Section Handout 2

This week's section handout is all about collections. There are two problems here – one that probes the mysteries of the English language, and one that probes the mysteries of life itself.

1. Xzibit Words

Some words contain other words as substrings. For example, the word "pirates" contains a huge number of words as substrings:

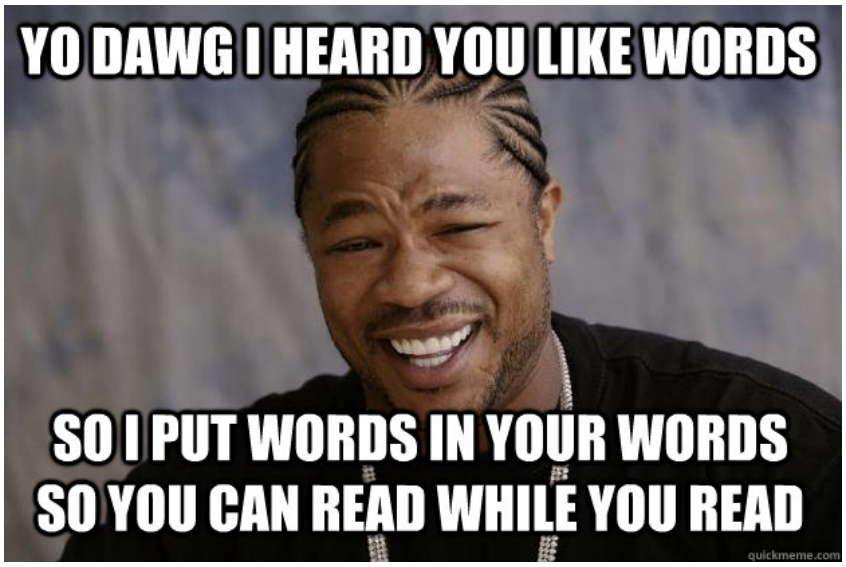
a	I	rat
at	irate	rate
ate	pi	rates
ates	pirate	
es	pirates	

Note that "pirates" is a substring of itself. The word "pat" is not considered a substring of "pirates," since even though all the letters of "pat" are present in "pirate" in the right order, they aren't adjacent to one another.

Let's call a word an "Xzibit word" if it contains a large number of words as substrings. Write a function

```
string mostXzibitWord(Lexicon& words);
```

that accepts as input a `Lexicon` of all words in English, then returns the word with the largest number of words contained as substrings.



2. RNA Protein Codes

RNA (ribonucleic acid) is a chemical that can encode genetic information. Plant and animal cells use RNA for a variety of cell functions, while viruses often use RNA as their primary genetic storage.

Each strand of RNA consists of a series of base pairs – adenine (A), guanine (G), uracil (U), and cytosine (C) – and a strand of RNA can be thought of as a string over those four letters. Because of this, computational geneticists often treat DNA and RNA as strings and run algorithms on those strings to deduce their properties.

RNA is used by a cell to encode **proteins**, biological molecules essential to cell function. Each protein consists of a series of **amino acids** that, strung together, serve some complex purpose. The actual letters in an RNA strand spell out what amino acids need to be combined together to produce the overall protein. So how exactly are these amino acids represented? In RNA, letters are grouped into clusters of three letters called **codons**. Each codon encodes a specific choice of amino acid. When decoding RNA, the cell reads these codons in sequence and assembles the protein one amino acid at a time. For example, the RNA strand

`GGGAUGAAUAUCUCGGCG`

would be treated at this sequence of three-letter codons:

`GGG` `AUG` `AAU` `AUC` `UCG` `GCG`

The cell would then use the codons to determine what amino acids to string together into a protein. In this case, these codons represent the following sequence of amino acids:

`GGG`	`AUG`	`AAU`	`AUC`	`UCG`	`GCG`
Glycine	Methionine	Asparagine	Isoleucine	Serine	Alanine

So the generated protein would have amino acids ordered as glycine, then methionine, then aspargine, then isoleucine, then serine, and finally alanine.

An important detail is that each strand of RNA does not encode just one protein; typically, a single strand of RNA encodes many different proteins. How, then, does the cell know where each protein starts and stops? There is an ingenious system set up for just this purpose. In an RNA strand, the codon `AUG` has two meanings – it can code for methionine (as shown above), but it also acts as a special "start of protein" marker. As a cell scans across an RNA strand, if it encounters the codon `AUG`, it begins assembling a protein starting at that location. It then continues to assemble the protein one amino acid at a time by decoding codons in sequence. The cell stops assembling base pairs when it encounters one of three "stop" codons (`UAA`, `UAG`, or `UGA`) that act as an "end-of-protein" marker. The cell can then keep scanning across the RNA until it finds another `AUG` start codon, at which point the process repeats. For example, consider this RNA strand:

`GCAUGGAUUAAUAUGAGACGACUAAUAGGAUAGUUACAACCCUUAUGUCACCGCCUUGA`

This would be decoded as follows:

`GC`	Skip letters until we find AUG.
`AUG``GAU`**`UAA`**	Read from AUG until we hit a stop codon (here, UAA)
`U`	Skip letters until we find AUG.
`AUG``AGACGACUAAUAGGA`**`UAG`**	Read from AUG until we hit a stop codon (here, UAG)
`UUACAACCCUU`	Skip letters until we find AUG.
`AUG``CACCGCCU`**`UGA`**	Read from AUG until we hit a stop codon (here, UGA)

Your job is to write a function

```
Vector<string> findProteins(string& rna, Map<string, string> codons);
```

that accepts as input a string of RNA and returns a **Vector** of all the proteins that would be formed from that RNA. The first parameter represents the actual string of RNA, and the second parameter is a **Map** from three-letter codons to the name of the amino acid they represent (or the special string **"stop"** if the codon is a stop codon). For example, running this function on the above RNA strand would return the **Vector** holding the stings

methionine, asparitic acid	(encoded by AUGGAU**UAA**)
methionine, arginine, arginine, leucine, isoleucine, glycine	(encoded by AUGAGACGACUAAUAGGA**UAG**)
methionine, serine, proline, proline	(encoded by AUGUCACCGCCU**UGA**)

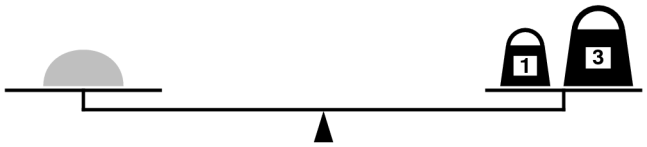
You can assume that all the proteins are properly terminated, which means that if you find an AUG codon then there will be a matching stop codon before the end of the protein.

3. Weights and Balances (Chapter 8, exercise 6, page 380)

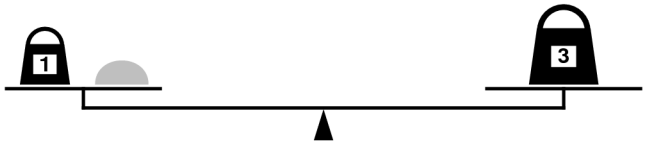
> I am the only child of parents who weighed, measured, and priced everything; for whom what could not be weighed, measured, and priced had no existence.
>
> —Charles Dickens, *Little Dorrit,* 1857

In Dickens's time, merchants measured many commodities using weights and a two-pan balance—a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:



It is somewhat more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Write a recursive function

```
bool isMeasurable(int target, Vector<int> & weights)
```

that determines whether it is possible to measure out the desired target amount with a given set of weights, which is stored in the vector **weights**.

As an example, suppose that the variable **sampleWeights** has been initialized as follows:

```
Vector<int> sampleWeights;
sampleWeights += 1, 3;
```

Given these values, the function call

```
isMeasurable(2, sampleWeights)
```

should return **`true`** because it is possible to measure out 2 ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

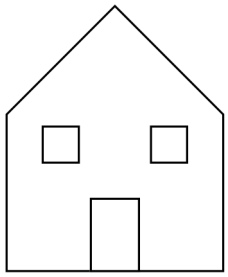
```
isMeasurable(5, sampleWeights)
```

should return **`false`** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

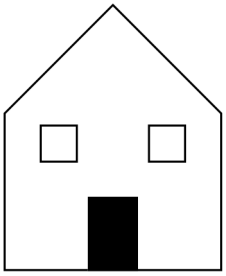
4. Filling a Region (Chapter 9, exercise 4, page 421)

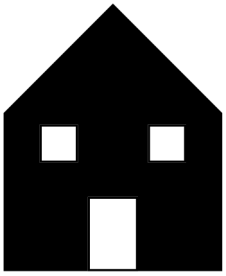
Most drawing programs for personal computers make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

For example, suppose you have just drawn the following picture of a house:

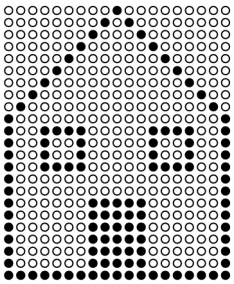


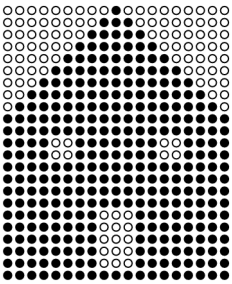
If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:





In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called ***pixels***. On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:





It is easy to represent a pixel grid using the type **`Grid<bool>`**. White pixels in the grid have the value **`false`**, and black pixels have the value **`true`**. Given this representation, write a function

```
void fillRegion(Grid<bool> & pixels, int row, int col)
```

that simulates the operation of the paintbucket tool by painting in black all white pixels reachable from the specified row and column without crossing an existing black pixel.

5. Generating Multiword Anagrams

Even before we got to recursion, I showed how it was possible to use the map data structures to find all *anagrams* of a given word, where an anagram is simply a string with the same letters in a different order. Recursion makes it possible to do much more with the anagram idea. In particular, you can write programs that extend the idea of anagrams to multiple words. Dan Brown used several multiword anagrams as clues in his 2003 best-selling thriller *The Da Vinci Code,* so that (if you ignore spaces) you discover that

o draconian devil	↔	leonardo da vinci
oh lame saint	↔	the mona lisa
so dark the con of man	↔	madonna of the rocks

Your job in this problem is to write a function

```
bool findAnagram(string letters, Lexicon & english,
                 Vector<string> & words);
```

that takes a string of letters with all spaces and punctuation removed, the well-worn lexicon of English words, and a vector to hold the words of the generated anagram. As soon as **`findAnagram`** discovers a series of English words that are an anagram of the specified letters, it should return **`true`** with those words stored in the vector. If no anagrams exist, **`findAnagram`** should return **`false`**. To avoid returning anagrams that consist of boring short words, you should also include a constant in your implementation setting the minimum size of a word.