

5 Recursion

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are
 - **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
 - **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts
- Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.
- To understand recursive functions, let us take an example of calculating factorial of a number. To calculate $n!$, we multiply the number with factorial of the number that is 1 less than that number.
In other words, $n! = n * (n-1)!$

PROBLEM	SOLUTION
5!	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function

Base case is when $n = 1$, because if $n = 1$, the result will be 1 as

$$1! = 1.$$

Recursive case of the factorial function will call itself but with a smaller value of n , this case can be given as

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$

Program to calculate the factorial of a given number

```
#include <stdio.h>
#include <conio.h>
long int Fact(int);           // FUNCTION DECLARATION
int main()
{
    int num;
    long int val;
    printf("\n Enter the number: ");
    scanf("%d", &num);
    val = Fact(num);
    printf("\n Factorial of %d = %ld", num, val);
    getch();
    return 0;
}
long int Fact(int n)
{
    if(n==1)
        return 1;
    else
        return (n * Fact(n-1));
}
```


✚ **Steps of a recursive program**

- Step 1: Specify the base case which will stop the function from making a call to itself.
 Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.
 Step 3: Divide the problem into smaller or simpler sub-problems.
 Step 4: Call the function from each sub-problem.
 Step 5: Combine the results of the sub-problems.
 Step 6: Return the result of the entire problem.

✚ **The advantages of using a recursive program include the following:**

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

✚ **The drawbacks/disadvantages of using a recursive program include the following:**

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its non recursive counterpart.
- It is difficult to find bugs, particularly while using global variables.

The advantages of recursion pay off for the extra overhead involved in terms of time and space required.

5.1 Recursion Vs Iteration

Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:

- It requires stack implementation.
- It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
- Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.

Although recursion provides a programmer with certain pitfalls, and quite sharp concepts about programming. Moreover recursive functions are often easier to implement and maintain, particularly in case of data structures which are by nature recursive. Such data structures are queues, trees, and linked lists. Below are the few key difference point between recursion and iteration

Recursion	Iteration
Recursion is the technique of defining anything in terms of itself.	It is a process of executing a statement or set of statements repeatedly, until some specific condition is specified.
There must be an exclusive if statement inside the recursive function, specifying stopping condition.	Iteration involves four clear cut steps initialization, condition, execution and updation.
Not all problems have recursive solution.	Any recursive problem can be solved iteratively.
Recursion is generally a worse option to go for simple problem or problems not recursive in nature.	Iterative counterpart of a problem is more efficient in terms of memory utilization and execution speed.

5.2 Types of Recursion

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on:

- whether the function calls itself directly or indirectly (direct or indirect recursion),
- whether any operation is pending at each recursive call (tail recursive or not), and
- the structure of the calling pattern (linear or tree-recursive).

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

Direct Recursion

```
int Func1 (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
    return Func1(x-1);
}
```

Indirect Recursion

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

Non-tail recursion

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
        return Fact1(n-1, n*res);
}
```

Tail recursion

- In simple words, a recursive function is said to be linearly recursive when the pending operation (if any) does not make another recursive call to the function. For example, observe the last line of recursive factorial function. The factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another recursive call to Fact. On the contrary, a recursive function is said to be tree recursive (or non-linearly recursive) if the pending operation makes another recursive call to the function. For example, the Fibonacci function in which the pending operations recursively call the Fibonacci function.

```
int Fibonacci(int num)
{
    if(num == 0)
        return 0;
    else if (num == 1)
        return 1;
    else
        return (Fibonacci(num - 1) + Fibonacci(num - 2));
}
```

Observe the series of function calls. When the function returns, the pending operations in turn calls the function

Fibonacci(7) = Fibonacci(6) + Fibonacci(5)
 Fibonacci(6) = Fibonacci(5) + Fibonacci(4)
 Fibonacci(5) = Fibonacci(4) + Fibonacci(3)
 Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
 Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
 Fibonacci(2) = Fibonacci(1) + Fibonacci(0)

Now we have, Fibonacci(2) = 1 + 0 = 1

Fibonacci(3) = 1 + 1 = 2
 Fibonacci(4) = 2 + 1 = 3
 Fibonacci(5) = 3 + 2 = 5
 Fibonacci(6) = 5 + 3 = 8
 Fibonacci(7) = 8 + 5 = 13

Tree recursion

5.3 The Fibonacci Series

The Fibonacci series can be given as

0 1 1 2 3 5 8 13 21 34 55

That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the n th term of the Fibonacci series. The general formula to do so can be given as As per the formula, $FIB(0) = 0$ and $FIB(1) = 1$. So we have two base cases. This is necessary because every problem is divided into two smaller problems.

$$FIB(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ FIB(n-1) + FIB(n-2), & \text{otherwise} \end{cases}$$

5.4 Tower of Hanoi

- The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve $n-1$ cases, then you can easily solve the n th case'.

Initial state:

- There are three poles named as origin, intermediate and destination.
- n number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
- The disks are numbered as 1, 2, 3, 4,, n .

Objective:

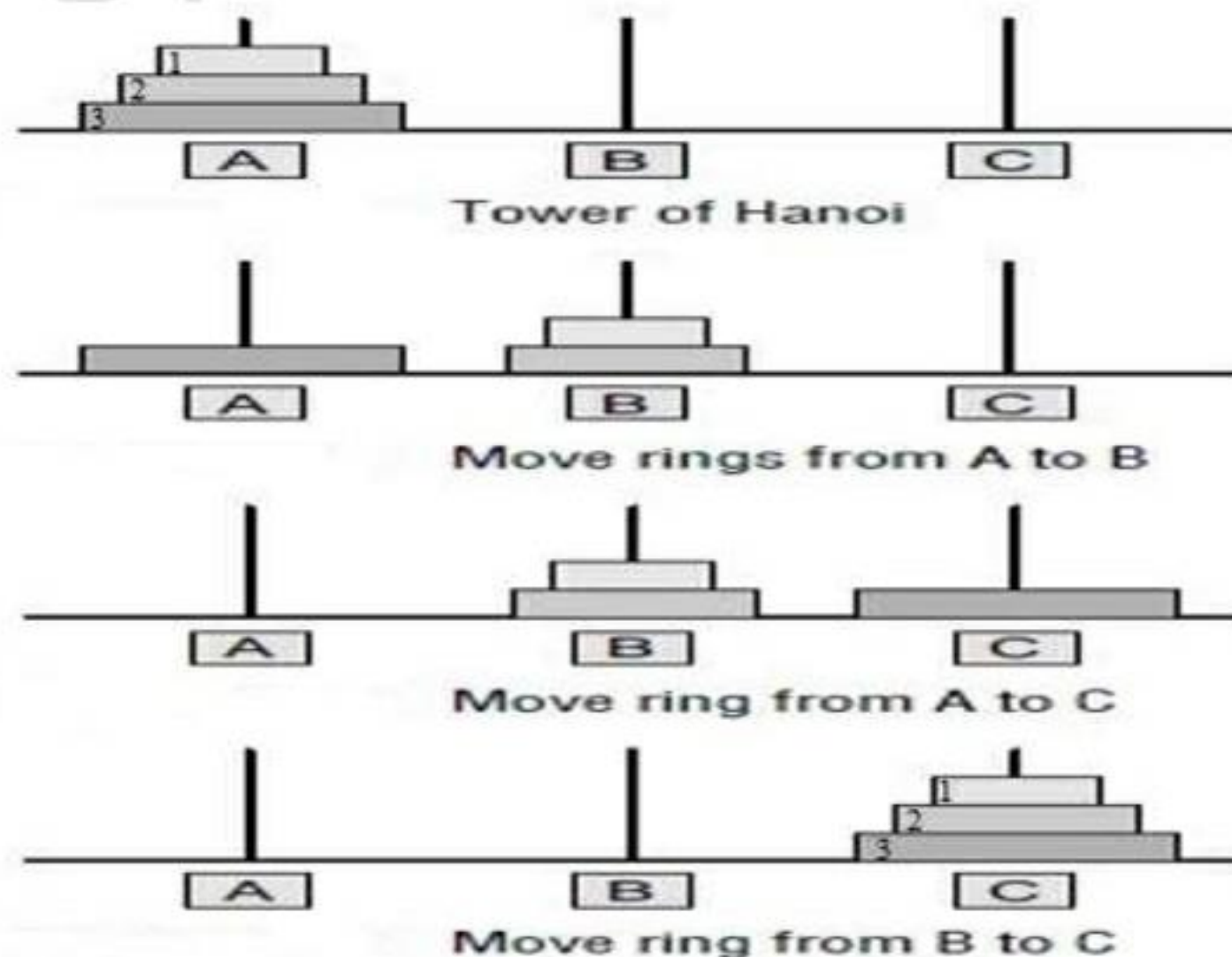
- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Conditions:

- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

Algorithm: - To move a tower of n disks from source to destination (where n is positive integer):

1. If $n == 1$
Move a single disk from source to dest.
2. If $n > 1$
 - I. Let temp be the remaining pole other than source and dest.
 - II. Move a tower of $(n-1)$ disks from source to temp.
 - III. Move a single disk from source to dest.
 - IV. Move a tower of $(n-1)$ disks from temp to dest.
3. Terminate.



Recursive solution of tower of Hanoi:

```
#include<stdio.h>
#include<conio.h>
int count =0;
// move n discs from Source (A) to Destination (C) using B as temporary
void TOH(int n,char source,char temp,char destination)
{
    if(n==1)
    {
        printf("Move Disc 1 from %c to %c\n",source,destination);
        count++;
        return;
    }
    //move n-1 discs from A to B using C as temporary
    TOH(n-1,source,destination,temp);
    printf("Move Disc %d from %c to %c\n",n,source,destination);
    count++;
    // move n-1 discs from B to C using A as temporary
    TOH(n-1,temp,source,destination);
}
int main()
{
    int n;
    printf("Enter the number of discs:");
    scanf("%d",&n);
    TOH(n,'A','B','C');
    printf("Total number of disc moves = %d\n",count);
    getch();
    return 0;
}
```

```
Enter the number of discs:3
Move Disc 1 from A to C
Move Disc 2 from A to B
Move Disc 1 from C to B
Move Disc 3 from A to C
Move Disc 1 from B to A
Move Disc 2 from B to C
Move Disc 1 from A to C
Total number of disc moves = 7
```