Tema 10: Recursividad

Héctor Xavier Limón Riaño

May 31, 2024

Contents

1 Introducción

- La recursividad es una forma declarativa para definir procesos que se repiten

- Tienen definición formal en matemáticas (explicado más adelante)
- En este sentido es similar a los ciclos de control imperativo como `for` y `while`
- A diferencia de las sentencias de control tradicionales, la recursividad aplica a nivel de funciones
- Se habla entonces de funciones recursivas (o no recursivas que son las que se han trabajado hasta ahora)
- Cualquier cosa que se pueda hacer con `for` o `while` puede lograrse con funciones recursivas
- Pero no todo lo que se puede hacer con recursividad se puede hacer con `for` y `while` (o resulta demasiado complejo), lo que lo vuelve una forma de repetición más general (y poderosa)
- En términos simples una función recursiva es aquella que se define en términos de ella misma, esto es, en algún punto de su propio cuerpo se manda a llamar a si misma (también puede ser de forma indirecta como se menciona más adelante)
- A muchos programadores se les dificulta el uso de recursividad, incluso a programadores profesionales
- Esto se debe a su naturaleza declarativa, siendo que la programación se enseña normalmente de forma imperativa
- En términos generales puede verse a la recursividad como una técnica de programación que permite resolver problemas que de otra forma resultan muy difíciles o imposibles de resolver con las estructuras de control tradicionales (se verán ejemplos concretos de esto en el tema de árboles)

1.1 Programación imperativa

- Se refiere a darle a la computadora paso por paso las instrucciones que debe ejecutar
- Es trabajo del programador dar todos los detalles del control del programa
- Esto es, el programador debe definir `cómo` resolver el problema

1.2 Programación declarativa

- El programador se preocupa principalmente por definir el problema
- Esto es, el programador define `qué` es el problema
- Existe un mecanismo de resolución automático que recibe definiciones de problemas y genera soluciones
- Este mecanismo puede ser por ejemplo un motor de resolución lógica (usado en programación lógica)
- Todos los sistemas operativos tienen un motor implícito de resolución de problemas recursivos, éste funciona a través de la pila de memoria del proceso (se explica más adelante)

```
# versión imperativa
def fibonacci(numero):
    if numero < 3:
        return 1
    anterior = 1
    actual = 1
    for _ in range(numero - 2):
        aux = actual
        actual = anterior + actual
        anterior = aux

    return actual

print(fibonacci(10))
```

```
55
```

```
# versión declarativa

def fibonacci(numero):
    if numero < 3:
        return 1
    return fibonacci(numero - 2) + fibonacci(numero - 1)

print(fibonacci(10))
```

```
55
```

- Las soluciones declarativas suelen ser más cortas y elegantes que las imperativas, pero suelen resultar más abstractas y difíciles de entender (sobre todo si no estás acostumbrado)
- En un curso posterior (lenguajes y paradigmas de programación) se discute el tema de programación declarativa más a fondo

2 Manejo de memoria

2.1 Memoria RAM

- La memoria RAM es una memoria de acceso rapido (a diferencia del disco duro)
- Basicamente se utiliza para tener acceso a los datos de los programas que se encuentran en ejecucion (llamados procesos)
- Todo programa en ejecucion esta cargado en la RAM (al menos parcialmente si no hay suficiente memoria)
- La memoria RAM puede verse como una matriz de celdas, cada celda guarda un byte
- Cada celda tiene una direccion de memoria definida lo que permite que el acceso a la celda sea aleatorio (no es necesario recorrer otras celdas para encontrar la celda de interes)

2.2 Espacios de memoria de un proceso

- Al ejecutarse un proceso, el sistema operativo reserva espacio en memoria para guardar los datos del proceso
- Lo procesos tienen 4 áreas de memoria principales (puede variar entre sistema operativos):
  - Call stack (pila de llamadas)
  - Heap
  - Datos (global)
  - Código

2.2.1 Call stack

- También llamado simplemente `stack`
- Es una zona de memoria que se comporta como una pila
- Su función principal es recordar el orden en que se hacen llamadas anidadas entre funciones
- También en varias lenguajes (aunque no es el caso de Python) es en esta zona donde se guardan por defecto las estructuras de datos creadas dentro de un función
- En esta memoria también se propagan los valores de retorno entre funciones
- Si un proceso excede su tamaño de pila asignado, se produce un error de `desbordamiento de pila` o `stack overflow`
- Esto puede suceder si se anidan demasiadas llamadas de funciones (varios miles)
- Un desbordamiento de pila provoca la terminación del proceso
- El programador no puede afectar directamente al Call Stack, los datos de esta zona se crean y destruyen automaticamente (lo hace el sistema operativo)
- Esta zona de memoria es la que permite que exista recursividad (explicado más adelante)

2.2.2 Heap

- Es un área especial del proceso designada para el uso de los programadores
- En algunos lenguajes como `C`, el programador puede reservar y borrar memoria en esta área
- El sistema operativo no limpia esta memoria (como si lo hace con el stack)
- En lenguajes de más alto nivel como `Python` el programador no toma decisiones sobre dónde colocar datos, se utiliza la opción más robusta de forma transparente

- Por ejemplo, en `Python` al crear una lista dentro de una función, se usa automáticamente el heap, de esta forma si la función regresa una referencia a la lista, no se corre el riesgo de que la memoria asociada a la lista sea limpiada de forma automática al terminar la función (como si pasaría si se guardara en el stack)
- Como se mencionó en temas anteriores, en lenguajes como `Python` se utiliza un recolector de basura para limpiar memoria que no se utiliza
- Más concretamente la limpieza se hace sobre el heap (dado que de por si el stack se limpia solo)

2.2.3 Datos

- Es un espacio especial para almacenar variables globales y

constantes

- Es un espacio fijo que se designa en tiempo de compilacion (en lenguajes compilados)

2.2.4 Código

- En este espacio se almacena el codigo del programa asociado al proceso
- De esta forma se puede llevar un control para que el procesador sepa que instruccion debe ejecutar a continuacion
- Es importante recordar en qué punto se encuentra la ejecución dado que los procesos entran y salen constantemente del procesador (el calendarizador del SO se encarga de esta función)

3 Funciones recursivas

- Como ya se mencionó, una función recursiva es aquella que se define en términos de ella misma
- Un ejemplo común es la función factorial (determina número de permutaciones de una colección)

$$(x)! = \begin{cases} 1 & \text{si } x = 0 \\ x * (x-1)! & \text{si } x > 0 \end{cases}$$

Ejemplo:

- $(4)! = 4 * (3)!$
- $(3)! = 3 * (2)!$
- $(2)! = 2 * (1)!$
- $(1)! = 1 * (0)!$
- $(0)! = 1$

- $(1)! = 1 * 1 = 1$
- $(2)! = 2 * 1 = 2$
- $(3)! = 3 * 2 = 6$
- $(4)! = 4 * 6 = 24$

- Las funciones recursivas tienen casos `base` y casos `recursivos`
- Un caso base es una entrada de la función para la cual se puede entregar directamente un valor
- Un caso recursivo es aquel que hace referencia a la propia función (hace una llamada a la propia función)
- Por ejemplo, la función factorial tiene un caso base y un caso recursivo
- Aunque dependiendo del problema puede haber varios casos base y recursivos

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

print(factorial(6))
```

```
720
```

3.1 Cómo funcionan las funciones recursivas

- Cada vez que se hace una llamada recursiva se apila la llamada en el stack, quedando pendiente la finalización de la función
- Al alcanzar un caso base se empieza a desapilar propagando resultados en la pila

- Dado que se trabaja con una pila (como se vio en el tema de pilas) las cosas suceden en el orden inverso después de la llamada recursiva
- Por ejemplo, una forma sencilla de imprimir del 5 al 1

```
def imprimir_inverso(n, maximo):
    if n <= maximo: # caso base implícito
        imprimir_inverso(n + 1 , maximo) # primero la llamada recursiva
        print(n)

imprimir_inverso(1, 5)
```

```
5
4
3
2
1
```

3.2 Diseño de soluciones recursivas

- Un error que cometen muchos programadores al crear funciones recursivas es pensar constantemente en lo que pasa en el stack
- NO hagas eso: es importante saber cómo funcionan las llamadas recursivas, pero pensar en ello aumenta la carga cognitiva y la complejidad
- En vez de eso debes pensar en los casos, esto es programación declarativa, no hay que pensar en el proceso detallado, sólo en definir el problema
- El manejo de la pila de llamadas es tu mecanismo de resolución automático, no es necesario pensar en cómo funciona

3.2.1 Consejos

- Empieza por los casos base, ¿En qué casos puedo regresar un resultado directamente?
- Luego piensa en los casos recursivos, ¿Qué debo hacer cuando no se puede resolver el caso, cómo debería tratarse?

- Considera que los parámetros que recibe tu función son muy importantes, hay que pensar muy bien en ellos
- En los casos recursivos los parámetros que pasas deberían variar de cierta forma, piensa bien en cuál es esa forma
- Si estás procesando listas, en cada llamada evalúa el elemento del frente y pasa el resto de la lista en los casos recursivos
- Muchas veces es mejor definir dos funciones: una recursiva y otra no recursiva que sólo manda a llamar a la función recursiva.
- La función recursiva puede requerir que se inicialicen parámetros de forma especial, cosa que se hace desde la función no recursiva
- También puede aprovecharse la función no recursiva para copiar memoria si es que la función recursiva modifica memoria

3.3 Recursividad directa

- Es la forma que se ha visto hasta el momento en el tema
- Se refiere a que en el cuerpo de la función se mande a llamar a la propia función

3.4 Recursividad indirecta

- Sucede cuando entre dos o más funciones se hacen llamadas entre si
- Es menos común y útil que la recursividad directa

```
def a(i):
    if i == 0:
        return True
    return b(i-1)

def b(i):
    return a(i)
```

4 Estilos de recursividad

- Al diseñar soluciones recursivas existen dos patrones de solución:
  - A la cola
  - No a la cola

4.1 Recursividad a la cola

- En general es el estilo más declarativo, por lo tanto suele dar soluciones cortas
- Se basa en dejar trabajo pendiente (en cola de espera, aunque en realidad sabemos que se usa una pila)
- Esto es, se hacen llamadas recursivas y luego se continúa con el código
- Este estilo no suele requerir agregar parámetros extra a la solución, por lo que tampoco es en muchas ocasiones necesario agregar una función no recursiva que inicialice los parámetros de la función recursiva
- Es ineficiente en memoria y procesador dado que requiere de apilar más trabajo

```
def mayor_lista(lista):
    if not lista:
        return -1
    frente = lista[0]
    resto = lista[1:]
    if not resto: # la lista tenía un solo elemento
        return frente
    mayor_resto = mayor_lista(resto) # trabajo pendiente adelante
    if frente > mayor_resto:
        return frente
    else:
        return mayor_resto
```

4.2 Recursividad no a la cola

- En cierto sentido es similar a un for y while
- Lo último que se hace en los casos recursivos es la llamada recursiva, esto es, no deja trabajo pendiente

- Normalmente requiere de agregar parámetros extra de control, por ejemplo índices o acumuladores de resultados
- En algunos lenguajes de programación, aunque no es el caso de `Python`, como `Common Lisp` este tipo de recursividad no apila llamadas en la pila (se saca de la pila la función que está terminando, por lo que no puede caer en un stack overflow)
- En esos lenguajes la recursividad no a la cola es muy similar en términos de eficiencia a un for o while tradicional
- En general es más eficiente que la recursividad a la cola

```
def mayor_rec(lista: list, mayor: int) -> int:
    """
    Regresa el número mayor, requiere inicialización.

    returns: int
    """
    if not lista: # lista vacía
        return mayor # mayor es un acumulador
    frente = lista.pop()
    if frente > mayor:
        return mayor_rec(lista, frente) # es lo último que se hace
    return mayor_rec(lista, mayor) # es lo último que se hace

# se requiere de una segunda función de inicialización
def mayor_lista(lista: list) -> int:
    """
    Regresa el mayor de la lista.

    lista: list
    returns: int
    """
    lista = lista[:]
    if not lista:
        return -1
    mayor = lista.pop()
    return mayor_rec(lista, mayor)
```