Homework #9 - Recursion

Name:

1 Recursion

You've probably seen code that looks like this before:

```
for (int i = 10; i > 0; i--) {
  cout << i << endl;
}
```

It's just a regular "for" loop that counts down from 10 to 1. Is there any other way to count from 10 to 1? Without a loop? And without just writing "cout" ten times? Turns out there is:

```
void countDown(int start) {
    if (start > 1) {
        cout << start << endl;
        countDown(start - 1);
    }
    else {
        cout << 1 << endl;
    }
}

int main() {

    countDown(10);

    return 0;
}
```

simpleRecursion.cpp

If you take a close look at the function at the top, you'll notice that **it calls itself**! Functions are completely allowed to do this. You can write a C++ function that uses itself. This can be a difficult concept to understand at first, but it's very powerful and useful once you understand it.

1.1 The Termination Condition

Usually, recursive functions have an if/else inside that checks the value of an **argument** passed to the function. If the argument meets a certain condition, the recursion can continue (the function calls itself again), but with the argument changed (in this case, reduced by 1). If the argument does **not** meet the condition, the function does a "return". When this happens, we say the recursion has **terminated**. This means the function stops calling itself, and the computer goes "back up the call stack". That means that all the functions that were called finish their work and go back into our program's main() function.

1.2 Factorial

Though it's hard to believe when you first see it, some algorithms are easiest to write if you write them recursively. A great example is **factorial**. You're probably familiar with factorial from math class. You write "factorial five" like this:

$$5!$$

And it is solved like this:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Imagine a recursive function to solve this. You could have a function that calls itself on 5, then 4, then 3....down to 1 then stops and **terminates the recursion**. Something like this:

```
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n-1);
    }
    else {
        return 1;
    }
}
```

factorial.cpp

When the value "n" gets to 1, the function stops calling itself and just returns "1". All the functions above it would have done "n * factorial(n-1)", so the computer ended up doing something like this:

```
factorial(5) * factorial(4) * factorial(3) * factorial(2) * factorial(1)
```

Doing factorial(1) just returns a 1 into the function factorial(2), which does the math "2 * 1" and returns a "2" to factorial(3), which does "3 * 2" which returns the value "6" to factorial(4) which does "4 * 6" and returns "24" to factorial(5) which does "5 * 24" and then returns the final answer "120". So the recursion **collapses**, kind of like a folding telescope. To illustrate this for you, imagine a program that did "cout << num;" and then called itself on "num + 1", stopping when "num == 4". If we started it at "0" we'd get something like this when it ran:

```
1  recursiveFunction(0)
2  printf(0)
3          recursiveFunction(0+1)
4          printf(1)
5                  recursiveFunction(1+1)
6                  printf(2)
7                          recursiveFunction(2+1)
8                          printf(3)
9                                  recursiveFunction(3+1)
10                                 printf(4)
```

Figure 1: A Simple Recursion

2 Now You Try

There is a famous **series** (a set of numbers counting up or down) in mathematics called the **Fibonacci series**. It goes like this:

```
1 1 2 3 5 8 13 21 34 55 89 144...
```

Can you see a pattern here? $0+1=1$, the next number in the series. Then $1+1=2$, the next number. $1+2=3$, $2+3=5$, $3+5=8$, and so on. Try to write a **recursive** C++ program that print out this series. You should have a function in your code that looks like this:

```
void fibonacci(int a, int b, int n){
  // YOUR CODE HERE
}
```

If you wanted the first 12 numbers, you'd set "a" and "b" to be "0" and "1" and you'd set "n" to "12". Like this:

```
int main() {

    cout << "The first 12 Fibonacci numbers: " << endl;
    fibonacci(0, 1, 12);

    return 0;
}
```

Which would print out:

```
1 1 2 3 5 8 13 21 34 55 89 144
```

Think about how the 3 arguments to fibonacci should **change** each time the function is called. Think about what number the fibonacci function should print out each time it is called. Think about what should happen when "n == 0". Have a look at the files **factorial.cpp** and **simpleRecursion.cpp** for inspiration.