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)

```
=  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
```

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**.

Array handling :

Given an array A containing n integers, let us develop a recursive solution to determine the number of odd integers in the array. Every element of A needs to be examined, and it is best to start from the last element. If it is odd we add one to the value being sent and call the function recursively to examine the remaining n-1 elements. The recursion can be stopped when there is a single element left in the array. Here is a recursive function:

```
int arrayOdd(int A[ ], int n)
{
     if(n < 1)
          return 0;
     else
          return A[n-1]%2 + arrayOdd( A, n-1);
}
```

Here is another way to write this recursive function:

```
int arrayOdd(int A[ ], int n)
{
     if(n < 1)
          return 0;
     else{
            if (A[n-1]%2)
               return 1 + arrayOdd( A, n-1);
            else
               return arrayOdd( A, n-1);
          }

}
```

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.

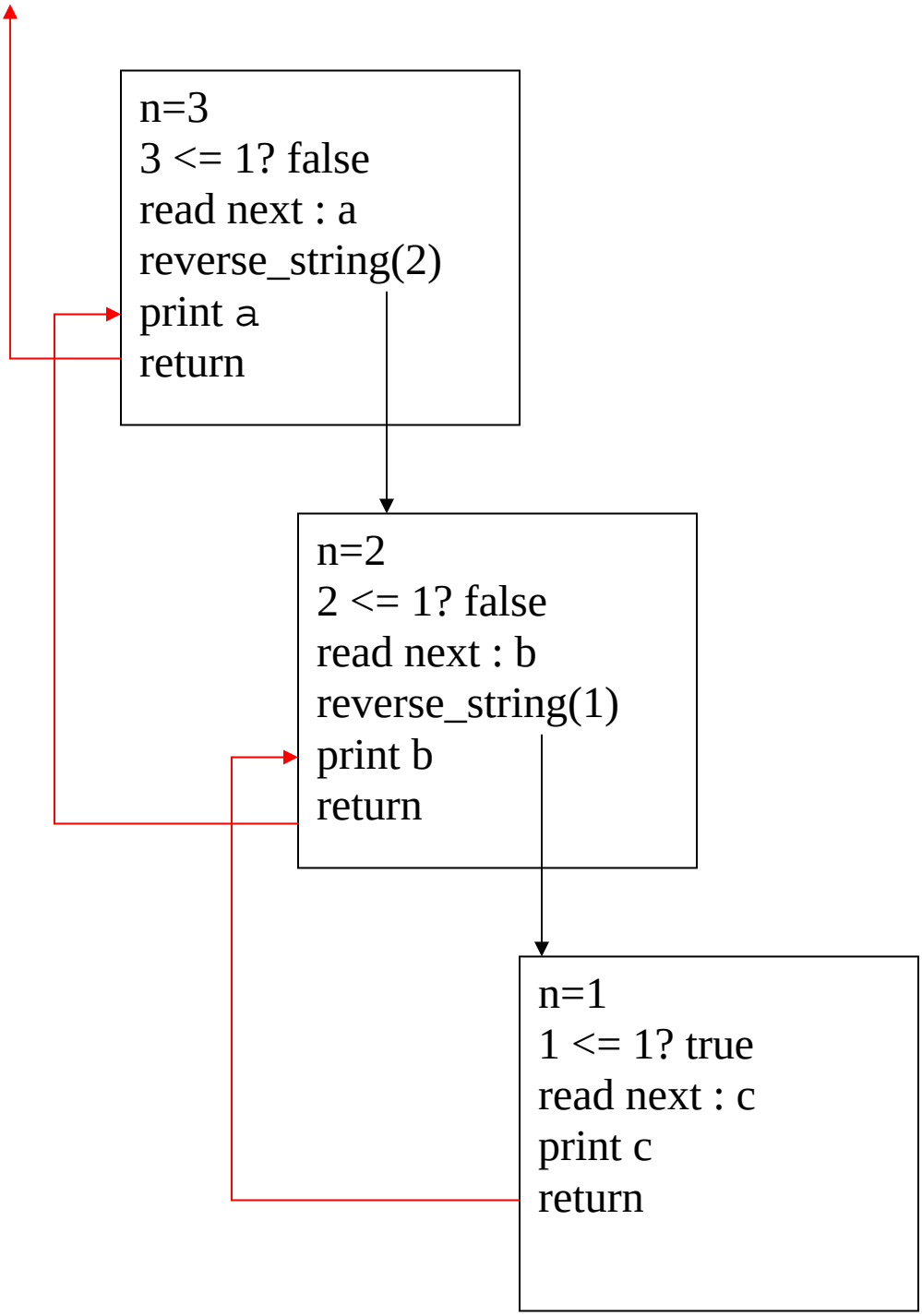
```
void print_reverse(int n)
{
     char next;

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

On the next page, you can see the trace of the function for a sample case where it handles a string of length 3.

Trace of print_reverse for input string `abc`

reverse_string(3);



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. The recursive solution would be very similar that used to compute factorial of a number. 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 based solution.

$x^n = x * x^{n-1}$

Here is a simple recursive function to carry out the exponentiation.

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

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

The problem size reduces from n to n – 1 at every stage, and at every stage two arithmetic operations are involved. Thus total number of operations T(n) can be expressed as sum of T(n-1) and two operations. We shall work out an explicit expression for T(n) later.

An efficient recursive algorithm for exponentiation:

.

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. This gives us a lead to come up with a more efficient solution to carry out exponentiation.

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 an efficient 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 just one more multiplication.

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

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

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

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

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

At every step the problem size reduces to half the size, and intuitively this function appears better than the previous one. We shall work out the relative performances of the two algorithms later.

Binary search (recursive version)
Divide and conquer implementation

We have already looked at the iterative version of binary search algorithm. 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 );
}
```