Chapter 11: Recursion

What are we studying in this chapter?

- Recursion
- Programming examples

11.1 Recursion

Recursion is a powerful tool but least understood by most novice students. Programming languages such as Pascal, C, C++ etc support recursion. Now, let us see "What is recursion? What are the various types of recursion?"

Definition: A recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem. Thus, recursion is a way to decompose a given task into smaller subtasks of same type where:

- Each subtask is a smaller example of the same task.
- The smallest example of the task has a non-recursive solution.

A recursive function is a function that calls itself during execution. This enables the function to repeat itself several times to solve a given problem. The various types of recursion are shown below:

- → Direct recursion
- → Indirect recursion

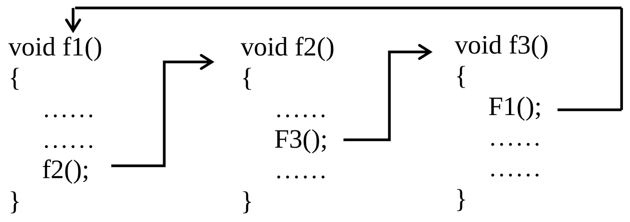
Direct recursion: A recursive function that invokes itself is said to have direct recursion. For example, the factorial function calls itself (detailed explanation is given later) and hence the function is said to have direct recursion.

```
int fact (int  n)
{
    if ( n == 0 ) return 1;

    return n*fact(n-1);
}
```

Direct recursion

- **Indirect recursion:** A function which contains a call to another function which in turn calls another function which in turn calls another function and so on and eventually calls the first function is called indirect recursion. It is very difficult to read, understand and find any logical errors in a function that has indirect recursion. For example, a function f1 invokes f2 which in turn invokes f3 which in turn invokes f1 is said to have indirect recursion. This is pictorially represented as shown below:



```
void f1()          void f2()          void f3()
{                  {                  {
    ……                 ……                 F1();
    ……                 F3();              ……
    f2();              ……                 ……
}                  }                  }
```

Indirect recursion

Now, the question is "How to design recursive functions?"

Every recursive call must solve one part of the problem using base case or reduce the size (or instance) of the problem using general case. Now, let us see "What is base case? What is a general case?"

Definition: A base case is a special case where solution can be obtained without using recursion. This is also called base/terminal condition. Each recursive function must have a base case. A base case serves two purposes:

1) It acts as terminating condition.
2) The recursive function obtains the solution from the base case it reaches.

For example, in the function factorial, 0! is 1 is the base case or terminal condition.

Definition: In any recursive function, the part of the function except base case is called general case. This portion of the code contains the logic required to reduce the size (or instance) of the problem so as to move towards the base case or terminal condition. Here, each time the function is called, the size (or instance) of the problem is reduced.

For example, in the function **fact**, n*fact(n-1) is general case. By decreasing the value of **n** by 1, the function **fact** is heading towards the base case.

So, the general rules that we are supposed to follow while designing any recursive algorithm are:

- Determine the base case. Careful attention should be given here, because: when base case is reached, the function must execute a return statement without a call to recursive function.

- Determine the general case. Here also careful attention should be given and see that each call must reduce the size of the problem and moves towards base case.

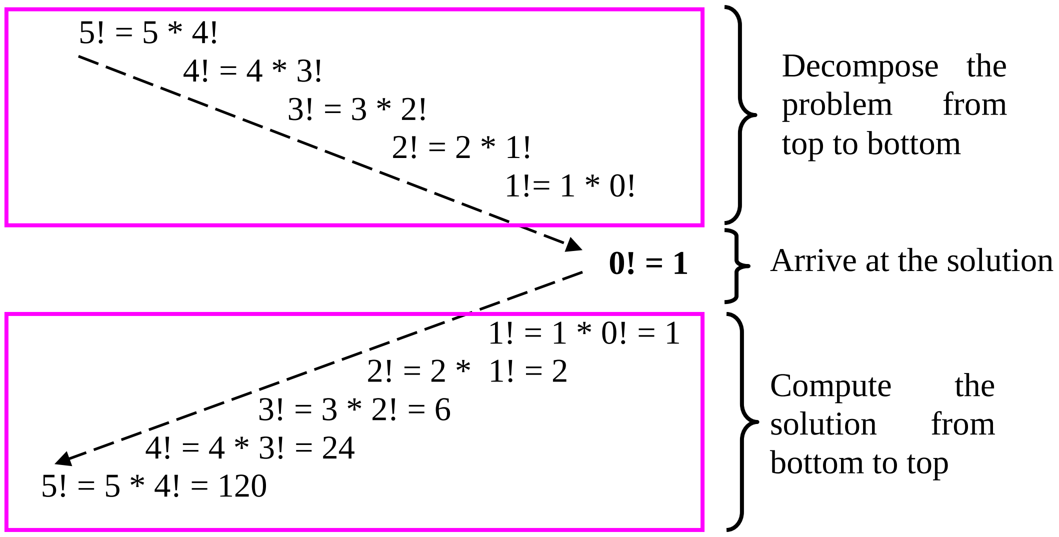
- Combine the base case and general case into a function.

Note: A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate and hence it is called infinite recursion and will crash the system.

11.1.1 Compute factorial of n

The function to compute factorial of a given number has been discussed earlier in section 8.3.9. Now, let us compute factorial of a number using recursion.

Now, let us see "How to compute factorial of 5 using recursion?" We can compute 5! as shown below:



Design: Now, let us see "How to design any recursive function?" Let us take an example of factorial function that uses recursion. Any recursive function has two elements: base case and general case.

- **Base case:** The statement that solves the problem is called base case. Every recursive function must have at least one base case. It is a special case whose solution can be obtained without using recursion. A base case serves two purposes:
  1) It acts as terminating condition.
  2) The recursive function obtains the solution from the base case it reaches.

In the above example, the base case is : 0! = 1

$$\begin{tikzcd} 0! \arrow[d] & = 1 \arrow[d] \\ \boxed{n! = 1 \text{ if } n == 0} \end{tikzcd}$$

- **General case:** The statement that reduces the size of the problem is called general case. This is done by calling the same function with reduced size. It contains the logic necessary to reduce the size of the problem. For example, while we compute 5!, initial size of the problem is 5 and then the problem is reduced to find 4!, 3!, 2!, 1! and finally we reached the base case 0! where 0! is 1. The general case can be written by looking at the problem size. It is given that we need to find 5! which can be written as shown below:

$$\begin{array}{rcccc} \textbf{i.e.,} & 5! = & 5 & * & 4! \\ & \downarrow & \downarrow & & \downarrow \\ \text{In general,} & n! = & n & * & (n-1)! \quad \text{if } n \mathrel{!=} 0 \end{array}$$

Thus, by looking at the base case and general case, the recursive definition to find factorial of *n* can be written as shown below:

$$\boxed{\begin{array}{ll} n! = 1 & \text{if } n == 0 \\ n! = n * (n-1)! & \text{otherwise} \end{array}} \quad \text{or} \quad \boxed{n! = \begin{cases} 1 & \text{if } n == 0 \\ n * (n-1)! & \text{otherwise} \end{cases}}$$

The above definition can also be written as shown below:

$$\boxed{F(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * \underbrace{F(n-1)} & \text{otherwise} \end{cases}}$$

Output: 5 * 4 * 3 *2 * 1

Exchanging the operands, Output: 1 * 2 * 3 * 4 * 5

Now, we can write the algorithm using C++ as shown below:

Example 11.1: Recursive C function to find the factorial of N

```
int fact(int n)
{
        if ( n == 0 )   return 1;           /* factorial of n when n = 0 */

        return          n*fact(n-1);        /* factorial of n when n > 0 */
}
```

The above function can be invoked as shown below:

Example 11.2: C program compute binomial co-efficient nCr = n! / ((n-r)! * r!)

```
#include <stdio.h>

/* Include: Example 11.1: Function to compute factorial of n */

void main()
{
        int     n;
        float   res;

        printf("Enter n and r\n");
        scanf("%d %d",&n, &r);

        res = fact(n) / ( fact(n-r) * fact(r));
        printf("%dC%d = %d\n",n, r, res)
}
```

Input

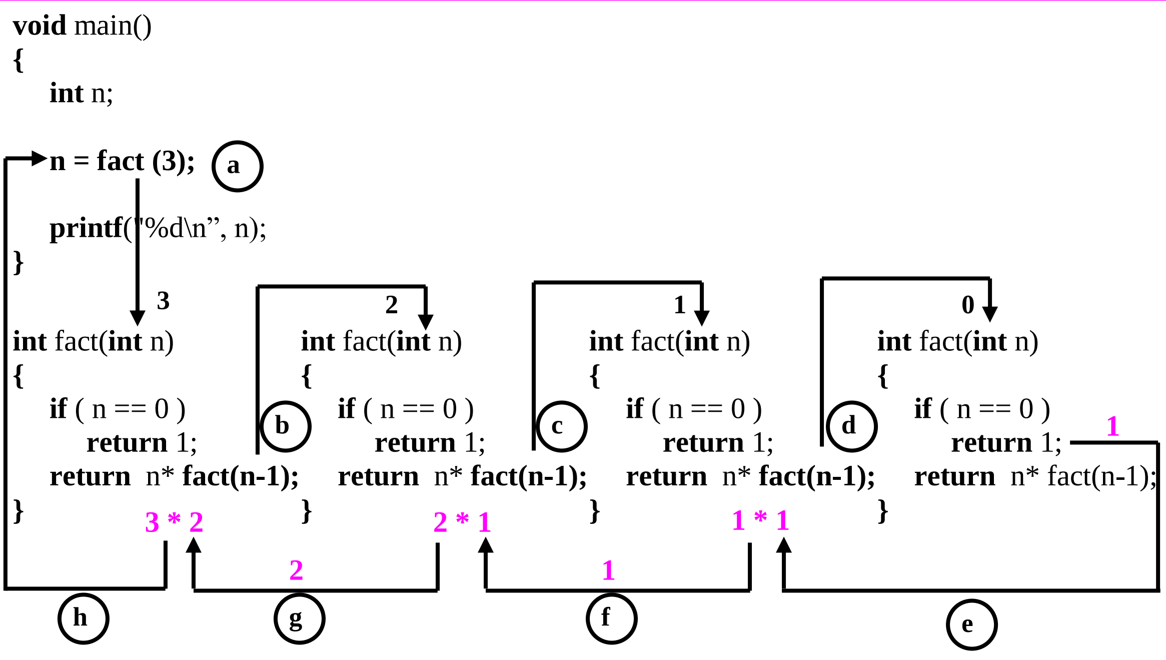
```
Enter n and r
      6      3
```

Output

$^6C_3 = 20$

11.1.2 How recursion works

Let us visualize how the same function is called repeatedly using recursion. The figure shows exactly what happens when recursive function **fact()** gets called. The execution sequence for the above factorial function is shown below:



```
void main()
{
   int n;

   n = fact (3);

   printf("%d\n", n);
}

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

Fig Steps in executing a recursive function

a. The first time when function **fact()** is called, 3 is passed to **n**.

b. Since **n** is not 0, the if-condition fails and **fact()** is called with argument **n-1** i.e., 2.

c. Since **n** is not 0, the if-condition fails and **fact()** is called again with argument **n-1** i.e., 1.

d. Since **n** is not 0, the if-condition fails and **fact()** is called again with argument **n-1** i.e., 0.

e. Since **n** is 0, control returns to previous call with value 1 and 1*1 = 1 is computed and sent to the previous point of invocation.

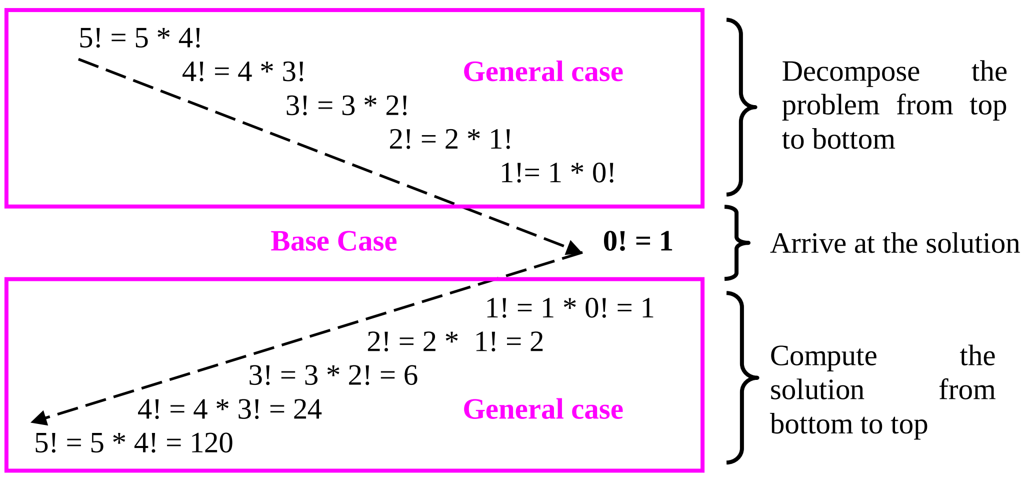
f. The resulting value **1** is passed to the point of previous invokation and 2*1 = 2 is computed and sent to the previous point of invokation.

g. The resulting value **2** is passed to the point of previous invokation and 3*2 = 6 is computed and sent to the previous point of invokation.

h. The resulting value **6** is passed to the point of previous invocation and **6** is copied into variable **n** in the main program.

In general, any recursive problem can be solved as shown below:

- Start from the given problem, decompose the problem by calling recursive function by reducing problem size and reach the base case.
- Once we reach the base case, the solution is known. The result obtained in base case is returned to more general case.
- As we solve each general case, we are able to solve the next higher general case and move towards the most general case which is the original problem as shown below:



Now, let us see "What are the rules for designing any recursive function?"

The general rules that we are supposed to follow while designing any recursive function are:

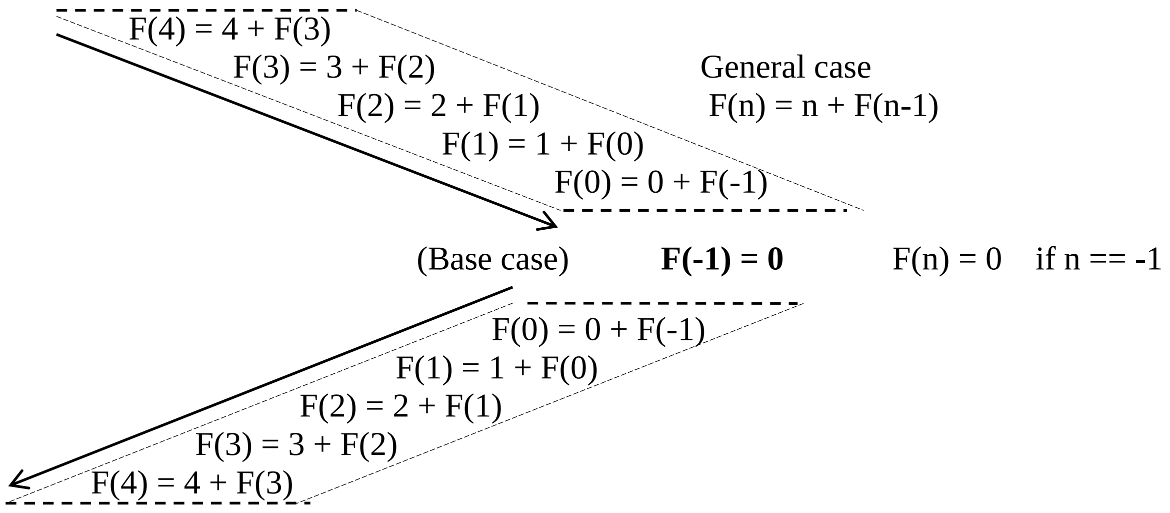
- Determine the base case. Careful attention should be given here, because: when base case is reached, the function must execute a return statement without a call to recursive function.
- Determine the general case. Here also careful attention should be given and see that each call must reduce the size of the problem moving towards base case.
- Combine the base case and general case into a function.

Note: A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate.

11.1.3 Sum of natural numbers

Consider the sum of series: 4 + 3 + 2 + 1 + 0

The sum of above series can be computed recursively as shown below:



So, the recursive definition can be written as shown below:

$$\boxed{F(n) = \begin{cases} 0 & \text{if } n == -1 \\ n + F(n-1) & \text{otherwise} \end{cases}}$$

Example 11.3: Recursive C function to find the sum of series

```
// 4 + 3+ 2 + 1 + 0 where n = 4
int F(int n)
{
        if ( n == -1 )  return 0;

        return        n + F(n-1);
}
```

OR

```
// 0 + 1 + 2 + 3 + 4 where n = 4
int F(int n)
{
        if ( n == -1 )  return 0;

        return        F(n-1) + n;
}
```

Note: If we exchange the terms in the expression n + F(n-1) shown using dotted rectangular box as: F(n-1) + n, we get the sum of the series: 0 + 1 + 2 + 3 + 4.

11.1.4 Sum of array elements

Note: By inserting an array *a* as the parameter and changing *n* to a[n] for the above recursive relation, we get sum of following series:

$$F(a, n) = a[4] + a[3] + a[2] + a[1] + a[0]$$

So, the recursive definition can be written as shown below:

$$F(a,n) = \begin{cases} 0 & \text{if } n == -1 \\ a[n] + F(a, n-1) & \text{otherwise} \end{cases}$$

Example 11.4: Recursive function to find sum of array elements from a[n-1] to a[0]

```
// sum = a[4] + a[3] + a[2] + a[1] + a[0]
float F(float a[], int n)
{
        if ( n == -1 )  return 0;

        return        a[n] + F(a, n-1);
}
```

```
// sum = a[0] + a[1] + a[2] + a[3] + a[4]
float F(float a[], int n)
{
        if ( n == -1 )  return 0;

        return        F(a,n-1) + a[n];
}
```

Note: The expression a[n] + F(a, n-1) can be written as F(a, n-1) + a[n]. If we use the expression a[n] + f(a, n-1) we get the sum of a[4] + a[3] + a[2] + a[1] + a[0]. But, if we use the expression f(a, n-1) + a[n] we get sum of a[0] + a[1] + a[2] + a[3] + a[4]. The complete program can be written as shown below:

Example 11.5: Program to find the sum of array elements

```
#include <stdio.h>

// Insert: Example 11.4 : Function to find sum of a[4] + a[3] + a[2] + a[1] + a[0]
                                    or
// Insert: Example 11.4: Function to find sum of a[0] + a[1] + a[2] + a[3] + a[4]

void main()
{
        int     n,  i;
        float   a[10], sum;

        printf ("Enter the number of elements\n");          // read number of items
        scanf (%d, &n);

        printf ("Enter %d items\n", n);                     // read n items
        for (i = 0; i < n; i++) scanf("%d", &a[i]);

        sum = F(a, n-1);                                    // find sum of n items

        printf("Result = %d\n", sum);                       // print the result
}
```

11.1.5 Print array elements and to read array elements

Consider the following array elements:

```
a[0] [1]  [2]  [3]  [4]
 10  20   30  40   50
```

Note: Changing a[n] to "print a[n]" and removing the + symbol in the above program, we can print array elements. So, the recursive definition to print array elements in reverse order can be written as shown below:

$$\boxed{F(a,n) = \begin{cases} 0 & \text{if } n == -1 \\ \underbrace{\text{print}(a[n])}, \ \underbrace{F(a, n-1)} & \text{otherwise} \end{cases}}$$

Output: 50 40 30 20 10

Exchanging the operands,

Output: 10 20 30 40 50

Note: Replacing name of function F() to PrintArray(), we can write the function as shown below:

Example 11.6: Recursive function to print array elements from a[0] to a[n-1]

```
void PrintArray (float a[], int n)
{
        if ( n == -1 )   return;

        PrintArray (a, n-1);

        printf ("%d\n", a[n]);
}
```

On similar lines, the array elements can be read from the keyboard using recursion. In the above function, replace printf() statement with scanf() statement as shown below:

Example 11.7: Recursive function to read array elements from a[0] to a[n-1]

```
void ReadArray (float a[], int n)
{
        if ( n == -1 )   return;

        ReadArray (a, n-1);

        scanf ("%d", &a[n]);
}
```

The complete program can be written as shown below:

Example 11.8: Program to print array elements

```
#include <stdio.h>

// Insert: Example 11.6: Function to print array elements
// Insert: Example 11.7: Function to read array elements

void main()
{
        int     n,  i;
        float   a[10];
        printf("Enter the number of elements\n");          // read number of items
        scanf (%d", &n);

        printf("Enter %d elements\n");                     // read n items
        ReadArray(a, n – 1);

        printf("The array elements are\n");

        PrintArray(a, n-1);                                // print array elements
}
```

11.1.6 Check for prime number using recursion

In this section, let us see "How to check whether the given number is prime or not using recursion?"

The recursive definition to check whether the given number is *prime* or not can be written as shown below:

$$F(m, i) \begin{cases} 1 & \text{if } i == 1 \\ 0 & \text{if } m \% i == 0 \\ F(m, i - 1) & \text{otherwise} \end{cases}$$

where

- Initial value of *i* is m/2 and the function is invoked as F(m, m/2)
- If m % i is zero, *m* is not prime and hence return 0
- If m % i is not zero, then decrement *i* by 1 and divide recursively
- When *i* is decremented by 1 recursively, if *i* reaches 1, then *m* is prime and hence return 1 indicating the given number is prime

Now, the given program can be written as shown below:

Example 11.9: Program to check whether the given number is prime or not using recursion

```
#include <stdio.h>
```

```
int is_prime(int m, int i)
{
        if (i ==1)  return 1;

        if( m % i == 0) return 0;

        return    is_prime(m, i-1);
}

int main()
{
        int n, flag;

        printf("Enter a positive number: ");
        scanf("%d",&n);

        flag = is_prime(n, n /2);

        if ( flag == 1 )
                printf("%d is a prime number",n);
        else
                printf("%d is not a prime number",n);
}
```

11.1.7 Convert from binary to decimal

In this section, let us see "How to convert a binary number to decimal number using recursion?"

The recursive definition to convert a number from binary number system to decimal number system can be written as shown below:

The recursive definition to convert from binary to decimal can be written as shown below:

$$\boxed{F(n, b) = \begin{cases} 0 & \text{if } (n == 0) \\ n \% 10 * b + f(n / 10, 2*b) & \text{otherwise} \end{cases}}$$

where *n* is a binary number and initial value of *b* is 1.

Note: To convert from decimal to binary, in the above recursive relation replace 10 by 2 and 2 by 10.

Example 11.10: Program to check whether the given number is prime or not using recursion

```
#include <stdio.h>

int f(int n, int base)
{
        int d;

        if (n == 0) return 0;

        return n % 10 *base + f(n/10, 2*base);
}

void main()
{
        int     n;

        printf("Enter a binary number:");
        scanf("%d", &n);

        printf("Decimal number = %d\n",f(n, 1));
}
```

Output

```
Enter a binary number: 1110

Decimal number = 14
```

Exercises:

1) What is recursion? What are the types of recursion?
2) Explain base case and general case with an example.
3) Write a recursive function to find the factorial of a number
4) Write a recursive function to find binomial co-efficient.
5) Write a recursive function to find the sum of natural numbers.
6) Write a recursive function to find the sum of elements of the array
7) Write a recursive function to print the array elements in reverse order
8) Write a recurve function to check whether the given number if prime or not
9) Write a recursive function to convert from binary to decimal
10) Write a recursive function to convert from decimal to binary