Programación II

Unidad 7
Recursión

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Objetivos

- **7.1 Describir el seguimiento de una función que hace una llamada a sí misma.**
- **7.2 Diseñar subprogramas (funciones) recursivas.**
- **7.3 Demostrar la corrección de un algoritmo recursivo mediante inducción.**
- **7.4 Aplicar la técnica de divide y vencerás para la resolución de problemas.**
- **7.5 Comparar la resolución de un mismo problema por iteración o por recursión.**
- **7.6 Eliminar la recursión de un algoritmo recursivo mediante el uso de pilas.**

Introducción (I)

- **Algoritmo recursivo**
  - Algoritmo que “se llama a sí mismo” salvo en algún/os caso/s concreto/s (caso/s base)
- **Ejemplos:**
  - Árboles: inserción/búsqueda en ABdB, recorridos de ABs (preorden, etc.), cálculo de nº nodos, nº hojas, profundidad, copiar un árbol en otro, comprobar si uno es espejo de otro, etc.
  - Funciones matemáticas: factorial, potencia, serie Fibonacci, etc.

Introducción (II)

- **Versión recursiva del cálculo del factorial de n**

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

```
int fact(int n) {
    if (n==1 || n==0)    // caso base
        return 1;
    return n*fact(n-1); // caso general, n > 1
}
```

Algoritmo recursivo: Define el algoritmo en función de un caso más sencillo

- **Versión iterativa del cálculo del factorial de n**

```
int fact(int n) {
    prod=1;
    for(i=n; i>0; i--)
        prod *= i;
    return prod;
}
```

Algoritmo iterativo: repetición explicita de un conjunto de instrucciones hasta que se cumple una condición dada

Algoritmos recursivos: Búsqueda binaria (I)

- **Búsqueda de una clave en una tabla de valores**
  - E.g.: T=

2	**8**	**6**	**5**	**7**	**3**	**9**	**1**	**4**

  buscar la clave k= 6, devolver índice de la clave en el array

- **Solución 1: Búsqueda lineal**

```
ind BLin(tabla T, dimension D, clave k)
    para i desde 0 hasta D-1:
        si T[i] == k:
            devolver i;
        devolver -1; //k no está en T
```

Recorre el array
Es un algoritmo iterativo

  - Ventajas:
    - Fácil de programar
    - No necesita que la tabla esté ordenada
    - Fácil de implementar para Listas Enlazadas
  - Desventaja: Poco eficiente:
    - Caso mejor: T[0] = k → 1 comparación
    - Caso peor: T[N-1] = k → N comparaciones
    - Caso medio: T[N/2 -1] = k → N/2 comparaciones → O(n)

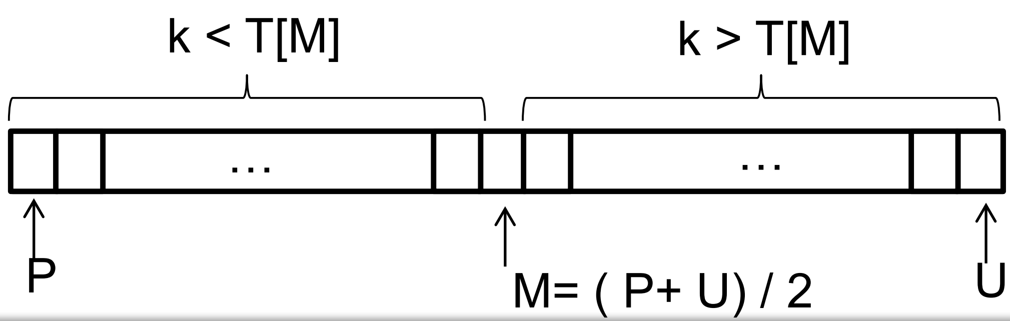
Algoritmos recursivos: Búsqueda binaria (II)

- **Alternativa: Búsqueda binaria**
- **Sobre tabla T <u>ordenada</u>** (requisito)

Clave k, T=

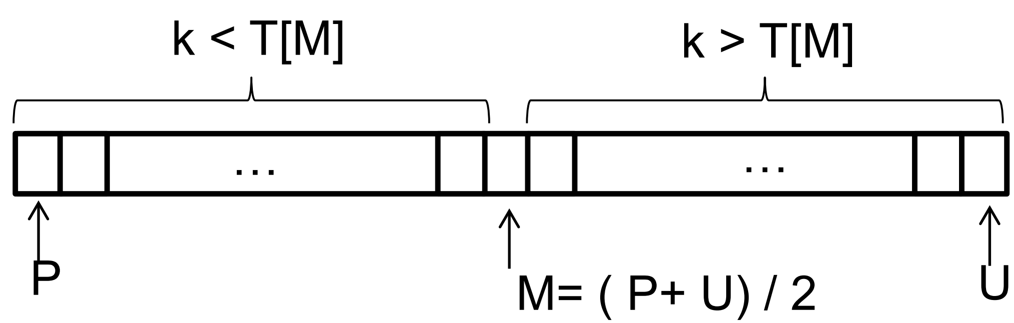
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

- **Idea: División de la tabla en partes más pequeñas**
  1. Comparar con elemento en la mitad de la tabla
     A. Si es mayor, entonces la clave estará en la segunda mitad
     B. Si es menor, entonces la clave estará en la primera mitad
  2. Redefinir los límites de la tabla y volver a buscar (recursión)
     – Definir los límites con dos variables: P (primero) y U (último)



- **PsC**

```
ind BBin(tabla T, ind P, ind U, clave k)
    si P>U:    // caso base
      devolver -1 // llamada errónea ó K no en T
    si no:
      M=(P+U)/2   //División entera (floor)
      si k == T[M]:
          devolver M
      si k < T[M]:
          devolver BBin(T, P, M-1, k) // Primera mitad
      si no:
          devolver BBin(T, M+1, U, k) // Segunda mitad
```



- **Ejemplo ejecución**

Clave 8, T=

1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8

llamada	P	U	M	
1	0	8	4	8 > 5
2	5	8	6	8 > 7
3	7	8	7	8 == 8

```
ind BBin(tabla T, ind P, ind U, clave k)
       si P > U:    // caso base
              dev -1 //Llamada errónea ó K no en T
       si no:
              M= (P+U)/2; //División entera (floor)
              si k == T[M]:
                     dev M
              si k < T[M]:
                     dev BBin(T, P, M-1, k) // Primera mitad
              si no:
                     dev BBin(T, M+1, U, k) // Segunda mitad
```

Algoritmos recursivos: Búsqueda binaria (III)

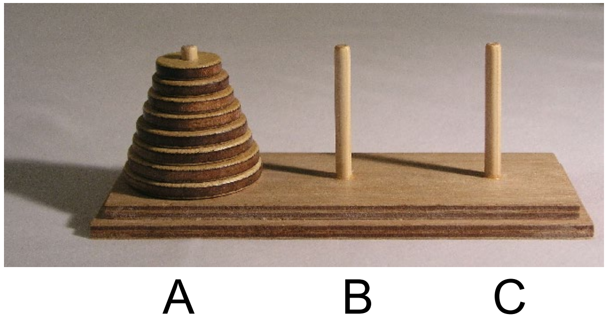
- **Inconvenientes recursión en el ejemplo de búsqueda:**
  - Algoritmo más complicado que BLineal
  - La tabla T tiene que estar ordenada
  - No es aplicable a Listas enlazadas (no hay forma sencilla de encontrar el punto medio).
- **Ventajas**
  - Eficiencia: Si T tiene N elementos, BBin hace a lo sumo $\approx \log_2(N)$ comparaciones, complejidad $\approx O(\log(N))$
    - Si $N=10^5$, BBin hace a lo sumo 17 comparaciones. BLin puede hacer de media $10^5/2 = 5x10^4$
    - Si se duplica tamaño de tabla, BBin hace 1 comparación más y BLin el doble

Algoritmos recursivos: Búsqueda binaria (III)

- **En general, la recursión es útil cuando:**
  - El problema a resolver se descompone fácilmente en subproblemas.
  - Cada subproblema es más fácil de resolver que el original.
  - Las soluciones a los subproblemas permiten construir o recuperar la solución final.

Algoritmos recursivos: Torres de Hanoi (I)

- **Ejemplo típico de algoritmo recursivo**



- **N discos ordenados en tamaño decreciente en un poste A**
- **Dos postes B y C**
- **Se requiere dar la lista de movimientos para pasar los discos de A a B (usando C como auxiliar) de modo que en A, B y C siempre se mantenga la ordenación en tamaños decrecientes**

```
Hanoi(int N, poste A, poste B, poste C)
```

Algoritmos recursivos: Torres de Hanoi (II)

```
Hanoi(int N, poste A, poste B, poste C)
```

- **Solución más sencilla: basarla en casos más simples**
  - Si N == 1, la solución es pasar A → B
  - Si N > 1:
    - Pasar todos los discos, menos el primero, al poste auxiliar, usando B (vacío o con discos ordenados de mayor a menos) como auxiliar
    - Pasar A → B
    - Pasar todos los discos del poste auxiliar a B, usando A (vacío) como poste auxiliar

```
Hanoi(int N, poste A, poste B, poste C)
      si N==1:
            print A → B
      si no:
            hanoi(N-1, A, C, B)
            print A → B
            hanoi(N-1, C, B, A)

H(3, A, B, C) ?
```

Algoritmos recursivos: Torres de Hanoi (III)


```
H(3, A, B, C)
          H(2, A, C, B)
                    H(1, A, B, C)
                              A → B
                    A → C
                    H(1, B, C, A)
                              B → C
          A → B
          H(2, C, B, A)
                    H(1, C, A, B)
                              C → A
                    C → B
                    H(1, A, B, C)
                              A → B
```

Resultado: A → B, A → C, B → C, A → B, C → A, C → B,
A → B

Algoritmos recursivos: Torres de Hanoi (IV)

- **Cuántos movimientos se requieren para solucionar las torres de Hanoi con N piezas?**
  - m(N) = 1 Si N == 1

    = m(N – 1) + 1 + m(N – 1) si N > 1 ➔ 1 + 2m(N – 1)

    m(N)= 1 + 2m(N-1)

    1 + 2(1 + 2m(N-2))

    1 + 2 + 2^2(1 + 2m(N – 3))

    1 + 2 + 2^2 + 2^3(1 + 2m(N – 4))= …=

Se puede demostrar por inducción

$$\sum_{i=0}^{N-2} 2^i + 2^{N-1} m(1) = \sum_{i=0}^{N-1} 2^i = 2^N - 1$$

  - m(64)= 2^{64}-1 ≈ 10^{19} por 1 mov por segundo= 5.84 × 10^{11} años

```
Hanoi(int N, poste A, poste B, poste C)
        si N == 1:
                print A → B
        si no:
                hanoi(N - 1, A, C, B)
                print A → B
                hanoi(N - 1, C, B, A)
```

Algoritmos recursivos: Ventajas/Desventajas

- **Ventajas**
  - Permite programar reduciendo a casos más sencillos → facilita la programación.

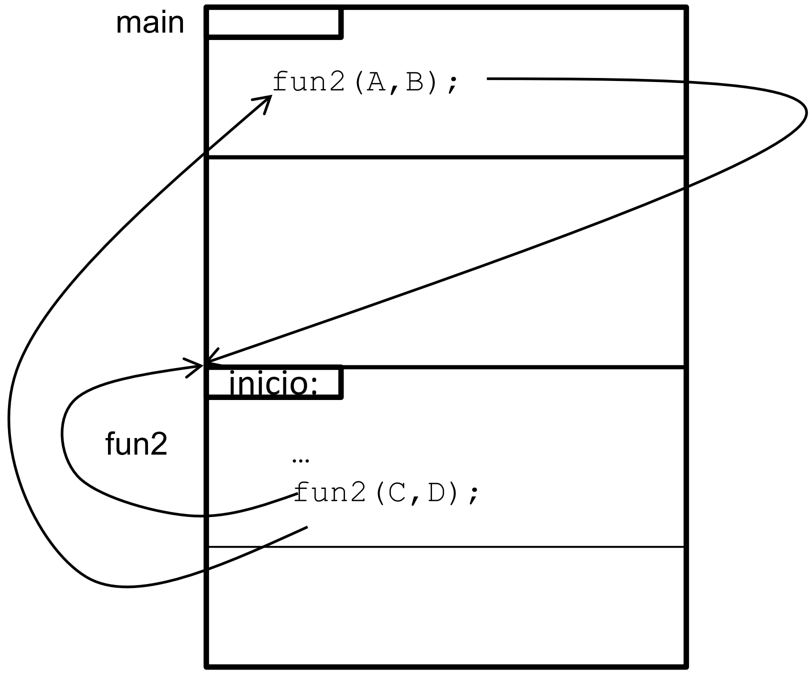
    Ej: Torres de Hanoi, Insertar en ABdB, nº nodos AB, etc…
  - Codificación más sencilla de entender, más elegante
- **Desventajas**
  - A veces es difícil seguir el flujo del programa. Ej: Hanoi.
  - Eficiencia: muchas llamadas recursivas a una función restan eficiencia a un algoritmo
  - Memoria: peligro de desbordar la pila de áreas de datos (AdD) de funciones (1AdD/llamada)
  - Algunos lenguajes no permiten recursión (ej. Ensamblador, Fortran) por no tener llamadas por valor (recuperación de valores al salir de func.).
- **Solución: quitar la recursión a un algoritmo recursivo**

Eliminación de la recursión

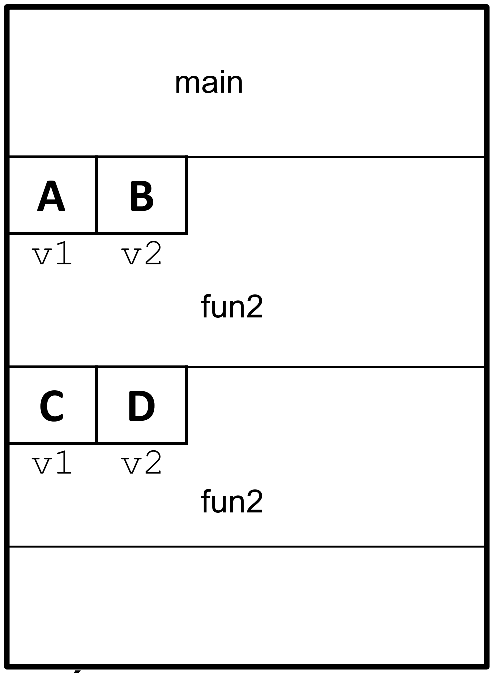
- **Convertir un algoritmo recursivo en un algoritmo iterativo**
- **A veces no es sencillo. Depende del tipo de recursión:**
  - Recursión de cola: sencillo
  - Recursión general: más complejo
- **Eliminar la recursión ➔ Simular el mecanismo de llamadas y retornos de funciones**

Recursión: Llamadas y pila de AdD (I)

```
main(){
   type A, B; status s;
   s = fun2(A,B);
}
status fun2(type v1, type v2){
   …
   return fun2(C,D);
}
```

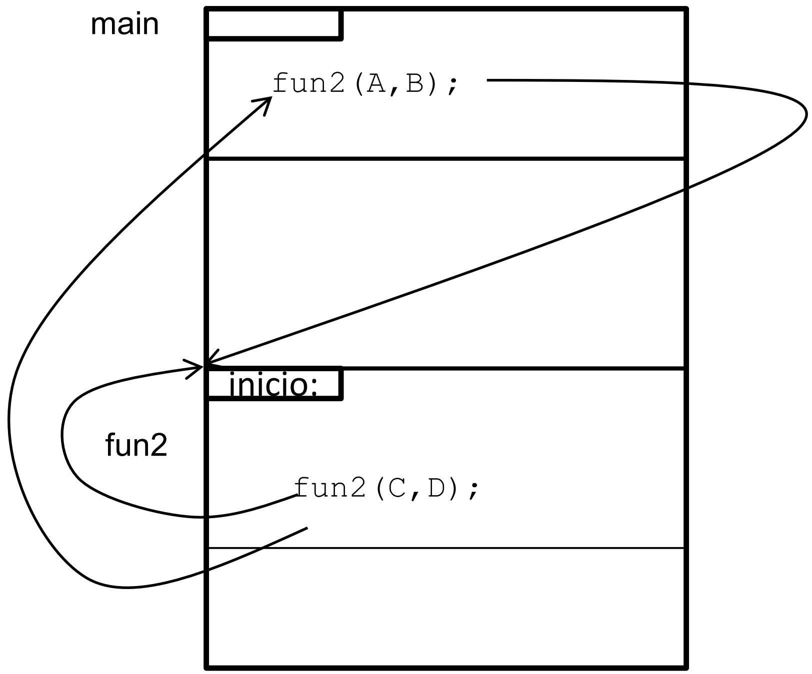


Código

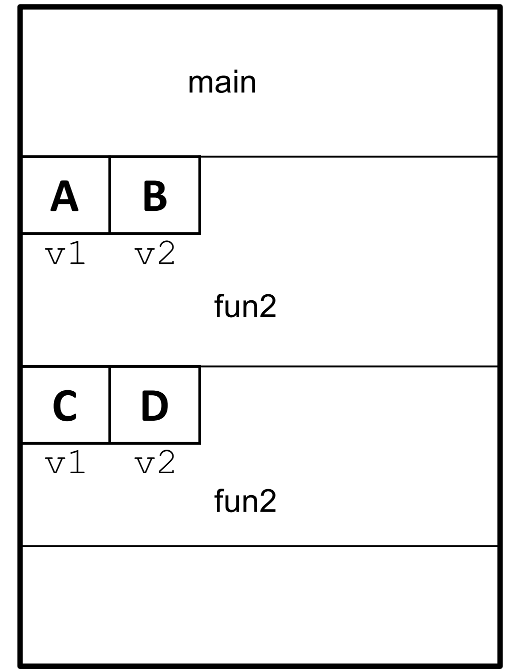


Áreas de datos (AdD)

Simulación de recursión



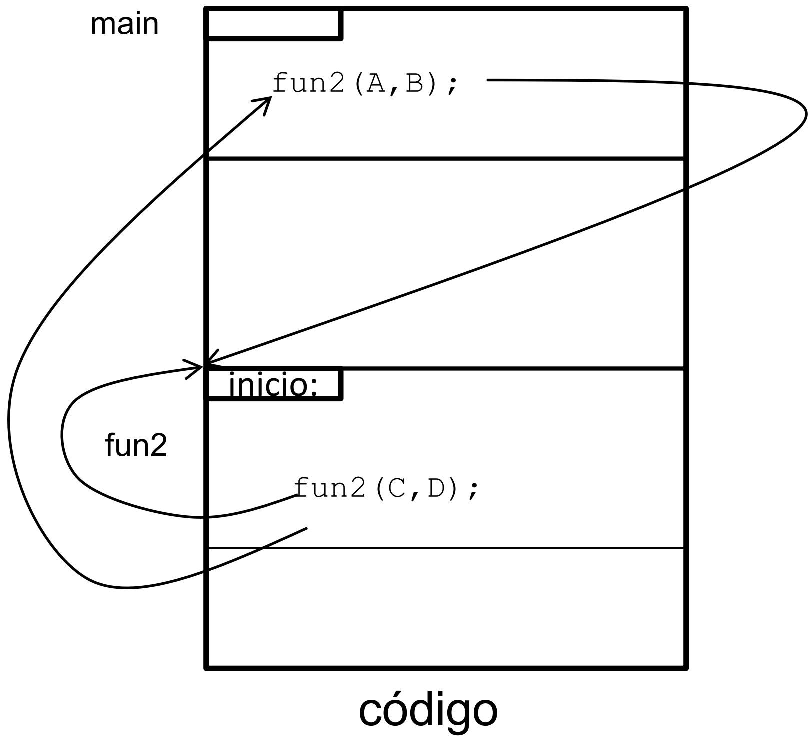
código

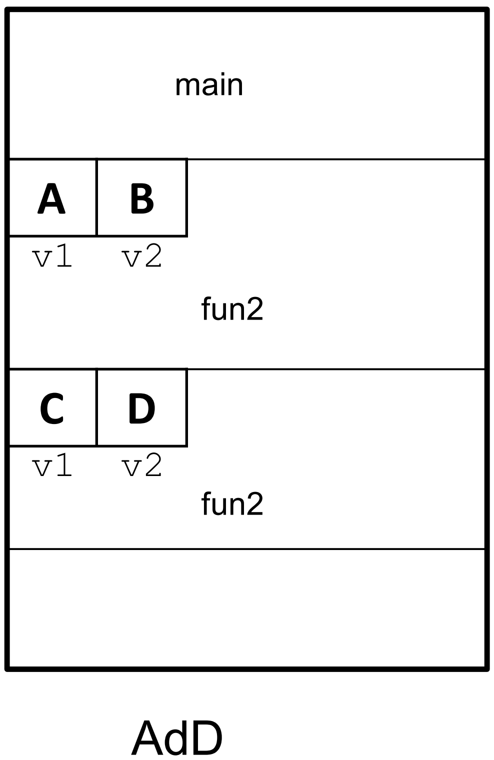


AdD

- **Llamada fun2 desde main**
  1. Crear AdD de fun2
  2. Guardar, justo antes de llamar a fun2, los valores de las variables del main y el punto de retorno (para saber por dónde continuar al volver de fun2)
  3. Copiar los valores de los argumentos de la llamada a fun2 (A y B) en el área de datos de fun2, en la memoria correspondiente a sus parámetros (v1 y v2)
  4. Ir al inicio del código de fun2 para ejecutarlo ya con los valores de los argumentos (A y B en v1 y v2)

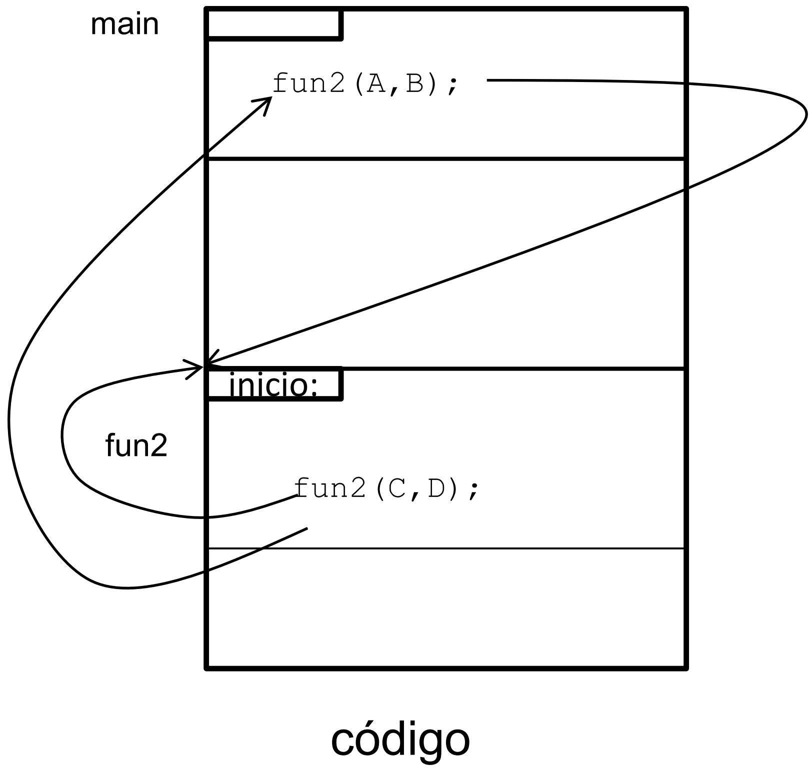
Simulación de recursión

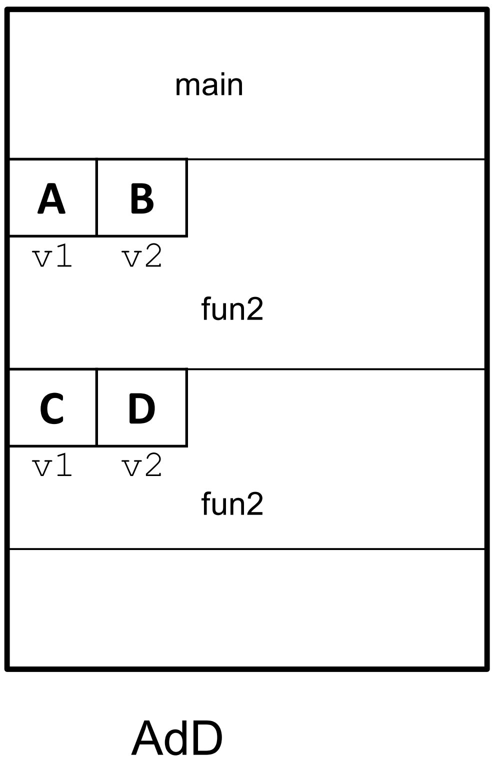




- **Llamada fun2 desde fun2**
  1. Crear una nueva AdD para la 2ª llamada a fun2.
  2. Guardar, justo antes de llamar de nuevo a fun2, los valores de las variables v1 y v2 y el punto de retorno (para saber por dónde continuar al volver de la 2ª llamada).
  3. Copiar los valores de los argumentos de la 2ª llamada (C y D) en el área de datos de fun2, en la memoria correspondiente a sus parámetros (v1 y v2).
  4. Ir al inicio del código de fun2 para ejecutarlo ya con los nuevos valores de los argumentos (C y D en v1 y v2).

Simulación de recursión



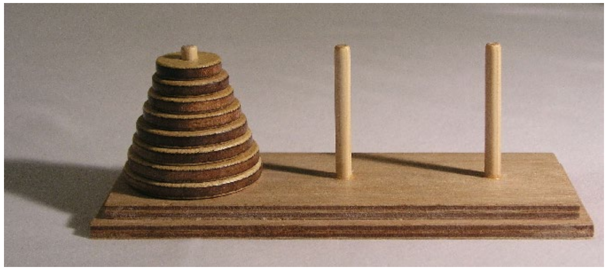


- **Retorno de fun2 a main/fun2**
  1. Almacenar el valor de retorno (si lo hubiera) en el AdD de la función que realizó la llamada (main/fun2)
  2. Recuperar los valores de las variables del AdD de dicha función
  3. Continuar la ejecución por el punto de retorno guardado anteriormente

Simulación de recursión

```
Hanoi(int N, poste A, poste B, poste C)
        inicio:
        si N == 1:
                print A → B
        si no:
                hanoi(N-1, A, C, B)
                print A → B
                hanoi(N-1, C, B, A)
        return;
```

```
Ej:
Hanoi(64, 1, 2, 3)
        inicio:
        si N == 1:
                print 1 → 2
        si no:
                hanoi(63, 1, 3, 2)
                print 1 → 2
                hanoi(63, 3, 1, 2)
        return;
```



Simulación de recursión: Mecanismo general

Uso de Pila para guardar y recuperar los datos de cada llamada:

1: Simulación de llamada

1.1. **Guardar valores de argumentos de entrada y variables locales** en pila de elementos (push), a modo de "Área de Datos".
Si hay varias llamadas recursivas a la misma función, guardar también la **identificación** del nº de la llamada (1, 2,…) para saber punto de retorno.

1.2. **Reasignar argumentos**

En el ejemplo, para simular la 1ª llamada: N= 63, A= 1, B= 3, C= 2

1.3. **Ir al inicio** de la función

2: Simulación de retorno

2.1. **Recuperar** antiguos valores de las variables locales (pop) e id de la llamada (si lo había, marcará el punto desde donde continuar).
Almacenar valor de retorno (si lo hay)

2.2. **Seguir** con la ejecución por el punto de retorno correspondiente.

Simulación de recursión: Recursión de Cola (RdC)

- **La <u>RdC</u> es una llamada recursiva tras la cual no se realiza ninguna operación en la función (sólo retornar)**

```
Hanoi(int N, poste A, poste B, poste C)
    inicio:
        si N == 1:
            print A → B
        si no:
            hanoi(N-1, A, C, B)
            print A → B
            hanoi(N-1, C, B, A)
```

¿Recursión? ¿Tipo(s)?

Recursión no de cola

Recursión de cola

- **En RdC no se opera sobre las variables locales después de la llamada recursiva**
  - Por tanto, no es necesario ni guardar sus valores (push) ni recuperarlos después de la llamada (pop)
- **RdC es fácil de quitar**

Simulación de recursión: Recursión de Cola

- **Simulación de llamadas en RdC**
  1.1.**NO** hace falta guardar estado de variables (porque no queda nada por hacer a la vuelta)
  1.2. Reasignar argumentos
  1.3. Ir al inicio de la función (de forma mecanica: goto)
- **Simulación de retornos en RdC (NO hace falta)**
  2.1. **NO** es necesario recuperar valor variables
  2.2. **NO** hay instrucciones a ejecutar después, salvo retornar

```
Hanoi(int N, poste A, poste B, poste C)
    inicio:
        si N == 1:
                print A → B
        si no:
                hanoi(N-1, A, C, B)
                print A → B
                N--; T=C; C=A; A=T;    //Reasignar args
                goto inicio;           //Ir a inicio
```

```
Hanoi(int N, poste A, poste B, poste C)
    si N == 1:
            print A → B
    si no:
            hanoi(N-1, A, C, B)
            print A → B
            hanoi(N-1, C, B, A)//simular
```

Simulación de recursión

Para eliminar recursión en general:

1. Simulación de llamada

1.1. Guardar argumentos entrada y variables locales (e id de llamada si hay varias llamadas recursivas)

1.2. Reasignar argumentos

1.3. Ir al inicio de la función

2. Simulación de retorno

2.1. Recuperar antiguos valores de las variables locales (pop) y almacenar retorno (si lo hay)

2.2. Seguir con la ejecución por el punto de retorno

Para eliminar recursión de cola (RdC):

- No hace falta 1.1, porque no hace faltar recuperar situación
- No hace falta 2.1-2.3, porque no hay nada que hacer tras el retorno

→ No hace falta pila

- **Basta con:**

  1.2. **Reasignar argumentos**

  1.3. **Ir al inicio de la función**

- **Eliminación mecánica: Usando goto, fácil.**

Simulación de recursión: Recursión de Cola

- **Desventajas uso de goto**
  - Rompe el flujo del programa, no es propio de programación estructurada
  - Algunos lenguajes no lo tienen
- **Alternativa: Versión estructurada → observar el código y quitar el goto**
  - Normalmente haciendo uso de una instrucción iterativa (e.g. while, for)
  - El caso base se usa para la condición del while

```
Hanoi(int N, poste A, poste B, poste C);
```

```
inicio:
  si N == 1:
        print A → B
  si no:
        hanoi(N-1, A, C, B)
        print A → B
        N--; T=C; C=A; A=T;
        goto inicio;
```

Versión estructurada:

```
mientras N > 1:
     hanoi(N-1, A, C, B)
     print A → B
     N--; T=C; C=A; A=T;
print A → B
```

Resumen completo de eliminación de RdC

- **Eliminar RdC = seguir los siguientes pasos:**
  1. Eliminación mecánica (fácil):
     - Reasignar valores de argumentos de entrada
     - Ir con goto al inicio de la función
  2. Pasar a código estructurado
     - Sustituir goto por bucle, usando el caso base de la recursión como condición de parada del bucle.
     - Se puede mejorar la versión de forma manual, basándose en la experiencia con muchos ejemplos.

Eliminar recursión de imprimeOrden

```
Status imprimeOrden(Lista *pl) {
    if (pl == NULL) return Error;
    return imprimeOrden_rec (pl->first);
}

Status imprimeOrden_rec (Nodo *pn) {
    if (pn == NULL) return OK;
    if (elemento_imprimir(pn->info) == -1)
            return Error;
    return imprimeOrden_rec (pn->next);
}
```

¿Recursión? ¿Tipo(s)?
¿Eliminación?

Eliminar recursión de imprimeOrden

```
Status imprimeOrden_rec (Nodo *pn) {
    if (pn == NULL) return OK;
    if (elemento_imprimir(pn->info) == -1)
        return Error;
    return imprimeOrden_rec (pn->next);
}
```

⟸ Eliminación RdC mecánica

```
Status imprimeOrden_norec (Nodo *pn) {
    inicio:
        if (pn == NULL) return OK;
        if (elemento_imprimir(pn->info) == -1)
            return Error;
        pn = pn->next;
        goto inicio;
}
```

Eliminar recursión de imprimeOrden

```
Status imprimeOrden_norec (Nodo *pn) {
    inicio:
        if (pn == NULL) return OK;
        if (elemento_imprimir(pn->info) == -1)
            return Error;
        pn = pn->next;
        goto inicio;
}
```

Pasar a código estructurado

```
Status imprimeOrden_norec (Nodo *pn) {
    while (pn != NULL) {
        if (elemento_imprimir(pn->info) == -1)
                return Error;
        pn = pn->next;
    }
    return OK;
}
```

Simulación de recursión: Ejemplo RdC, Bbin (I)

```
ind BBin(tabla T, ind P, ind U, clave k)
        si P > U:    // caso base
                devolver -1 //Llamada errónea ó K no en T
        si no:
                M = (P+U)/2  // División entera (floor)
                si k = T[M]:
                        devolver M
                else si k < T[M]
Recursión de cola       devolver BBin(T, P, M-1, k) // Primera mitad
                else
Recursión de cola       devolver BBin(T, M+1, U, k) // Segunda mitad
```

Ejercicio: identificar recursión y tipos. Eliminar la recursión

1º Mecánicamente (sirve código no estructurado)

2º Convertir en código estructurado

Simulación de recursión: Ejemplo RdC, Bbin (II)

- **Versión automática (con goto)**

```
ind BBin(tabla T, ind P, ind U, clave k)
    inicio:
        si P > U:                // caso base
                devolver -1    // Llamada errónea ó K no en T
        si no:
                M=(P+U)/2        // División entera (floor)
                si k = T[M]
                        devolver M     //Encontrado
                si k < T[M]:
                        U = M-1
                        goto inicio
                si no:
                        P = M+1
                        goto inicio
```

Simulación de recursión: Ejemplo RdC, Bbin (II)

- **Versión estructurada (sin goto)**

```
ind BBin(tabla T, ind P, ind U, clave k)

        mientras U >= P:     // caso base era P < U
            M = (P + U)/2    // División entera (floor)
            si k = T[M]:
                devolver M       // encontrado
            si k < T[M]:
                U = M-1
            si no:
                P = M+1

        devolver -1    // No encontrado
```

Ejercicio E5: 6

Dar el PsC de una función definida de la siguiente manera:

a(m,n) = n + 1 si m == 0
a(m,n) = a(m-1, 1) si m != 0, n == 0
a(m,n) = a(m-1, a(m, n-1)) si m != 0, n != 0

```
int  a(m, n)
        si m == 0
                devolver n + 1
        si no, si n == 0
                devolver a(m-1, 1)
        si no devolver a(m-1, a(m, n-1))
```

Identificar recursión y quitar recursión de cola
(se puede retocar el código para facilitar la eliminación)

Ejercicio E5: 6 Sol (II)

- **Retocar el código**

```
int  a(m, n)
       si m==0:
              devolver n+1
       si no, si n==0:
              devolver a(m-1, 1)
       si no:
              devolver a(m-1, a(m, n-1))
```

```
int  a(m, n)
      si m==0:
             devolver n+1
      si no, si n==0:
             devolver a(m-1, 1)    ➡ RdC
      si no:
             n= a(m, n-1)
             devolver a(m-1,n)    ➡ RdC
```

Ejercicio E5: 6 Sol (II)

- **Eliminación mecánica (goto's)**

```
int  a(m, n)
   inicio:
      si m==0
             devolver n+1
      else si n==0
             m = m-1
             n = 1
             goto inicio
      else
             n = a(m, n-1)
             m = m-1
             goto inicio
```

Ejercicio E5: 6 Sol (II)

- **Versión estructurada (sin goto's)**

```
int  a(m, n)
     inicio:
     si m==0
          devolver n+1
     else si n==0
          m = m-1
          n = 1
          goto inicio
     else
          n = a(m, n-1)
          m = m-1
          goto inicio
```

```
int  a(m, n)
   while m > 0
       si n == 0
           m = m-1
           n = 1
       else
           n= a(m, n-1)
           m= m-1
   devolver n+1
```

Ejercicios Recursión

1.a) Identificar las llamadas recursivas presentes en las funciones de recorridos de árboles PreOrden, PostOrden y SimOrden, e indicar de qué tipo son cada una de ellas (de cola ó no de cola)

b) Eliminar la recursión de cola en los algoritmos anteriores en 2 pasos: primero de forma mecánica y después pasando a una versión con código estructurado.

2. a) Identificar las llamadas recursivas de la función que calcula el número de nodos de un árbol binario e indicar de qué tipo es cada una de las llamadas.

b) En caso de que alguna sea de cola, eliminarla.

Ejercicios Recursión

3. El cálculo del máximo común divisor entre dos enteros puede calcular de la siguiente forma

mcd(x,y)= y si (y <= x) y (x mod y == 0)

mcd(x,y) = mcd(y,x) si (x < y)

mcd(x,y) = mcd(y, x mod y) en cualquier otro caso

a) Dar el código C de una función recursiva que implemente dicho cálculo

int mcd (int x, int y)

b) Marcar en el código anterior las llamadas recursivas e indicar, para cada una de ellas, de qué tipo es
(de cola ó no de cola)

c) En caso de existir recursión de cola, eliminarla.

Ejercicios Recursión

4. Dado el siguiente PSC de recorrido de orden medio de un Árbol Binario:

```
Status oPrevio (arbol T)
   visitar (T)
   oPrevio (izq(T))
   oPrevio (der(T))
```

Identificar los distintos tipos de recursión y eliminar aquella(s) llamada(s) recursiva(s) que sean de cola de forma mecánica (versión no estructurada y versión estructurada).

Ejemplos de eliminación de recursión general

Eliminación de recursión general (I)

- **Pasos eliminación:**

Previos:

1. Retocar el código si hace falta (ver ejemplo de factorial)
2. Definir elemento de pila E = estructura que guarda variables locales, argumentos de entrada y punto de retorno (si hay varios posibles)
3. Definir variable R para recoger y devolver el valor de retorno de la función (si lo hay)

Eliminar mecánicamente:

1. Leer y "traducir" código recursivo = simular llamada (1.1-1.3) y retorno (2.1, 2.2) de forma mecánica, con gotos
2. Revisar código, eliminar código redundante (ej: Hanoi)

Pasar a versión estructurada:

1. Eliminación de goto's (+/- fácil, no mecánico)
2. (no siempre) Interpretar el código resultante ➔ aclarar qué hace y, si se ve claro, modificar función para simplificarla

Eliminación de recursión general: factorial (I)


```
int fact(int n)
    if (n == 1) //caso base
        return 1;
    else
        return n * fact(n - 1)
```

```
int fact(int n)

if (n == 1) //caso base
        return 1;
else   //caso general
        R= fact(n - 1)
        R= R * N
        dev R
```

inicio ➔ A

retorno ➔ B

A. Simulación de llamada

1. Guardar argumentos entrada y variables locales (e id de llamada si varias llamadas) en elemento de pila E
2. Reasignar argumentos
3. Ir al inicio de la función

B. Simulación de retorno

1. Recuperar antiguos valores de las variables locales (pop) y almacenar retorno (si lo hay)
2. Seguir con la ejecución por el punto de retorno

Ejemplo eliminación de rec. general: factorial (I)

- **División en "inicio" y "retorno"**
  - **inicio: código hasta la primera llamada recursiva**
  - **retorno: código hasta retorno recursivo**

inicio ➔ A

```
int fact(int n)
        if (n == 1) //caso base
                return 1;
        else    //caso general
                R= fact(n - 1)
                R= R * N
                dev R
```

retorno ➔ B

Ejemplo eliminación de rec. general: factorial (II)

- Eliminación mecánica

```
int fact(int N)
        pilaIni(p)
        ini:
                si N == 1
                        R= 1; goto ret;
                si no:
                        E.N= N;          //1.1: Guardar args y vars locales
                        push(E,P);
 1                      N--;             //1.2. Reajustar argumentos
                        goto ini;        //1.3. Ir al inicio
        ret:
                si pilaVacia(P) == F:
                        //R=R;
                        pop(E, P);       // 2.1. Recuperar variables locales
 2                      N= E.N;
                        R= R * N;        // 2.2. Continuar ejecución
                        goto ret;
                devolver R;
```

```
int fact(int n)
        si(n == 1) //caso base
                return 1;
        si no  //caso general
                R= fact(n - 1)
                R= R * N
                dev R
```

Elemento de pila: E
E solo tiene N
→ guardar y recuperar N directamente

Eliminación de recursión general: factorial (III)

- Eliminación mecánica

```
int fact(int N)
      pilaIni(p)
      ini:
                  si N == 1
                          R= 1; goto ret;
                  si no:
                          ~~E.N= N;~~
   1                      push(N,P);   //1.1: Guardar args y vars locales
                          N--;         //1.2. Reajustar argumentos
                          goto ini;    //1.3. Ir al inicio
      ret:
                  si pilaVacia(P) == F:
                          pop(N, P);   // 2.1. Recuperar variables locales
                          ~~N= E.N;~~
   2                      R= R * N;    // 2.2. Continuar ejecución
                          goto ret;

                  devolver R;
```

```
int fact(int n)
        si(n == 1) //caso base
                return 1;
        si no  //caso general
                R= fact(n - 1)
                R= R * N
                dev R
```

Eliminación de recursión general: factorial (IV)

- Eliminación mecánica resulta en:

```
int fact(int N)
    pilaIni(p)
    ini:
            si N==1
                    R=1; goto ret;
            si no:
                    push(N,P);
                    N--;
                    goto ini;
    ret:
            si pilaVacia(P) == F:
                    pop(N, P);
                    R= R * N;
                    goto ret;
            devolver R;
```

- Version estructurada (sin goto's)

```
int fact(int n)
        pilaIni(p);

        mientras N > 1:
                push(N,P);
                N--;
        R= 1;


        mientras pilaVacia(P) == F:
                pop(N, P);
                R= R * N;

        devolver R;
```

Eliminación de recursión general: Hanoi(I)

```
Hanoi(int N, poste A, poste B, poste C)
      si N = 1
            print A → B; return
      else
            hanoi(N-1, A, C, B)
            print A → B
            hanoi(N-1, C, B, A)
            return
```

parte A, bajo ini

parte B, bajo ret

- **En elemento de pila**
  - E= [N, A, B, C, ret]
    - N, A, B, C ➔ argumentos de la función
    - ret == control de punto de retorno
- **No hay valor de retorno → no es necesaria variable R**

Eliminación de recursión general: Hanoi(II)

- Eliminación mecánica

```
Hanoi(int N, poste A, poste B, poste C)
      pilaIni(p)
      ini:
             si N = 1
                     print A → B; goto ret
             else
Guardar variables locales  E.N= N; E.A= A; E.B= B; E.C= C; E.ret= 1
                           push(E,P)
Reasignar argumentos       N--; aux= C; C= B; B= aux
                           goto ini
      ret:
             si pilaVacia(P) = F:
                     pop(E,P)
                     N= E.N; A= E.A; B= E.B; C= E.C
                     si E.ret == 1: // Primera llamada
                              A → B
                              E.N= N; E.A= A; E.B= B; E.C= C; E.ret= 2
                              push(E,P)
                              N --; aux = C; C = A; A = aux
                              goto ini
                     si E.ret == 2:
                              goto ret
```

```
Hanoi(int N, A, B, C)
  si N == 1
    print A → B;
  else
    hanoi(N-1, A, C, B)
    print A → B
    hanoi(N-1, C, B, A)
  return
```

Eliminación de recursión general: Hanoi(III)

- Eliminación código redundante

```
Hanoi(int N, poste A, poste B, poste C)
      pilaIni(p)
      ini:
            si N = 1
                  print A → B; goto ret
            else
                  E.N= N; E.A= A; E.B= B; E.C= C; E.ret= 1
                  push(E,P)
                  N--; aux= B; C= B; B= aux; C= E.B; B= E.C
                  goto ini
      ret:
            si pilaVacia(P) = F
                  pop(E,P)
                  N= E.N; A= E.A; B= E.B; C= E.C
                  si E.ret = 1 //Primera llamada
                        A → B  E.A → E.B;  //imprime
                        E.N= N; E.A= A; E.B= B; E.C= C; E.ret= 2
                        push(E,P)
                        N --; A=E.C; C=E.A
                        goto ini
                  else si E.ret == 2:
                        goto ret
```

Eliminación de recursión general: Hanoi(IV)

- Eliminación goto’s fáciles

```
Hanoi(int N, poste A, poste B, poste C)
      pilaIni(p)
      ini:
            mientras N > 1
                  E.N= N; E.A= A; E.B= B; E.C= C; E.r= 1
                  push(E,P)
                  N--; C= E.B; B= E.C
            A → B
            mientras pilaVacia(P) = F
                  pop(E,P)
                  si E.r = 1 //Primera llamada
                        E.A → E.B
                        E.r= 2;
                        push(E,P);
                        N --; C= E.A; A= E.C
                        goto ini ¿¿CÓMO QUITAR ESTE??

                  //si E.r = 2:
                  //      goto ret  ya lo hace, pq sigue el bucle
```

Eliminación de goto's cruzados (I)

- **Situación de goto's cruzados**

```
ini:

          mientras A:
                    …
                    …
          mientras B:
                    …
                    …
                    si C:
                              …
                              goto ini
```

- **Objetivo, dejar de ejecutar el bucle B y volver al inicio**
  - Cuando el bucle B termine, terminar toda la ejecución
  - Solución: usar un flag y un bucle que englobe a los dos bucles anteriores

Eliminación de goto's cruzados (II)

```
ini:
    mientras A:
        …
        …
    mientras B:
        …
        …
        si C:
            …
            goto ini
```

```
flag= 1
mientras flag = 1:

    mientras A:
        …
        …

    flag= 0
    mientras B y flag == 0:
        …
        …
        si C:
            …
            flag= 1  //sale de B y
                     //vuelve arriba
```

Eliminación de recursión general: Hanoi(V)

- Eliminación goto cruzado

```
Hanoi(int N, poste A, poste B, poste C)
       pilaIni(p)
       flag= 1
       mientras flag = 1:

              mientras N > 1:
                     E.N= N; E.A= A; E.B= B; E.C= C; E.r= 1
                     push(E,P)
                     N--; t= C; C= B; B= T

              A → B

              flag= 0
              mientras pilaVacia(P) = F y flag = 0:
                     pop(E,P)
                     si E.r == 1 //Primera llamada
                            E.A → E.B
                            push(E,P)
                            N --; C= E.A; A= E.C
                            flag= 1
```