Recursion

All code examples of functions we have seen so far have involved one function f() calling a different function g(). However, a function f() may also call itself. Such a function is referred to as a *recursive* function.

Recursion is a useful programming technique. Many examples for recursion could be given. We consider two classic examples of recursion; namely computing the factorials and the Towers of Hanoi problem.

To illustrate the basic idea behind recursion, we first consider the problem of calculating the factorials.

The factorial function

The factorial n! of an integer n >= 0 is defined as:

$$n! = \begin{cases} 1 & if \;\; n \;\; is \;\; 0 \\ 1 \times 2 \times \ldots \times n & if \;\; n > 0 \end{cases}$$

For example, 4! = 1 × 2 × 3 × 4 = 24 and 5! = 1 × 2 × 3 × 4 × 5 = 120.

The value of 5! can be simply computed by multiplying the value of 4! by the integer 5 as 5! = 5 × 4! = 5 × 24 = 120.

Thus, to calculate $n!$ for any positive integer n, we need only know the value of 0! and the relation between one factorial and the next, which is $n! = n \times (n - 1)!$.

Recursive function definition

We can use recursion to solve a problem when the problem can be repeatedly reduced to smaller and smaller versions of itself, until finally a reduced version is reached that is small enough to be solved directly.

The general form of recursive function definition consists of two things.

1. A *base case* (also called the *terminating case*).
The value of the function is calculated with no recursive calls.

2. A *recursive step*. This step involves one or more recursive calls to the function.
These recursive calls involve smaller versions of the same problem.

We can rewrite the definition of a factorial using this recursive approach as follows.

$$n! = \begin{cases} 1 & if \;\; n \;\; is \;\; 0 \\ n \times (n-1)! & if \;\; n > 0 \end{cases}$$

The following factorial function implements this definition to compute the factorial of any integer n >= 0.

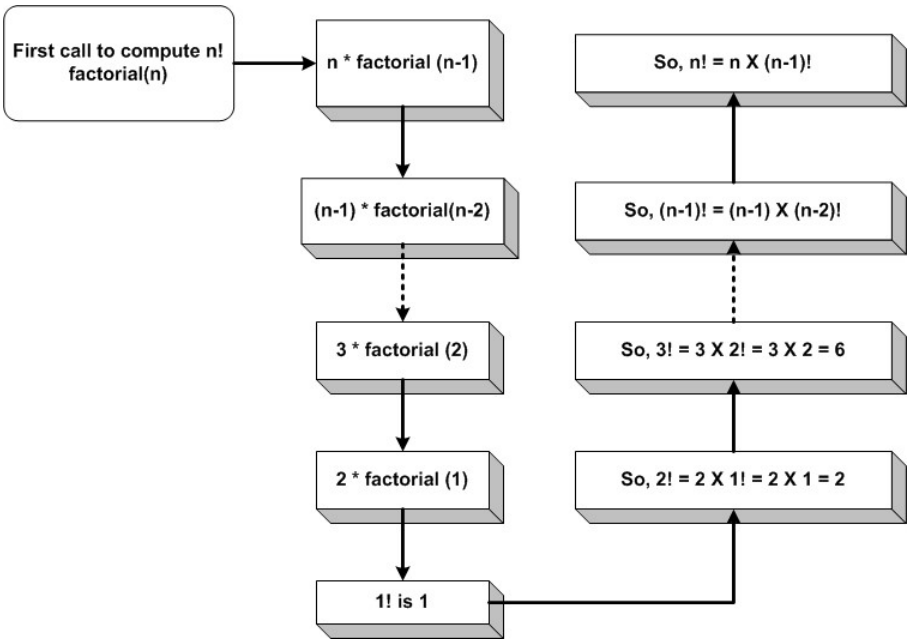
```
function factorial(n: number) {
   if ( n == 0 || n == 1 )
      return 1; // base case
   else
      return n * factorial(n-1); // recursive step
}
```

We have to keep the following two things in mind when we trace a recursive function call.

- First, no recursive call can be completed until a recursive call for a terminating (base) case has been completed.

- Second, once a terminating case call has been reached, then earlier recursive calls are completed in the reversed order in which they were called.

Our factorial function computes n! recursively as depicted below.



Trace of factorial(4) call

When the function call factorial(4) is made to compute 4!, four calls are made recursively to the factorial function.

The first call is factorial(4). This call will not be completed immediately because factorial(4) calls factorial(3), which calls factorial(2), which calls factorial(1).

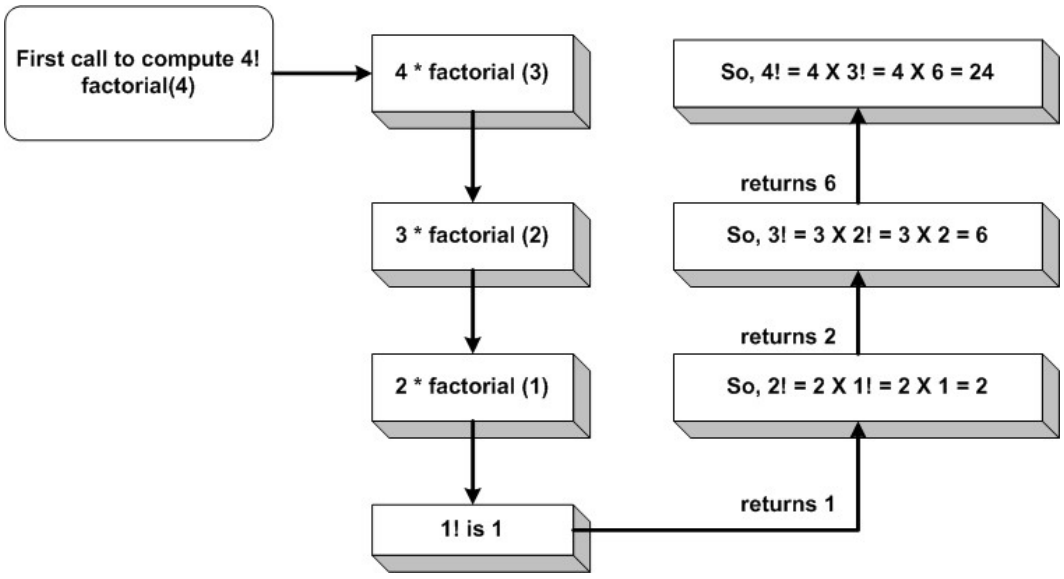
- The first call that gets completed is factorial(1) that returns 1 to the place from which factorial(1) was called.

- Next the call factorial(2) will be completed, and the value 2 is returned to the place from which factorial(2) was called.

- Then the call factorial(3) will be completed, and the value 6 is returned to the place from which factorial(3) was called.

- Finally, the original call of factorial(4) will be completed which returns the value 24.

The following flow chart illustrates the sequence of recursive calls for factorial(4).



Recursion versus iteration

Any code that can be accomplished using recursion can also be done in some other way without using recursion. For example, the factorial function can be written nonrecursively as follows.

```
int factorial(int n)
{
   int fact = 1;
   for (int i = 1; i <= n; i++)
      fact = fact * i;
   return fact;
} // end function factorial
```

The nonrecursive version of a function typically uses a loop construct in place of recursion. Therefore, the nonrecursive version is referred to as an iterative version.

- A recursive version of a function can sometimes be much simpler than an iterative version.

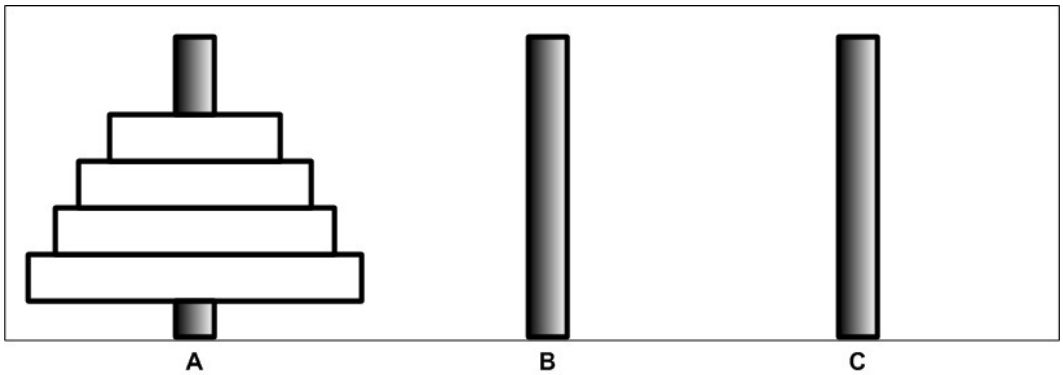
- A recursively written function will usually run slower and use more storage than an equivalent iterative version, because the computer must keep track of all the recursive calls.

- However, using recursion can sometimes make our job as a programmer simpler and can produce code that is easier to understand.

A problem that is harder to solve iteratively

The *Towers of Hanoi* is a well-known problem that can be easily solved using recursion; but very challenging to solve iteratively.

We are given three pegs, A, B, and C, and n disks. These n disks are of different sizes. Initially, peg A contains all the disks in order of size (with the smallest on top) as shown below.



The object is to move the disks from peg A to peg C obeying the following rules:

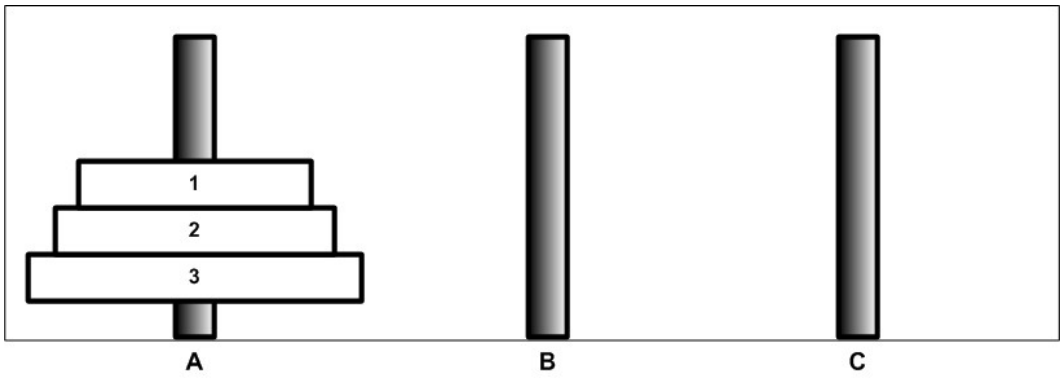
1. When a disk is moved, it must be placed on one of the three pegs.

2. Only one disk may be moved at a time, and it must be the top disk from a peg.

3. A larger disk can not be placed on the top of a smaller disk.

Solution to Towers of Hanoi for n = 3

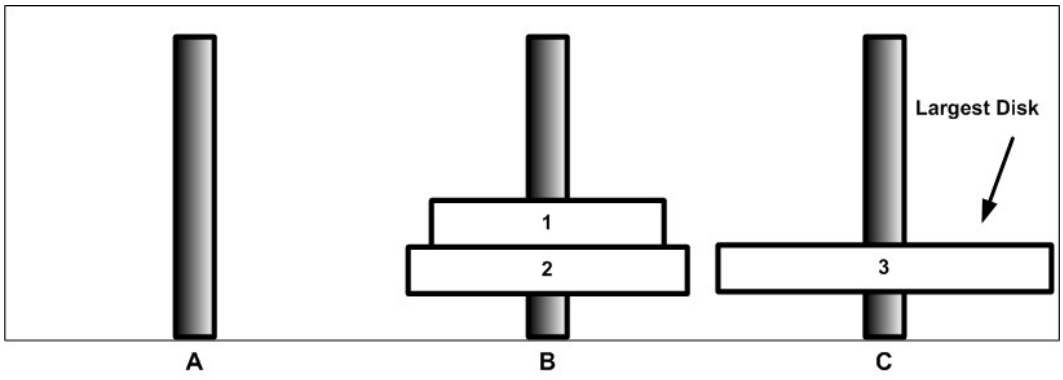
Consider the Towers of Hanoi problem involving 3 disks.



The shortest solution to the problem requires 7 moves. The first four moves are as follows

1. Move the first disk from peg A to peg C.
2. Move the second disk from peg A to peg B.
3. Move the first disk from peg C to peg B.
4. Move the third disk from peg A to peg C.

The following figure depicts the situation after the first 4 moves.



You determine the remaining three moves.

A recursive solution to the Towers of Hanoi problem

A recursive solution to the problem can be described as follows.

If n = 1.
Simply move the disk from peg A to C.

If n = 2.
Move the first disk from peg A to B.
Then move the second disk from A to C. Then move the first disk from B to C.

If n = 3.
Use the technique already established in step 2 to move the first two disks from peg A to B using C as an intermediate peg.

Then move the third disk from A to C.

Then use the technique in step 2 to move the first two disks from B to C using A as an intermediate peg.

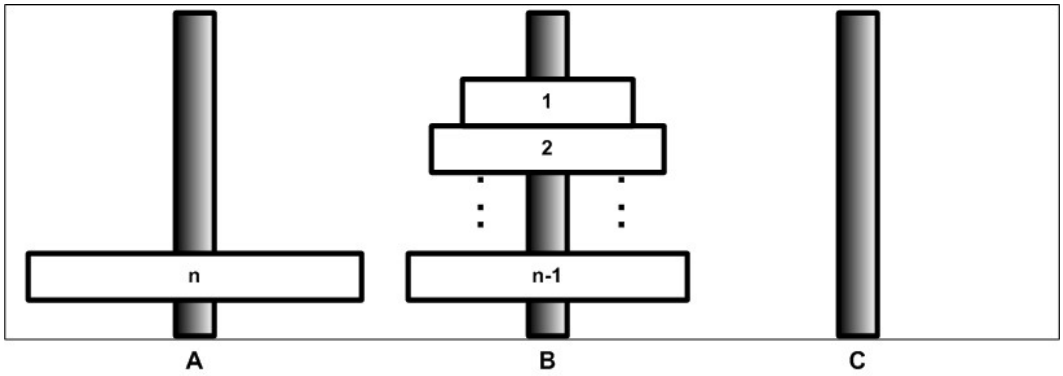
...

For general n.
Step 1: Use the technique in the previous step to move n-1 disks from A to B using C as an intermediate peg.

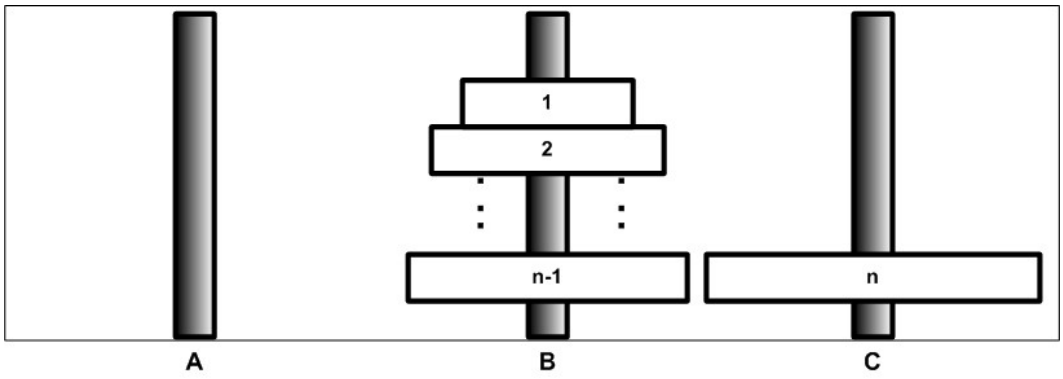
Step 2: Next move the largest (the only) disk from A to C.

Step 3: Then use the technique in the previous step to move n-1 disks from B to C using A as an intermediate peg.

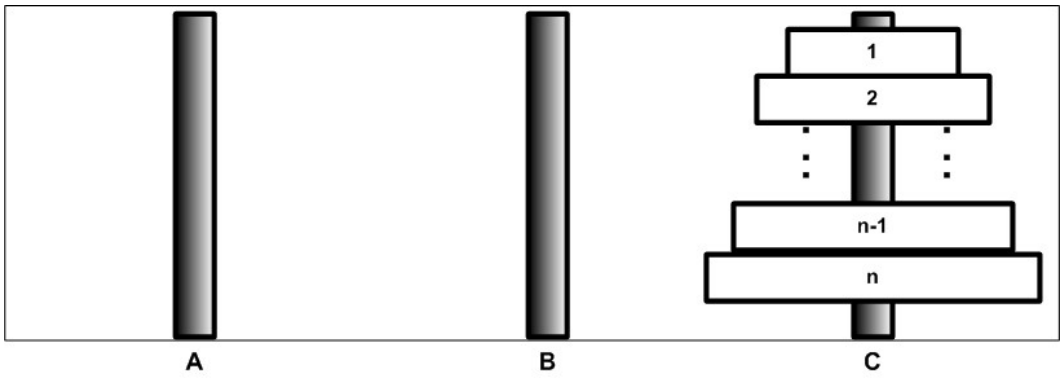
After *Step 1* the placement of disks on the three pegs will be as follows.



After *Step 2* the placement of disks on the three pegs will be as follows.



After *Step 3* the placement of disks on the three pegs will be as follows.



Notice that this procedure for solving the Towers of Hanoi problem describes itself in terms of a smaller version of itself. That is, the procedure describes how to solve the problem for n disks in terms of a solution for n-1 disks.

In general, any problem that admits a recursive solution is approached in this manner.

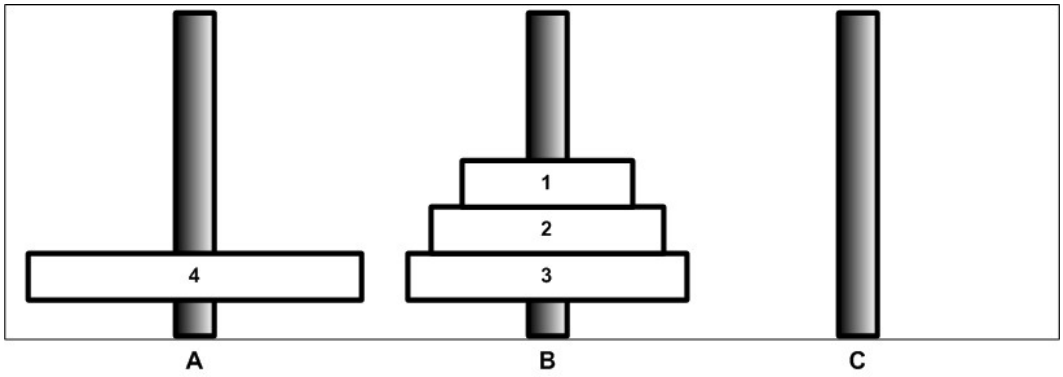
The following program solves the Towers of Hanoi problem using the recursive function moveHanoi.

```
import * as readline from 'readline';

function moveHanoi(n: number, source: string, dest: string,
inter: string) {
    if (n == 1) // terminating case
        console.log("Move disk from ", source, " to ", dest);
    else {
        moveHanoi(n-1, source, inter, dest);
        // REFERENCE POINT 1
        // At this point implementation of Step 1 is complete.
        console.log("Move disk from ", source, " to ", dest);
        // REFERENCE POINT 2
        // At this point implementation of Step 2 is complete.
        moveHanoi(n-1, inter, dest, source);
        // REFERENCE POINT 3
        // At this point implementation of Step 3 is complete.
    }
}

console.log("This program solves the Towers of Hanoi
problem");
let rl = readline.createInterface(process.stdin,
process.stdout);
rl.question("Enter the number of disks: ", numberOfRings => {
    moveHanoi(parseInt(numberOfRings), 'A', 'C', 'B');
    rl.close();
});
```

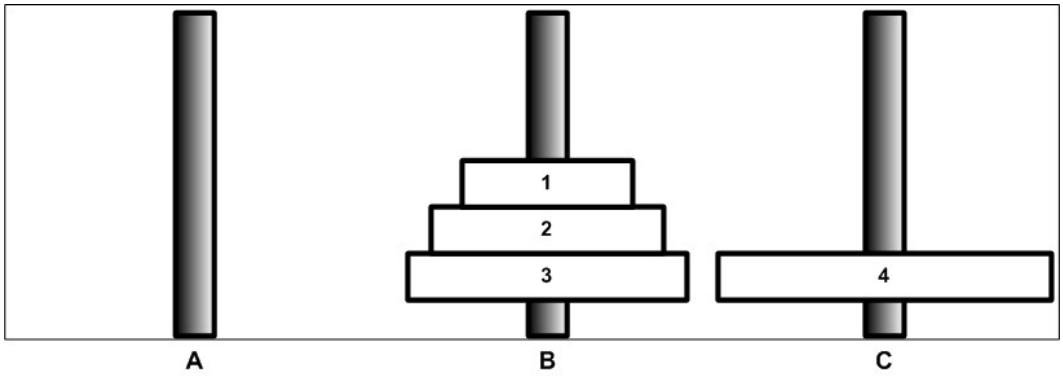
The output of a sample run of the program is as follows.

```
This program solves the Towers of Hanoi problem
Enter the number of disks: 4 ←
Move disk from A to B Move ring from A to C Move disk from B to C
Move disk from A to B Move disk from C to A Move disk from C to B
Move disk from A to B
– The call – moveHanoi(n-1, source, inter, dest) is completed
– REFERENCE POINT 1 Move disk from A to C
– REFERENCE POINT 2
Move disk from B to C
Move disk from B to A Move disk from C to A Move disk from B to C
Move disk from A to B Move disk from A to C
Move disk from B to C
– The call – moveHanoi(n-1, inter, dest, source) is completed
– REFERENCE POINT 3
```

After *Reference Point 1* the placement of disks on the three pegs will be as follows.



After *Reference Point 2* the placement of disks on the three pegs will be as follows.



After *Reference Point 3* the placement of disks on the three pegs will be as follows.