Recursividad

1. Definición
2. Procedimientos recursivos
3. Mecánica de recursión
4. Transformación de algoritmos recursivos a iterativos
5. Recursividad en el diseño
6. Complejidad de los algoritmos recursivos

Definición

- Un método es recursivo si esta parcialmente definido en términos de sí mismo.
- Un método recursivo permite la definición de soluciones más cortas y eficientes.
- Ejemplo: definición de la función factorial
  - N! = 1 para N=0 o N=1
  - N! = N*(N-1)! para N>1

Tipos de recursión

- **Directa.** El método contiene al menos una llamada a sí mismo.
- **Indirecta.** Existe una secuencia de llamadas a métodos donde al menos una de estas llega al método inicial.

```
// recursión directa
   public static int factorial(int n) {
       if (n<=1) return 1;
       else return n*factorial(n-1);
   }
```

```
// recursion indirecta
   public static int que(int n) {
       if (n <= 1) return n;
       else
           if (n mod 2 ==0) return magia((int)n/2);
           else return 1+magia(n*3+1);
   }
 public static int magia(int n) {
       if (n==1) return 1;
       else
           if ((n mod 2)==1) return que(n-1);
           else return que((int)n/2);
   }
```

```
// recursión directa
   public static int factorial(int n) {
       if (n<=1) return 1;
       else return n*factorial(n-1);
   }
```

~~factorial(1)~~	n=1	1
~~factorial(2)~~	n=2	2
~~factorial(3)~~	n=3	**6**
factorial(4)	n=4	24
factorial(5)	n=5	5*24

System.out.println(factorial(5));

```
// recursion indirecta
   public static int que(int n) {
      if (n <= 1) return n;
      else
          if (n % 2 ==0) return magia((int)n/2);
          else return 1+magia(n*3+1);
   }
 public static int magia(int n) {
      if (n==1) return 1;
      else
          if ((n % 2)==1) return que(n-1);
          else return que((int)n/2);
   }
```

que(1) n=1 1
magia(2) n=2 que(1)
que(4) n=4 magia(2)

System.out.println(que(5));

Procedimientos recursivos

- Definir una expresión en la gramática de un lenguaje de programación.
- Visualizar todos los archivos almacenados en un disco.
- Mostrar a todos los miembros de una familia representados en un árbol genealógico.
- Sumar todos los elementos de una lista de datos.
- Etc.

10,4,5,2,7,9,1	10+suma(n-1)
4,5,2,7,9,1	4+suma(n-1)
5,2,7,9,1	
..	
1	
.	

Gramática de Java

http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.7

Block:
{ *BlockStatements* }
BlockStatements:
BlockStatement
BlockStatements BlockStatement
BlockStatement :
LocalVariableDeclarationStatement
ClassDeclaration
Statement

Statement:
Block
if *ParExpression Statement [*else *Statement]*
for (*ForInitOpt* ; *[Expression]* ; *ForUpdateOpt*)
Statement
while *ParExpression Statement*
do *Statement* while *ParExpression* ;
try *Block (Catches* | *[Catches]* finally *Block)*
switch *ParExpression* {
SwitchBlockStatementGroups }
synchronized *ParExpression Block*
return *[Expression]* ;
throw *Expression* ;
break *[Identifier]*
continue *[Identifier]* ;
ExpressionStatement
Identifier : *Statement*

Mecánica de recursión

- Un método recursivo contiene dos **elementos básicos** que son fundamentales para su funcionamiento.
  - **Caso Base:** Existe al menos una solución para algún valor determinado.
  - **Progreso:** Cualquier llamada a si mismo debe progresar (acercarse) a un caso base.

Define los elementos básicos de los ejemplos de la lámina anterior.

Demostración por *inducción matemática*

- La inducción matemática es una técnica para demostrar teoremas.
- Esta técnica puede ser usada para demostrar que los algoritmos recursivos funcionan correctamente.
- Algunas aplicaciones son demostraciones de teoremas que cumplen los números enteros positivos.

Ejemplo

- La suma de los primeros n números enteros positivos está dada por los siguientes elementos básicos:
  - Caso base: 1 cuando n=1
  - Progreso: n + suma de los primeros n-1 enteros positivos

```
// suma de enteros recursiva
    int suma(int n){
        if (n==1) return 1;
        else return n + suma(n-1);
    }
```

Fórmula matemática $\sum_{i=1}^{n} i = n(n+1)/2$

Demostración por inducción matemática

1(1+1)/2
1(2)/2
2/2
1

1. Demostrar el caso base: es obvio que cuando n=1 el resultado=1
2. Asumir la hipótesis como válida para todos los valores de n>1

$$\sum_{i=1}^{n} i = n(n+1)/2$$

3. Demostrar que la fórmula es válida para n+1

$$\sum_{i=1}^{n+1} i = n+1 + \sum_{i=1}^{n} i$$

$$n+1 + \sum_{i=1}^{n} i = (n+1) + n(n+1)/2$$

$$= n+1 + (n^2 + n)/2$$

$$= (2n + 2 + n^2 + n)/2$$

$$= (n^2 + 3n + 2)/2$$

$$= (n+1)(n+2)/2$$

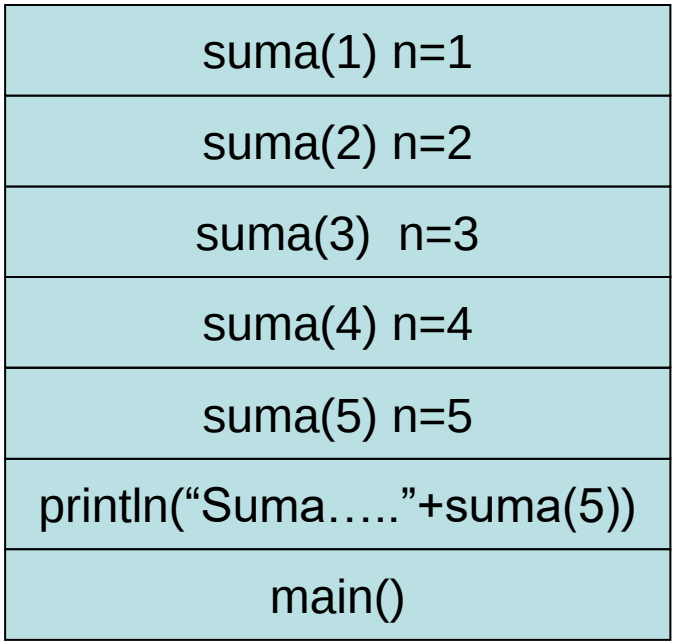
$$\sum_{i=1}^{n+1} i = (n+1)(n+1+1)/2$$

$$= (n+1)(n+2)/2$$

Implementación de la recursión

- En muchos lenguajes de programación, las funciones, procedimientos o métodos recursivos se solucionan (ejecutan) mediante una **pila de registros de activación.**
- Un registro de activación de un método contiene el estado actual de todas las variables definidas en él.

```
Class SumaEnteros
// suma de enteros recursiva
    static int suma(int n){
        if (n==1) return 1;
        else return n + suma(n-1);
    }
   public static void main(String[] args){
        System.out.println("Suma de los primeros 5 números enteros "+suma(5));
   }
}
```



Pila de registros de activación de los métodos

Demasiada recursión??

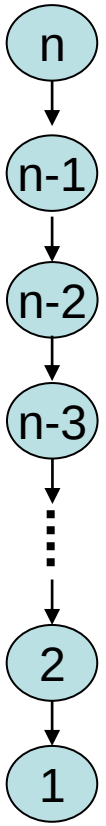
- La ejecución de métodos recursivos consume tiempo y espacio además de limitar el valor de n para el cual se puede ejecutar el programa.
- La recursión debe evitarse si es posible usar un ciclo repetitivo.
- Una función recursiva para calcular los números de Fibonacci realizará una multitud de cálculos repetidos.

```
// Funcion fibonacci
   public static int fib(int n) {
       if (n==1) return 0;
       else if (n==2) return 1;
              else return fib(n-1)+fib(n-2);
   }
```

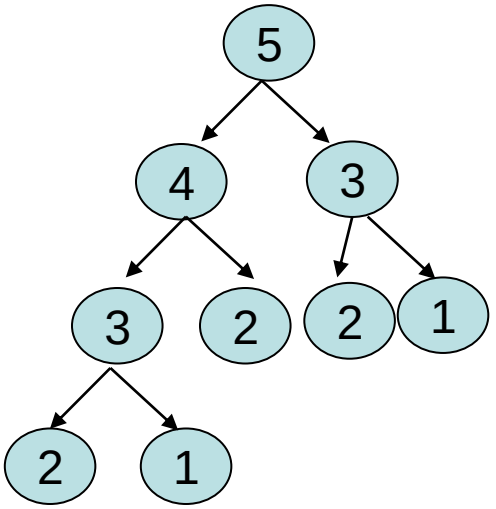
Construye el árbol de llamadas para fib(5)

Transformación de algoritmos recursivos a iterativos

- Cuando una función realiza una sola llamada recursiva, el árbol que representa su ejecución se muestra como una cadena donde cada vértice tiene un solo hijo.
- Este árbol puede convertirse en un programa iterativo que ahorra espacio y tiempo de ejecución.

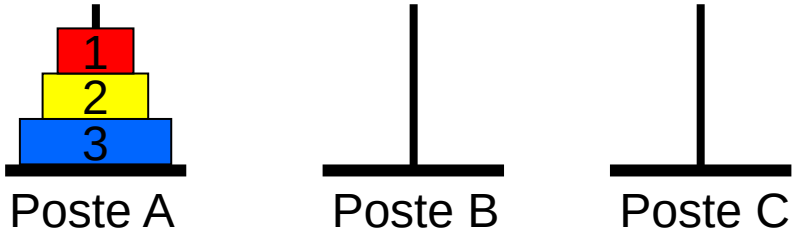


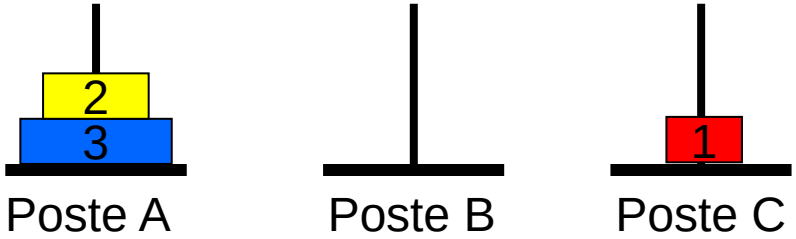
- El método para calcular los números de Fibonacci genera una estructura de árbol de dos hijos, no de una cadena.
- Soluciones para transformar el método:
  - Solución 1.
    - Mantener la información calculada hasta el momento y utilizarla si es necesario.
    - Calcular y almacenar la información que no está en la tabla
  - Solución 2.
    - Escribir un método iterativo que use variables temporales que “muevan” sus valores entre ellas para calcular nuevos números.

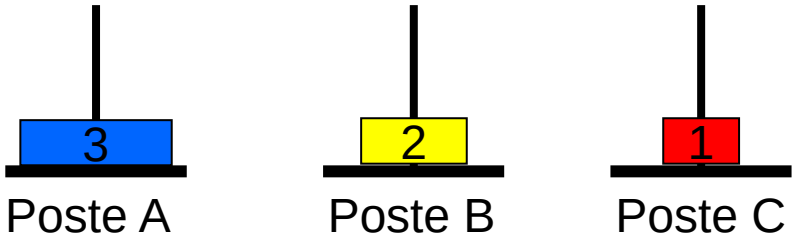


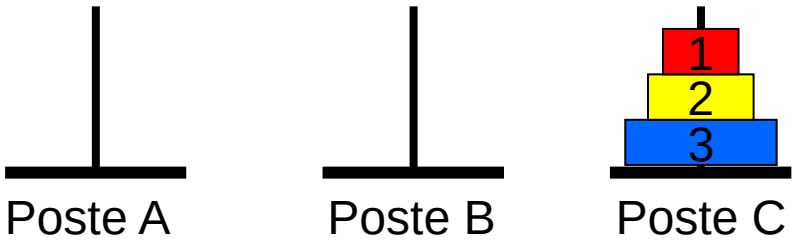
Ejemplo: Las torres de Hanoi

- Juego definido en el siglo XVII en Europa.
- Elementos:
  - tres postes.
  - n discos de diferentes tamaños.
- Reglas:
  - Mover los n discos del poste A al poste C usando el poste B como intermediario.
  - Solo se puede mover un disco a la vez.
  - Ningún disco puede colocarse encima de otro más pequeño.









Como se puede mover el disco más grande?

Mover primero los discos mas pequeños dejando libre el poste final.

```
mueve (discos, inicio, ayuda, final){
  if (discos > 0) {
     mueve(discos-1, inicio, final, ayuda)
     S.o.p("mueve de "+inicio +" a "+ayuda)
     mueve(discos-1, final, ayuda, inicio)
  }
}
```

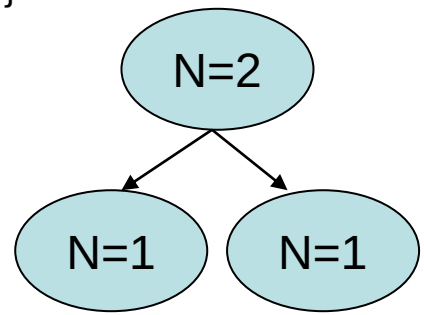
Ejercicio

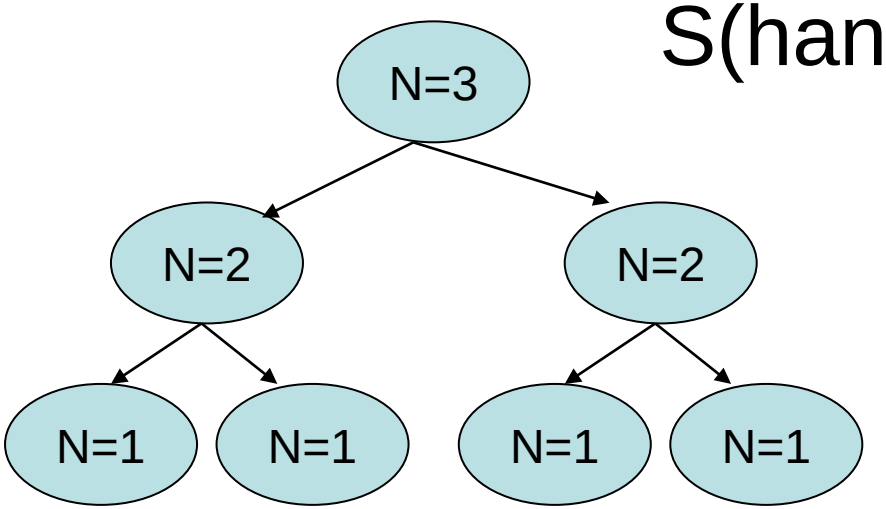
- Cuantos movimientos de discos se requieren para:
  - Mover 1 disco?
  - Mover 2 discos?
  - Mover 3 discos?
  - Mover 4 discos?
  - Mover n discos?

```
public class Hanoi{
    static int contar =0;
    public static void mover (int disco, int a, int b, int c){
        contar++;
        if (disco == 1)
            System.out.println("Mueve el disco "+disco +" del poste: "+a + " al poste: "+b);
        else{
                mover(disco-1, a,c,b);
                System.out.println("Mueve el disco "+disco +" del poste: "+a + " al poste: "+b);
                mover(disco-1, c,b,a);
        }
    }
    public static void main(String[] args){
        System.out.println("Inicia el movimiento de "+ 3 +" discos ");
        mover(3,1,3,2);
        System.out.println("Total de entradas a Mover "+contar);
    }
}
```

$T(hanoi) = \boldsymbol{O}(2^n)$

$S(hanoi) = \boldsymbol{O}(n)$





Análisis de complejidad

- El análisis de complejidad de tiempo y espacio de algoritmos recursivos depende de dos factores:
  - La profundidad (el número de niveles) de las llamadas recursivas hechas antes de llegar a la terminación. A mayor profundidad, mayor espacio y tiempo de ejecución.
  - La cantidad de recursos consumidos en cada nivel de la recursión.

- La ejecución de un programa se puede diagramar trazando la ruta de ejecución y creando una jerarquía con la llamada inicial en el tope.
- El diagrama resultante puede utilizarse para analizar el tiempo y el espacio del algoritmo.

Ejercicios

- Sumar números en un arreglo (usar [ ])
- Sumar elementos de una lista (usar ArrayList)
- Impresión de datos de un arreglo (respetar orden)
- Elevar a una potencia un numero
- Convertir decimal a binario MCD
- Función de Ackerman
- Números de Catalán
- Búsqueda binaria
- Juegos
- Etc.

Para revisión ANTES del buscaminas