CSci 1113, Spring 2018
Lab Exercise 8 (Week 9): Recursion!

Recursion

Recursion is an abstraction that is defined in terms of *itself*. Examples include mathematical abstractions such as Fibonacci series, Factorials, and Exponentials, and other abstractions such as the Towers of Hanoi puzzle, the Eight Queens problem and Fractal Geometry. *Recursion* is actually an alternate form of *repetition*. So far, we've employed repetition using loops (*while*, *do-while*, and *for*), but many interesting and challenging computational problems are difficult to solve in an iterative fashion. Often these problems have very simple and elegant recursive solutions.

After working though the following exercises using the methods discussed in class, you should begin to get a feel for "thinking recursively" as a problem solving technique.

Warm-up

You are encouraged to use pencil/pen and paper on the warm-up problems.

1) Consider the following statement: "construct a recursive function to display the digits of any positive integer in reverse order on the console display". This is a void function that would take a single argument:

```
void reverseNum(int n);
```

and, given an integer value such as:

```
reverseNum(5786);
```

would output the following to the console display:

```
6875
```

    a. Start by identifying base case(s) that is trivial to solve. For this problem, perhaps the easiest integer to "reverse" would be any value with a single digit. In this case, you would simply display the input value and be done. Write the C++ statement or statements to implement the base case and return:

    b. Next, determine if there is some way the original problem can be represented as the base case *combined* with a recursive call to the function using a reduced version of the original input. For this particular problem, note that if the argument contains more than one digit, reversing it entails displaying the rightmost digit first, followed by the rest of the digits in reverse order. The "*rest of the digits in reverse order*" is simply a reduced version of the original problem! First, write the C++ statement or statements that will display the "rightmost" digit:

    c. Now, write the *recursive* reduction step that calls the function for the "*rest of the digits*":

d. Combining these two steps forms the complete reduction case. If this is the only reduction step, it is simply performed for any but the base case(s). Structure the steps above as an `else` clause (you'll add it to an `if` clause in the next step) that contains the entire reduction step:

e. Before you complete the function, take a minute to ensure that (i) the base case represents a correct solution, and (ii) the reduction step is guaranteed to get you "closer" to the base case. Now combine the base and reduction steps and write the complete function definition:

2) Consider a *recursive* function to find the maximum value in an array of integers.   The function declaration is:

```
int maxValue( int vals[], int size, int start );
```

For this function, we need to know the size of the array *and* the starting index of the array (because both will change when a recursive call is made). You may assume that there is at least one value in the array:

a. Describe the base case: [Hint: in what size array would it be easiest to find the maximum value?]

b. Describe the reduction step: [Hint: describe the original problem using the base case in combination with some reduced form of the problem.]

c. Describe why the reduction step will (i) solve the problem and (ii) move closer to the base case:

d. Write the complete function definition for `maxValue`:

Stretch

1) Factorial

A factorial of a n = n * (n-1) * (n-2) * ... * 3 * 2 * 1. For example, 5! (five factorial) is 5*4*3*2*1 = 120.

Individually write (using pencil/pen and paper) a *recursive* function `factorial` that will take an integer value *n* as an argument, and will return *n!* . Then compare your function to your partner's, together (i) come up with a function implementation you both agree on, and (ii) write it as a C++ function on the computer. Then write a short C++ program that will call your function and display the factorial of 10.

Workout

1) **Binomial Coefficients**

In probability and statistics applications, you often need to know the total possible number of certain outcome combinations. For example, you may want to know how many ways a 2-card BlackJack hand can be dealt from a 52 card deck, or you may need to know the number of possible committees of 3 people that can be formed from a 12-person department, etc. The *binomial coefficient* (often referred to as "*n* choose *k*") will provide the number of combinations of *k* things that can be formed from a set of *n* things. The binomial coefficient is written mathematically as:

$$\binom{n}{k}$$

which we refer to as "*n* choose *k*".
Binomial coefficients can be defined recursively:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \binom{n}{0} = \binom{n}{n} = 1$$

Individually, write a *recursive* function named `choose(int n, int k)` that will compute and return the value of the binomial coefficient. Then compare your function to your partner's, and together (i) come up with a function implementation you both agree on, and (ii) write it as a C++ function on the computer.

2) **Greatest Common Divisor**

In a previous lab you implemented an iterative version of the Euclidean algorithm for finding the Greatest Common Divisor between two integer values. This problem has an elegant and simple recursive solution.

Recall that the `greatest common divisor` of two integers *a* and *b*, GCD(*a*,*b*), not both of which are zero, is the largest positive integer that divides both *a* and *b*. The Euclidean algorithm for finding this greatest common divisor of *a* and *b* is as follows:

   * Divide *a* by *b* to obtain the integer quotient *q* and remainder *r* so that $a = bq + r$. Note: if *b* = 0, then GCD(*a*,*b*) = *a*.

   * Now, GCD(*a*,*b*) = GCD(*b*,*r*) so replace *a* with *b* and *b* with *r*, and repeat this procedure.

Since the remainders are decreasing, eventually a remainder of 0 will result. The last nonzero remainder is GCD(*a*,*b*). For example,

1260 = 198 x 6 + 72	GCD(1260,198) = GCD(198,72)
198 = 72 x 2 + 54	= GCD(72,54)
72 = 54 x 1 + 18	= GCD(54,18)
54 = 18 x 3 + 0	= 18

Write a *recursive* implementation of the Euclidean GCD function:

```
int gcd(int a, int b);
```
[Hint: Note that when the remainder is zero, the divisor will be the GCD]

Write a short program to test your GCD function with the following values:
     1280,198
     197,13
     120,60

3

Challenge

Here is an extra challenge problem. You should be able to complete the warm-up, stretch and workout problems in the lab. Try this if you have extra time or would like additional practice outside of lab.

- Multiple Implementations

Sometimes a recursive implementation provides an efficient solution to a task, and sometimes it is inefficient. This challenge problem asks you to implement the Fibonacci sequence in different ways than the recursive implementation from class. Specifically, write two additional *non-recursive* versions for finding the first 20 Fibonacci numbers. (Your programs may do all the computations in main() rather than having the computation of each Fibonacci number done in a function.)

- A version that only holds in memory the current and two previous Fibonacci numbers as it computes the numbers one by one.
- A version that computes the numbers and stores them in an array as it computes them.

Once you have implemented these techniques think about their advantages and disadvantages when compared to each other and to the recursive implementation. More generally, when is a recursive implementation preferable, and when are other possible implementations preferable?