UNIVERSIDAD DE MÁLAGA

TEMA 3

Abstracción Procedimental
Contenido del Tema

UNIVERSIDAD DE MÁLAGA

TEMA 3

Abstracción Procedimental
Contenido del Tema

3.2.5. Interfaz

3.2.6. Criterios de modularización

3.2.7. Variables locales y globales. Efectos laterales

3.2.8. Precondiciones y Postcondiciones. Tratamiento de situaciones excepcionales

3.3. Recursividad

3.3.1. Concepto de Recursividad

3.3.2. Ejemplos

3.3.3. Recursividad frente a Iteración

Diseño Descendente

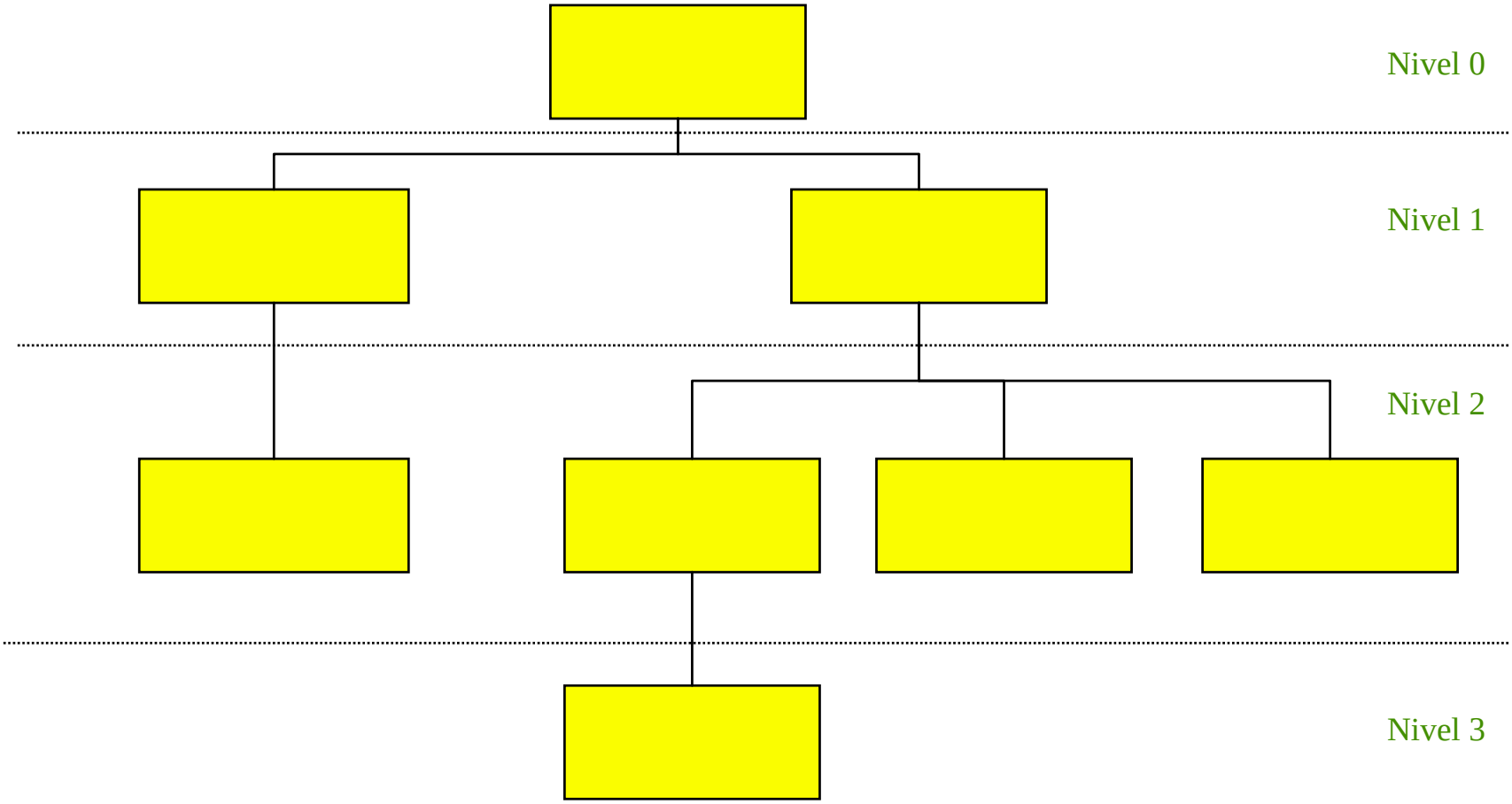
- En la mayoría de los problemas reales, el algoritmo que los soluciona es demasiado largo y complejo para implementarlo mediante un único texto (programa).

- Desventajas de este tipo de programas:
  - Rigidez e inflexibilidad de los programas.
  - Pérdida excesiva de tiempo en corrección de errores.
  - Imposibilidad de reutilizar el programa o fragmentos suyos en proyectos futuros.

Diseño Descendente

- Se hace necesaria la utilización de alguna metodología de diseño que evite estos inconvenientes
- La metodología de Diseño Descendente (“refinamientos sucesivos”, “Top-Down”, “divide y vencerás”) se ha mostrado como la más adecuada.
- Un **problema** se descompone en varios subproblemas y estos a su vez se descomponen en otros subproblemas hasta llegar a un nivel de **descomposición** que permita la solución sencilla de los diferentes **subproblemas**.
- La **solución** al problema inicial viene dada por la **composición** de cada una de las **soluciones** a los subproblemas

Diseño Descendente



Diseño Descendente

- Diseño descendente."Refinamientos sucesivos" o "programación modular". “Divide y vencerás”.
  - Fases:
    1. Estudio del problema.
    2. Descomposición en subproblemas más sencillos y lo más independientes posible.
    3. Repetir el paso 2 para cada subproblema hasta que la solución de los mismos sea trivial.
    4. Implementar las soluciones.
    5. Reconstruir la solución global desandando el camino.

Diseño Descendente

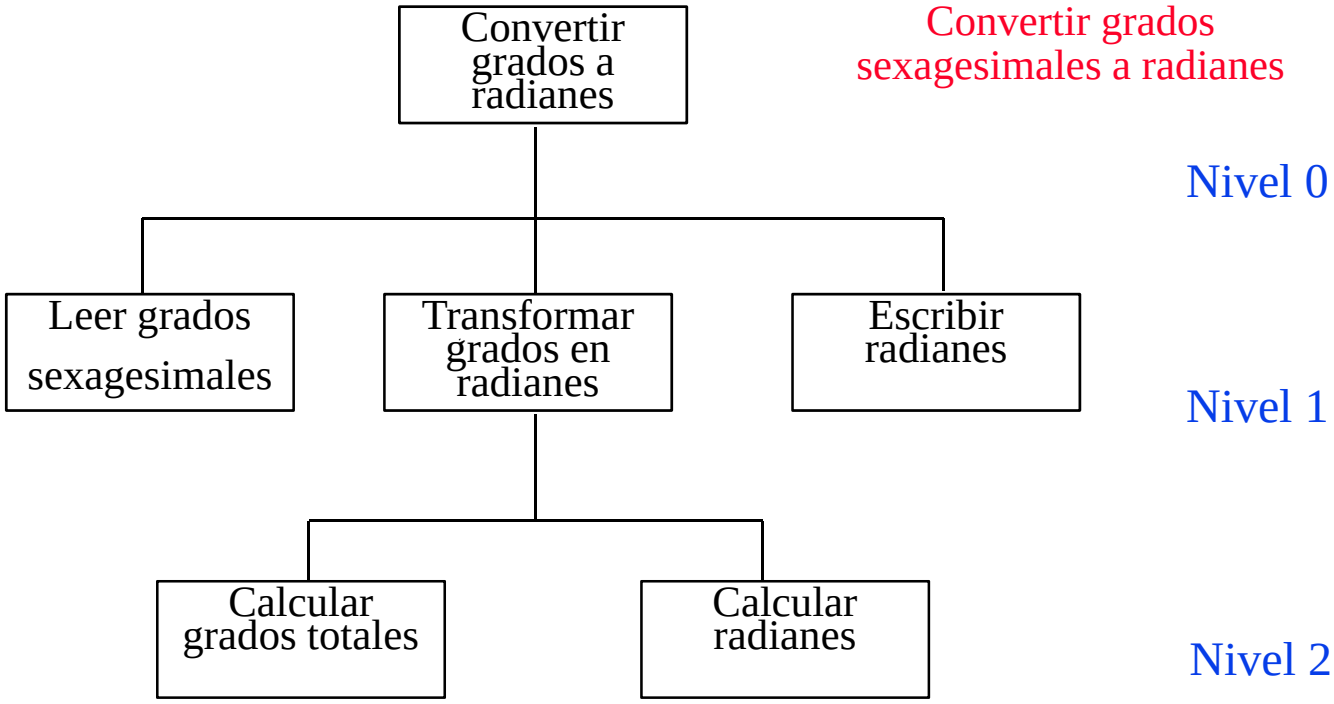
- Características:
  - Diseño de lo general (abstracto) a lo particular.
  - Las decisiones complejas se posponen.
  - Estructura jerárquica de niveles. En cada nivel se encuentra el problema (o solución) completo.
  - En el nivel superior se encuentra la definición del problema y el nivel inferior la solución completa.

Diseño Descendente

- ¿Cómo hacemos la división en módulos?
  - Funcionalidad clara y bien definida
  - Aislados. Sin dependencias con otros módulos
  - Entrada/Salida simple y bien definida
- Abstracción:
  - Nos centramos en la funcionalidad de los módulos
    - Qué problema resuelve
    - Qué datos necesita
    - Qué datos produce

Diseño Descendente

Ejemplo: Convertir grados sexagesimales a radianes



Diseño Descendente

- Módulo: Convertir grados sexagesimales a radianes
  - Funcionalidad: Convertir grados sexagesimales a radianes
  - Datos de Entrada: grados, minutos, segundos de **teclado**
  - Datos de Salida: radianes en **pantalla**

Diseño Descendente

- Módulo: Leer grados sexagesimales
  - Funcionalidad: Lee grados, minutos y segundos
  - Datos de Entrada: grados, minutos y segundos de **teclado**
  - Datos de Salida: grados, minutos y segundos

```
cin>>grados>>minutos>>segundos;
```

Diseño Descendente

- Módulo: Transformar grados en radianes
  - Funcionalidad: Transformar grados, minutos y segundos en radianes
  - Datos de Entrada: grados, minutos y segundos
  - Datos de Salida: radianes equivalentes

```
Calcular_grados_totales
Calcular_radianes
```

Diseño Descendente

- Módulo: Escribir radianes
  - Funcionalidad: escribe radianes equivalentes
  - Datos de Entrada: radianes equivalentes
  - Datos de Salida: radianes equivalentes por **pantalla**

```
cout<<radianes;
```

Diseño Descendente

- Módulo: **Calcular_grados_totales**
  - Funcionalidad: calcula grados, de grados, minutos y segundos
  - Datos de Entrada: grados, minutos y segundos
  - Datos de Salida: grados totales

```
Grados_to=grados+(minutos/60)+(segundos/3600)
```

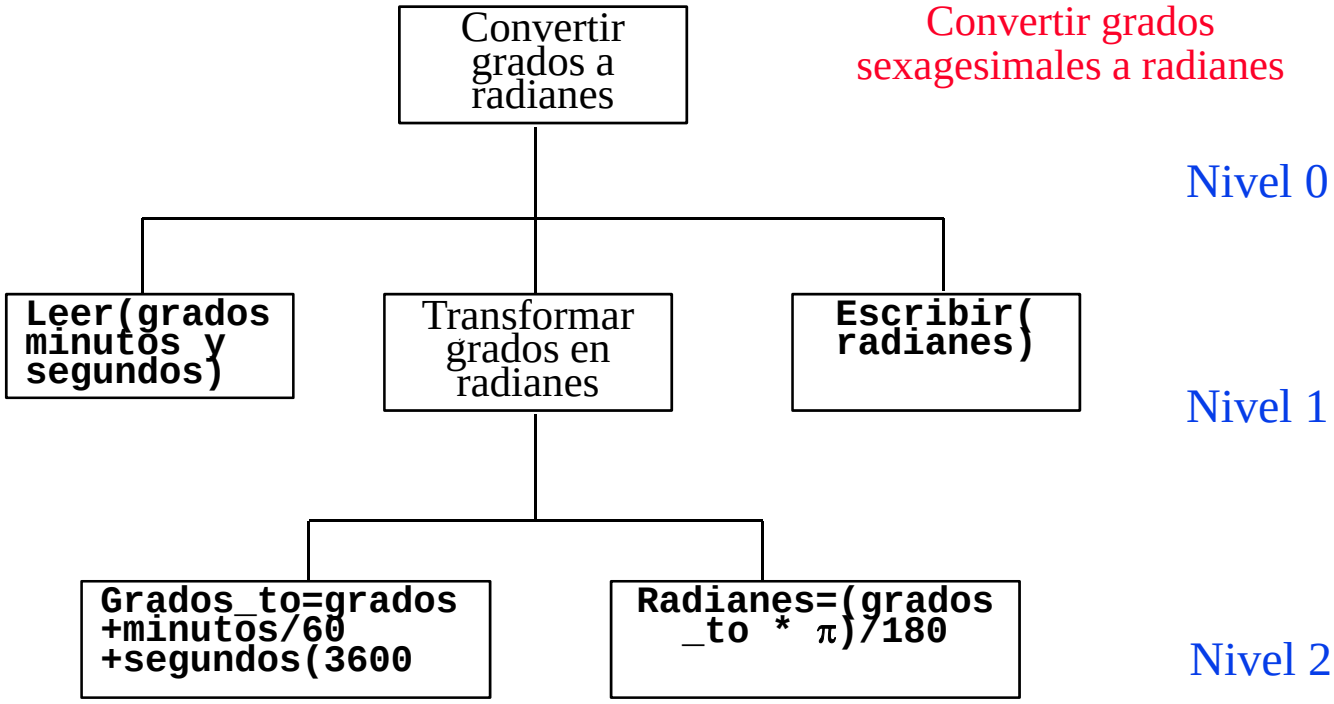
Diseño Descendente

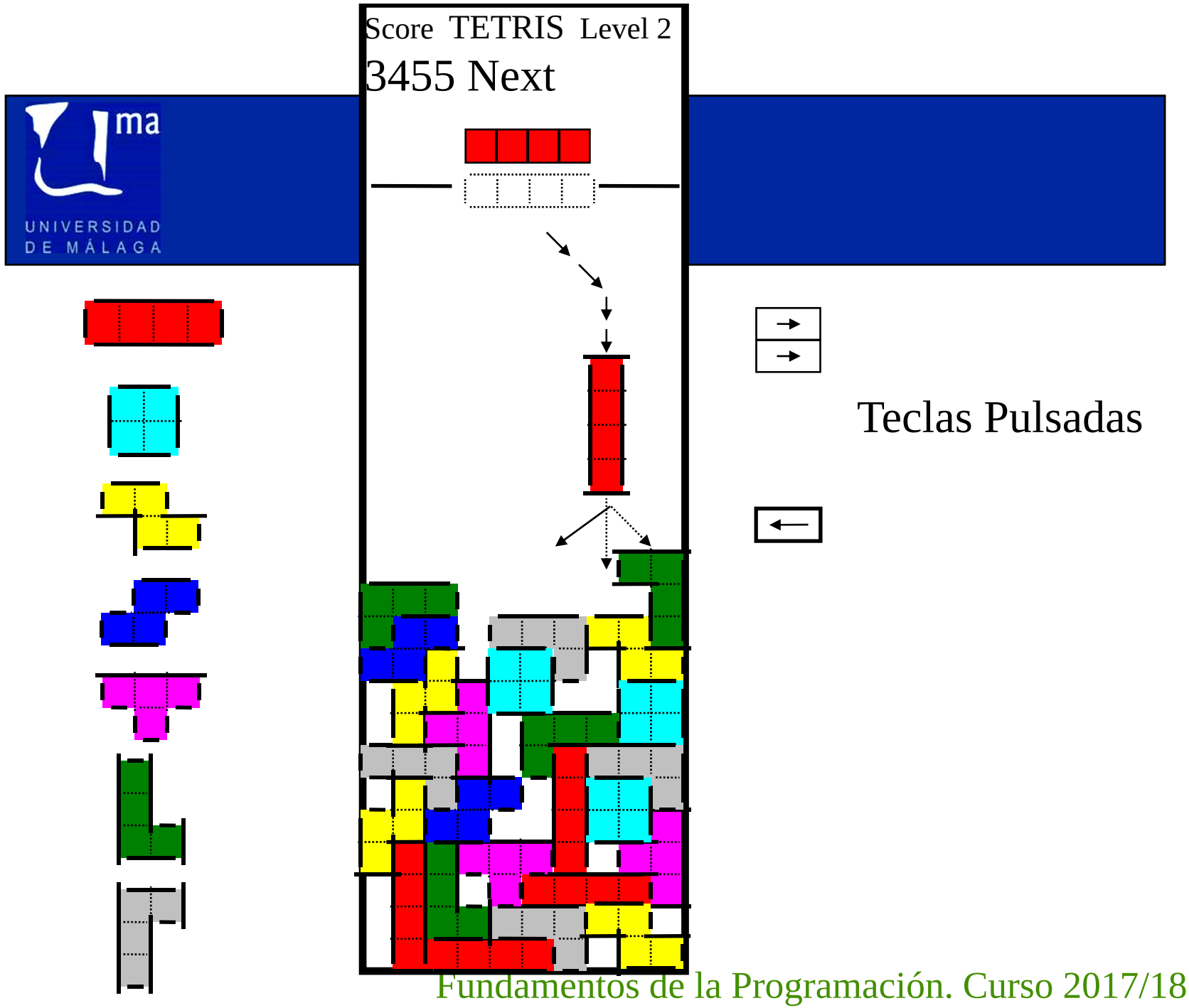
- Módulo: **Calcular_radianes**
  - Funcionalidad: calcula radianes a partir de grados totales
  - Datos de Entrada: grados totales
  - Datos de Salida: radianes equivalentes

```
radianes=(grados_to * π)/180
```

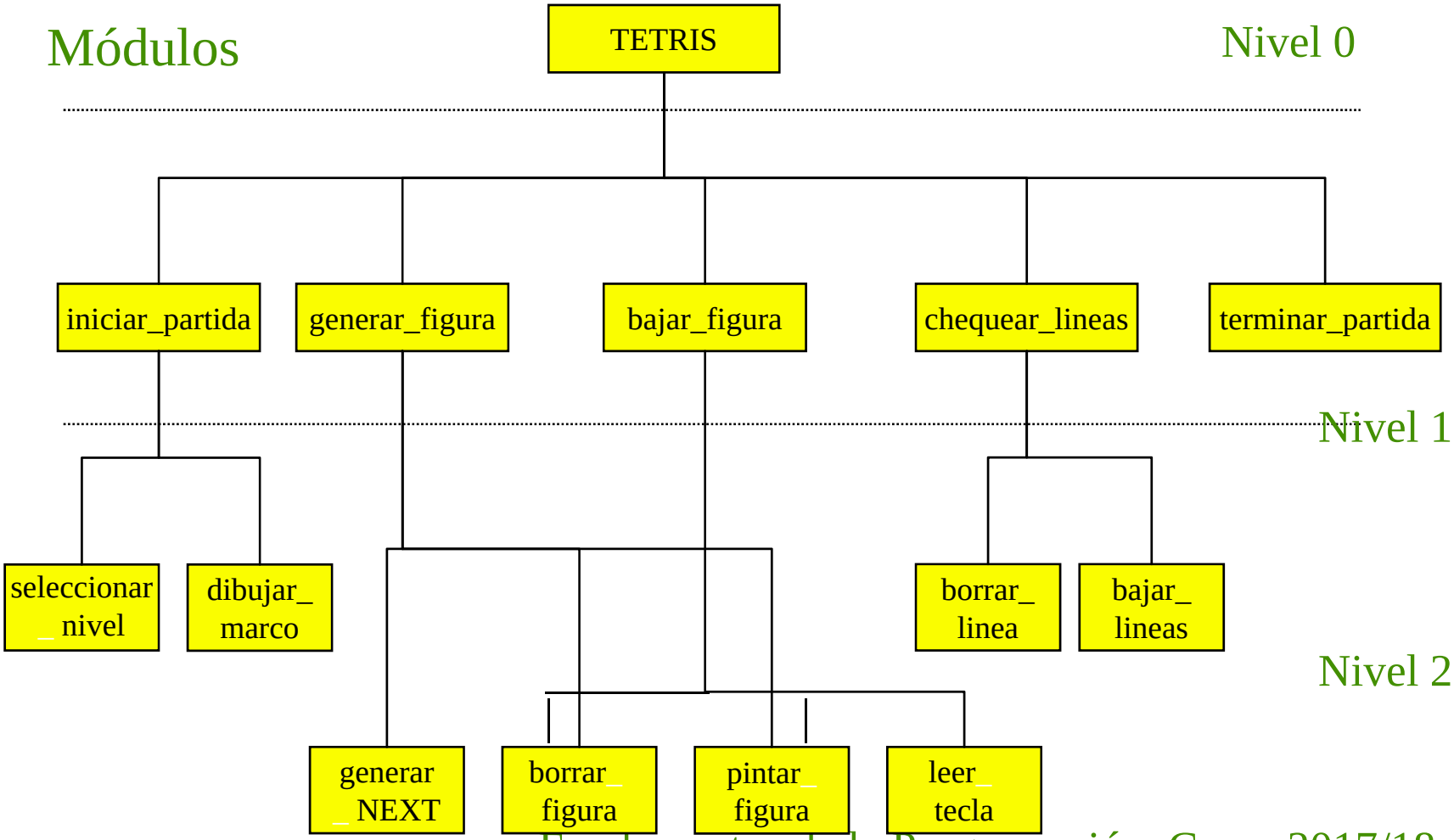
Diseño Descendente

Ejemplo: Convertir grados sexagesimales a radianes




Score TETRIS Level 2
3455 Next
Teclas Pulsadas

Diseño Descendente



Diseño Descendente

- Este enfoque proporciona indudables ventajas:
  - **Simplificación del diseño** de los programas. Un **programa** (solución al problema inicial) se estructura como una **composición de subprogramas** (soluciones a los diferentes subproblemas).
  - Mejor **comprensión y legibilidad** de los programas.
  - **Facilita la depuración** de errores.
  - **Programación aislada**.
  - Posibilidad de **reutilización** de los subprogramas.
  - Abstracción Procedimental

Diseño Descendente

Abstracción Procedimental

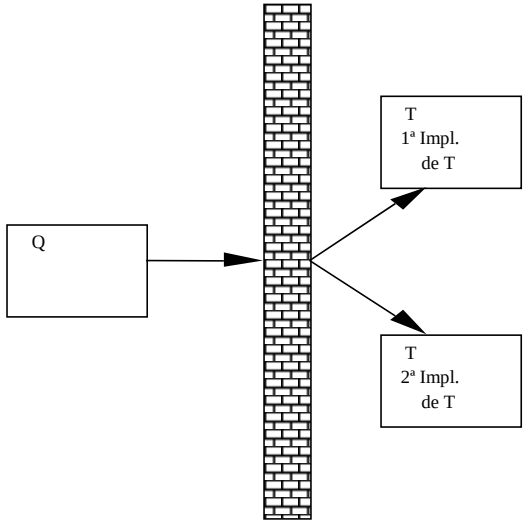
El programador, cuando diseña un subprograma:

- **NO tiene que saber cómo** otro subprograma (que necesita) resuelve su subproblema
- **SI tiene que saber qué** es lo que resuelve
  Funcionalidad de los módulos
  - Qué problema resuelve
  - Qué datos necesita
  - Qué datos produce

Diseño Descendente

Abstracción Procedimental

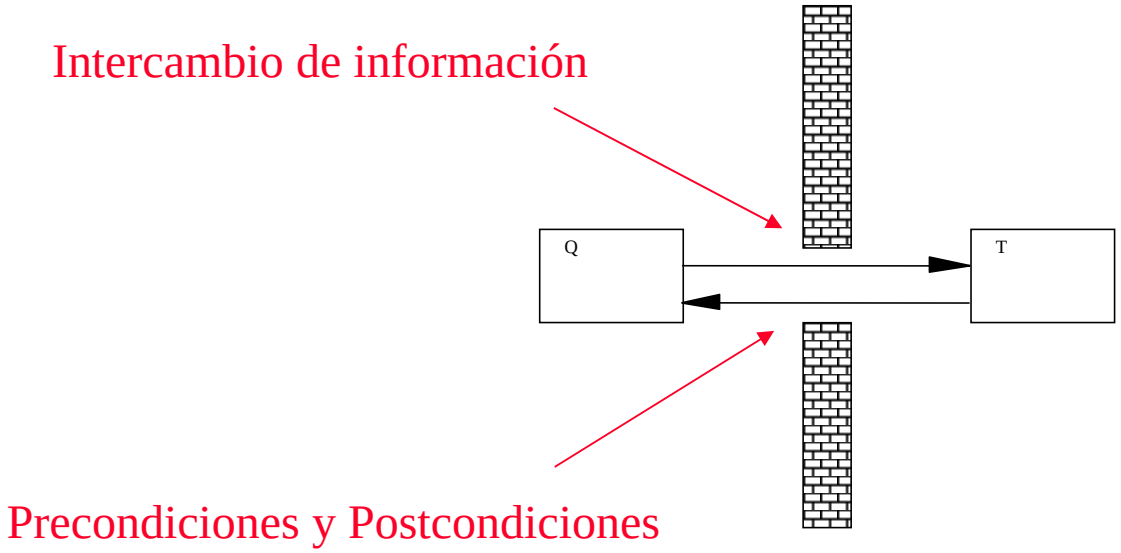
Aisla (encapsula) los diferentes subprogramas que componen un programa



Diseño Descendente

Abstracción Procedimental

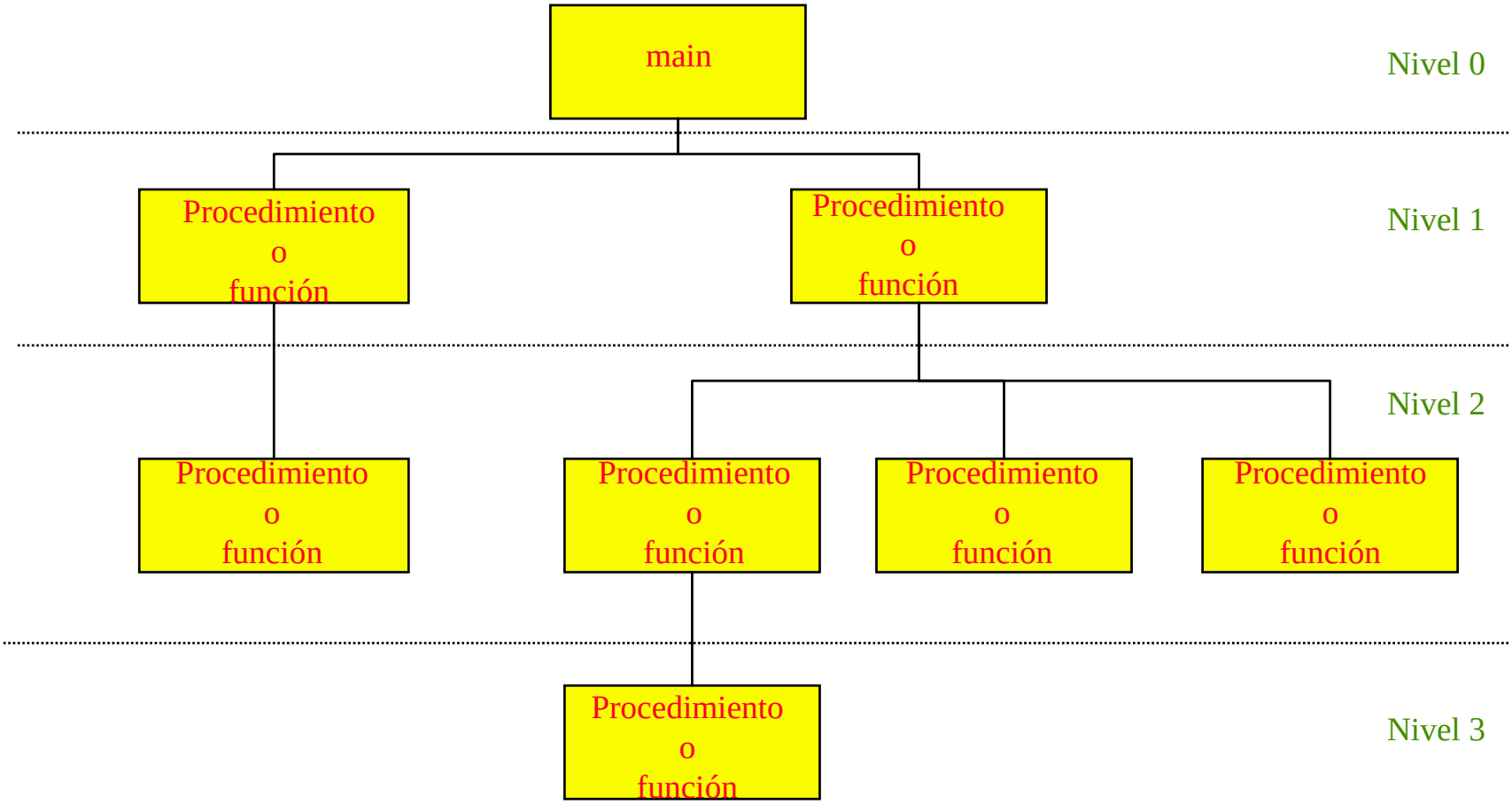
Aislamiento no puede ser total



Procedimientos y Funciones

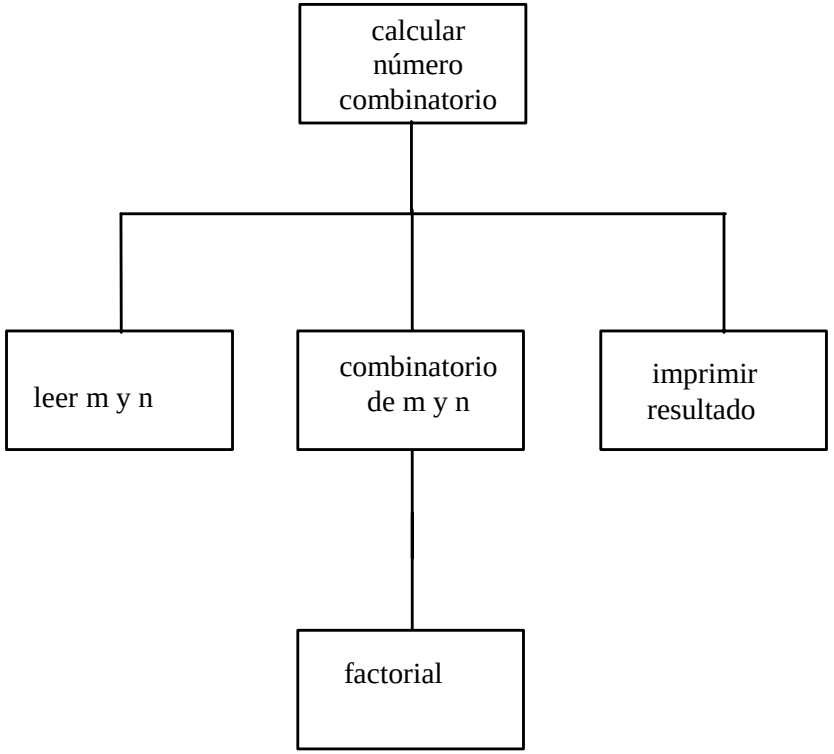
- Los **Procedimientos** y las **Funciones** son las herramientas que ofrecen los lenguajes de programación de alto nivel para codificar los distintos subprogramas.
- La solución al problema inicial, denominada programa principal se codificará con la función **main**.

Procedimientos y Funciones



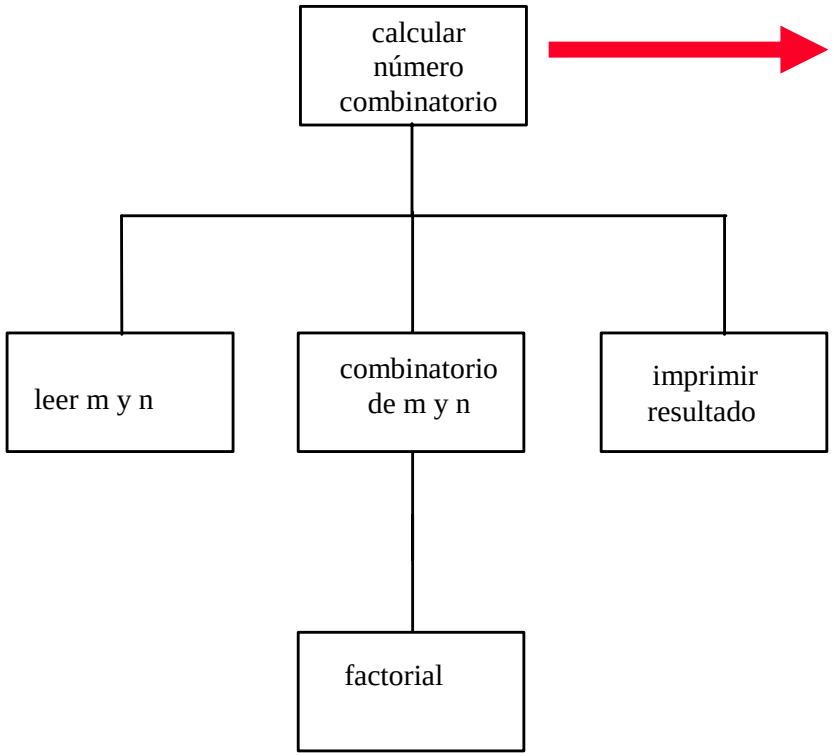
Ejemplo:
Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$



Ejemplo:
Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$

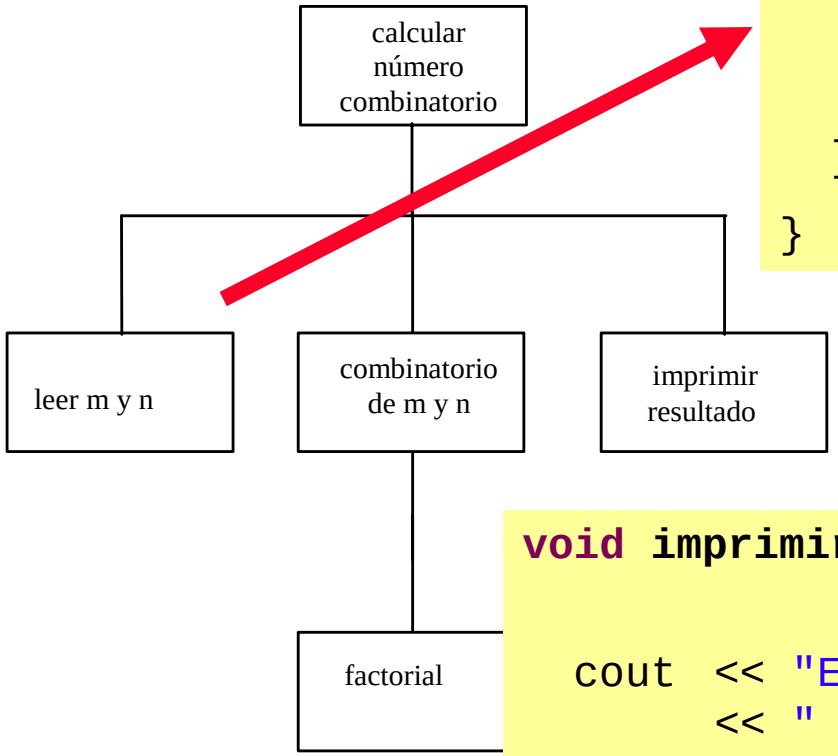



```
int main() {
        unsigned m,n,comb;
        leerDatos(m,n);
        comb = combinatorio(m,n);
        imprimirResultado(m,n,comb);
}
```

Abstracción Procedimental
- leerDatos
- combinatorio
- imprimirResultado

Ejemplo:
Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$

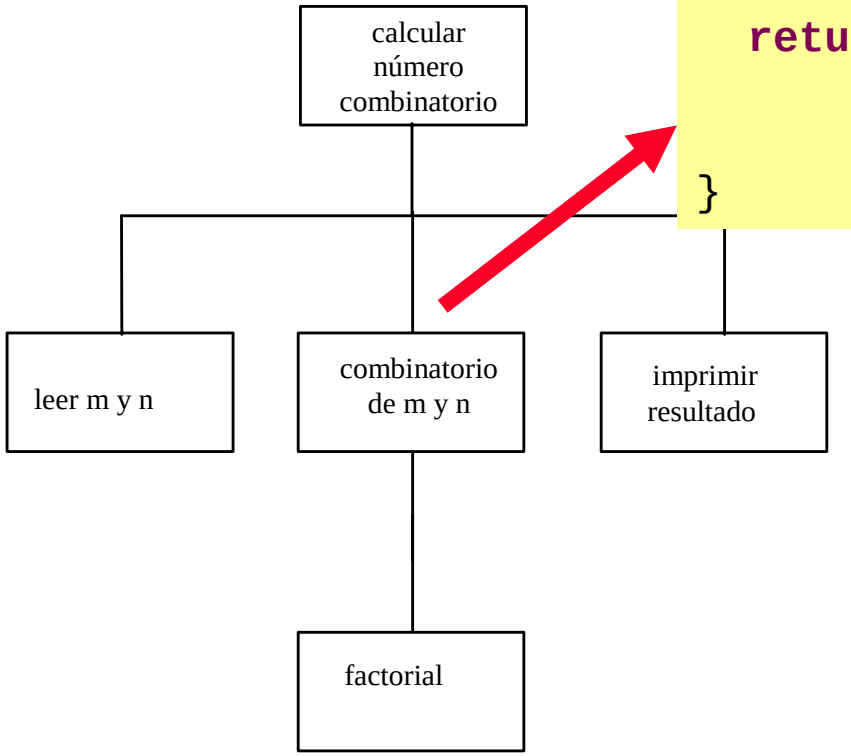



```
void leerDatos(unsigned& m, unsigned& n) {
  do {
    cout << "Introduzca m y n (m >= n): ";
    cin >> m >> n;
  } while (m < n);
}
```

```
void imprimirResultado(unsigned m, unsigned n,
                       unsigned res) {
  cout  << "El numero combinatorio de " << m
        << " sobre " << n << " es: " << res << endl;
}
```

Ejemplo:
Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$

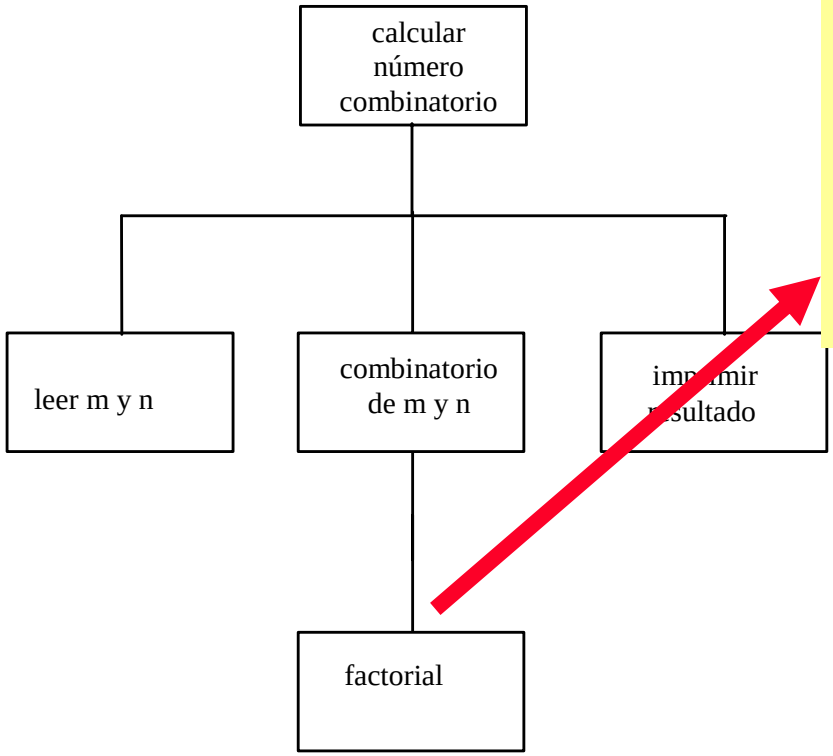



```
unsigned combinatorio(unsigned m, unsigned n)
{
  return factorial(m) /
         (factorial(n) * factorial(m-n));
}
```

Abstracción Procedimental
- factorial

Ejemplo:
Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$




```
unsigned factorial(unsigned x) {
  unsigned fact = 1;
  for (unsigned i = 2; i <= x; i++) {
        fact = fact * i;
  }
  return fact;
}
```

Definición y Declaración de Procedimientos y Funciones

Definición de un Procedimiento

cabecera

cuerpo

```
void nombreProcedimiento(Parámetros Formales) {
  declaración de variables
  secuencia de sentencias
}
```

```
void imprimirResultado(unsigned m, unsigned n, unsigned res) {
  cout  << "El numero combinatorio de " << m  << " sobre " << n
        << " es: " << res << endl;
}
```

Definición y Declaración de Procedimientos y Funciones

Definición de un Procedimiento

cabecera

cuerpo

```
void nombreProcedimiento(Parámetros Formales) {
  declaración de variables
  secuencia de sentencias
}
```

```
void leerDatos(unsigned& m, unsigned& n) {
  do {
    cout << "Introduzca m y n (m >= n): ";
    cin >> m >> n;
  } while (m < n);
}
```

Definición y Declaración de Procedimientos y Funciones

Definición de una Función

cabecera

cuerpo

```
TipoResultado nombreFuncion(Parámetros Formales) {
  declaración de variables
  secuencia de sentencias (última sentencia "return"
                           para devolver el resultado)
}
```

```
unsigned factorial(unsigned x) {
  unsigned fact = 1;
  for (unsigned i = 2; i <= x; i++) {
        fact = fact * i;
  }
  return fact;
}
```

Definición y Declaración de Procedimientos y Funciones

- En el ejemplo del número combinatorio hemos visto que primero codificamos la función `main` y después los procedimientos y funciones que ésta necesita.
- Este es el orden adecuado de hacerlo ya que encaja con la metodología del diseño descendente y la abstracción procedimental.
- Pero, al igual que ocurre con el resto de entidades en C++ (variables, constantes, …), los procedimientos y funciones deben aparecer antes de ser utilizados.
- Una alternativa es el uso de prototipos (declaración)

```
cabecera_proc_o_func;
```

Llamada a Procedimientos y Funciones. Parámetros Reales

Llamada a un Procedimiento

```
nombreProcedimiento(Parámetros Reales);
```

Constituye por sí misma una sentencia

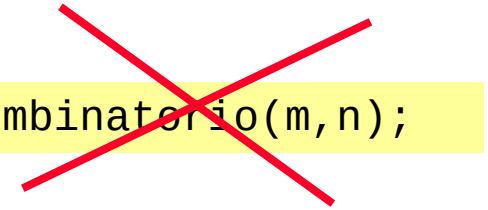
```
leerDatos(m,n);
```

Llamada a Procedimientos y Funciones. Parámetros Reales

Llamada a una Función

```
nombreFuncion(Parámetros Reales)
```

NO constituye por sí misma una sentencia




```
comb = combinatorio(m,n);
```

Reglas de uso de Parámetros

1) Número de parámetros formales = Número de parámetros reales.
2) El i-ésimo parámetro formal se corresponde con el i-ésimo parámetro real
3) El tipo del i-ésimo parámetro formal debe ser igual que el tipo del i-ésimo parámetro real.
4) Los parámetros de un procedimiento o una función pueden ser de cualquier tipo, al igual que cualquier variable.
5) Los nombres de un parámetro formal y su correspondiente real pueden o no ser iguales.
6) Un parámetro formal pasado por valor (sin &) permite variables, constantes y expresiones como parámetro real. En cambio, un parámetro formal pasado por referencia (con &) requiere una variable como parámetro real. (Más adelante se verá el paso por valor y por referencia en detalle)

Reglas de uso de Parámetros Ejemplos

Supongamos que en la función `main` se declaran las siguientes variables:

```
double x,y;
int m;
char c;
```

y tenemos un procedimiento con la siguiente cabecera:

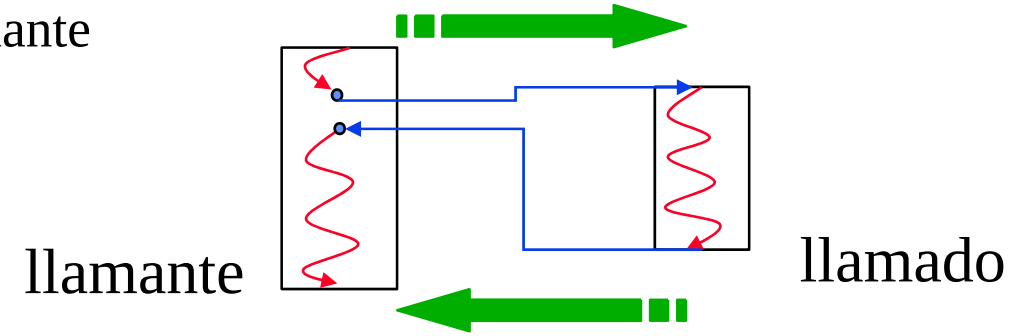
```
void prueba(int a, int b, double& c, double& d, char e)
```

Estudia cuáles de las siguientes llamadas a `prueba` desde `main` son incorrectas y cuál es la razón:

```
prueba(m+3,10,x,y,c)
prueba(m,19,x,y)
prueba(35,m*10,x,c,y)
prueba(m,3.5,x,y,c)
prueba(30,10,x,x+y,c)
prueba(30,10,m,x,c)
prueba(m,m*m,y,x,c)
prueba(m,10,35.0,y,'E')
prueba(30,10,c,d,e)
```

Flujo de Control

- Cuando se produce una llamada a un subprograma:
  - Se establecen las vías de comunicación entre llamante y llamado (a través de los parámetros)
  - Se crean las variables declaradas en el llamado
  - El flujo de control pasa a la primera instrucción del llamado
- Cuando acaba el subprograma llamado:
  - Se destruyen las variables creadas en el llamado
  - El flujo de control continúa por la instrucción que sigue a la de llamada en el llamante



Paso de Parámetros por Valor y por Referencia

– Ya sabemos que los parámetros constituyen las vías de comunicación para intercambiar información entre subprogramas (procedimientos y funciones):
  - Del llamante al llamado -> información de **entrada**
  - Del llamado al llamante -> información de **salida**
  - En ambos sentidos -> información de **entrada/salida**
– En la práctica, los lenguajes de programación normalmente implementan este flujo de información mediante:
  - **Paso por valor**
  - **Paso por referencia**

Paso de Parámetros por Valor y por Referencia

- Paso por Valor

  Se **almacena** en el parámetro formal **una copia** del parámetro real. Cualquier **modificación** en el formal **no afecta** al real.

  Sintaxis: `TipoDato parametro`

- Paso por Referencia

  Se **almacena** en el parámetro formal **una referencia** a la variable situada como parámetro real. Cualquier **modificación** en el formal implica una modificación del real.

  Sintaxis: `TipoDato& parametro`

Paso de Parámetros por Valor y por Referencia

- Paso de parámetros por Valor
  - Sólo permite comunicación de entrada (Ej. `imprimirResultado`)
  - Aísla
  - Permite variables, constantes y expresiones como parámetro real
  - Duplica memoria y realiza copia
  - Utiliza más memoria con los tipos estructurados

```
void imprimirResultado(unsigned m, unsigned n,
                       unsigned res) {

  cout  << "El numero combinatorio de " << m
        << " sobre " << n << " es: " << res << endl;

}
```

Paso de Parámetros por Valor y por Referencia

- Paso de parámetros por Referencia
  - Permite comunicación de salida si el parámetro formal se utiliza para comunicar un valor al llamante (p. ej. `leerDatos`) y de entrada/salida si se modifica el valor contenido en el parámetro formal (p. ej. `intercambiar`).

```
void leerDatos(unsigned& m, unsigned& n) {
  do {
    cout << "Introduzca m y n (m >= n): ";
    cin >> m >> n;
  } while (m < n);
}
```

```
void intercambiar(int& x, int& y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

Paso de Parámetros por Valor y por Referencia

- Paso de parámetros por Referencia
  - También permite comunicación de entrada si no hay modificación del parámetro real. Normalmente se utiliza para los tipos de datos estructurados (se verá en el próximo tema)
  - No aisla.
  - Sólo permite una variable como parámetro real.
  - No duplica memoria y no realiza copia.
  - Utiliza menos memoria en el caso de tipos estructurados.

UNIVERSIDAD DE MÁLAGA

Interfaz

Genéricamente. Define la interacción entre dos entidades independientes

En programación. Define la forma en que se comunican y cooperan dos subprogramas.

D**iseño del interfaz** de un subprograma:

– Qué información necesita para resolver el problema

– Qué información produce como resultado

– Bajo qué condiciones se realiza el intercambio de información

UNIVERSIDAD DE MÁLAGA

Criterios de Modularización

- Los subprogramas también se denominan **módulos**.
- No existen mecanismos formales para determinar **cómo descomponer un problema en módulos**, es una **labor subjetiva**.
- Podemos guiarnos por algunos criterios generales:
  - Acoplamiento
  - Cohesión

Criterios de Modularización

ACOPLAMIENTO

- Un objetivo en el diseño descendente es crear módulos aislados e independientes.
- Debe haber alguna interacción entre módulos para formar un sistema coherente.
- Dicha interacción se conoce como acoplamiento.
- Maximizar la independencia será **minimizar el acoplamiento**.

Criterios de Modularización

COHESIÓN

- Hace referencia al grado de relación entre las diferentes partes internas a un módulo.
- Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro de un módulo es tal que posteriores modificaciones podrán resultar complicadas.
- Se busca **maximizar la cohesión** dentro de cada módulo.

Criterios de Modularización
Ej. Escribir si un número es primo

Opción 1

```
void escribirSiPrimo(unsigned num) {
  unsigned divisor;

  divisor = 2;
  while ((divisor < num) &&
         (num % divisor != 0)) {
    divisor++;
  }
  if (divisor < num) {
    cout << num << " NO es primo\n";
  } else {
    cout << num << " SI es primo\n";
  }
}
```

BAJA COHESIÓN

Opción 2

```
bool esPrimo(unsigned num) {
  unsigned divisor;

  divisor = 2;
  while ((divisor < num) && (num % divisor != 0)) {
    divisor++;
  }
  return divisor >= num;
}
void escribirSiPrimo(unsigned num) {
  if (esPrimo(num)) {
    cout << num << " SI es primo\n";
  } else {
    cout << num << " NO es primo\n";
  }
}
```

ALTA COHESIÓN

Criterios de Modularización
Ej. Suma de la serie $1, x, x^2/2!, x^3/3!, \ldots$

```
int main() {

  double x, suma;

  cout << "X: ";
  cin >> x;
  suma = 0;
  cout << "La suma de la serie es: "
       << sumaSerie(suma,x) << endl;
}
```

```
double sumaSerie(double s, double x) {

  double term, res;
  unsigned i;

  i = 0;
  res = s;
  do {
    term = calcTerm(i,x);
    res = res + term;
    i++;
  } while (term >= UMBRAL);
  return res;
}
```

ALTO ACOPLAMIENTO

Criterios de Modularización
Ej. Suma de la serie $1, x, x^2/2!, x^3/3!, \ldots$

```
int main() {

  double x;

  cout << "X: ";
  cin >> x;
  cout << "La suma de la serie es: “
       << sumaSerie(x) << endl;
}
```

```
double sumaSerie(double x) {

  double term, res;
  unsigned i;

  i = 0;
  res = 0;
  do {
    term = calcTerm(i,x);
    res = res + term;
    i++;
  } while (term >= UMBRAL);
  return res;
}
```

BAJO ACOPLAMIENTO

UNIVERSIDAD DE MÁLAGA

Variables Locales

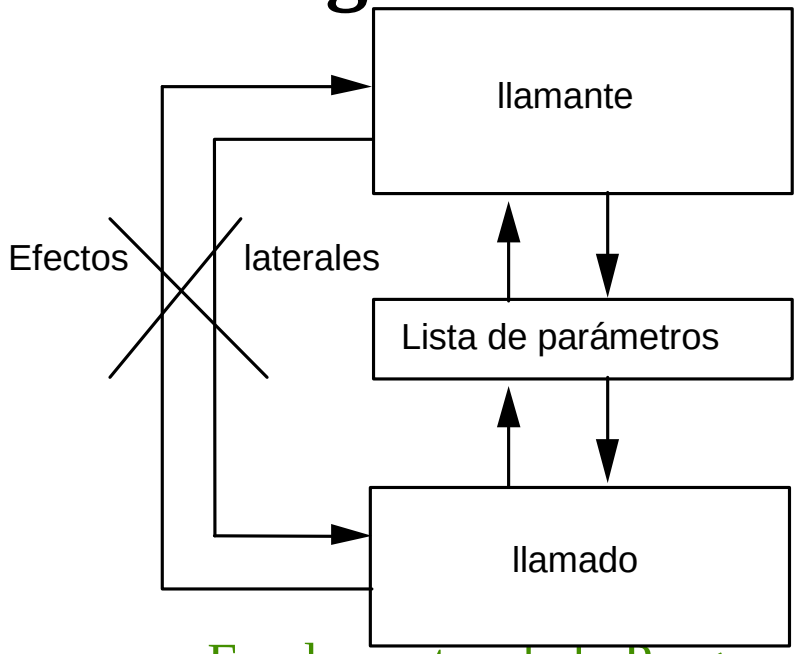
- Las variables declaradas dentro de un procedimiento o una función (incluida la función main) se denominan **variables locales** a ese procedimiento o función.
- Los parámetros formales también son variables locales.
- Una variable local sólo puede ser accedida dentro del procedimiento o función que la declara y a partir del punto donde se declara (**regla de ámbito**).
- Una variable local se crea cuando el procedimiento o función es llamado y se destruye cuando la ejecución del mismo termina.

Variables Globales

- C++ permite también declarar variables fuera de cualquier procedimiento o función. Estas variables se denominan **variables globales**
- Son accesibles por cualquier procedimiento o función que aparezca a partir de su declaración y hasta el final del fichero que contiene el programa.
- Debido a los problemas que conlleva su utilización, a lo largo de esta asignatura **NO utilizaremos variables globales**. PROHIBIDO
  - Aumenta el acoplamiento
  - Reduce la posibilidad de reutilización
  - Aumenta la posibilidad de cometer errores que son difíciles de encontrar

Efectos Laterales

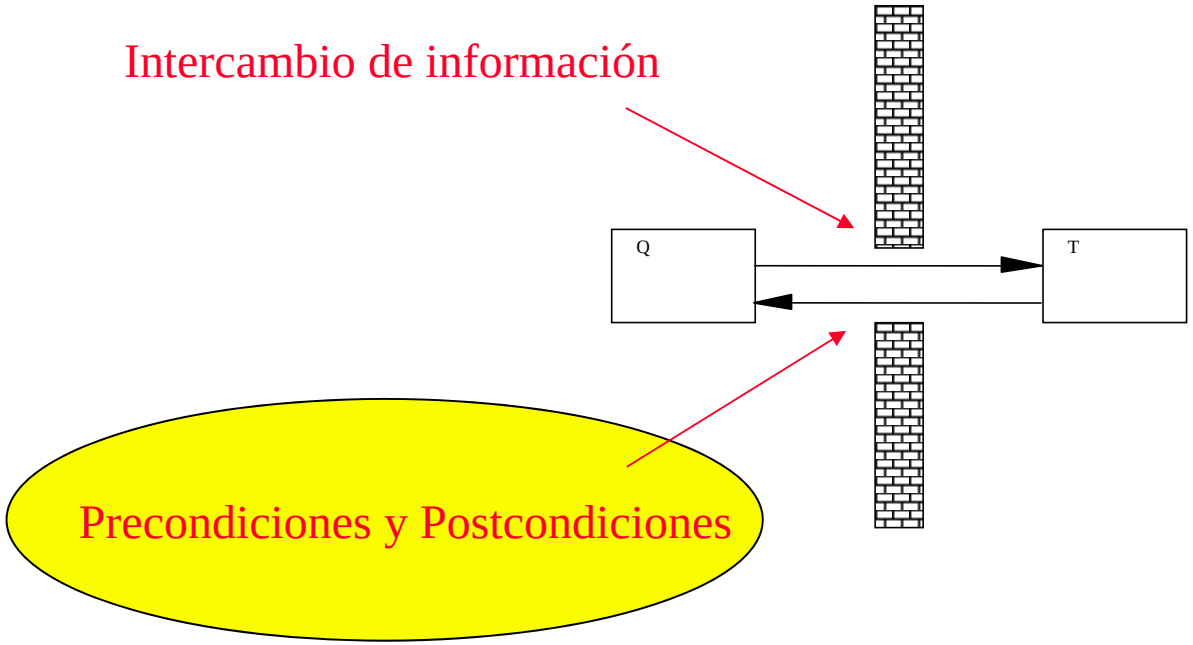
Un efecto lateral es cualquier **intercambio** de información entre dos subprogramas realizado **a través de variables globales**.



Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

Abstracción Procedimental

Aislamiento no puede ser total



Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- *Precondiciones*: condiciones que deben cumplirse antes de que el subprograma se ejecute, con objeto de garantizar que se puede realizar la tarea.

- *Postcondiciones*: condiciones que el subprograma garantiza tras finalizar su ejecución, suponiendo que las precondiciones se cumplieron cuando el subprograma fue llamado.

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- Cuando se codifica un subprograma es buena práctica anteponer unos comentarios especificando claramente las precondiciones y postcondiciones del mismo.

```
// precondicion: m >= n
// postcondicion: devuelve combinatorio de m sobre n
unsigned combinatorio(unsigned m, unsigned n) {
  return factorial(m) / (factorial(n) * factorial(m-n));
}
```

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- De cualquier forma esto no es suficiente, pues el procedimiento o función que utilice (llame) el subprograma diseñado, puede o no tener en cuenta las precondiciones de uso del mismo.
- Por ejemplo, un subprograma podría llamar a `combinatorio` con dos valores de `m` y `n` tales que `m < n`, saltándose la precondición de uso de `combinatorio` que establece que se debe cumplir `m >= n`. ¿Qué ocurriría?
- En algún momento el sistema terminará el programa con un error de ejecución inesperado o bien terminará dándonos un resultado incorrecto.

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- Cuando se diseñan procedimientos y funciones es fundamental tener en cuenta este tipo de situaciones, ya que estamos diseñando de forma separada un subprograma que será utilizado (llamado) por otro.
- Para ello podemos hacer uso del lanzamiento de excepciones.

```
// precondicion: m >= n
// postcondicion: devuelve combinatorio de m sobre n
unsigned combinatorio(unsigned m, unsigned n) {
  if (m < n) {
    throw "Error: (m < n) en funcion combinatorio";
  }
  return factorial(m) / (factorial(n) * factorial(m-n));
}
```

Recursividad

- Técnica de programación alternativa al uso de estructuras iterativas para la resolución de procesos repetitivos.
- Soluciones elegantes y simples
- Estructuradas y modulares
- Subprograma (Proc o Func) recursivo

*El que **se llama a sí mismo** para resolver una "**versión más pequeña**" del problema para el que se ha diseñado*

UNIVERSIDAD DE MÁLAGA

Recursividad

- Ejemplo: factorial de un número

  Definición iterativa:

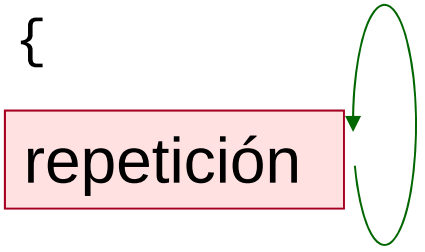
$$n! = \begin{cases} 1 & si \quad n = 0 \\ n(n-1)(n-2)...1 & si \quad n > 0 \end{cases}$$

Recursividad

- Algoritmo factorial iterativo

```
unsigned factorialIter(unsigned n) {
  unsigned fact;

  if (n == 0) {
    fact = 1;
  } else {
    fact = 1;
    for (unsigned i = 2; i <= n; i++) {
      fact = fact * i;
    }
  }
  return fact;
}
```

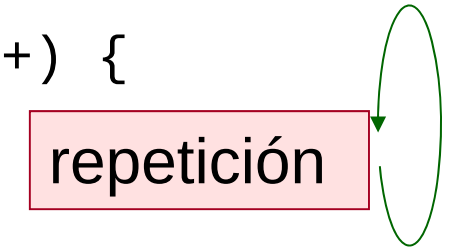


Recursividad

- Aunque también podemos prescindir del **if**

```
unsigned factorialIter(unsigned n) {
  unsigned fact = 1;

  for (unsigned i = 2; i <= n; i++) {
    fact = fact * i;
  }
  return fact;
}
```



UNIVERSIDAD DE MÁLAGA

Recursividad

- Ejemplo: factorial de un número

  Definición recursiva:

$$n! = \begin{cases} 1 & si \quad n = 0 \\ n\,(n-1)! & si \quad n > 0 \end{cases}$$

Recursividad

- Algoritmo factorial recursivo

```
unsigned factorialRec(unsigned n) {
  unsigned fact;

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

problema más pequeño

Llamada recursiva

Ejemplo de ejecución

`factorialRec(4)`

RES

?

n

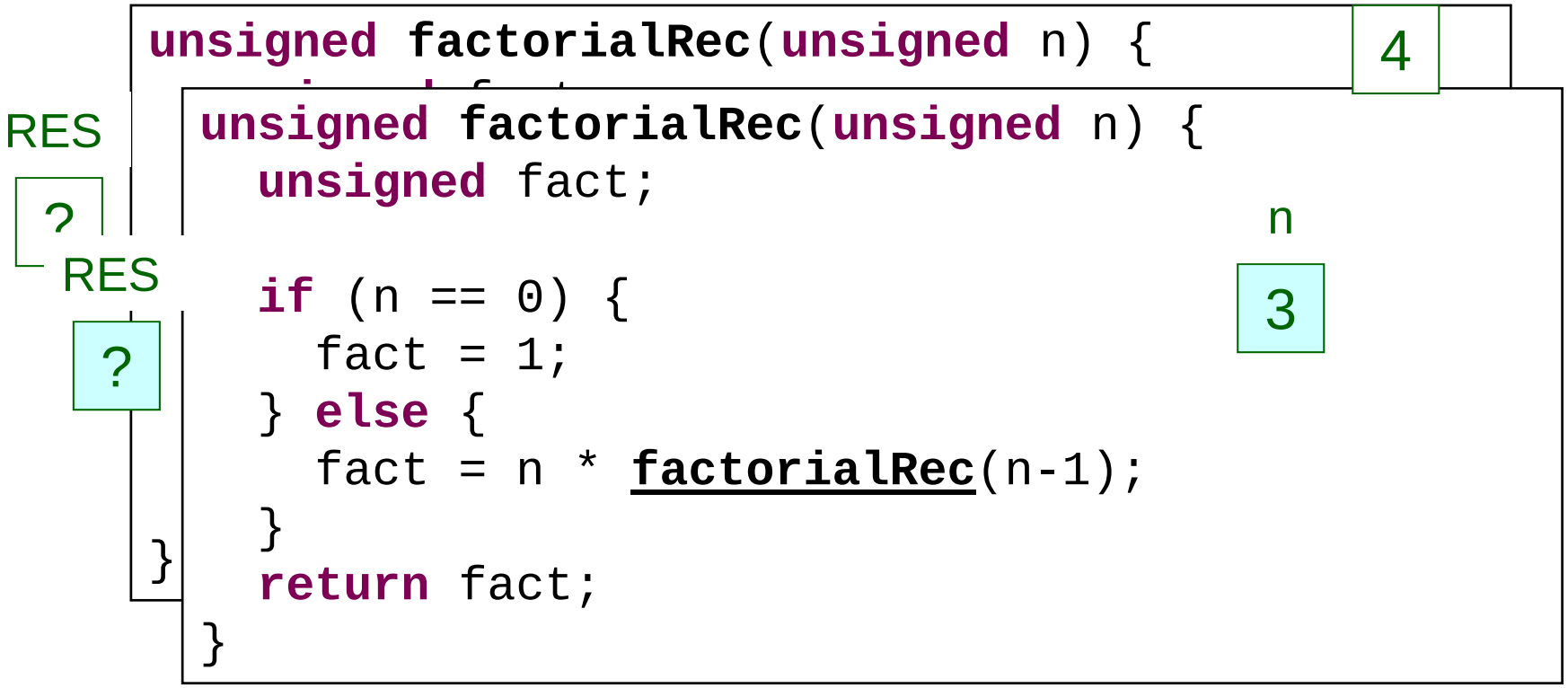
4

```
unsigned factorialRec(unsigned n) {
  unsigned fact;

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

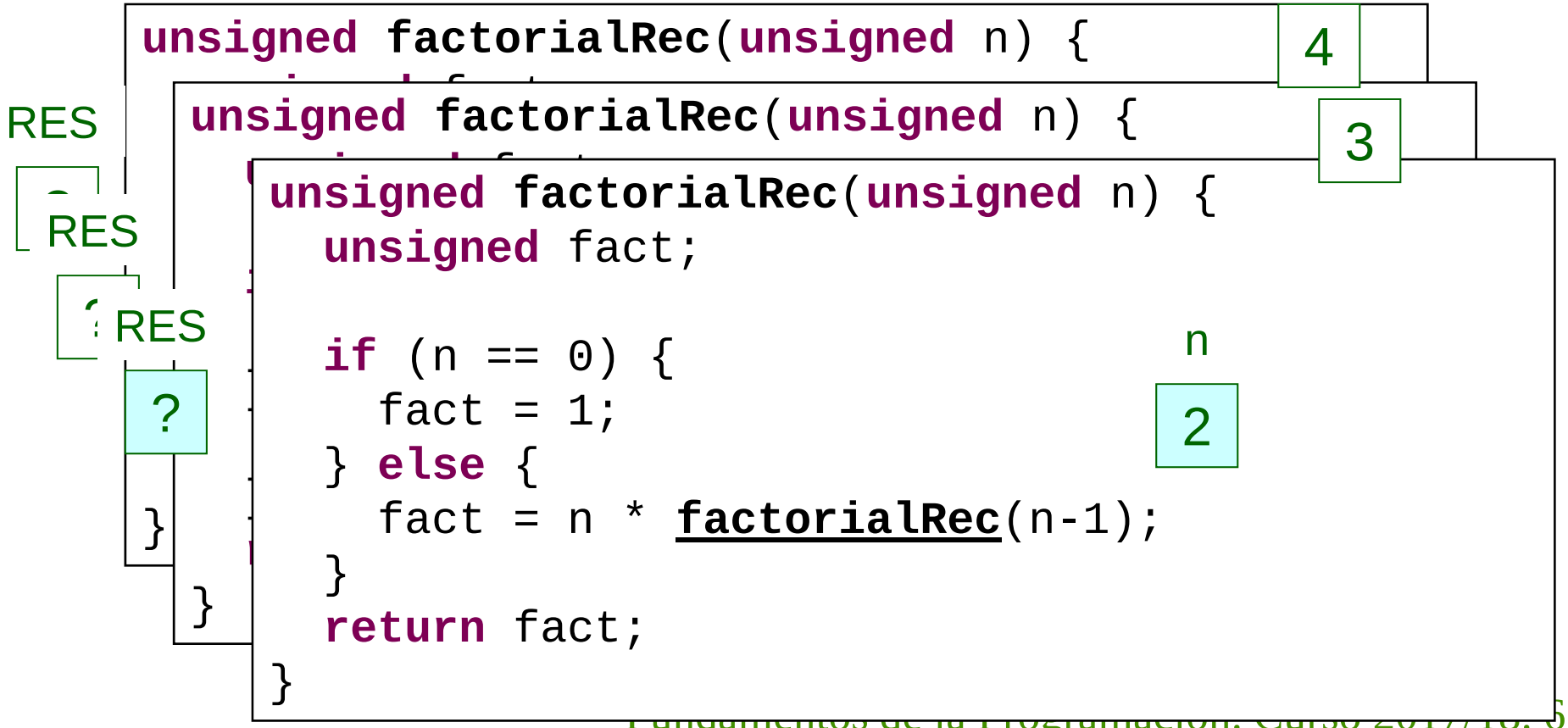
Ejemplo de ejecución

`factorialRec(4)`



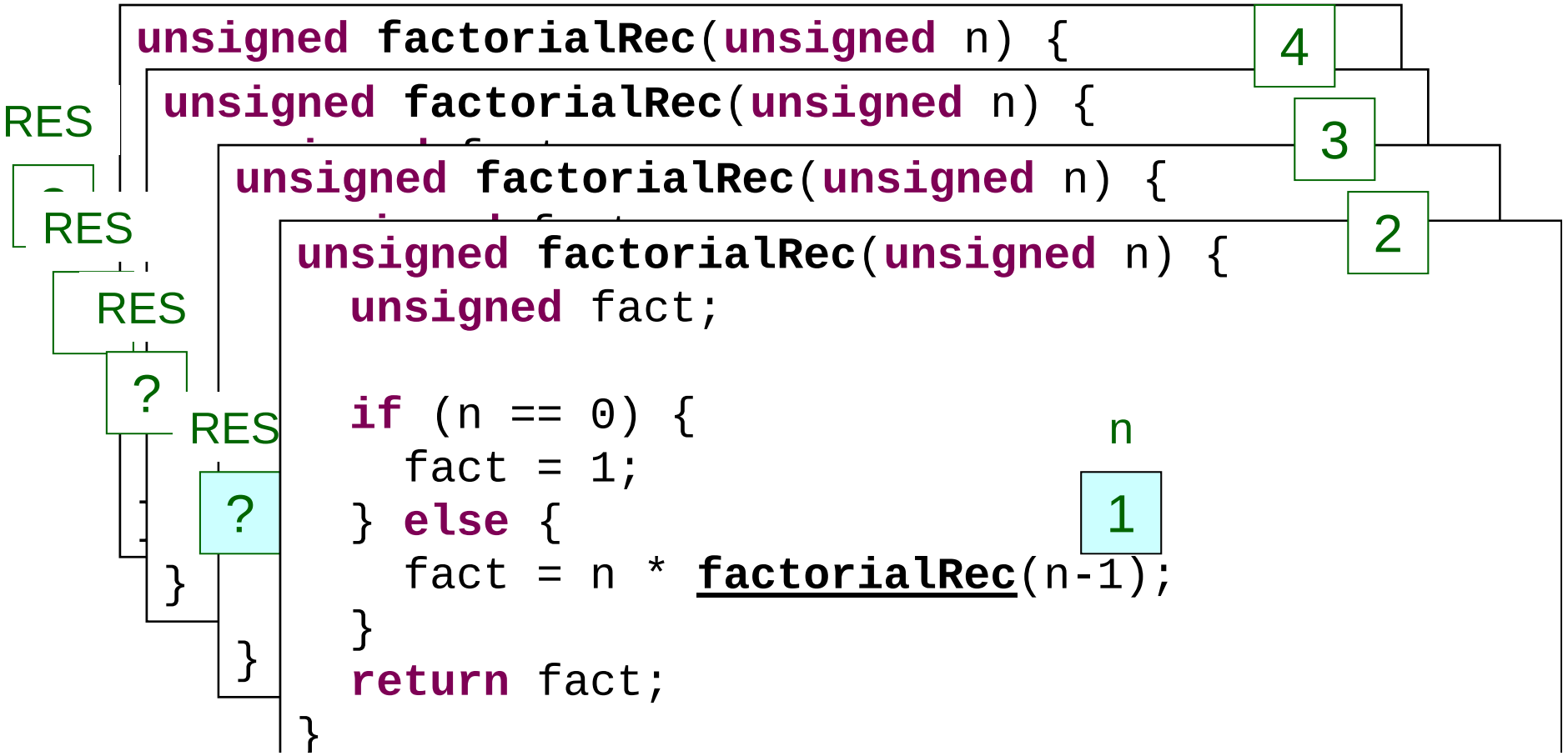
Ejemplo de ejecución

`factorialRec(4)`



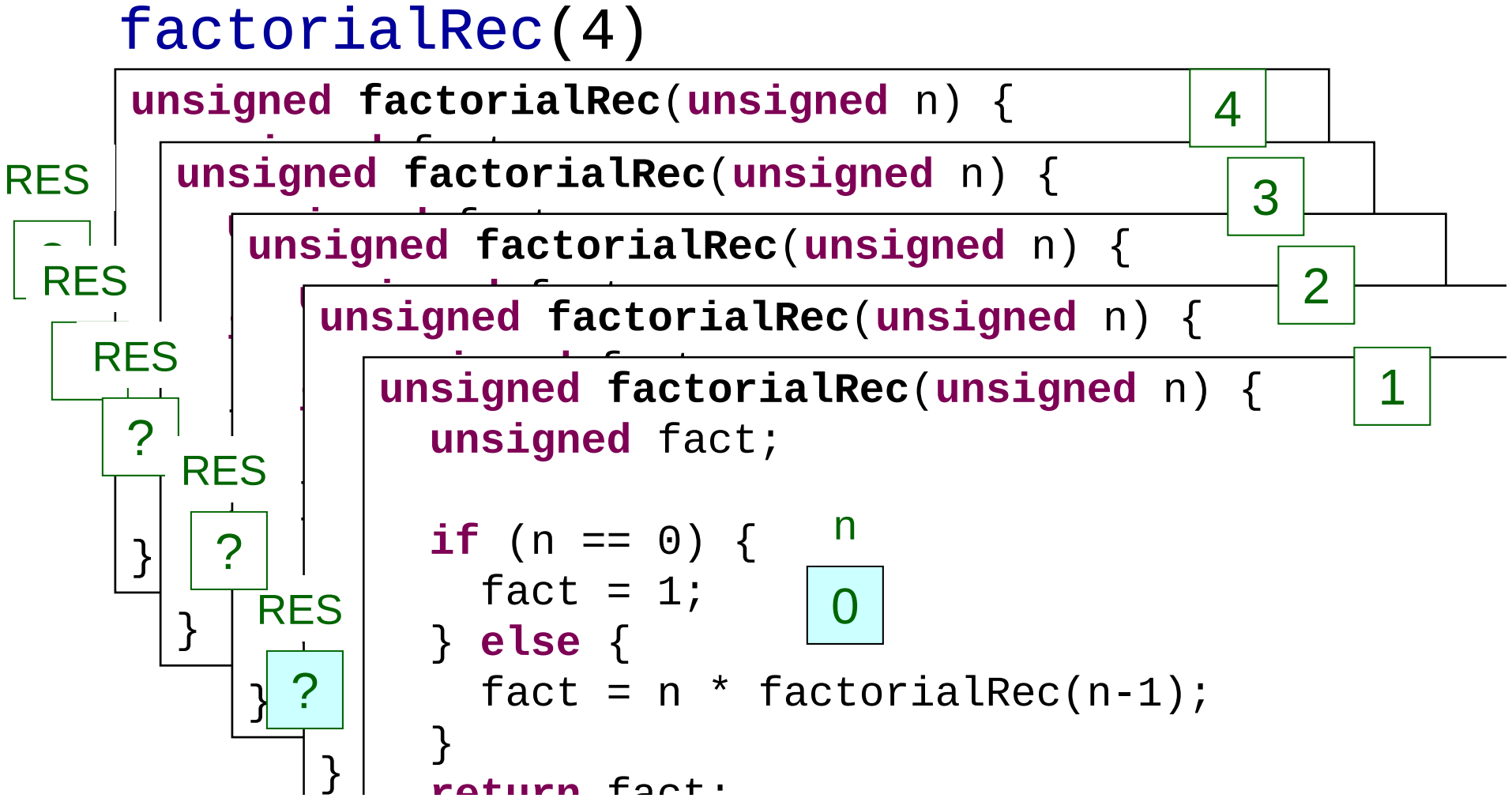
Ejemplo de ejecución

`factorialRec(4)`

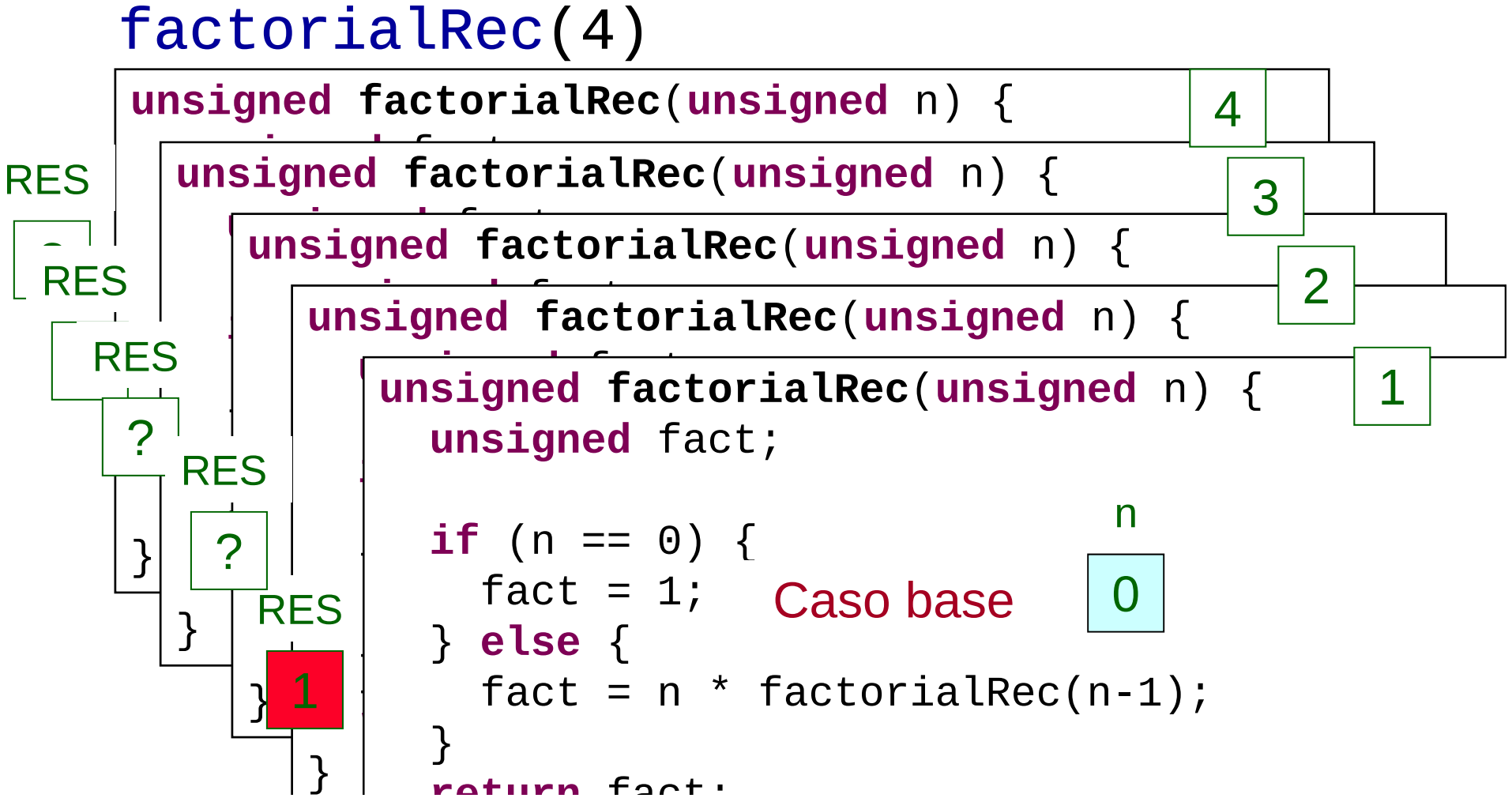


UNIVERSIDAD DE MÁLAGA

Ejemplo de ejecución

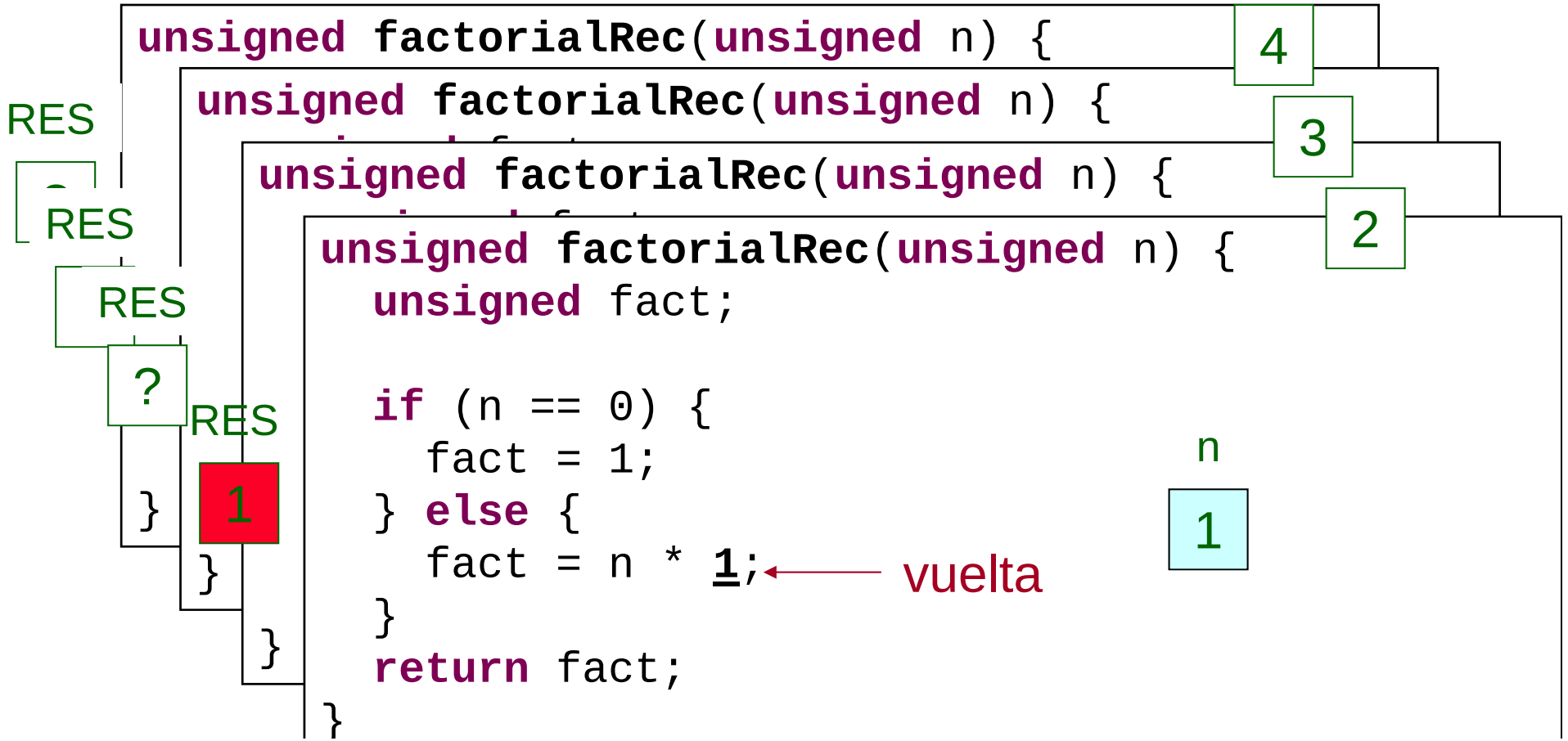


Ejemplo de ejecución



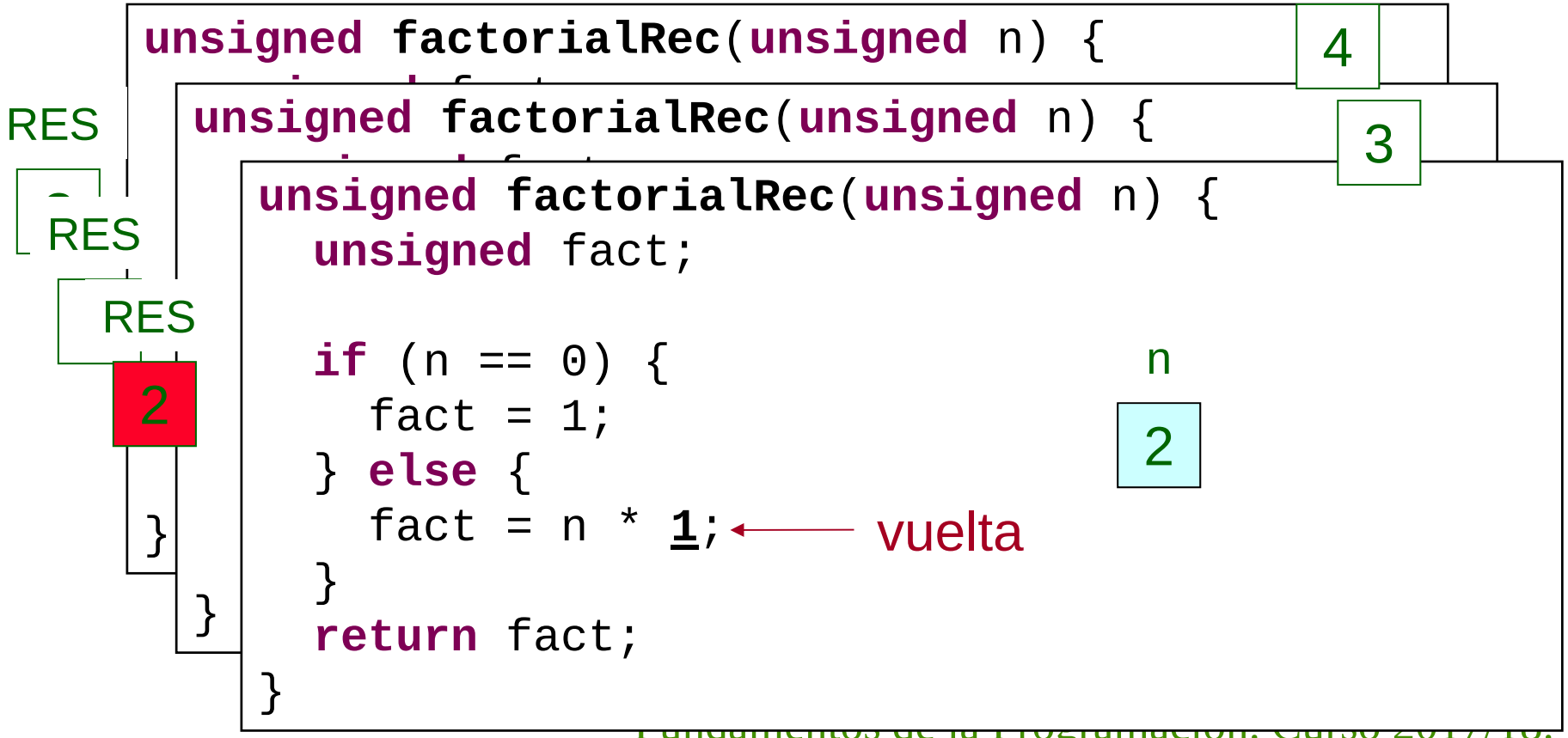
Ejemplo de ejecución

`factorialRec(4)`



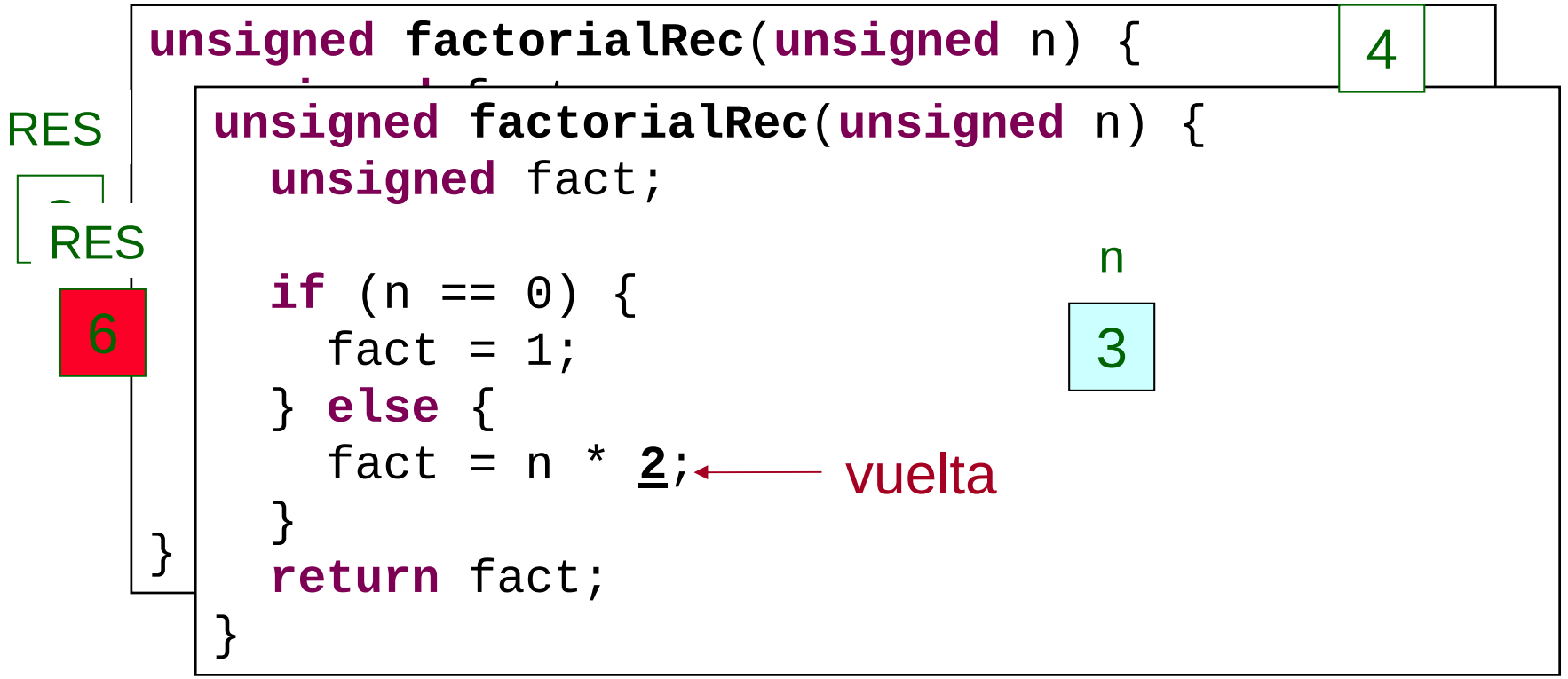
Ejemplo de ejecución

`factorialRec(4)`



Ejemplo de ejecución

`factorialRec(4)`



Ejemplo de ejecución

`factorialRec(4)`

RES

24

```
unsigned factorialRec(unsigned n) {
  unsigned fact;

  if (n == 0) {
    fact = 1;
  } else {
    fact = n * 6;
  }
  return fact;
}
```

n

4

vuelta

Recursividad

Otro Ejemplo sencillo

- Disponemos de una pareja de conejos y deseamos saber cuántas parejas hay al cabo de n meses si:
  - Los conejos nunca mueren
  - Una pareja de conejos pueden reproducirse al comienzo de su tercer mes de vida
  - Cada pareja madura genera una nueva al comienzo de cada mes

Recursividad

$$Parejas(n) = \begin{cases} 1 & \text{si} \quad n \leq 2 \\ Parejas(n-1) + Parejas(n-2) & \text{si} \quad \text{otro caso} \end{cases}$$

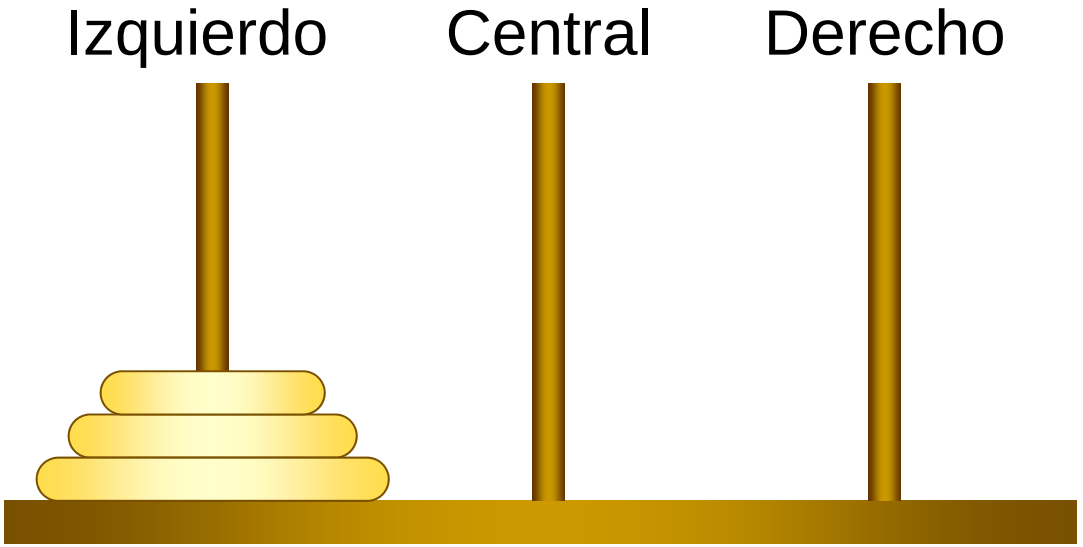
```
unsigned parejas(unsigned n) {
  unsigned res;

  if (n <= 2) {
    res = 1;
  } else {
    res = parejas(n-1) + parejas(n-2);
  }
  return res;
}
```

Un Ejemplo más complejo

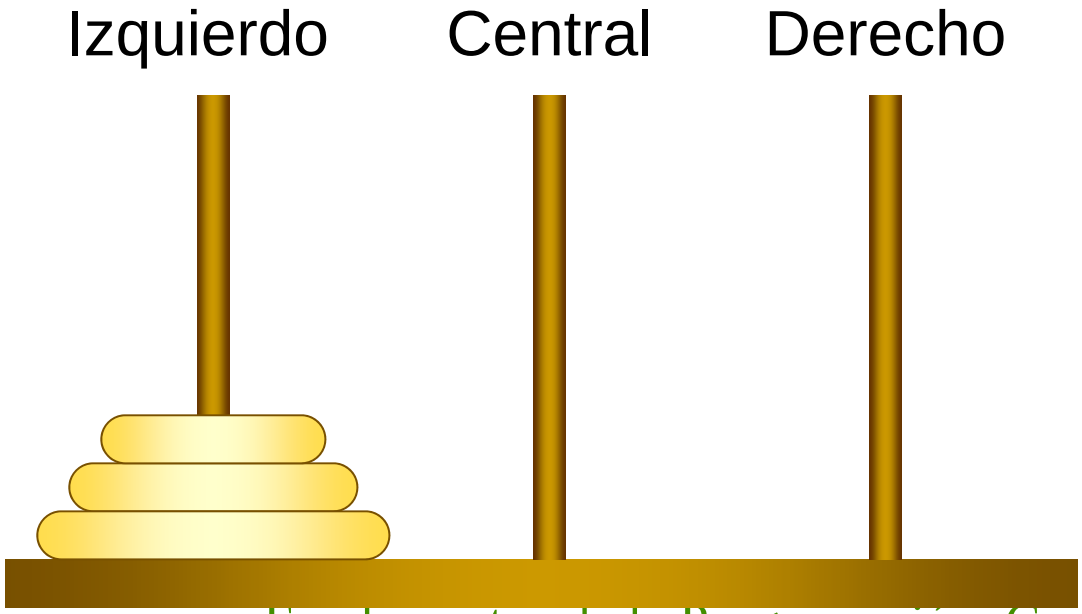
- Torres de Hanoi

  Se tienen 3 palos de madera (izquierdo, central, derecho). El palo izquierdo tiene ensartados un montón de discos concéntricos de tamaño decreciente, de manera que el disco mayor está abajo y el menor arriba.



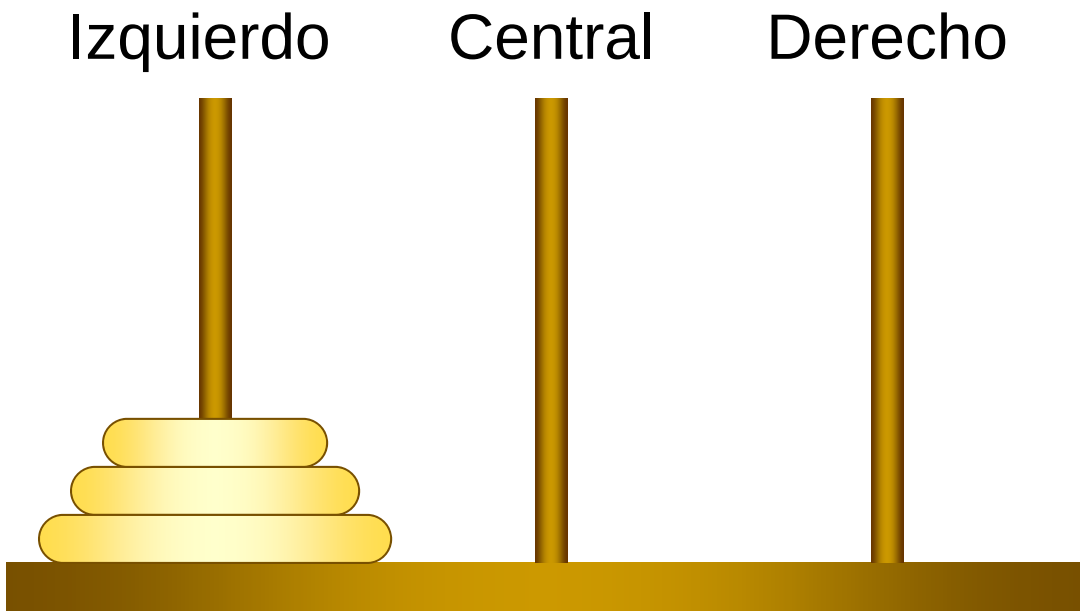
Recursividad

- **Problema.** Mover los discos del palo izquierdo al derecho respetando las siguientes reglas:
  - Sólo se puede mover un disco cada vez.
  - No se puede poner un disco encima de otro más pequeño.
  - Después de un movimiento todos los discos han de estar en alguno de los tres palos.



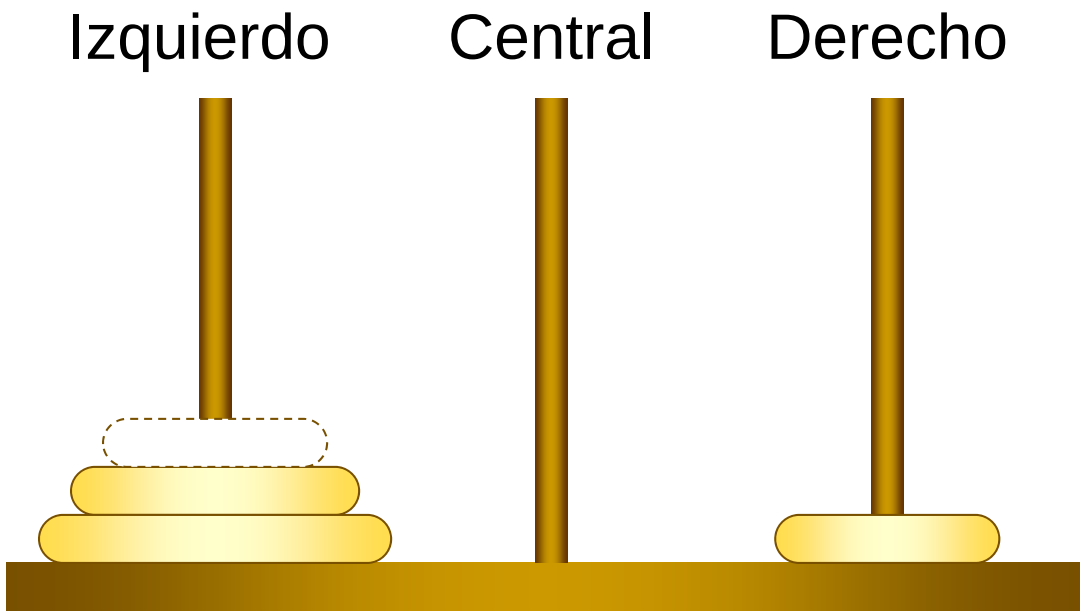
Recursividad

- Ejemplo: 3 discos



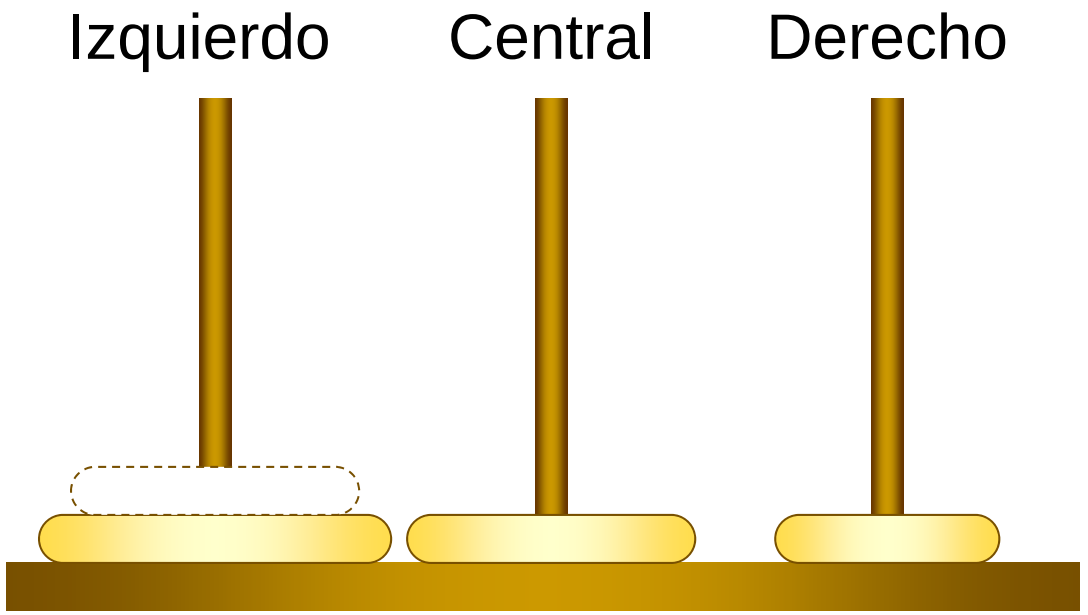
Recursividad

- Ejemplo: 3 discos



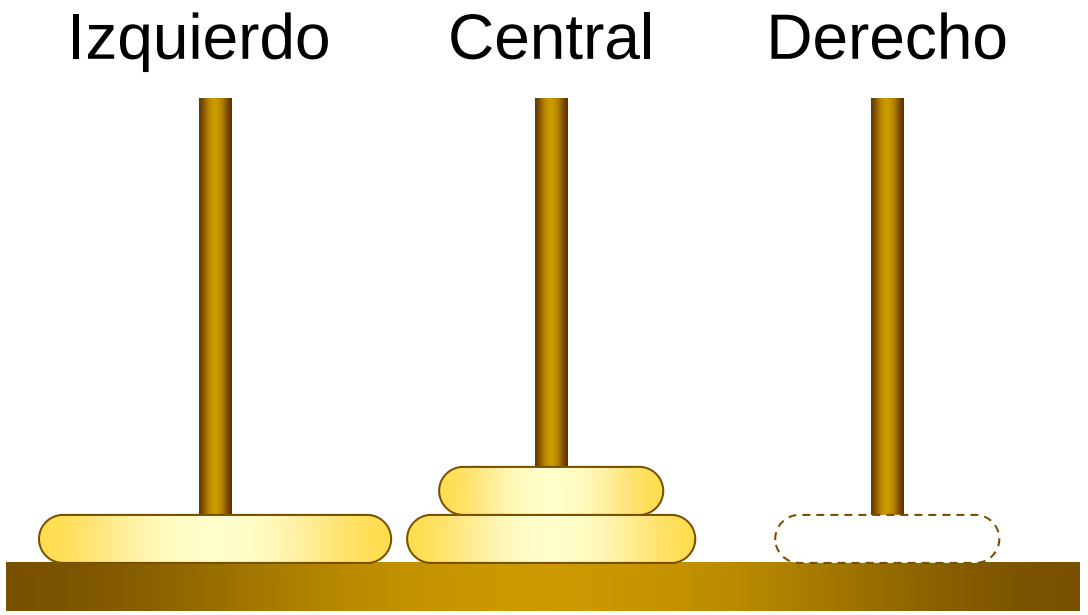
Recursividad

- Ejemplo: 3 discos



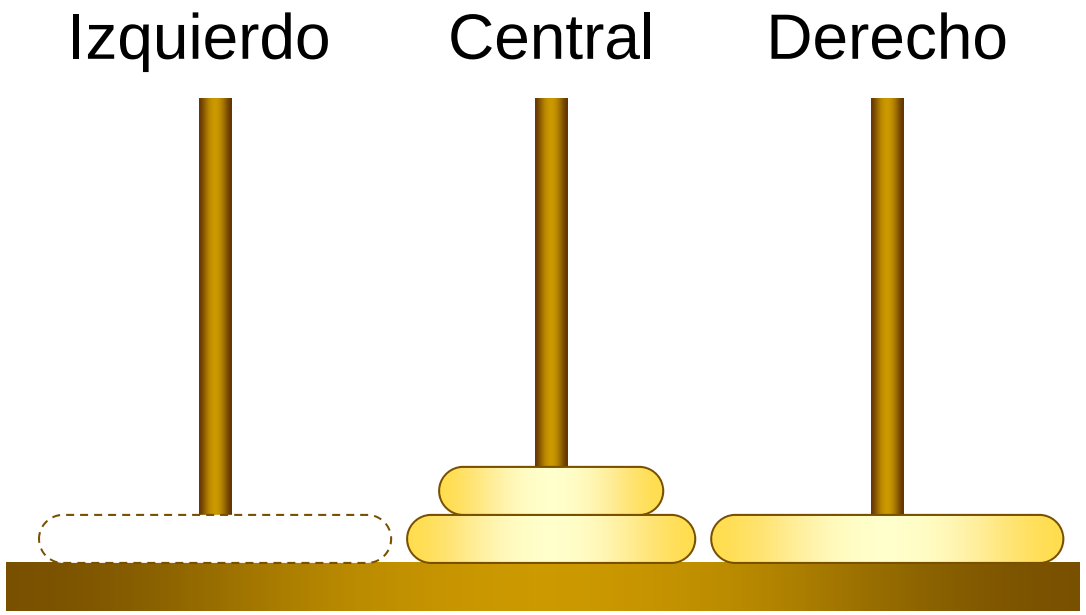
Recursividad

- Ejemplo: 3 discos



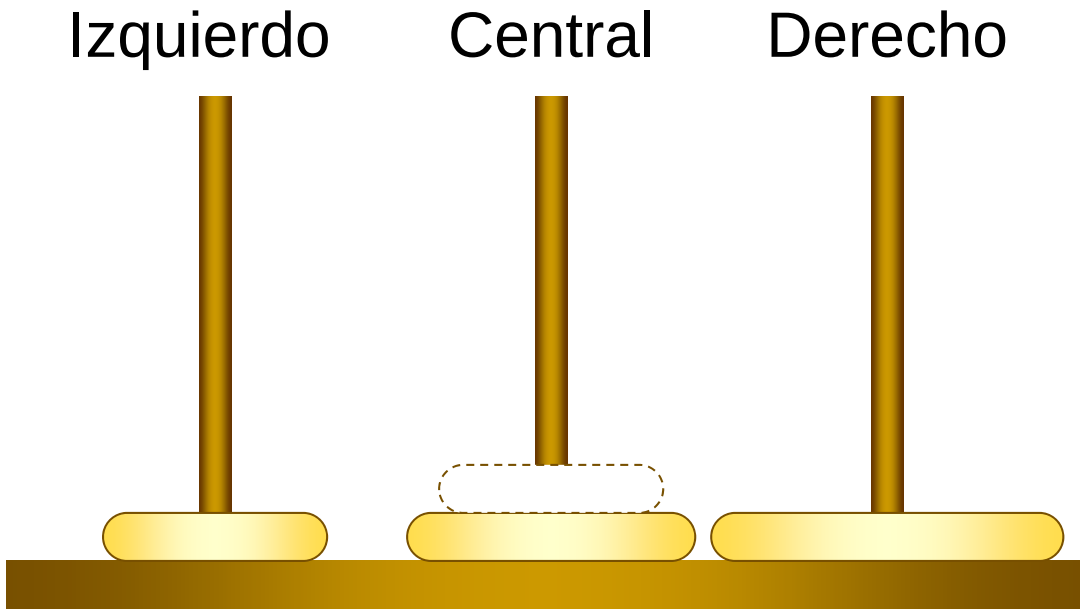
Recursividad

- Ejemplo: 3 discos



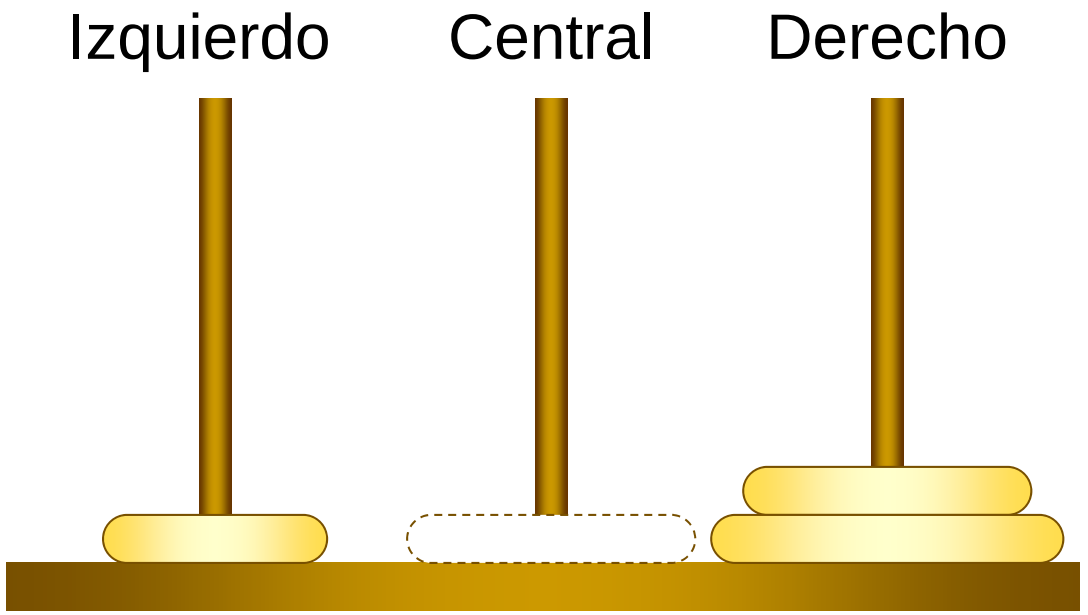
Recursividad

- Ejemplo: 3 discos



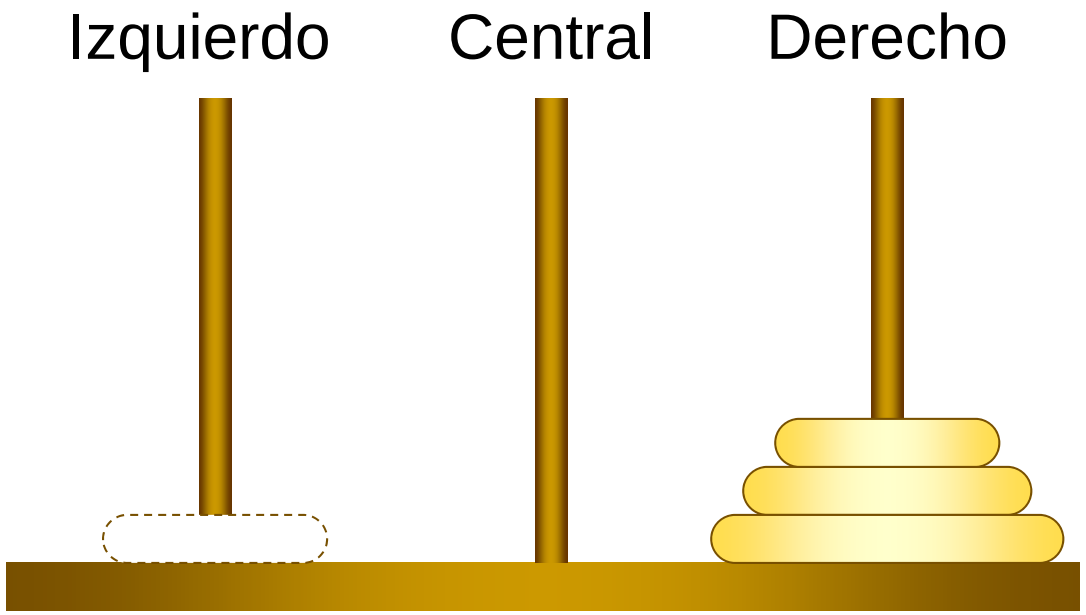
Recursividad

- Ejemplo: 3 discos



Recursividad

- Ejemplo: 3 discos



UNIVERSIDAD DE MÁLAGA

Recursividad

Solución Recursiva

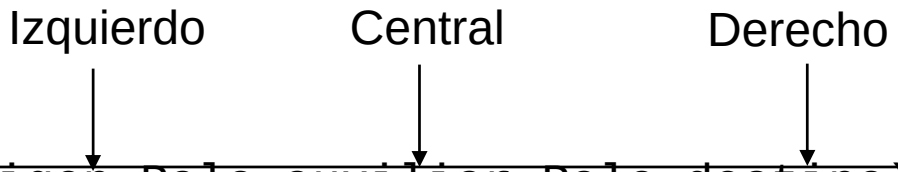
- Si n=1, el problema se resuelve inmediatamente sin más que mover el disco del palo `origen` al `destino`.

- El problema de mover n discos (n>1) desde un palo `origen` (en nuestro caso el izquierdo) a un palo `destino` (en nuestro caso el derecho), utilizando el otro palo como `auxiliar` (en nuestro caso el central) se puede expresar en función del problema de mover n-1 discos, descomponiendo la solución en 3 fases:

Recursividad

1.- Mover los n-1 discos superiores del palo `origen` al palo `auxiliar`, utilizando en este paso el palo `destino` como palo auxiliar.
2.- Mover el disco que queda del palo `origen` al `destino`.
3.- Mover los n-1 discos del palo `auxiliar` al palo `destino`, utilizando el palo `origen` como palo auxiliar.

Recursividad




```
void mueve(unsigned n,Palo origen,Palo auxiliar,Palo destino) {
  if (n == 1) {
    mueveUno(origen,destino);
  } else {
    mueve(n-1,origen,destino,auxiliar);
    mueveUno(origen,destino);
    mueve(n-1,auxiliar,origen,destino);
  }
}
```

Recursividad

Conclusiones de los ejemplos:

1.- Los subprogramas vistos **se invocan a sí mismos** (esto es lo que los convierten en recursivos).

2.- Cada **llamada recursiva** se hace con un parámetro de menor valor que el de la anterior llamada. Así, cada vez se está invocando **a otro problema idéntico pero de menor tamaño**.

3.- Existe un caso especial en el que se actúa de forma diferente, esto es, ya no se utiliza la recursividad. Lo importante es que la forma en la que el tamaño del problema disminuye asegura que se llegará a este caso especial o **caso base**.

UNIVERSIDAD DE MÁLAGA

Recursividad

En general, para determinar si un algoritmo recursivo está bien diseñado debemos plantearnos **tres preguntas**:

1.- ¿Existe una o varias salidas no recursivas o **casos base** del subprograma, y éste funciona correctamente para ellas? ¿Se alcanzan?

2.- ¿Cada llamada resursiva al subprograma se refiere a un **caso más pequeño** del problema original?

3.- Suponiendo que la(s) llamada(s) recursiva(s) funciona(n) correctamente, así como el caso base, ¿funciona correctamente todo el subprograma?

UNIVERSIDAD DE MÁLAGA

Recursividad vs. Iteración

- Ya sabemos que la **recursividad** es una técnica potente de programación para resolver problemas que a menudo produce **soluciones simples y claras**.

- Sin embargo, la recursividad también tiene algunas desventajas, las cuales se enmarcan en el campo de la eficiencia. **Muchas veces un algoritmo iterativo es más eficiente que su correspondiente recursivo**. Existen dos factores que contribuyen a ello:

Recursividad vs. Iteración

1. La sobrecarga asociada con las llamadas a subprogramas.

- Cuando se produce una llamada a un procedimiento o una función se deben establecer las vías de comunicación entre el llamante y el llamado (parámetros) y se deben crear las variables locales del llamado.
- Después, al finalizar la ejecución del llamado, se deben destruir todas esas variables locales y los parámetros utilizados.
- Todo esto conlleva una sobrecarga tanto en tiempo de ejecución como en utilización de memoria.

Recursividad vs. Iteración

1. La sobrecarga asociada con las llamadas a subprogramas.

- **Esta sobrecarga es mayor cuando se utiliza la recursividad**, ya que una simple llamada inicial a un subprograma puede generar un gran número de llamadas recursivas.
- No debemos usar la recursividad por el gusto de usarla, cuando podemos diseñar un algoritmo iterativo igual de sencillo. Por ejemplo, deberíamos usar la función iterativa factorialIter en lugar de su equivalente recursiva factorialRec.

Recursividad vs. Iteración

1. La sobrecarga asociada con las llamadas a subprogramas.

- Aunque bien es cierto que muchos **compiladores** pueden realizar en “algunos casos concretos” **optimizaciones** al traducir un subprograma recursivo de forma que la ejecución realmente es tan eficiente como la del equivalente iterativo.

UNIVERSIDAD DE MÁLAGA

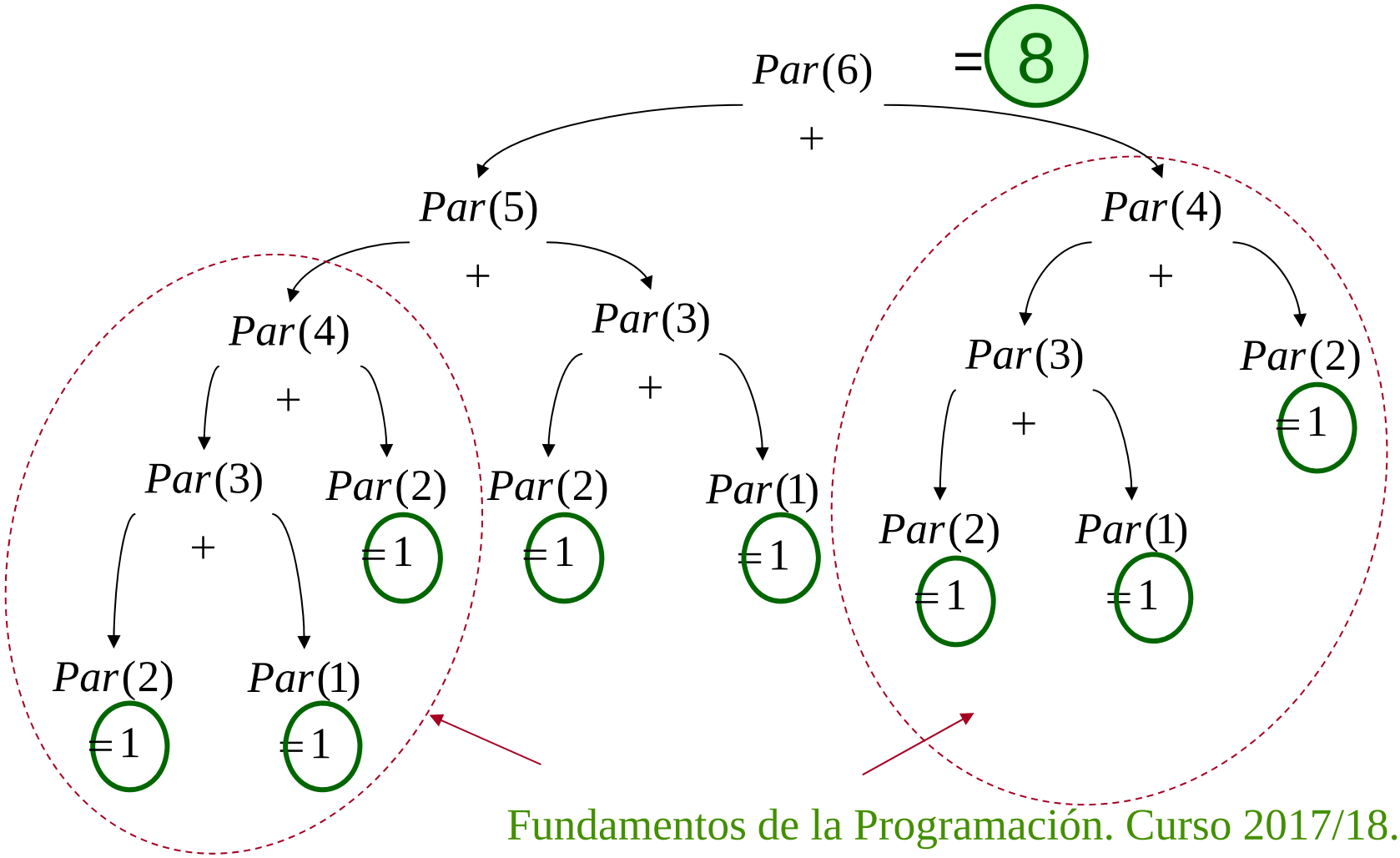
Recursividad vs. Iteración

2. La ineficiencia inherente de algunos algoritmos recursivos.

- Esta ineficiencia es debida al **método empleado** para resolver el problema concreto que se esté abordando.

- Por ejemplo, en nuestra función parejas apreciamos un gran inconveniente: los mismos valores son calculados varias veces. Por ejemplo, para calcular parejas(6), tenemos que calcular parejas(4) dos veces, parejas(3) tres veces, … Mientras más grande sea n, mayor ineficiencia.

Recursividad vs. Iteración



Recursividad vs. Iteración

- Podemos diseñar un algoritmo más eficiente (parejasEfic) que evite este inconveniente, aunque requiere un poco más de esfuerzo y no es tan inmediato.
- En realidad, la función parejasEfic no es recursiva, sino que llama a otra función parejasEficRec que sí lo es.
- Los parámetros de parejasEficRec son:
  - el mes **n** para el que queremos calcular cuántas parejas de conejos hay
  - un contador **cont** que marca el mes que se va intentando en cada paso (hasta llegar a **n**)
  - y los valores de las parejas de conejos (**ant2** y **ant1**) que hay en los dos meses anteriores a **cont**.

Recursividad vs. Iteración

```
unsigned parejasEficRec(unsigned ant2, unsigned ant1, unsigned cont,
                        unsigned n) {
  unsigned res;
  if (cont == n) {
    res = ant2 + ant1;
  } else {
    res = parejasEficRec(ant1,ant2+ant1,cont+1,n);
  }
  return res;
}
unsigned parejasEfic(unsigned n) {
  unsigned res;
  if (n <= 2) {
    res = 1;
  } else {
    res = parejasEficRec(1,1,3,n);
  }
  return res;
}
```

Recursividad vs. Iteración

En cualquier caso, también podemos diseñar un algoritmo iterativo sencillo:

```
unsigned parejasIter(unsigned n) {
  unsigned ant2,ant1,res;
  if (n <= 2) {
    res = 1;
  } else {
    ant2 = 1;
    ant1 = 1;
    for (unsigned cont = 3; cont<=n; cont++) {
      res = ant2 + ant1;
      ant2 = ant1;
      ant1 = res;
    }
  }
  return res;
}
```

Recursividad vs. Iteración

- A pesar de todo esto, **en muchas circunstancias** el uso de la **recursividad** permite a los programadores especificar **soluciones naturales y sencillas** que serían **difíciles** de resolver **mediante técnicas iterativas**.

- En estos casos la **sencillez y naturalidad** de la solución **compensa la posible ineficiencia** en la ejecución.

- Por **ejemplo** el problema de las **Torres de Hanoi**.