Recursion - I

Recursion is a powerful problem-solving strategy. Solves large problems by reducing them to smaller problems of the **same form.** The subproblems have the same form as the original problem.

To illustrate the basic idea, imagine you have been appointed as funding coordinator for a large charitable organization. Your job is to raise one million dollars in contributions to meet the expenses of the organization. If somebody can write out a check for the entire amount, your job is easy. On the other hand, it may not be easy to locate persons who would be willing to donate one million dollars.

However people don't mind donating smaller amounts. If lets say $100 is a good enough amount, than all you have to do is to call 10,000 friends to complete the task. Well, it is going to be a tall order for you, but you know that your organization has a reasonable supply of volunteers across the country.

You may start off by finding 10 dedicated supporters in different parts of the country and appoint them regional coordinators. So each person now has to raise $100,000. It is simpler than raising one million, but hardly qualifies to be easy.

Maybe they can adopt the same strategy, i.e. delegate parts of the job to 10 volunteers each within their region and asking each to raise $10,000. The delegation process can continue until there are volunteers who have to go around raising donations of $100 each from individual donors.

The following structure is a psuedocode for the problem: **Ask funding coordinator to collect fund**

```
void collect (int fund)
{
     if ( fund <=100) {
          contact individual donor.
     } else {
          find 10 volunteers.
          Get each volunteer to collect fund/10 dollars
          Pool the money raised by the volunteers.  }
}
```

Notice that the *basic nature* of the problem remains the same, i.e. collect n dollars, where value of n is smaller each time it is called. To solve the problem you can invoke the same function again.

Having a function to call itself is the key idea of **recursion** in the context of programming. A structure providing a template for writing recursive functions is as follows:

```
If  (test for a simple case) {
        Compute simple solution without recursion

} else  {
        break the problem into similar subproblems
        solve these by calling  function recursively.
        Reassemble the solution to the subproblems
```

Recursion is an example of ***divide-and-conquer problem solving strategy***. It can be used as an alternative to iterative problem solving strategy.

Example of a recursive function: 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);

when main calls factorial, computer creates a new stack frame and copies the argument value into the formal parameter n.

main fact1

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

n = 4

To evaluate function fact1, it reaches the point where it needs the value of fact (n-1) to be multiplied by n This initiates a recursive call. At this stage picture is like this

main fact1

n=4

```
if (n ==0)
          return (1);
     else
          return (n *fact(n-1));
                     ?
```

As current value of n is 4, n-1 takes the value 3, and another fact call is invoked, as shown below

main fact1 fact2

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

n= 3

and the process continues till fact is called 5 times and n gets the value 0:

Main fact1 fact2 fact3 fact4 fact5

```
if (n ==0)
          return (1);
     else
          return (n * factorial(n-1));
```

n = 0

Now the situation changes.

Because n is 0, the function parameter can return its result by executing the statement return(1);

The value 1 is returned to the calling frame, which now becomes the top of stack as shown:

Main fact1 fact2 fact3 fact4

```
if (n ==0)
          return (1);
     else
          return (n * factorial(n-1));
                      value 1
```

n = 1

Now the computation proceeds back through each of recursive calls. In above frame n=1 so it returns the value 1 to its caller , now the top frame shown here:

Main fact1 fact2 fact3

```
if (n ==0)
          return (1);
     else
          return (n * factorial(n-1));
                     value 1
```

n =2

Because n is 2, the value 2 is passed back to previous level:

Main fact1 fact2

```
if (n ==0)
          return (1);
     else
          return (n * factorial(n-1));
                      value 2

 n = 3
```

Now the value returned equals 3 x 2 to previous level as shown:

Main fact1

```
if (n ==0)
          return (1);
     else
          return (n * factorial(n-1));
                      value 6

 n = 4
```

Finally the value 4 x 6 (= 24) is returned to the main program.

We can summarize the steps involved in ***tracing this function*** for n= 4

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

fact (4) = 4 * fact (3)
= 4 * 3 * fact (2)
= 12 * 2 * fact (1)
= 24 * 1 * fact (0)
= 24 * 1
= 24

Example of another recursive function: to 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:
$m^2 \quad + \quad [\,(m + 1)^2 + (m + 2)^2 + \ldots\ldots\ldots + (n)^2\,]$

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);
 }
```

Tracing a recursive function.

To understand recursion, let us trace the working of couple of recursive functions step-by-step.

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

```
sumsq(2, 5) = m²  + sumsq (3,5)
          = 4 +     sumsq(3,5)
          = 4 +     9 +      sumsq(4,5)
          = 13 +             16 +        sumsq(5,5)
          = 29 +                          25
          = 54
```

2.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
```

2. 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)

b) value(3, 3)

a) foo(1, 5)

```
=  5 * foo( 0, 6)
=  5 * 1
= 5
```

b) foo(3, 3)

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

Problem solving using Recursion:

In this section we shall study some recursive solutions to well known problems. For most of them it is also possible to find iterative solutions. We shall also discuss the time complexity of some of the algorithms by using the **recurrence relations**.

Exponentiation:

We consider exponentiation which involves raising an integer base to an integer power. The basic strategy would be to use successive multiplication by the base. However, as the result becomes quite large even for small powers, it would work only if there is a machine to hold such large integers. For larger powers, one could use a stack.

Evaluation of x^n requires n-1 multiplications. Here is a simple recursive function to carry out the exponentiation.

```
power( x,  n)
{
      if ( n==0)
          return 1;
      if(n==1)
          return x;

       else
           return ( x * power( x , n – 1 )  ;
}
```

Complexity Analysis :

The problem size reduces from n to n – 1 at every stage, and at every stage one multiplication operation is involved. Thus total number of operations T(n) can be expressed as sum of T(n-1) and one operation as the following ***recurrence relation***:

$T(n) = T(n - 1) + 1$ ..(1)

In turn T(n-1) operations can expressed as a sum of T(n-2) and one operation

$T(n - 1) = T(n - 2) + 1$ …(2)

Substituting for $T(n - 1)$ from relation (2) in relation(1) yields

$T(n) = T(n - 2) + 2$ …(3)

Also we note that

$T(n - 2) = T(n - 3) + 1$ …(4)

Substituting for $T(n - 2)$ from relation(4) in relation (3) yields

$T(n) = T(n - 3) + 3$ … (5)

Following the pattern in relations (1) , (3) and (5), we can write a generalized relation

$T(n) = T(n - k) + k$ …(6)

To solve the generalized relation (6), we have to find the value of k. We note that T(1) , that is the number of operations to raise a number x to power 1 needs just one operation. In other words

$T(1) = 1$ ...(7)

Thus we set the first term on right hand side of (6) to 1 to get

$n - k = 1$

that is $k = n - 1$

Substituting this value of k in relation (6), we get

$T(n) = T(1) + n - 1$

Now substitute the value of T(1) from relation (7) to yield the solution

$T(n) = n$ ..(8)

When the right hand side of the relation does not have any terms involving T(..), we say that the recurrence relation has been solved. So what is the time complexity of the exponentiation algorithm? The right hand side of (8) indicates that it is simply O(n) .

An efficient recursive algorithm for exponentiation:

Let us now develop an efficient version of the exponentiation algorithm. Note that x can be raised to power 16 by raising x^2 to the power 8

$x^{16} = (x^2)^8$

which can be viewed as two different operations
$y = x^2$

$x^{16} = (y)^8$
Thus instead of multiplying 15 times, we can get the result by multiplying x^2 seven times. Further we note that y^8 can be obtained by a similar process.

$y^8 = (y^2)^4$

Thus at every stage the number of multiplications is reduced by half.

Note the similarity with binary search. A higher power can be obtained from its lower power (i.e. power/2).

We present below a recursive algorithm. Raising a number to power n can be reduced to the problem of raising x^2 to power n/2. The problem size can be reduced by recursively, till n equals 1. However note that when the power is odd, it needs one more multiplication.

$x^{17} = (x^2)^8 . x$

Here is the algorithm which takes care of both even and odd powers:

```
power( x,  n)
{
      if ( n==0)
          return 1;
      if(n==1)
          return x;

      if (n is even)
          return power ( x * x,  n / 2);

      else
           return power( x * x, n / 2 ) * x  ;
}
```

Complexity of the efficient recursive algorithm for exponentiation:

At every step the problem size reduces to half the size. When the power is an odd number, an additional multiplication is involved. To work out time complexity , let us consider the ***worst case,*** that is we assume that at every step an additional multiplication is needed. Thus total number of operations T(n) will reduce to number of operations for n/2, that is T(n/2) with an additional multiplication operation. We are now in a position to write the ***recurrence relation*** for this algorithm as

$T(n) = T(n/2) + 1$ ..(1)
$T(1) = 1$ ..(1)

To solve this recurrence relation, we note that T(n/2) can be expressed as

$T(n/2) = T(n/4) + 1$ ..(2)

Substituting for $T(n) = T(n/2) + 1$ from (2) in relation (1) , we get

$T(n) = T(n/4) + 2$

By repeated substitution process explained above, we can solve for T(n) as follows

$T(n) = T(n/2) + 1$
$= [T(n/4) + 1] + 1$
$= T(n/4) + 2$
$= [T(n/8) + 1] + 2$
$= T(n/8) + 3$

Now we know that there is a relationship between 3 and 8, and we can rewrite the recurrence relation as
$T(n) = T(n/ 2^3) + 3$

We can continue the process one step further, and rewrite the relation as

$T(n) = T(n/2^4) + 4$

Now we can see a pattern running through the various relations and we can write the generalized relation as

$T(n) = T(n/2^k) + k$

Since we know that T(1) = 1, we find a substitution so that the first term on the right hand side reduces to 1. The following choice will make this possible

$2^k = n$

We can get the value of k by taking log of both sides to base 2, which yields

$k = \log n$

Now substituting this value in the above relation, we get

$T(n) = 1 + \log n$

Thus the time complexity of this recursive algorithm in terms of Big-O is

$O(\log n)$

Thus we can say that this algorithm runs in LOGARTHIMIC TIME, and obviously is much more efficient than the previous algorithm.