Recursividad

Algoritmos y Estructuras de datos

Prof. Ing. Gomez Pablo

29 de marzo de 2023

Introducción

Temario

- Recursividad
- Recursividad Simple, Múltiple, Directa, Indirecta
- Factorial
- Fibonacci
- Fibonacci Simple
- Recursividad de Cola
- De recursivo a iterativo

Introducción

FCEFyN

Definición

Una función es aquella que se llama a si misma en su cuerpo
$f(x) = \ldots . f(x) \ldots .$

Recursividad

FCEFyN

Condiciones de una función recursiva

- Debe existir una condición de corte de la recursión
- Se debe garantizar que la condicion sea alcanzada eventualmente

Recursividad

Definición

Se escriben como una función definida por tramos

$$f(x) = \begin{cases} \dots & \Rightarrow \text{si se cumple condición} \\ x * fact(x-1) & \Rightarrow \text{si no se cumple condición} \end{cases}$$

Recursividad

FCEFyN

Iterative

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

$$n! = \prod_{k=1}^{n} k$$

Names: n, total, k, fact_iter

Recursive

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

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names: n, fact

Tipos de recursión

Recursión Simple

Existe una única llamada a la función en el cuerpo

Recursión Múltiple

Hay dos o más llamadas a la función en el cuerpo

Tipos de recursión

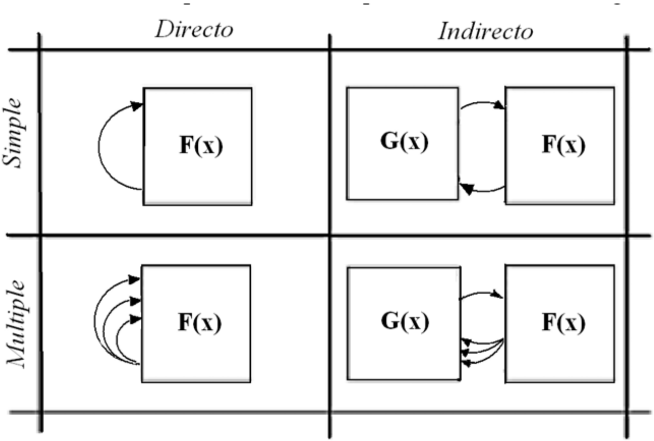
Recursión Directa

La función se llama siempre a sí misma.

Recursión Indirecta

La función la a una segunda función que vuelve a llamar a la primera.

Tipos de recursión



Recursividad

Ejemplo : Función Factorial

$$fact(n) = \begin{cases} 1 & \Rightarrow n = 1 \\ n * fact(n-1) & \Rightarrow n > 1 \end{cases}$$

Despliegue de la recursividad

$$\begin{aligned} fact(5) &= 5 * fact(4) \\ &= 5 * 4 * fact(3) \\ &= 5 * 4 * 3 * fact(2) \\ &= 5 * 4 * 3 * 2 * fact(1) \\ &= 5 * 4 * 3 * 2 * 1 \end{aligned}$$

Factorial recursivo

```
#include <iostream>
int fact( int n) {
    if (n < 1) {error}
    if (n==1) {
        return 1;
    } else {
        return fact(n-1) * n ;
    }
}
```

Recursividad

FCEFyN

Ejemplo : Sucección de Fibonacci

$$fib(x) = \begin{cases} 1 & \Rightarrow x = 1 \\ 1 & \Rightarrow x = 2 \\ fib(x-1) + fib(x-2) & \Rightarrow x > 2 \end{cases}$$

Despliegue de la recursividad

$$fib(1) = 1$$
$$fib(2) = 1 + 1 = fib(1) + fib(1) = 2$$
$$fib(3) = 1 + 2 = fib(1) + fib(2) = 3$$
$$fib(4) = 2 + 3 = fib(2) + fib(3) = 5$$
$$fib(x) = fib(x-2) + fib(x-1)$$

Fibonacci recursivo

```
#include <iostream>
using namespace std;
int fib(x){
    if(x< 3){
        return 1
    }
    else{
        return fib(x-1) + fib(x-2)
    }
}

int main() {
  int result = fib(7);
  cout<< result<<endl;
  return 0;
}
```

Recursividad

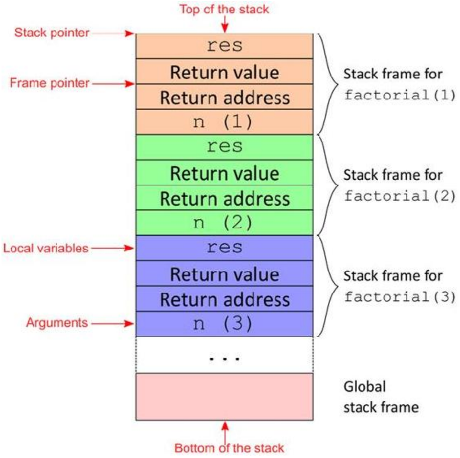
Llamado de funciones

Como ya sabemos cada vez que se llama a una función se crea una nueva entrada en la pila. Esta estructura se llama Stack Frame y consiste en los campos necesarios para continuar la ejecución luego de procesar la función.

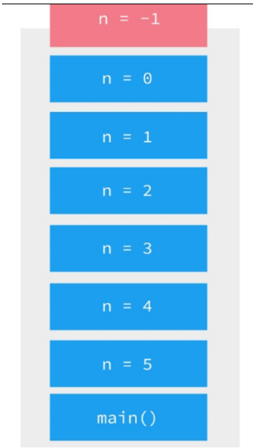
Stack Frame

- Puntero a la instrucción próxima instrucción cuando retorne.
- Todos los argumentos pasados a la función.
- Las variables locales
- El valor de retorno.

Stack frames



Stack overflow



Recursividad

Recursiva doble a recursiva simple

$$fib(x) = \begin{cases} 1 & \Rightarrow x < 3 \\ fib_{\text{aux}}(3, 1, 1, x) & \Rightarrow x > 3 \end{cases}$$

$$fib_{\text{aux}}(y, a_1, a_2, x) = \begin{cases} a_1 + a_2 & \Rightarrow y = x \\ fib_{\text{aux}}(y + 1, a_1 + a_2, a_1, x) & \Rightarrow y < x \end{cases}$$

Fibonacci recursivo simple

```c
int fibaux (int y, int a1, int a2, int x){
      if(y==x){
         return a1 + a2;
      } else{
         return fibaux(y+1, a1+a2, a1, x);
      }
}
int fib2(int x){
      if (x<3){
        return 1;
      } else{
         return fibaux(3, 1, 1, x);
      }
}
```

De recursivo a iterativo

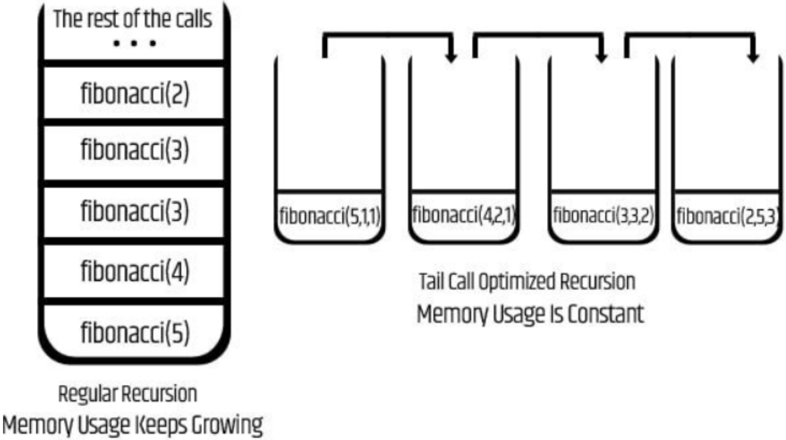
Sólo la función auxiliar es recursiva

Recursión de cola

El valor que retorna la función es el llamado a la función recursiva (no sería recursiva a la cola si al resultado de la recursión, le aplicara una nueva función).

En general es conveniente dejarlo expresado como una función recursiva de cola, para poder hacer el paso a una función iterativa. En principio siempre que se tenga esta condición, se puede transformar en una función iterativa que va a ser más eficiente en términos de memoria y no hay riesgo de quedarse sin memoria del stack por excesivas llamadas a la función recursiva

Optimización de la memoria



De recursivo a iterativo

f es una función recursiva de cola $f : T_1 x T_2 \Rightarrow T_3$

T_1 es producto cartesiano de los dominios que no varian de llamada en llamada recursiva
T_2 es prod cartesiano de los dominios que sí varian de llamada en llamada recursiva
T_3 es prod cartesiano de los dominios de la imagen

En la función recursiva simple tenemos las variables:
$t_1 : T_1, \quad t_2 : T_2, \quad t_3 : T_3$

De recursivo a iterativo

Se hará una transformación de f del tipo:

$$f(t_1, t_2) = \begin{cases} h(t_1, t_2) & \Rightarrow d(t_1, t_2) \\ f(t_1, s(t_1, t_2)) & \Rightarrow !d(t_1, t_2) \end{cases}$$

$d : T_1 x T_2 \Rightarrow Booleano \Rightarrow$ devuelve true si se ha alcanzado la condición de recursión y false en caso contrario.

$h : T_1 x T_2 \Rightarrow T_3 \Rightarrow$ caso devuelto por la función cuando se alcanza el corte (por eso devuelve variable de tipo t_3)

$s : T_1 x T_2 \Rightarrow T_2 \Rightarrow$ es la función que describe cómo varían los dominios (t_2) de llamada en llamada (depende de t_1 y t_2 pero siempre debe devolver t_2)

De recursivo a iterativo

El programa con recursión de cola quedaría

```
t3 f(t1, t2){
    if(d(t1, t2)){
        return h(t1, t2);
    } else{
        return f(t1, s(t1, t2));
    }
}
```

Pasando a iterativo

```
t3 f(t1, t2){
    while(!d(t1, t2)){
        t2 = s(t1, t2));
    }
    return h(t1, t2);
}
```

De recursivo a iterativo

Dada la expresión matemática de la función recursiva simple a la cola de Fibonacci:

$$fib(x) = \begin{cases} 1 & \Rightarrow x < 3 \\ fib_{aux}(3, 1, 1, x) & \Rightarrow x > 3 \end{cases}$$

$$fib_{aux}(y, a_1, a_2, x) = \begin{cases} a_1 + a_2 & \Rightarrow y = x \\ fib_{aux}(y', a_1', a_2', x) & \Rightarrow y < x \end{cases}$$

$T_1 : Nat$
$T_2 : NatxNatxNat$
$T_3 : Nat$

De recursivo a iterativo

- Identificar las variables de tipo $t_1 \Rightarrow x$
- Identificar las variables de tipo $t_2 \Rightarrow$ y, a_1, a_2
- Identificar $d(t_1, t_2)$ (condición de corte de la recursión) $\Rightarrow y < x$
- Identificar $h(t_1, t_2)$ (valor devuelto cuando la condición de corte es alcanzada) $\Rightarrow a_1 + a_2$ (de esto se deduce que T_3 es de tipo Nat)
- Identificar $s(t_1, t_2)$ (como varían las variables de una iteración a otra) $\Rightarrow$
  $y' = y + 1$
  $a_1' = a_1 + a_2$
  $a_2' = a_1$
- Escribir la forma iterativa.

De recursivo a iterativo

FCEFyN

```
int fib(int x){
    if (x<3){
     return 1;
    } else{
      return fib_sec(3, 1, 1, x);
    }
}
int fib_sec(y, a1, a2, x){
    while(y<x){
        int aux = a1;
        y++;
        a1 = a1 + a2;
        a2 = aux;
    }
    return a1 + a2;
}
```

De recursivo a iterativo

```
int fib(x){
    if x<=2 {
        return 1;
    } else {
        int y = 3;
        int a1 = 1;
        int a2 = 1;
        while (!(y==x)) {
            y=y+1;
            int aux = a1;
            a1=a1+a2;
            a2=aux;
        }
        return a1+a2;
    }
}
```

De recursivo a iterativo

Encontrar el mayor elemento en un arreglo

$f(a[int]) \Rightarrow int$

$f(a[]) = f_{aux}(a[], i, m, size)$

$$f_{aux}(a[], i, m, size) = \begin{cases} m & \Rightarrow i = size \\ f_{aux}(a[], i+1, max(m, a[i]), size) & \Rightarrow i < size \end{cases}$$

$$max(x, y) = \begin{cases} x & \Rightarrow x > y \\ y & \Rightarrow x \leq y \end{cases}$$

$T_1 : Nat[]xNat$

$T_2 : NatxNat$

$T_3 : Nat$

De recursivo a iterativo

- Identificar las variables de tipo $t_1 \Rightarrow a[], size$
- Identificar las variables de tipo $t_2 \Rightarrow i, m$
- Identificar $d(t_1, t_2) \Rightarrow i = size$
- Identificar $h(t_1, t_2) \Rightarrow m$
- Identificar $s(t_1, t_2) \Rightarrow$
  $i' = i + 1$
  $m = max(m, a[i])$
- Escribir la forma iterativa.

De recursivo a iterativo

```
int max(x, y){
     if(x>y) return x;
     else return y;
}
```

De recursivo a iterativo

```
//Forma recursiva
int maxArray(a[]){
      int size = a[].size();
         if (size>1){
            return maxArrayAux(a[], 0, a[0], size);
      }
}
int maxArrayAux(a[], int i, int m, int size){
   if(i==size){
      return m;
   } else{
      return
         maxArrayAux(a[], i+1, max(m, a[i]), size);
   }
}
```

De recursivo a iterativo

```
//Forma iterativa
int maxArraySec(a[]){
    int i = 0;
    int m = a[0];
    int size = a[].size();

    while(i != size){
        i++;
        m = max(m, a[i]);
    }

    return m;
}
```