

Section Handout

Discussion Problem 1: Publishing Stories

Social networking sites like Facebook, LinkedIn, Orkut, Friendster, and MySpace typically record and publish stories about actions taken by you and your friends. Stories such as:

John Dixon accepted your friend request.
Jeff Barbose is no longer in a relationship.
Scott James wrote a note called "The Two Percent Solution".
Arlene Heitner commented on Melodie Bowsher's video.
Antonio Melara gave The French Laundry a 5-star review.

are created from story templates like

{name} accepted your friend request.
{name} is no longer in a relationship.
{name} wrote a note called "{title}".
{name} commented on {target}'s video.
{actor} gave {restaurant} a {rating}-star review.

The specific story is generated from the skeletal one by replacing the tokens—substrings like "**{name}**", "**{title}**", and "**{rating}**"—with event-specific values, like "**John Dixon**", "**The Two Percent Solution**", and "**5**". The token-value pairs can be packaged in a **Map<string>**, and given a story template and a data map, it's possible to generate an actual story.

Write the **substituteTokens** function, which accepts a story template (like "**{actor} gave {restaurant} a {rating}-star review.**") and a **Map<string>** (which might map "**actor**" to "**Antonio Melara**", "**restaurant**" to "**The French Laundry**", and "**rating**" to "**5**"), and builds a string just like the story template, except that the tokens have been replaced by the text they map to.

Assume the following is true:

- '**{**' and '**}**' exist to delimit token names, but won't appear anywhere else. In other words, if you encounter the '**{**' character, you can assume it marks the beginning of a token that ends with a '**}**'.
- We guarantee that all tokens are in the **Map<string>**. You don't need to do any error checking.
- Note that the '**{**' and the '**}**' aren't included in the map. "**{name}**", for instance, is replaced with whatever "**name**" identifies in the map.

The prototype is:

```
string substituteTokens(string storyTemplate, Map<string>& data);
```

Discussion Problem 2: Superwords

A superword is an English word where **every** prefix and **every** suffix is also a word (assuming that all the single letters—"a", "b", "c", etc.—are English words). For instance, "**amuser**" is a superword, because all prefixes ("**a**", "**am**", "**amu**", "**amus**", and "**amuse**") and all suffixes ("**muser**", "**user**", "**ser**", "**er**", and "**r**") are English words as well. Neat!

Implement the **collectLongestSuperwords**, which given a reference to a **Lexicon**, returns all of the longest superwords as a **Vector<string>**. Your approach: You should iterate over the entire **Lexicon**, deciding whether each word is a superword, and if it's among the longest of superwords you've seen so far, you should add it to the **Vector<string>**. [You should only return the longest superwords, so that all strings in the return **Vector<string>** are of the same length.]

```
Vector<string> collectLongestSuperwords(Lexicon& english);
```

Lab Problem 1: URL Parameter Map

A URL has many components. a protocol, a domain, and a file path are almost always included. Sometimes the URL includes a **query**, which the portion of the URL that appears after the **?** and (if present) before the **#** (which defines the URL's hash). Here are some examples:

- <http://www.google.com/ig?hl=en&source=iglk>
- <http://www.facebook.com/profile.php?id=1160&viewas=214707>
- <http://store.apple.com/us/browse/family/iphone?mco=MTAyNTM5ODU>

And while you may recognize the **?** as a common character within URLs, you may not realize that the part following the **?** is the serialization of a **Map<string>**. The query string is an **&**-delimited string of key-value pairs, where each key-value pair takes the form **<key>=<value>**. When a web server gets an HTTP request, it digests the query string and re-hydrates it into **Map<string>** form, and uses that map to programmatically shape how the response page is generated. That's why www.google.com/ig?hl=en&source=iglk generates English, and www.google.com/ig?hl=fr&source=iglk generates French. One little wrinkle: the value is technically optional, so that query strings like **type=5&seeall** and **source_id=9074&read=** are legitimate. When a key is present without a value, then the value is arbitrary and the **presence** of the key is the only thing of interest. In such cases, the **Map<string>** should include the key and attach an arbitrary value to it.

Write a function that takes a legitimate URL, extracts the query string, and returns a **Map<string>** with all of its key-value pairs. You can safely assume that the URL is properly formatted, and that the structure of an URL is no more complicated than what's described here. Make sure you deal with URLs that don't include a query string, and URLs that include a hash.

```
Map<string> extractQueryMap(string url);
```

This problem is as much about string manipulation as it is about the **Map**, so don't be surprised if you're using some advanced string methods that haven't come up in any previous examples.

Ensure that:

- substrings of the form **<key>=<value>** contribute a relevant key-value pair to the map when **<value>** is one or more characters long.
- substrings of the form **<key>=** and **key** contribute the relevant key and an arbitrary value to the map.
- you ignore anything beyond the optional hash, the start of which is marked by '#'.
So:

- **http://www.google.com/mail/?shva=1&hl=en**, and
- **http://www.google.com/mail/?shva=1&hl=en#spam**

have the same parameter map.

The lab code has been set up as a unit test that exercises your **extractQueryMap** function to confirm that it's working in a variety of situations. You're to complete the implementation of **extractQueryMap** so that all unit tests pass. I've included my implementation of an **explode** function that does some automatic tokenizing around the provided delimiter character. You'll be able to figure it out. ☺

Lab Problem 2: Chain Reactions

Chain Reaction is a popular one-player game making its way through the Internet and various mobile platforms. In our version, we've given a collection of locations—**(x, y)** coordinates in the plane—of **land mines**. The detonation of any single land mine prompts all land mines within a certain distance to simultaneously detonate a second later, which themselves set off more land mines another second later, and so forth, and so forth. The chain reaction continues until there are no active land mines, or until none of the remaining land mines happen to fall within the threshold distance of those that have exploded.

- You get 0 points for the land mine you initially [and manually] detonate.
- You get 100 points for each land mine that detonates at the one-second mark.
- You get 400 points for each land mine that detonates at the two-second mark.
- You get 900 points for each land mine that detonates at the three-second mark.
- In general, you get $100n^2$ points for each land mine that detonates at the n -second mark.

Implement the core of the Chain Reaction game, as described above. The user gets to select a land mine that prompts the chain reaction—optimal or not—and computes the score. The user is then prompted with another set of mines, and then another, and then another, until the user quits the application.

I've included a good amount of starter code that implements most of the game, save for the actual sweep that emulates the chain reaction. I've also included a sample application as well, so you know how the starter code and the final solution differ.