La Sénia
Institut d'Educació Secundària
Paiporta

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Recursividad

Curso 2023 - 2024

1. Introducción

Una función recursiva es aquella que se llama a sí misma de forma que se van obteniendo versiones más sencillas del problema.

La recursividad es una alternativa a la iteración.

En términos de tiempo de ejecución y ocupación de memoria, la solución recursiva es menos eficiente que la iterativa.

Sin embargo, existen situaciones en las que la recursividad es una solución simple y natural a un problema que en otro caso será difícil de resolver.

2. Características

- La característica principal de la recursividad es que siempre existe una forma de finalizar la recursión (caso **base** o salida)
- Las llamadas recursivas pueden tener una parte que se resuelve y otra se llama a sí misma con una versión más sencilla (paso de **recursión**)
- Si una función recursiva sin caso base produce un bucle infinito:

```
public void infinito(){
    infinito();
}
```

3. El problema recursivo

1. Obtener una definición exacta del problema a resolver.

2. Determinar el tamaño del problema completo que hay que resolver. Este tamaño determinará los valores de los parámetros en la llamada inicial a la función.

3. Resolver el caso base en el que el problema puede expresarse no recursivamente.

4. Resolver el caso general correctamente en términos de un caso más pequeño: la llamada recursiva.

3. El problema recursivo

Ejemplo. Factorial de un número

El factorial de un entero no negativo n, esta definido como:

$$n! = n * (n-1) * (n-2) * \ldots *4*3 *2 *1$$

- 1! es igual a 1
- 0! se define como 1

3. El problema recursivo

Factorial de un número (forma iterativa)

El factorial de un entero k puede calcularse de manera iterativa como sigue:

```
public static int factorialIterativo(int n) {
        int fact = 1; //Caso base
        for (int i = n; i >= 1; i--) {  //Iteración
            fact = fact * i;
        }
        return fact;
    }
```

3. El problema recursivo

Resolver el problema de forma recursiva

1. Definición exacta (multiplicar el número por sus anteriores)
2. A partir de un número, el factorial se calcula mediante la multiplicación de él mismo y sus anteriores reduciendo el problema a multiplicaciones de números más bajos.
3. El caso base es 1! o 0! En ambos casos el resultado es 1
4. La recursividad se plantea como:

$$n! = n * (n-1)!$$

3. El problema recursivo

Factorial de un número (forma recursiva)

Podemos definir el factorial como:

Si n > 0 y 1 → n * (n-1)!

si n=1 o n=0 → 1

```
public static int factorialRecursivo(int n) {
    int fact;
    if (n == 0) {
        fact = 1;   //caso base
    } else {
        fact = n * factorialRecursivo(n - 1);   //caso recursivo
    }

    return fact;

}
```

3. El problema recursivo

Factorial de un número: Pila de llamadas

factorialRecursivo(1)	return 1
2*factorialRecursivo(1)	2*1 – return 2
3*factorialRecursivo(2)	3*2 – return 6
4*factorialRecursivo(3)	4*6 – return 24
5*factorialRecursivo(4)	5*24 – return 120
factorialRecursivo(5)	Solución 120

4. Tipos de Recursividad

Recursividad **simple**: Sólo aparece una llamada recursiva y se pueden transformar fácilmente en algoritmos iterativos.

Recursividad **múltiple**: Dentro de la función hay más de una llamada a sí misma. Es más difícil de hacer de forma iterativa.

Recursividad **anidada**: En alguno de los argumentos de la llamada recursiva hay una nueva llamada a sí misma.

Recursividad **cruzada**: La función provoca una llamada a sí misma a través de otras funciones que son llamadas desde ella misma.

4. Tipos de Recursividad

Recursividad simple

Ejemplo: **Factorial**

```
    public static int factorialRecursivo(int N) {
        int fact;
        if (N == 0) {
            fact = 1;   //Caso base
        } else {
            fact = N * factorialRecursivo(N - 1);   //caso recursivo
        }
        return fact;
    }
}
```

Recursividad múltiple

Ejemplo: **Fibonacci.** Cada número es igual a la suma de sus dos anteriores

```
public static int fibonacci(int N) {
    int res;
    if (N == 0) {
        res = 0;
    } else if (N == 1) {
        res=1;
    } else {
        res = fibonacci(N - 1) + fibonacci(N - 2);
    }
    return res;
}
```

```
0
1
1
2
3
5
8
13
21
34
55
```

4. Tipos de Recursividad

Recursividad anidada.

Ejemplo: **Ackermann**

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

```
    public static long ackermann(long M, long N) {
        long res;
        if (M == 0) {
            res = N+1;
        } else if (M>0 && N == 0) {
            res=ackermann(M-1,1);
        } else {
            res = ackermann(M - 1, ackermann(M,N - 1));
        }
        return res;
    }
}
```

4. Tipos de Recursividad

Recursividad cruzada.

Una función llama a otra función y ésta a su vez llama a la función que inicial.

Ejemplo: **Par / Impar**

```
public static int par(int N) {
        int res;
        if (N == 0) {
            res = 1; // 1 como true
        } else { res = impar(N - 1); }
        return res;
}

public static int impar(int N) {
        int res;
        if (N == 0) {
            res = 0; // 0 como false
        } else { res = par(N - 1); }
        return res;
}
```

5. Recursividad o Iteración

Eficiencia: Una solución no recursiva es más eficiente en términos de tiempo y espacio de computadora. La solución recursiva puede requerir gastos considerables, y deben guardarse copias de variables locales y temporales.

Recursos: El sistema puede no tener suficiente espacio para ejecutar una solución recursiva de algunos problemas ya que algunos valores son calculados una y otra vez causando que la capacidad de la computadora se exceda antes de obtener una respuesta.

Claridad: En algunos casos una solución recursiva es más simple y más natural de escribir.

6. Ejercicios

Esquemas recursivos de ORDENACIÓN.

- inserción directa
- divide y venceras
- mergeSort u ordenación rápida, quicksort

Esquemas recursivos de BÚSQUEDA.

- binaria o dicotómica, rápida o quicksort)

6. Ejercicios

Esquemas recursivos de RECORRIDO ascendente

```
public static void recorrerAscendente(tipoBase[] a, int inicio, int fin){
    if (inicio>fin){
        tratarVacio();
    } else {
        tratar(a[inicio]);
        recorrerAscendente(a, inicio+1, fin);
    }
}
```

6. Ejercicios

Esquemas recursivos de RECORRIDO descendente

/* inicio=izq y izq-1<=fin<a.length */

```
public static void recorrer(tipoBase[] a, int inicio, int fin) {
    if (fin<inicio){
        tratarVacio();
    } else {
        tratar(a[fin]);
        recorrer(a, inicio, fin-1);
    }
}
```

6. Ejercicios

Esquemas recursivos de BÚSQUEDA

/* 0<=inicio<=der+1 y fin=der*/

```
public static int buscar(tipoBase[] a, int inicio, int fin) {
    int resMetodo = -1;
    if (inicio<=fin) { //No hacer nada }
    else{
        if (propiedad(a[inicio])){
            resMetodo = inicio;
        }else{
            resMetodo = buscar(a, inicio+1, fin);
        }
    }
    return resMetodo;
}
```