

Section Handout

Problem 1: Making Change

Everyone who's worked a cash register (that is to say, me, at least) has had to deal with the intellectually stimulating task of making change. The customer is owed \$.90, so do I give her 9 dimes? 3 quarters, a dime, and a nickel? 90 pennies? We're used to 25, 10, 5, and 1 cent denominations, but how might I give change if coins are worth 20, 13, 4, and 1 cents instead?

There are clearly many different configurations of coins that will work, but let's say that we're interested in the combination that uses the fewest coins. One approach would be to use as many high value coins (i.e. quarters) as possible, then move on to use dimes for what is leftover, then nickels and finally pennies if needed. This type of algorithm is known as a *greedy* algorithm, since at any given moment it makes the choice that looks best at the moment, the hope being that the locally optimal choice will lead to a globally optimal solution. However, what if I had no nickels in my change drawer and was trying to make \$.31? The greedy solution chooses 1 quarter and 6 pennies, which is worse than the optimal solution of 3 dimes and 1 penny. Clearly we need something even smarter to find the truly optimal arrangement. Recursion to the rescue!

Write a function **MakeChange** that takes an amount along with the **Vector** of available coin values. You can assume you have as many of each coin as you want (i.e. if your cash drawer has pennies, it has an infinite supply of them). The function should return the minimum number of coins required that sum to the given amount. If the amount cannot be made (for example, if you try to make \$.31 and have no pennies), the function should return -1. You do not have to print or return the coin combination, just return the minimum number of coins in the optimal configuration.

There are several different ways to recursively decompose this problem. Let your creativity lead you toward the one that makes the most sense to you.

```
int MakeChange(int amount, Vector<int>& denominations);
```

(Can you write a function that computes the total number of ways to make change using the specified denominations? That's a related but equally interesting problem.)

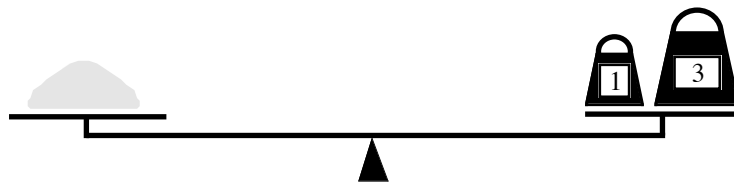
Problem 2: Old-Fashioned Measuring [Courtesy of Eric Roberts]

I am the only child of parents who weighed, measured, and priced everything; for whom what could not be weighed, measured, and priced had no existence.

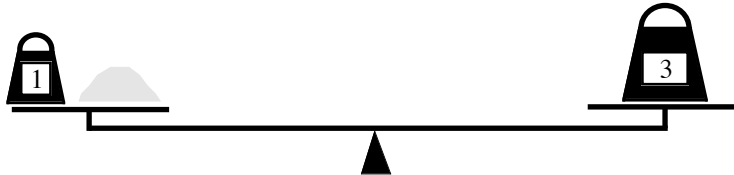
— Charles Dickens, *Little Dorrit*, 1857

In Dickens's time, merchants measured many commodities using weights and a two-pan balance—a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:



It is somewhat more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Write a recursive function

```
bool IsMeasurable(int target, Vector<int>& weights);
```

that determines whether it is possible to measure out the desired target amount with a given set of weights. The available weights are stored in the **Vector** called **weights**. For instance, the sample set of two weights illustrated above could be represented using a **Vector<int>**

```
Vector<int> weights;
weights.add(3);
weights.add(1);
```

Given these values, the function call

```
IsMeasurable(2, weights);
```

should return **true** because it is possible to measure out 2 ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

```
IsMeasurable(5, weights);
```

should return **false** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

The fundamental observation you need to make for this problem is that each weight in the **Vector** can be either:

- a. Put on the opposite side of the balance from the sample
- b. Put on the same side of the balance as the sample
- c. Left off the balance entirely

If you consider one of the weights in the **Vector** and determine how choosing one of these three options affects the rest of the problem, you should be able to come up with the recursive insight you need to solve the problem.

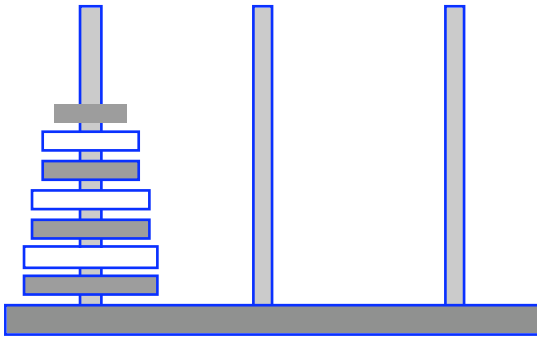
Problem 3: Towers Of Hanoi Revisited

One of the first procedural recursion problems we discussed in lecture was the classic Towers Of Hanoi problem. The recursive solution lists a series of moves needed to move a tower of disks from one location to a second. The solution I provided in class was more or less this:

```
void 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);
    }
}

void MoveDisk(string start, string dest)
{
    cout << "Move top disk from " << start << " to " << dest << "." << endl;
}
```

Consider the similar problem of moving **another** type of tower from the leftmost needle to the rightmost needle.



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);
    }
}

```

Correct**Incorrect****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);
    }
}

```

Correct**Incorrect****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);
    }
}

```

Correct**Incorrect****Version 4:**

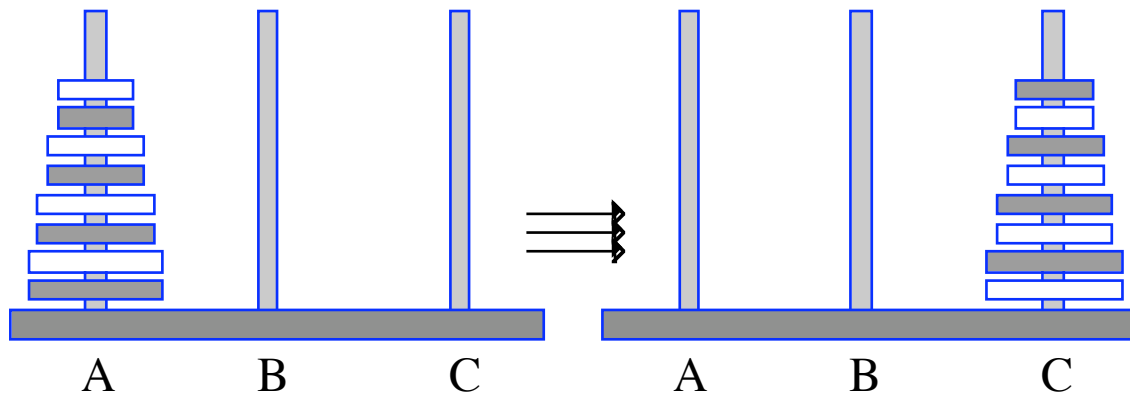
```

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**Incorrect**

Consider the similar problem of moving yet another type of tower from the leftmost needle to the rightmost needle. Note the color scheme of the tower on the right is different; this time it's inverted.



Again, note that each size is represented twice: once as white and once as dark grey. Using the same **MoveDisk** utility we used above, **write** a recursive procedure which provides the list of moves needed to move a tower of height **numDisks** from one peg to another. You must ensure that:

- only one disk is moved at a time
- no disk is ever placed on top of a smaller disk
- the alternating color scheme of the initial tower is **inverted** in the final tower—that means white on the bottom, and then grey, white, grey, white, grey, etc.

```
void InvertTower(int numDisks, string start, string dest, string temp);
```

Problem 4: Twiddles

Two English words are considered to be *twiddles* whenever the letters at each position are either the same, neighboring letters (meaning next to each other in the alphabet), 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 just before **l**. A more dramatic 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**. And just to be clear, **a** and **z** are **not** next to each other in the alphabet—there's no wrapping around at all.

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: any word is considered to be a twiddle of itself, so it's okay to print it.)

```
void ListTwiddles(string str, Lexicon& lex);
```

Problem 5: 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>& anagram);
```

FindAnagram should return **false** if an anagram couldn't be found. Otherwise, it returns **true** and populates the supplied **Vector<string>** with the sequence of words making up the anagram. You should impose a minimum word length of 3, and you should return **true** as soon as you find your first anagram, and **false** as soon as you can tell you'll never get an anagram out of the supplied string.

Once we have **FindAnagram**, it's possible to find anagrams for virtually anything:

```
Anagram Generator: Welcome
-----
Please enter a string of letters: jerrycainjunior
>> ninja juicy error
Please enter a string of letters: peterpawlowski
```

```
>> walker wipe opts
Please enter a string of letters: dorissullivan
>> avion dull sris
Please enter a string of letters: colleenmichellecrandall
>> mecca rill lend leech llano
Please enter a string of letters: oprahwinfrey
>> whir fray nope
Please enter a string of letters: everyonesalittlebitcrazy
>> byte cant eyra riel zits levo
Please enter a string of letters: bbbbbbb
Nothing. :(
Please enter a string of letters:
```

Just write the **FindAnagram** function. Don't worry about writing the entire program.