CS 2B
Lab Exercise

Name:_____

Due: In Two Weeks.

TOPIC: Stacks and Recursion Elimination

You have seen in your lectures, readings, and computer explorations, that the power of recursive problem solving (programming) lies in its ability to express a solution to a problem in terms of a simpler version of the same problem. For example, the computation of n! (n factorial) can be expressed recursively as:

```
n! = 1  if   n = 0   otherwise  n! = (n)(n-1)!
```

Notice, to avoid infinite regresses, the solution must end in a trivial or base case. In the above, n = 0 stops the recursion and gives the trivial result 0! = 1.

You should also be aware that although recursion may provide a nice elegant mathematical solution to a particular problem, it might not be the most efficient approach to solving the problem on a computer. (Or by hand for that matter!) To demonstrate this, consider computing 10! using the above recursive definition or using the following iterative definition:

```
n! = (n)(n-1)(n-2)(n-3)...(2)(1)
```

Which technique do you think provides you with the least amount of grief? How about on a computer? What must take place during each recursive call? Which method will use more computer resources (time and space)? Which will be harder for you to code and implement in a target language which supports both recursion and iteration? These questions should always be examined when constructing the solution to a problem. The trade off between mathematical elegance/ease of implementation and computer space/time requirements should be in the forefront. To drive this point home, consider the computation of the elements of the Fibonacci sequence 1, 1, 2, 3, 5, 8, ... A mathematically simple recursive algorithm for computing the first n Fibonacci numbers can be defined as follows:

$f_n = 1$ if $n = 1$ or 2 otherwise $f_n = f_{n-1} + f_{n-2}$

Also, an iterative routine using a array '`f`' to store each element in the sequence (`f[0]` is not used) could be defined as follows:

```
f[1]= 1,  f[2] = 1
for k = 3 to n do f[k] = f[k-1] + f[k-2]
```

It can be shown that the recursive version implemented as defined is of exponential time complexity (wow!), whereas the iterative one is linear!!!! - Quite a significant difference for large n.

So it will behoove us to keep our eyes open for non recursive solutions to problems whose solutions have been presented recursively. One, because it might provide a much more computer efficient solution, and two, you might be working in an environment which does not even support recursion.

This lab will examine three topics, two of which should enlighten you as to what happens during a recursive process:

1. Removal of Tail Recursion using Iteration

2. Examination of a Stack class

3. Removal of Recursion via Runtime Stack Simulation.

Removal of Tail Recursion

An algorithm is said to be tail recursive if the recursive step is the last step in the algorithm. The implementation of the recursive definition of factorial above is an example of tail recursion. i.e.

```
public static long fac(int n)
{
   long returnValue;
   if (n == 0)
      returnValue = 1;
   else
      returnValue = n * fac(n-1);
   return returnValue;
}
```

Notice the recursive call `fac(n-1)` is the last step in the algorithm.

Recursion can be eliminated from a tail recursive algorithm by noting what happens during the recursive call. In the above example, the value parameter n is decreased by one and control loops back to the beginning of the algorithm. It is possible to accomplish the same effect without recursion by using a simple looping construct like a while do loop and a local variable instead of a subprogram parameter. For example, the above recursive factorial function could be expressed iteratively by removing the tail recursion as follows:

```
public static long fac(int n)
{
   long returnValue = 1;
   while (n > 0)
   {
      returnValue = fac * n;
      n--;
   }
   return returnValue;
}
```

Another example which you might research is the relationship between recursive and non recursive binary search algorithms.

So you might get some practice doing this on your own, you are to accomplish the following:

A. Consider the following method containing a tail recursive call:

```
public static int tailRecursion(int m, int n)
{
  if (m == 0 || n == 0)
    return 1;
  else
    return 2 * tailRecursion(m-1, n+1);
}
```

Implement a recursive C++ or Java method to compute `tailRecursion(m,n)` and test your method on a wide variety of different input values just to see what gets returned. Note: you will be evaluated on your test data set, so choose it carefully to ensure that it adequately 'tests' your method.

B. Implement a non recursive method for the above method, `tailRecursion`, which computes the return values iteratively. Again carefully design a set of input values for verification and 'test' your method.

Class Stack

During this course you will be slowly introduced to the concept of Object Oriented Programming. The three main properties which characterize an OOP language are:

Encapsulation - Combining data with the methods which manipulate it ---> an Object

Inheritance - Building an object hierarchy with each descendent inheriting access to all its ancestors' code and data

Polymorphism - name sharing of actions

It can be shown that these lead to more structure, modularity, abstraction, and reusability. These create code which is extensible and easy to maintain.

Our first encounter will be with encapsulation.

The C++ STL (Standard Templet Library) and Java's , `java.util` package provide implementations of the Stack ADT (abstract data type). A stack, as you know, encapsulates a LIFO (last in first out) structure.

C. Create a program which demonstrates all the features of the stack class from either the STL or package java.util. Use an instance of an Integer Stack in your demo. Later in the course we will examine possible implementations for the Stack ADT. Now, just enjoy an already designed one.

Recursion and Recursion Elimination using System Stack Simulation

Recursion is implemented in via the system stack. Each recursive call to a method requires that the following information be pushed onto the system stack:

Formal Parameters
Local Variables
Return Address

This information, collectively, is referred to as a stack frame. The stack frame for each method call will be different. For example, the stack frame for a parameterless/no local variable method will contain just a return address. All this is transparent to us as users because this happens magically due to the compiler. When we make a call to a recursive method, the compiler has taken care of all the above required stack maintenance. To aid in our understanding of the process that takes place during the execution of a typical recursive routine, it is instructional to simulate it by managing this stack frame ourselves (not let C++/Java do it) with our own Stack.

The following strategy can be used to the remove the recursion from a recursive routine, although not elegantly. There might be a far more pleasing method for a particular routine, but this technique is very instructional. It allows you simulate the system stack by declaring your own stack structure and manage the recursion. It is accomplished as follows:

i) Each time a recursive call is made in the algorithm, push the necessary information onto your stack.

ii) When you complete processing at this deeper level, pop the simulated stack frame and continue processing in the higher level at the point dictated by the return address popped from the stack.

iii) Use an iterative control structure that loops until there are no return addresses left to pop from the stack.

The use of a swich statement might help with the return address location.

To demonstrate this, here is a possible system stack simulation for the recursive factorial definition stated above. Study this example and 'run' it for say 4!. Notice that the stack frame in this simulaton is a simplification of what really happens. It contains the formal parameter n and two return addresses. The initial return address (30), pushed onto the stack, is used to act as the final return to the calling routine. Since the function itself will take care of this return, it is being used here to assign the final value to the function (fac). Keep in mind that this is not a true simulation of the actual system stack during the execution of this recursive routine, but does give the feeling of what is going on to keep track of current parameters and where control must be transferred.

```
public static int fac(int n)
{
    int address = 10; // Entry point for each each "call"
    int tempFac = n;

    Stack<Integer> s = new Stack<Integer>();

    s.push(30); // Initial return address
    s.push(tempFac);
    while(!s.isEmpty())
    {
       switch(address)
       {
          case 10:
          {
             n = s.pop();
             if(n < 2)
             { // The base case
                address = s.pop();
                s.push(1);
             }
             else
             {
                s.push(n);
                s.push(20); // Where should I return?
                s.push(n-1);
                address = 10; // Make another "call"
             }
             break;
          }
          case 20:
          {  // Compute and return
             tempFac = s.pop();
             n = s.pop();
             address = s.pop();
             tempFac = n * tempFac;
             s.push(tempFac);
             break;
          }
          case 30:
             // The final return value
             tempFac = s.pop();
       }
    }
    return tempFac;
 }
```

Here's a little problem for you:

The following is a recursive algorithm for computing the coefficients in the (binomial) expansion of

$$(a+b)^n = C_{n,0}\ a^n + C_{n,1}\ a^{n-1}b^1 + \ldots + C_{n,n}\ b^n$$

where

$C_{n,m}$ = a call to:

```
public static int comb(int n, int m)
{
    int returnValue;
    if  ((n == 1) || (m == 0) || (n == m))
      returnValue = 1;
    else
      returnValue = comb(n-1,m) + comb(n-1,m-1);
    return returnValue;
}
```

D. Implement the above method in C++ or Java and generate the binomial coefficients for n = 0 to n = 10. Display these values as elements of Pascal's triangle:

```
          1
        1  1
      1  2  1
    1  3  3  1
  1  4  6  4  1
       .
       .
       .
10 rows deep.
```

E. Design a system stack simulation for the above recursive function and use it to generate Pascal's triangle above. Use an instance of the stack class from part C

<u>Turn in all appropriate hard/soft copy and be prepared to defend A. - E. above.</u>