Recursion

Great fleas have little fleas upon their
backs to bite 'em,
And little fleas have lesser fleas, and
so *ad infinitum.*
And the great fleas themselves, in turn,
have greater fleas to go on;
While these again have greater still,
and greater still, and so on.

Recurrence Relationships

Many interesting objects are defined by *recurrence relationships*. For example,

a) Factorials: n! = 1 when n=0, and n*(n-1)! when n > 0

b) Greatest Common Divisor(GCD) of a, b;

    (assume a>b)
    if b==0, then GCD(a, b) = a;
    otherwise, if b==1, GCD(a, b) = 1;
    otherwise GCD(a, b) = GCD(b, a%b))

c) Fibonacci numbers

    F(0) =0, F(1) = 1; F(n) = F(n-1)+F(n-2) when n>1

d) A LIST is either

- The *empty list* which contains no elements, or
- An element known as *first*, followed by a list.

These definitions are all *self-referential*. Each of the objects is defined in terms of itself. Such items are easily dealt with by recursive functions.

Example: The Factorial function

A recursive function for computing x!

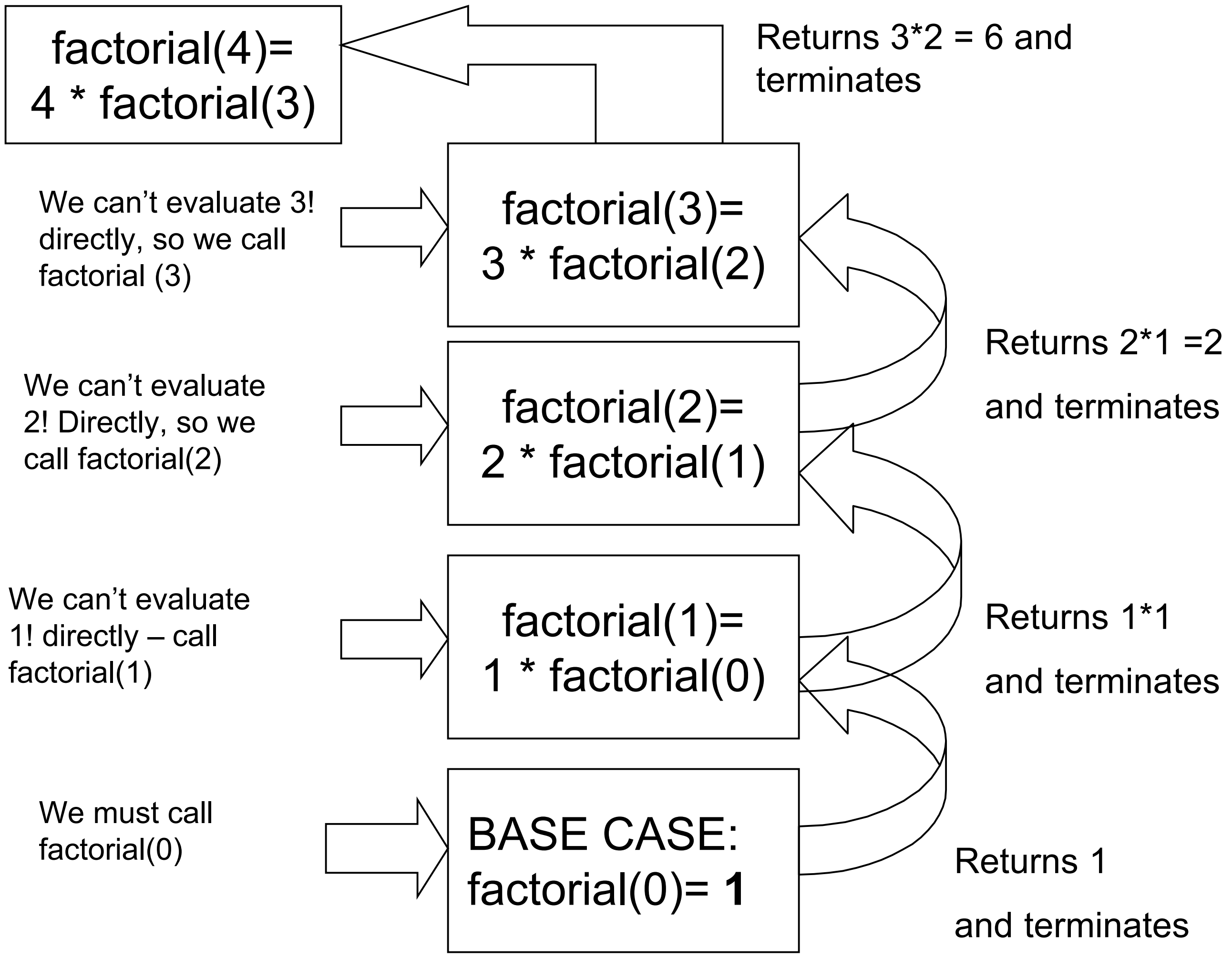
```
int factorial (int x)
{
   //base case
   if (x ==0) return 1;

   //recurrence case
   return x * factorial (x – 1);
}
```

This function illustrates all the important ideas of recursion

- A **base (or stopping) case**
  - Code first tests for stopping condition (is x ==0 ?)
  - Provides a direct (non-recursive) solution for the base case, (0! = 1)
- The **recurrence case**
  - Expresses solution to problem in 2 (or more) smaller parts
  - Invokes itself (factorial) to compute (at least one of) the smaller parts, which eventually reaches the base case

Trace of a call to Factorial: int z = factorial(4)



***finally*, factorial(4) computes 4*6, returns 24, and terminates**

Example 2: count zeros in an array

The problem is: given a vector of integers, how many of its elements are zero?

Thinking about the problem:

Suppose we examine just the last element of the vector. If it’s zero, then the total number of zeros is just one more than the number of zeros in the rest of the vector; otherwise, the total is the same as the number of zeros in the rest of the vector. All we need to know is the position of the last element and the number of zeros in the rest of the vector. Also, our knowledge of C++ tells us that the first position of a vector is position [0].

We can sketch a solution as follows:

```
int countZeros( vector V, int lastPosition)
{
        if (V[lastPosition] == 0)
             return 1 + count of zeros in the rest of
                the array;
        else
         return  count of zeros in the rest of the
                array;
}
```

Coding the recurrence relationship, then, will need the index of the last element of the vector we are examining. Each recursive call will be to the next lower position in the vector.

We need to identify a base case, and that is a vector with just 1 element.

Putting these ideas together, our finished code is:

```
int countZeros( const vector<int> & V, int lastPosition){
        // base case
    if (lastPosition == 0) return V[0]==0? 1: 0;
        //recurrence
    if (V[lastPosition] == 0)
        return 1 +  countZeros(V, lastPosition – 1);
    else
        return   countZeros(V, lastPosition – 1);
}
```

Example 3: Another way to count zeros

We may also think of a vector as having 2 halves: the number of zeros in the vector is just the sum of the zeros in the two halves. We will recursively count the zeros in a piece of the array by splitting it into halves and summing the counts of zeros in each half. As before, the base case arises when the function examines just 1 element. We need the recursive function to receive as parameters the positions of the first and last elements of the part of the vector being examined.

```
int CountZeros2( const vector<int> & V, int
   lowIndex, int highIndex) {
// base case occurs when lowIndex and highIndex
   are equal;
   if (lowIndex == highIndex)
         return V[lowIndex] == 0? 1: 0;
// recurrence part requires us to count the zeros in
   each half, and add them:
   int mid = (lowIndex + highIndex) / 2;
   return
     CountZeros2(V, lowIndex, mid)
   +CountZeros2(V, mid+1, highIndex)
}
```

Writing Recursive Functions

If we happen to have the recurrence relationship, then writing a recursive function to implement it is largely a mechanical process:

1. Test first for the base case. If it is true, provide a solution for the base case and **STOP**
2. Split the problem into (at least) 2 parts, one (or possibly both) of which is similar in form to the original problem.

That is about all there is to writing a recursive function, and to write it correctly, we must ensure that the function terminates:

3: Guarantee that eventually, the subparts will reach the base case. Otherwise, your code may run forever (or until it crashes, whichever comes first)

Nonterminating Recursive Function

These are ill-formed versions of the factorial function:

```
int BadFactorial(int x) {
   return x * BadFactorial(x-1); //Oops! No Base Case
}

int AnotherBadFactorial(int x) {
   if (x == 0) return 1;
    return x* (x-1) *  AnotherBadFactorial(x -2);
   //Oops! When x is odd, we never reach the base case!!
}
```

Linear and tree recursion

The factorial function and the first version of counting zeros are said to be **linear recursive** functions. A function is linear recursive when no pending operation involves another recursive function call (to the same function). For example in **fact**, the pending operation is a multiplication.

The second count of zeros (countzeros2) requires another recursive function call along with the pending operation (addition). When a recursive function requires at least 1 (or more) recursive call to evaluate the pending function, then it is called **tree recursive**.

Pending Operations and Tail Recursion

The functions we just examined required us to perform an addition or multiplication after the recursive function returns a value. When a recursive function has operations that are performed after the recursive call returns, the function is said to have **pending operations.**

A recursive function with no pending operations after the recursive call completes is defined to be **tail recursive.** It is desirable to have tail-recursive functions, because

a) the amount of information that gets stored during computation is independent of the number of recursive calls, and

b) some compilers can produce optimized code that replaces tail recursion by iteration (saving the overhead of the recursive calls)

From these definitions, it is clear that tree recursive functions can’t be tail recursive (but in almost all cases, a linear-recursive version can be written – see exercises)

It is possible to rewrite a non-tail-recursive function as tail recursive We will need to keep track of intermediate results, instead of letting the recursive call mechanism do that for us.

Converting Recursion to Tail-recursion

The general idea is to use an auxiliary parameter to hold intermediate results, and to incorporate the pending operation by suitably manipulating the auxiliary parameter. It is usually convenient to introduce an auxiliary function - the reason for this is to keep the user interface simpler – the user of the function doesn't need to provide an auxiliary parameter.

For Factorial(x), the pending operation is to multiply the value of factorial(x-1) by x; This suggests initializing the pending value to 1, and multiplying this by the parameter x.

When we do so, we get this version of x!

A tail-recursive Factorial Function

We will use *indirect* recursion and an auxiliary function to rewrite factorial as tail-recursive: **int factAux (int x, int result) {**

```
  if (x==0) return result;
  return factAux(x-1, result * x);
}
int tailRecursiveFact( int x) {
    return factAux (n, 1);
}
```

It’s important to see that we have removed the pending operation by using an intermediate variable, the parameter **result**, to keep track of the partial computation of x! – this results in a tail-recursive function

Equivalence **of recursion, while loops**

We can rewrite any recursive function as an iterative function (using a for- or while loop). An iterative factorial function is

```
int iterativeFact( int x) {
int result = 1;
for (int i = 1; i <= x; ++i)
        result *= I;
return result;
}
```

how can we get from the recursive function to the iterative one?

Tail Recursion to Iterative functions

A tail recursive function has this form

```
F(x) {
  if (baseProperty)
      return G(x) // work done in base case
  return F(H(x));
//H(x) is the work done in the recursive case
```

We can mechanically derive an iterative version:

```
F(x){
   int temp = x;
   while (! BaseProperty) {
        temp = x;
        x = H(temp);
}
   return G(x);
}
```

(thanks to Tom Anastasio for this idea)

Example: Factorial (what else?)

Recall the tail recursive factorial:

```
fact(x,1) {    //where x is argument, 1 is intermediate result
   if (x==0) return result;
   return fact(x-1, result*x)
}
```

Base is x==0

G(x) = nothing. Just return result

H(x) = result= result * x; x=x-1;

Example (continued)

Substituting the parts into the iterative skeleton, we get:

```
{result = 1;
temp = x;
while (!(x==0)){ //G(x) is empty. Just return result
        temp = x;
        result = result * temp;   //H(x) is
        x = x – 1;
}
```

Review Problems:

1. Using the definition given for fibonacci function, write a recursive function to compute F(n)
2. Rewrite the function you wrote in 1 as a tail-recursive function
3. Rewrite the tail-recursive function you wrote in 2 as an iterative function
4. (very hard) what is the run-time efficiency (big Oh) for each of the functions you wrote?

5. Trace the calls (as was done for factorial) to evaluate F(5) for each of the 3 functions you wrote above. How many times did your function call F(1)?
6. Define tail recursion, base case, recurrence relation, and tree recursion.
7. When should a recursive function test for the base case? Why?