



Diseño recursivo de algoritmos

Estructuras de datos y Algoritmos
2011-2012

Felipe Ibañez
felipe.anfurrutia@ehu.es



Agenda

- Definición
- Diseño e implementación
- Cómo funciona
- Tipos de recursividad
- Recursión vs Iteración
- Ejemplos / Problemas
- Técnicas de implementación
- Errores típicos en recursividad



Definición

- Cualquier estructura repetitiva (ciclos) puede ser definido de forma iterativa y de forma recursiva.
 - Un proceso iterativo es aquél que requiere de la repetición explícita de cierta acción.
 - Un proceso es recursivo si está definido total o parcialmente en términos de sí mismo.



Ejemplo Matrushka

- La Matrushka es una artesanía tradicional rusa. Es una muñeca de madera que contiene otra muñeca más pequeña dentro de sí. Esta muñeca, también contiene otra muñeca dentro. Y así, una dentro de otra.





Ejemplo: Factorial iterativo

- Para calcular $4!$, por ejemplo, se puede utilizar un proceso iterativo o uno recursivo.
- De manera iterativa:
 - $4! = 1 * 2 * 3 * 4 = 24$



Ejemplo: Factorial recursivo

- Para definir $4!$ de manera recursiva se tiene que definir en términos de factorial, es decir, en la definición (parte derecha de la igualdad) tiene que aparecer el factorial.
- De manera recursiva:
 - $4! = 4 * 3!$



Ejemplo: Factorial recursivo (2)

$4! = 4 * 3! = 4 * 6 = 24$	Sustituyendo 3!
$3! = 3 * 2! = 3 * 2 = 6$	Sustituyendo 2!
$2! = 2 * 1! = 2 * 1 = 2$	Sustituyendo 1!
$1! = 1 * 0! = 1 * 1 = 1$	Sustituyendo 0!
$0! = 1$	



Observaciones

- Se observa que en el cálculo de 4! fue muy importante el hecho de que 0! es igual a 1.
- ¿Qué hubiera pasado si 0! se calculará como se calculó los demás factoriales?
Nunca se hubiera terminado el proceso, o sea, tendríamos un proceso infinito.



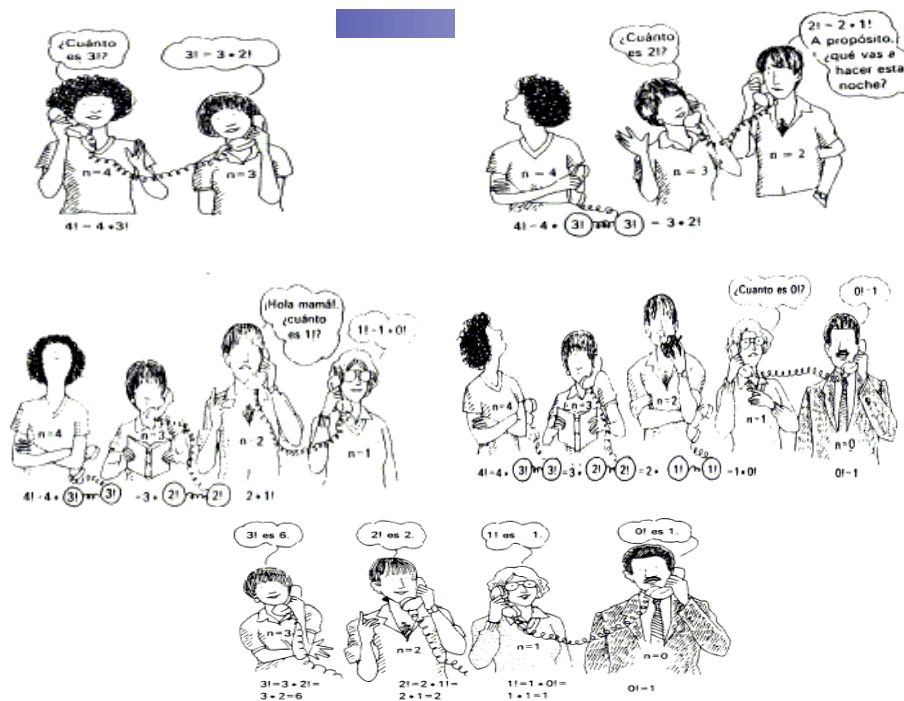
Observaciones (2)

- Entonces, toda definición recursiva debe tener, al menos, una **definición base**. Esta definición base proporciona la solución de **salida de la recursividad**.




Observaciones (3)

- Para calcular $4!$, $3!$, $2!$ y $1!$ Se uso una **definición recursiva**, por lo que se estuvo entrando en recursión varias veces y cuando se alcanzó la definición base se comenzó a salir de recursión.



Conclusiones:

- Un proceso es recursivo si está **definido** total o parcialmente **en términos de sí mismo**.
- Todo proceso recursivo debe tener, al menos: **una definición base** y **una definición recursiva**.



Diseño e implementación de algoritmos recursivos

- La solución recursiva a un problema de repetición se obtiene respondiendo dos preguntas:
 1. ¿Cómo se resuelve el caso más pequeño del problema?
 - La respuesta a esta pregunta debe ser no-recursiva y plantear una condición de salida, es decir, proporcionar la definición base.
 - En el cálculo de factorial, la pregunta sería: ¿cuál es el número más pequeño para el que se puede obtener factorial?
Para $N = 0 \rightarrow n! = 1$



Diseño e implementación de algoritmos recursivos (2)

- La solución recursiva a un problema de repetición se obtiene respondiendo dos preguntas:
 2. ¿Cómo se resuelve un caso general del problema, sabiendo que ya se tiene el caso anterior más pequeño?
Para $N > 0 \rightarrow N * (N-1) !$

Diseño e implementación de algoritmos recursivos (3)

■ Solución del diseño:

- Caso base: si $N = 0 \rightarrow n! = 1$ (definición base)
- Caso general: si $N > 0 \rightarrow n! = n * (n-1)!$ (definición recursiva)

■ Implementación:

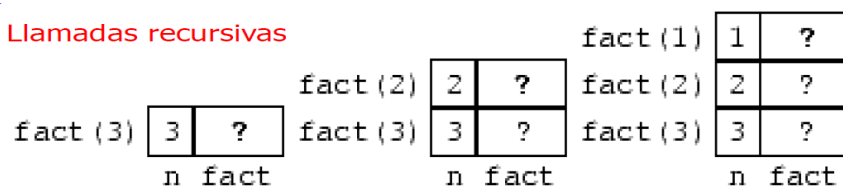
/** Dado un entero no negativo n , devuelve el factorial de n */

```
public int fact (int n) {
    if (n == 0)
        return 1;
    else
        return fact(n - 1) * n;
}
```

Es importante determinar un caso base, es decir un punto en el cual existe una condición por la cual no se requiera volver a llamar a la misma función.

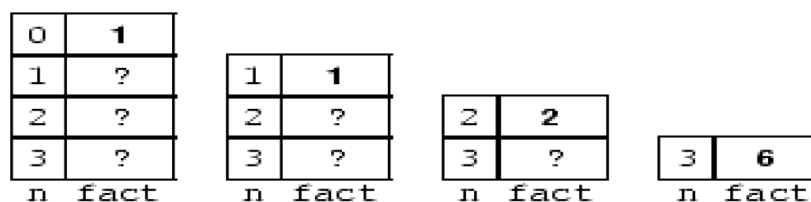
¿Cómo funciona la recursividad?

Llamadas recursivas



- **NOTA:** Cuando un procedimiento recursivo se llama recursivamente a si mismo varias veces, para cada llamada se crean *copias independientes* de las variables y parámetros declaradas en el procedimiento

Resultados de las llamadas recursivas





Tipos de recursividad

■ **recursión directa:**

- ☐ Cuando un procedimiento incluye una llamada a sí mismo

■ **recursión indirecta:**

- ☐ Cuando un procedimiento llama a otro procedimiento y éste causa que el procedimiento original sea invocado



¿Por qué escribir programas recursivos?

- Son mas cercanos a la descripción matemática.
- Generalmente mas fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.



¿Cuándo usar recursividad?

- ☐ Para simplificar el código.
- ☐ Cuando la estructura de datos es recursiva
ejemplo : árboles.

¿Cuándo no usar recursividad?

- ☐ Cuando los métodos usen arrays largos.
- ☐ Cuando el método cambia de manera impredecible de campos.
- ☐ Cuando las iteraciones sean la mejor opción.



Recursión vs. iteración

■ Repetición

- ☐ Iteración: ciclo explícito
- ☐ Recursión: repetidas invocaciones a método

■ Terminación

- ☐ Iteración: el ciclo termina o la condición del ciclo falla
- ☐ Recursión: se reconoce el caso base

■ Observaciones:

- ☐ En ambos casos podemos tener ciclos infinitos
- ☐ Considerar que resulta más positivo para cada problema:
 - la elección entre eficiencia (iteración) o una buena ingeniería de software,
 - la recursión resulta normalmente más natural.



Otros ejemplos de recursividad

- Inversión de una cadena de caracteres
- **Números triangulares**
- Torres de hanoi
- **Calcular la serie de fibonacci**
- Verificar si una cadena de caracteres es palindromo



Ejemplo: inversión de una cadena

- `public String invertirCadena(String s)`



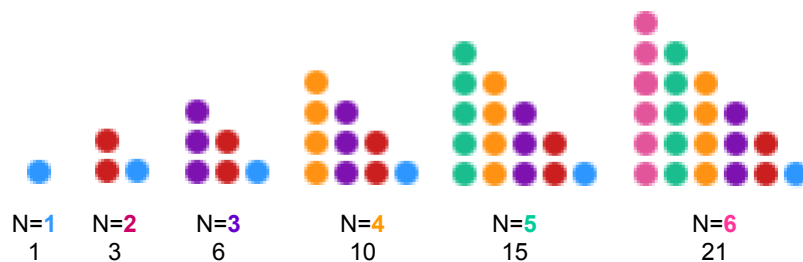
Ejemplo: palindromo

- Un palíndromo es una cadena que se lee (se escribe, en este caso) igual de izquierda a derecha que de derecha a izquierda. Escribir una función que determine cuando una cadena es o no un palíndromo.



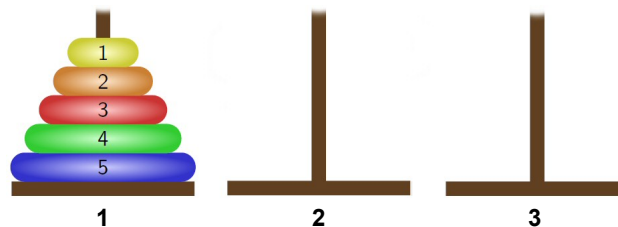
Ejemplo: Números Triangulares

- Valores: 1, 3, 6, 10, 15, 21, ...
¿Cuál es el siguiente valor de la serie?
El termino n se obtiene sumando n al termino anterior
- Los Pitagoríacos (grupo de matemáticos bajo las ordenes de Pitágoras), encontro un mística relación:



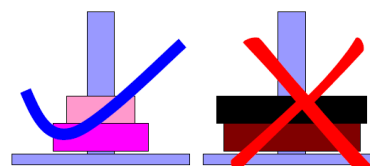
Ejemplo: Torres de Hanoi

- Tenemos tres astas **1, 2 y 3**, y un conjunto de cinco aros, todos de distintos tamaños.
- El enigma comienza con todos los aros colocados en el asta **1** de tal forma que ninguno de ellos debe estar sobre uno más pequeño a él; es decir, están apilados, uno sobre el otro, con el más grande siempre abajo, y encima de él, el siguiente en tamaño y así sucesivamente.



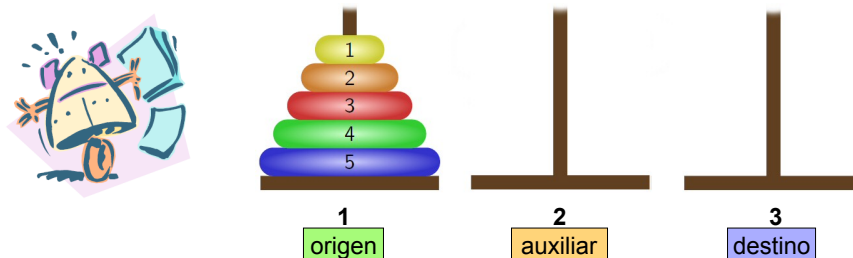
Ejemplo: Torres de Hanoi (2)

- El propósito del enigma es lograr apilar los cinco aros, en el mismo orden, pero en el asta **3**.
- Una restricción es que durante el proceso, puedes colocar los aros en cualquier asta, pero debe apegarse a las siguientes reglas:
 - Solo se puede mover el aro superior de cualquiera de las astas
 - Un aro más grande nunca puede estar encima de uno más pequeño



Ejemplo: Torres de Hanoi (3)

- ¿Cómo resolvemos el problema?
 - Para encontrar cómo se resolvería este problema, debemos ir viendo cómo se resolvería cada caso.



Ejemplo: Torres de Hanoi (4) Resolviendo el problema

- Entonces, por lo que hemos podido ver, el programa podría definirse de la siguiente manera:
 - Si es un solo disco, lo movemos de 1 **origen** a 3. **destino**
 - En otro caso, suponiendo que **n** es la cantidad de aros que hay que mover
 - Movemos los **n-1** aros superiores - es decir, sin contar el más grande- de 1 a 2 (utilizando a 3 como **auxiliar**).
 - Movemos el último aro (el más grande) de 1 **origen** a 3. **destino**
 - Movemos los aros que quedaron en 2 a 3 (utilizando la 1 como **auxiliar**).

Ejemplo: Torres de Hanoi (5) Implementación

```
public void TorresHanoi(int n, int origen, int destino, int auxiliar) {
    if (n > 0) {
        TorresHanoi(n-1, origen, auxiliar, destino);
        System.out.println("Mover disco " + n + " desde " + origen + " a " + destino);
        TorresHanoi(n-1, auxiliar, destino, origen);
    }
}

public static void main(String[] args){
    TorresHanoi(5, 1, 3, 2);
}
```

Ejemplo: Serie de Fibonacci

- Valores: 0, 1, 1, 2, 3, 5, 8, ...
¿Cuál es el siguiente valor de la serie?
El termino n se obtiene sumando los dos terminos anteriores. La fórmula recursiva es: $fib(n) = fib(n-1) + fib(n-2)$
- Diseño:
 - ☐ Caso base: $fib(0) = 0$ y $fib(1) = 1$
 - ☐ Caso recursivo: $fib(i) = fib(i-1) + fib(i-2)$
- Implementación:


```
public static int fib(int n) {
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

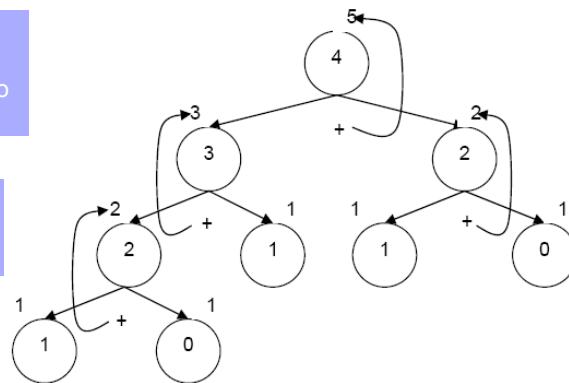
Ejemplo: Serie de Fibonacci (2)

■ Traza del calculo recursivo

La ejecución de Fibonacci(4) gráficamente, no formalmente, la podemos ver de la siguiente manera:

Las llamadas recursivas se van realizando hacia abajo

Y se van terminando hacia arriba



Ejemplo: Serie de Fibonacci (3)

Instrucción de regreso	n	fibonacci	
1	4	3 + 2 = 5	← Resultado
2a	3	2 + 1 = 3	
2a	2	1 + 1 = 2	
2a	1	1	
2b	0	1	
2b	1	1	
2b	2	1 + 1 = 2	
2a	1	1	
2b	0	1	

```

public static int fib(int n) {
    if ( n<=1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
           2a       2b
  
```


Trampas sutiles: Código ineficiente

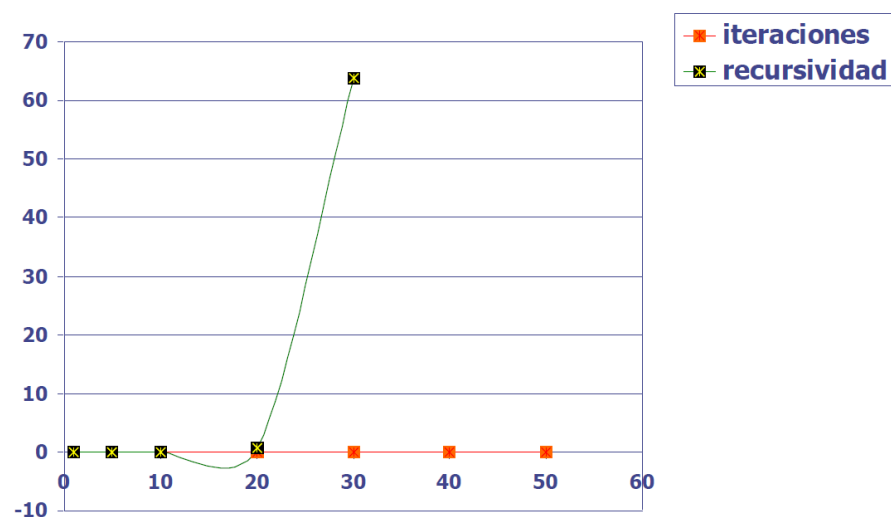
```
public static int fib(int n) {
    if ( n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

fib(100) toma 50 años
en dar el resultado

```
public static int fib (int n) {
    int f1 = 1, f2 = 1, nuevo;
    while (n > 2) {
        nuevo = f1 + f2;
        f1 = f2;
        f2 = nuevo;
        n--;
    }
    return f2;
}
```

fib(100) toma tan sólo unos
microsegundos en dar el resultado

Iteración vs. Recursión: Serie fibonacci





Técnicas de implementación

- Divide y Venceras
- Programación dinámica (dynamic programming)
- Vuelta atrás (Backtracking)



Técnicas: Divide y Venceras

¿En qué consiste ?

■ *Dividir*

- ☐ **Descomponer** el problema a resolver en un nº de subproblemas mas pequeños hasta llegar al caso base

■ *Vencer*

- ☐ **Resolver** sucesiva e independientemente todos los **subproblemas**
- ☐ **Combinar las soluciones** obtenidas de esta forma, para obtener la solución al problema original.



Técnicas: Divide y Venceras

¿Cuándo debe utilizarse ?

- Decimos que un algoritmo recursivo utiliza esta técnica cuando:
 - Contiene **al menos dos llamadas** recursivas (es lo que se denomina recursión en cascada)
 - Ambas llamadas son **disjuntas** (sin superposiciones) para no hacer cálculos redundantes.



Ejemplos

- Algoritmos de ordenación: mergesort y quicksort
- Algoritmo de búsqueda binaria:

```
public static boolean busBin( int[] a, int limIzq, int limDrch, int valor){
    if (limIzq == limDrch) return valor == a[limDrch];
    else {
        int m = (limIzq + limDrch) / 2;
        if (valor == a[m]) return true;
        else {
            if (valor > a[m])
                return busBin(a, m+1, limDrch, valor);
            else
                return busBin(a, limIzq, m-1, valor);
        }
    }
}
```



Problema

■ Obtención de la secuencia de suma máxima

Dada una sucesión de enteros (posiblemente negativos) A_1, A_2, \dots, A_N encontrar (e identificar la secuencia correspondiente a) el valor máximo de $\sum_{k=i}^j A_k$. Consideramos que la secuencia contigua de suma máxima es la vacía, de suma cero, si todos los enteros son negativos.



Diseño iterativo

- Realizar una búsqueda exhaustiva:
 - Calculamos la suma de cada posible subsecuencia y elegimos la máxima
 - Tendremos $O(N^2)$ subsecuencias, por lo tanto, el tiempo de ejecución será $O(N^2)$
 - (ver MaxSumTest.java)
- Cómo podemos mejorar?



Diseño recursivo

- Ejemplo: $A = \{4, -3, 5, -2, -1, 2, 6, -2\}$
- Dividiremos el problema en dos partes iguales
- Entonces la secuencia máxima puede aparecer en una de estas tres formas:
 - Caso 1: está totalmente incluida en la 1ª mitad
 - Caso 2: está totalmente incluida en la 2ª mitad
 - Caso 3: comienza en la primera mitad, pero termina en la segunda
- Tiempo de ejecución: $O(N \cdot \log N)$



Técnica: programación dinámica

- Resuelve los subproblemas generados por un planteamiento recursivo, de forma no recursiva
- Guardando los valores computados en una tabla



Problema

■ Cambio de monedas

Para una divisa con monedas C_1, C_2, \dots, C_N (unidades), ¿cuál es el **mínimo número de monedas** que se necesitan para devolver K unidades?



Diseño iterativo

- Ejemplo: divisas= 1, 5, 10, 25 y obtener: 63
- Utilizando un **algoritmo devorador**: toman decisiones locales en cada paso.
 - En cada paso “coje todo lo que puedas”
 - Resultado: 2 de 25, 1 de 10 y 3 de 1 = 6
- Es una forma simple de hacer las cosas, pero no siempre funciona correctamente.
 - ¿y si también existe la moneda de 21?



Diseño recursivo

- Condición al problema: debe haber moneda de 1
- Estrategia simple para reunir K unidades:
 - Si con una moneda ya podemos devolver el cambio solicitado, ésta (1) es la cantidad mínima de monedas
 - En caso contrario, para cada posible valor i podemos calcular de forma independiente el número mínimo de monedas que se necesitan para reunir i y $K - i$ unidades, y elegimos el i que minimice la suma de ambos



Diseño recursivo (2)

- Ejemplo: $K=63$,
- Ejemplos de los subproblemas:
 - $1+62 \rightarrow 1+4 = 5$ monedas,
 - $2+61 \rightarrow 2+4 = 6$ monedas,
 - ...,
 - $21+42 \rightarrow 1+2 = 3$ monedas,
 - $31+32 \rightarrow 2+3 = 5$ monedas
- Problema: el calculo de algunos subproblemas se vuelve a repetir, calculándolo recursivamente. Igual que en el caso de Fibonacci
- Tiempo de ejecución: $O(2^N)$



Diseño recursivo (3)

- Estrategia alternativa:
 - Reducir recursivamente el tamaño del problema especificando una primera moneda
- Ejemplo: $K = 63$ y $M = 1, 5, 10, 21, 25$
 - 1 de 1 + num. monedas para 62, calc. recur.
 - 1 de 5 + num. monedas para 58, calc. recur.
 - 1 de 10 + num. monedas para 53, calc. recur.
 - 1 de 21 + num. monedas para 42, calc. recur.
 - 1 de 25 + num. monedas para 38, calc. recur.
- Mejora: se realizan 5 llamadas recur. y no K
- Problema: se siguen repitiendo subproblemas.



Diseño recursivo: prog. dinámica

- El truco: ir guardando los resultados de los subproblemas en una tabla
- Funciona?: la solución a un problema grande depende solamente de las soluciones de problemas más pequeños
- Ejemplo: ir calculando las soluciones óptimas para 1 unidad, 2, 3, y así sucesivamente hasta K , probando con las N monedas
- Tiempo de ejecución: $O(N.K)$
- (ver MakeChange.java)



Técnica: Vuelta atrás

- Los algoritmos de vuelta atrás usan la recursión para probar sistemáticamente todas las posibilidades
- Se almacena el resultado que se obtendrá después de probar el caso y entre todas las probadas, se selecciona la mejor o las que se pueden realizar



Errores típicos en recursividad

1. Olvidarnos del caso base
2. No asegurarse que la llamada recursiva nos lleva hacía un caso base
3. Solapamiento en las llamadas recursivas → tiempo de ejecución exponencial
4. Asumir que el tiempo de ejecución es lineal.
Los algoritmos recursivos se analizan utilizando formulas recursivas.
5. Utilizar la recursión, en vez de un simple bucle es un mal estilo