Lecture

NOTE: FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

Stack

A ***Stack*** is a collection of objects inserted and removed according to the Last In First Out (***LIFO***) principle.

Recursion using Stack:

The most fundamental use of stacks relates to how functions are called. One function can call another function which can call a third and so on to any depth until there is no more stack left to support further calls.

Recursion

The ability to call a function from within another function allows a strategy called recursion to be used. In this scenario a function takes some action to reduce the complexity of a problem and then calls itself until the problem becomes simple enough to solve without further calls.
If a large list of items is to be sorted for example, sorting the entire list is going to involve a lot comparisons and moving items around. Sorting half the list will be easier so we can code a function that passes half the list to another version of itself that in turn halves the list and passes it on to a third version. Eventually a version will get a list containing one or maybe no items. Such lists can be deemed already sorted in both cases.
A simpler problem is to compute the factorial value of a number. Factorial ***n***, represented as ***n!*** is the number of different ways ***n*** items can be arranged in a sequence of the items, just the ticket for working out our chances of winning the next offering of Lotto.
One way of thinking of this is to say we have ***n*** choices for the first item to put into a sequence, ***n-1*** choices for the second and so on until we get a single choice for the last. If we start by creating ***n*** sequences then each of the original sequences clones into ***n-1*** sequences to accommodate the ***n-1*** choices available at the next choosing. Each of these ***n(n-1)*** sequences then clones again to satisfy each of the ***n-2*** choices remaining, and so on.
To get the total of possible sequences, ***n*** needs to be multiplied by the product of all the other choices for each choosing down to the last. In other words ***n!=n(n-1)!***
Here is a recursive factorial method:

```
public int factorial(int n)
{
    if(n <= 1)
        return 1;
    else
```

```
        return n * factorial(n-1);
}
```

Example: Consider we have to find the factorial of 4. This is how the recursion will solve

4!

4 * 3!

4 * 3 * 2!

4 * 3 * 2 * 1!

4 * 3 * 2 * 1 (1 comes from base case)

Now see how processor solves this problem using stack.

Computer have to solve **factorial(4)**

Which is **4 * factorial(3)**

Which is **3 * factorial(2)**

Which is **2 * factorial(1)**

Which is **1**

Now no further calls so,

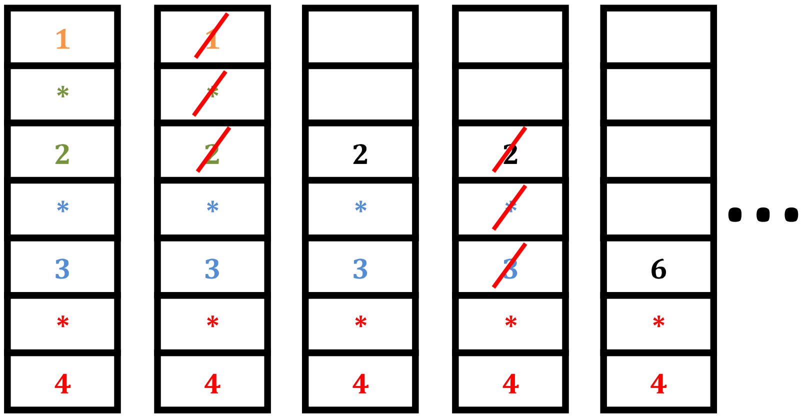
Lets empty the stack out.

2 * 1 = **2** (pushed back onto Stack)

3 * 2 = **6** (pushed back onto Stack)

4 * 6 = **24** (pushed back onto Stack)

24 Stack is empty so return the answer



Using Stack to solve factorial

We need to understand how we can use stack to solve recursion This method has more complexity then other versions of solving factorial like iterative or recursive but it will help us better understand the functionality of stack.

```
public int factorial(int n)
{
    MyStack.nSize = n;
    MyStack st = new MyStack();
    int f = 1;
    while(n>=1)
    {
        st.push(n);
        n--;
    }
    while(!st.isEmpty())
    {
        int x = st.pop();
        f = f * x;
    }
    return f;
}
```

Your Task: What will be the recursive method and stack method for pow(n, x)?