CS106X Handout 15S
Autumn 2012 October 10th – 12th, 2012

Section Solution

Discussion Problem 1 Solution: Farey Series, Take II

```
static void generateFareySeries(Vector<fraction>& series,
                                const fraction& left, const fraction& right,
                                int n) {
   fraction mediant = {
      left.numerator + right.numerator,
      left.denominator + right.denominator
   };

   if (mediant.denominator > n) return; // denominator is too big? bail!
   generateFareySeries(series, left, mediant, n);
   series.add(mediant);
   generateFareySeries(series, mediant, right, n);
}

static Vector<fraction> generateFareySeries(int n) {
   Vector<fraction> series;
   fraction zero = {0, 1};
   fraction one = {1, 1};
   generateFareySeries(series, zero, one, n);
   return series;
}
```

Discussion Problem 2 Solution: Twiddles

Key observation: finding twiddles is the same as fixing the first letter (one of up to five possibilities) and appending some twiddle of the remaining letters. A **`'c'`** at **`str`**'s position 0, for instance, encodes the fact that **`'a'`**, **`'b'`**, **`'c'`**, **`'d'`**, or **`'e'`** might contribute to a potential twiddle at position 0. And for each of those five possibilities at position 0, there are five contributions at position 1, and for each of those 25 possible possibilities between 0 and 1 combined, there are five independent contributions that might be made at position 2, and so on, and so on.

```
static void listTwiddles(const string& str, const Lexicon& lex) {
    listTwiddles("", str, 0, lex);
}
```

- The 0th argument is the empty string to clarify that no decisions made been made at the outset.
- The 2nd argument is **`0`** to be clear that **`str[0]`** is the character that tells us how me might extend the empty string into five different prefixes of length 1.

```
static void listTwiddles(const string& prefix, const string& str, int index,
                         const Lexicon& lex) {

    if (!lex.containsPrefix(prefix)) return; // not strictly necessary
    if (index >= str.size()) {
        if (lex.contains(prefix))
            cout << prefix << endl;
        return;
    }

    for (char ch = str[index] - 2; ch <= str[index] + 2; ch++) {
        if (isalpha(ch)) {
            listTwiddles(prefix + ch, str, index + 1, lex);
        }
    }
}
```

Discussion Problem 3 Solution: Letter Rectangles and Words

My implementation wraps the three-argument version around a call to a four-argument version. The overloaded version—that one that really does all of the work—keeps track of the running prefix built up by an ordered selection of (possibly rotated) rectangles leading up to the call. Initially, we haven't selected any rectangles, which is why my wrapper passes an empty string in as the 0th parameter.

```
static void gatherWords(const Vector<string>& rects,
                        const Lexicon& english, Lexicon& words) {
    Vector<string> copy = rects;
    gatherWords("", copy, english, words);
}

static void gatherWords(const string& prefix, Vector<string>& rects,
                        const Lexicon& english, Lexicon& words) {
    if (!english.containsPrefix(prefix)) // prefix is nonsense?
        return; // pretend we never made this call
    if (english.contains(prefix)) // prefix is a word?
        words.add(prefix); // incidentally print, but continue

    for (int i = 0; i < rects.size(); i++) {
        string rect = rects[i];
        rects.remove(i); // temporarily remove so it doesn't get used twice
        gatherWords(prefix + rect[0] + rect[1], rects, english, words);
        gatherWords(prefix + rect[1] + rect[0], rects, english, words);
        rects.insert(i, rect); // insert back so it can be used deeper down
    }
}
```

Lab Problem 1 Solution: Making Change

The exported **`countWaysToMakeChange`** takes two parameters, but my implementation wraps around a single call to a three-argument version. The third argument dictates the lowest index within **`denominations`** the call is allowed to use while constructing the various ways to make change. Tacking on the 0 in the wrapped call makes it clear that all indices—from index 0 forward—are fair game.

```
static int countWaysToMakeChange(const Vector<int>& denominations, int amount) {
   return countWaysToMakeChange(denominations, amount, 0);
}
```

The three-argument version partitions the total number of ways to make change into two categories—those that require one or more contributions of **`denoms.get(start)`**, and those that forbid any contributions of **`denoms.get(start)`**. (Note that we're constrained to use **`get`** instead of **`operator[]`**, because **`operator[]`** currently can't be levied against a **`const Vector`**.)

```
static int countWaysToMakeChange(const Vector<int>& denoms,
                                 int amount, int start) {
   if (amount == 0) return 1; // there's 1 way to not give any change
   if (amount < 0) return 0;  // it's impossible to make negative change
   if (start >= denoms.size()) return 0;  // no permitted denominations

   return
      countWaysToMakeChange(denoms, amount - denoms.get(start), start) +
      countWaysToMakeChange(denoms, amount, start + 1);
}
```