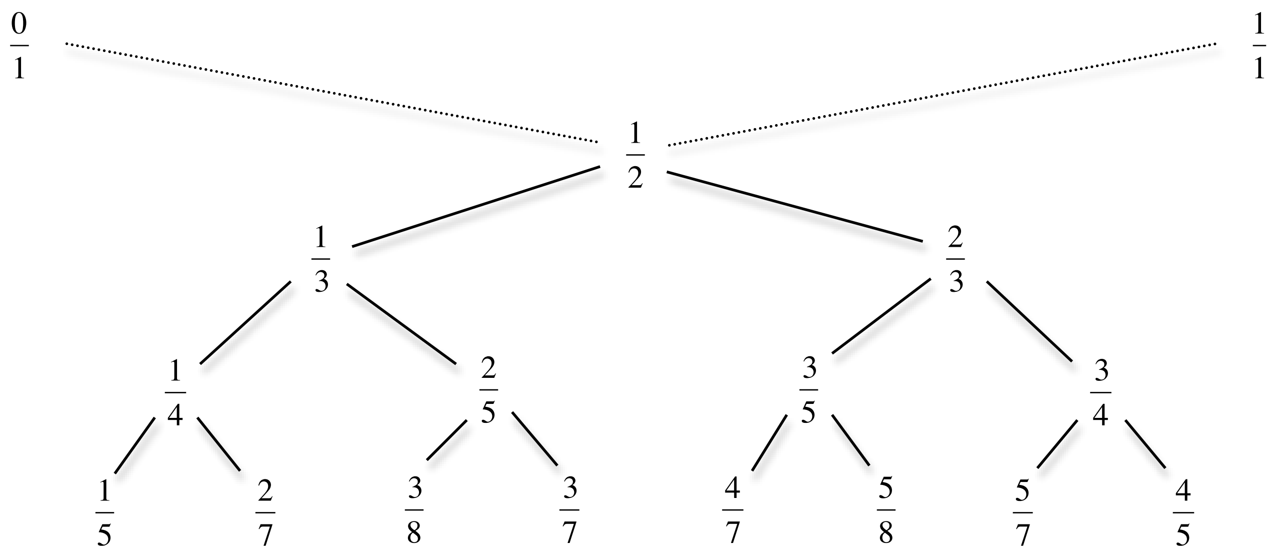
Section Handout

Discussion Problem 1: Farey Series, Take II

Let's circle back to the Farey series we discussed a few weeks ago, and work on a new, more interesting implementation of `generateFareySeries`. (Recall that the Farey series of order n is the ordered set of all reduced fractions in (0, 1) with denominators of n or less.) For this version, we are going to rely on the following construction, which is an adaptation of something known as the Stern-Brocot tree:



Each fraction is $\frac{n_L + n_R}{d_L + d_R}$, where $\frac{n_L}{d_L}$ is the closest ancestor up and to the left, and $\frac{n_R}{d_R}$ is the closest ancestor up and to the right. $\frac{3}{7}$, for example, is produced from $\frac{2}{5}$ (first ancestor up and to the left) and $\frac{1}{2}$ (first ancestor up and to the right.)

This manner of enumerating fractions has three interesting properties (stated without proof):

- each fraction generated by the construction is in reduced form,
- every single reduced fraction between 0 and 1 will eventually be formed, and
- $\frac{n_L}{d_L}$ is always less than $\frac{n_L + n_R}{d_L + d_R}$, and $\frac{n_L + n_R}{d_L + d_R}$ is always less than $\frac{n_R}{d_R}$.

By blindly trusting the construction and the three properties just mentioned, provide a recursive implementation of **`generateFareySeries`** that succeeds in populating a **`Vector<fraction>`**, and does so in time that's proportional to the length of the series being generated. **`generateFareySeries`** will need to declare the **`Vector<fraction>`** and pass it by reference to a helper function that actually does the recursion and populates the vector in such a way that **`Vector::add`** is the only dynamic vector method you ever call, and the fractions are laid down in increasing order.

```
Vector<fraction> generateFareySeries(int n);
```

Discussion Problem 2: NEWS Puzzles

NEWS puzzles are maze-like constructions where the goal is to navigate from one location to another. You'd be free to wander up, down, left, and right as you please, except that moves are constrained by a set of rules. Some are obvious, but some aren't. Here they are:

- If a particular coordinate is labeled with N, E, W, or S, then the next coordinate must reside due north, east, west, or south, respectively.
- If a particular coordinate is labeled with a +, then the next coordinate might be in any of the four major compass directions.
- If a particular coordinate isn't labeled, then that coordinate is considered a bridge, and you must pass straight through that coordinate—that is, if you move east into an unlabeled coordinate, you must move east again.

						E	E	E	S	S	W						
				S	W		+	W		+	N	W	W				
		S	W		+	N	W	S	+	S	E	E		E	S		
S	W	+	S	W	N	S	+		W	+		W	+		+	E	S
+	E	E		S		W	W	+	E	N	+	N		W	S	+	
E	E		+		N	E	E		+	N	N	+	E	+	S	N	W
		N	W	W	S	+	E	S	E		E	N		N	W		
				N	W		+		W	+	E	E	N				
						N	W	E	E	N	W						

The puzzle itself is modeled as a **`Grid<rule>`**, where a **`rule`** is encoded via an enumerated type like so:

```
enum rule {
  North,
  East,
  West,
  South,  // rule limiting moves to one compass direction
  Bridge, // rule identifying a coordinate as a bridge
  Any,    // rule allowing movement in any four directions
  OffLimits // "rule" that says the coord isn't part of the puzzle
};
```

Pair the above with the familiar definition of a **coord**:

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

and you're equipped to implement a recursive backtracking routine which decides whether a path from one coordinate to another exists while respecting all rules. For simplicity, you should assume that:

- the **coord** type magically works with all relational operators, including < and ==, so you don't have to write any comparison functions.
- the initial location houses either an N, E, W, S, or a +, so that there's no confusion as to how you can initiate a search.

Implement the **pathExists** predicate function, which returns **true** if and only if one can travel from **start** to **finish** while respecting the rules. You are required to use recursive backtracking to explore as many possibilities as necessary to confirm or refute the existence of a path. You needn't return the path itself—just the **true** or the **false**. You should wrap **pathExists** around a helper function that actually manages the recursion.

```
bool pathExists(Grid<rule>& maze, coord start, coord finish);
```

Lab Problem 1: Generating Anagrams

An anagram is a word or phrase formed by the rearrangement of the letters of another word or phrase. Here are a few of the funnier ones I found a long time ago while surfing **http://wordsmith.org/anagram**.

partial men is an anagram of **parliament**.
Old West Action is an anagram of **Clint Eastwood**
The American First Lady, Laura Bush is an anagram of **I am after a cuter husband: Hillary's!**
Firefox browser is an anagram of **fix errors of Web**

We're interested in writing code that, given a string of lowercase letters (i.e. **"oldwestaction"**), manages to find any one of its anagrams. Here's the prototype I want you to work with:

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

Implement the **findAnagram** function as described above. I've supplied a read-find-print loop program that you can use to test your implementation. Remember that your **findAnagram** may generate words in a different order than mine does, and your implementation might (and very well may) produce a different collection of words altogether. It's not important to replicate the sample output below; it's only important that you find anagrams when I do, and that you don't when I don't. Remember that the minimum word length is 4 (which explains why **tap**—illustrated below—doesn't generate anything.)

```
Here's a hip little program that figures out how to permute a string of
characters into a word or concatenation of words.

Enter some letters [or just hit enter to quit]: tap
That's not all that good a collection of letters, because I can't find any
anagrams. Try something else...
Enter some letters [or just hit enter to quit]: epts
Those letters can be rearranged to form the word "pets".
Enter some letters [or just hit enter to quit]: happy
Those letters can be rearranged to form the word "happy".
Enter some letters [or just hit enter to quit]: pspay
Those letters can be rearranged to form the word "sappy".
Enter some letters [or just hit enter to quit]: pspayyhda
Ooo, we found an anagram consisting of 2 words: "days" "happy"
Enter some letters [or just hit enter to quit]: pspayyhdayhda
Ooo, we found an anagram consisting of 3 words: "dyad" "hypha" "yaps"
Enter some letters [or just hit enter to quit]: yseireahfd
Ooo, we found an anagram consisting of 2 words: "fished" "year"
Enter some letters [or just hit enter to quit]: reifusaoopweisa
Ooo, we found an anagram consisting of 3 words: "woofs" "asea" "euripi"
Enter some letters [or just hit enter to quit]: onomatopoeia
Those letters can be rearranged to form the word "onomatopoeia".
Enter some letters [or just hit enter to quit]: clinteastwood
Ooo, we found an anagram consisting of 3 words: "wood" "tats" "cline"
Enter some letters [or just hit enter to quit]: zxfghro
That's not all that good a collection of letters, because I can't find any
anagrams. Try something else...
Enter some letters [or just hit enter to quit]:
```

This one takes a long time.