels.
- **[a-z]*a[a-z]*e[a-z]*i[a-z]*o[a-z]*u[a-z]*y[a-z]*** matches the six words that contain all six vowels (this time counting y) where a, e, i, o, u, and y appear in that order. Congratulations to *abstemiously*, *adventitiously*, *autoeciously*, *facetiously*, *halfseriously*, and *sacrilegiously* for being part of this distinguished set.

For this problem, you'll be led through the decomposition of a recursive function called **matches** that decides whether a regex matches a string of lowercase letters (presumably a word in the English language). Over the course of the problem, we'll confirm you're fluent with C++ strings and pass-by-reference.

a. Your implementation of **matches** should benefit from a helper function called **expand**, which takes a single character set and returns a sorted string of all of the lowercase letters it expands to, as with:

set	expand(set)
[a-g]	abcdefg
[x-ya-g]	abcdefgxy
[a-empw-z]	abcdempwxyz
[aeiou]	aeiou
[a-ea-ed-fa-eeeee]	abcdef

Your implementation should be able to handle redundancies like those you see in the last example above, and the string of lowercase letters returned should be sorted in lexicographic order. You should assume that the first character is always `'['`, the last character is always `']'`, there's at least one character between the `'['` and the `']'`, and that the character set identifies only lowercase letters and is otherwise well formed. This part doesn't involve recursion, but it will test your ability to manipulate strings.

```
static string expand(const string& set) {
```

b. Your implementation of **matches** will also benefit from a second helper function called **split**, which takes a nonempty regular expression and pulls off the portion that might be matched by a word's first character. Here's the interface you're coding to:

```
static void split(const string& regex, string& first,
                  bool& starred, string& rest);
```

Assuming that **first** and **rest** are **string**s and **starred** is a **bool**, the following illustrates how **first** and **rest** would be populated for the provided regexes:

regex	**split(regex, first, starred, rest)**
awxyz	**first** gets **"a"**, **starred** gets **false**, **rest** gets **"wxyz"**
[ae]*w*	**first** gets **"[ae]"**, **starred** gets **true**, **rest** gets **"w*"**
z	**first** gets **"z"**, **starred** gets **false**, **rest** gets **""**
z*	**first** gets **"z"**, **starred** gets **true**, **rest** gets **""**

To be clear, **starred** is populated with **true** if any only if the leading portion placed in **first** is optional and repeatable, and **rest** is populated with everything beyond **first** and, if present, the companion *.

```
static void split(const string& regex, string& first,
                  bool& starred, string& rest) {
```

c. Using the **expand** and **split** functions you've already implemented, present your implementation of **matches**, which uses *recursive backtracking* to decide whether the supplied regex matches the supplied word. Because backtracking is required, you should only make as many recursive calls as needed in order to produce a **true** or **false**. This is your chance you convey your understanding of recursive backtracking, pass by reference, and string manipulation, all in the same problem.

```
static bool matches(const string& regex, const string& word) {
```