Section Solutions #3

Problem 1: Set Callbacks

a)
```
int CompareEntry(entryT one, entryT two)
{
    if(one.lastName < two.lastName)
      return -1;
    else if(one.lastName > two.lastName)
      return 1;
    else if(one.firstName < two.firstName)
      return -1;
    else if(one.firstName > two.firstName)
      return 1;
    else
      return 0;
}
```

You would declare the set as:

```
Set<entryT> addresses(CompareEntry);
```

If you had multiple friends with the same first and last name, the Set would consider them duplicates and eliminate one of them. If you did not want this to happen, you would need to rewrite the comparison function to take other factors (such as phone number) into account.

b)
```
int CompareStringCaseInsensitive(string one, string two)
{
    string lowerOne = ConvertToLowerCase(one);
    string lowerTwo = ConvertToLowerCase(two);
    if(lowerOne < lowerTwo)
      return -1;
    else if(lowerOne > lowerTwo)
      return 1;
    else
      return 0;
}
```

You would declare the set as:

```
Set<string> caseInsensitiveSet(CompareStringCaseInsensitive);
```

Problem 2: Maps

The easiest way to solve this problem is to convert the `pointT` to an appropriate string representation. For example, the point (4, 2) could be represented as the string "4-2" and this string used as the key into the map to add or get the city for this point. Note,

however, that you do need some kind of separator between the x and y coordinates; otherwise you won't be able to differentiate (2, 12) from (12, 2).

```
Map<string> nameMap;
for(int cityIndex = 0; cityIndex < cities.size();cityIndex++)
{
    cityT currCity = cities[cityIndex];
    string key = IntegerToString(currCity.coordinates.x) + "-" +
                 IntegerToString(currCity.coordinates.y);
    nameMap.add(key, currCity.name);
}
```

Problem 3: Cartesian Products

```
int PairCmpFn(pairT one, pairT two)
{
  if (one.first == two.first && one.second == two.second)
  {
    return 0;
  }
  else if (one.first < two.first)
  {
    return -1;
  }
  else if (one.first == two.first && one.second < two.second)
  {
    return -1;
  }
  else
  {
    return 1;
  }
}

Set<pairT> CartesianProduct(Set<string> & one, Set<string> & two)
{
  Set<pairT> result(PairCmpFn);
  Set<string>::Iterator oneIt = one.iterator();

  while(oneIt.hasNext())
  {
    string first = oneIt.next();
    Set<string>::Iterator twoIt = two.iterator();

    while (twoIt.hasNext())
    {
      pairT pair;
      string second = twoIt.next();

      pair.first = first;
      pair.second = second;

      result.add (pair);
    }
```

```
  }

  return result;
}
```

Problem 4: Cannonballs

```
/*
 * Function: Cannonball
 * Usage: n = Cannonball(height);
 * ------------------------------
 * This function computes the number of cannonballs in a stack
 * that has been arranged to form a pyramid 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. The function Cannonball computes
 * the total number based on the height of the stack.
 */
int Cannonball(int height)
{
  if (height == 0)
  {
    return (0);
  }
  else
  {
    return (height * height + Cannonball(height - 1));
  }
}
```

Problem 5: ReverseString

```
string ReverseStringRecursive (string str)
{
  if (str.length() == 0)
  {
    return "";
  }

  return ReverseStringRecursive(str.substr(1)) + str[0];
}

string ReverseStringIterative (string str)
{
  string result = "";
  for (int i = str.length() - 1; i >= 0; i--)
  {
    result += str[i];
  }

  return result;
}
```

Which one is easier to come up with? The recursive one, of course (well, maybe)! Which one is more efficient? The iterative version does not create a lot of stack frames to store

its data and doesn't have to make very many function calls. In most cases, it will be the faster of the two functions.

Problem 6: GCD

```
int GCD(int x, int y)
{
  if ((x % y) == 0)
  {
    return y;
  }
  else
  {
    return GCD (y, x % y);
  }
}
```

Problem 7: Old-Fashioned Measuring

```
bool RecIsMeasurable(int target, Vector<int> & weights, int index)
{
  if (target == 0)
  {
    return true;
  }

  if (index >= weights.size())
  {
    return false;
  }

  return (RecIsMeasurable(target + weights[index], weights, index + 1)
       || RecIsMeasurable(target, weights, index + 1)
       || RecIsMeasurable(target - weights[index], weights,
                          index + 1));
}

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

Problem 8: List Mnemonics

```
/*
 * Function: ListMnemonics
 * Usage: ListMnemonics(str);
 * --------------------------
 * This function lists all of the mnemonics for the string of digits
 * stored in the string str. The correspondence between digits and
 * letters is the same as that on the standard telephone dial. The
 * implementation at this level is a simple wrapper function that
 * provides the arguments necessary for the recursive call.
 */
void ListMnemonics(string str)
```

```
{
    RecursiveMnemonics("", str);
}



/*
 * Function: RecursiveMnemonics
 * Usage: RecursiveMnemonics(prefix, rest);
 * ----------------------------------------
 * This function does all of the real work for ListMnemonics and
 * implements a more general problem with a recursive solution
 * that is easier to see. The call to RecursiveMnemonics generates
 * all mnemonics for the digits in the string rest prefixed by the
 * mnemonic string in prefix. As the recursion proceeds, the rest
 * string gets shorter and the prefix string gets longer.
 */
void RecursiveMnemonics(string prefix, string rest)
{
  if (rest.length() == 0)
  {
    cout << prefix << endl;
  }
  else
  {
    string options = DigitLetters(rest[0]);
    for (int i = 0; i < options.length(); i++)
    {
      RecursiveMnemonics(prefix + options[i], rest.substr(1));
    }
  }
}

/*
 * Function: DigitLetters
 * Usage: digits = DigitLetters(ch);
 * ---------------------------------
 * This function returns a string consisting of the legal
 * substitutions for a given digit character. Note that 0 and
 * 1 are handled just by leaving that digit in its position.
 */
string DigitLetters(char ch)
{
  switch (ch) {
    case '0': return ("0");
    case '1': return ("1");
    case '2': return ("ABC");
    case '3': return ("DEF");
    case '4': return ("GHI");
    case '5': return ("JKL");
    case '6': return ("MNO");
    case '7': return ("PRS");
    case '8': return ("TUV");
    case '9': return ("WXY");
    default: Error("Illegal digit");
  }
}
```