RECURRENCIA
RECURSIÓN o
RECURSIVIDAD

- **Definición**
  - Definición de diccionario

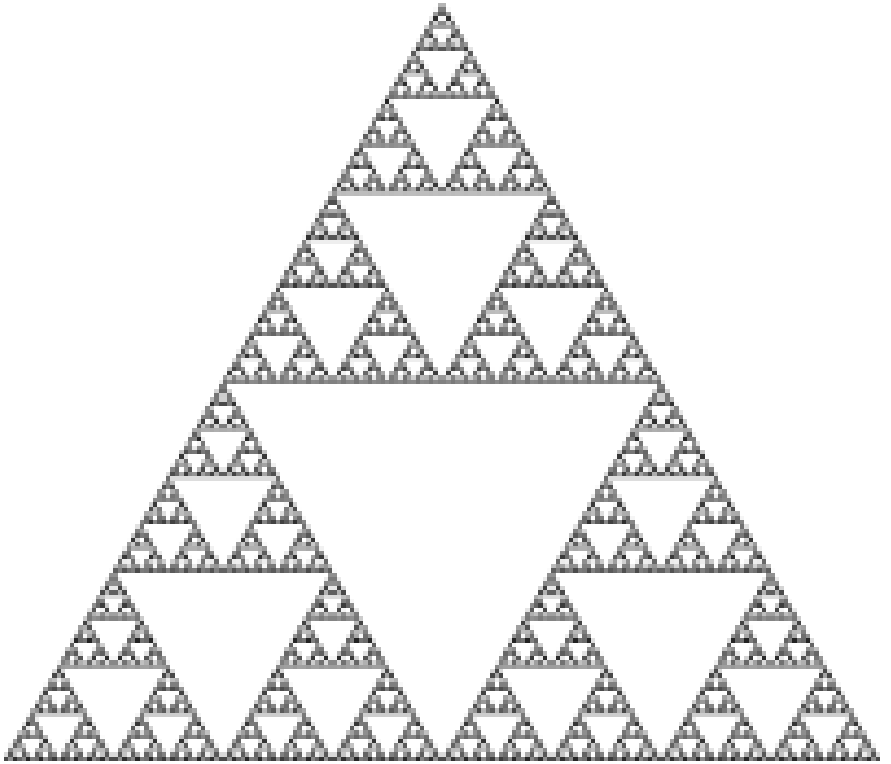
    Recursividad: Véase *Recursividad*

Definición

- Técnica que para resolver problemas basándose en la propia definición del mismo para su resolución.

- Se apoya en el concepto de “Divide y vencerás”, donde se pretende resolver un problema complejo a través de la resolución de subproblemas (relacionados con el original) más simples.

$$\phi = 1 + \frac{1}{\phi} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}$$



- **Ejemplos**
  - Función factorial

```
n! = 1          si n=0
n*(n-1)!        si n>0
```

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

```
 factorial (4) =
        4 * factorial(3) =
        4 * 3 * factorial(2) =
        4 * 3 * 2 * factorial (1) =
        4 * 3 * 2 * 1 * factorial(0) =
        4 * 3 * 2 * 1 * 1 = 24
```

Ejemplos - Iterativo

- Función factorial

```
n! = 0          si n=0
n*(n-1)!        si n>0
```

```
int factorial(int n) {
   int resultado = 1;
   int i;
   if (n<0) return -1;
   if (n==0) return 1;

   for (i=1; i<=n; i++)
      resultado *= i;

   return resultado;
}
```

- Legibilidad?
- Eficiencia?

Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo

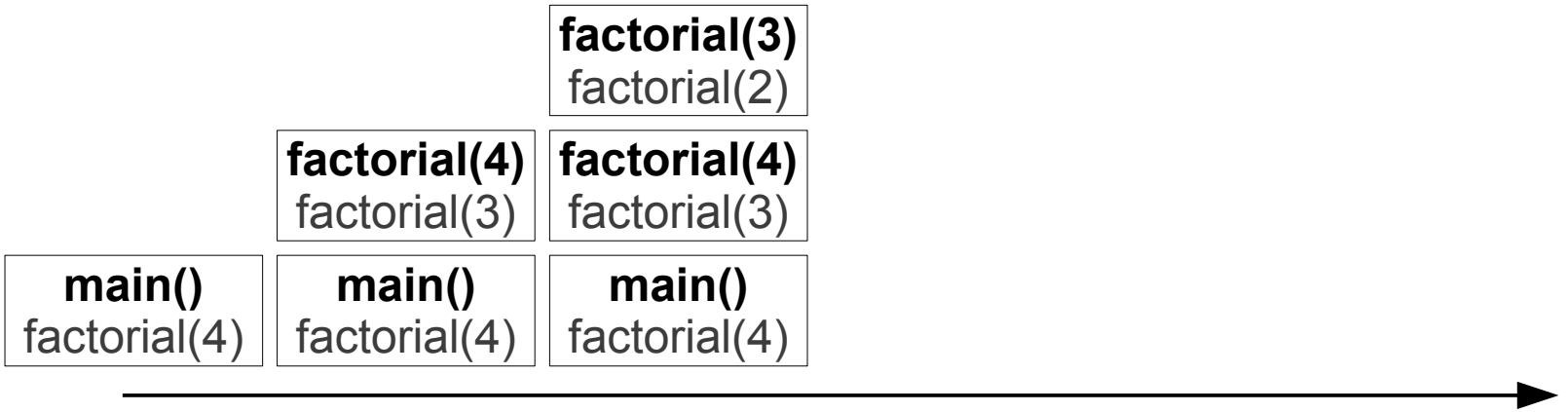
main()
factorial(4)

Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo

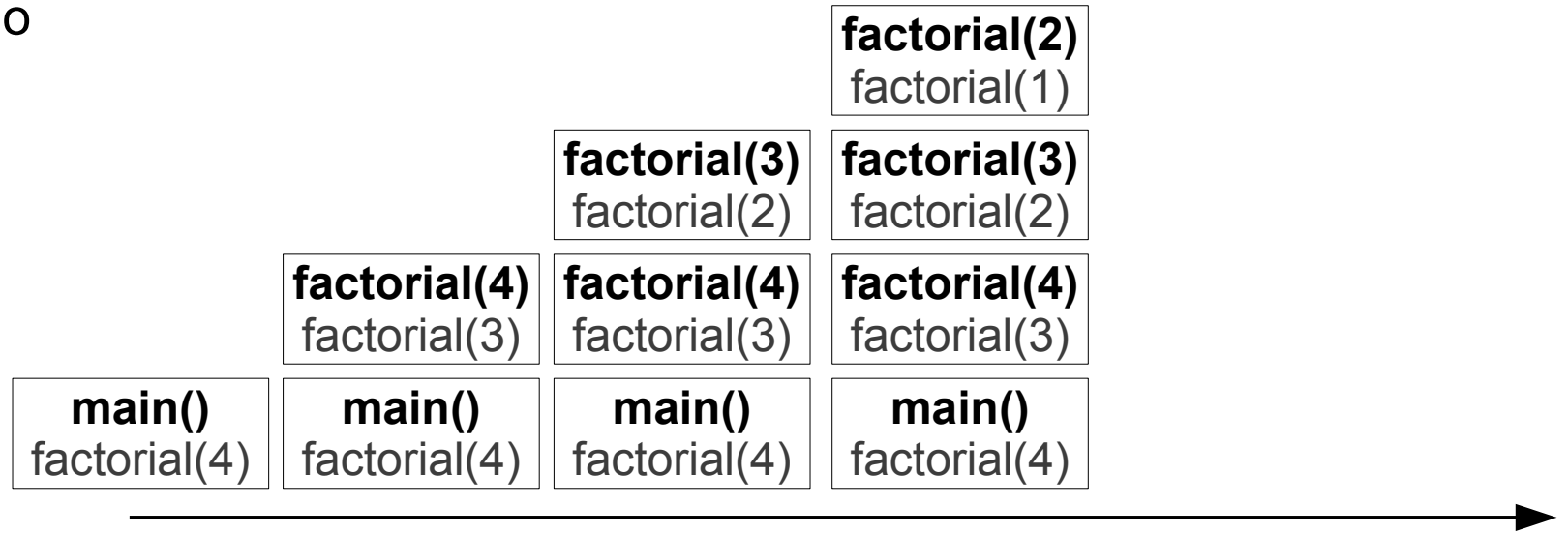
Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo



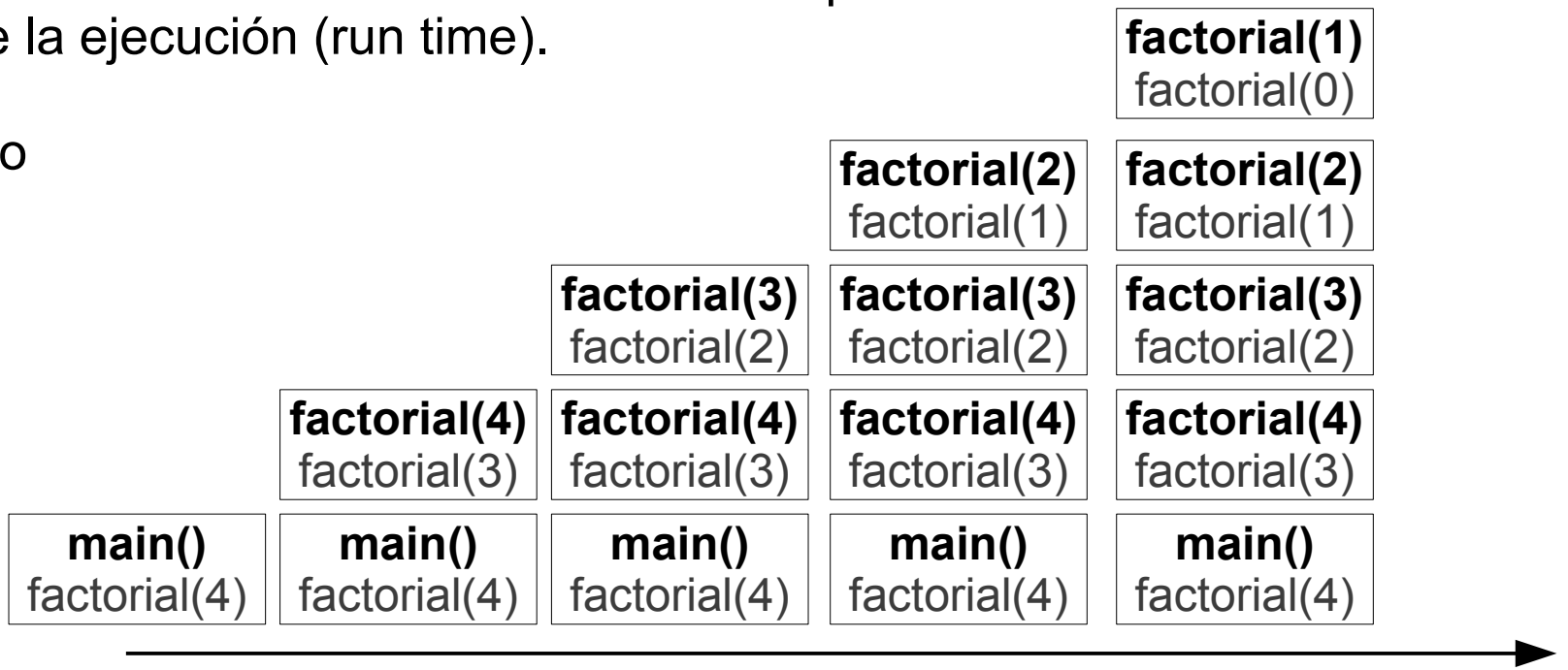
Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo



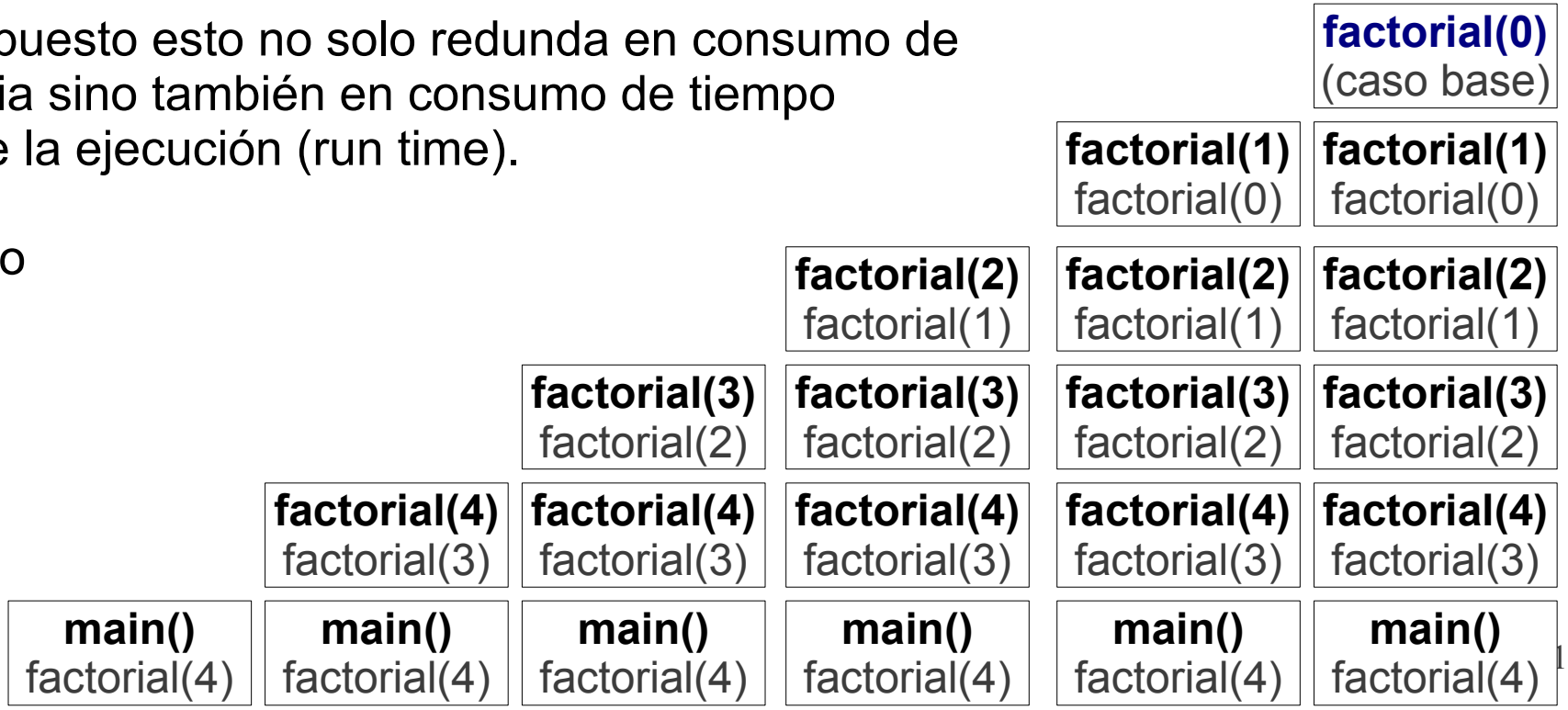
Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo



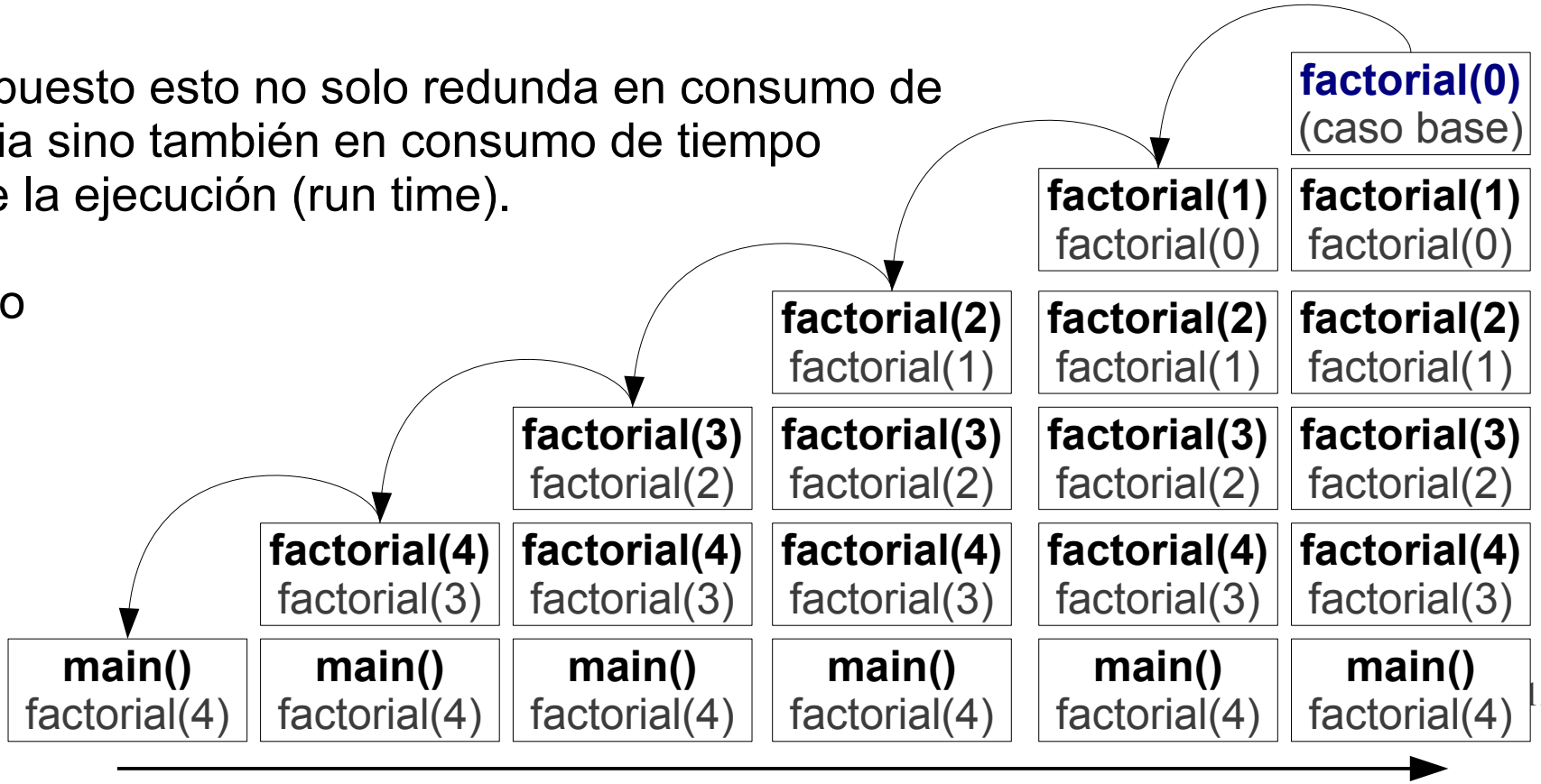
Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo



Stack frames

- Cada invocación de función requiere que se asigne un stack frame para almacenar variables automáticas, variables locales a la función, etc.
- Puede suceder que el stack se torne sobrecargado, dado que las diferentes invocaciones se van anidando, es decir, se van creando encima del stack del invocante
- Por supuesto esto no solo redunda en consumo de memoria sino también en consumo de tiempo durante la ejecución (run time).
- Ejemplo



Consideraciones

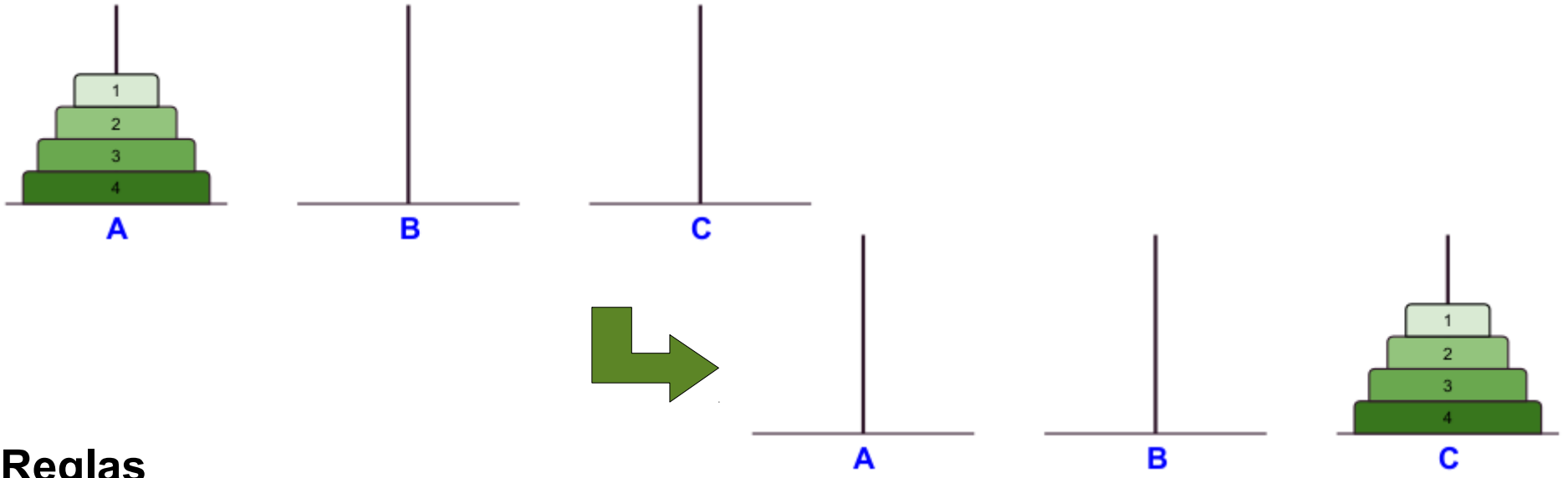
- Casos base, deben existir sino se alcanzará el límite de stack frame nesting
- Eficiencia/performance: invocar funciones tiene su costo en almacenamiento y tiempo
- Generalmente la recursión requiere un código más breve (minimalista) vs iteración y resulta más conveniente para analizar y mantener

- **Torres de Hanoi**

- **Video: Torres de Hanoi.flv**

Objetivo

- Mover toda la torre desde la espiga A a la espiga C

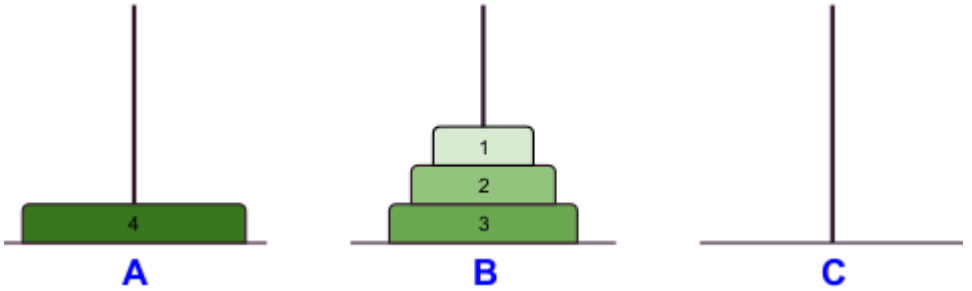


Reglas

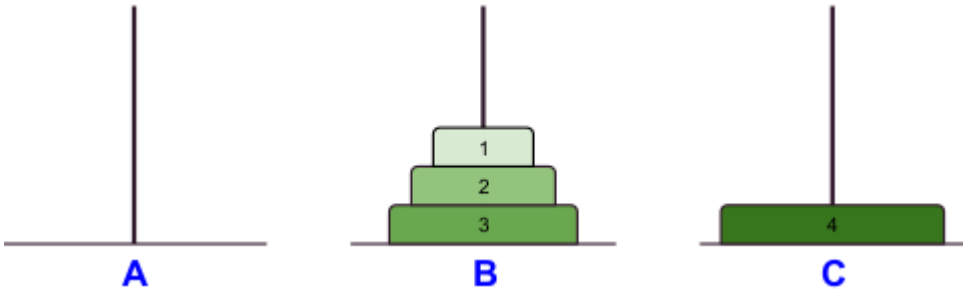
- Solo se puede mover un disco por vez
- Nunca puede colocarse un disco de mayor diámetro encima de uno de menor diámetro

- **Torres de Hanoi**
  - **Resolución** (para 4 discos)

PASO 1: Mover 3 discos de A a B usando C como espiga auxiliar

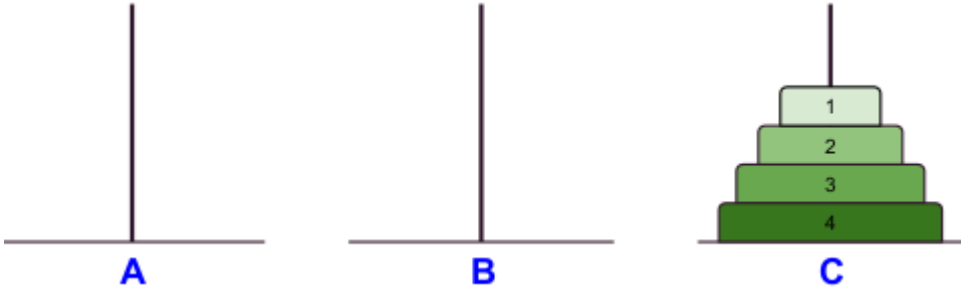


PASO 2: Mover el último disco de A a C



- **Torres de Hanoi**

PASO 3: Mover los 3 discos de B a C usando A como espiga auxiliar



Pasos Generales para resolución con N discos

PASO 1: Mover (N-1) discos de A a B
PASO 2: Mover disco N (el más grande) de A a C
PASO 3: Mover (N-1) discos de B a C

- **Torres de Hanoi**

Pasos Generales para resolución con N discos

PASO 1: Mover (N-1) discos de A a B
PASO 2: Mover disco N (el más grande) de A a C
PASO 3: Mover (N-1) discos de B a C

Versión recursiva

```
void hanoi (int n, char ori, char dest, char aux) {
  if (n == 1) {
     printf("Mover disco de %c a %c.\n", ori, dest);
  } else {
     //Mover n-1 discos de ori a dest usando aux
     hanoi(n-1, ori, aux, dest);
     printf("Mover disco de %c a %c.\n", ori, dest);
     //Mover n-1 discos de aux a dest usando ori
     hanoi(n-1, aux, dest, ori);
  }
}

int main() {
  hanoi(n, A, C, B);
```

Recursión de Cola (Tail recursion)

- Siguiendo la misma estrategia de reducción del problema a uno o más problemas simples pero evitando el frame stack nesting (anidamiento pila) se plantea el concepto de **recursión de cola (tail recursion)**

- Una **función es recursiva de cola** si no ejecuta ninguna instrucción luego de retornar, por tal motivo no hace falta mantener su frame, es más, no deberían crearse nuevos frames durante las invocaciones recursivas, sino reutilizarlo.

- La recursión de cola es generalmente tan eficiente como la iteración (depende del compilador). Compiladores que permiten optimizar código detectan funciones tail recursivas y las convierten en contrucciones iterativas.

Recursión de Cola (Tail recursion)

Ejemplo

Sumar dos números: a + b

Versión recursiva

```
sumar_rec(a, b)
        Si a==0  → b
        Si a>=1  → 1 + sumar_rec(a-1, b)
```

```
sumar_rec(3, 8)
                = 1 + sumar_rec(2, 8)          pues a >= 1
                = 1 + 1 + sumar_rec(1, 8)      pues a >= 1
                = 1 + 1 + 1 + sumar_rec(0, 8)  pues a >= 1
                = 1 + 1 + 1 + 8 = 11           pues a == 0
```

Recursión de Cola (Tail recursion)

Ejemplo

Sumar dos números: a + b

Versión recursiva de cola

sumar_tail_rec(a, b, c) con c inicialmente valiendo 0

Si a==0 → b+c

Si a>=1 → sumar_tail_rec(a-1, b, c+1)

sumar_tail_rec(3, 8, **0**)

= sumar_tail_rec(2, 8, 1) pues a >= 1

= sumar_tail_rec(1, 8, 2) pues a >= 1

= sumar_tail_rec(0, 8, 3) pues a >= 1

= 8 + 3 = 11 pues a == 0

Observar que no hay apilamiento de stack frames

- **Recursión de Cola (Tail recursion)**

  - **Ejemplo**

    Sumar dos números: a + b

    Versión recursiva de cola (más simple)

```
sumar_tail_rec(a, b)
        Si a==0  → b
        Si a>=1  → sumar_tail_rec(a-1, b+1)
```

```
sumar_tail_rec(3, 8)
                = sumar_tail_rec(2, 9)      pues a >= 1
                = sumar_tail_rec(1, 10)     pues a >= 1
                = sumar_tail_rec(0, 11)     pues a >= 1
                = 11                        pues a == 0
```

- **Recursión de Cola (Tail recursion)**

  - **Ejemplo**
    Factorial

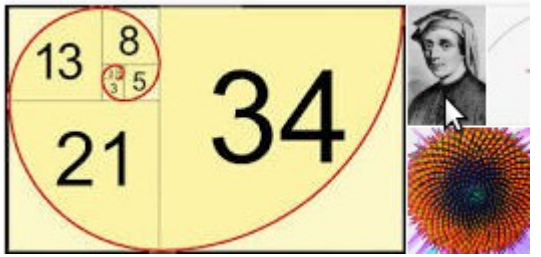
    Versión recursiva de cola

    factorial_tail_rec (n, acum) con acum inicializado en 1
    Si n==0 → acum
    Si n>0 → factorial_tail_rec(n-1, acum*n)

    factorial_tail_rec(4, 1)

= factorial_tail_rec(3, 1*4)	pues n>0
= factorial_tail_rec(2, 4*3)	pues n>0
= factorial_tail_rec(1, 12*2)	pues n>0
= factorial_tail_rec(0, 1*24)	pues n>0
= 24	pues n==0

Recursión de Cola (Tail recursion)



Ejemplo

Serie de Fibonacci:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …, n-2, n-1, n, ...

Versión recursiva

```
fib_rec (n)
        Si n==1 o n==2      → 1
        Si n > 2            → fib_rec(n-1) + fib_rec(n-2)
```

```
fib_rec(6)
    = fib_rec(5) + fib_rec(4)
    = fib_rec(4) + fib_rec(3) + fib_rec(3) + fib_rec(2)
    = fib_rec(3) + fib_rec(2) + fib_rec(2) + fib_rec(1) + fib_rec(2) + fib_rec(1) + 1
    = fib_rec(2) + fib_rec(1) + 1 + 1 + 1 + 1 + 1 + 1
    = fib_rec(2) + fib_rec(1) + 6
    = 1 + 1 + 6
    = 8
```

- **Recursión de Cola (Tail recursion)**

  - **Ejemplo**

Versión recursiva de cola

fib_tail_rec (n, a, b), inicializados a=0, b=1
Si n==1 → b
Si n > 1 → fib_tail_rec(n-1, b, a+b)

fib_tail_rec(6, 0, 1)

= fib_tail_rec (5, 1, 0+1)	pues n > 1
= fib_tail_rec (4, 1, 1+1)	pues n > 1
= fib_tail_rec (3, 2, 2+1)	pues n > 1
= fib_tail_rec (2, 3, 2+3)	pues n > 1
= fib_tail_rec (1, 5, 3+5)	pues n > 1
= 8 (valor de b)	pues n == 1

- **Recursión de Cola (Tail recursion)**

  - **Ejercicios**

    - Función de Ackermann
      http://es.wikipedia.org/wiki/Función_de_Ackermann

    - Producto de dos enteros positivos prod_rec(a,b) & prod_tail_rec(a, b, ?)

    - Potencia de dos enteros positivos pto_rec(a,b) & pot_tail_rec(a, b, ?)