# Recursividad

Un objeto es recursivo cuando se define en función de sí mismo, es decir, interviene en su propia definición; La recursividad es la propiedad mediante la cual un subprograma o rutina puede llamarse a sí mismo.

Utilizando la recursividad, la resolución de un problema se reduce a uno esencialmente igual, pero algo menos complejo.

Características que deben cumplir los problemas recursivos:

* La recursividad debe terminar alguna vez: caso base
* Cada nueva formulación estamos más cerca del caso final (o base).

La recursividad es muy útil cuando no se conoce la solución a un problema, pero éste puede ser dividido para simplificarlo. Se trata de un concepto directamente relacionado con las matemáticas; por ejemplo, el cero es un número natural y el sucesor de cada número natural es otro número natural.

# Procedimientos recursivos

Un Procedimiento recursivo es aquel que se llama así mismo, solo que no regresa valor.

Cada método (función o procedimiento), tiene ciertas reglas, las cuales se mencionan a continuación:

* La Función Recursiva Debe tener ciertos argumentos llamados valores base para que esta ya no se refiera a sí misma.
* El Procedimiento Recursivo es donde Cada vez que la función se refiera a sí misma debe estar más cerca de los valores base.

Propiedades de procedimientos recursivos

1. Debe existir criterio base para que este se llame a sí mismo.
2. Cada vez que el procedimiento se llame a si mismo debe estar más cerca del criterio base.

## El Método De Las Tres Preguntas

Se usa para verificar si hay dentro de un programa funciones recursivas, se debe responder a 3 preguntas.

1. La pregunta caso bas:  
   ¿Hay salida NO recursiva del procedimiento o función y la rutina funciona correctamente para este caso base?
2. La pregunta llamador más pequeño  
   ¿Cada llamada al procedimiento o función se refiere a un caso más pequeño del problema original?
3. La pregunta caso general  
   Suponiendo que las llamadas recursivas funcionan correctamente ¿funciona correctamente todo el procedimiento o función?

## Recursividad

Se puede utilizar el siguiente método para escribir cualquier rutina recursiva.

1. Primero obtener una función exacta del problema a resolver.
2. A continuación, determinar el tamaño del problema completo que hay que resolver, este tamaño determina los valores de los parámetros en la llamada inicial al procedimiento o función.
3. Resolver el caso base en el que el problema puede expresarse no recursivamente, esto asegura una respuesta afirmativa a la pregunta base.
4. Por último, resolver el caso general correctamente en términos de un caso más pequeño del mismo problema, es decir una respuesta afirmativa a las preguntas 2 y 3 del método de las 3 preguntas.

## Ejemplos de casos de recursividad

La función potencia , vista en unidades anteriores, realizaba n iteraciones para poder obtener el valor de. Sin embargo, es posible optimizarla teniendo en cuenta que:

* si  es par.
* si  es impar.

Antes de programar cualquier función recursiva es necesario decidir cuál será el caso base y cuál el caso recursivo. Para esta función, tomaremos  como el caso base, en el que devolveremos ; y el caso recursivo tendrá dos partes, correspondientes a los dos posibles grupos de valores de .

def potencia(b,n):

""" Precondición: n debe ser mayor o igual que cero.

Devuelve: b\^n. """

# Caso base

if n <= 0:

return 1

# n par

if n % 2 == 0:

pot = potencia(b, n/2)

return pot \* pot

# n impar

else:

pot = potencia(b, (n-1)/2)

return pot \* pot \* b

El uso de la variable  en este caso no es optativo, ya que es una de las ventajas principales de esta implementación: se aprovecha el resultado calculado en lugar de tener que calcularlo dos veces. Vemos que este código funciona correctamente:

>>> potencia(2,10)

1024

>>> potencia(3,3)

27

>>> potencia(5,0)

1

El orden de las llamadas, haciendo un seguimiento simplificado de la función será:

potencia(2,10)

pot = potencia(2,5) # b → 2 n → 10

pot = potencia(2,2) # b → 2 n → 5

pot = potencia(2,1) # b → 2 n → 2

pot = potencia(2,0) # b → 2 n → 1

return 1 # b → 2 n → 0

return 1 \* 1 \* 2 # b → 2 n → 1 pot → 1

return 2 \* 2 # b → 2 n → 2 pot → 2

return 4 \* 4 \* 2 # b → 2 n → 5 pot → 4

return 32 \* 32 # b → 2 n → 10 pot → 32

Se puede ver, entonces, que para calcular  se realizaron  llamadas a potencia, mientras que en la implementación más sencilla se realizaban  iteraciones. Y esta optimización será cada vez más importante a medida que aumenta , por ejemplo, para  se realizarán 8 llamadas recursivas, para,  llamadas.

Para transformar este algoritmo recursivo en un algoritmo iterativo, es necesario simular la pila de llamadas a funciones mediante una pila que almacene los valores que sean necesarios. En este caso, lo que apilaremos será si el valor de  es par o no.

def potencia(b,n):

""" Precondición: n debe ser mayor o igual que cero.

Devuelve: b^n. """

pila = []

while n > 0:

if n % 2 == 0:

pila.append(True)

n /= 2

else:

pila.append(False)

n = (n-1)/2

pot = 1

while pila:

es\_par = pila.pop()

if es\_par:

pot = pot \* pot

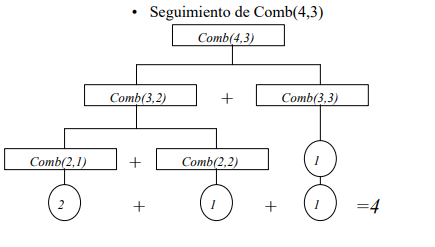
else:

pot = pot \* pot \* b

return pot

Imagen que contiene captura de pantalla

Descripción generada con confianza alta



# Referencias

1. Aaron M. Tenenbaum. (1981). Data Structures Using Pascal. United States of America: Prentice-Hall.
2. Bruno López Takeyas. (2012). Estructura dedatos orientada a objetos. México: Alfaomega.
3. Dr. Osvaldo Cairó. (2006). Estructura de datos. México: McGraw-Hill.
4. Robert L. Kruse. (1988). Estructura de Datos y Diseño de Programas. México: Prentice-Hall.
5. Luis Joyanes Aguilar. (1998). Estructura de datos: Algoritmos, Abstracción y Objetos. Madrid: McGraw-Hill.