Section Handout #3

Problem 1: Set Callbacks

1. When using a Set to store user-defined types, it is the client's responsibility to write the appropriate comparison callback and pass that function to the Set constructor. The comparison callback also allows the client to customize how entries are compared (and thus control the order they are returned by the iterator) as well as determining which entries will be coalesced as duplicates.

a) You're building an address book and plan on using a Set to store the entries. Each entry is represented by this structure:

```
struct entryT {
  string firstName;
  string lastName;
  string phoneNumber;
};
```

Write a comparison callback that compares entries using last name as the primary key and first name as the secondary key. Show the necessary declaration for creating a Set of `entryT` using this callback.

Given the above setup, what happens if you have two friends with the same first and last name?

b) You plan on using a Set to store a collection of strings. You wish to treat "Word" and "word" as the same element The default comparison function for Set works on strings, but it compares case-sensitively. Write a comparison callback that instead compares strings without regard to case and use it to declare a Set of string which operates case-insensitively.

Problem 2: Maps

You are writing a program to manipulate a geographic map of major world cities. You have a list of city names and their coordinates. When a user clicks on a point, you want to report what city is there. To efficiently support this operation, you plan to use a Map to associate a pointT with the name of the city at that point. However, Maps require that the keys be of string type. Brainstorm a way to resolve this apparent incompatibility and complete the code started below to load the Map with city data.

```
struct pointT {
    int x, y;
};

struct cityT {
    string name;
```

```
    pointT coordinates;
};

Vector<cityT> cities = ...


// complete from here to show loading Map with city data
```

Problem 3: Cartesian Products

The French mathematician Rene Descartes invented the notion of a "Cartesian product" of sets, defined as follows:

> Let **A** and **B** be sets. The *Cartesian product* of **A** and **B**, is the set of all pairs **(a, b)** where **a** is a member of set **A** and **b** is a member of set **B**. The pair **(a, b)** is called an ordered pair, and will be defined in our solution by the following struct:
>
> ```
> struct pairT
> {
>   string first, second;
> };
> ```

Write a function to compute the cartesian product of two sets of strings sets:

```
Set<pairT> CartesianProduct(Set<string> & one, Set<string> & two);
```

For example, the Cartesian Product of `{"A", "B", "C"}` and `{"X", "Y"}` is:

```
{("A", "X"), ("A", "Y"), ("B", "X"), ("B", "Y"), ("C", "X"), ("C",
"Y")}
```

You will also need to write an appropriate `PairCmpFn` that compares two `pairTs`.

Problem 4: Cannonballs

Suppose that you have somehow been transported back to 1777 and the Revolutionary War. You have been assigned a dangerous reconnaissance mission: evaluate the amount of ammunition available to the British for use with their large cannon which has been shelling the Revolutionary forces. Fortunately for you, the British—being neat and orderly—have stacked the cannonballs into a single pyramid-shaped stack with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Unfortunately, however, the Redcoats are also vigilant, and you only have time to count the number of layers in the pyramid before you are able to escape back to your own troops. To make matters worse, computers will not be invented for at least 150 years, but you should not let that detail get in your way. Your mission is to write a recursive function `Cannonball` that takes as its argument the height of the pyramid and returns the number of cannonballs therein.

```
int Cannonball(int height);
```

Problem 5: ReverseString

Given a string, create a function `ReverseString` that returns the string in reverse order. Consider both recursive and iterative techniques for solving this problem. Which one is easier to come up with?

```
string ReverseString(string str);
```

Problem 6: GCD

The greatest common divisor (g.c.d.) of two nonnegative integers is the largest integer that divides evenly into both. In the third century B.C., the Greek mathematician Euclid discovered that the greatest common divisor of x and y can always be computed as follows:

> If x is evenly divisible by y, then y is the greatest common divisor. Otherwise, the greatest common divisor of x and y is always equal to the greatest common divisor of y and the remainder of x divided by y.

Use Euclid's insight to write a recursive function `int GCD(int x, int y)` that computes the greatest common divisor of x and y.

Problem 7: Old-Fashioned Measuring

> *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 int vector **`weights`**. For instance, the sample set of two weights illustrated above could be represented using the following code:

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

Given this vector, the function call

```
IsMeasurable(2, sampleWeights)
```

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, sampleWeights)
```

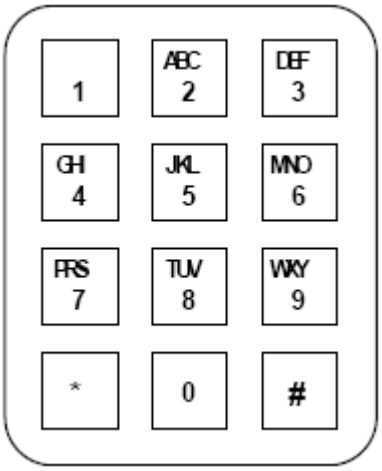
should return **`false`** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

The fundamental observation you need to make for this problem is that each weight in the vector can be either:
1. Put on the opposite side of the balance from the sample
2. Put on the same side of the balance as the sample
3. 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 8: List Mnemonics

On the standard Touch-Tone™ telephone dial, the digits are mapped onto the alphabet (minus the letters *Q* and *Z*) as shown in the diagram below:



In order to make their phone numbers more memorable, service providers like to find numbers that spell out some word (called a **mnemonic**) appropriate to their business that makes that phone number easier to remember. For example, the phone number for a recorded time-of-day message in some localities is 637-8687 (NERVOUS).

Imagine that you have just been hired by a local telephone company to write a function `ListMnemonics` that will generate all possible letter combinations that correspond to a given number, represented as a string of digits. For example, if you call `ListMnemonics("723")` your program should generate the following 27 possible letter combinations that correspond to that prefix:

```
PAD PBD PCD RAD RBD RCD SAD SBD SCD
PAE PBE PCE RAE RBE RCE SAE SBE SCE
PAF PBF PCF RAF RBF RCF SAF SBF SCF
```

The function declaration is listed below:

```
void ListMnemonics(string str);
```