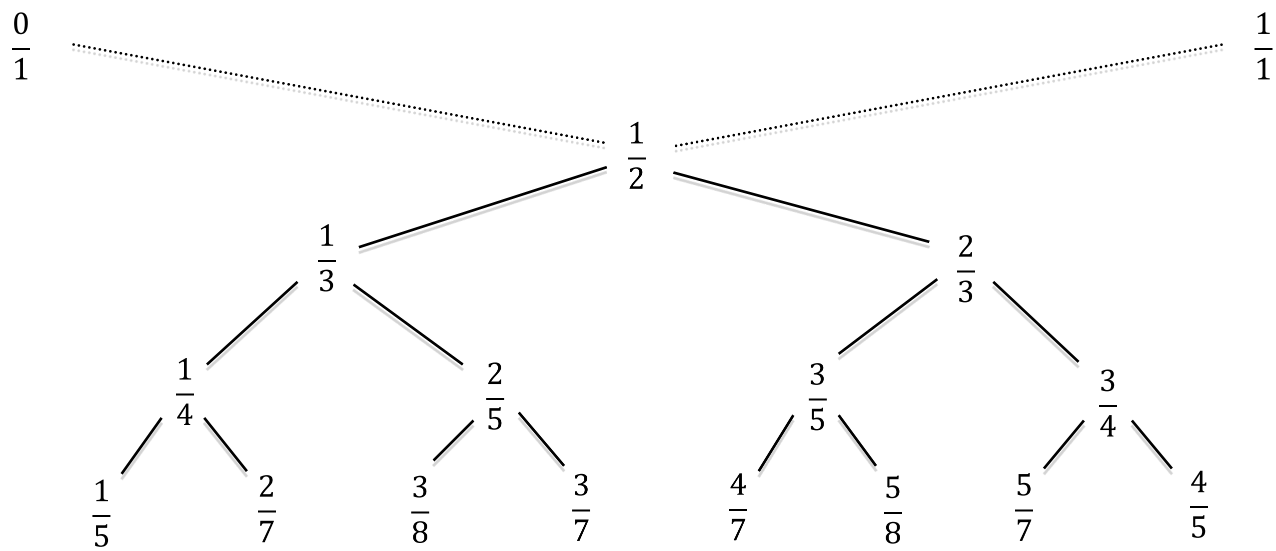
CS106X
Autumn 2019

Handout 12
October 7th, 2019

Section Handout

Problem 1: Farey Series, Take II

Let's circle back to the Farey series I presented in lecture and work on a new, more interesting implementation for **`generateFareySeries`**. (Recall that the Farey series of order n is the ordered series 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, but you can trust that they're correct):

- 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 simply trusting the construction and the three properties mentioned above, provide a recursive implementation of **`generateFareySeries`** that succeeds in populating a

`Vector<fraction>` with the Farey series of order n (where **`n`** is supplied) 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 **`add`** is the only dynamic method you ever call, and the fractions are laid down in increasing order.

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

Problem 2: Twiddles

Two English words are considered *twiddles* if the letters at each position are either the same, neighboring letters, or next-to-neighboring letters. For instance, **`sparks`** and **`snarls`** are twiddles. Their second and second-to-last characters are different, but **`p`** is just two past **`n`** in the alphabet, and **`k`** comes right before **`l`**. A more comical example: **`craggy`** and **`eschew`**. They have no letters in common, but **`craggy`**'s **`c`**, **`r`**, **`a`**, **`g`**, **`g`**, and **`y`** are **`-2`**, **`-1`**, **`-2`**, **`-1`**, **`2`**, and **`2`** away from the **`e`**, **`s`**, **`c`**, **`h`**, **`e`**, and **`w`** in **`eschew`**, respectively. And to be clear, **`a`** and **`z`** are **not** next to each other in the alphabet—there's no wrapping around.

Write a recursive procedure called **`listTwiddles`** which accepts a string **`str`** and a reference to an English language **`Lexicon`**, and prints out all those English words that just happen to be **`str`**'s twiddles. You'll probably want to write a wrapper function. (Note: a word is considered to be a twiddle of itself, so it's okay to print it.)

```
static void listTwiddles(const string& str, const Lexicon& lex);
```

Problem 3: Letter Rectangles and Words

You are given a large collection of short, fat rectangles, where each half of each rectangle contains a single letter, as with:

Given the option to rearrange, ignore, and rotate pieces, you're charged with the task of identifying all of the even-length English words that can be formed by chaining together some subset of the pieces (where some may have been rotated). For the above set of pieces, the list of printed words should surely include **`"plum"`**, since the third-to-last rectangle can be placed after the second-to-last rectangle (rotated so that the **`'p'`** precedes the **`'l'`**) to form **`"plum"`**. Given the above set of rectangles, you should also identify fun words like **`"allele"`**, **`"lark"`**, **`"muscle"`**, **`"scales"`**, and **`"umbrella"`**, in addition to quite a few others. Note that each rectangle can be used at most one time per word, so that words like **`"sees"`** and **`"museum"`** can't be formed.

Collectively implement the recursive function **`gatherWords`**, which accepts references to a **`Vector<string>`** called **`rects`** (where each **`string`** is two characters), a **`Lexicon`** constant called **`english`**, and an initially empty **`Lexicon`** called **`words`**, and populates **`words`** with the collection of those words, and only those words, that can be formed using the rectangles in **`rects`**. You should implement this using a wrapper function.

```
static void gatherWords(const Vector<string>& rects,
                        const Lexicon& english, Lexicon& words);
```

Problem 4: Making Change

For this problem, implement the following function:

```
static int countWaysToMakeChange(const Vector<int>& denominations, int amount)
```

The **`countWaysToMakeChange`** routine recursively computes the number of ways to make change for the specified amount given an unlimited number of coins of the specified denominations. For instance, **`countWaystoMakeChange({1, 5}, 10)`** should return 3, since we can make ten cents with ten pennies, one nickel and five pennies, or two nickels.

```
int main() {
   Vector<int> denominations;
   denominations += 25, 10, 5;
   cout << "Number of ways to make change for a dollar using " << denominations
        << ": " << countWaysToMakeChange(denominations, 100) << endl;
   denominations += 1;
   cout << "Number of ways to make change for a dollar using " << denominations
        << ": " << countWaysToMakeChange(denominations, 100) << endl;
   return 0;
}
```

Once properly implemented, the above **`main`** function should output the following:

```
Number of ways to make change for a dollar using {25, 10, 5}: 29
Number of ways to make change for a dollar using {25, 10, 5, 1}: 242
```