Universidad Nacional de San Martín

Licenciatura en Ciencia de Datos

Algoritmos II

PARTE 1

RECURSIÓN DE PILA Y COLA

Recursión de pila

```
1   funcion resolver(problema)
2     si problema es simple entonces
3       devolver solucion
4     sino
5       dividir problema en subproblema1..N
6       resolver(subproblema1)
7       resolver(subproblema2)
8       ..
9       resolver(subproblemaN)
10      combinar_soluciones
11      devolver solución
12    finSi
13  finFuncion
```

Recursión de pila

Del pseudocódigo anterior:

-Cuando se realiza la **primera invocación recursiva** (línea 6) la **información** de la **instancia actual** de la operación resolver se almacena en la **pila de ejecución.**

-El programa comienza a ejecutar una **nueva instancia recursiva y suspende la ejecución de la instancia actual.** Será completada luego de que finalice la invocación recursiva.

LAS OPERACIONES QUE TODAVÍA TIENEN SENTENCIAS PENDIENTES DE EJECUCIÓN VAN A ESTAR EN BLOQUES INFERIORES DE LA PILA

Recursión de pila

Def: La recursión de pila es aquella que se apoya en la estructura de la pila de ejecución para resolver el problema

Prop: La recursión de pila sucede cuando en el caso recursivo nos quedan operaciones pendientes por hacer luego de la invocación recursiva.

Recursión de pila

Ventajas	**Desventajas**
Más elegante	**SATURACIÓN DE PILA SI RECURSIÓN ES MUY PROFUNDA!!!** (recordar que la pila de ejecución es **FINITA**)
Forma **MÁS NATURAL**	

Recursión de pila: Factorial

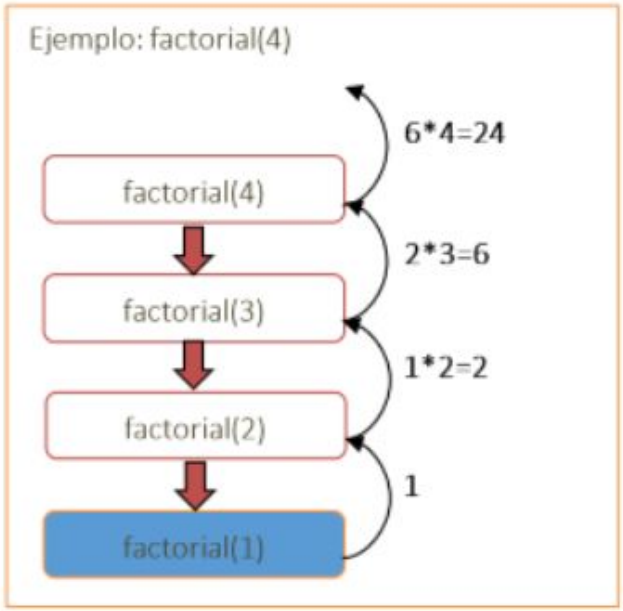
```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        resultado_parcial = factorial(n-1)
        return resultado_parcial * n
```

La variable resultado_parcial **no tendrá valor** asociado hasta que no **finalice** la invocación recursiva.

La instancia **actual** se suspende y quedan por hacer dos acciones:

- **multiplicar** el resultado de la llamada recursiva por n
- **devolver** el resultado

Recursión de pila: factorial



Cuando se llega al caso base comienza la **“vuelta hacia atrás”**

Recursión de pila

Cuando trabajamos con recursión de pila es común asociarse a la idea de construirla **de atrás hacia el principio.**

Recursión de cola

-Se utiliza ante casos de **recursión simple.**

-No requiere apoyarse en la pila de ejecución.

-Debe evitar realizar **operaciones posteriores** con el resultado parcial de la invocación recursiva: devuelve el resultado final a partir del resultado que provea la invocación recursiva.

-**NO EXISTE L LA "VUELTA ATRÁS" PARA CONSTRUIR EL RESULTADO FINAL: SE CONSTRUYE A MEDIDA QUE SE ENTRA EN LA RECURSIÓN Y SE DEVUELVE EL RESULTADO FINAL AL LLEGAR AL CASO BASE**

-EL RESULTADO COMIENZA A CONSTRUIRSE DESDE EL PRINCIPIO: ENFOQUE "DESDE ARRIBA HACIA ABAJO" (TOP-DOWN)

-EN LA RECURSIÓN DE COLA LA ÚLTIMA SENTENCIA ES LA INVOCACIÓN RECURSIVA!!!!

Recursión de cola: factorial

```
def factorial(n: int) -> int:
    def factorial_interna(n: int, acumulador: int) -> int:
        if n <= 1:
            return acumulador
        else:
            return factorial_interna(n-1, acumulador * n)
    return factorial_interna(n, 1)
```

Recursión de cola: Factorial

`def factorial_interna(n: int, acumulador: int) -> int:`: Dentro de la función `factorial`, hay otra función definida llamada `factorial_interna`. Toma dos argumentos: `n` (el número para calcular el factorial) y `acumulador` (el valor acumulado del factorial).

`if n <= 1:`: e verifica si `n` es menor o igual a 1. Si es así, retorna el `acumulador` actual, ya que el factorial de 0 o 1 es 1. 4. `return acumulador`: Esto es lo que se devuelve cuando `n` es menor o igual a 1. Es el **CASO BASE**

5. `else:`: **PARTE RECURSIVA**

6. `return factorial_interna(n-1, acumulador * n)`. En cada llamada recursiva, `n` se reduce en 1, y el `acumulador` se multiplica por el valor actual de `n`. La recursión continúa hasta que `n` alcanza 1.

7. `return factorial_interna(n, 1)`: Finalmente, se llama a la función `factorial_interna` con los argumentos `n` y `1`. Esta llamada inicial inicia el proceso de cálculo del factorial, con un valor inicial de `acumulador` igual a 1.

Recursión de cola: factorial

-La **función interna** es útil porque la externa la inicializa y es ocultado a quien la consume. Ej: acumulador en 1.

-La función recursiva **cambió su caso base**: devuelve el acumulador, y su **caso recursivo**: ahora la última sentencia es la invocación recursiva de **factorial_interna**

-LA CONSTRUCCIÓN DE LA SOLUCIÓN SE PRODUCE PREVIO A LA INVOCACIÓN Y SE PASA COMO 2DO ARGUMENTO

Recursión de cola: factorial

Se podría haber descompuesto el retorno del caso recursivo:

```
def factorial(n: int) -> int:
    def factorial_interna(n: int, acumulador: int) -> int:
        if n <= 1:
            return acumulador
        else:
            resultado_parcial = acumulador * n
            return factorial_interna(n-1, resultado_parcial)
    return factorial_interna(n, 1)
```

Recursión de cola: factorial

-Con la descomposición se resalta que la **última operación** del caso recursivo no queda **nada pendiente por calcular,** sólo devolver el resultado.

-Cada invocación sucesiva a instancias de problemas más pequeños da un **resultado parcial** que se acerca cada vez más a la **solución final.**

Recursión de pila y cola: comparación

```python
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        resultado_parcial = factorial(n-1)
        return resultado_parcial * n

def factorial(n: int) -> int:
    def factorial_interna(n: int, acumulador: int) -> int:
        if n <= 1:
            return acumulador
        else:
            return factorial_interna(n-1, acumulador * n)
    return factorial_interna(n, 1)
```

Recursión de cola: iteración

-LA MAYOR VENTAJA DE LA **RECURSIÓN DE COLA** ES QUE PODEMOS TRANSFORMARLA EN UNA **ITERACIÓN SIEMPRE!!!** POR ESO SE LLAMA TAMBIÉN **FALSA RECURSIÓN**

-Algunos lenguajes tienen **tail call optimization** (detecta la recursión de cola y la transforma automáticamente en una iteración). **PYTHON NO => es necesario hacer la conversión manualmente.**

La transformación, en términos genéricos consiste en:

1) Cambiar el **if** por un **while**
2) La condición **if(caso base)** pasa a ser la **condición del while**
3) El **caso recursivo** es el **cuerpo** del while
4) El **retorno** del **caso base** es el **retorno final** de la solución iterativa

Recursión de cola: iteración factorial

```
def factorial(n: int) -> int:
    solucion = 1
    while n > 1:
        solucion *= n
        n -= 1
    return solucion
```

En este caso, es posible ya que se cumplen las condiciones de
1)**CONMUTATIVIDAD:**
a+b= b+a
a*b=b*a
2)**ASOCIATIVIDAD**
(a+b)+c = a +(b+c)
a*(b*c) = (a*b)*c

Eliminando la recursión

Significa evitar la recursión de pila, transformándola en una recursión de cola (o iteración)

Eliminando la recursión: acumulando la solución parcial

Es el ejemplo del factorial ya visto, que se apoya en un acumulador

```
def factorial(n: int) -> int:
    def factorial_interna(n: int, acumulador: int) -> int:
        if n <= 1:
            return acumulador
        else:
            return factorial_interna(n-1, acumulador * n)
    return factorial_interna(n, 1)
```

Eliminando la recursión: acumulando la solución parcial

```
def resta_lista(xs: list[int]) -> int:
    if xs == []:
        return 0
    else:
        return xs[0] - resta_lista(xs[1:])
```

Calcula la resta acumulativa

```
def resta_lista(xs: list[int]) -> int:
    def resta_lista_interna(xs: list[int], acumulador: int) -> int:
        if xs == []:
            return acumulador
        else:
            return resta_lista_interna(xs[1:], acumulador - xs[0])
    return 0 if xs == [] else resta_lista_interna(xs[1:], xs[0])
```

Eliminando la recursión: utilizando pila explícita

-Se usa cuando no podemos utilizar la recursión de pila con un acumulador.

-**GESTIONAMOS NUESTRA PROPIA PILA DE EJECUCIÓN EN UN OBJETO PILA O STACK.**

-Simulamos manualmente el **apilado** y **desapilado**

Eliminado de la recursión: utilizando listas explícitas

```
def resta_lista(xs: list[int]) -> int:
    def apilado(xs: list[int], pila: list[int]):
        # Esta función apila los elementos de la lista `xs` en
la pila.
        # Cada elemento se agrega a la pila uno por uno.
        if xs != []:
            pila.append(xs[0])
            apilado(xs[1:], pila)
```

(sigue el código en la siguiente diapo, desapilado está identado al mismo nivel que apilado, la pueden ver completa en el github)

```python
def desapilado(pila: list[int], acumulador: int) -> int:
        # Esta función desapila los elementos de la pila y calcula la resta acumulada.
        # Inicialmente, el acumulador está establecido en 0.
        if pila == []:
            return acumulador
        else:
            # Se extrae un elemento de la pila y se resta del acumulador.
            # Luego, se llama recursivamente a la función desapilado con el nuevo
acumulador.
            return desapilado(pila, pila.pop() - acumulador)

    # Se crea una pila vacía.
    pila = []
    # Se llama a la función apilado para apilar los elementos de la lista en la pila.
    apilado(xs, pila)
    # Se llama a la función desapilado para calcular la resta acumulada.
    return desapilado(pila, 0)
```