

## Tema 3: Algoritmos recursivos

- ❑ ¿qué es?
- ❑ en programación
- ❑ pila de contextos
- ❑ vs iteración
- ❑ backtracking

# ¿qué es?

---

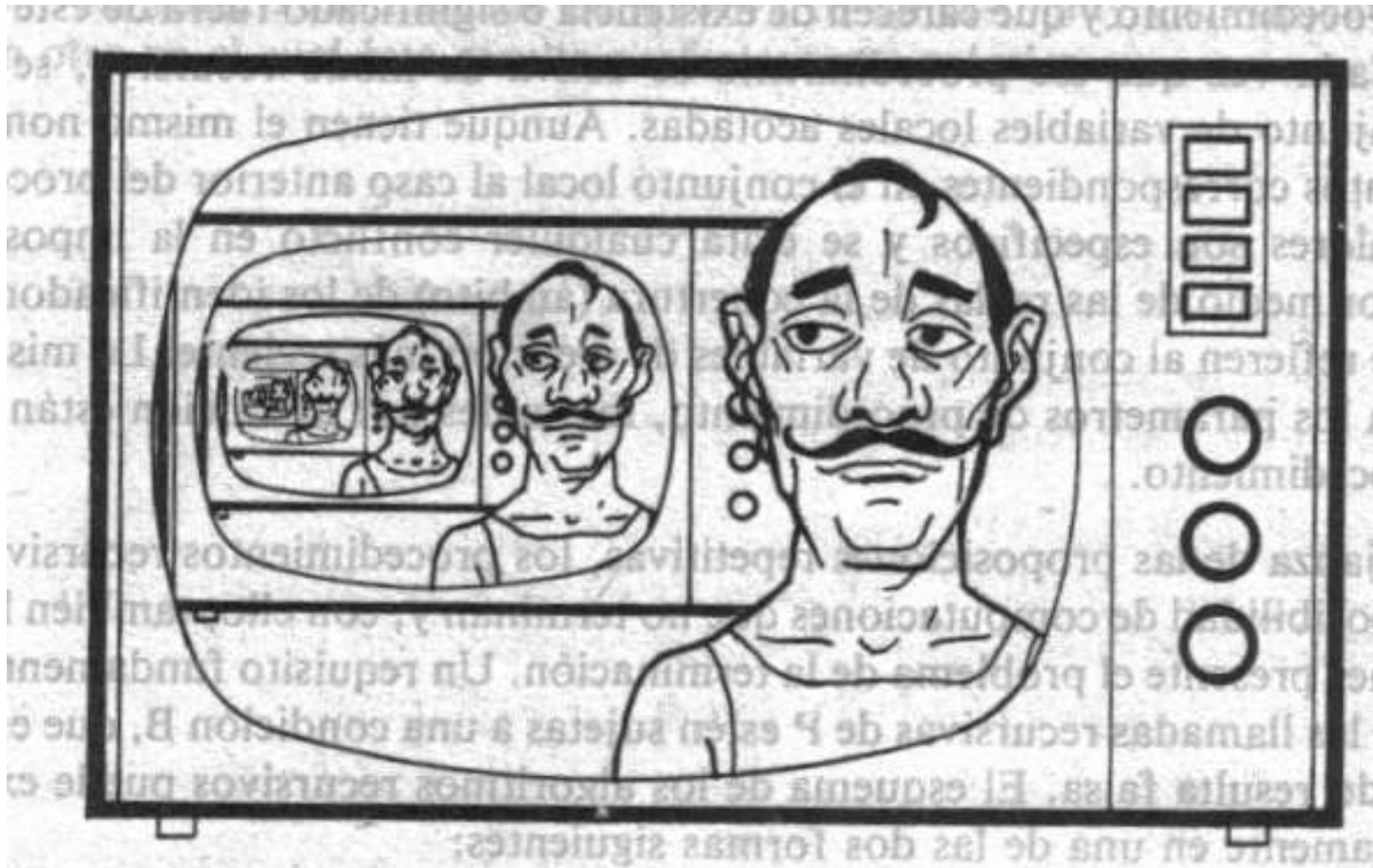


## ¿qué es?

---



## ¿qué es?



*Un objeto es recursivo cuando forma parte de si mismo, o se define en función de si mismo*

## ¿qué es?

---

- Se usa mucho en matemáticas
  - Números naturales
    - 0 es un número natural
    - El sucesor inmediato de un número natural también es un número natural
  - Factorial:
 
$$0! = 1$$

$$n! = n * (n-1)! \quad (\forall n > 0)$$
  - Fibonacci
 

Si  $n=0$  o  $n=1$ ,  $\text{fibonacci}(n) = n$

$$\forall n > 1, \text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

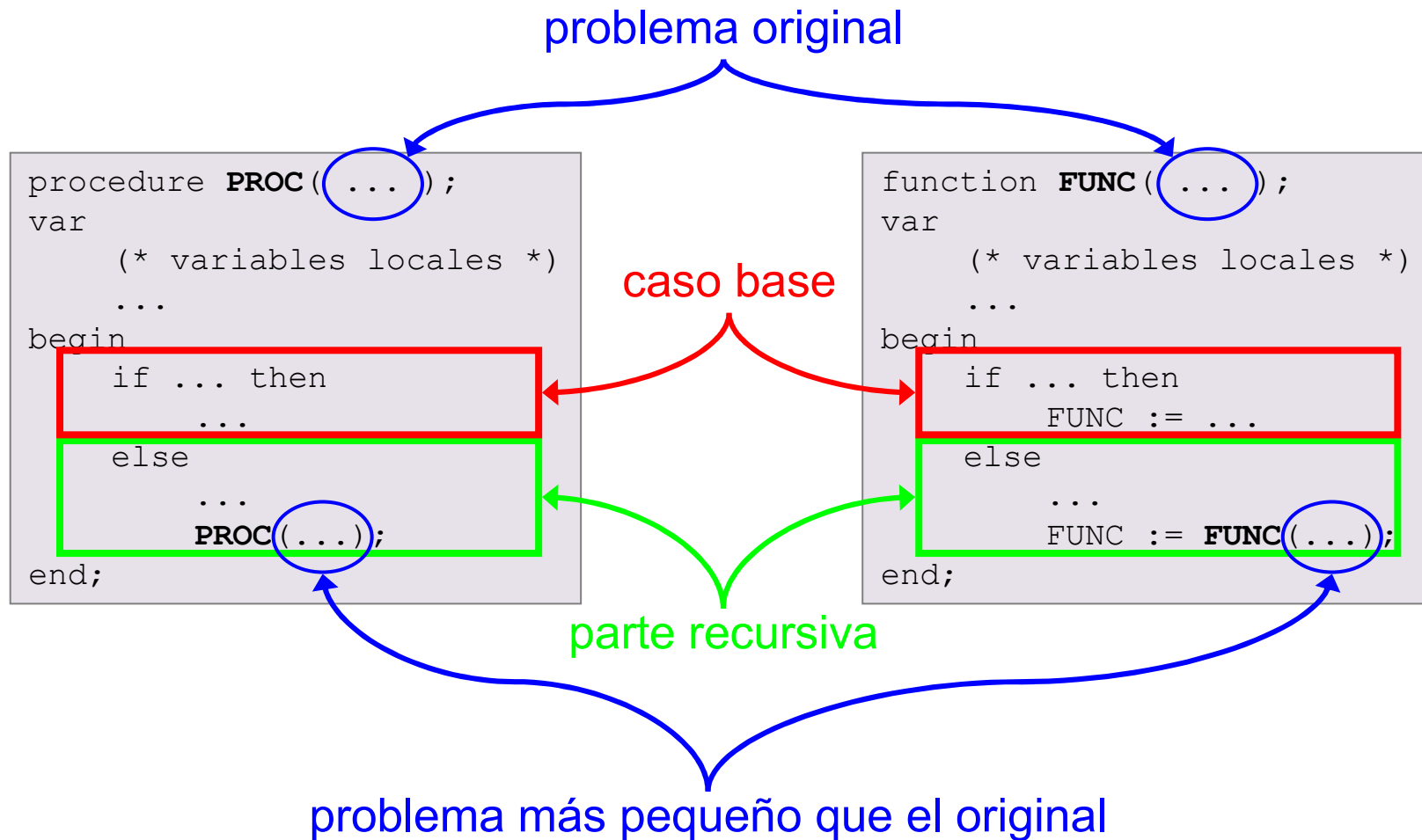
## en programación

---

- Muy utilizada en programación
- Herramienta muy potente
- Muchos algoritmos se definen mejor en términos recursivos
- Un procedimiento o una función es recursivo si contiene llamadas a sí mismo
- Todo problema recursivo se puede convertir en iterativo

# en programación

procedimiento recursivo | función recursiva

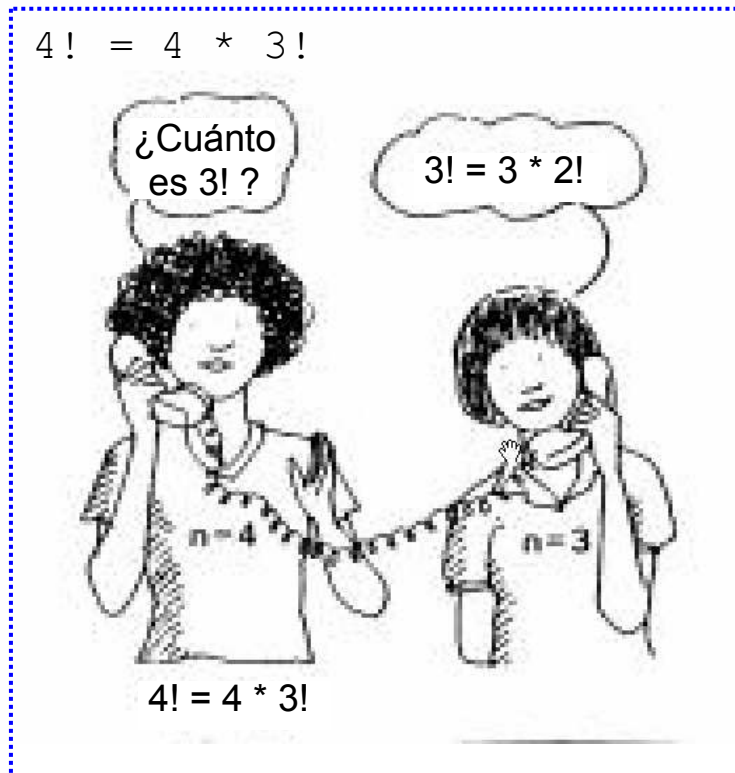


# en programación > factorial

Caso base	$0! = 1$
Recursividad	$n! = n * (n-1)!, \forall n > 0$

```
function fact( n: integer ): integer;
begin
  if n=0 then
    fact := 1
  else
    fact := n * fact( n-1 );
end;
```

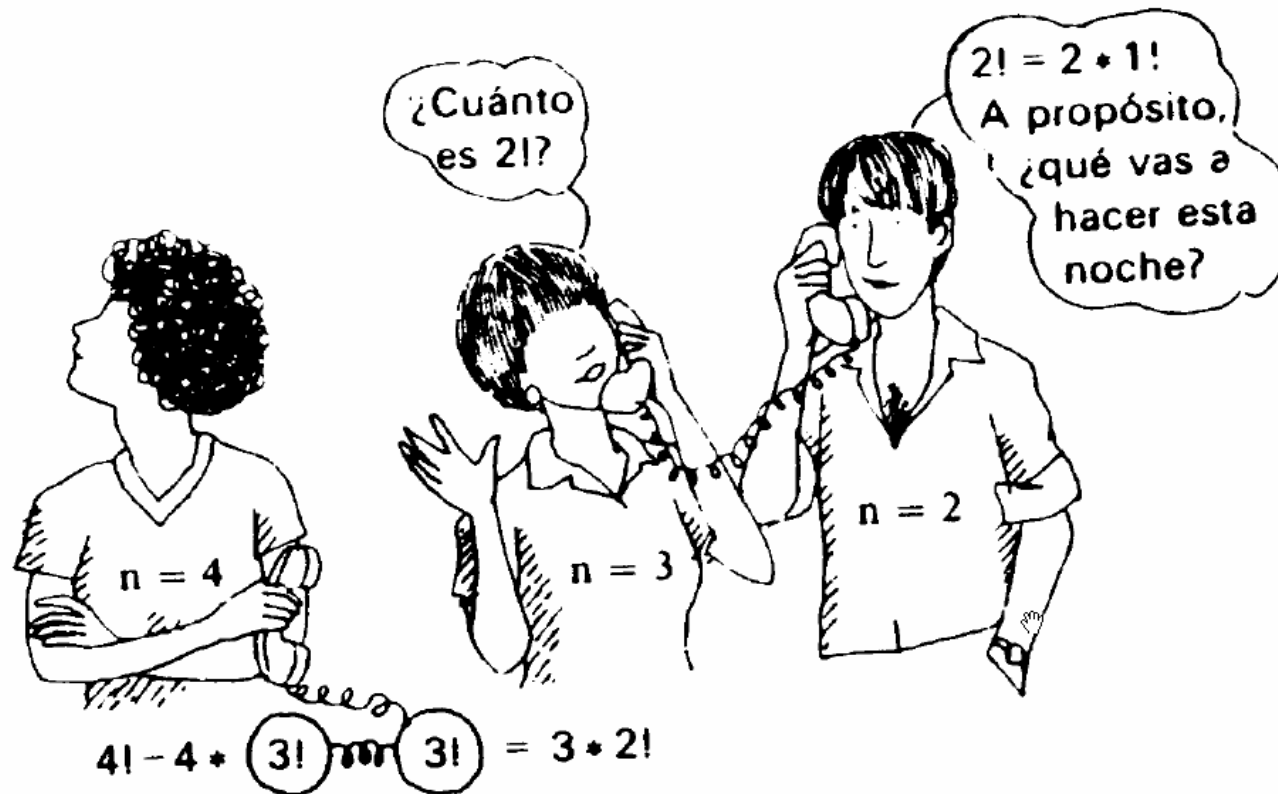
¿Cómo funciona la recursividad?





# ¿cómo funciona?

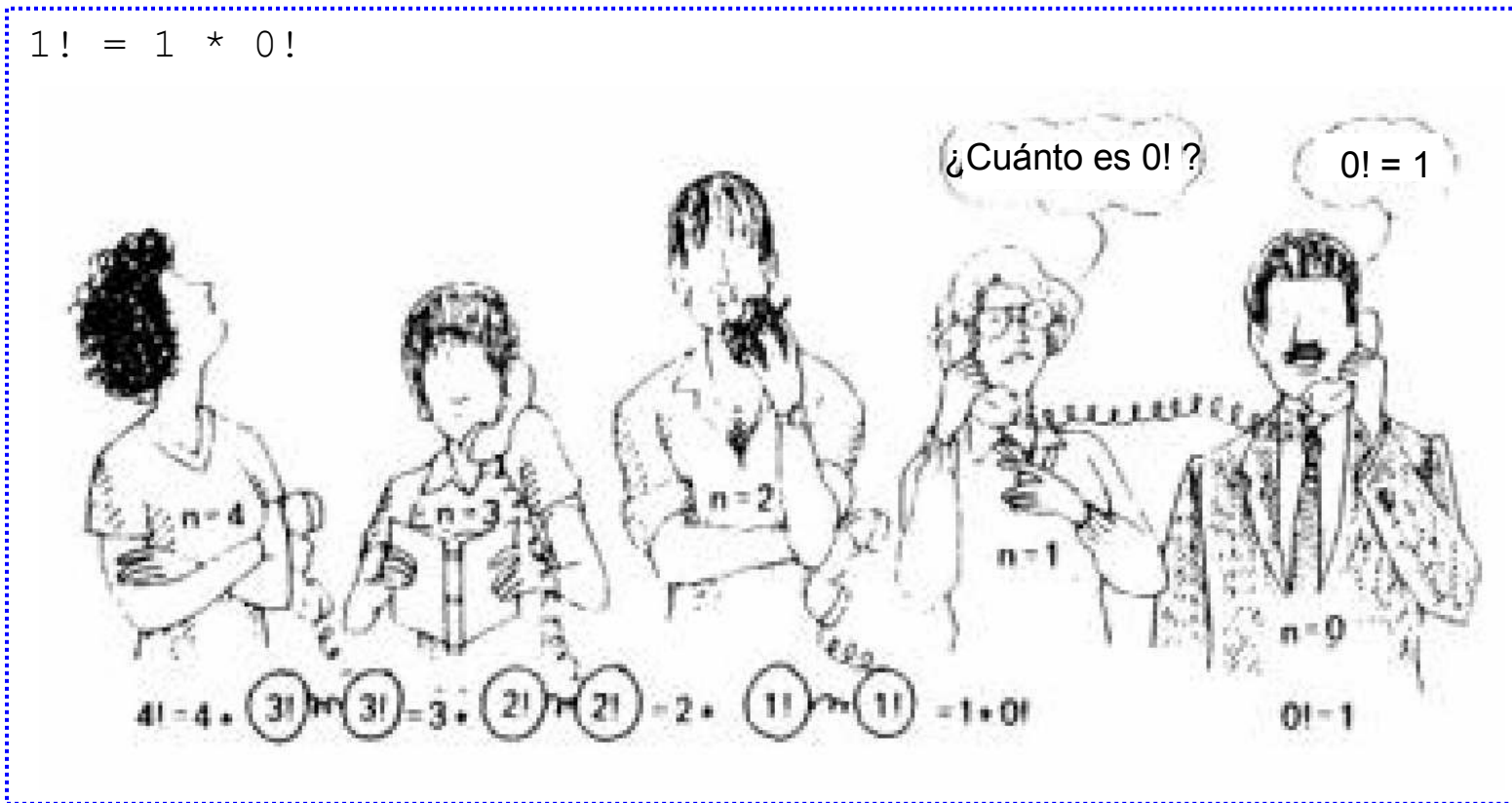
$$3! = 3 * 2!$$



# ¿cómo funciona?



# ¿cómo funciona?



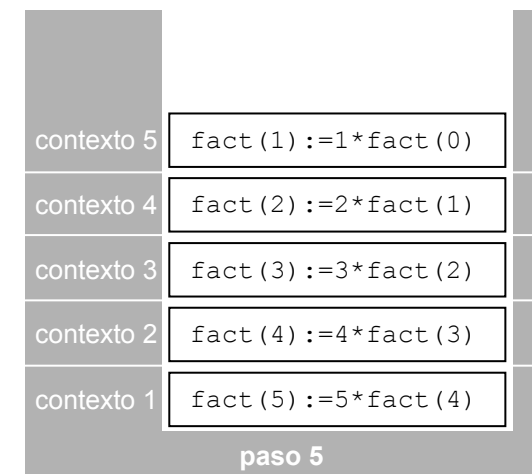
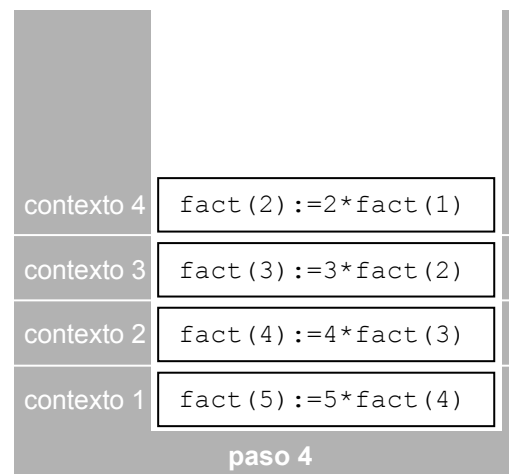
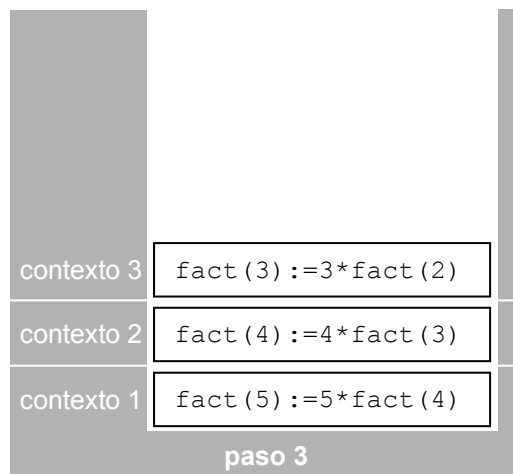
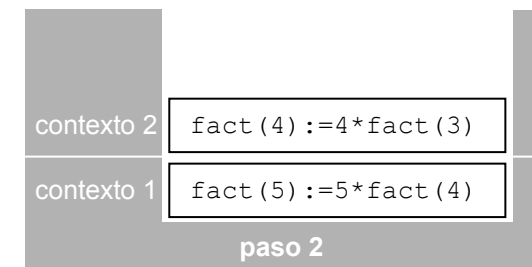
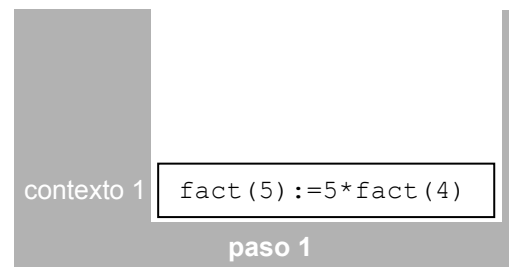
# ¿cómo funciona?



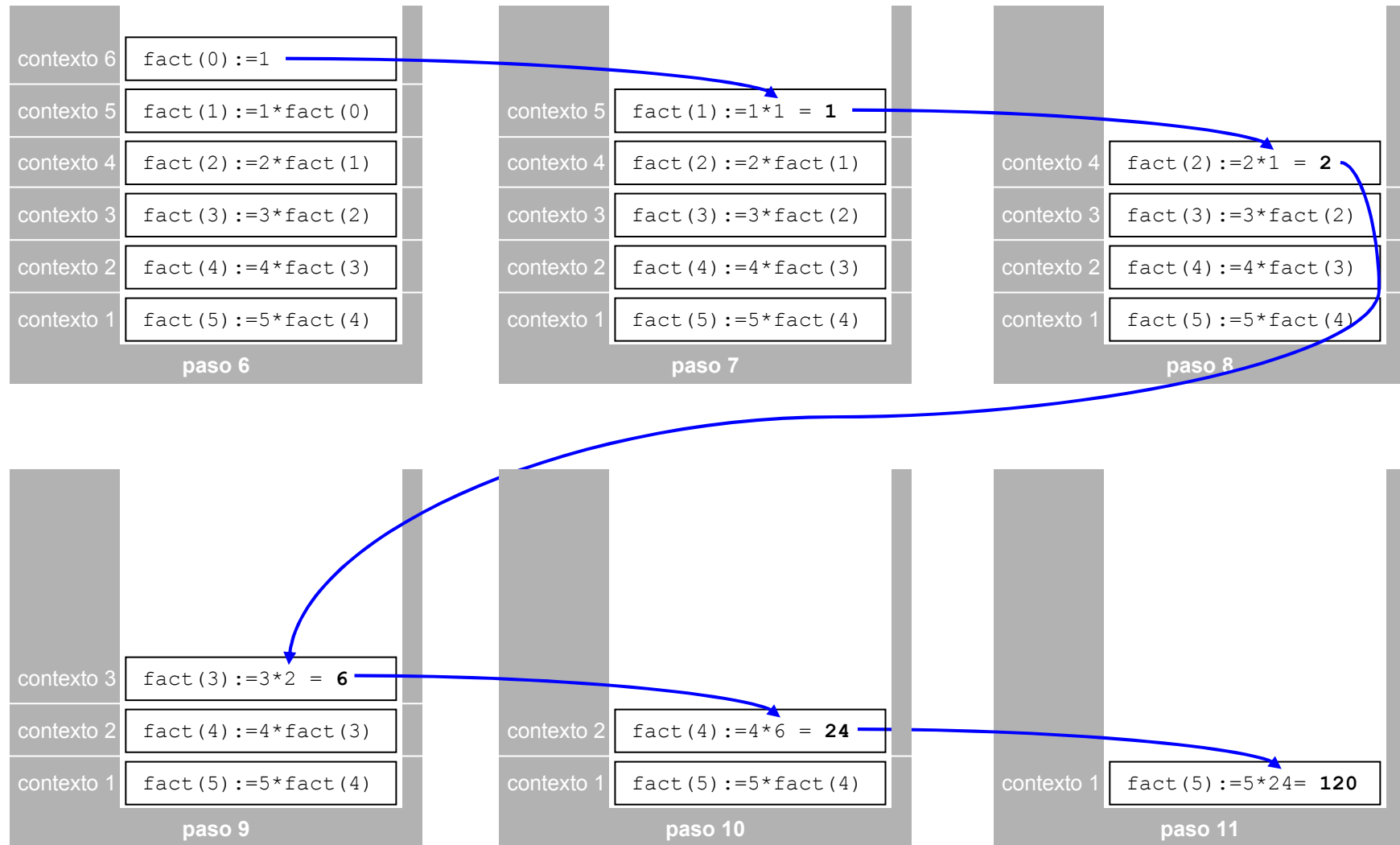
# pila de contextos

Caso base	$0! = 1$
Recursividad	$n! = n * (n-1)!, \forall n > 0$

```
function fact( n: integer ): integer;
begin
  if n=0 then
    fact := 1
  else
    fact := n * fact( n-1 );
end;
```



# pila de contextos



# condiciones de recursividad

## Caso base

Siempre hay una condición de salida donde se rompe la recursividad (la recursividad infinita provoca un desbordamiento de la pila de memoria)

```
function fact( n: integer ): integer;  
begin  
    if n=0 then  
        fact:= 1  
    else  
        fact:= n * fact( n-1 );  
    end;
```

## Convergencia

En cada fase de la recursividad estamos más cerca del caso base (cada llamada conduce a problemas mas pequeños)

```
function fact( n: integer ): integer;  
begin  
    if n=0 then  
        fact:= 1  
    else  
        fact:= n * fact( n-1 );  
    end;
```

n-1 converge a 0 (caso base)

## condiciones de recursividad › errores típicos

### No existe **caso base**

Se produce una recursividad infinita y por tanto un desbordamiento de la pila de memoria

```
function fact( n: integer ): integer;  
begin  
    fact:= n * fact( n-1 );  
end;
```

### No hay **convergencia**

En cada paso estamos más lejos de la solución y la pila se acaba desbordando

```
function fact( n: integer ): integer;  
begin  
    if n=0 then  
        fact:= 1  
    else  
        fact:= n * fact( n+1 );  
    end;
```

n+1 no converge a 0



1. ¿Existe una **salida no recursiva**?  
¿funciona bien el algoritmo para este caso?
2. ¿Cada llamada recursiva se refiera a un **caso más pequeño** del problema original?
3. ¿Es correcta la solución en aquellos **casos no base**?

## ejercicio › multiplicación rusa

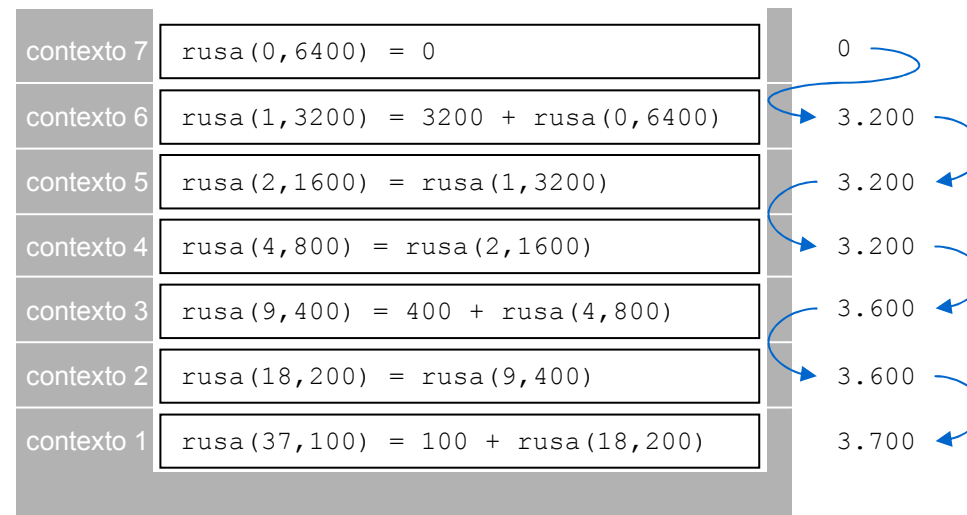
Dibujar la pila de contextos de activación del algoritmo para la multiplicación rusa de los números 37 y 100

Caso base	$a=0 \Rightarrow 0$
Recursividad	$\text{si } a \text{ es par} \Rightarrow \text{rusa}(a \text{ div } 2, b*2)$ $\text{si } a \text{ es impar} \Rightarrow b + \text{rusa}(a \text{ div } 2, b*2)$

```
function rusa( a, b: integer ): integer;  
begin  
  if a = 0 then  
    rusa:= 0  
  else begin  
    if a mod 2 = 0 then  
      rusa:= rusa( a div 2, b*2 )  
    else  
      rusa:= b + rusa( a div 2, b*2 )  
    end  
  end  
end;
```

## ejercicio › multiplicación rusa › pila de contextos

Caso base	$a=0 \Rightarrow 0$
Recursividad	si $a$ es par $\Rightarrow \text{rusa}(a \text{ div } 2, b*2)$ si $a$ es impar $\Rightarrow b + \text{rusa}(a \text{ div } 2, b*2)$



## recursividad vs iteración

### Usamos recursividad

- La solución recursiva sea sencilla
- Los algoritmos que por naturaleza sean recursivos (la solución iterativa implementa una gestión en pila)

```
function factorial( n: integer ): integer;  
begin  
    if n=0 then  
        factorial:= 1  
    else  
        factorial:= n * factorial( n-1 );  
    end;  
end;
```

### Usamos iteración

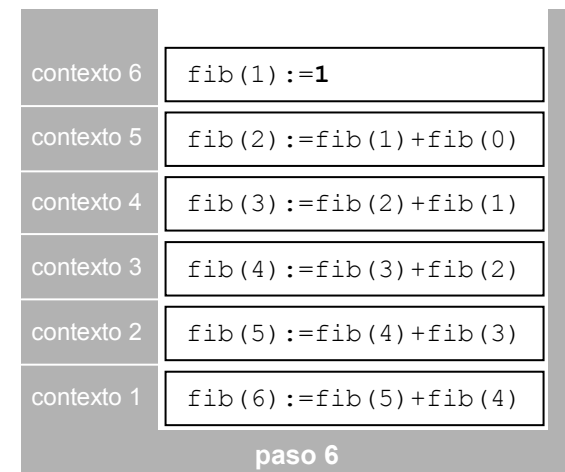
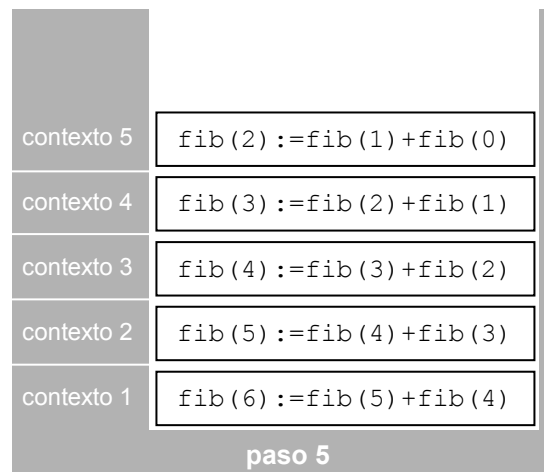
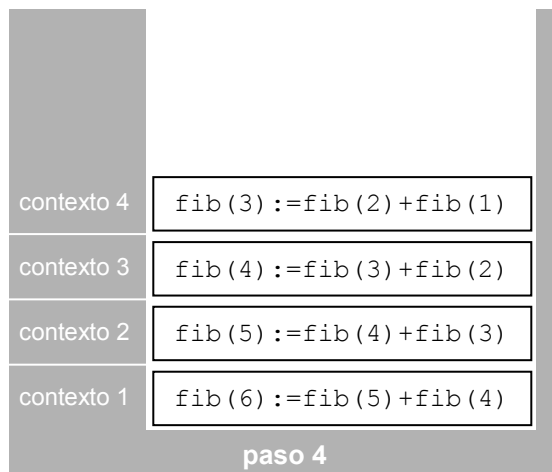
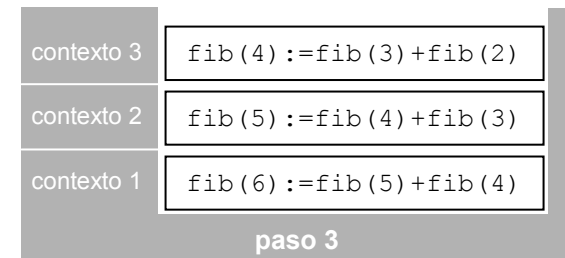
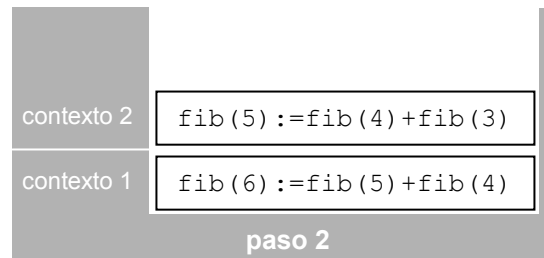
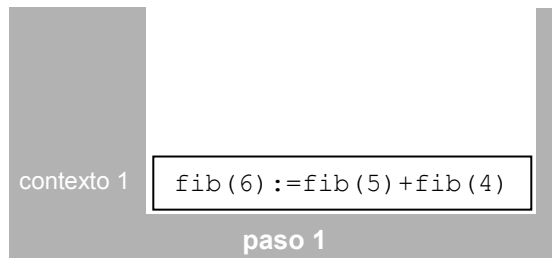
- Cuando la solución iterativa sea más obvia
- Cuando la solución recursiva no sea efectiva

```
function factorialIterativo( n: integer ): integer;  
var  
    i,f: integer;  
begin  
    i:=0; f:=1;  
    while i<n do begin  
        i:=i+1;  
        f:=f*i;  
    end;  
    factorialIterativo:=f;  
end;
```

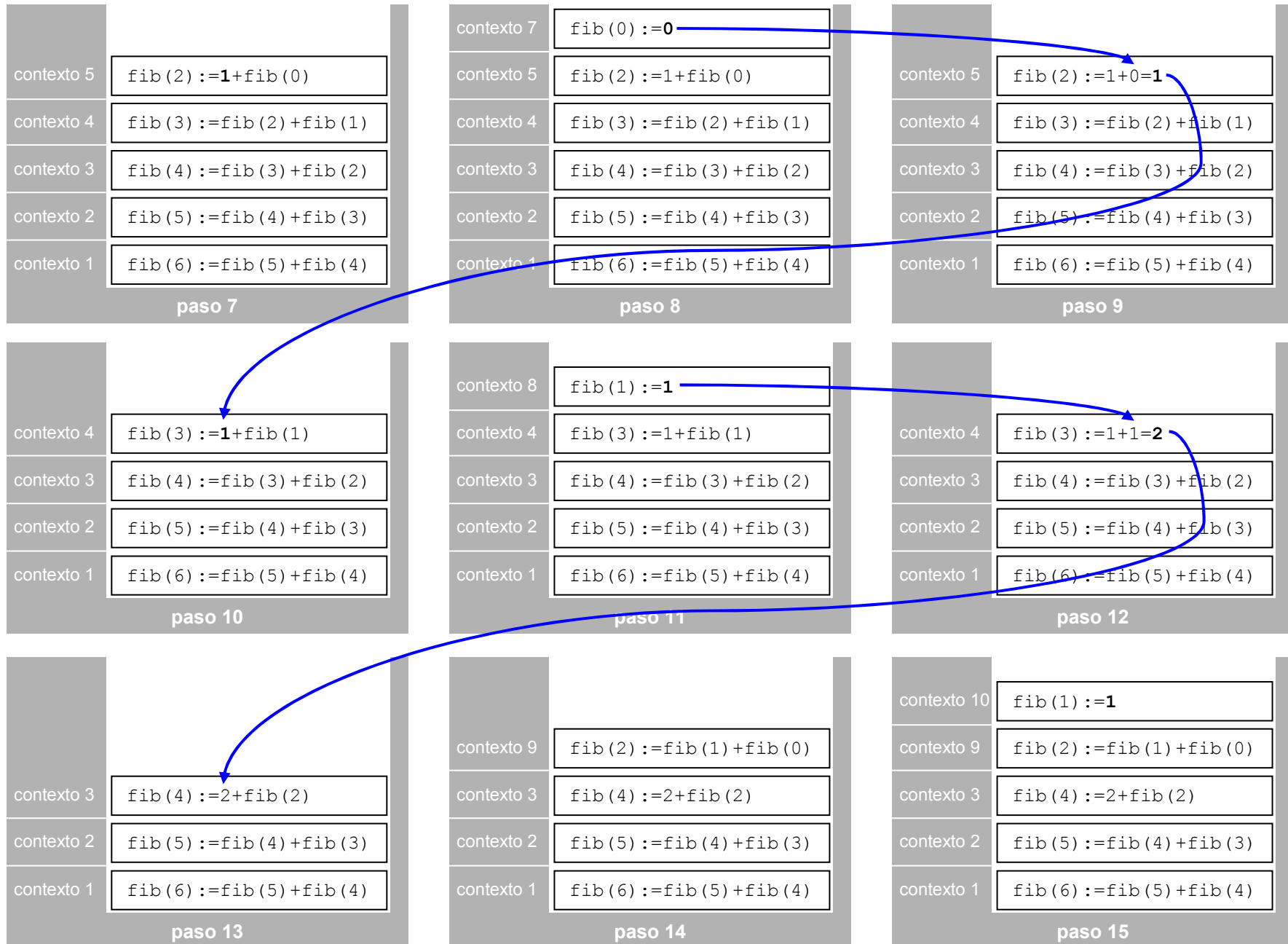
# recursividad vs iteración > recursividad ineficiente

Caso base	$\text{fib}(n)=n$ , si $n \leq 1$
Recursividad	$\text{fib}(n)=\text{fib}(n-1)+\text{fib}(n-2)$ , $\forall n > 1$

```
function fib( n: integer ): integer;
begin
  if n<=1 then
    fib:= n
  else
    fib:= fib(n-1) + fib(n-2);
  end;
```



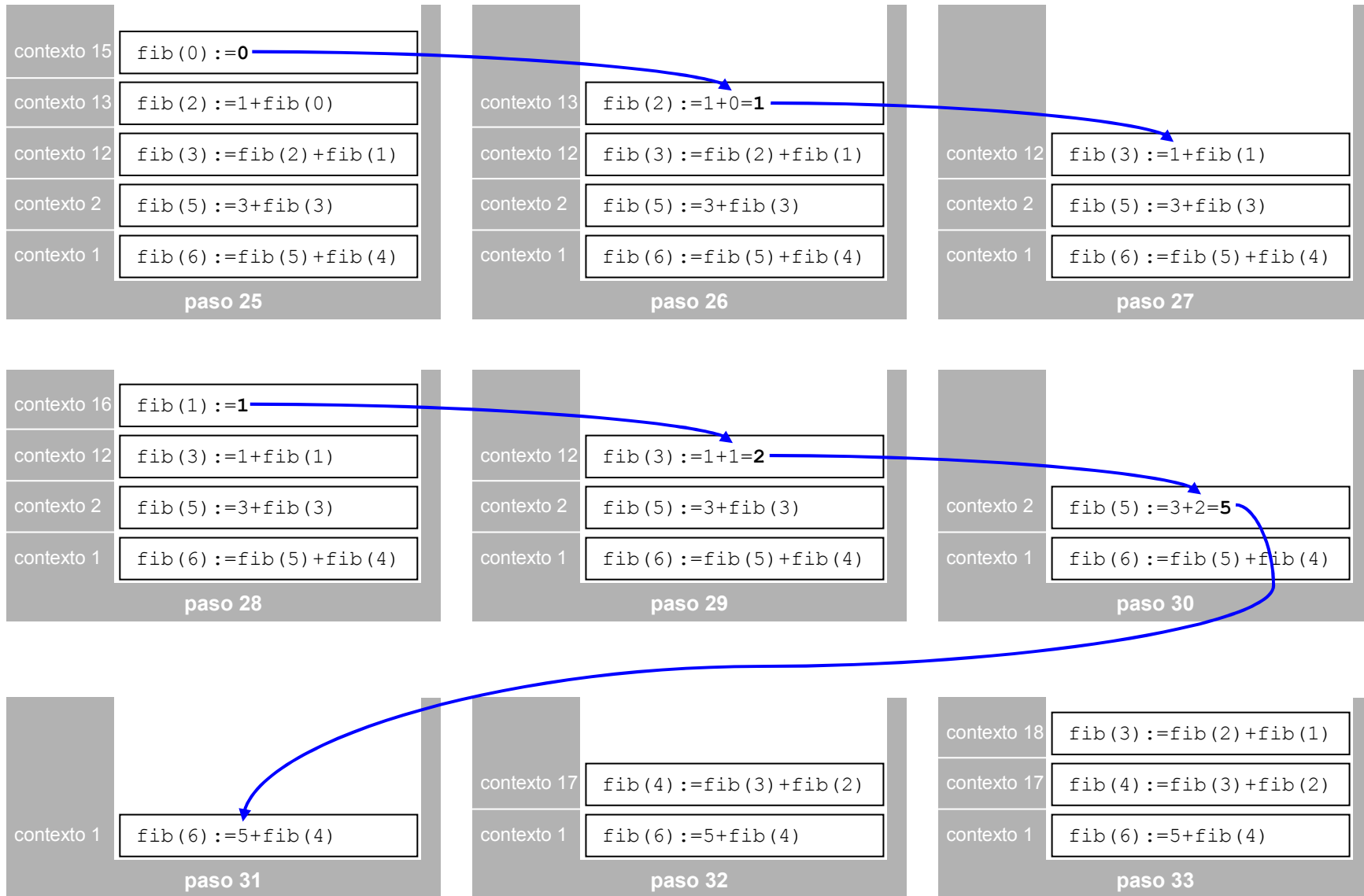
# recursividad vs iteración > recursividad ineficiente



# recursividad vs iteración > recursividad ineficiente

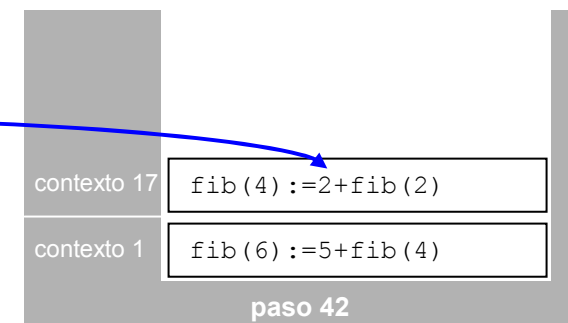
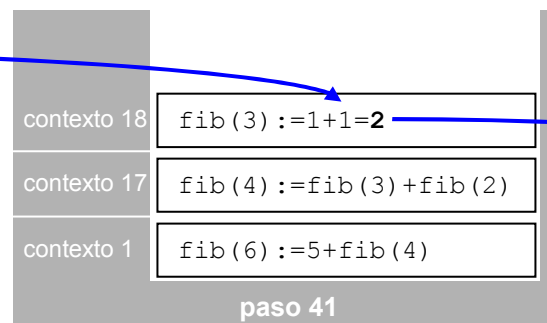
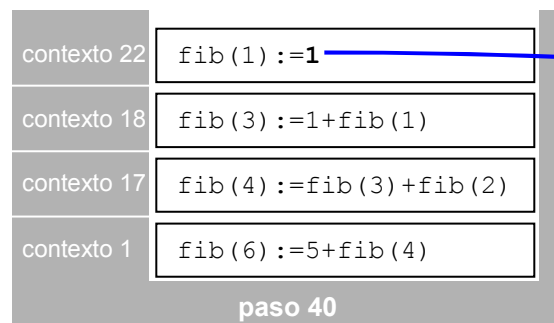
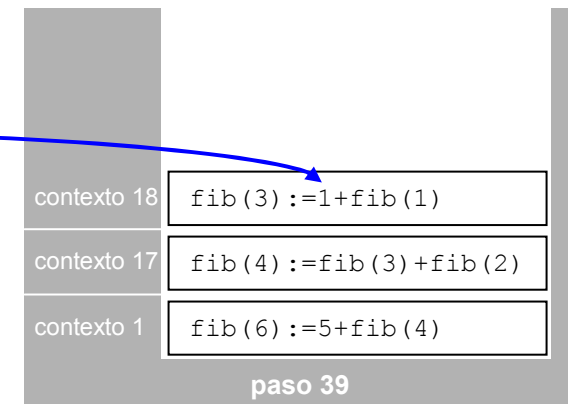
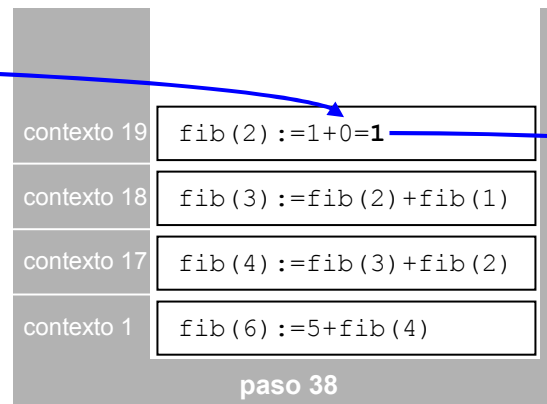
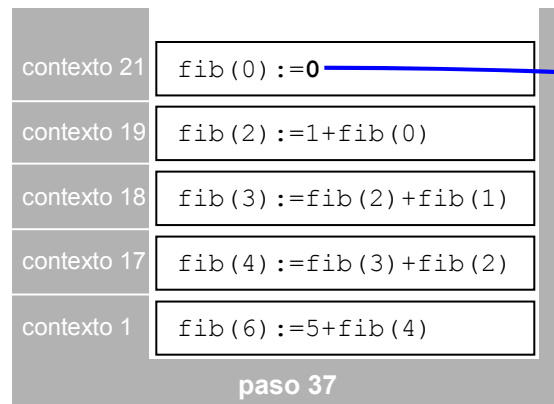
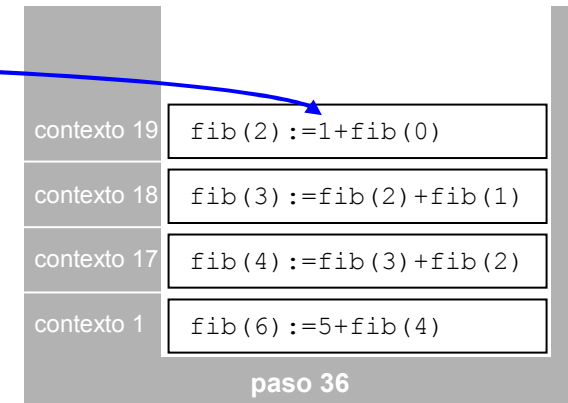
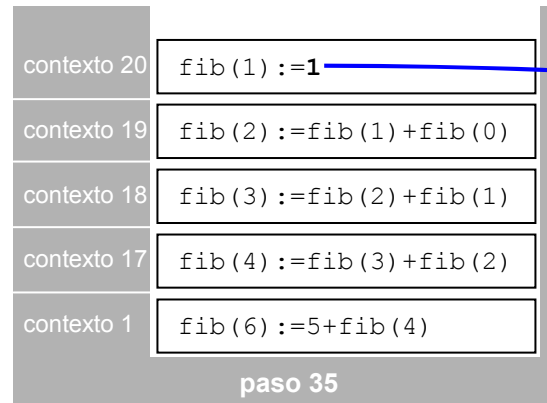
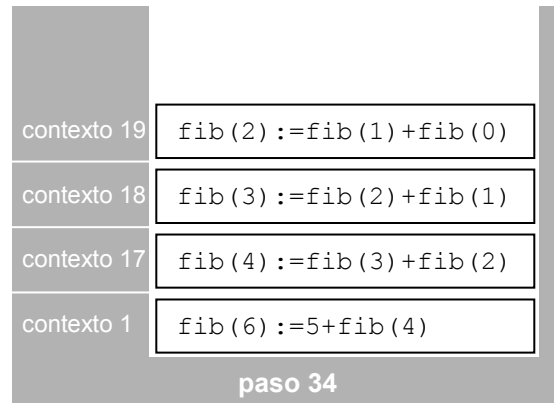


# recursividad vs iteración › recursividad ineficiente

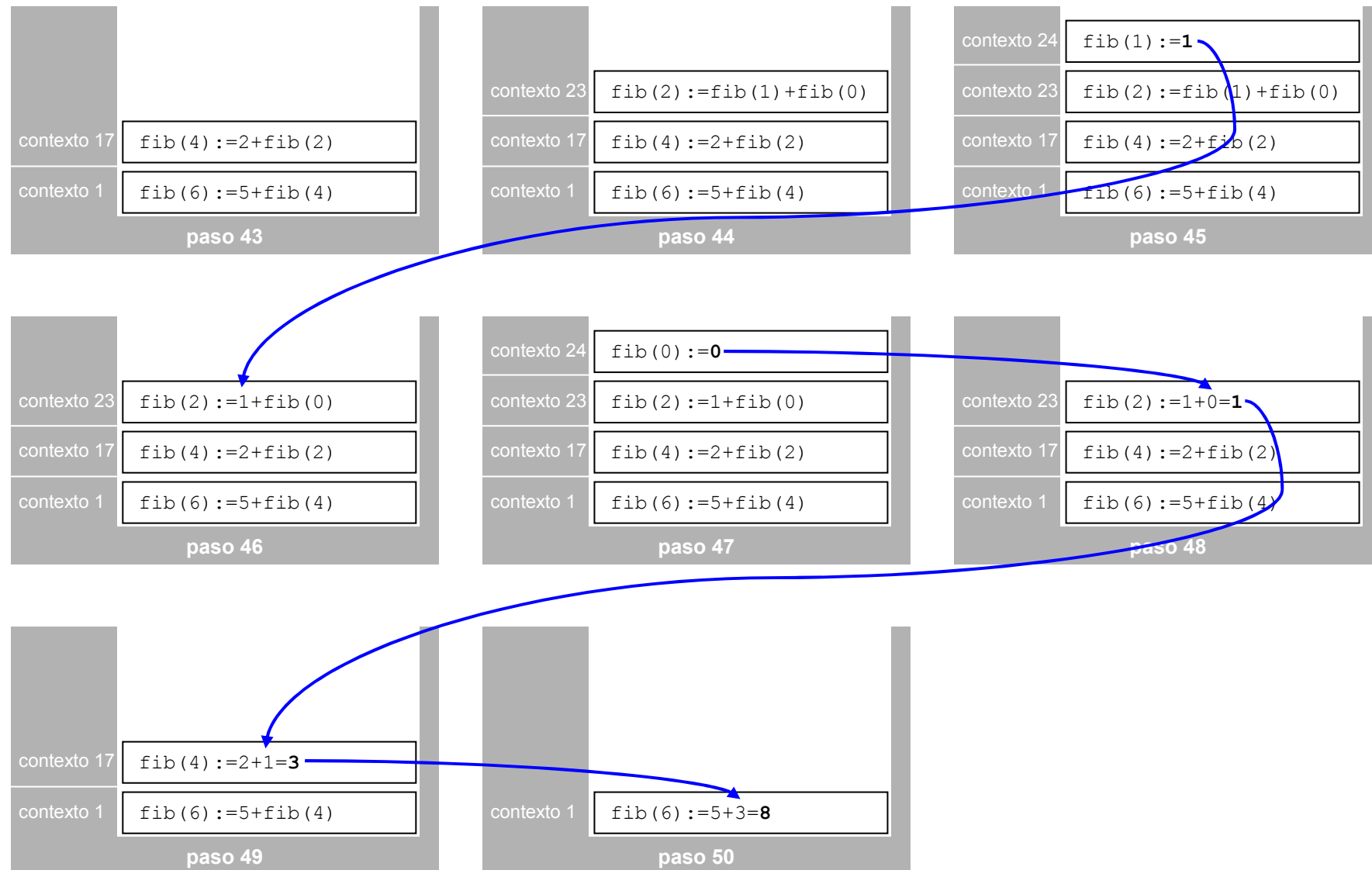




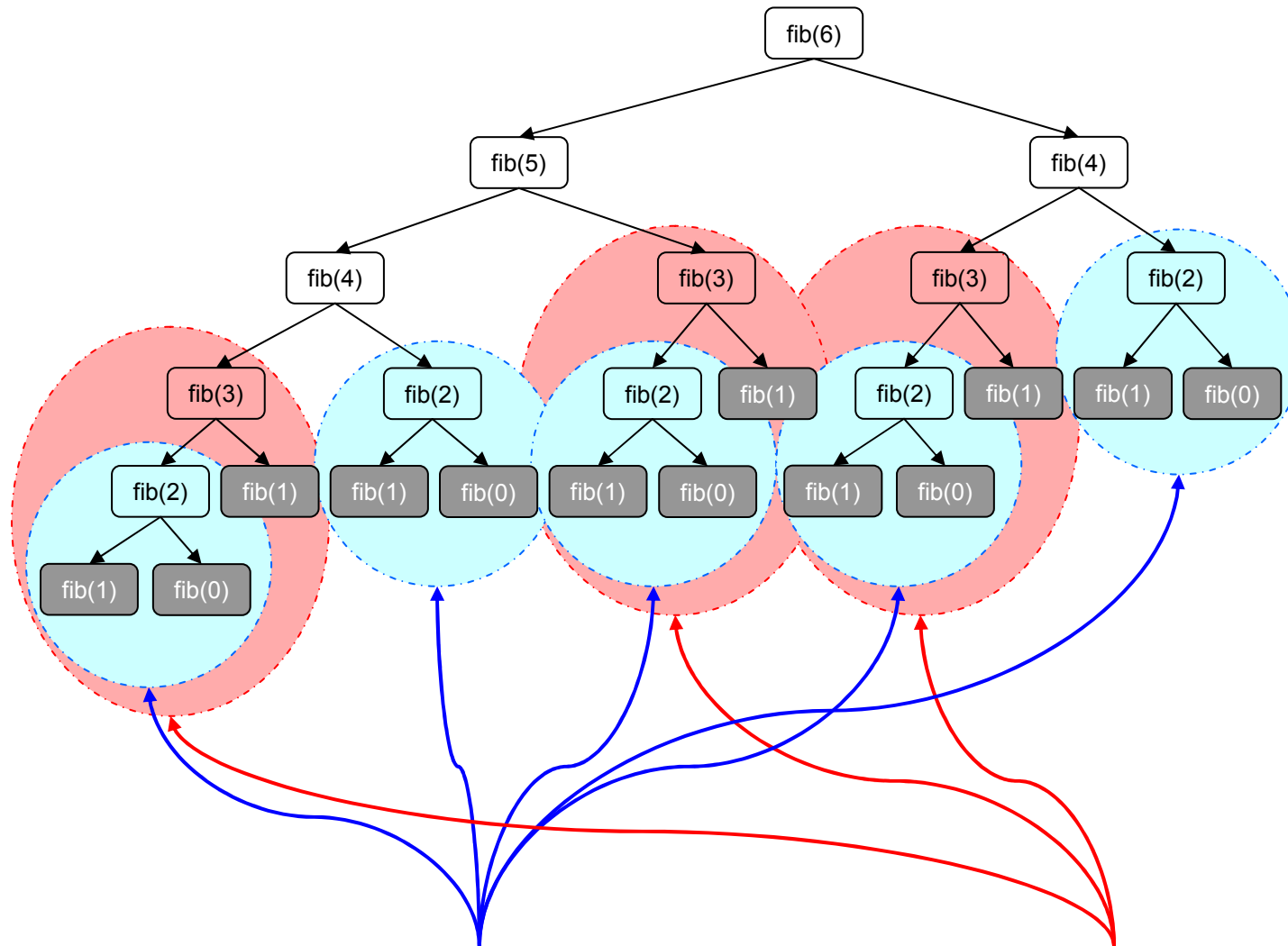
# recursividad vs iteración › recursividad ineficiente



# recursividad vs iteración › recursividad ineficiente



# recursividad vs iteración > recursividad ineficiente



Se realizan muchos cálculos innecesarios

## recursividad vs iteración › recursividad ineficiente

```
function fibonacci( n: integer ): integer;  
begin  
    if (n=0) or (n=1) then  
        fibonacci:= n  
    else  
        fibonacci:= fibonacci(n-1) + fibonacci(n-2);  
    end;  
end;
```

```
function fibonacciIterativo( n: integer ): integer;  
var  
    i,x,y,z: integer;  
begin  
    if (n=0) or (n=1) then  
        fibonacciIterativo:= n  
    else begin  
        x:=0; y:=1;  
        for i:=2 to n do begin  
            z:=x+y;  
            x:=y;  
            y:=z;  
        end;  
        fibonacciIterativo:=z;  
    end;  
end;
```

No es intuitivo pero es más eficiente

## ejercicio › potencia

Realizar una función recursiva que calcule la potencia entera de un número real. Debemos usar  $a^n = a^{n-1} * a$  siempre que  $n$  sea mayor que 0.

1. Determinar primero el caso base y el recursivo
2. Implementar la función en pascal
3. Hacer una implementación iterativa equivalente

Caso base	$a^0 = 1$
Recursividad	$a^n = a * a^{n-1}, \forall n > 0$

```
function potencia( base: real;  
    expon: integer ): real;  
begin  
    if expon=0 then  
        potencia:=1  
    else  
        potencia:= base *  
            potencia( base, expon-1);  
    end;
```

```
function potenciaIterativo( base: real;  
    expon: integer ): real;  
var  
    r: real;  
    i: integer;  
begin  
    r:=1;  
    for i:=1 to expon do  
        r:= r*base;  
    potenciaIterativo:=r;  
end;
```

## ejercicio › suma de elementos de un array

Realizar una función recursiva que calcule la suma de los elementos de un array

Caso base	$n=1 \Rightarrow v[n]$
Recursividad	$n>1 \Rightarrow v[n] + \text{sumar}(v, n-1)$

```
function sumar( v:lista; n:integer ): integer;  
begin  
  if n=1 then  
    sumar:=v[n]  
  else  
    sumar:=v[n]+sumar(v,n-1);  
end;
```

## ejercicio › invertir un número

Para invertir un número, básicamente se imprime la última cifra y se invierte el número sin la última cifra.

Caso base	$n < 10 \Rightarrow n$
Recursividad	$n \geq 10 \Rightarrow$ última cifra seguida de invertir el número sin esa cifra

```
procedure invertir( dec: integer );  
begin  
  if dec < 10 then  
    write( dec )  
  else begin  
    write( dec MOD 10 );  
    invertir( dec DIV 10 );  
  end  
end;
```

Convertir el procedure en una función que devuelva el número invertido

```
function invertido( dec: integer ): integer;  
var  
  s: string;  
begin  
  if dec < 10 then  
    invertido := dec  
  else begin  
    str( dec, s );  
    invertido := ( dec MOD 10 ) * potencia(10, length(s)-1) +  
                invertido( dec DIV 10 );  
  end;  
end;
```

## ejercicio › decimal a binario

Para convertir un número a binario se toma el resto de dividir entre 2 y a la izquierda se pone la conversión a binario del resto. Escribir una función que devuelva un string correspondiente al número recibido como parámetro en binario.

num	resto	cociente
11	1	5
5	1	2
2	0	1
1	1	
resultado 1011		

```
procedure dec2bin( n:integer );
begin
    if n<=1 then
        write( n );
    else begin
        dec2bin ( n div 2 );
        write( n mod 2 );
    end;
end;
```

```
function dec2bin( dec: integer ): string;
var
    resultado: string;
begin
    if dec<=1 then begin
        str( dec, resultado );
        dec2bin:= resultado
    end else begin
        str( dec MOD 2, resultado );
        dec2bin:= dec2bin( dec DIV 2 ) + resultado;
    end
end;
```



## ejercicio › máximo común divisor

El algoritmo de Euclides para obtener el máximo común divisor de dos números es:

```
function mcd( a, b: integer ): integer;
begin
  if a mod b = 0 then
    mcd:= b
  else
    mcd:= mcd( b, a mod b )
end;
```

Trazar el algoritmo para las llamadas:

mcd( 34, 25 )

mcd( 15, 21 )

a	b	resto
==	==	=====
34	25	9
25	9	7
9	7	2
7	2	1
2	1	0
mcd $\Rightarrow$ 1		

a	b	resto
==	==	=====
15	21	15
21	15	6
15	6	3
6	3	0
mcd $\Rightarrow$ 3		

## ejercicio › multiplicación rusa

Escribir una versión iterativa del algoritmo de multiplicación rusa

```
function rusa( a, b: integer ): integer;  
begin  
    if a = 0 then  
        rusa:= 0  
    else begin  
        if a mod 2 = 0 then  
            rusa:= rusa( a div 2, b*2 )  
        else  
            rusa:= b + rusa( a div 2, b*2 )  
        end  
    end  
end;
```

```
function rusaIterativa( a, b: integer ): integer;  
var  
    resultado: integer;  
begin  
    resultado:= 0;  
    while a>0 do begin  
        if a mod 2 <> 0 then  
            resultado:= resultado + b;  
            a:= a div 2;  
            b:= b * 2;  
        end;  
        rusaIterativa:= resultado;  
    end;  
end;
```

## ejercicio › búsqueda binaria

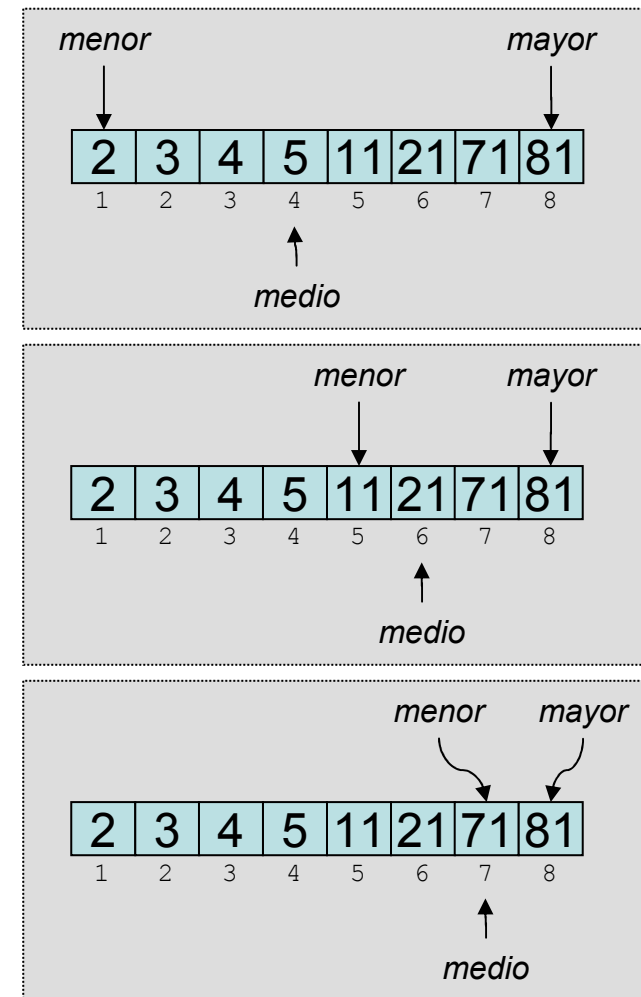
### Implementar de forma recursiva el algoritmo de búsqueda binaria

*busquedaBinaria( lista, 71, 1, 8, posición, encontrado)*

```

procedure busquedaBinaria( lista: tipoLista;
  elemento: tipoElemento; p,u: integer;
  var posicion: integer;
  var encontrado: boolean );
var
  mayor, menor, medio: integer;
begin
  menor:= p;
  mayor:= u;
  encontrado:= false;
  while (mayor>=menor) and
    not encontrado do begin
    medio:= (mayor+menor) div 2;
    if elemento=lista.elementos[medio] then
      encontrado:= true
    else if elemento<lista.elementos[medio] then
      mayor:= medio-1
    else
      menor:= medio+1
    end;
    if encontrado then
      posicion:= medio
    else
      posicion:= menor
    end;
  end;
end;

```



*posición*  $\Rightarrow$  7, *encontrado*  $\Rightarrow$  true

## ejercicio › búsqueda binaria (cont)

```
procedure busquedaBinRecursiva(  
  lista:tipoLista; elemento:tipoElemento;  
  menor,mayor:integer;  
  var posicion:integer; var encontrado:boolean );  
var  
  medio: integer;  
begin  
  medio:= (mayor+menor) div 2;  
  if elemento = lista.elementos[medio] then begin  
    encontrado:= true;  
    posicion:= medio  
  end else if mayor<menor then begin  
    encontrado:= false;  
    posicion:= menor  
  end else if elemento<lista.elementos[medio] then  
    busquedaBinRecursiva(lista,elemento,  
      menor,medio-1,posicion,encontrado)  
  else  
    busquedaBinRecursiva(lista,elemento,  
      medio+1,mayor,posicion,encontrado);  
end;
```

## ejercicio › multiplicación

1. Realizar una función recursiva que reciba dos números enteros y devuelva su producto. La función sólo podrá utilizar el operador '+'
2. Realizar la equivalente función iterativa
3. Trazar la función iterativa para las llamadas
  - i.  $a=0, b=3$
  - ii.  $a=3, b=0$

Caso base	$b=0 \Rightarrow 0$
Recursividad	$a + \text{multiplicacion}(a, b-1)$

```
function multiplicacion( a,b:integer ): integer;
begin
  if b=0 then
    multiplicacion:=0
  else
    multiplicacion:= a + multiplicacion( a, b-1 );
end;
```

```
a b
= =
0 3 +0
0 2 +0
0 1 +0
0 0
⇒ 0
```

```
a b
= =
3 0
⇒ 0
```

```
function multiplicacionIterativa( a,b:integer ): integer;
var
  r,i: integer;
begin
  r:=0;
  for i:=1 to b do
    r:= a + r;
  multiplicacionIterativa:=r;
end;
```

## ejercicio › orden en recursividad

¿Cuál sería la salida de los siguientes procedimientos recursivos de abajo si se invocan con el array “Liberia”, “Malawi”, “Etiopía”, “Burundi” y n=4?

```
procedure primeroWrite( v:tArrayString; n:tRango );
begin
    if n=1 then
        writeln( v[n] )
    else begin
        writeln( v[n] );
        primeroWrite( v, n-1 );
    end;
end;
```

Burundi  
Etiopía  
Malawi  
Liberia

Liberia  
Malawi  
Etiopía  
Burundi

```
procedure primeroRecursividad( v:tArrayString; n:tRango );
begin
    if n=1 then
        writeln( v[n] )
    else begin
        primeroRecursividad( v, n-1 );
        writeln( v[n] );
    end;
end;
```

## ejercicio › palíndromo

Un **palíndromo** es una palabra, número o frase que se lee igual hacia adelante que hacia atrás.

Implementar una función que dado un string devuelva si es palíndromo o no.

```
function palindromo( s:string; p,u:integer ): boolean;  
begin  
  if p>=u then  
    palindromo:= true  
  else if s[p]<>s[u] then  
    palindromo:= false  
  else  
    palindromo:= palindromo( s, p+1, u-1 );  
end;
```

## ejercicio › suma\_frac

---

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} + \frac{1}{n}$$

```
function sumaFrac( n:integer ): real;
var
  res : real;
begin
  if n = 0 then
    res := 1
  else
    res := sumaFrac(n-1) + 1/n;
  sumaFrac := res;
end;
```



## ejercicio › sumatorio

Sabiendo que, para  $\text{inf}, \text{sup} \in \mathbb{Z}$ , tales que  $\text{inf} \leq \text{sup}$ , se tiene

$$\sum_{i=\text{inf}}^{\text{sup}} a_i = \begin{cases} a_i, & \text{si } \text{inf} = \text{sup} \\ \sum_{i=\text{inf}}^{\text{med}} a_i + \sum_{i=\text{med}+1}^{\text{sup}} a_i & \text{si } \text{inf} < \text{sup} \end{cases}$$

(siendo  $\text{med} = (\text{inf} + \text{sup}) \text{ div } 2$ )

```
function sumatorio( a,inf,sup:integer ): real;
var
  m: integer;
begin
  if inf = sup then
    sumatorio:= a
  else begin
    m:= (inf+sup) div 2;
    sumatorio:= sumatorio(a, inf, m) + sumatorio(a, m+1, sup);
  end;
end;
```

## ejercicio › cifra i-esima

---

12983298

3 0

```
function cifraiesima( n,i:integer ): integer;
begin
  if i = 0 then
    cifraiesima:= n Mod 10
  else
    cifraiesima:= cifraiesima( n div 10, i-1 );
end;
```

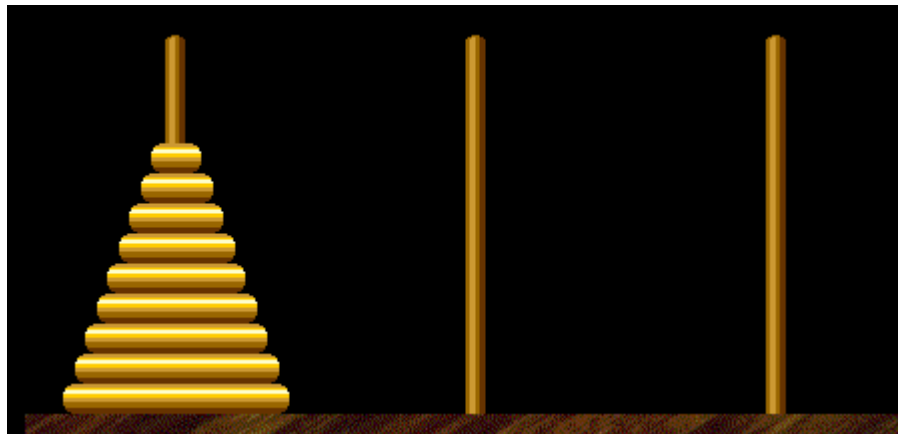
## ejemplo › torres de Hanoi

### La leyenda de las Torres de Hanoi

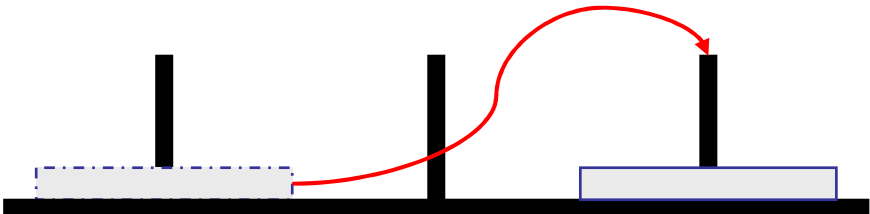
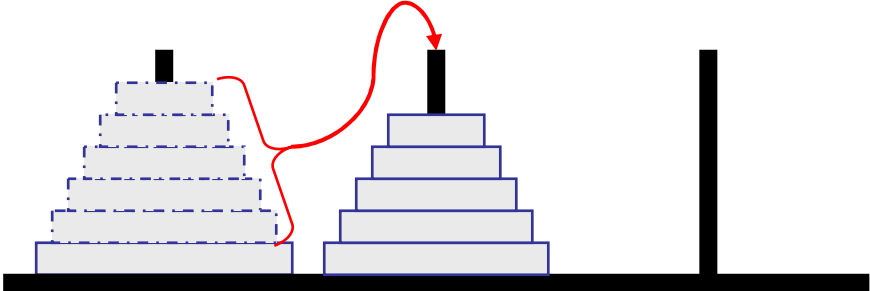
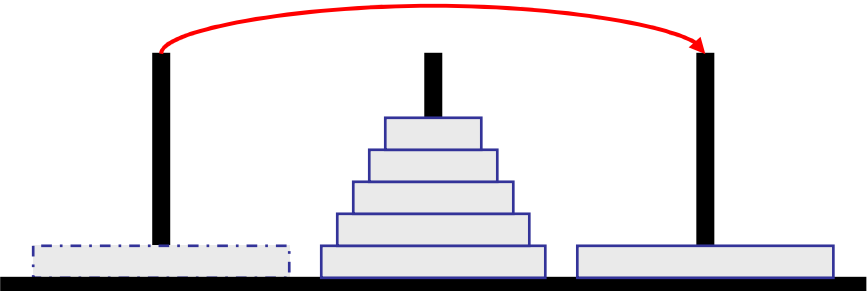
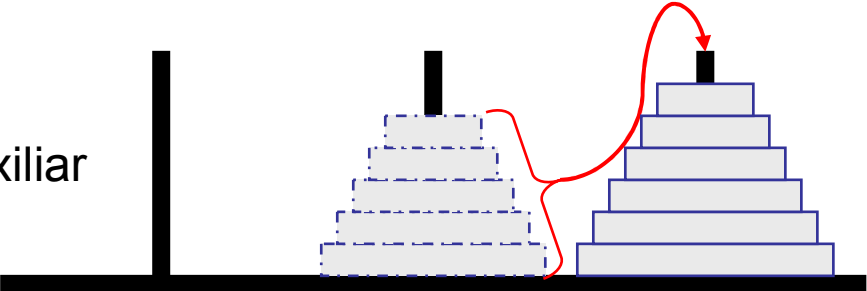
*Cuenta la leyenda que Dios al crear el mundo, colocó tres varillas de diamante con 64 discos en la primera. También creó un monasterio con monjes, los cuales tienen la tarea de resolver esta Torre de Hanoi divina. El día que estos monjes consigan terminar el juego, el mundo acabará. **"El sacerdote de turno no debe mover más de un disco a la vez, y no puede situar un disco de mayor diámetro encima de otro de menor diámetro"***

*El mínimo número de movimientos que se necesita para resolver este problema es de  $2^{64}-1$ . Si los monjes hicieran un movimiento por segundo, los 64 discos estarían en la tercera varilla en poco menos de 585 mil millones de años.*

<http://www.ciuvilanova.org/Jocs/Hanoi/index.htm>



# ejemplo › torres de Hanoi

Caso base	<p data-bbox="405 288 1048 336">Hay que mover un único disco</p> 
Recurividad	<p data-bbox="405 517 1061 564">Hay que mover <math>N</math> discos (<math>N &gt; 1</math>)</p> <p data-bbox="405 655 1043 815">1. <b>NUEVO PROBLEMA:</b> Mover <math>N-1</math> discos al palo auxiliar</p> 
	<p data-bbox="405 983 965 1086">2. Mover el último disco a su destino</p> 
	<p data-bbox="405 1246 1205 1406">3. <b>NUEVO PROBLEMA:</b> Mover los <math>N-1</math> discos del auxiliar a su destino</p> 

## ejemplo › torres de Hanoi

```

procedure hanoi( disco:tDiscos; posteOrigen,posteDestino,posteAuxiliar:tPoste );
begin
    if disco=1 then
        writeln( 'mover disco ', disco, ' desde ', posteOrigen, ' hasta ', posteDestino )
    else begin
        hanoi( disco-1, posteOrigen, posteAuxiliar, posteDestino );
        writeln( 'mover disco ', disco, ' desde ', posteOrigen, ' hasta ', posteDestino );
        hanoi( disco-1, posteAuxiliar, posteDestino, posteOrigen );
    end;
end;

```

```

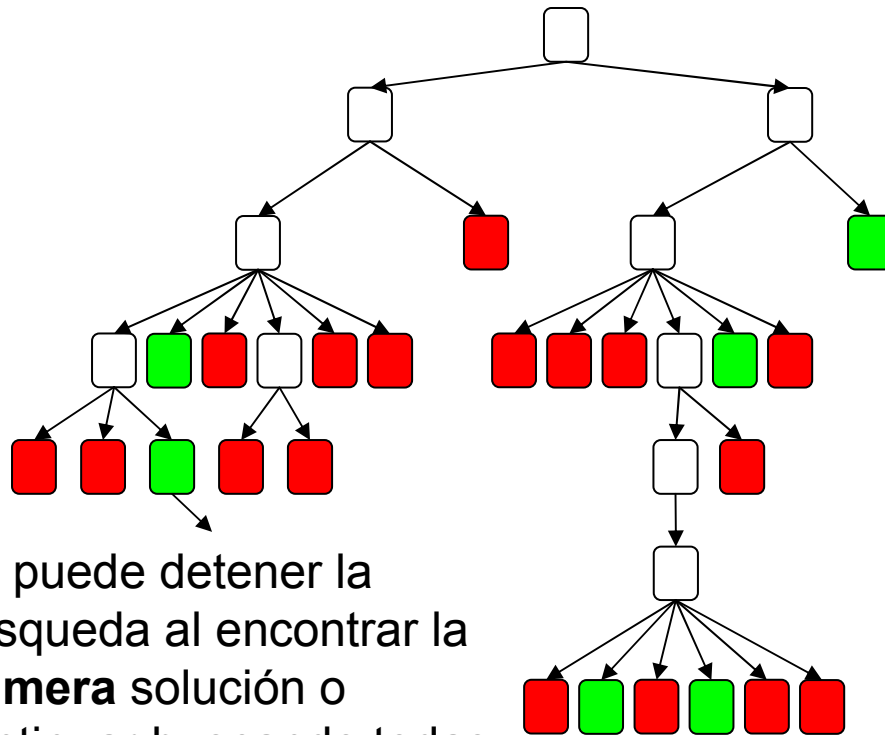
mover disco 1 desde 1 hasta 2
mover disco 2 desde 1 hasta 3
mover disco 1 desde 2 hasta 3
mover disco 3 desde 1 hasta 2
mover disco 1 desde 3 hasta 1
mover disco 2 desde 3 hasta 2
mover disco 1 desde 1 hasta 2
mover disco 4 desde 1 hasta 3
mover disco 1 desde 2 hasta 3
mover disco 2 desde 2 hasta 1
mover disco 1 desde 3 hasta 1
mover disco 3 desde 2 hasta 3
mover disco 1 desde 1 hasta 2
mover disco 2 desde 1 hasta 3
mover disco 1 desde 2 hasta 3

```



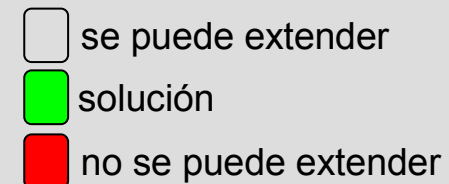
# backtracking

- Backtracking = rastreo inverso = vuelta atrás
- Fuerza bruta
- Método sistemático para probar todas las combinaciones
- El problema es cuando el árbol es demasiado grande
- <http://es.wikipedia.org/wiki/Backtracking>



Se puede detener la búsqueda al encontrar la **primera** solución o continuar buscando todas

- El árbol se explora en **profundidad**
- Es clave seleccionar bien los **candidatos** para extender cada nodo
- Se podría utilizar la **heurística** para decidir qué candidatos se extienden antes



# backtracking

Caso base	Se ha encontrado una solución <input checked="" type="checkbox"/> No se puede extender <input type="checkbox"/>
Recursividad	Determinar formas de extender el árbol <input type="checkbox"/> Llamada recursiva para cada forma de extender el árbol

```

procedure backtrack( solucionParcial );
var
    candidatos;
begin
    if esUnaSolucion( solucionParcial ) then
        procesarSolucion( solucionParcial )
    else
        construirCandidatos( solucionParcial,
                             candidatos );
    para todos los candidatos
        extender( solucionParcial, candidato,
                  solucionExtendida );
        backtrack( solucionExtendida );
end;
    
```

## backtracking › paso por referencia del estado

**Problema:** al pasar solucionParcial por valor se crea una copia y se desbordará la pila de contextos si el árbol es grande (se realizan muchas llamadas recursivas)

**Solución:** pasar solucionParcial por referencia e implementar una función volver que recupere el estado previo

```
procedure backtrack( var solucionParcial );
var
    candidatos;
begin
    if esUnaSolucion( solucionParcial ) then
        procesarSolucion( solucionParcial )
    else
        construirCandidatos(solucionParcial,candidatos );
        para todos los candidatos
            extender( solucionParcial, candidato );
            backtrack( solucionParcial );
            volver( solucionParcial );
end;
```

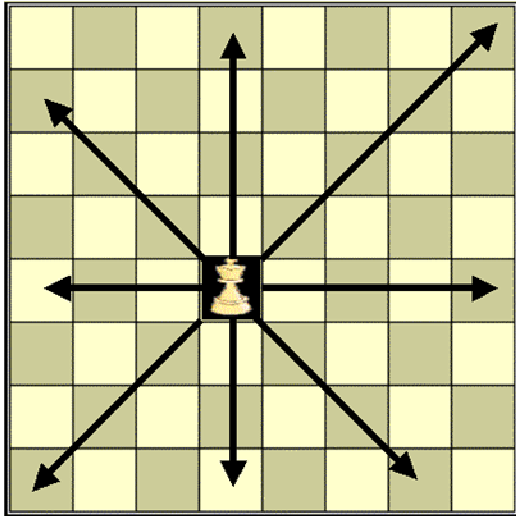


## backtracking › implementación alternativa

```
procedure backtrack2( solucionParcial );  
var candidatos;  
begin  
    construirCandidatos( solucionParcial, candidatos );  
    para todos los candidatos  
        extender( solucionParcial, candidato );  
        if esUnaSolucion( solucionParcial ) then  
            procesarSolucion( solucionParcial )  
        else if sePuedeExtender( solucionParcial ) then  
            backtrack2( solucionParcial );  
        volver( solucionParcial );  
end;
```

## backtracking › problema de las 8 reinas

### Problema de las 8 reinas



Colocar ocho reinas en un tablero de ajedrez, de forma que ninguna reina pueda amenazar a otra

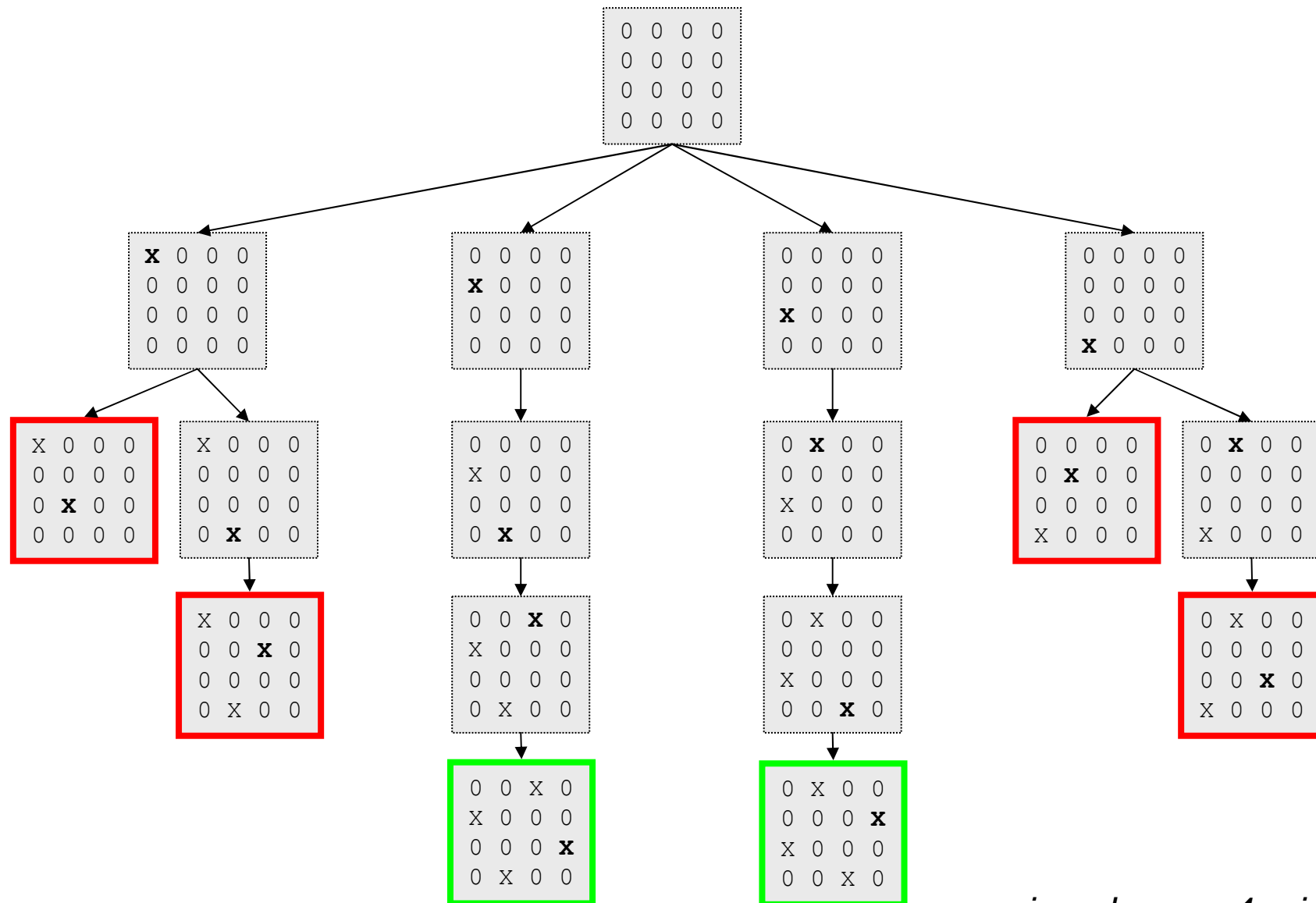
### Recursividad

En cada paso se intenta colocar una reina en la siguiente fila

### Construir candidatos

En cada llamada recursiva se buscarán todas las posiciones no amenazadas de la siguiente fila

# backtracking › problema de las 8 reinas



*ejemplo para 4 reinas*

# backtracking › problema de las 8 reinas

```

type
  TTablero = array [1..N,1..N] of integer;
  TPosicion = record
    x: 1..N;
    y: 1..N;
  end;
  TPosiciones = array[1..N] of TPosicion;

procedure inicializarTablero( var tablero:TTablero );
var
  i,j: integer;
begin
  for i:= 1 to N do
    for j:= 1 to N do
      tablero[i,j]:= 0;
    end;
  end;

function esUnaSolucion( fila:integer ): boolean;
begin
  esUnaSolucion:= fila > N;
end;

procedure extender( var tablero:TTablero;candidato:TPosicion);
begin
  tablero[candidato.x,candidato.y] := 1;
end;

procedure volver( var tablero:TTablero; posicion:TPosicion );
begin
  tablero[posicion.x,posicion.y]:=0;
end;

```

Representación del estado (nodos o soluciones intermedias)

¿Tenemos una solución?

Extender un nodo con un nuevo candidato

Volver atrás eliminando un candidato

## backtracking › problema de las 8 reinas

---

```
procedure construirCandidatos( tablero:TTablero; fila:integer;
    var candidatos:TPosiciones );
var
    i,j: integer;
    nuevaPosicion: TPosicion;
begin
    nuevaPosicion.x:=0; nuevaPosicion.y:=0;
    for i:=1 to 8 do
        candidatos[i]:= nuevaPosicion;

        j:=1;
        for i:=1 to N do begin
            nuevaPosicion.x:= i;
            nuevaPosicion.y:= fila;
            if not amenazada( tablero, nuevaPosicion ) then begin
                candidatos[j]:= nuevaPosicion;
                j:=j+1;
            end;
        end;
    end;
end;
```

# backtracking › problema de las 8 reinas

```
function amenazada( tablero:TTablero; posicion:TPosicion ): boolean;
var
    nuevaPosicion: TPosicion;
    otraReina: boolean;
begin
    otraReina:= false;

    nuevaPosicion.x:= 1;
    while (nuevaPosicion.x <= N) and not otraReina do begin
        if tablero[nuevaPosicion.x,posicion.y] <> 0 then
            otraReina:= true;
            inc( nuevaPosicion.x );
        end;

    nuevaPosicion.y:= 1;
    while (nuevaPosicion.y <= N) and not otraReina do begin
        if tablero[posicion.x,nuevaPosicion.y] <> 0 then
            otraReina:= true;
            inc( nuevaPosicion.y );
        end;

    nuevaPosicion.x:= posicion.x-1;
    nuevaPosicion.y:= posicion.y-1;
    while (nuevaPosicion.x>0) and (nuevaPosicion.y>0) and not otraReina do begin
        if tablero[nuevaPosicion.x,nuevaPosicion.y] <> 0 then
            otraReina:= true;
            dec( nuevaPosicion.x );
            dec( nuevaPosicion.y );
        end;

    ... igual para las otras 3 diagonales

    amenazada:= otraReina;
end;
```

¿reina en la misma fila?

¿reina en la misma columna?

¿reina en la diagonal hacia arriba y hacia la izquierda?  
Igual para las otras 3 direcciones de diagonales

## backtracking › problema de las 8 reinas


```
procedure backtrack (  
    var tablero:TTablero;  
    fila:integer;  
    unaSolucionSolo: boolean;  
    var noSoluciones: integer;  
    var movimientos: integer );  
var  
    i:integer;  
    candidatos:TPosiciones;  
begin  
    movimientos:= movimientos+1;  
    if esUnaSolucion( fila ) then begin  
        procesarSolucion( tablero, movimientos );  
        noSoluciones:= noSoluciones+1;  
    end else begin  
        construirCandidatos( tablero, fila, candidatos );  
        i:=1;  
        while seguirBuscando( candidatos[i], unaSolucionSolo, noSoluciones ) do begin  
            extender( tablero, candidatos[i] );  
            backtrack( tablero, fila+1, unaSolucionSolo, noSoluciones, movimientos);  
            volver( tablero, candidatos[i] );  
            inc(i);  
        end;  
    end;  
end;  
end;
```

# backtracking › problema de las 8 reinas

<b>X</b>	0	0	0	0	0	0	0
0	0	0	0	0	0	<b>X</b>	0
0	0	0	0	<b>X</b>	0	0	0
0	0	0	0	0	0	0	<b>X</b>
0	<b>X</b>	0	0	0	0	0	0
0	0	0	<b>X</b>	0	0	0	0
0	0	0	0	0	<b>X</b>	0	0
0	0	<b>X</b>	0	0	0	0	0



# backtracking › problema del salto del caballo

	27	6	17	2	25	8	23
5	18	1	26	7	22	11	14
28	63	20	3	16	13	24	9
19	4	29	62	21	10	15	12
44	47	42	53	30	61	36	51
41	56	45	48	35	52	31	60
46	43	54	39	58	33	50	37
55	40	57	34	49	38	59	32

## Problema del salto del caballo

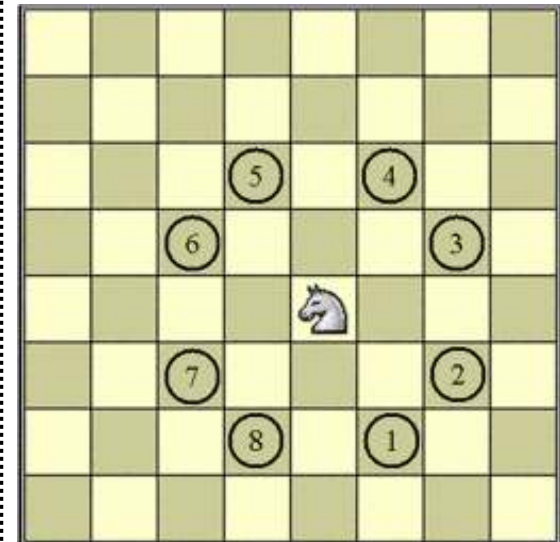
Recorrer todas las casillas de la ajedrez con un caballo y sin pasar dos veces por la misma

## Recursividad

En cada paso se intenta mover el caballo a una casilla no visitada

## Construir candidatos

En cada llamada recursiva se buscarán posibles saltos del caballo desde su posición actual



# backtracking › problema del salto del caballo

```

type
  TTablero = array [1..N,1..N] of integer;
  TDesplaz = array [1..2,1..8] of integer;
  TPosicion = record
    x: 1..N;
    y: 1..N;
  end;
  TPosiciones = array[1..8] of TPosicion;

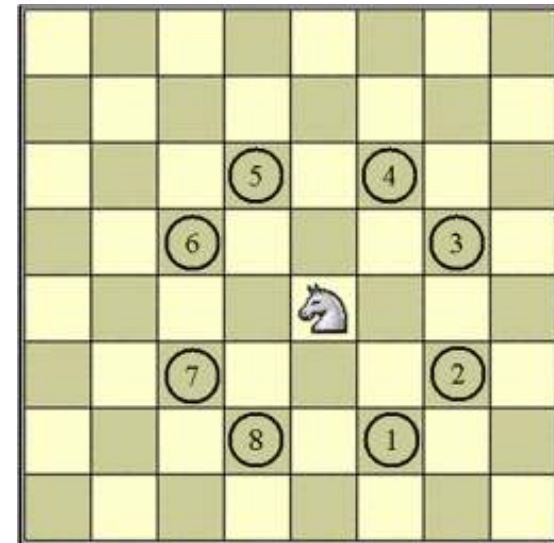
procedure inicializarTablero( var tablero:TTablero );
var
  i,j: integer;
begin
  for i:= 1 to N do
    for j:= 1 to N do
      tablero[i,j]:= 0;
    end;
  end;

procedure inicializarMatrizDespl( var D:TDesplaz );
begin
  D[1,1]:= 2; D[2,1]:= 1;
  D[1,2]:= 1; D[2,2]:= 2;
  D[1,3]:= -1; D[2,3]:= 2;
  D[1,4]:= -2; D[2,4]:= 1;
  D[1,5]:= -2; D[2,5]:= -1;
  D[1,6]:= -1; D[2,6]:= -2;
  D[1,7]:= 1; D[2,7]:= -2;
  D[1,8]:= 2; D[2,8]:= -1;
end;

```

Cada casilla tendrá un 0 si no ha sido visitada o el número del salto si ha sido visitada

Se mantiene en un array del tipo Tdesplaz los 8 saltos del caballo



## backtracking › problema del salto del caballo

```
procedure construirCandidatos( var tablero:TTablero;
    var movimientosCaballo:TDesplaz;
    posicionActual:TPosicion; var candidatos:TPosiciones );
var
    i,j: integer;
    nuevaPosicion: TPosicion;
begin
    nuevaPosicion.x:=0; nuevaPosicion.y:=0;
    for i:=1 to 8 do candidatos[i]:= nuevaPosicion;

    j:=1;
    for i:=1 to 8 do begin
        nuevaPosicion.x:= posicionActual.x + movimientosCaballo[1,i];
        nuevaPosicion.y:= posicionActual.y + movimientosCaballo[2,i];
        if esUnaPosicionValida( nuevaPosicion ) and
            not posicionVisitada( nuevaPosicion, tablero ) then begin
            candidatos[j]:= nuevaPosicion;
            j:=j+1;
        end;
    end;
end;

function esUnaPosicionValida( posicion:TPosicion ): boolean;
begin
    esUnaPosicionValida:= (posicion.x in [1..N]) and (posicion.y in [1..N]);
end;

function posicionVisitada( posicion:TPosicion; var tablero:TTablero ): boolean;
begin
    posicionVisitada:= tablero[posicion.x,posicion.y] <> 0;
end;
```

## backtracking › problema del salto del caballo

```
procedure backtrack (  
    salto:integer;  
    posicionActual: TPosicion;  
    var tablero:TTablero;  
    movimientosCaballo:TDesplaz;  
    unaSolucionSolo: boolean;  
    var noSoluciones: integer;  
    var movimientos: integer );  
var  
    i:integer;  
    candidatos:TPosiciones;  
begin  
    movimientos:= movimientos+1;  
    if esUnaSolucion( salto ) then begin  
        procesarSolucion( tablero, movimientos );  
        noSoluciones:= noSoluciones+1;  
    end else begin  
        construirCandidatos(tablero,movimientosCaballo,posicionActual,candidatos);  
        i:=1;  
        while seguirBuscando(candidatos[i],unaSolucionSolo,noSoluciones) do begin  
            extender( tablero, salto, candidatos[i] );  
            backtrack(salto+1,candidatos[i],tablero,movimientosCaballo,  
                unaSolucionSolo,noSoluciones,movimientos);  
            volver( tablero, candidatos[i] );  
            inc(i);  
        end;  
    end;  
end;  
end;
```

## backtracking › problema del salto del caballo

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

¡ más de 8 millones de movimientos para encontrar una solución !