Análisis y diseño de algoritmos

Sesión 03

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis y diseño de algoritmos recursivos utilizando un lenguaje de programación

Un punto aparte, el logaritmo

- $3^2 = 9$
- Igualmente $\log_3 9 = 2$
  - “El log en base 3 de 9 es 2."
- La forma de pensar acerca del logaritmo es:
  - “El log en base x de y es el número al que puede elevar x para obtener y."
  - Dígase a si mismo “El log es el exponente." (y repítelo una y otra vez hasta que lo crea)
  - Trabajaremos más con logaritmos en base 2
- $\log_2 32 = ?$ $\log_2 8 = ?$ $\log_2 1024 = ?$ $\log_{10} 1000 = ?$

¿Cuándo ocurren los Logaritmos?

- Los algoritmos tienden a tener un término logarítmico cuando usan la técnica divide y vencerás (la cual estudiaremos más adelante)
- El tamaño del conjunto de datos se obtiene dividiendo entre 2

```
public int foo(int n) {
    // pre n > 0
    int total = 0;
    while (n > 0) {
        n = n / 2;
        total++;
    }
    return total;
}
```

Número de iteración	Valor de n
1	n
2	n/2
3	n/4
4	n/8
5	n/16
…	…
k	$n/2^{k-1}$

- ¿Cuál es el orden (notación Big O) del código anterior?

A. O(1) B. O(logN) C. O(N)

D. O(Nlog N) E. $O(N^2)$

Ejercicio

Demostrar que la búsqueda binaria en un arreglo de n componentes ordenados en forma ascendente tiene un orden logarítmico de base 2 de la cantidad de elementos del arreglo = $O(\log_2 n)$.

```
public static int bsearch( int [] a, int n, int x )
{ int ini = 0, fin = n-1;
    while( ini <= fin ) {
        int medio = (ini + fin) / 2;
        if( a[medio] == x ) return medio;
        else if( a[medio] > x ) fin = medio-1;
        else ini = medio + 1;
    }
    return -1;
}
```

Ejercicio

Tamaño de la entrada: n = cantidad de componentes de a

Peor caso: x no está en a

```
public static int bsearch( int [] a, int n, int x ) {
    int ini = 0, fin = n-1;                          c1
    while( ini <= fin ) {                            Tiempo de la condición: c2
        int medio = (ini + fin) / 2;                 Tiempo del cuerpo: c3
        if( a[medio] == x ) return medio;
        else if( a[medio] > x ) fin = medio-1;
        else ini = medio + 1;
    }
    return -1;                                       c4
}
```

Sea k = cantidad de iteraciones del while, entonces

$T(n) = c_1 + k(c_2+c_3) + c_2 + c_4$.

La pregunta es cómo definir k en función de n.

Ejercicio

¿Cómo estimar k en función de n?

El peor caso es cuando “x” no está en “a”.

Veamos cómo vamos descartando componentes del arreglo en función del número de iteración del while: Si tenemos n componentes, en cada iteración se descarta la componente del medio del arreglo, entonces de las n-1 componentes que falta revisar sólo se va considerar la mitad, entonces quedan (n-1)/2 componentes para la siguiente iteración, y así sucesivamente.

Calculemos cuál es caso para la iteración genérica k.

Número de iteración	Componentes por revisar
1	n
2	(n-1)/2
3	(n-3)/4
4	(n-7)/8
5	(n-15)/16
…	…
k	$\frac{n-(2^{k-1}-1)}{2^{k-1}}$

Ejercicio

Entonces vimos que en la iteración k, la cantidad de componentes que quedan por revisar es:

$$\frac{n-(2^{k-1}-1)}{2^{k-1}}$$

Como x no está en el arreglo a, en la última iteración completa que realiza el while queda una componente del arreglo por revisar.
Entonces:

$$\frac{n-(2^{k-1}-1)}{2^{k-1}}=1; \qquad n-(2^{k-1}-1)=2^{k-1};$$

$$n-2^{k-1}+1=2^{k-1}; \qquad n+1=2^{k-1}+2^{k-1};$$

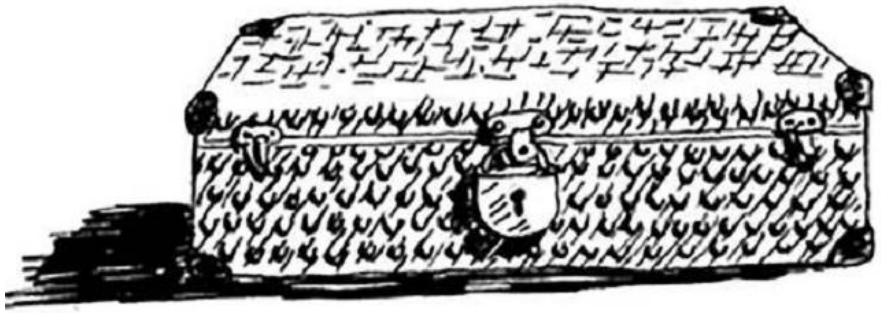
$$n+1=2\times 2^{k-1}; \qquad n+1=2^{k}; \qquad \log_2(n+1)=k.$$

Con lo que T(n) = $c_1 + \log_2(n+1)(c_2+c_3) + c_2 + c_4$
Luego, el algoritmo es $O(\log_2(n))$, lo que se quería demostrar

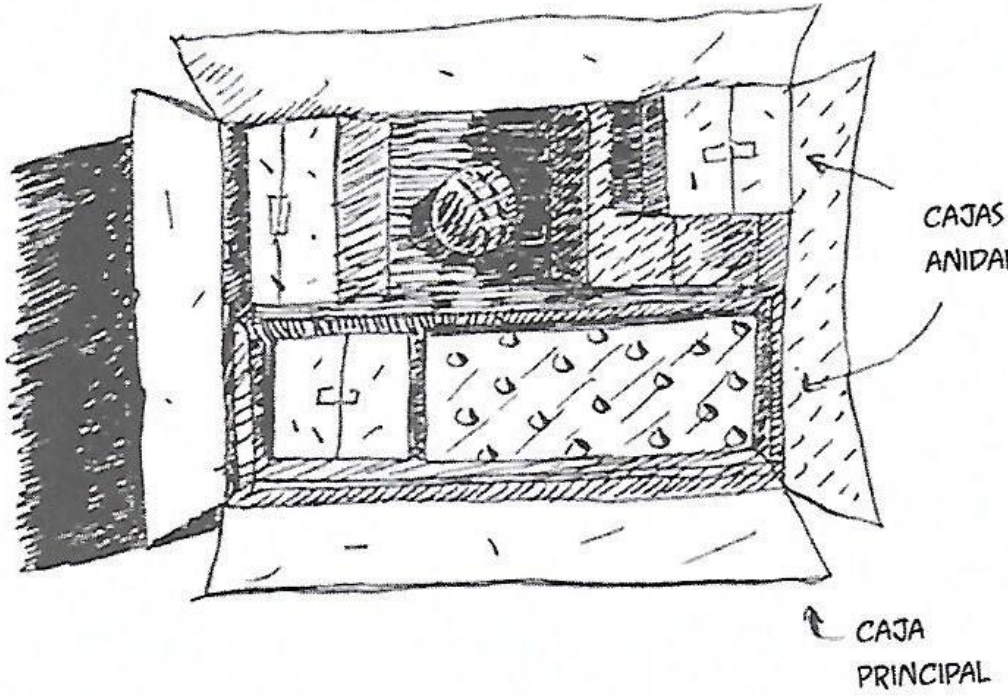
Recursividad

Encuentra la llave

Supongamos que estás revisando el ático de tu abuela y te encuentras con una misteriosa maleta cerrada.



La abuela te dice que la llave de la maleta probablemente esté en la siguiente caja:



Esta caja contiene más cajas, Incluso con más cajas dentro de cada caja. La llave está en alguna de las cajas. ¿Cuál es su algoritmo para buscar la llave? Piense en un algoritmo.

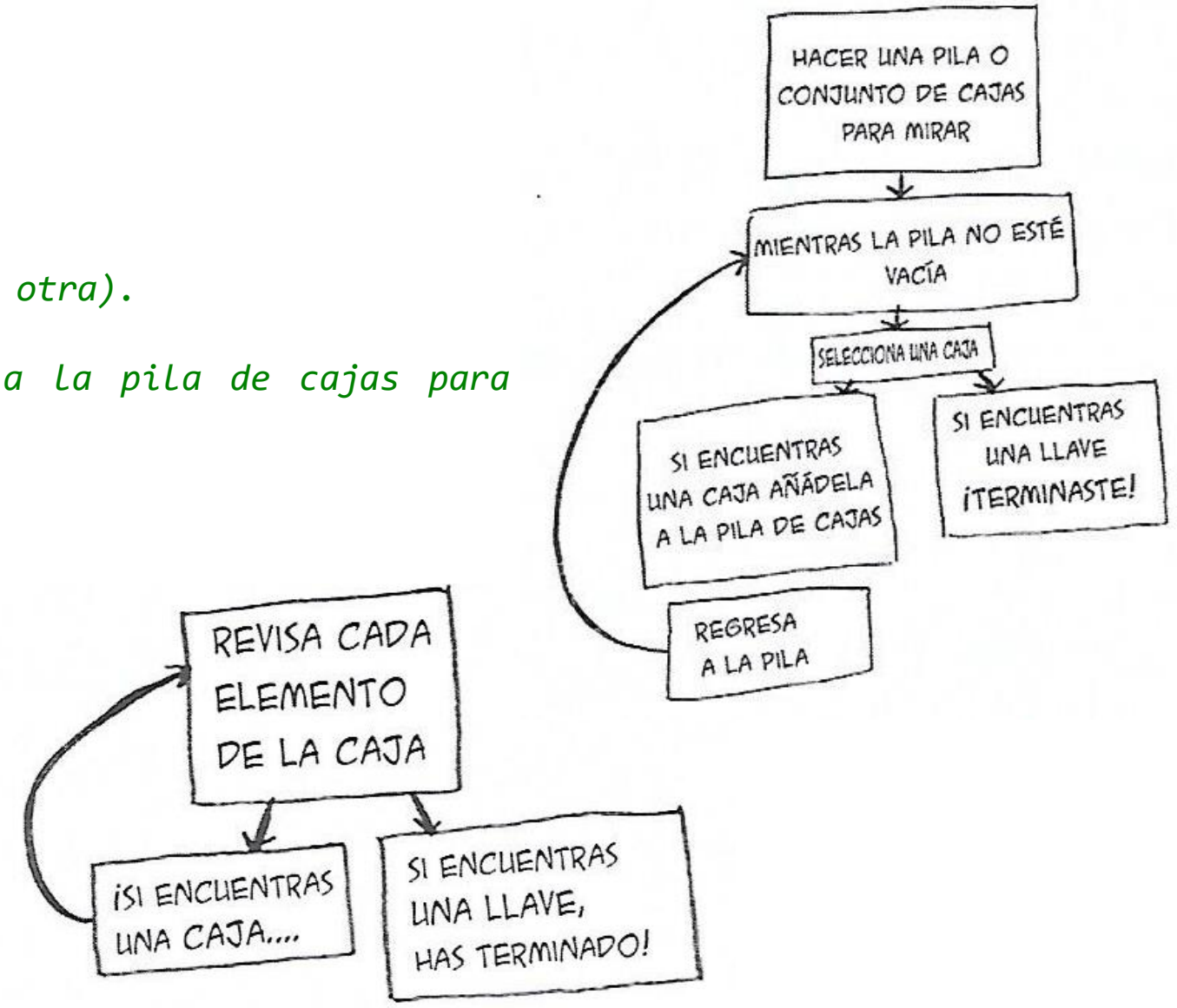
Recursividad

Aquí tienes un posible enfoque:

1.Haz una pila de cajas para revisar (una sobre otra).
2.Toma una caja y busca dentro de ella.
3.Si encuentras adentro otra caja, agréguela a la pila de cajas para buscar en ella después.
4.Si encuentras la llave, ¡ya terminaste!
5.Repite.

Aquí hay un enfoque alternativo:

1. Mira en la caja.
2. Si encuentra otra caja, repite el paso 1.
3. Si encuentras una llave, ¡ya terminaste!



¿Qué enfoque le parece más fácil?

El primer enfoque usa un ciclo while. Mientras que la pila no está vacía, toma una caja y mira a través de ella. La segunda forma usa recursividad. La recursión es cuando una función se llama a sí misma.

Recursividad

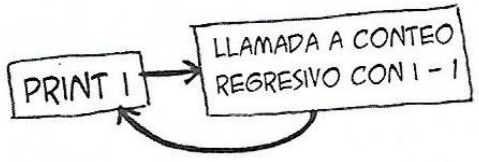
Ambos enfoques logran lo mismo ¿Cuál es más claro?. La recursión siempre se utiliza para clarificar una solución de un problema vía algoritmos. Quizás no hay beneficio de rendimiento al usar la recursividad; de hecho, los bucles son a veces mejores para el rendimiento. "Los bucles pueden lograr un aumento en el rendimiento de su programa. La recursividad puede lograr un aumento de rendimiento para su programador. ¡Elija cuál es más importante en su situación!

Debido a que una función recursiva se llama a sí misma, es fácil escribir una función de forma incorrecta que termina en un ciclo infinito. Por ejemplo, supongamos que desea escribir una función que imprime una cuenta atrás, como esta:

```
> 3...2...1
```

Se puede escribir recursivamente como sigue:

```
public class AED_Programa8 {
    public static void main(String[] args) {
        conteo_regresivo(3);
    }
    public static void conteo_regresivo(int i){
        System.out.println(i);
        conteo_regresivo(i-1);
    }
}
```



Ciclo infinito

Escribe este código y ejecútalo. Notarás un problema: ¡esta función se ejecuta infinitamente!

```
> 3...2...1...0...-1...-2...
```

Recursividad

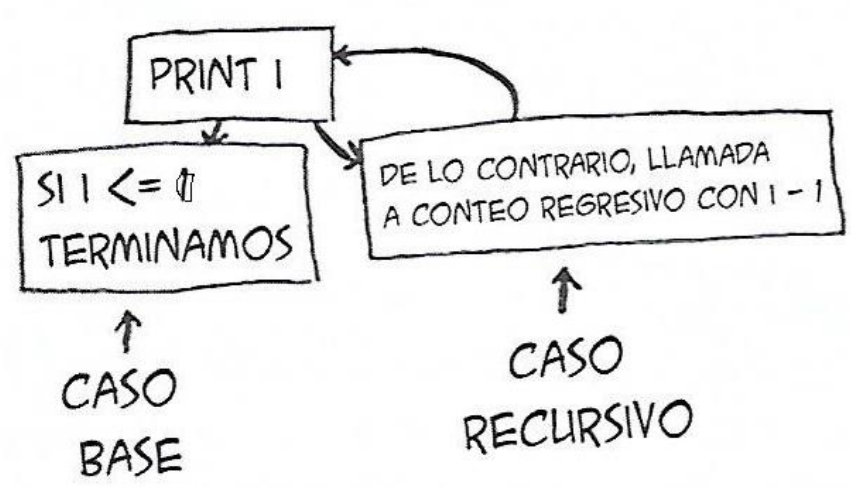
Cuando escribe una función recursiva, debe decirle cuándo detener la recursión. es por eso que cada función recursiva tiene dos partes: **el caso base y el caso recursivo**. El caso recursivo consiste en que la función se llama a sí misma. El caso base es cuando la función no se llama a sí misma nuevamente ... de tal forma que no crea un ciclo infinito.

Agreguemos un caso base a la función de cuenta atrás:

```
public class AED_Programa8 {
    public static void main(String[] args) {
        conteo_regresivo(3);
    }
    public static void conteo_regresivo(int i){
        System.out.println(i);
        if(i<=0){
            return;
        }
        conteo_regresivo(i-1);
    }
}
```

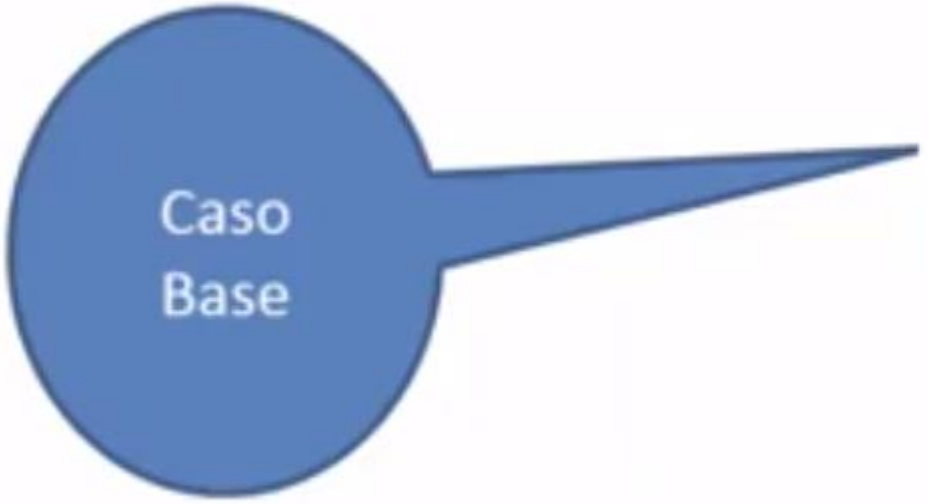
Caso Base

Caso Recursivo



¡Ahora la función trabaja como se quiere!

Recursividad



Recursividad - Método de las tres preguntas

La pregunta Caso-Base:

¿Existe una salida no recursiva o caso base del algoritmo?
Además, ¿el algoritmo funciona correctamente para ella?

La pregunta Más-pequeño:

¿Cada llamada recursiva se refiere a un caso más pequeño del problema original?

La pregunta Caso-General:

¿Es correcta la solución en aquellos casos no base?

Recursividad

Recursividad

La capacidad de un algoritmo de llamarse a sí mismo

Caso Base

Es el caso para el que tenemos una respuesta

Caso General

El caso general expresa la solución en términos de una llamada a sí mismo con una versión más pequeña del problema

N!

El clásico primer ejemplo de algoritmo recursivo

- N!

  5! = 5 * 4 * 3 * 2 * 1 = 120

```
// pre n >= 0
public int fact(int n){
int res = 1;
  for(int i = 2; i <= n; i++)
        res *= i;
return res;
}
```

- Definición matemática del factorial

  0! = 1

  N! = N * (N - 1)!

  La definición es recursiva.

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

Recursividad

El factorial de un entero positivo n, se define como el producto de todos los números enteros positivos desde 1 hasta n:

$$n! = 1 \times 2 \times 3 \times 4 \times \ldots \times (n-1) \times n$$

*n! = n * (n – 1)!*

Caso Base

Factorial(0) = 1 (0! es 1)

Caso General

Factorial(n) = n * Factorial(n-1)

```java
public class AED_Programa9 {
    public static void main(String[] args) {
        // TODO code application logic here
        AED_Programa9 factorial=new AED_Programa9();
        System.out.println("El factorial de 5 mediante un método recursivo "+
                factorial.factorialRecursivo(5));
    }
    public int factorialRecursivo(int n) {
        //Caso Base
        if (n == 0) {
            return 1;
        } else {
            //Caso General
            return n * factorialRecursivo(n - 1);
        }
    }
}
```

Recursividad

- La recursión implica repetir el conjunto de sentencias como una subtarea de sí mismo.
- La ejecución de un procedimiento conduce a una nueva ejecución del mismo procedimiento.
- Se forman múltiples activaciones del procedimiento, y todas menos una, está esperando que se complete el resto de activaciones del procedimiento.

Como analogía, considere el proceso de llevar a cabo conversaciones telefónicas con la funcionalidad de llamada en espera. En ese caso, se deja a un lado una conversación telefónica que aún no se ha completado, mientras que se procesa otra llamada entrante. El resultado es que dos conversaciones tienen lugar. Sin embargo, no se llevan a cabo una detrás de la otra, sino que en su lugar una de las conversaciones se realiza dentro de la otra.

Ejercicios

¿Qué secuencia de números sería impresa por el siguiente método recursivo si comenzamos asignando a N el valor 1? ¿Cuál es la condición de terminación ?

```java
public class AED_Programa10 {
    public static void main(String[] args) {
        // TODO code application logic here
        ejercicio1(1);
    }
    public static void ejercicio1(int n){
        System.out.println(n);
        if(n<3){
            ejercicio1(n+1);
        }
        System.out.println(n);
    }

}
```

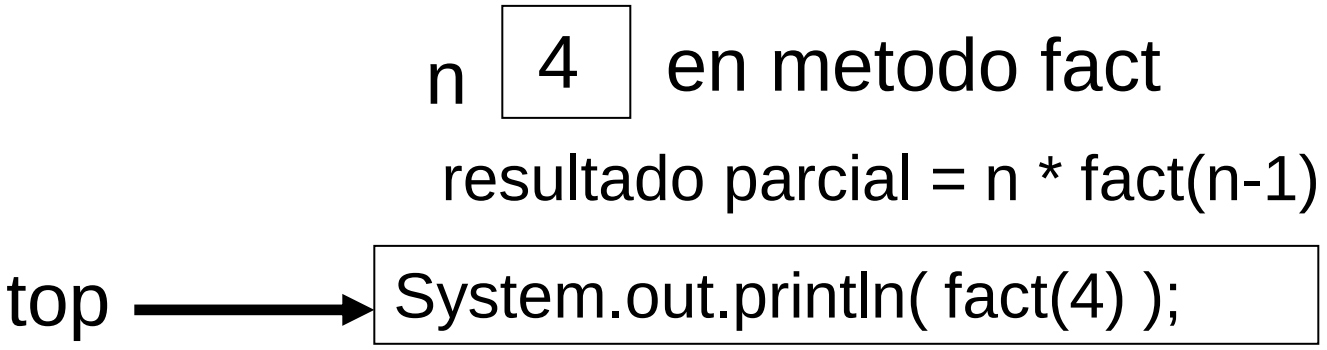
```java
public class AED_Programa10 {
    public static void main(String[] args) {
        ejercicio2(100);
    }
    public static void ejercicio2(int n){
        if(n>0){
            System.out.println(n);
            ejercicio2(n/2);
        }
        System.out.println(n+1);
    }
}
```

¿Qué secuencia de números sería impresa por el siguiente método recursivo si comenzamos asignando a N el valor 100? ¿Cuál es la condición de terminación ?

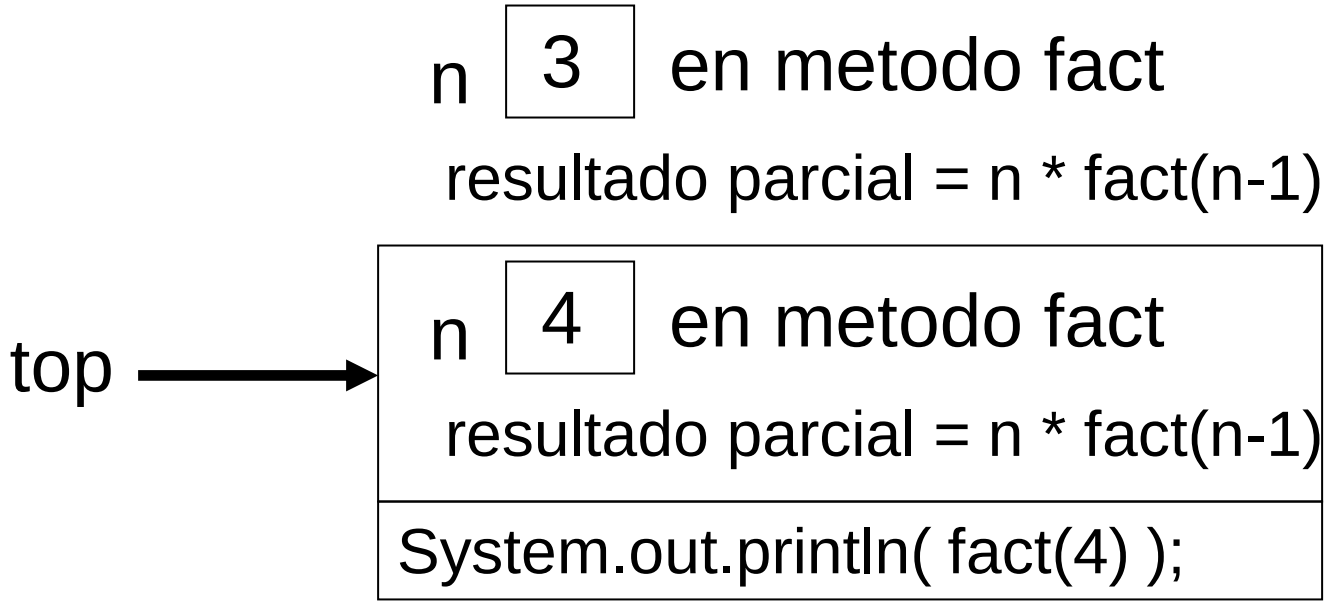
Seguimiento con la pila de programas

System.out.println(fact(4));

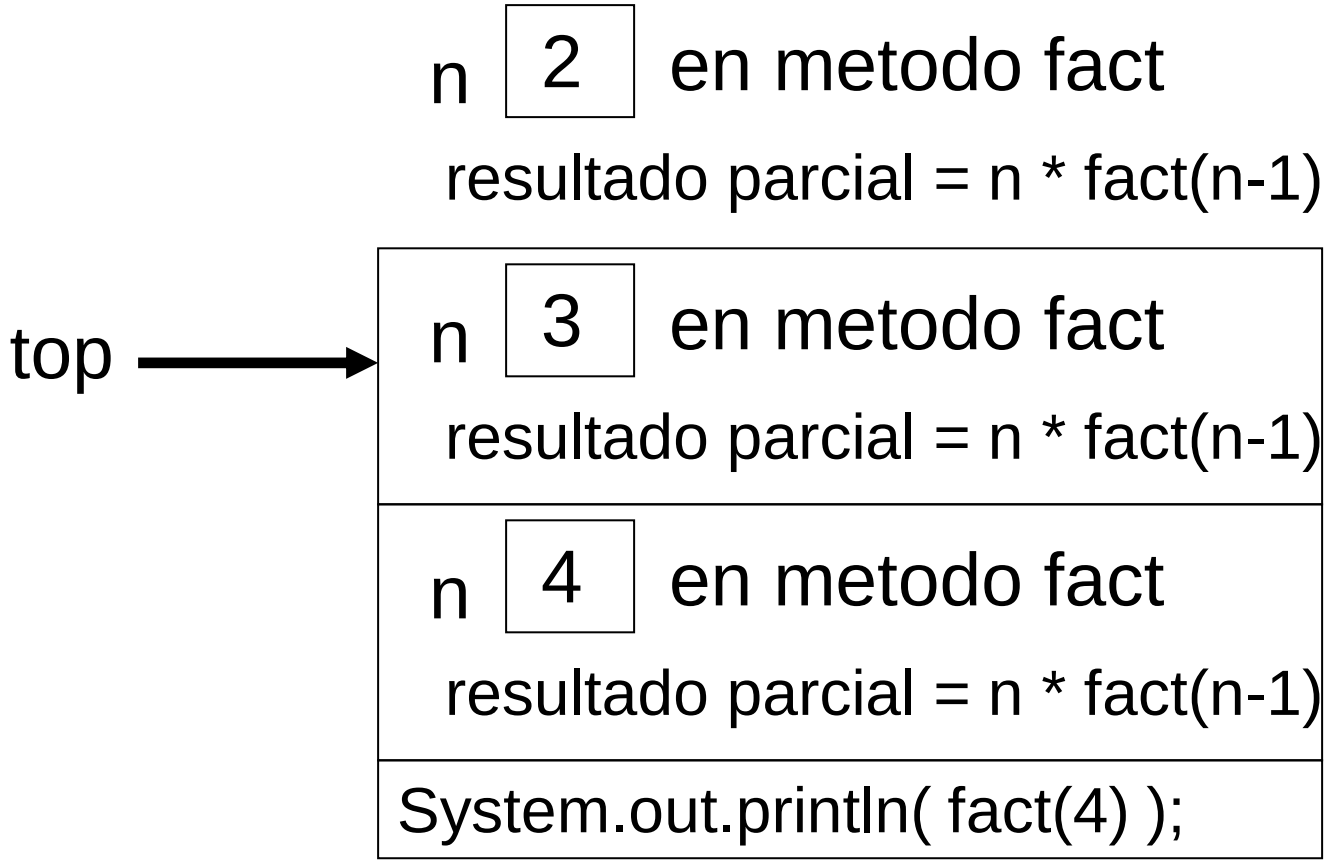
Llamada al método fact con 4



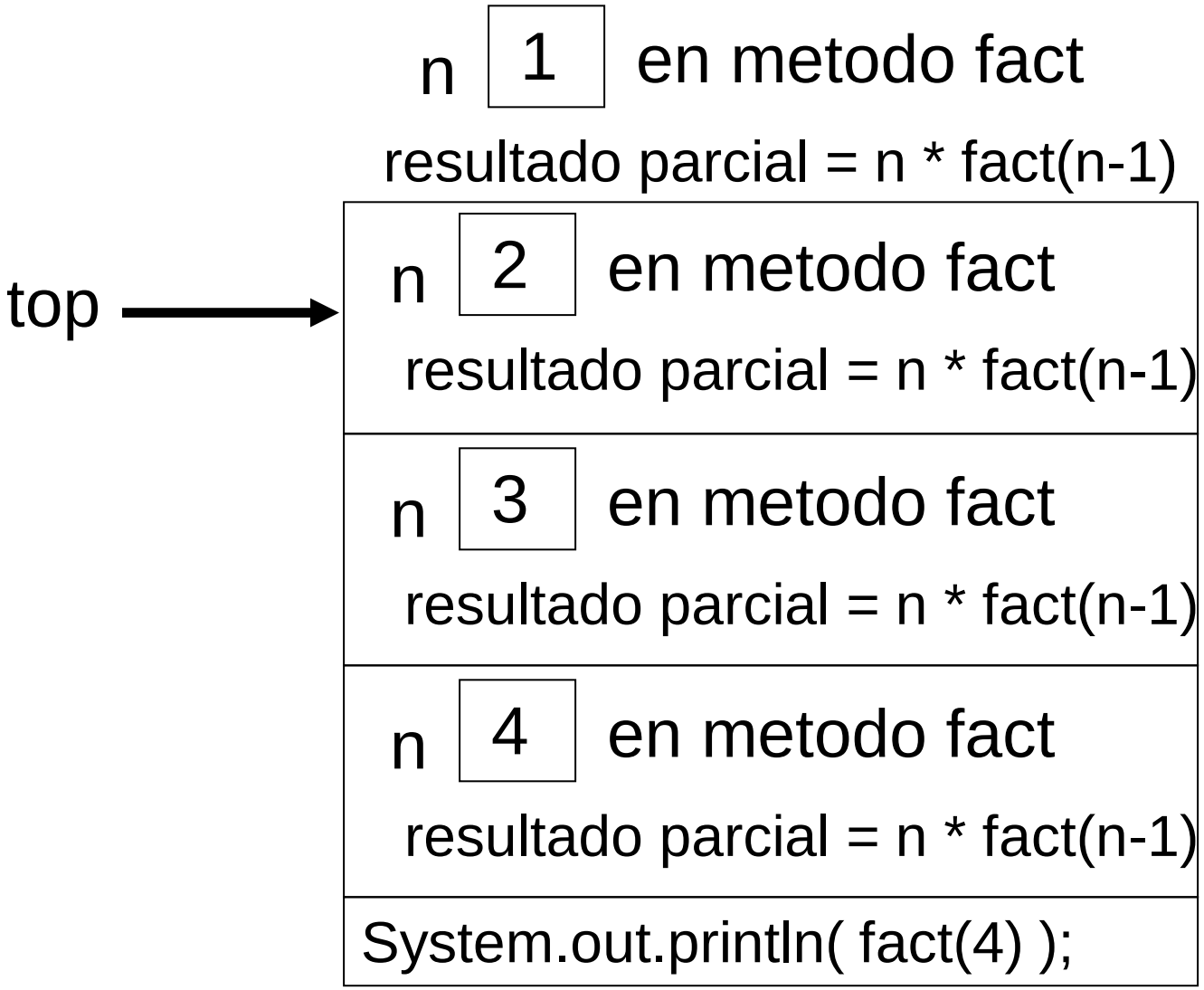
Llamada al método fact con 3



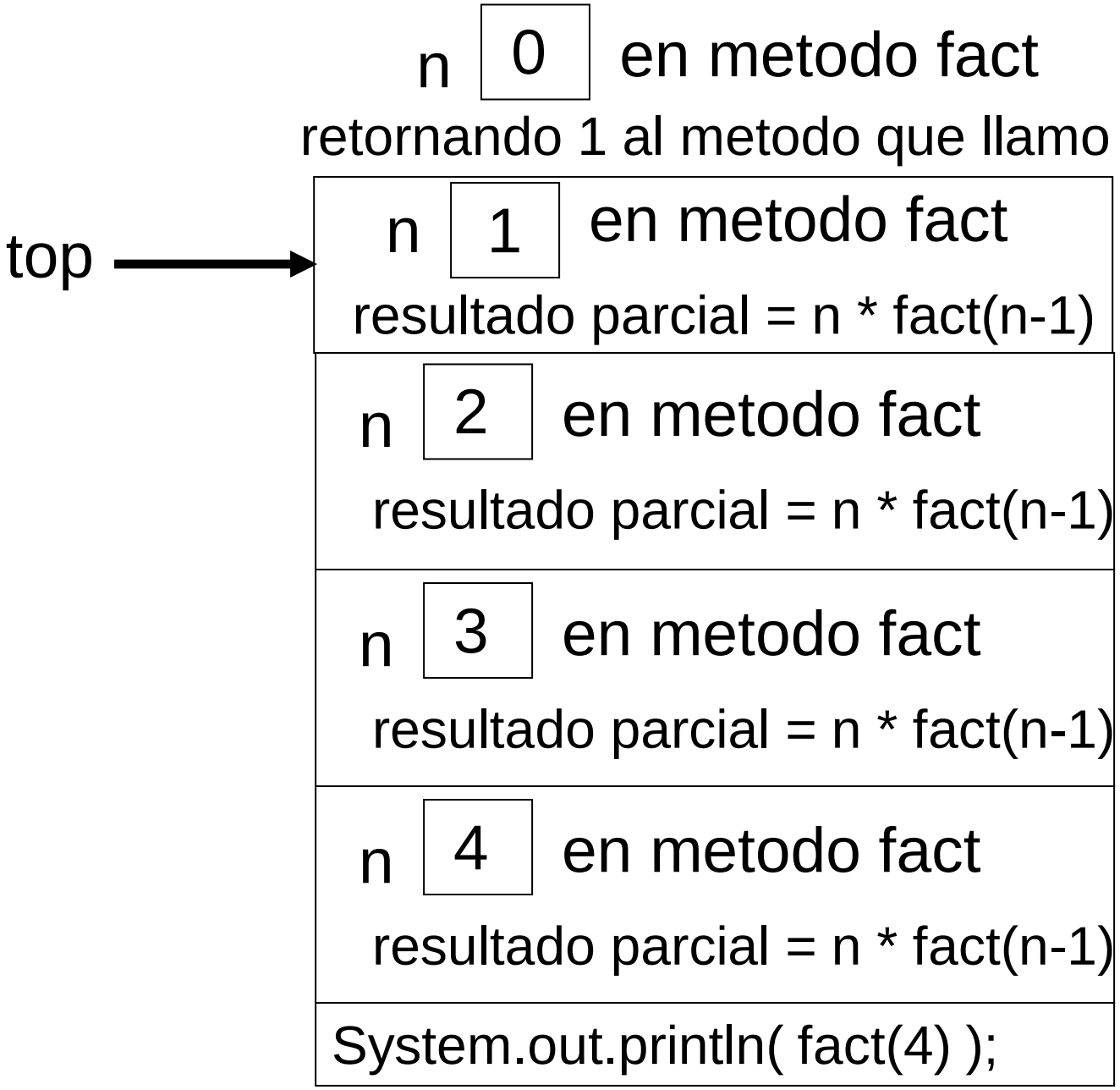
Llamada al método fact con 2



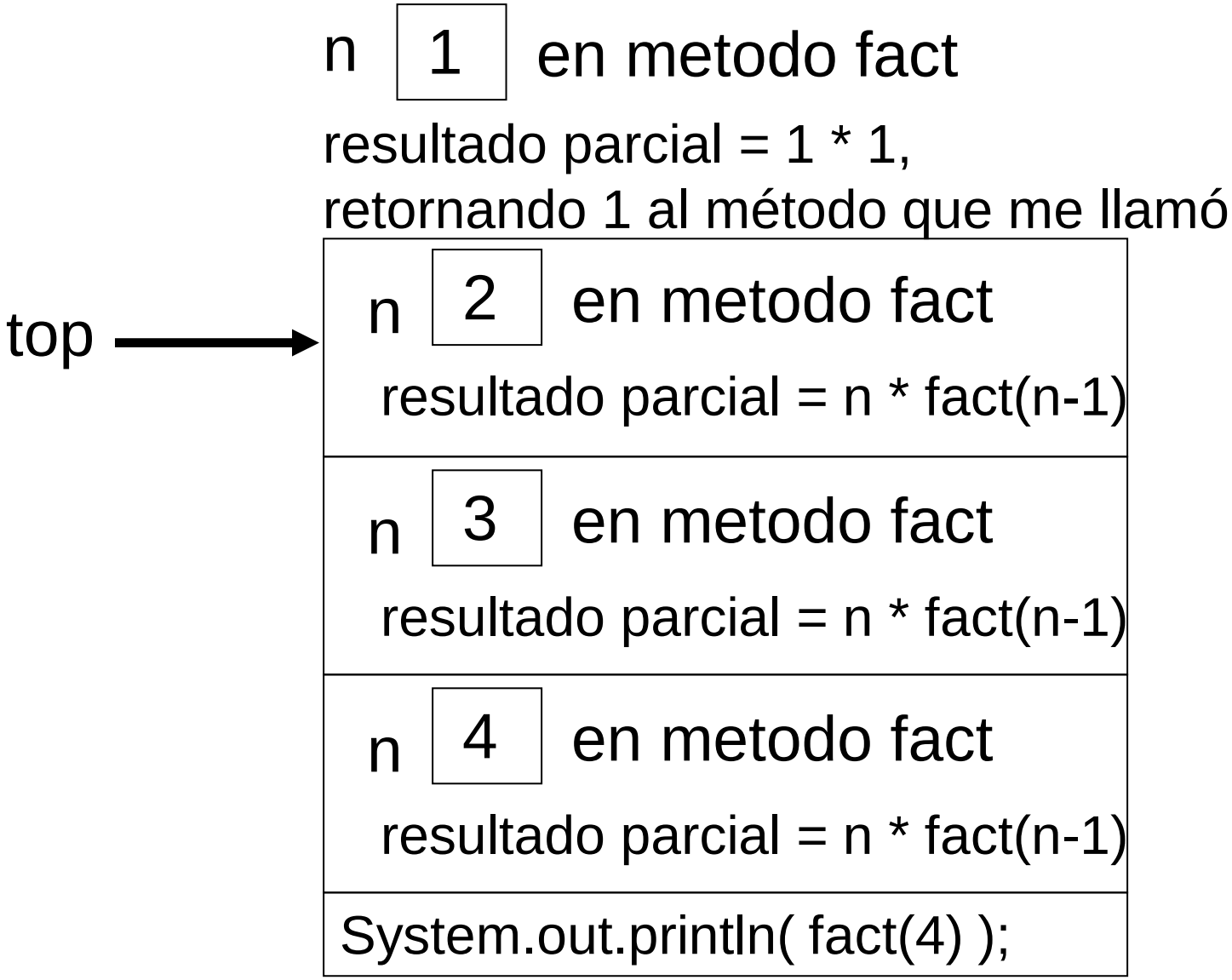
Llamada al método fact con 1



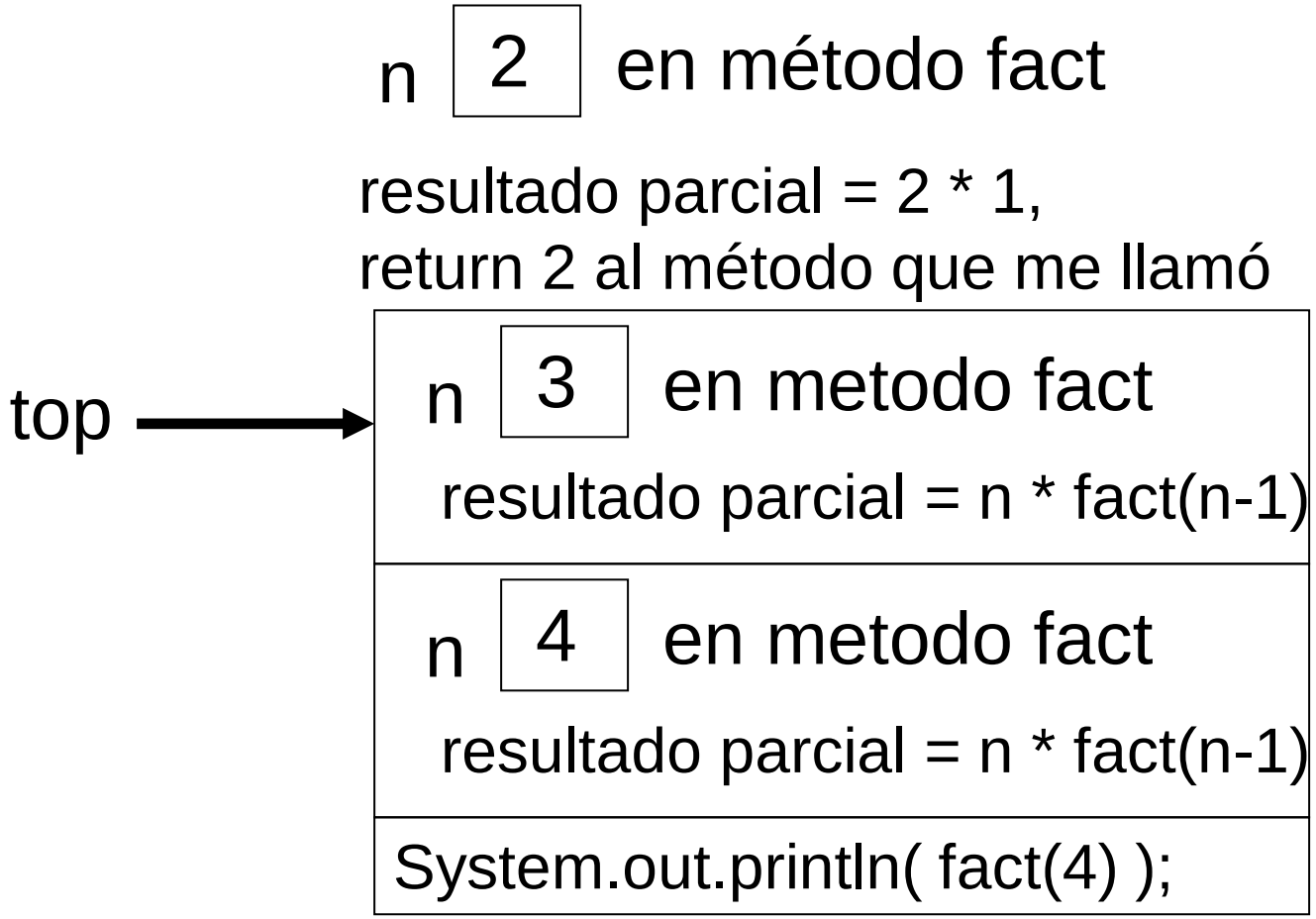
Llamada al método fact con 0 y retornando 1



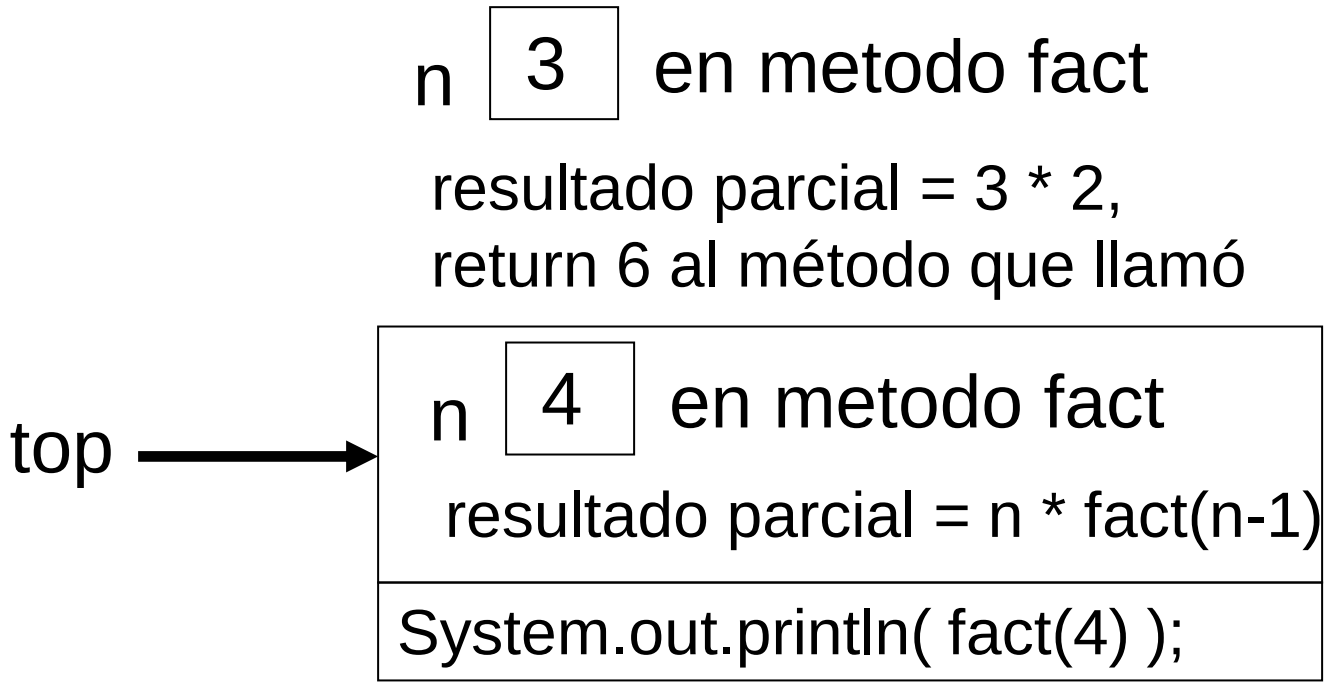
Retornando 1 de fact(1)



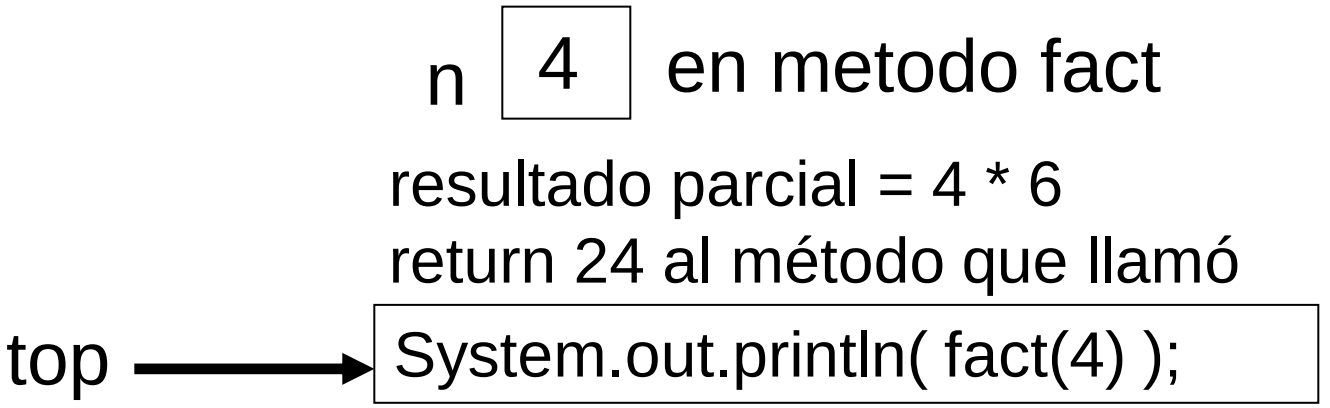
Retornando 2 de fact(2)



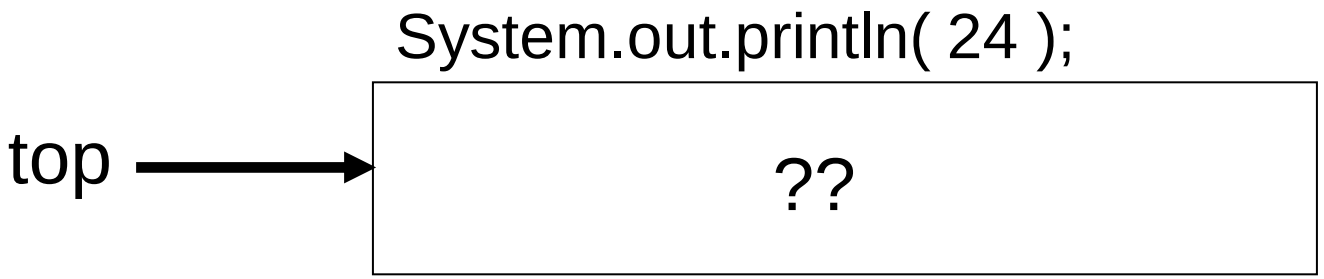
Retornando 6 de fact(3)



Retornando 24 de fact(4)



Llamando a System.out.println



Escribir un método recursivo

Primero, identifique una forma de hacer que el problema sea más simple en forma progresiva.

Identifique una condición, denominada “condición de detención”, que esté asociada con la versión más simple del problema. Es el caso base.

Use una declaración if para verificar la condición de detención. El cuerpo if debe contener la solución de la versión más simple del problema.

Típicamente el cuerpo else debe contener una o varias llamadas al mismo método con uno o más valores de argumento que lo hagan más simple en forma progresiva. Una vez que se llama al método, éste continúa llamándose a sí mismo en forma automática con condiciones más simples en forma progresiva, hasta que se satisface la condición de detención.

El proceso de llamada no debe omitir la condición de detención. En realidad debe alcanzarlo

•Señal transmitida:
- "Reporte sus bajas más bajas en todos los barcos más alejados".

•Señal devuelta:
- "Mis bajas más bajas en todos los barcos más alejados son ..."

Escribir un método recursivo

El cuerpo del método recursivo necesita una instrucción if cuya condición es la condición de detención:

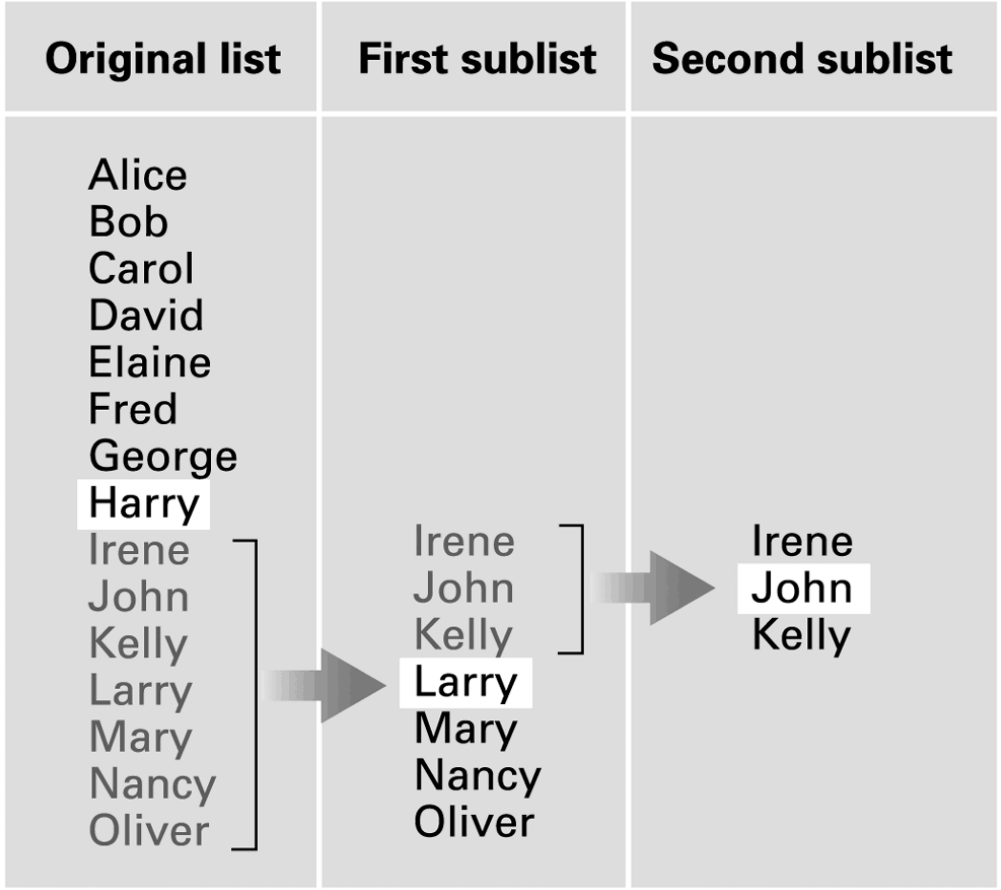
if (< *condición de detención*>){
Resuelva el problema local y return.
}

else{
Haga las siguientes llamadas recursivas.
Procese la información devuelta por llamadas recursivas y return.
}

Algoritmo de Búsqueda Binaria usando recursividad

Se supone que la lista está ordenada. La búsqueda es similar a la búsqueda de una palabra en el diccionario. Se decide comenzar la búsqueda con la entrada central de la lista, o la primera entrada de la segunda mitad de la lista. Si no hay éxito en la búsqueda habremos reducido la búsqueda a la primera o segunda mitad de la lista , dependiendo si es mayor o menor que la entrada que hemos analizado. Luego continuamos aplicando la misma técnica. En este escenario entonces se puede usar recursividad.

"Se busca John"



Un primer boceto del Algoritmo de búsqueda binaria

```
if (Lista está vacía) {
    Informar de que la búsqueda ha fallado}
else {
    EntradaAComparar = entrada central en la lista
    if (ValorObjetivo == EntradaAComparar) {
        Informar de que la búsqueda ha tenido éxito}
    if (ValorObjetivo < EntradaAComparar) {
        Buscar ValorObjetivo en la parte de la Lista
        anterior a EntradaAComparar e informar del resultado
        de dicha búsqueda}
    if (ValorObjetivo > EntradaAComparar){
        Buscar ValorObjetivo en la parte de la Lista
        posterior a EntradaAComparar e informar del
        resultado de dicha búsqueda }
}
```

El algoritmo de búsqueda binaria en Pseudocódigo

```
Public static int Buscar (Lista, ValorObjetivo){
    if (Lista está vacía){
        Informar que la búsqueda ha fallado}
    else{
        EntradaAComparar = Entrada central de la Lista
        if (ValorObjetivo == EntradaAComparar){
            Informar que la búsqueda ha tenido éxito }
        if (ValorObjetivo < EntradaAComparar){
            Sublista = Parte de la Lista anterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
        if (ValorObjetivo > EntradaAComparar){
            Sublista = Parte de la lista posterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
             }
      }
```

El algoritmo de búsqueda binaria – Caso 1

```
Public static int Buscar (Lista, ValorObjetivo){
    if (Lista está vacía){
        Informar que la búsqueda ha fallado}
    else{
        EntradaAComparar = Entrada central de la Lista
        if (ValorObjetivo == EntradaAComparar){
            Informar que la búsqueda ha tenido éxito }
        if (ValorObjetivo < EntradaAComparar){
            Sublista = Parte de la Lista anterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
        if (ValorObjetivo > EntradaAComparar){
            Sublista = Parte de la lista posterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
             }
     }
```

Lista
Alice
Bill
Carol
David
Evelyn
Fred
George

EntradaAComparar

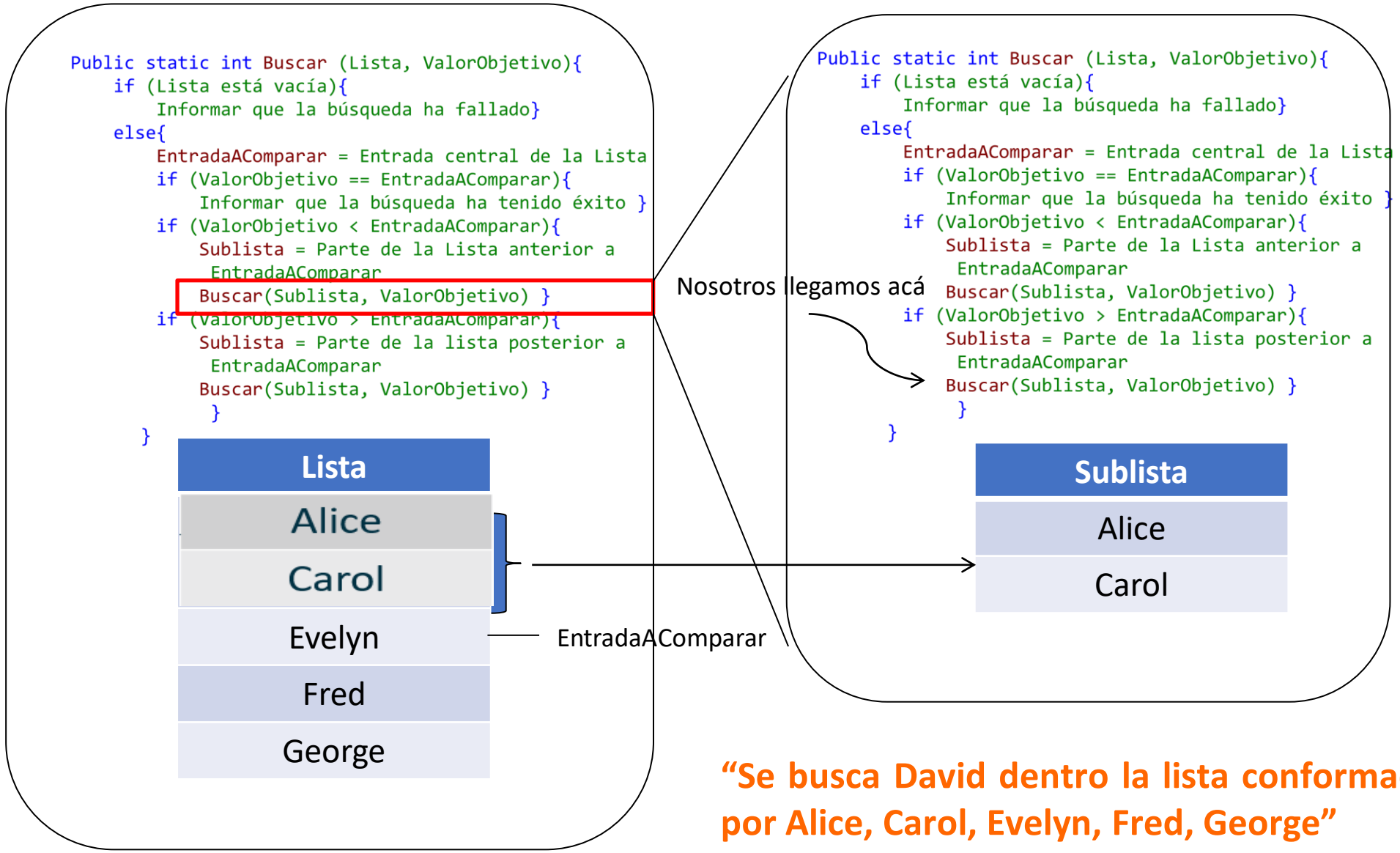
Aquí se retorna

```
Public static int Buscar (Lista, ValorObjetivo){
    if (Lista está vacía){
        Informar que la búsqueda ha fallado}
    else{
        EntradaAComparar = Entrada central de la Lista
        if (ValorObjetivo == EntradaAComparar){
            Informar que la búsqueda ha tenido éxito }
        if (ValorObjetivo < EntradaAComparar){
            Sublista = Parte de la Lista anterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
        if (ValorObjetivo > EntradaAComparar){
            Sublista = Parte de la lista posterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
             }
     }
```

Sublista
Alice
Bill
Carol

“Se busca Bill dentro la lista conformada por Alice, Bill, Carol, David, Evelyn, Fred, George”

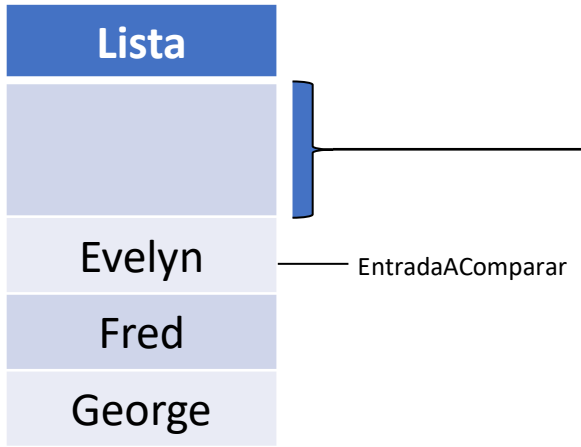
El algoritmo de búsqueda binaria – Caso 2



"Se busca David dentro la lista conformada por Alice, Carol, Evelyn, Fred, George"

El algoritmo de búsqueda binaria – Caso 2 continuación

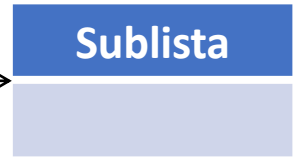
```
Public static int Buscar (Lista, ValorObjetivo){
    if (Lista está vacía){
        Informar que la búsqueda ha fallado}
    else{
        EntradaAComparar = Entrada central de la Lista
        if (ValorObjetivo == EntradaAComparar){
            Informar que la búsqueda ha tenido éxito }
        if (ValorObjetivo < EntradaAComparar){
            Sublista = Parte de la Lista anterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
        if (ValorObjetivo > EntradaAComparar){
            Sublista = Parte de la lista posterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
             }
     }
```




```
Public static int Buscar (Lista, ValorObjetivo){
    if (Lista está vacía){
        Informar que la búsqueda ha fallado}
    else{
        EntradaAComparar = Entrada central de la Lista
        if (ValorObjetivo == EntradaAComparar){
            Informar que la búsqueda ha tenido éxito }
        if (ValorObjetivo < EntradaAComparar){
            Sublista = Parte de la Lista anterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
        if (ValorObjetivo > EntradaAComparar){
            Sublista = Parte de la lista posterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
             }
     }
```

Aquí retornamos

```
Public static int Buscar (Lista, ValorObjetivo){
    if (Lista está vacía){
        Informar que la búsqueda ha fallado}
    else{
        EntradaAComparar = Entrada central de la Lista
        if (ValorObjetivo == EntradaAComparar){
            Informar que la búsqueda ha tenido éxito }
        if (ValorObjetivo < EntradaAComparar){
            Sublista = Parte de la Lista anterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
        if (ValorObjetivo > EntradaAComparar){
            Sublista = Parte de la lista posterior a
             EntradaAComparar
            Buscar(Sublista, ValorObjetivo) }
             }
     }
```



"Se busca David dentro la lista conformada por Alice, Carol, Evelyn, Fred, George"

Busqueda Binaria recursiva en Java

```java
package Semana2;
import javax.swing.*;
public class AED_Programa11 {
    public static void main(String[] args) {
        int[] lista={1,3,5,7,9,10,16,19,21,24,27,29,31,31,33,36,39,41,43,45};
        int elemento=Integer.parseInt(JOptionPane.showInputDialog("Ingrese el elemento que desea buscar del 1 al 45"));
        int resultado=busqueda_binaria_recursiva(lista, elemento);
        if(resultado!=-1){
        System.out.println("El numero buscado está en la posición "+ resultado);
        }else{
            System.out.println("El numero buscado no está en la lista");
        }
    }
    public static int busqueda_binaria_recursiva(int[] lista, int elemento){
        return busqueda_binaria_recursiva(lista,elemento,0,lista.length-1);
    }
    public static int busqueda_binaria_recursiva(int[] lista, int elemento, int menor, int mayor){
        int medio=menor+(mayor-menor)/2;
        if((menor>=mayor)&&(lista[menor]!=elemento)){
            return -1;
        }else if(lista[medio]==elemento){
            return medio;
            }else if(elemento<lista[medio]){
                return busqueda_binaria_recursiva(lista,elemento,menor,medio-1);
            }
                return busqueda_binaria_recursiva(lista,elemento,medio+1,mayor);

        }
    }
```

Cálculo de tiempo de algoritmo recursivo

- Paso 1: Determinar la entrada
- Paso 2: Determinar el tamaño de la entrada
- Paso 3: Definir una recurrencia para T(n)
- Paso 4: Obtener una definición no recursiva para T(n)
- Paso 5: Determinar orden de tiempo de ejecución
- Paso 6: Hacer prueba por inducción para ver que las expresiones para T(n) de (3) y (4) son equivalentes.

Factorial

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

Factorial

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

En el caso base, testear n=0 junto con retornar 1 toma tiempo c_1, pues son dos operaciones primitivas.

En el caso recursivo, testear n=0, restar 1 a n, invocar fact (sin contar el tiempo que toma ejecutar fact(n-1), multiplicar por n y retornar toma tiempo c_2, pues son todas operaciones primitivas.

Factorial

- <u>Paso 1:</u> Entrada es “n”
- <u>Paso 2:</u> Tamaño de la entrada es “n”
- <u>Paso 3:</u> Definir recurrencia para T(n):

$$T(n) = \begin{cases} c_1 & si \quad n = 0 \\ c_2 + T(n-1) & si \quad n > 0 \end{cases}$$

Factorial

- Paso 4: Derivar definición no recursiva de T(n):

$T(n) = c_2 + T(n-1) = c_2 + (c_2 + T((n-1)-1)$

$= 2c_2 + T(n-2) = 2c_2 + (c_2 + T((n-2)-1)$

$= 3c_2 + T(n-3) = 3c_2 + (c_2 + T((n-3)-1)$

$= 4c_2 + T(n-4) = \ldots =$

$= ic_2 + T(n-i)$ (1)

Termina cuando $n-i = 0$, luego $n = i$ (2)

Reemplazo (2) en (1) y obtengo:

$$T(n) = nc_2 + T(0) = nc_2 + c_1$$

- Paso 5: Obtener orden de tiempo de ejecución:

$$T(n) \text{ es } O(n)$$

Factorial

•<u>Paso 6:</u> Prueba por inducción de $T(n) = nc_2+c_1$

<u>Caso base:</u> $T(0) = c_1 = 0c_2 + c_1$

<u>Caso inductivo:</u>

$T(n) = c_2 + T(n-1) =$ *(x definición recursiva de T(n))*

$= c_2 + ((n-1)c_2 + c_1)$ *(x hipótesis inductiva)*

$= c_2 + (nc_2 - c_2 + c_1)$ *(x distributividad de *)*

$= c_2 + nc_2 - c_2 + c_1$ *(x asociatividad)*

$= nc_2 + c_1$ *(anulo c_2 positivo y negativo)*

Búsqueda binaria

Problema: Buscar entero “x” en arreglo de enteros ordenado “a” de “n” componentes

```
public static int bsearch( int [] a, int n, int x ) {
    return bsearch_aux( a, 0, n-1, x );
}
private static int bsearch_aux(int [] a, int ini, int fin, int x ) {
    if( ini <= fin ) {
            int medio = (ini + fin) / 2;
            if( a[medio] == x ) return medio;
            else if( a[medio] > x ) then
                        return bsearch_aux( a, ini, medio-1, x)
                  else
                        return bsearch_aux( a, medio+1, fin , x)
    }
    else  return -1;  // x no está en el arreglo
}
```

- Paso 1: Entrada: Arreglo a y x
- Paso 2: Tamaño de entrada:

  n = cantidad de componentes de a
- Paso 3: Definición recursiva de T(n):

$$T(n) = \begin{cases} c_1 & ,si \quad n = 0 \\ c_2 + T(\frac{n-1}{2}) & ,si \quad n \geq 1 \end{cases}$$

- <u>Paso 4:</u> Obtener definición no recursiva de T(n)

$T(n) = c_2 + T((n-1)/2) = c_2 + (c_2 + T(((n-1)/2 – 1)/2)))$

$= 2c_2 + T((n-3)/4) = 2c_2 + (c_2 + T(((n-3)/4 – 1)/2)))$

$= 3c_2 + T((n-7)/8) = 3c_2 + (c_2 + T(((n-7)/8 – 1)/2)))$

$= 4c_2 + T((n-15)/16) = 4c_2 + (c_2 + T(((n-15)/16 – 1)/2)))$

$= 5c_2 + T((n-31)/32) = \ldots$

$= ic_2 + T((n-(2^i-1))/2^i)$ (1)

Termina cuando $(n-(2^i-1))/2^i = 0$, luego $n-(2^i-1) = 0$.

Entonces, $n-2^i+1=0$; por lo tanto, $n+1 = 2^i$ y $i = \log_2(n+1)$. (2)

Reemplazo (2) en (1):

$T(n) = \log_2(n+1)c_2 + T(0) = \log_2(n+1)c_2 + c_1$.

- Paso 5: Dar orden de tiempo de ejecución:

  $T(n) = \log_2(n+1)c_2 + c_1$ es $O(\log_2(n+1))$.

- Paso 6: Prueba inductiva (por inducción transfinita)

Caso base: $T(0) = c_1 = \log_2(0+1)c_2 + c_1 = c_1$

Caso inductivo: $T(n) = c_2 + T((n-1)/2)$ *(x def. T(n) rec.)*

$= c_2 + (\log_2((n-1)/2+1)c_2 + c_1)$ *(x hipótesis inductiva)*

$= c_2 + (\log_2((n-1+2)/2))c_2 + c_1$

$= c_2 + (\log_2(n+1) - \log_2(2))c_2 + c_1$

(xq log (a/b) = log(a) – log(b))

$= c_2 + (\log_2(n+1) - 1)c_2 + c_1$

$= c_2 + \log_2(n+1)c_2 - c_2 + c_1$

$= \log_2(n+1)c_2 + c_1$.

Teorema Maestro

Si tenemos un algoritmo cuya ecuación de recurrencia es:

$$\mathcal{T}(n) = A\mathcal{T}\left(\frac{n}{B}\right) + \mathcal{O}\left(n^C\right)$$

A: cantidad de llamados recursivos

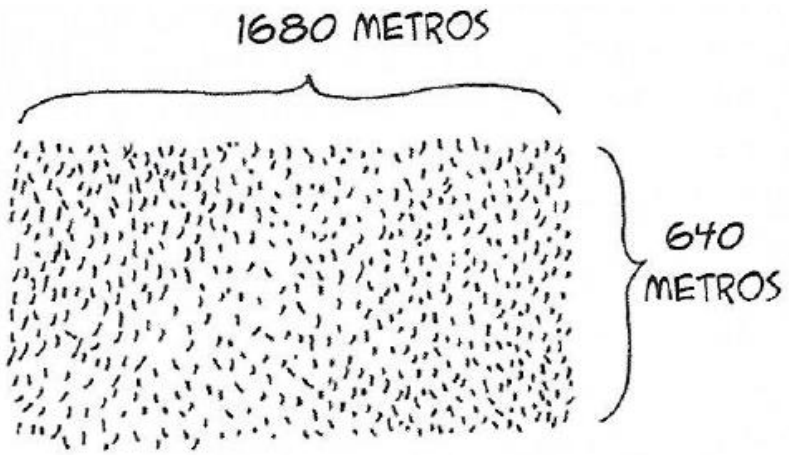
B: proporción del tamaño original con el que llamamos recursivamente

$\mathcal{O}(n^C)$: el costo de *partir y juntar* (todo lo que no son llamados recursivos)

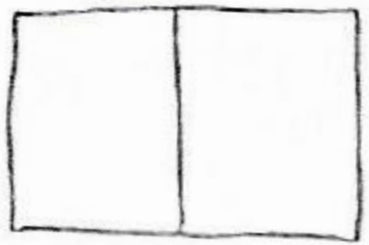
$$\text{Si } \log_B(A) \begin{cases} < & \rightarrow \mathcal{T}(n) = \mathcal{O}\left(n^C\right) \\ = C & \rightarrow \mathcal{T}(n) = \mathcal{O}\left(n^C \log_B n\right) = \mathcal{O}\left(n^C \log n\right) \\ > & \rightarrow \mathcal{T}(n) = \mathcal{O}\left(n^{\log_B A}\right) \end{cases}$$

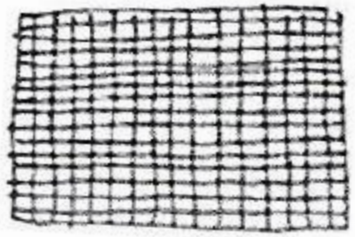
Recursividad: Divide y Vencerás (D&V)

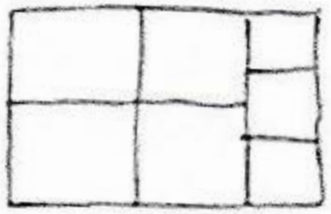
Supón que eres un granjero con una parcela de tierra



Quieres dividir la tierra uniformemente en parcelas cuadradas. Quieres parcelas lo más grande posibles, así que ninguna de las siguientes funcionará


LAS CAJAS NO SON CUADRADAS

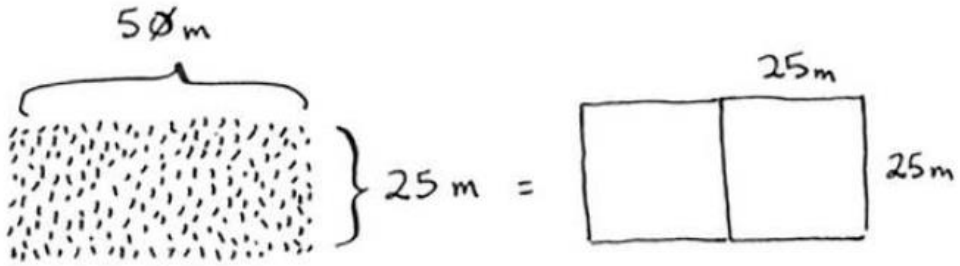

LAS CAJAS SON DEMASIADO PEQUEÑAS


TODAS LAS CAJAS TIENEN QUE SER DEL MISMO TAMAÑO

Para resolver un problema con D&V hay dos pasos:

1. ***Define el caso base. Este debe ser el caso más sencillo posible***
2. ***Divide o disminuye el tamaño del problema hasta que encuentres el caso base***

El caso más sencillo sería que un lado sea múltiplo del otro

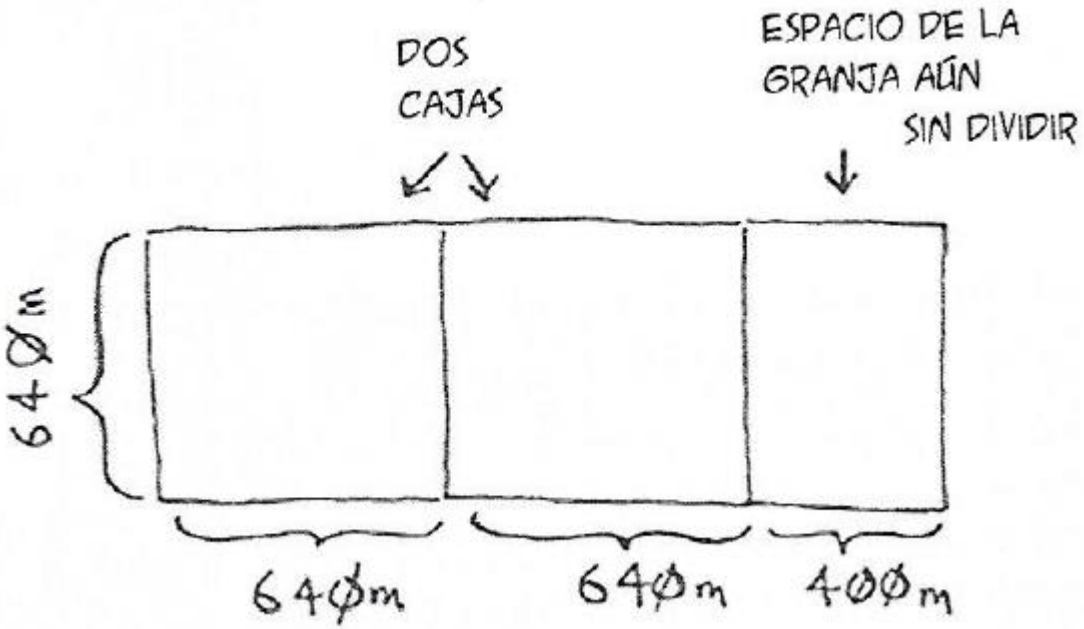


En este caso la parcela más grande sería 25mx25m. Se necesita dos de esas para dividir la tierra

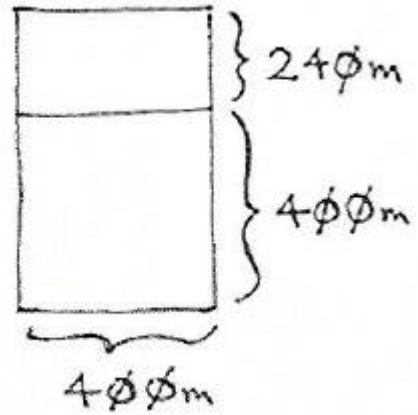
Recursividad: Divide y Vencerás (D&V)

Ahora tienes que encontrar el caso recursivo. Aquí es donde aparece Divide y Vencerás Cada llamada recursiva tienes que reducir el problema

¿Cómo disminuyes tu problema aquí? Comencemos por marcar las parcelas más grandes que pudiéramos usar

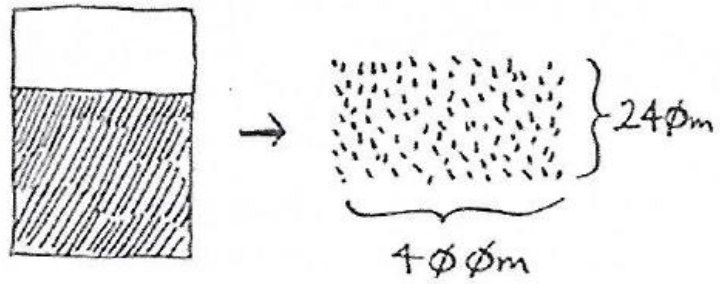


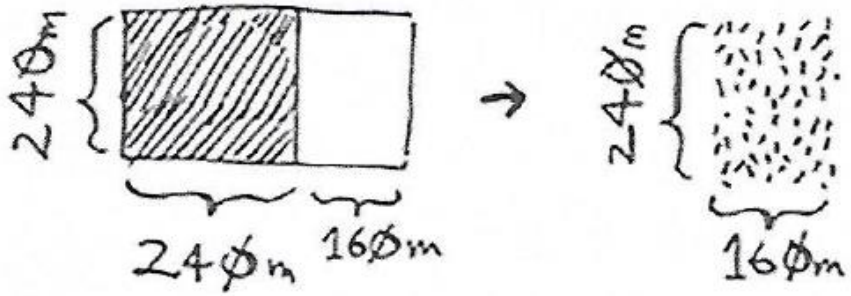
Según el gráfico aún queda tierra por dividir ¿porqué no aplicar el mismo algoritmo a este segmento?



Aquí puedes dibujar una parcela para obtener otro segmento aún más pequeño 240x160m

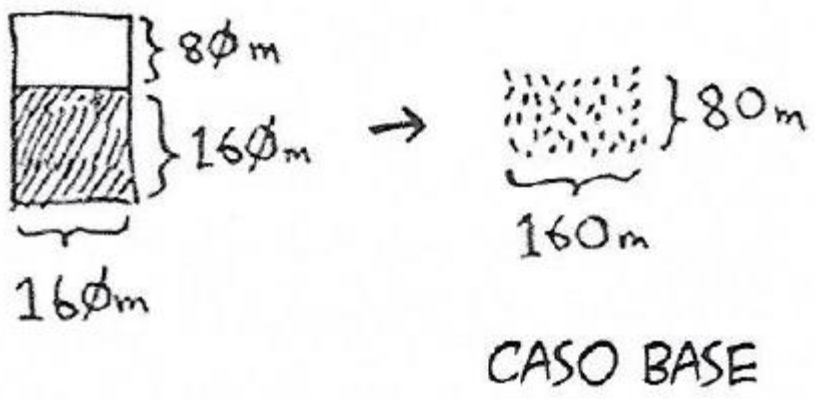
Esto nos deja con un segmento menor de 400 x 240m

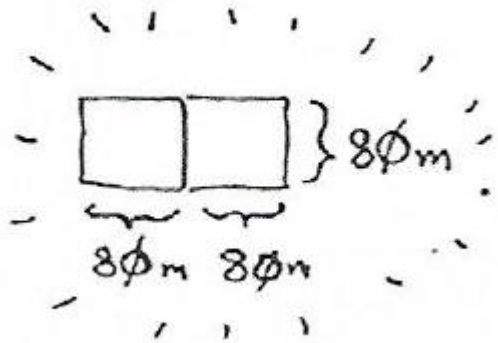




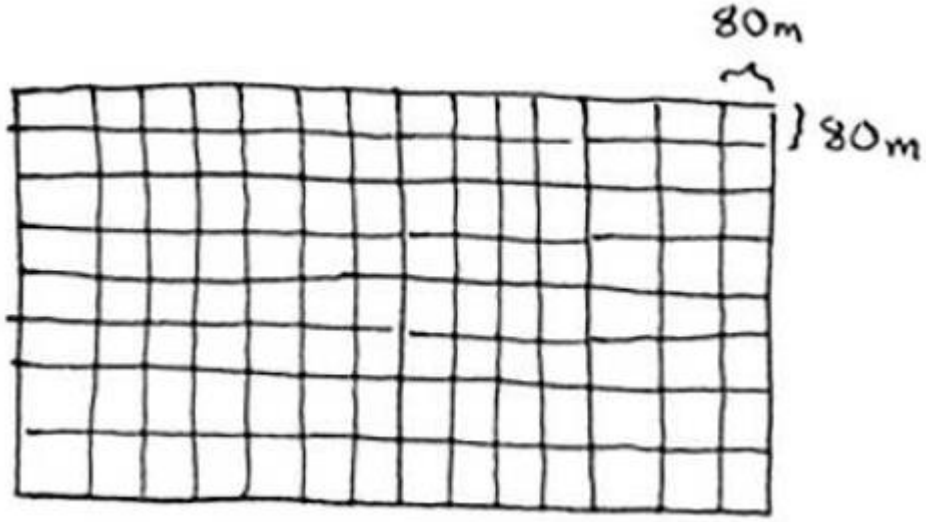
Recursividad: Divide y Vencerás (D&V)

Luego dibujas una parcela para obtener otro segmento aún más pequeño. !Ey¡, ahora estas en el caso base 80 es un divisor de 160. Si divides este segmento no queda ningún espacio a repartir





Entonces, para la granja original, el mayor tamaño de parcela que puedes utilizar es de 80 x 80 m



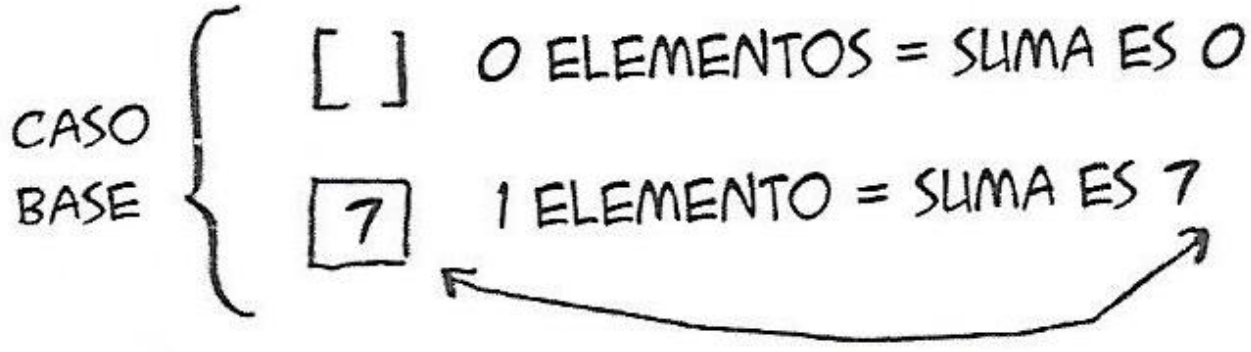
Recursividad: Divide y Vencerás (D&V)

Ejercicios

Recibes un arreglo de números

Crear una función recursiva que sume los elementos de un arreglo. (es fácil con un ciclo repetitivo). Tomar en cuenta los siguientes pasos:

Paso 1: Encuentra el caso base ¿Cuál es el arreglo más sencillo que puedes recibir? Si obtienes un arreglo de 0 o 1 elemento, es bien fácil de sumar.



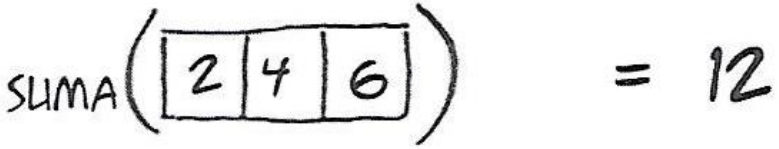
Recursividad: Divide y Vencerás (D&V)

Ejercicios

Recibes un arreglo de números

Crear una función recursiva que sume los elementos de un arreglo. (es fácil con un ciclo repetitivo). Tomar en cuenta los siguientes pasos:

Paso 2: Necesitas acercarte a un arreglo vacío con cada llamada recursiva. ¿Cómo reduces el tamaño del problema?



Es lo mismo que

Tu función suma puede funcionar como sigue:

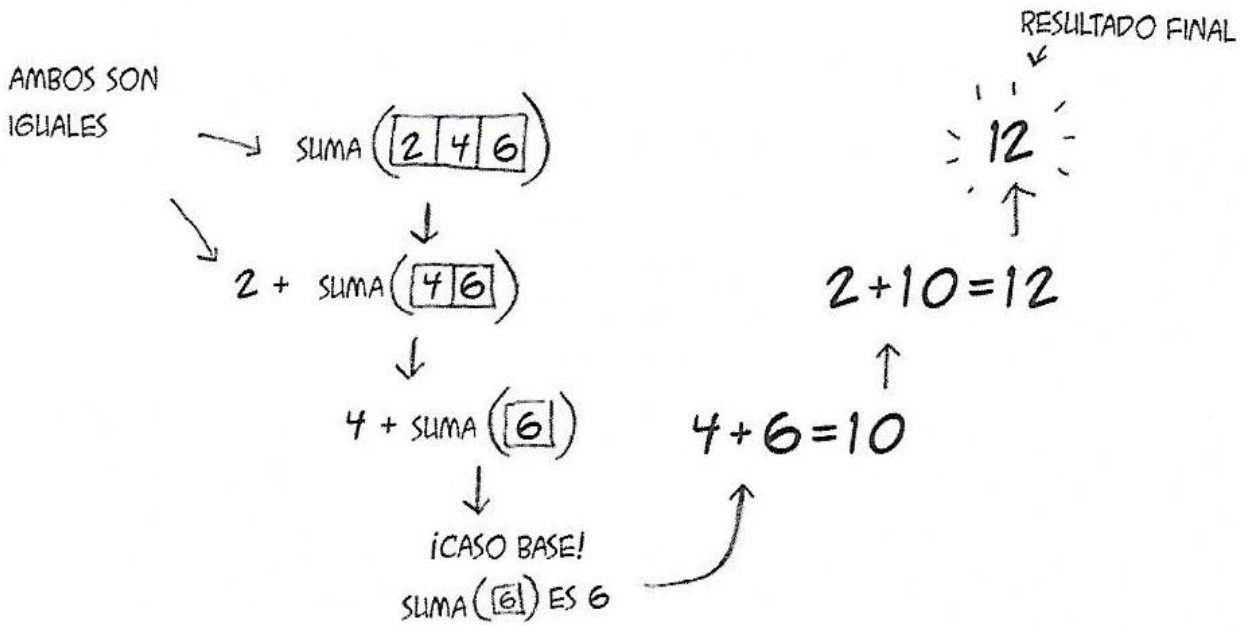
Recursividad: Divide y Vencerás (D&V)

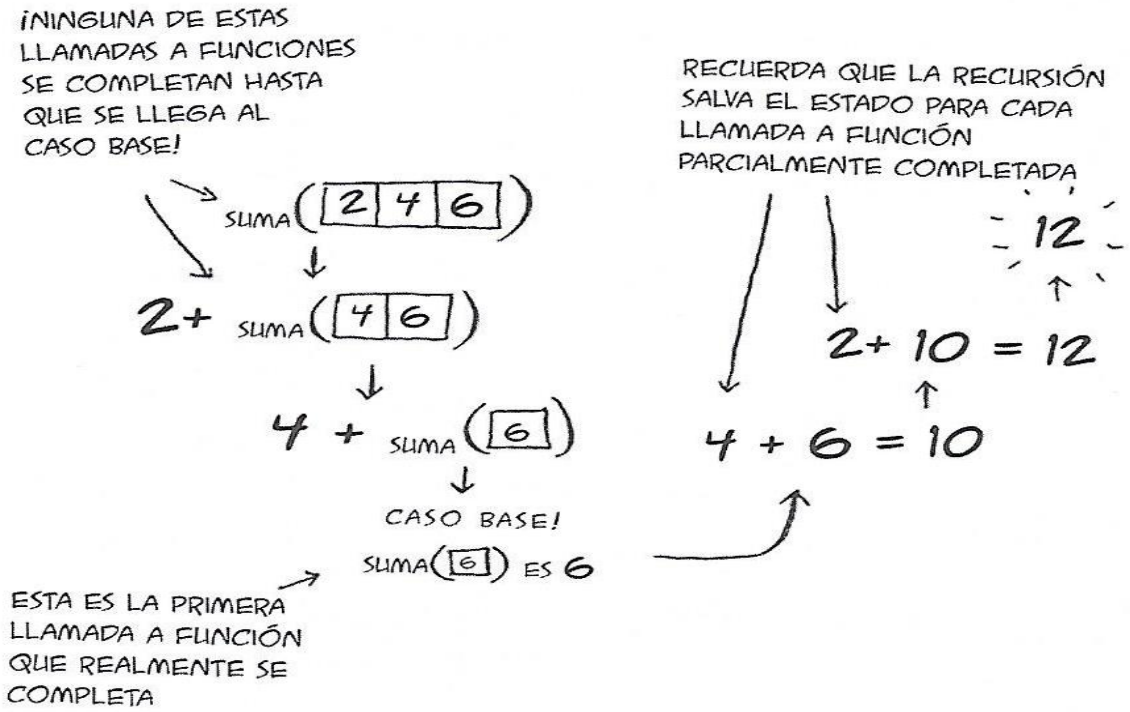
Ejercicios

Recibes un arreglo de números

Crear una función recursiva que sume los elementos de un arreglo.

La función en acción:





Ejercicio

Crear una función recursiva que multiplique los elementos de un arreglo

Pista: cuando escribes una función recursiva que involucra un arreglo, el caso base es por lo general un arreglo vacío o de un solo elemento.

Ejercicio

```java
package pClases;

import java.util.Scanner;

public class Recursividad_Arreglos {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Cuantos numeros quiere ingresar?");
        int arr[] = new int[sc.nextInt()];

        System.out.println("Cargar los Valores :");
        for (int i = 0; i < arr.length; i++) {
            arr[i] = sc.nextInt();
        }
        // LLAMADA A LA FUNCIÓN PARA MULTIPLICAR LOS VALORES DE UN ARRAY DE FORMA RECURSIVA
        System.out.println("La multiplicación de los valores es "+ mulValores(arr, arr.length - 1));
    }
```

Ejercicio

```
    // EL MÉTODO RECOGERÁ EL ARRAY Y LA POSICIÓN DEL ELEMENTO A MULTIPLICAR
    public static int mulValores(int array[], int posArray) {
        // INICIALIZAMOS UNA VARIABLE CON LA POSICIÓN DEL ARRAY ((NO ES NECESARIO))
        int tam = posArray;
        int rta;
        /* COMPROBAMOS QUE ÉL TAMAÑO DEL NO SEA CERO, YA QUE SI EL TAMAÑO
        ES CERO INTENTARÁ EN LA SIGUIENTE LLAMADA ENTRAR EN LA POSICIÓN -1
        DEL ARRAY DANDO UN ERROR */
        if (tam == 0) {
            return array[tam];
        } else {
            /* SI EL TAMAÑO NO ES IGUAL A CERO, MULTIPLICAMOS EL RESULTADO
            DEL VALOR PASADO POR PARÁMETRO POR EL VALOR DE LA
            POSICIÓN ANTERIOR */
            rta = (array[tam]) * mulValores(array, tam - 1);

        }
        return rta;
    }
}
```

Recursividad: Divide y Vencerás (D&V)

Ejercicios

- Escribir una función recursiva para contar los elementos de una lista.
- Escribir una función recursiva para encontrar el mayor de una lista.

Pista: Cuando escribes una función recursiva que involucra un arreglo, el caso base es por lo general un arreglo vacío o de un solo elemento.

Guía de Laboratorio

¿Preguntas?

Resumiendo y Repasando…

- Recursión es que un método se llama a si mismo.
- Cada método recursivo tiene dos casos: el caso base y el caso recursivo.
- La recursión implica repetir el conjunto de sentencias como una subtarea de sí mismo.
- La ejecución de un procedimiento conduce a una nueva ejecución del mismo procedimiento.
- Se forman múltiples activaciones del procedimiento, y todas menos una, está esperando que se complete el resto de activaciones del procedimiento.

Resumiendo y Repasando…

- Todas las llamadas a los métodos van a la pila de llamadas.
- La pila de llamadas puede crecer mucho, lo cual puede ocupar mucha memoria
- Divide & Vencerás funciona dividiendo el problema en piezas más pequeñas. Si estas usando D&V en una lista, el caso base probablemente sea un arreglo vacío o un arreglo de un elemento.

FIN