CSCI205 Computer Science III
Lab Assignment: Recursion

Simple recursion exercises.

These exercises will help you get the hang of recursion without worrying about overly complex problems. During your progress through these exercises you should begin to focus on the “spacial complexity” of recursive algorithms. The notion of “space” in this context refers to “memory space” or the amount of memory that an algorithm consumes in proportion to the “problem size”. You will be asked to analyze both the abstract “time complexity” and the spacial complexity. This will result in two uses of O notation; one for the space and one for the time.

Recursion Review: Euclid's algorithm for greatest common denominator

Recursion is a problem solving approach that allows programmers to break a problem down into smaller and smaller instances of the same problem. This process occurs repeatedly until a trivial, known solution is reached. When the known solution is reached, the process reverses and the ultimate solution is “assembled” by combining and solving each subsequent “larger solution”.

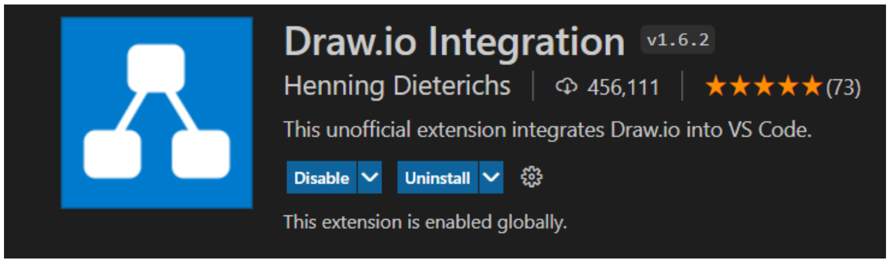
Most programming languages support the concept of recursion by allowing functions and methods to “call themselves”. Following the problem solving approach of breaking a problem down into smaller and smaller instances, each recursive call should shrink the problem in some meaningful way.

When studying the following instance of Euclid’s GCD algorithm, notice how the arguments to the recursive call are modified and re-ordered, such that each recursive step of the problem defines a smaller sub-problem of factors of the original values. Run this algorithm in a debugger using 12 and 8 as arguments.

Make a sketch of the call stack in a diagramming program showing the stack frames for each method call.

I like to use this free program: https://app.diagrams.net/

there is a VS Code extension that allows you to use this right in the VSC interface



This unofficial extension integrates Draw.io into VS Code.

This extension is enabled globally.

Don't forget about the return address (the first item pushed on the stack when a method is called: **32 or 64 bit integer** depending on your system).

- What is the spacial complexity?
- What is the temporal complexity?

Calculate the depth of the stack and trace the returns. Run this using other test cases.

```
Recursive: euclid_gcd(a, b):
        if b == 0 return a
        else return gcd(b, a mod b)
```

This type of recursion is called **tail recursion** as the recursive call is the last statement in the routine and the return value is not saved or modified as the stack is unwinding. Notice the difference between this example and the factorial and Fibonacci examples. In those two approaches the math happened "on the way up" from the base case. In Euclid's algorithm the math happens "on the way down" as part of the argument modification.

Compare this to the iterative implementation. Count the number of division statements. Does this correlate to the stack depth of the recursive implementation?

```
Iterative: euclid_gcd(a, b):
        while b != 0
            t = b
            b = a mod b
            a = t
        return a
```

Solve the following simple recursion exercises. Include all solutions in a single C++ source file. Give me a **main.cpp** file that clearly shows the results. Print messages to show which problem is currently running. Also print the Big O notation for both temporal and spacial complexity for each algorithm. You will need to modify the arguments to these methods in order to achieve peak recursion. It is often beneficial to write helper methods for these as the arguments for recursive functions can be complex and un-intuitive.

Here's an example: Given an array of ints, recursively compute the number of times that the value 11 appears in the array. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass **index+1** to move down the array. The initial call will pass in index as 0.

```
int array[9] = {1, 2, 11, 3, 4, 11, 4, 5, 11};

elevens = count11(array);

int count11(int array[]){
    // this public method allows to call this without worrying
    // about specifying a starting index
    return count11(array, 0);
}
```

```
int count11(int array[], int index){

        // this is the recursive algorithm. Don't make people call this because                //
        // passing the index as an argument is a tad obscure

        if (index >= array length) return 0;

        if (array[index] == 11)

                return 1 + count11(array,index+1);

        else

                return count11(array,index+1);
```

Define each of these exercises in a single *main.cpp.* Include ample comments describing the logic.

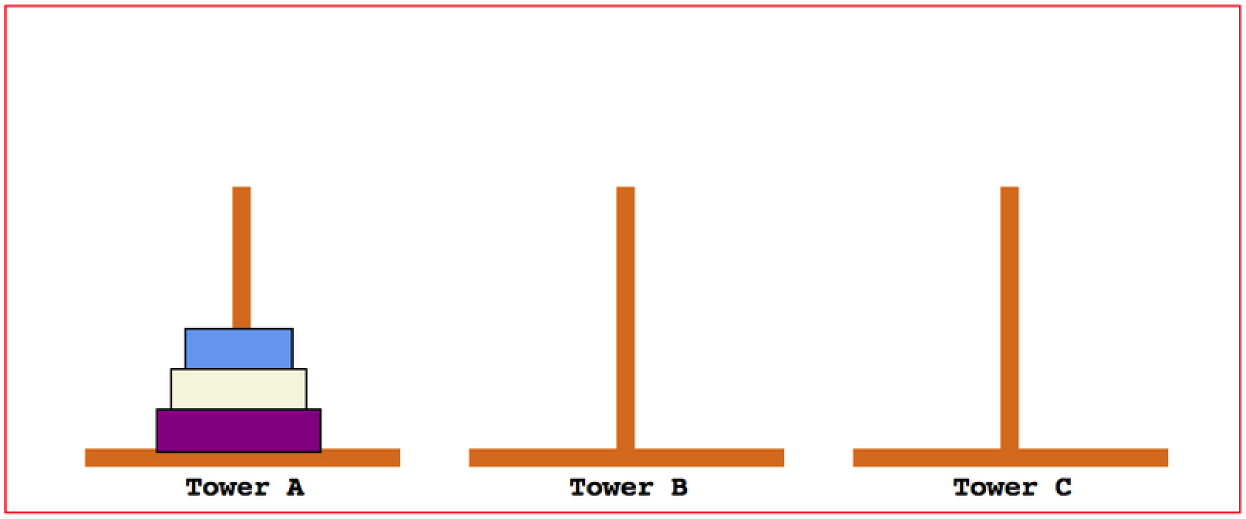
Some of these exercises will require you to use the Stack template you built for a previous lab. You are required to use your own structures for these problems. You may not use class from STL (standard template library).

Include comments on spacial and time complexity for each exercise. BE DETAILED and include a description of how you came to your conclusions.

1. Simulate recursion by writing an ***iterative*** method that reverses a string using an explicit stack of your design.
2. Write a ***recursive*** method that reverses a string using the runtime stack
3. Write a ***recursive*** method that prints the numbers 1 – 50
4. Write a ***recursive*** method ***sum(int[] array)*** to calculate the sum of numbers contained in an array. Decide if you want the math to occur with the argument passing or during the return
5. Write a recursive method ***len(int n)*** to count the digits of a given integer. Do not treat it as a String.
6. We have several puppies standing in a line, numbered 1, 2, ... N. The odd puppies (1, 3, . . .) have 2 ears. The even puppies (2, 4, . . .) have 3 ears, because they are from Chernobyl. Recursively return the number of "ears" in the puppy line 1, 2, ... N (without loops or multiplication).
7. Write the recursive method ***strip(string text, char letter)*** that accepts a String and a character and recursively "cleanses" the string of that character.

   Example: strip("123-45-6789", '-') => "123456789"
8. Given a string, return true if it is a nesting of zero or more pairs of parentheses, like "(())" or "((()))". **Suggestion**: check the first and last chars, and then recur on what's inside them.
9. Modify the ***print()*** method from ***your Linked List class*** to use recursion instead of iteration.
10. Modify the ***reverse()*** method from your ***Linked List class*** to use recursion instead of iteration.

For this exercise, you will study and implement the recursive algorithm for the famous Towers of Hanoi puzzle.



1. **Understanding the Towers of Hanoi**

    a. Read a general description of the Towers of Hanoi here: **https://en.wikipedia.org/wiki/Tower_of_Hanoi**

    b. Play with this puzzle here: **https://www.mathsisfun.com/games/towerofhanoi.html** until you can easily solve this puzzle using 4 disks

    c. The recursive solution has been provided for you

    d. Run the solution and study the output . . . it may add some clarity if you add some prints to communicate when the recursive methods are returning.

    e. Run the solution in a debugger, paying close attention to the winding and unwinding of the call stack.

    f. Using a diagramming tool like **Draw.io (preferable) draw** the recursive call tree for a ***3-disk solution***. You will probably need to use landscape mode. Try to fit this diagram on a single page. Focus on understanding how a dual recursive method creates the various call branches and how the returns are managed.

2. **For this step, you will implement the recursive algorithm for the Towers of Hanoi using Stacks for the pegs.**

    a. The Stacks must be structures that you have defined and must include your **LinkedList<T>** type as the underlying memory management **do not use C++ STL structures.** Label these stacks **A, B and C**

    b. Create a Disk class that will represent the disks on the pegs. This class can be s simple as a single integer that represents the disk's number on the original stack. Include some way to track the size of the disk

c. Write the recursive solution as a recursive function in **towers_main.cpp.** Include a parameter-less helper called **towers_of_hanoi()**

d. Initially run the algorithm with a Disk count of 6 on peg A

e. Include sufficient print statements to communicate what the algorithm is doing.

f. Display the contents of the pegs prior to running and after running

g. Include sufficient counts to prove the **$O(2^N - 1)$** efficiency. Display the counts along with a meaningful message

h. Run the algorithm with various disk counts . . . does the efficiency hold on each collection?

Submission: Push all mentioned files. List<T>, Stack<T>, main, diagrams . . . etc. Please include everything.

Do not include compiled binaries or any hidden directories created by VS Code or your OS or anything that does not apply to your code being reviewed.

The diagrams should be exported as image files. Ensure that your build process completes with no warnings and your code runs leak and error free. It will be run against valgrind.