

Section Solution

Solution 1: Making Change

```
int MakeChange(int amount, Vector<int>& denominations, int start)
{
    if (amount == 0) return 0;
    if (amount < 0) return -1;
    if (start == denominations.size()) return -1;

    int minCoins = -1;
    int minCoinsOfRestWith = MakeChange(amount - denominations[start],
                                         denominations, start);

    if (minCoinsOfRestWith != -1)
        minCoins = 1 + minCoinsOfRestWith;

    int minCoinsWithout = MakeChange(amount, denominations, start + 1);
    if ((minCoins == -1) || (minCoinsWithout != -1 && minCoinsWithout < minCoins))
        minCoins = minCoinsWithout;

    return minCoins;
}

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

Solution 2: Old-Fashioned Measuring

```
bool IsMeasurable(int target, Vector<int>& weights, int start)
{
    if (target == 0) return true;
    if (weights.size() == start) return false;

    return ( IsMeasurable(target + weights[start], weights, start + 1) ||
             IsMeasurable(target, weights, start + 1) ||
             IsMeasurable(target - weights[start], weights, start + 1) );
}

bool IsMeasurable(int target, Vector<int>& weights)
{
    return IsMeasurable(target, weights, 0);
}
```

Solution 3: Towers Of Hanoi Revisited

Version 1:

```

MoveTower(int numDisks, string start,
          string dest, string temp) {
    if (numDisks > 0) {
        MoveTower(numDisks - 1, start, temp, dest);
        MoveDisk(start, dest);
        MoveTower(numDisks - 1, temp, dest, start);
    }
}

```

Version 2:

```

MoveTower(int numDisks, string start,
          string dest, string temp) {
    if (numDisks > 0) {
        MoveTower(numDisks - 2, start, temp, dest);
        MoveDisk(start, dest);
        MoveDisk(start, dest);
        MoveTower(numDisks - 2, temp, dest, start);
    }
}

```

Version 3:

```

MoveTower(int numDisks, string start,
          string dest, string temp) {
    if (numDisks > 0) {
        MoveTower(numDisks - 2, start, dest, temp);
        MoveDisk(start, temp);
        MoveDisk(start, temp);
        MoveTower(numDisks - 2, dest, start, temp);
        MoveDisk(temp, dest);
        MoveDisk(temp, dest);
        MoveTower(numDisks - 2, start, dest, temp);
    }
}

```

Version 4:

```

MoveTower(int numDisks, string start,
          string dest, string temp) {
    if (numDisks > 0) {
        MoveTower(numDisks - 2, start, dest, temp);
        MoveDisk(start, temp);
        MoveTower(numDisks - 2, dest, start, temp);
        MoveDisk(temp, dest);
        MoveTower(numDisks - 2, start, temp, dest);
        MoveDisk(start, dest);
        MoveTower(numDisks - 2, temp, dest, start);
    }
}

```

Correct

Simply imagine the white disks as being the slightest bit smaller than their dark grey cousins. Then we have the same type of tower as we dealt with in class (to see this, forget about the fact that the disks are shaded different colors.)

Incorrect

This will not work, as it should be clear that the largest white disk will be on bottom after the two calls to MoveDisk are complete. However, the even number of recursive calls will ensure that all pairs except the bottom pair will maintain their original order.

Correct

Again, make that leap, and trust the recursion. Because the two bottom disks are moved in phases, their final orientation is exactly as it was originally. Try this with 4 disks and you'll see. (This is really just like Version 1, except that each level here does the work of two levels there.)

Incorrect

Notice that the bottom two disks are inverted, so even if the recursion were to somehow work, the bottom two disks would be placed incorrectly. The actual end result is precisely the same as that of version 2.

```

void InvertTower(int numDisks, string start, string dest, string temp)
{
    if (numDisks > 0) {;
        InvertTower(numDisks - 2, start, temp, dest);
        MoveDisk(start, dest);
        MoveDisk(start, dest); // these two calls invert the largest two
        InvertTower(numDisks - 2, temp, start, dest);
        InvertTower(numDisks - 2, start, dest, temp);
    }
}

```

I suspect this implementation will surprise a good number of you. But if the recursive call really does invert the tower atop the bottom two disks, you need an odd number of recursive calls to truly invert the tower.

Solution 4: Twiddles

Here's one implementation that keeps track of which characters have been twiddled so far, and which ones have yet to be. The wrapper function includes a third parameter called **position** to mark the dividing line between what's already being handled by an active recursive call, and which characters have yet to be changed.

```

void ListTwiddles(string str, Lexicon& lex)
{
    ListTwiddles(str, 0, lex);
}

void ListTwiddles(string str, int position, Lexicon& lex)
{
    if (position == str.length()) {
        if (lex.containsWord(str)) {
            cout << str << endl;
        }
        return;
    }

    char saved = str[position];
    for (int delta = -2; delta <= 2; delta++) {
        char ch = saved + delta;
        if (islower(ch)) { // islower technically not needed
            str[position] = ch;
            ListTwiddles(str, position + 1, lex);
        }
    }
}

```

Another approach could work like the permutations and the subsets examples we covered in class. Maintain a working prefix and recursively extend that prefix in one of five ways:

```

void ListTwiddles(string str, Lexicon& lex)
{
    ListTwiddles("", str, lex);
}

void ListTwiddles(string prefix, string remaining, Lexicon& lex)

```

```

{
    if (!lex.containsPrefix(prefix)) {
        return;
    }

    if (remaining == "") {
        if (lex.containsWord(prefix)) {
            cout << prefix << endl;
        }
        return;
    }

    for (int delta = -2; delta <= 2; delta++) {
        char ch = remaining[0] + delta;
        if (islower(ch))
            ListTwiddles(prefix + ch, remaining.substr(1), lex);
    }
}

```

The **islower** calls aren't technically necessary. It's fine to generate strings with non-alphabetic characters in them, because they'll just fail the **containsWord** test every time.

Solution 5: Generating Anagrams

```

bool FindAnagramWithFixedPrefix(string prefix, string rest,
                                Lexicon& lex, Vector<string>& words)
{
    if (!lex.containsPrefix(prefix)) return false;
    if (lex.containsWord(prefix) && prefix.length() >= 4) {
        if (rest == "" || FindAnagram(rest, lex, words)) {
            words.add(prefix);
            return true;
        }
    }

    for (int i = 0; i < rest.length(); i++) {
        if (FindAnagramWithFixedPrefix(prefix + rest[i],
                                        rest.substr(0, i) + rest.substr(i + 1),
                                        lex, words)) return true;
    }

    return false;
}

bool FindAnagram(string letters, Lexicon& lex, Vector<string>& words)
{
    return FindAnagramWithFixedPrefix("", letters, lex, words);
}

```