Section Solution

Problem 1 Solution: Farey Series, Take II

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

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

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

Problem 2 Solution: Twiddles

Key observation: finding twiddles is the same as deciding on 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, means that **'a'**, **'b'**, **'c'**, **'d'**, or **'e'** can be the leading character of a twiddle. And for each of those five possibilities, there are five possible contributions at position 1, and for each of those 25 possible possibilities for 0 and 1 combined, there are five independent possibilities that might sit at position 2, and so on, and so on.

My approach uses a wrapper function:

```
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 have been made yet.
- The 2nd argument is **0** to be clear that **str[0]** and its near neighbors are capable of extending the empty string into a string of length 1.

Wrapper functions are all about making the implicit explicit.

```
static void listTwiddles(const string& prefix, const string& str,
                         size_t 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);
      }
   }
}
```

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 0^{th} 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 add, 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
   }
}
```

Problem 4 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 onward—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[start]`**, and those that forbid any contributions of **`denoms[start]`**.

```
static int countWaysToMakeChange(const Vector<int>& denoms, int amount, int start) {
   if (amount == 0) return 1; // there's one way to produce 0 cents (deep)
   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[start], start) +
      countWaysToMakeChange(denoms, amount, start + 1);
}
```