INTRODUCTION

In this activity, you are to practice writing recursive functions.

RECURRENCES

Recall the *factorial* function:

```
int fact(int n)
    {
    if (n == 1) return 1;
    else return n * fact(n - 1);
    }
```

A mathematical way to formalize a recursive function is known as a *recurrence*. Here is the recurrence for *factorial*:

fact (*n*)	is	1	if *n* is equal to 1
fact (*n*)	is	*n* * *fact* (*n* - 1)	otherwise

You can see how closely the recurrence matches the actual C function. From looking at the C function, you can see that the function counts *down*. That is to say, *n* gets smaller and smaller upon each recursive call. It is also possible to define a factorial function that counts *up*. Here is the recurrence for such a function.

fact (*m*,*n*)	is	*m*	if *m* is equal to *n*
fact (*m*,*n*)	is	*m* * *fact* (*m* + 1,*n*)	otherwise

Implementing this recurrence yields:

```
int fact(int m,int n)
    {
    if (m == n) return n;
    else return m * fact(m + 1,n);
    }
```

In this version, the formal parameter *m* holds the current count, while the parameter *n* holds the final count.

Unfortunately, this new version of factorial requires us to pass an extra argument on the initial call:

```
fact(1,6) should give us a result 720
```

Moreover, if we start the current count at anything but 1, our factorial function computes the wrong result in almost all cases. We solve this problem by calling our two-argument function from a one-argument function.

```
int factorial(int n)
    {
    # now call our 2-argument function
    # with the correct initial count
    return fact(1,n);
    }


# define our 2-argument function

int fact(int m,int n)
    {
    if (m == n) return n;
    else return m * fact(m + 1,n);
    }
```

See how the *factorial* function calls the *fact* function. We can indicate that a wrapper function is desired by adding a line to the recurrence equation. For the previous function, we would have the following recurrence equation:

factorial (*n*)	is	*fact* (1,*n*)	
fact (*m*,*n*)	is	*m*	if *m* is equal to *n*
fact (*m*,*n*)	is	*m* * *fact* (*m* + 1,*n*)	otherwise

INSIDE VERSUS OUTSIDE UPDATES

The two versions of factorial shown above exhibit *outside* updates. In an outside update, the return value of the recursive call is combined with some value and the result of that combination is returned. For example, for the counting-down version of factorial, we have this return statement for the recursive case:

```
n * factorial(n - 1)
```

See how the return value of the recursive call is combined (using the multiplication operator) with the current value of *n*.

It is also possible to structure the recursion so that the update is interior to the recursive call (this is also known as *tail recursion*). Here is the recurrence for factorial that features an inside update:

fact (*a*,*n*)	is	*a*	if *n* is equal to 1
fact (*a*,*n*)	is	*fact* (*a* * *n*,*n* - 1)	otherwise

If we were to implement this recurrence, we would get something like:

```
int fact(int a,int n):
    if (n == 1) return a;
    else return fact(a * n,n - 1);
```

The first formal parameter of the function, *a*, is known as the *accumulator*, since it accumulates the final result.

Like the counting up version of *fact*, we need to pass the proper initial value for the accumulator in order for this version of the function to work properly. In this case, the proper value is 1:

```
fact(1,6) gives 720
```

Like the counting-up version of *fact*, we also need to wrap this function within a function that takes a single argument. As before, we choose a different name for the wrapper function:

```
int factorial2(int n)
    {
    return fact(1,n);
    }

int fact(int a, int n)
    {
    if (n == 1) return a;
    else return fact(a * n,n - 1);
    }
```

In your *clab* directory, create a directory named *recur*. In the *recur* directory, create a file named *recur.c*.

In it, place an inside-update version of the counting-up version of factorial. Hint: this version will take three arguments: the accumulator, the current count, and the final count.

Test your implementation using a main function to call your factorial function and print out the results in a meaningful way.

GENERALIZATION

Factorial (when restricted to positive inputs) is just a specialized version of the general product function that you see in Mathematics texts:

$$\prod_{i=m}^{n} f(i)$$

For factorial, *m* is set to 1 and the function *f* is set to the identity function (the function that when given a value simply returns the given value).

In your *recur.c* file, define a function named *prod* that takes three parameters: *m*, *n*, and *f*. This is the recurrence for *prod*:

$prod(m,n,f)$	is	$f(m)$	if m is equal to n
$prod(m,n,f)$	is	$f(m) * prod(m+1,n,f)$	otherwise

Define two more functions. The first is one named *id* that computes the identity function. It should take a single argument and simply return that argument. The second is the square function:

```
int sqr(int ) { return x * x; }
```

Now add the following test cases to your *main* function:

```
print("prod(1,5,&id) is ",prod(1,5,id),"(should be 120)")
print("prod(1,5,&sqr) is ",prod(1,5,square),"(should be 14400)")
```

The first print statement test whether we can compute factorial correctly. The second tests whether we can compute the product of the squares from 1 to 5. In this, you should use function pointer, which we will learn few lectures later. While defining the prod function, we will use the following definition.

```
int prod(int m, int n, int (*fptr)(int x))
    {
    if (m == n) return fptr(m);
    else return fptr(m) * prod(m+1,n,fptr);
    }
```

EGYPTIAN MULTIPLICATION

The ancient Egyptians were perhaps the first people on earth to come up with the idea of binary arithmetic when they developed their method of multiplication. The Egyptian Multiplication method is a tabular calculation method that lends itself to a straightforward recursive implementation. Here is the recurrence for the first half of the algorithm for multiplying *a* times *b*:

$emult(a,b)$	is	$up(a,b,1)$	
$up(a,b,c)$	is	$up(a,2*b,2*c)$	if c is less than a
$up(a,b,c)$	is	$down(a,b,c,0)$	otherwise

Note the use of the wrapper function.

Here is the recurrence for the second half of the algorithm:

$down(a,b,c,d)$	is	d	if a is less than one
$down(a,b,c,d)$	is	$down(a-c,b/2,c/2,d+b)$	if c is less than or equal to a
$down(a,b,c,d)$	is	$down(a,b/2,c/2,d)$	otherwise

Here is an implementation of the first line of the first recurrence equation:

```
int emult(int a,int b)
    {
    return up(a,b,1);
    }
```

Define the *up* and *down* functions and place them in *recur.c*. Test the *emult* function with your *main* function.

SUBMISSION

To submit your activity, make sure you are in the *recur* directory. Run the command:

```
submit clab mr recur <your-email-iiitb.org-address>
```