BIRZEIT UNIVERSITY

Recursive Programming

Introduction

When we write a method for solving a particular problem, one of the basic design techniques is to break the task into smaller subtasks. For example, the problem of adding (or multiplying) n consecutive integers can be reduced to a problem of adding (or multiplying) n-1consecutive integers:

```
1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]

1 * 2 * 3 *... * n = n * [1 * 2 * 3 * .. * (n-1)]
```

Therefore, if we introduce a method `sumR(n)` (or `timesR(n)`) that adds (or multiplies) integers from 1 to n, then the above arithmetics can be rewritten as

```
sumR(n) = n + sumR(n-1)

timesR(n) = n * timesR(n-1)
```

Such functional definition is called a **recursive** definition, since the definition contains a call to itself. On each recursive call the argument of `sumR(n) (or timesR(n))` gets smaller by one. It takes n-1 calls until we reach the **base case** - this is a part of a definition that does not make a call to itself. Each recursive definition requires base cases in order to prevent infinite recursion.

In the following example we provide iterative and recursive implementations for the addition and multiplication of n natural numbers.

```
int sum(int n)
{
   int res = 0;
   for(int i = 1; i = n; i++)
      res = res + i;

   return res;
}
```

```
int sumR(int n)
       {
         if(n == 1)
            return 1;
         else
            return n + sumR(n-1);
       }
```

To solve a problem recursively means that you have to first redefine the problem in terms of a smaller sub-problem of the same type as the original problem. In the above summation problem, to *sum-up n* integers we have to know how to *sum-up n-1* integers. Next, you have to figure out how the solution to smaller sub-problems will give you a solution to the problem as a whole. This step is often called as a *recursive leap of faith*. Before using a recursive call, you must be convinced that the recursive call will do what it is supposed to do. You do not need to think how recursive calls works, just assume that it returns the correct result.

BIRZEIT UNIVERSITY

Examples of a recursive function:

1. Compute factorial of a number

Example : Let us consider the Factorial Function

n! = n * (n-1) * (n-2) * … * 2 * 1

0! = 1

Iterative solution:

```
int fact(int n)
{
      int p, j;
      p = 1;
      for ( j=n; j>=1; j--)
      p = p* j;
      return ( p );
}
```

Recursive definition:

In the recursive implementation there is no loop. We make use of an important mathematical property of factorials. Each factorial is related to factorial of the next smaller integer :

n! = n * (n-1)!

To make sure the process stops at some point, we define 0! to be 1. Thus the conventional mathematical definition looks like this:

n! = 1 **if n = 0**

n! = n*(n-1)! **if n > 0**

This definition is recursive, because it defines the factorial of n in terms of factorial of n – 1.

The new problem has the same form, which is, now find factorial of n – 1 .

In C:

```
int fact(int n)
{
      if (n ==0)
            return (1);
      else
            return (n * fact(n-1));
}
```

The Nature of Recursion

1) One or more simple cases of the problem (called the *stopping cases*) have a simple non-recursive solution.
2) The other cases of the problem can be reduced (*using recursion*) to problems that are closer to stopping cases.
3) Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

In general:

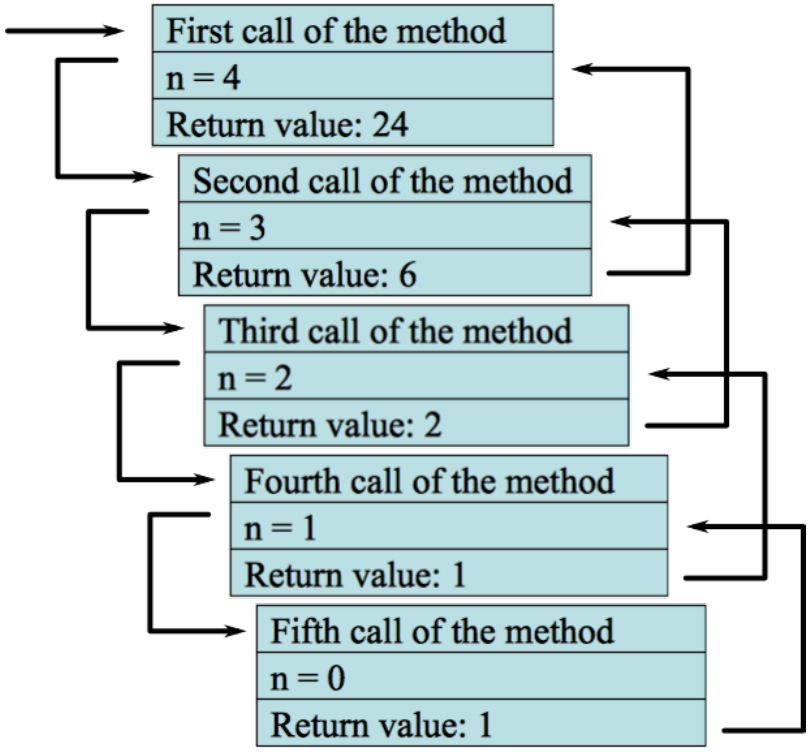
```
if (stopping case)
      solve it
else
      reduce the problem using recursion
```

Tracing a Recursive Function

Let us try to follow the logic the computer uses to evaluate any function call. It uses a stack to keep track of function calls. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement (this gives the computer the return point after execution of the function)

In the factorial example, suppose "main" has a statement

f= factorial (4);



2. Find sum of squares of a series.

Here we are interested in evaluating the sum of the series

$m^2 + (m + 1)^2 + (m + 2)^2 + \ldots\ldots\ldots + (n)^2$

We can compute the sum recursively, if we break up the sum in two parts as shown below:

$m^2 + [(m + 1)^2 + (m + 2)^2 + \ldots\ldots\ldots + (n)^2 \quad]$

Note that the terms inside the square brackets computes the sum of the terms from m+1 to n.

Thus we can write recursively

sumsq(m, n) = m^2 + sumsq(m+1, n)

The sum of the terms inside the square brackets can again be computed in similar manner by simply replacing m with m+1. The process can be continued till m reaches the value of n. Then sumsq(n, n) is simply (n)2

Here is the recursive function:

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

Trace the above recursive function to find sumsq(2,5).

```
sumsq(2, 5) = m^2  + sumsq (3,5)
            = 4 +      sumsq(3,5)
            = 4 +      9 +      sumsq(4,5)
            = 13 +           16 +     sumsq(5,5)
            = 29 +                                25
            = 54
```

3. Consider the following recursive function:

```
int speed (int N)
{
   if (N == 2) return 5;
   if (N % 2 == 0)
               return (1 + speed(N/2));

   else
               return (2+speed(3 + N));
 }
```

Trace the function for N= 7.

```
Speed(7) = 2 + speed(10)
             =  2 + 1 + speed(5)
             =  3 + 2 + speed(8)
             =  5 + 1 + speed (4)
             =  6 + 1 + speed (2)
             =  7 + 5
             = 12
```

4. Consider the following recursive function

```
int value(int a, int b) {

     if (a <= 0)
          return 1;
     else
          return (b*value(a-1,b+1));
}
```

Let us trace the calls

```
a) value(1, 5)
=  5 * value( 0, 6)
=  5 * 1
= 5

b) value(3, 3)
= 3 * value(2, 4)
= 3 * 4 * value(1, 5)
= 3 * 4 * 5 * value( 0, 6)
= 3 * 4 * 5 * 1
= 60
```

5. To print a user-entered string in reverse order

Here is a recursive function that reads the characters of a string from the keyboard, as they are being typed, but prints them out in the reverse order. The function needs to know how many characters would be read before it starts printing. Obviously, printing cannot start until all the characters have been read. The function uses an internal stack of the computer to store each character as it is being read.

```
void print_reverse(int n)
{
     char next;

     if (n == 1) {       /* stopping case */
          scanf("%c",&next);
          printf("%c", next);
     }
     else {
          scanf("%c", &next);
          print_reverse(n-1);
          printf("%c",next);
     }
     return;
}
```

6. Binary Search

Given an array with elements in the ascending order, the problem is to search for a target element in the array. The function should return the position of the element which matches the target. If the target is not found, it should return -1.
Here is the recursive version of the binary search. The stopping case is when the target is found. If the target is not found, then the function is recursively called again with either the left half of the array or the right half of the array.

```
int targetsearch ( int target, int A[], int  n )
{
       return Rbinary(target, A, 0, n-1);

}


int Rbinary ( int target, int A [ ], int left, int right)
{
   int mid;
   if ( left > right )
         return -1 ;
   mid =  ( left + right )/ 2;
   if   ( target ==  A[mid] )
           return mid;
   if  ( target < A [mid])
           return Rbinary (target, A, left, mid-1 );
   else
           return Rbinary (target, A, mid+1, right );
}
```