Recursividad

2025

Hugo Araya Carrasco

Se dice que algo es recursivo si se define en función de sí mismo o a sí mismo.

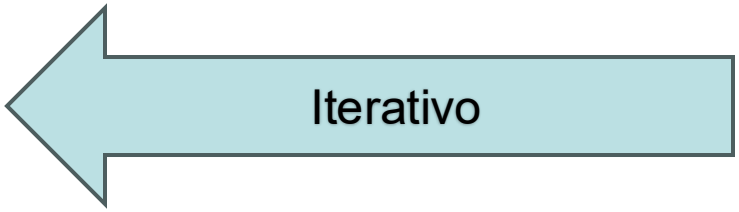
El caso es que las definiciones recursivas aparecen con frecuencia en matemáticas, e incluso en la vida real.

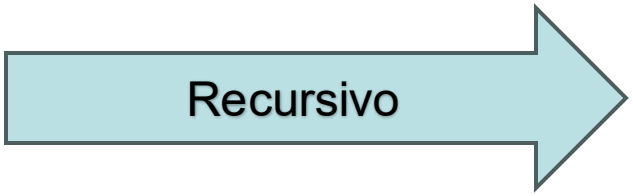
En matemáticas, tenemos múltiples definiciones recursivas:
- Números naturales:
  (1) 1 es número natural.
  (2) el siguiente número de un número natural es un número natural

El factorial: n!, de un número natural (incluido el 0):

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n \times (n-1)!, & \text{si } n > 0 \end{cases}$$

```
int factorial(int n){
    int i = 1, f = 1;
    while (i <= n){
        f = f * i;
        i++;
    }
    return f;
}
```






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

Asimismo, puede definirse un programa en términos recursivos, como una serie de pasos básicos, o **paso base** (también conocido como condición de parada), y un **paso recursivo**, donde vuelve a llamarse al programa.

En un computador, esta serie de pasos recursivos debe ser finita, terminando con un paso base.

Es decir, a cada paso recursivo se reduce el número de pasos que hay que dar para terminar, llegando un momento en el que no se verifica la condición de paso a la recursividad.

Ni el paso base ni el paso recursivo son necesariamente únicos.

Por otra parte, la recursividad también puede ser indirecta, si tenemos un procedimiento P que llama a otro Q y éste a su vez llama a P. También en estos casos debe haber una condición de parada.

Existen ciertas estructuras cuya definición es recursiva, tales como los árboles, y los algoritmos que utilizan árboles suelen ser en general recursivos.

A continuación, se expone un ejemplo de programa que utiliza recursión indirecta, y nos dice si un número es par o impar.

Al igual que el programa anterior, hay otro método mucho más sencillo de determinar si un número es par o impar, basta con determinar el resto de la división entre dos. x

Por ejemplo: si hacemos par(2) devuelve 1 (cierto). Si hacemos impar(4) devuelve 0 (falso).

```
 /* declaración de funciones, para evitar errores */
int par(int n);
int impar(int n);

int par(int n){
       if (n == 0)
              return 1;
       return impar(n-1);
}
int impar(int n){
       if (n == 0)
              return 0;
       return par(n-1);
}
```

¿Qué pasa si se hace una llamada recursiva que no termina?

Cada llamada recursiva almacena los parámetros que se pasaron al procedimiento, y otras variables necesarias para el correcto funcionamiento del programa.

Por lo tanto si se produce una llamada recursiva infinita, esto es, que no termina nunca, llega un momento en que no quedará memoria para almacenar más datos, y en ese momento se abortará la ejecución del programa.

Para probar esto se puede intentar hacer esta llamada en el programa factorial definido anteriormente:
factorial(-1);

¿Cuándo utilizar la recursión?

Para empezar, algunos lenguajes de programación no admiten el uso de recursividad, como por ejemplo el ensamblador o el FORTRAN. Es obvio que en ese caso se requerirá una solución no recursiva (iterativa).

Tampoco se debe utilizar cuando la solución iterativa sea clara a simple vista. Sin embargo, en otros casos, obtener una solución iterativa es mucho más complicado que una solución recursiva, y es entonces cuando se puede plantear la duda de si merece la pena transformar la solución recursiva en otra iterativa.

Posteriormente se explicará como eliminar la recursión, y se basa en almacenar en una pila los valores de las variables locales que haya para un procedimiento en cada llamada recursiva. Esto reduce la claridad del programa. Aún así, hay que considerar que el compilador transformará la solución recursiva en una iterativa, utilizando una pila, para cuando compile al código del computador.

Ejercicio

La famosa sucesión de Fibonacci puede definirse en términos de recurrencia de la siguiente manera:

$$Fib(n) = \begin{cases} 1, & \text{si } n = 1, n = 2 \\ Fib(n-1) + Fib(n-2), & \text{si } n > 2 \end{cases}$$

¿Cuantas llamadas recursivas se producen para Fib(6)?.

Codificar un programa que calcule Fib(n) de forma recursiva.

Nota: no utilizar estructuras de datos, puesto que no queremos almacenar los números de Fibonacci anteriores a n; sí se permiten variables auxiliares.

Dados dos números a (número entero) y b (número natural mayor o igual que cero) determinar a^b.

```
int potencia(int a, int b){
        if (b == 0)
              return 1;
       else
              return a * potencia(a, b-1);
}
```

La condición de parada se cumple cuando el exponente es cero. Por ejemplo, la evaluación de potencia(-2, 3) es:

```
potencia(-2, 3) ->
(-2) · potencia(-2, 2) ->
(-2) · (-2) · potencia(-2, 1) ->
(-2) · (-2) · (-2) · potencia(-2, 0) ->
(-2) · (-2) · (-2) · 1
```

Dado un array constituido de números enteros y que contiene N elementos siendo N >= 1, devolver la suma de todos los elementos.

```
int sumarray(int num[ ], int pos, int N){
   if (pos == N-1)
      return num[pos];
   else
      return num[pos] + sumarray(num, pos+1, N);
}...

int num[5] = {2,0,-1,1,3};
int N = 5;
printf("%d\n",sumarray(num, 0, N));
```

Dado un array constituido de números enteros, devolver la suma de todos los elementos. En este caso se desconoce el número de elementos. En cualquier caso se garantiza que el último elemento del array es -1, número que no aparecerá en ninguna otra posición.

```
int sumarray(int num[], int pos){
  if (num[pos] == -1)
       return 0;
  else
      return num[pos] + sumarray(num, pos+1);
}

...
int num[5] = {2,4,1,-3,1};
printf("%d\n",sumarray(num, 0));
```

Dado un array constituido de números enteros y que contiene N elementos siendo N >= 1, devolver el elemento mayor.

```
int mayor(int num[], int pos){
        int aux;
        if (pos == 0)
                return num[pos];
        else {
                aux = mayor(num, pos-1);
                if (num[pos] > aux)
                        return num[pos];
                else
                        return aux;
                }
}
...
int num[5] = {2,4,1,-3,-1};
int N = 5;
printf("%d\n", mayor(num, 4));
```

1) Dado un array constituido de números enteros y que contiene N elementos siendo N >= 1, devolver el elemento mayor. En este caso escribir un procedimiento, es decir, que el elemento mayor devuelto sea una variable que se pasa por referencia.

2) La función de Ackermann, siendo n y m números naturales, se define de la siguiente manera:

```
Ackermann(0, n) = n + 1
Ackermann(m, 0) = A(m-1, 1)
Ackermann(m, n) = A(m-1, A(m, n-1))
```

3) Dado un array constituido de números enteros y que contiene N elementos siendo N >= 1, escribir una función que devuelva la suma de todos los elementos mayores que el último elemento del array.

4) Dado un array constituido de números enteros y que contiene N elementos siendo N >= 1, escribir una función que devuelva cierto si la suma de la primera mitad de los enteros del array es igual a la suma de la segunda mitad de los enteros del array.

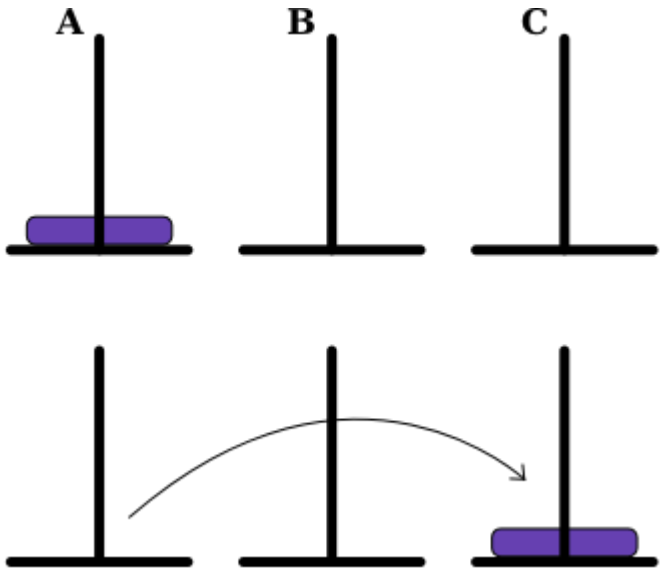
- **El Problema**
- Fue presentado en 1883 por el matemático francés Edouard Lucas (1842 - 1891).

- El problema consistía en lo siguiente: hay que mover una torre, compuesta de discos de diferentes tamaños, de una aguja a otra. Sólo hay dos reglas:
- Sólo se puede mover un disco a la vez
- No está permitido mover un disco encima de otro más pequeño

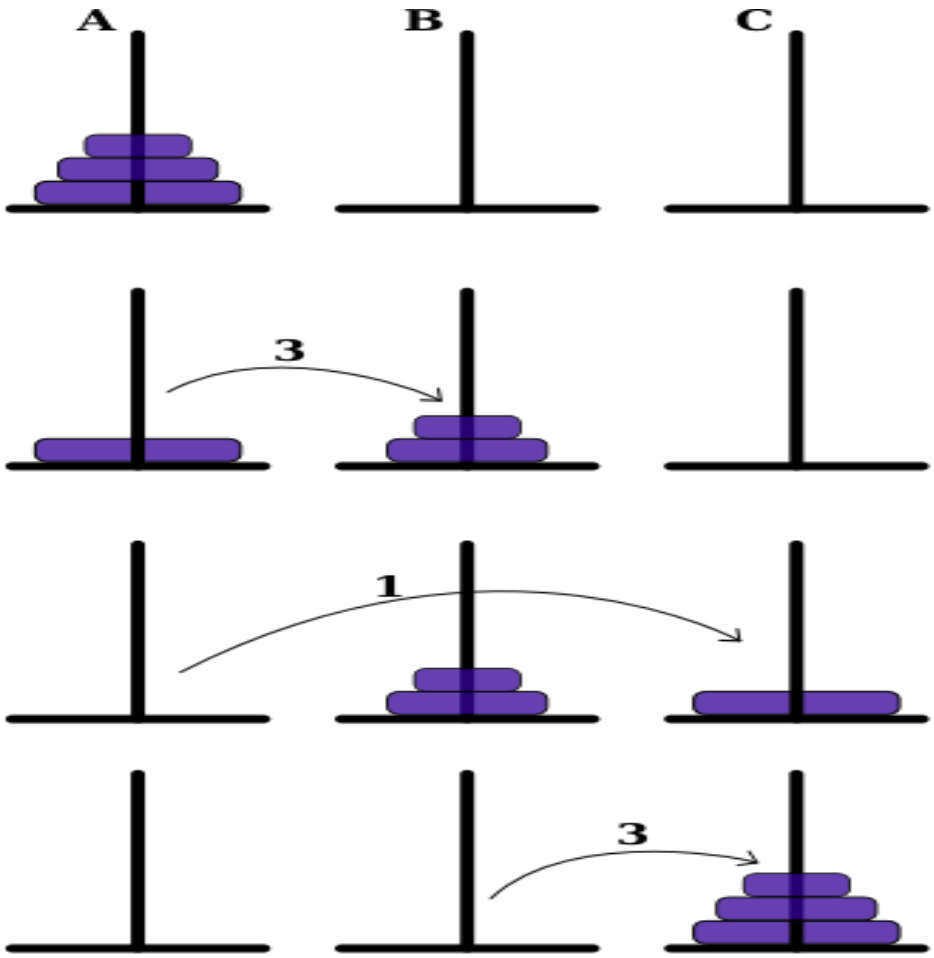
- Según Lucas el juego había sido rescatado por el Profesor N. Claus de Siam. La leyenda contaba que en el templo de Benarés, Dios colocó durante la creación 64 anillos en una aguja. Desde entonces los bramanes, durante incontables generaciones, han estado moviendo los discos de acuerdo con las reglas de arriba. Cuando hayan conseguido mover todos los discos, el templo se derrumbará y el mundo se desvanecerá.

La solución

Para resolver esto, empezamos por supuesto con algo más fácil y lo más fácil que hay es una torre compuesta por un único anillo. ¿Cuántos movimientos necesitamos? Evidentemente uno solo.



¿Y con 3 discos?



```
void hanoi(int n,int com, int aux, int fin){

   if(n==1)
       printf("%c -> %c",com,fin);
   else{
       hanoi(n-1,com,fin,aux);
       printf("\n%c -> %c\n",com,fin);
       hanoi(n-1,aux,com,fin);
   }
}
```