Tema 5- Diseño Recursivo y Eficiente

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Objetivos Principales

- Introducir la recursión como una herramienta de diseño alternativa a la más conocida estrategia iterativa.
- Comparativa de métodos recursivos frente a sus versiones iterativas.
- Aprender las ventajas e inconvenientes del uso de la recursión a la hora de resolver un problema concreto.
- Objetivos específicos:
  - Estudio del Esquema General Recursivo y los diferentes tipos de recursión.
  - Estudiar la Complejidad Temporal de los métodos recursivos.
  - Estudiar la estrategia Divide y Vencerás (DyV).

Bibliografía Básica

- Weiss, M.A. Estructuras de datos en Java. Adisson-Wesley, 2000. Capítulo 7, apartados del 7.1 al 7.4 y Capítulo 8, apartados del 8.5 al 8.7

- Bibliografía Complementaria:
  - Brassard G., Bratley P. Fundamentos de Algoritmia. Prentice Hall 1997. Capítulos 4 y 7.
  - F.J. Ferri, J.V. Albert, G. Martín. Introducció a l'anàlisi i disseny d'algorismes. Universitat de València, 1998. Capítulo 3.

Introducción a la Recursión

- Los términos Inducción, Inferencia o Recursión denominan una estrategia de resolución de problemas:
  - A partir de un conjunto finito de casos sencillos de un problema para los que se conoce la solución, se realiza una **hipótesis** de cuál es la solución general de dicho problema:
    - La **hipótesis** para el caso general se expresa en términos de la(s) solución(es) del mismo problema para el(los) caso(s) más sencillo(s).
    - Debe ser **validada** o demostrada mediante un proceso de prueba, sin el cual no deja de ser más que una hipótesis.
- En general se denomina recursivo a cualquier ente (definición, proceso, estructura, etc.) que se define en función de sí mismo.
  - Ejemplo: La clase NodoLEG<E>

Ejemplo de Recursión: Factorial

- El factorial de un número n se define como:
  - n! = n * (n-1) * (n-2) * … * 1
- Se puede calcular de manera recursiva asumiendo los siguientes casos:
  - Caso Base (n = 0): factorial(n) = 1
  - Caso Recursivo (n > 0): factorial(n) = n*factorial(n-1)
- Transcripción algoritmica:

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

- Por simplicidad, se ha omitido el tratamiento de errores (el factorial no está definido sobre números negativos).

Corrección de un Algoritmo Recursivo

- Para comprobar que un algoritmo recursivo es **correcto** se debe realizar:
  1. Prueba de **Terminación**:
     - Se debe demostrar que el algoritmo termina en un número finito de pasos, tanto para el caso base como para el caso recursivo.
  2. Prueba de la **Hipótesis de Inducción**:
     - Se debe demostrar la corrección de la hipótesis de inducción, es decir, que el algoritmo hace lo que tiene que hacer.
- La prueba de terminación garantiza que el algoritmo no realiza un número infinito de llamadas recursivas.
  - Es decir, que en algún instante dado, ¡acabará!

Estrategias para la Prueba de Terminación

1. **Enumerar las llamadas** originadas por la principal.
   - factorial(n) → factorial(n-1) → factorial(n-2) → factorial(0)
   - Cálculo del número de llamadas vía ecuación de recurrencia:
     - numLlamadas(N = 0) = 0
     - numLlamadas(N > 1) = 1 + numLlamadas(N-1)
     - Resolviendo por substitución: N llamadas recursivas (finito)
2. Demostrar que los valores de los **parámetros** de las sucesivas llamadas conforman una sucesión ordenada decrecientemente que **converge** al valor para el que se alcanza el **caso base**.
   - Si la talla del problema decrece al menos en una unidad en cada llamada recursiva al cabo de un tiempo finito se alcanzará el caso base.

Esquema General Recursivo

- El Esquema General Recursivo es la estructura principal de cualquier algoritmo recursivo.

```
public static TipoResultado metodoRecursivo(TipoDatos x) {
  TipoResultado resMetodo, resLlamada1, resLlamada2, ... , resLlamadaN;
  if ( casoBase(x) )    resMetodo = solucionBase(x);
  else  {
         resLlamada1 = metodoRecursivo( anterior1(x) );
         resLlamada2 = metodoRecursivo( anterior2(x) );
         ...
         resLlamadaN = metodoRecursivo( anteriorN(x) );
         resMetodo = combinar(x , resLlamada1,... , resLlamadaN);
  }
  return resMetodo;
}
```

Mapear el algoritmo diseñado de cálculo del factorial al Esquema General Recursivo.

Taxonomía de la Recursión

- Los algoritmos recursivos se **clasifican** en función del:
  - Número de llamadas recursivas en el caso general.
  - Utilización o no del método **combinar**
1. Recursión **Lineal**: Cuando en el caso general se produce una única llamada recursiva.
   a) **Final**: El resultado de la llamada recursiva es directamente el resultado del método recursivo (NO se combina el resultado).
   b) **No Final**: El resultado de la llamada recursiva no es el resultado del método recursivo (SÍ se combina el resultado).
2. Recursión **Múltiple**: Cuando en el caso general se produce más de una llamada recursiva.

Ejemplo de Recursión Lineal Final

- Cálculo del máximo común divisor entre dos números

```
static int  maximoComunDivisor(int n , int m)    {
 int resMetodo, resLlamada;
 if    ( n == m )    resMetodo = n ;
 else {
     if (n> m)  resLlamada = maximoComunDivisor(n-m,m);
     else          resLlamada = maximoComunDivisor(n,m-n);
     resMetodo = resLlamada;
    }
return resMetodo;
}
```

¿Cuántas llamadas recursivas se producen en el caso general de la recursión?

- Únicamente se produce una llamada recursiva en el caso general y el resultado del método es directamente el resultado de la llamada recursiva.
  - Se trata de recursión lineal final

Ejemplo de Recursión Lineal No Final

- Cálculo del factorial de un número.

```
static int factorial(int n){
    int resMetodo, resLlamada;
    if   (n==0)      resMetodo = 1 ;
    else {
       resLlamada = factorial(n-1) ;
       resMetodo = n * resLlamada ;
         }
    return resMetodo;
}
```

¿Cuántas llamadas recursivas se producen en el caso general de la recursión?

¿El resultado del método es directamente el resultado de la llamada recursiva?

- Únicamente se produce una llamada recursiva en el caso general. El resultado del método se combina (multiplica) para generar el resultado de la llamada recursiva
  - Se trata de Recursión Lineal No Final.

Ejemplo de Recursión Múltiple

- Cálculo de la sucesión de Fibonacci
  - fibonacci(n) calcula el término n-ésimo de la serie de Fibonacci

```
static  int  fibonacci(int n) {
   int resMetodo, resLlamada1 , resLlamada2;
   if ( n > 1 )  {
    resLlamada1 = fibonacci(n - 1) ;
    resLlamada2 = fibonacci(n - 2) ;
    resMetodo = resLlamada1 + resLlamada2;
   }
   else   resMetodo = 1 ;
   return resMetodo;
}
```

¿Cuántas llamadas recursivas se producen en el caso general de la recursión?

¿El resultado del método es directamente el resultado de la llamada recursiva?

- Dos llamadas recursivas. El resultado del método se combina (suma) para obtener el resultado de la llamada.
  - Se trata de Recursión Múltiple

Secuencia de Llamadas: Recursión Lineal Final

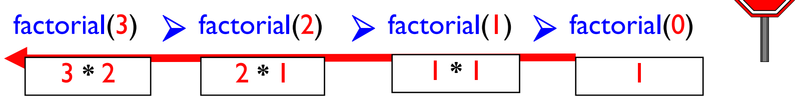
- Una llamada a maximoComunDivisor(25,15) genera la siguiente secuencia de llamadas:
  - mcd(25,15) → mcd(10,15) → mcd(10,5) →mcd(5,5) → Valor 5

```
static int  maximoComunDivisor(int n , int m)    {
  if ( n == m ) return n ;
  else {
       if (n > m)  return maximoComunDivisor(n-m, m);
       else           return maximoComunDivisor(n, m-n);
}
```

- La recursión lineal final implica que al alcanzar el caso base se llega automáticamente al final de la ejecución del algoritmo recursivo.

Secuencia de Llamadas: Recursión Lineal No Final

- factorial(3) genera la siguiente secuencia de llamadas:

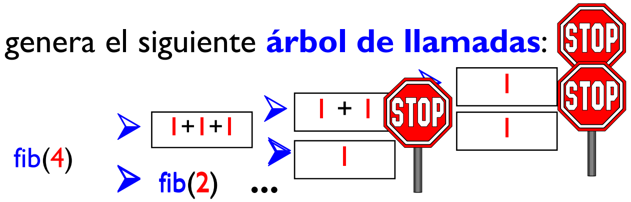



```
public static int factorial(int n){
    if ( n == 0 ) return 1; else return n * factorial(n-1);
}
```

- El resultado final indica que factorial(3) = 6.
- La recursión lineal no final implica deshacer en orden inverso la secuencia de llamadas generadas.
  - Permite calcular los resultados a devolver por cada llamada recursiva que estaba pendiente de ejecución.

Árbol de Llamadas: Recursión Múltiple (I)

- fibonacci(4) genera el siguiente **árbol de llamadas**:

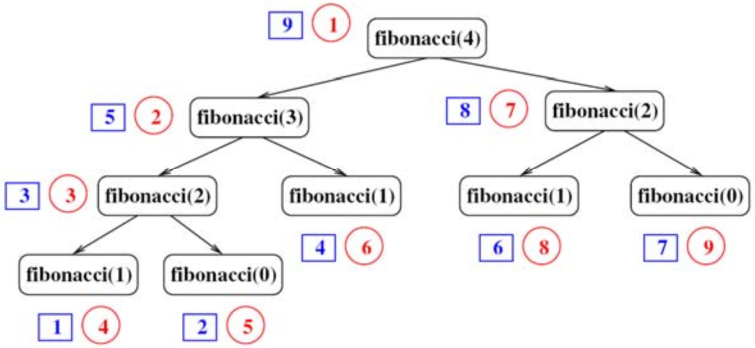



```
static int fibonacci(int n) {
    if ( n <= 1 )  return 1;
    else return fibonacci(n - 1) + fibonacci(n - 2) ;
}
```

- El árbol de llamadas se va generando conforme se realizan las llamadas recursivas.
  - Cada nodo del árbol tendrá tantos hijos como llamadas recursivas provoque.

Árbol de Llamadas: Recursión Múltiple (II)

- Árbol de llamadas tras una invocación a fibonacci(4):



- Cuadrados Azules: Orden en el que finalizan las llamadas recursivas.
- Círculos Rojos: Orden en el que se producen las llamadas recursivas.

Ejecución de un Método Recursivo

- La ejecución de un método requiere ocupar una zona de la memoria RAM que está compuesta por:
  - Zona reservada para el código de la Máquina Virtual Java
  - Zona de memoria estática, reservada para almacenar variables globales.
  - Zona de memoria dinámica (montículo o heap):
    - Zona donde se almacenan los nuevos objetos (creados mediante *new*).
    - Zona de registros de activación (pila o stack).

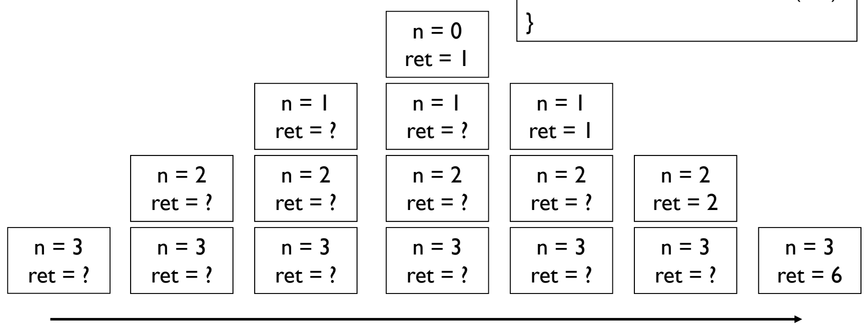
Registros de Activación

- Cada llamada utiliza un **Registro de Activación** que:
  - Posee su propio conjunto de variables locales y parámetros
  - Mantiene una referencia al código donde se debe seguir ejecutando tras el return.
  - Se **apila** sobre los registros de activación existentes.
  - Se **desapila** tras el return.
- Existen tantos R.A. como llamadas pendientes.
  - De todos ellos, en cada momento **sólo hay uno activo**, el que está en el tope de la pila.
- La comunicación entre las sucesivos Registros de Activación (llamadas) debe hacerse por medio de los parámetros.

Ejemplo de Gestión de Registros de Activación

- Cálculo de factorial(3)

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



Evolución de la Pila de Registros de Activación

Recursión vs Iteración

- La recursión requiere una sobrecarga (espacial y temporal) por la gestión de los registros de activación.
- Demasiadas llamadas anidadas desbordan la pila de recursión (StackOverflowError).
- Una forma sencilla de provocar la excepción es con el siguiente extracto de código:
  - void a(){a();}
- A veces, es posible obtener un algoritmo equivalente iterativo mediante la transformación recursivo-iterativa.
  - Fácil de obtener para métodos con recursión lineal final.
  - Más compleja para otros esquemas de recursión

Calculando el Límite de la Pila de Recursión

```
public static void desbordaPila(int i){
        System.out.println("Estoy en el nivel " + i);
        desbordaPila(i+1);
}
```

- Muestra por pantalla el número de llamadas recursivas acumuladas hasta que se desborda la pila de recursión provocando un StackOverflowError.
  - Límite computado: 5872 llamadas recursivas anidadas . Este número depende del tamaño de cada registro de activación.

```
public static void desbordaPila(int i){
        desbordaPila(i+1);
        System.out.println("Estoy en el nivel " + i);
}
```

¿Este código obtiene el mismo resultado?

Cálculo del Factorial: Recursivo vs Iterativo

- Versión Recursiva:

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

- Versión Iterativa:

```
static int factorial(int n){
    int res = 1;
    for (int i = 1; i <=n ; i++) res = res*i;
    return res;
}
```

- Coste Espacial Versión Iterativa: Dos variables.
- Coste Espacial Versión Recursiva: Depende del valor de n (Registros de Activación)

Máximo Común Divisor: Recursivo vs Iterativo

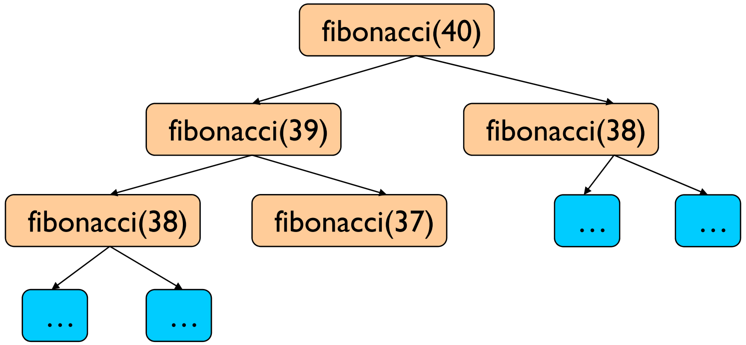
- Versión Iterativa

```
static int maximoComunDivisorIterativo(int N, int M) {
  int n = N, m = M;
  while ( n != m )
     if ( n > m ) n = n - m;
     else m = m - n;
  return n;
}
```

- Coste Espacial MCD (Versión Iterativa): 2 variables (independiente del valor de n y m).
- Coste Espacial MCD (Versión Recursiva): Espacio requerido proporcional al número de registros de activación en la pila.

Transformación Recursivo-Iterativa: Fibonacci

numLlamadas(N≥3) = numLlamadas(N - 2) + numLlamadas(N - 1)



- Se repiten muchos cálculos en llamadas recursivas idénticas.

Transformación Recursivo-Iterativa: Fibonacci

```
static  int  fibonacciIterativo(int n)    {
   int f1=0, f2=1;
   for(int i=0; i<n; i++) {
        int aux = f1;
        f1 = f2;
        f2 = aux + f2;
    }
   return f1;
}
```

- Demasiada recursión ocasiona un coste excesivo, aunque el algoritmo queda mejor expresado de forma recursiva.
- Regla General: Aplicar la estrategia recursiva sólo a la resolución de problemas lo suficientemente complejos.