Algoritmos recursivos

Un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva.

El código 1a implementa una versión recursiva de la función Factorial N → N
El código 1b implementa una versión iterativa de la misma función.

Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño u orden n, cada nueva ejecución recurrente del mismo se planteará sobre problemas, de igual naturaleza que el original, pero de un tamaño menor que n.

De esta forma, al ir reduciendo progresivamente la complejidad del problema que resolver, llegará un momento en que su resolución sea más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva).

En esa situación diremos que estamos ante un **caso base** de la recursividad.

```
static public int factorialRecur(int n){

      if (n == 1){
            return 1; // caso base
      }else{
            return n * factorialRecur(n-1);
      }
}
```

Código 1ª: factorial recursivo

```
static public int factorialIter(int n){
      int ret = n;
      for (int i=n-1; i>1; i--){
            ret = ret * i;
      }
      return ret;
}
```

Código 1b: factorial iterativo

Las claves para construir un subprograma recursivo son:

(1) Cada llamada recursiva se debería definir sobre un problema de menor complejidad (algo más fácil de resolver).

(2) Ha de existir al menos un caso base para evitar que la recurrencia sea infinita.

Si ocurren (1) y (2) tenemos garantizado que el programa finalizara en algún momento.

En el caso de `factorialRecur(), cada llamada recursiva se realiza con un problema de tamaño n-1.`

El caso base ocurre cuando n = 1.

Iterativo Vs Recursivo

Las versiones recursivas suelen ser más breves(menos líneas de código), pero más lentas (tiempo de ejecución).

Y las versiones iterativas suelen ser más declarativas (esta claro como se resuelve el problema), pero se ven en problemas a la hora de recorrer estructuras de naturaleza recursiva, por ejemplo, un árbol binario.

Por lo tanto, no hay una regla general que indique cuando resolver un problema de una manera u otra.

Tipos de recursión

Hay dos tipos de recursión.

Una recursión es lineal, si existe una única llamada recursiva, por ejemplo, en `factorialRecur()`.

La recursión es no lineal si existen dos o mas llamadas recursivas, como por ejemplo, en fibonacciRecur() (Ver Código 2a).

```
static int fibonacciRecur(int n){   O( (1+√5/2)^n )
    if (n <= 1)
        return n;        // f0 = 0 y f1= 1
    else
        return fibonacciRecur(n - 1) + fibonacciRecur(n - 2);
            }
```

Código 2a: recursión no lineal

Ejercicio1: Como demostrar que fibonacciRecur() finaliza?
(1)Todas las llamadas recursivas se invocan con un valor < n
(2)Tiene un caso no recursivo en el cual el programa finaliza.

Ejercicio2: Como será la versión iterativa?

```
static int fibonacciIter(int n){    // O(n)
      int n0 = 0;
      int n1 = 1;
      int n2 = n0 + n1;

      if (n==0) { return n0;}
      if (n==1) { return n1;}

      if (n>2) {
            for (int i=2; i<n; i++){
                  n2  = n1 + n0;
                  n0 = n1;
                  n1 = n2;
            }
       }
     return n2;
}
```

Código 2b: fibonacci iterativo

Observaciones:

El código 2a y el código 2b hacen lo mismo?

Lamentablemente no hay una formula general, para demostrar que dos códigos hacen lo mismo.

Afortunadamente para fibo, si se puede demostrar que ambos códigos son equivalentes. Pero la demostración escapa al alcance de la materia.

Sin embargo, en la clase de complejidad demostraremos que versión es la que tarda más tiempo en finalizar.

Desenrollo

Desenrrollar es una técnica que permite pasar de un algoritmo recurviso a uno iterativo. Veremos esta técnica solo para instancias particulares

Consiste en reemplazar código recursivo por su definición no recursiva en el algoritmo original, mientras sea posible.

Ejemplo1: Desenrollo de factorial(5)

factorial(5) = 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1 // este punto se resuelve de manera iterativa

Algunas observaciones del desenrollo

1) Nos permite testear (aunque no demostrar) correctitud
2) Nos permite conjeturar una complejidad sin conocerla. En este caso se puede conjeturar O(n)

Ejemplo2: Búsqueda binaria en vectores ordenados

```
//devuelvo el índice dentro de A o -1
static int busquedaBinaria(int A[],int X, int i, int j){
int medio;
if (i>j) return -1;
medio = (i+j) / 2;

if (A[medio] < X) return busquedaBinaria(A,X,medio+1,j);
else if (A[medio] > X) return busquedaBinaria(A,X,i,medio-1);
else return medio;
}
```

Desenrollo para

```
int [] lista = {1,3,5,7,8};
System.out.println(busquedaBinaria(lista,3,0,4));

busquedaBinaria(lista,3,0,4) =
   medio = (0 + 4) / 2 = 2
   Como 0 < 4       // i < j
   Como 5 > 3       // A[2] > 3
= busquedaBinaria(lista,3, 0 ,2 - 1) = busquedaBinaria(lista,3,0 ,1) =
   medio = (0 + 1) / 2 = 1
   Como 0 < 1       // i < j
   Como 3 == 3       // A[medio] == X

= return 1
```

Algunas observaciones del desenrollo

1) Podemos suponer que como tiene 2 llamadas recursivas, tengo mayor complejidad que en el ejemplo anterior? Veamos por qué no.

   Se puede demostrar que ambas llamadas no pueden ocurrir “al mismo tiempo”, es decir, que son excluyentes entre sí.

   Sin embargo, si siempre ocurre una, se podría concluir erróneamente una complejidad lineal.

   En este caso ,no siempre es posible, podremos demostrar que la complejidad es a lo sumo Log(n). La clave esta en ver que el parámetro de entrada (j + i) converge más rápido que en el ejemplo anterior. Sea **k** = j +i

Demostración:

Està claro que el algoritmo va a continuar mientras (**k** / 2^i) > 1 porque a partir de ese punto no se pueden generar nuevos índices.

Además, en cada llamada recusiva, esta distancia se acorta a la mitad.

Entonces: Si i es el número de llamada recursiva

El algoritmo va a continuar mientras

k / 2^i > 1 ⇔ // pasamos el 2^i
k > 2^i ⇔
log (k) > log(2^i) ⇔ // aplico Log
log (k) > i

Demostramos que la cantidad de llamadas recursivas i, es a lo sumo log (k).