dit
UPM

Tema 1: Introducción /recursión

José A. Mañas

http://jungla.dit.upm.es/~pepe/doc/adsw/index.html

13.2.2018

ETSIT UPM

índice

- concepto: recursión
  - forma de resolver un problema recurriendo a una versión más simple de sí mismo
- eficiencia
- fallos típicos

recursión

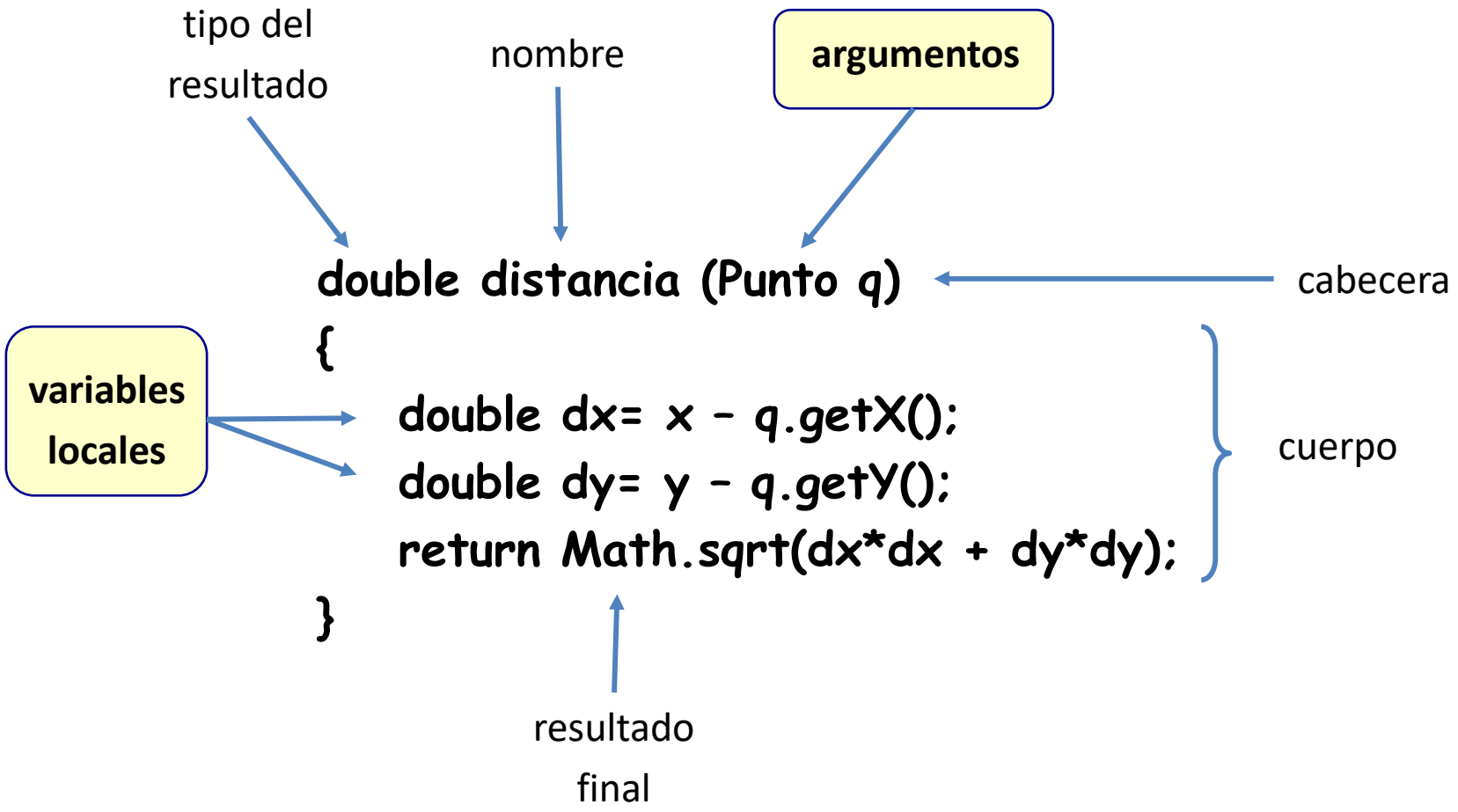
- suma de una serie aritmética
  - S(a, k, n)= a, a+k, a+2k, a+3k, …, a+nk
  - S(a, k, 0) = a
  - S(a, k, n) = S(a, k, n-1) + a+nk

```
public int suma(int n) {
      if (n == 0)
         return a;
      else
         return suma(n - 1) + a + n * k;
   }
```

recursión

- Algoritmos que se resuelven recurriendo a un subproblema similar más simple
  - similares a las pruebas por inducción
- Necesitan
  - una condición de parada
  - converger en cada paso hacia la parada
- Ventaja
  - en muchos problemas la corrección del algoritmo es evidente

anatomía de un método java



ejemplo

```
public class Palindromo {
   public static boolean check(String s) {
      return check(s, 0, s.length() - 1);
   }

   private static boolean check(String s, int a, int z) {
      if (a >= z)
         return true;
      if (s.charAt(a) != s.charAt(z))
         return false;
      return check(s, a + 1, z - 1);
   }
```

set up

condiciones de terminación

paso de recursión

argumentos y variables locales

- Se crean con la llamada
- Desaparecen al acabar

- Si hay varias llamadas simultáneas,
  - cada llamada tiene sus variables propias

factorial recursivo

n! = n * (n-1)!

```
int factorial (int n) {
    if (n < 1) return 1;
    return n * factorial(n-1);
}
```

- Hay que cerciorarse de que no es un círculo vicioso

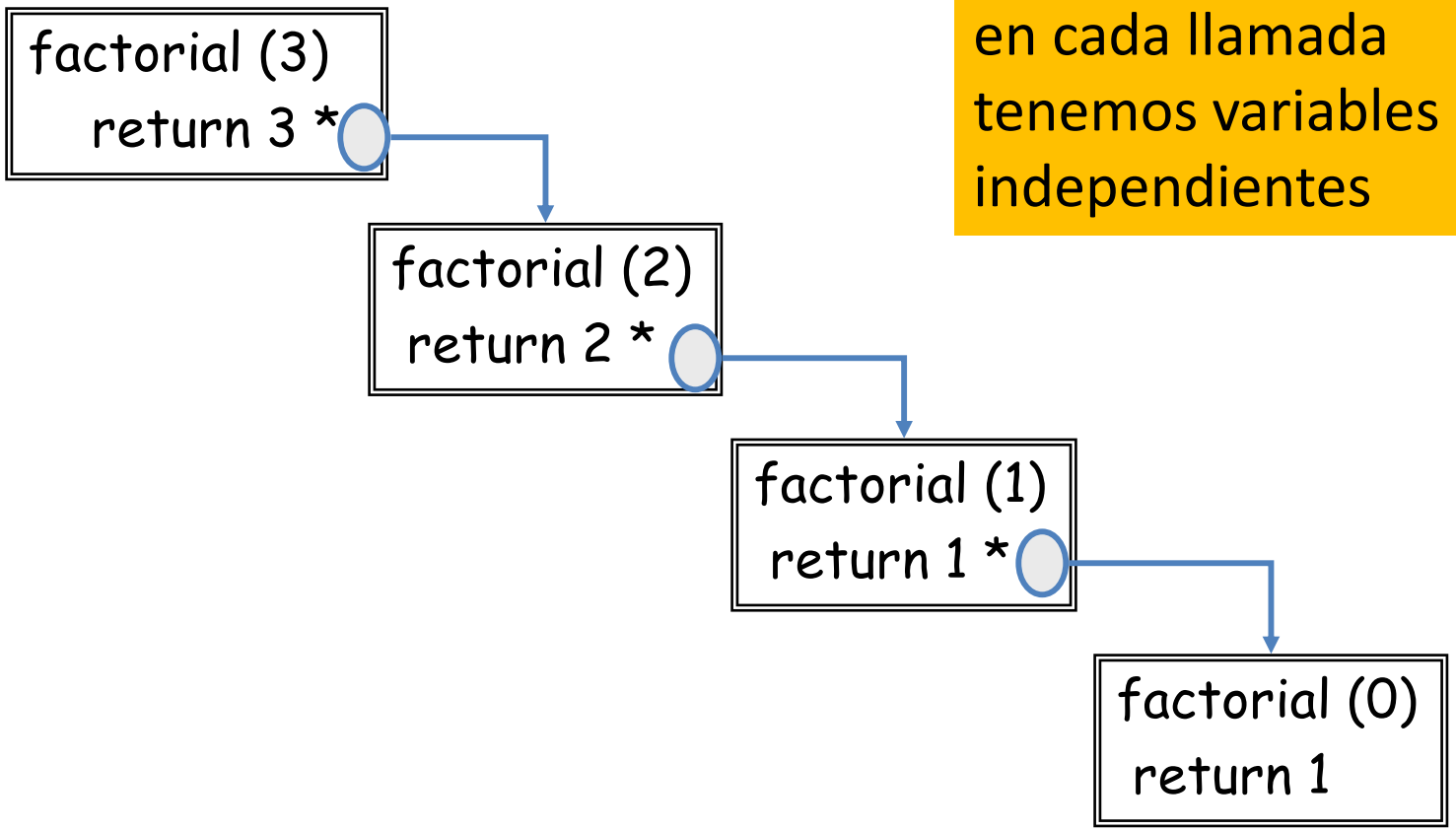
```
int factorial (int n) {
    if (n == 0) return 1;
    return n * factorial(n-1);
}
```

factorial recursivo

```
public int factorial(int n) {
    if (n < 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
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 * factorial( 0 ) ) ) ) )
               = 5 * ( 4 * (3 * (2 * ( 1 * 1 ) ) ) )
               = 5 * 4 * 3 * 2 * 1 * 1
               = 120
```

factorial recursivo



en cada llamada
tenemos variables locales
independientes

eficiencia

- sin dudar de que efectivamente se encuentre la solución,
- la eficiencia del proceso se ve perjudicada por la necesidad de mantener información del estado previo para poder regresar tras resolver el subproblema más sencillo

recursión → iteración

- puede ser fácil pasar a iteración
- tail recursión: lo último que se hace es llamarse a sí mismo y no hay que conservar el estado anterior

```
private static boolean checki(String s, int a, int z) {
      while (a < z) {
         if (s.charAt(a) != s.charAt(z))
            return false;
         a++;
         z--;
      }
      return true;
   }
```

factorial iterativo

```
public int metodo2(int n) {
    int resultado = 1;
    for (int i = 1; i <= n; i++)
        resultado *= i;
    return resultado;
}
```

- ¿Usa menos memoria? probablemente
- ¿Tarda menos? Hay que medirlo

¿recursión o iteración?

- Son dos formas de conseguir que un programa haga muchas veces lo mismo con alguna variante
- Toda iteración puede pasar a recursión
  - todo bucle puede convertirse en un método que se llama a sí mismo
  - normalmente es fácil de programar
- Toda recursión puede pasar a iteración
  - todo método recursivo se puede escribir como bucle
  - puede ser difícil de programar

iteración → recursión

```
while (p) { x; }
```

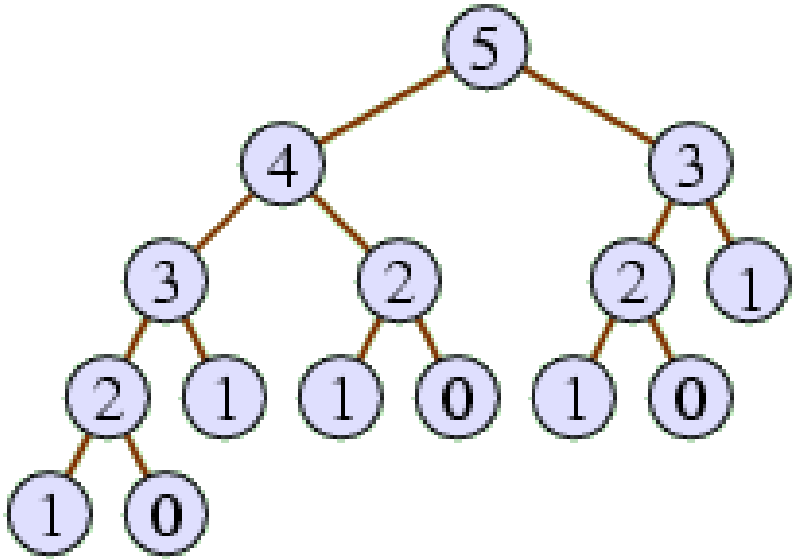
```
void metodo(...) {
  if (p) {
     x;
     metodo(...);
  }
}
```

```
variables = valor inicial;
while (p(variables))  {
    modifica_variables();
}
return variable final
```

```
void metodo(variables) {
  if (p(variables)) {
     modifica_variables();
     return metodo(variables);
  }
  return variable final
}
```

serie de fibonacci

- Leonardo de Pisa, filius Bonacci, (1202)
- 0 1 1 2 3 5 8 13 21 34 55 89 144 ...
- f(n) = f(n-1) + f(n-2)
  f(0) = 0
  f(1) = 1



programación

recursivo

```
int fibo(int n) {
        if (n < 1)
            return 0;
        if (n == 1)
            return 1;
        return fibo(n - 1) + fibo(n - 2);
}
```

iterativo

```
int fibo(int n) {
        if (n < 1)
            return 0;
        int f0 = 0;
        int f1 = 1;
        for (int i = 2; i <= n; i++) {
            int fn = f0 + f1;
            f0 = f1;
            f1 = fn;
        }
        return f1;
}
```

ventajas colaterales

```
int min(int[] data) {
    return min(data, 0, data.length);
}

int min(int[] data, int a, int z) {
    if (z - a < N)
        return min1(data, a, z);
    int m = (a + z) / 2;
    int m1 = min(data, a, m);
    int m2 = min(data, m, z);
    if (m1 < m2)
        return m1;
    else
        return m2;
}
```

```
int min1(int[] data, int a, int z) {
    int m = data[a];
    for (int i = a + 1; i < z; i++) {
        int x = data[i];
        if (x < m)
            m = x;
    }
    return m;
}
```

fallos típicos

- que la condición de terminación de pase por alto
  - ej. igualdad con reales
  - conviene poner condiciones “embudo”
- que los valores no converjan a la situación de parada
- usar variables de clase (static)
  de las que hay 1 para todas las llamadas:
  CAOS

ejercicio 1

- repeated squaring

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

https://en.wikipedia.org/wiki/Exponentiation_by_squaring

ejercicio 2

Función de Ackermann

- A(m, n) =

n+1	si m = 0
A(m-1, 1)	si m > 0 & n = 0
A(m-1, A(m, n-1))	si m > 0 & n > 0

- ejemplos

A(0, 0) = 1	**A(0, 1) = 2**	**A(0, 2) = 3**	**A(0, 5) = 4**	**A(0, 4) = 5**
A(1, 0) = 2	A(1, 1) = 3	A(1, 2) = 4	A(1, 3) = 5	A(1, 4) = 6
A(2, 0) = 3	A(2, 1) = 5	A(2, 2) = 7	A(2, 3) = 9	A(2, 4) = 11
A(3, 0) = 5	A(3, 1) = 13	A(3, 2) = 29	A (3, 3) = 61	A(3, 4) = 125

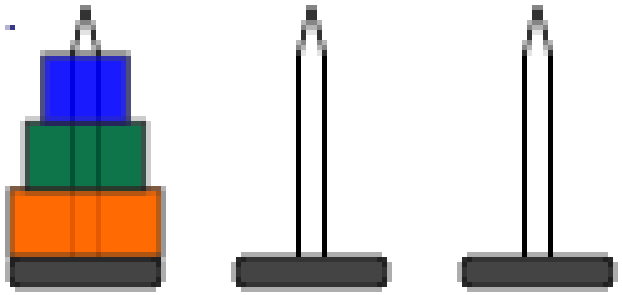
https://en.wikipedia.org/wiki/Ackermann_function

ejercicio 3 (Euclides)

- máximo común divisor recursivo e iterativo
- gcd(a, b) =
  - gcd(b, a) si a < b
  - a si b = 0
  - gcd(b, a%b) si a > b

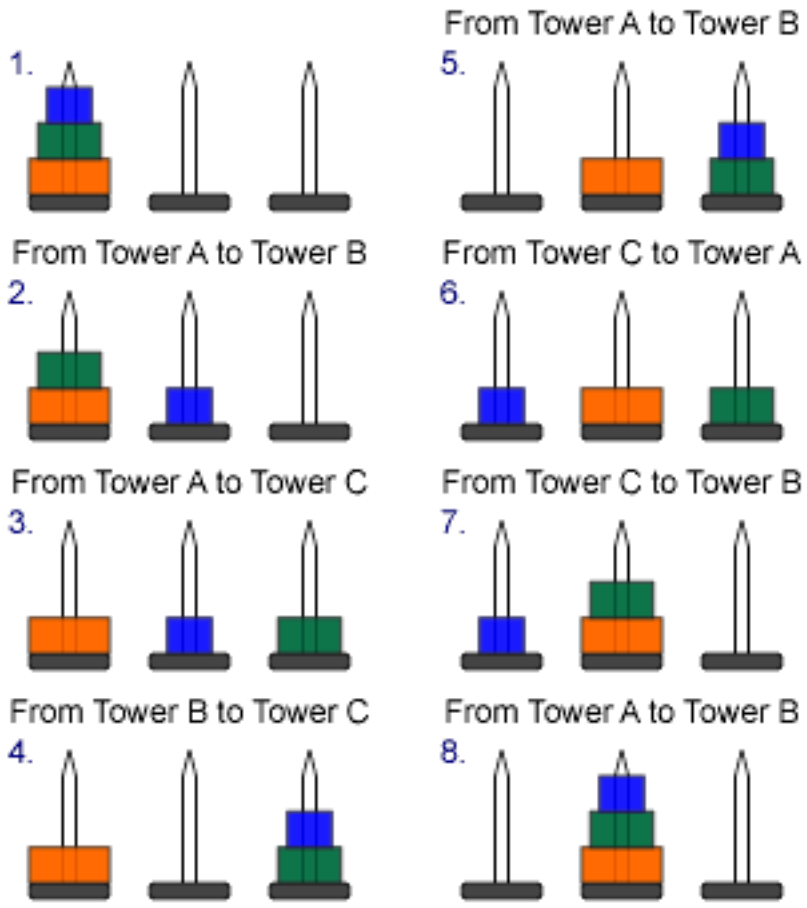
ejercicio 4.1

- torres de hanoi
- hay que mover N discos de la torre A a la torre B de forma que
  - sólo se puede mover un disco cada vez
  - nunca puede haber un disco de mayor tamaño sobre otro de menor tamaño
  - se puede usar una torre[1].



ejercicio 4.2

- solución



ejercicio 4.3

- programar
  mueve(int n, int A, int B, int C)
- si n = 1, se mueve 1 de A a B
- si n > 1
  1. se mueven n-1 de A a C
  2. se mueve 1 de A a B
  3. se mueven n-1 de C a B

ejercicio 5

- Binary search begins by comparing the middle element of the array with the target value. If the target value matches the middle element, its position in the array is returned. If the target value is less than or greater than the middle element, the search continues in the lower or upper half of the array, respectively, eliminating the other half from consideration.

https://en.wikipedia.org/wiki/Binary_search_algorithm

ejercicio 6

- dadas 2 listas ordenadas de String, combinarlas respetando el orden
- List<String> merge(List<String>, List<String> b)

https://en.wikipedia.org/wiki/Merge_sort