RECURSIVIDAD

PROGRAMACIÓN 2024

EL PROBLEMA DE SER UN PROGRAMADOR

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.

Un ejemplo: basta con apuntar una cámara al monitor que muestra la imagen que muestra esa cámara.

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):
  - (1) si n = 0 entonces: 0! = 1
  - (2) si n > 0 entonces: $n! = n \cdot (n-1)!$

ITERATIVO

```
def factorial(n):
    i = 1
    f = 1
    while i <= n:
        f = f * i
        i = i+ 1
    return f

if __name__ == "__main__":
    numero = input("Numero: ")
    numero_f = factorial(numero)
    print numero_f
```

RECURSIVO

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

if __name__ == "__main__":
    numero = input("Numero: ")
    numero_f = factorial(numero)
    print numero_f
```

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).

```
def par(n):
    if n == 0:
        return 1
    else:
        return impar (n-1)

def impar(n):
    if n == 0:
        return 0
    else:
        return par(n-1)

if __name__ == "__main__":
    numero = input("Ingrese numero: ")
    respuesta = par(numero)
    if respuesta == 0:
        print "Falso"
    else:
        print "Verdadero"
```

¿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

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …`

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

(1) Fib(1) = 1 ; Fib(0) = 0
(2) Fib(n) = Fib(n-1) + Fib(n-2); si n >= 2

¿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.

```
def potencia(a, b):
    if b == 0:
        return 1
    else:
        return a * potencia(a, b-1)

if __name__ == "__main__":
    a = input("Base: ")
    b = input("Exponente: ")
    resultado = potencia(a, b)
    print resultado
```

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
```

Dada una lista de N números enteros, devolver la suma de todos los elementos (versión recursiva).

```
def suma_lista(lista, pos, N):
   if pos == N-1:
      return lista[pos]
   else:
      return lista[pos] + suma_lista(lista, pos+1, N);

if __name__ == "__main__":
   lista = [2,0,-1,1,3]
   N = len(lista)
   resultado = suma_lista(lista, 0, N)
   print "Suma es: ", resultado
```

Dada una lista de N números enteros, devolver el elemento mayor.

```
def mayor(lista, pos):
    if pos == 0:
       return lista[pos]
    else:
       aux = mayor(lista, pos-1)
       if lista[pos] > aux:
           return lista[pos]
       else:
           return aux

if __name__ == "__main__":
    lista = [2, 4, 1, -3, -1, 10]
    N = len(lista)
    resultado = mayor(lista, N-1)
    print "Resultado: ", resultado
```

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))
```

Escriba um programa recursivo que calcule el valor de la función para unos determinados valores de n y m.

Analice paso a paso la función e indique que problemas se pueden presentar.

El Clasico Problema de las Torres de Hanoi

- 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

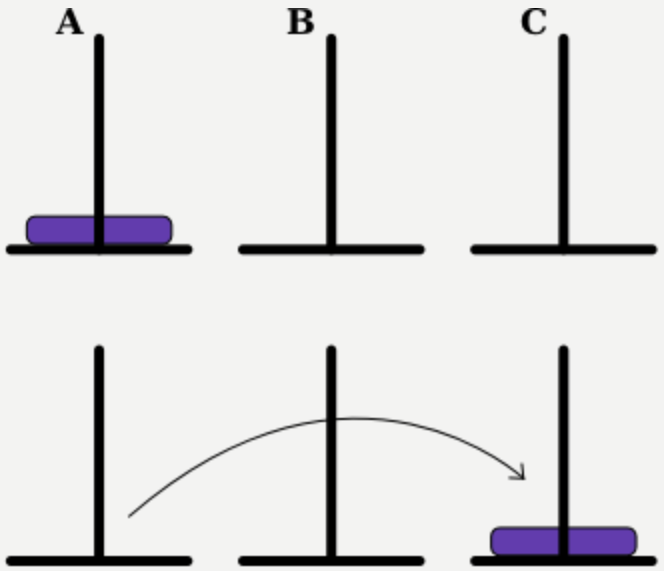
- Según Lucas el juego había sido rescatado por el Profesor N. Claus de Siam. La leyenda contaba que en el templo de Benarés, Dios colocó durante la creación 64 anillos en una aguja. Desde entonces los bramanes, durante incontables generaciones, han estado moviendo los discos de acuerdo con las reglas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

La solución

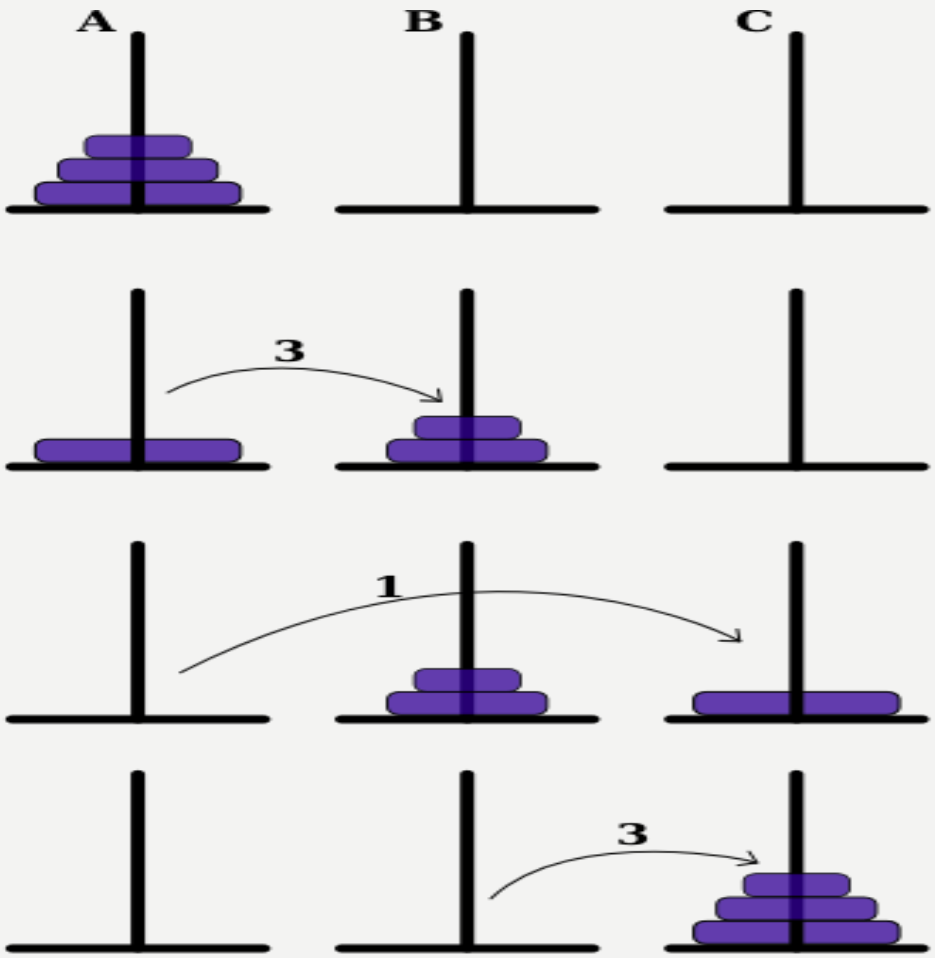
Para resolver esto, empezamos por supuesto con algo más fácil y lo más fácil que hay es una torre compuesta por un único anillo.

¿Cuántos movimientos necesitamos?

Evidentemente uno solo.



Y CON 3 DISCOS?



```
def hanoi(n, com, aux, fin):
    if n == 1:
        print "Mover desde ", com, " a ", fin
    else:
        hanoi(n-1, com, fin, aux)
        print "Mover desde ", com, " a ", fin
        hanoi(n-1, aux, com, fin)

if __name__=="__main__":
    n = input("Cantidad de discos: ")
    hanoi(n, 1, 2, 3)
```