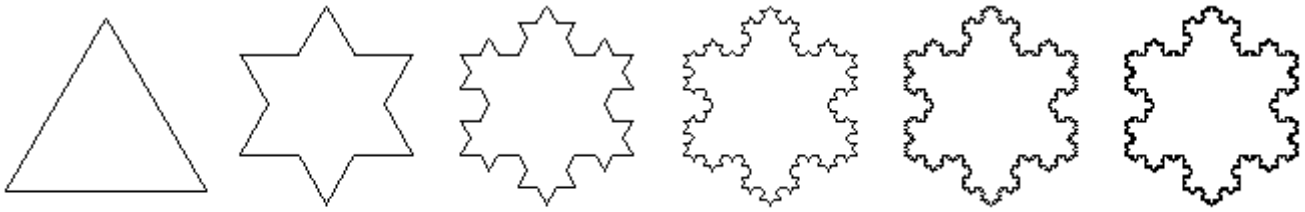
Recursividad

1. Definición

La recursividad, también llamada recursión o recurrencia, consiste en especificar un proceso basado en su propia definición, es decir, la definición de un objeto es recursiva si en ella se hace referencia al objeto definido. La recursividad se puede ver también como el proceso de repetir elementos de una manera autosimilar (la autosimilaridad es la propiedad de un objeto en el que el todo es exacta o aproximadamente similar a una parte de sí mismo).



En el campo de la algoritmia, una acción será recursiva si hace referencia (directa o indirectamente) a sí misma dentro de su código; es decir, si se llama a sí misma (Rabasa y Santamaría, 2004).

2. Estructura o Patrón de un Proceso Recursivo

Un método recursivo debe tener dos partes:

- Caso Base: Es una solución trivial para alguno de sus argumentos. Esta parte constituye la terminación o condición de parada en la que se dejan de hacer llamadas recursivas.
- Paso Recursivo: Está constituido por una relación de recurrencia mediante la cual se definen los valores sucesivos. En esta parte se replantea el problema original mediante una versión más simple o más pequeña del mismo. Debido a que este subproblema se parece al problema original, se realiza una llamada recursiva para que trabaje con este.

3. Iteración frente a Recursión

Característica	Iteración	Recursión
Estructura de control	Utiliza una estructura repetitiva.	Utiliza una estructura de selección.
Repetición	Utiliza explícitamente una estructura repetitiva.	Consigue la repetición mediante llamadas recursivas a funciones.
Condición de salida	Finaliza cuando la condición del bucle no se cumple.	Finaliza cuando se reconoce un caso base o la condición de salida se alcanza.
Eficiencia	Es eficiente en cuanto a la ocupación de memoria de algunos problemas pero ineficientes en algunos métodos de ordenación.	Es ineficiente en ocupación de espacio de memoria y de tiempo de ejecución pero eficiente en algunos métodos de ordenación.

4. Clasificación

Puede haber varias clasificaciones posibles, según el criterio elegido. La siguiente clasificación de la recursividad atiende al número de llamadas recursivas realizadas y el lugar en el cual estas se encuentran:

4.1. Recursividad lineal

Es aquella en la cual el cuerpo de la función contiene una sola llamada recursiva y si hay más de una, no están en la misma rama de ejecución.

Recursividad Lineal Final: Se da cuando después de realizar la llamada recursiva no existen operaciones pendientes sobre los datos, de esta manera no hace falta representar la pila de variables de cada llamada y el método recursivo podría ser transformado en uno iterativo de manera sencilla.

Ejemplo: El M.C.D. de *a* y *b*.

```
func MCD(entero a, b): entero
inicio
   si(b = 0) entonces
         retornar(a)
   sino
         retornar(MCD(b, a mod b))
   fsi
ffunc
```

Recursividad Lineal no Final: Se da cuando después de realizar la llamada recursiva existen operaciones pendientes sobre los datos para completar el proceso.

Ejemplo: El factorial de un número.

```
func factorial(entero n): entero
inicio
   si(n = 0 ∨ n = 1) entonces
         retornar(1)
   sino
         retornar(n * factorial(n-1))
   fsi
ffunc
```

4.2. Recursividad no Lineal

Es un tipo de recursividad en la cual el cuerpo de la función contiene varias llamadas explícitas a sí misma en la misma rama de ejecución. Cuando se tiene dos llamadas se habla de recursividad binaria.

Recursividad en Cascada: Se da cuando en cada rama de ejecución existe más de una llamada recursiva.

Ejemplo: Fibonacci.

```
func fibonacci(entero n): entero
Inicio
   si(n = 0 ∨ n = 1) entonces
         retornar(n)
   sino
         retornar(fibonacci(n-1)*fibonacci(n-2))
   fsi
fin
```

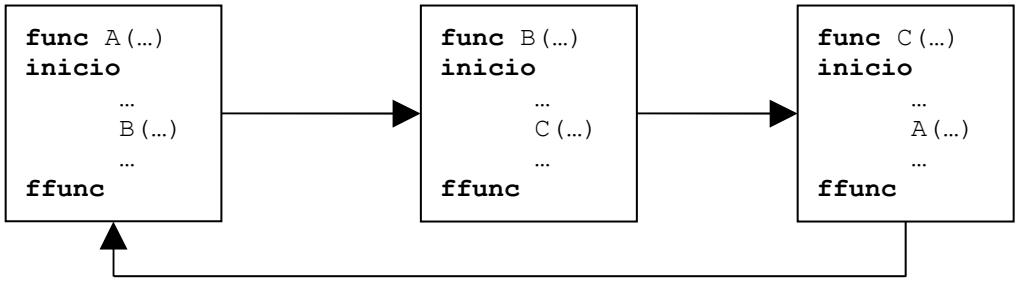
Recursividad Anidada: En este tipo de recursividad la función recibe como uno de sus parámetros una llamada recursiva.

Ejemplo: Función de Ackermann.

```
func ackermann(entero m, n): entero
inicio
   si(m = 0) entonces
         retornar(n+1)
   sino
         si(m > 0 ∧ n = 0) entonces
               ackermann(m-1, 1)
         sino
               si(m > 0 ∧ n > 0) entonces
                     retornar(ackermann(m-1, ackermann(m, n-1)))
               fsi
         fsi
   fsi
ffunc
```

4.3. Recursividad Mutua

En la recursividad mutua, también conocida como recursividad indirecta, la recursividad se presenta cuando varias acciones nominadas se llaman entre sí. Cuando una acción A, puede generar una secuencia de llamadas a otras acciones, que termina con una invocación a la acción original, se dice que A es indirectamente recursiva:



Ejemplo: Determinar si un número es par o impar.

```
func impar(entero n): entero
inicio
   si(n = 0) entonces
         retornar(falso)
   sino
         retornar(par(n-1))
   fsi
ffunc

func par(entero n): entero
inicio
   si(n = 0) entonces
         retornar(verdadero)
   sino
         retornar(impar(n-1))
   fsi
ffunc
```

5. Implementación de la Recursividad

La implementación de la recursividad se basa en el uso de la pila de memoria del programa.

Cuando una operación (incluyendo la principal) realiza una llamada a otra operación, el sistema debe saber donde reanudar la ejecución del programa una vez que la operación llamada finaliza, además de cierta información de estado de la operación que realiza la llamada, requerida para su ejecución exitosa, por ejemplo, los valores de sus variables locales y los valores de los parámetros actuales de la operación que invoca (si los tiene). Esta información se guarda en un bloque contiguo de memoria llamado registro de activación.

Un registro de activación se guarda en la pila de memoria del programa, y existe mientras se este ejecutando la operación que lo posea. Generalmente contiene la siguiente información (Aho, Sethi & Ullman, 1998):

Valor Devuelto
Parámetros Actuales
Enlace de Control
Enlace de Acceso
Estado de la Máquina
Datos Locales
Valores Temporales

a) Valor Devuelto: Este campo es utilizado por la operación autora de una llamada a una función, para guardar el valor de retorno obtenido luego de su ejecución.

b) Parámetros Actuales: Este campo es utilizado por la operación autora de la llamada para guardar los valores de los parámetros actuales de la operación que la recibe.

c) Enlace de Control: Apunta al registro de activación de la operación autora de la llamada.

d) Enlace de Acceso: Hace referencia a los datos no locales guardados en otros registros de activación.

e) Estado de la máquina: Contiene información del estado de la máquina justo antes de hacer la llamada: Contador del programa, registros de la máquina, entre otros.

f) Datos Locales: Guarda las variables locales de la operación.

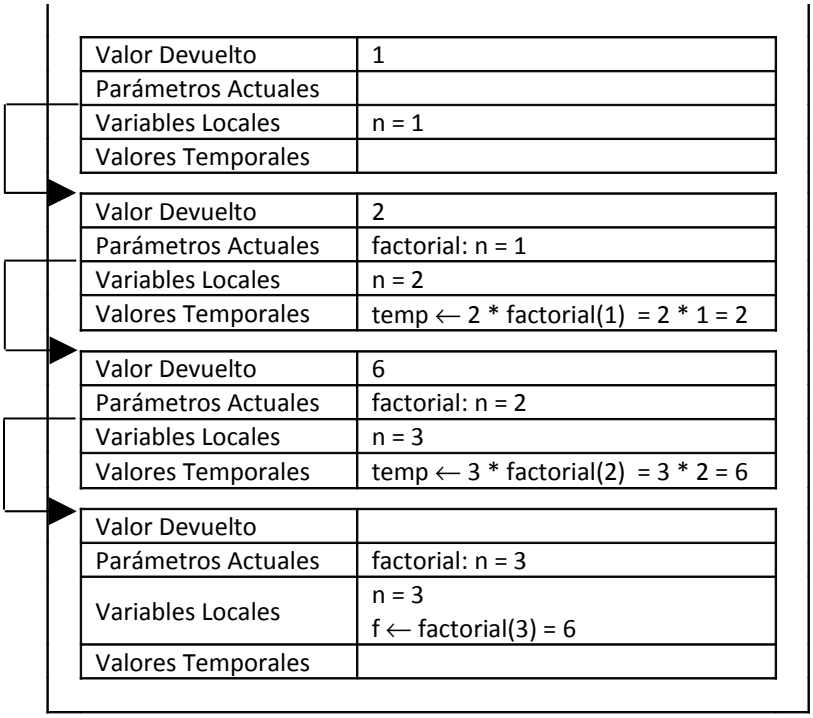
g) Valores temporales: Es un espacio de memoria reservado para guardar los valores temporales que pueden surgir durante la ejecución de la operación, por ejemplo, en la evaluación de una expresión.

De todos los campos del registro de activación, se hará énfasis en tres: Valor devuelto, parámetros actuales y datos locales.

Ejemplo:

```
01 algoritmo calculo_factorial
02 func factorial(entero n): entero
03 inicio
04       si(n = 0 ∨ n = 1) entonces
05             retornar(1)
06       sino
07             retornar(n * factorial(n-1))
08       fsi
09 ffunc
10 inicio
11       var
12             entero n, f
13       escribir("Calculo del factorial de un número")
14       escribir("Introduzca un número entero positivo: ")
15       leer(n)
16       f ← factorial(n)
17       escribir("El factorial es ", f)
18 fin
```

Pila de Memoria



6. Referencias

Aho A., Sethi R. y Ullman J. (1998). Compiladores: Principios, Técnicas y Herramientas. Naucalpán de Juárez , México: Addison Wesley Longman de México S.A. de C.V.

Deitel H. M. y Deitel P. J. (1995). Cómo Programar en C/C++ (Segunda ed.). Naucalpán de Juárez, México: Prentice Hall Hispanoamericana S.A.

Martínez, A., Rosquete, D. (2009). NASPI: Una Notación Estándar para Programación Imperativa. Télématique.

Rabasa D., A. y Santamaría A., L. (2004). Metodología de Programación. Principios y Aplicaciones. Alicante, España: Editorial Club Universitario.