

Section Handout

Problem 1: String Explosions

Given a string **str** and a character **delim**, write a function called **Explode** which returns the "explosion" of **str** in a **Vector<string>**, which is the ordered collection of **str**'s **delim**-delimited substrings. Here are some examples of how **Explode** should work:

```
Explode("171.64.42.111", '.');  
should return ["171", "64", "42", "111"]
```

```
Explode("usr/class/cs106x/WWW", '/');  
should return ["usr", "class", "cs106x", "WWW"]
```

```
Explode("XOXXOOOXOXX", 'X')  
should return ["", "O", "", "OOO", "", "OO", ""]
```

Note the last example makes it clear what happens when the **delim** characters appear on either end of the **str**, or when two instances of **delim** appear consecutively. In general, a string **str** with **k** instances of **delim** returns a **Vector<string>** of length **k + 1**.

```
Vector<string> Explode(string str, char delim);
```

Problem 2: 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 with map creation, so don't be surprised if you're using some advanced string methods that haven't come up in any previous examples.

Problem 3: Happy Numbers

Starting with any positive integer n , replace n with the sum of the squares of its digits, and repeat the process until the number equals 1, or until you arrive at a previously arrived at number (and have thus entered a cycle). Numbers eventually leading to 1 are called happy numbers, and the rest are called unhappy, or sad, numbers.

Using a **Set<int>**, write the **IsHappy** predicate function, which accepts a positive number and returns true if and only if the number is happy, and false otherwise.

```
bool IsHappy(int n);
```

Problem 4: FilterPossibilities

Implement the **FilterPossibilities** function, which takes a reference to a **Set<string>** called **words** and a reference to a **Map<int>** called **letters**, and removes strings from **words** unless they meet the requirements imposed by the **letters** map. The **letters** map maps characters (stored as one-character strings) to the maximum number of times that letter may appear in a word if it's to be retained by the **words** set. (If the character doesn't appear as a key in the map, then there are no restrictions on that character.)

For instance, if the **words** set contains "**abacus**", "**gizmo**", "**holler**", "**llama**", and "**portions**", and the **letters** map maps "**a**" to 1, "**l**" to 1, and "**p**" to 3, then **FilterPossibilities** would remove "**abacus**" (too many a's), "**holler**" (too many l's), and "**llama**" (too many of both!), leaving just "**gizmo**" and "**portions**".

```
void FilterPossibilities(Set<string>& words, Map<int>& letters);
```