TEMA 3: La iteración y sus formas

3.1 La necesidad de Iterar

3.2 Composiciones iterativas

- Composición MIENTRAS
- Composición REPETIR-HASTA
- Composición ITERAR-DETENER
- Equivalencias
- Composición RECORRIENDO

3.3 Ejemplos de problemas iterativos

3.4 Esquema general de una iteración

- El problema de la parada

3.1 La necesidad de iterar

- **Problema:** se pretende calcular el valor medio de las calificaciones obtenidas en un examen por un grupo de alumnos. El número de alumnos no puede ser fijado de antemano. Se proporcionan las notas de los alumnos, para terminar se introduce como nota un número negativo.
- Léxico del Algoritmo

```
nota : Real;        // nota leída
s : Real;           // suma de notas acumulada
n : Entero;         // número de alumnos acumulado
```

3.1 La necesidad de iterar

- **Algoritmo**

```
s ← 0;
n ← 0;
Leer (nota);
SI nota  ≥ 0 ENTONCES
    s ← s + nota;
    n ← n + 1;
    Leer (nota);
    SI nota  ≥ 0 ENTONCES
        s ← s + nota;
        n ← n + 1;
        Leer (nota);
```

{ se repite hasta que v < 0 }

3.2 Composiciones iterativas

- Una iteración siempre debe tener un punto (al menos) de parada.
- Cualquier iteración se compone de tres partes de sentencias fundamentales:
  - Inicialización: tratamientos previos a la iteración.
  - Cuerpo: sentencias que se repiten.
  - Tratamientos de finalización: sentencias que se ejecutan cuando la iteración se detiene.
- Composiciones iterativas:
  - MIENTRAS
  - REPETIR-HASTA
  - ITERAR-DETENER
  - RECORRIENDO

3.2 Composiciones iterativas: Composición MIENTRAS

- Es una composición con comprobación de parada al principio
- **Sintaxis**: **MIENTRAS** <condición > **HACER** <s> **FIN_MIENTRAS**

donde <condión> es una expresión booleana y <s> una secuencia de instrucciones

- **Semántica:** Ejecutar <s> mientras <condición> sea verdad
- **Esquema General**

$A_{inicial}$ — Acciones de inicialización

MIENTRAS cond **HACER**

$\{e_k\}$

A_{cuerpo} — Tratamiento elemental repetido

$\{e_{k+1}\}$

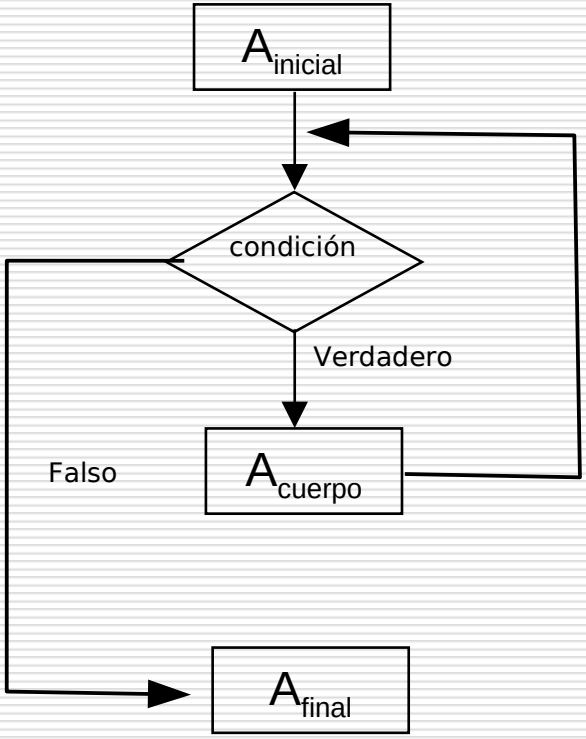
FIN_MIENTRAS;

$\{e_n\}$

A_{final} — Acciones de finalización

3.1 Composiciones iterativas: Composición MIENTRAS

Diagrama de flujo de la composición **MIENTRAS**:



3.2 Composiciones iterativas: Composición MIENTRAS

- Con esta composición ya podemos resolver el problema inicialmente planteado:

```
LÉXICO
  nota :  Real;          // nota leída de la entrada
  suma : Real;           // lleva cuenta de la suma total de las notas
  numElem : Entero; // lleva cuenta del número de notas leídas
ALGORITMO
    suma ← 0; numElem ← 0; Leer (nota);
    MIENTRAS nota ≥ 0 HACER
          suma ← suma + nota; numElem ← numElem + 1;
          Leer (nota)
    FIN_MIENTRAS;
    SI numElem > 0 ENTONCES
        Escribir ("Valor medio de las notas: ", suma/numElem)
    SI_NO Escribir ("No hay notas que promediar")
    FIN_SI
FIN.
```

3.2 Composiciones iterativas: Composiciones REPETIR e ITERAR

S_1

REPETIR

S_2

HASTA condición;

S_3

- Iteración con parada al final
- S_2 siempre se ejecuta al menos una vez
- La *condición* es de parada, no de continuación como en el MIENTRAS

S_1

ITERAR

S_2

DETENER condición

S_3

FIN_ITERAR;

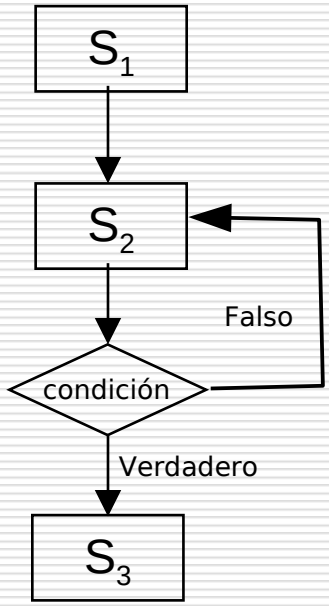
S_4

- Iteración con parada en el medio
- S_2 siempre se ejecuta al menos una vez, S_3 puede que no
- La *condición* es de parada, como en el REPETIR, no de continuación como en el MIENTRAS

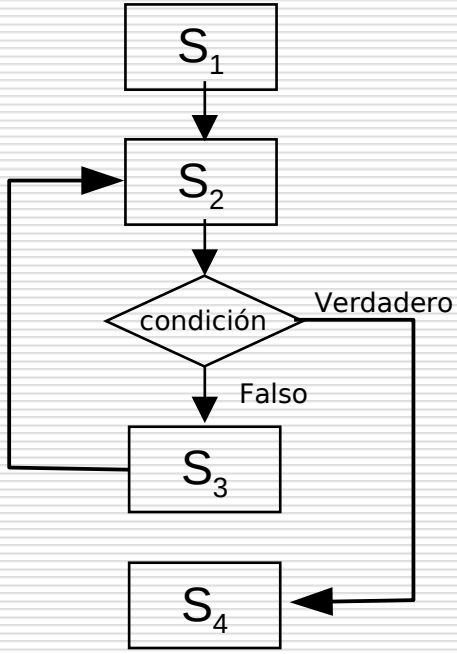
3.2 Composiciones iterativas: Composiciones REPETIR e ITERAR

- Diagramas de flujo de estas iteraciones:

REPETIR:

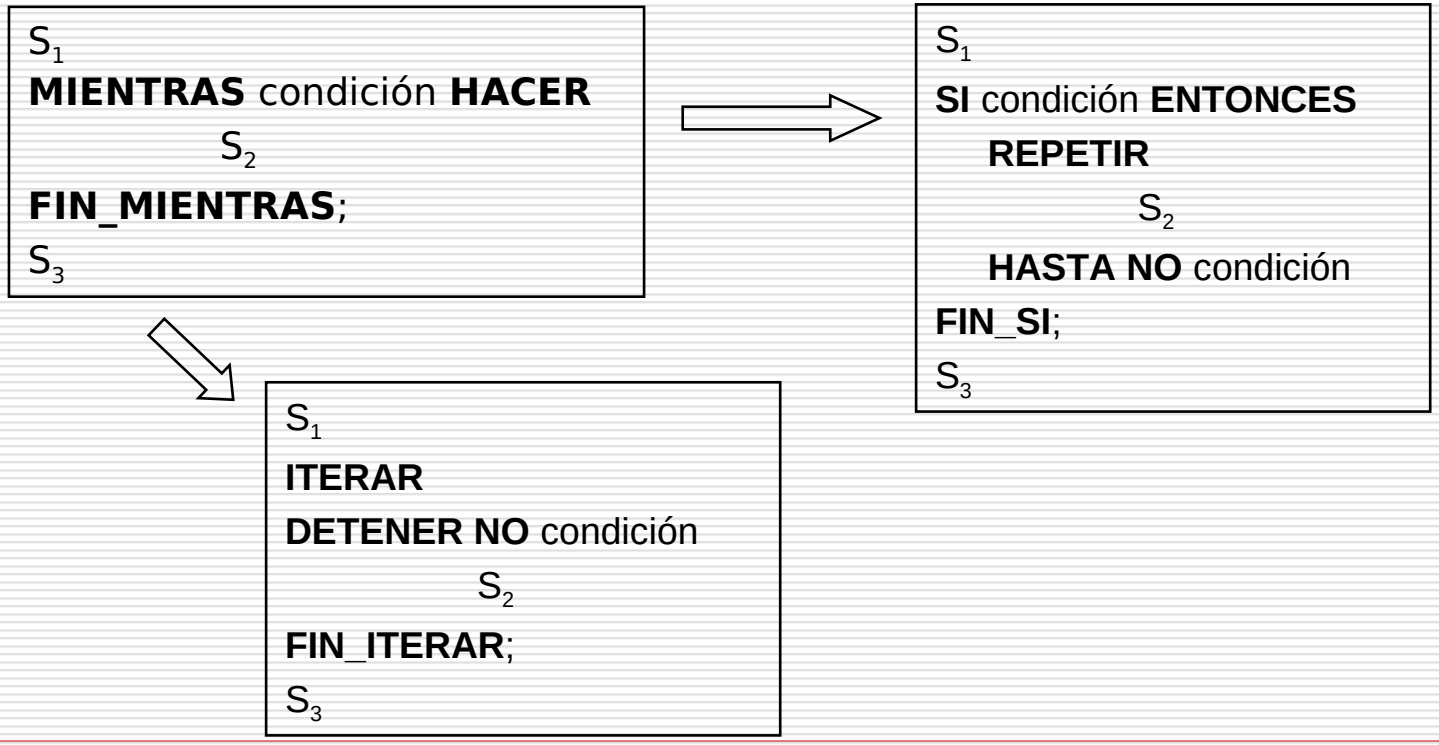


ITERAR:



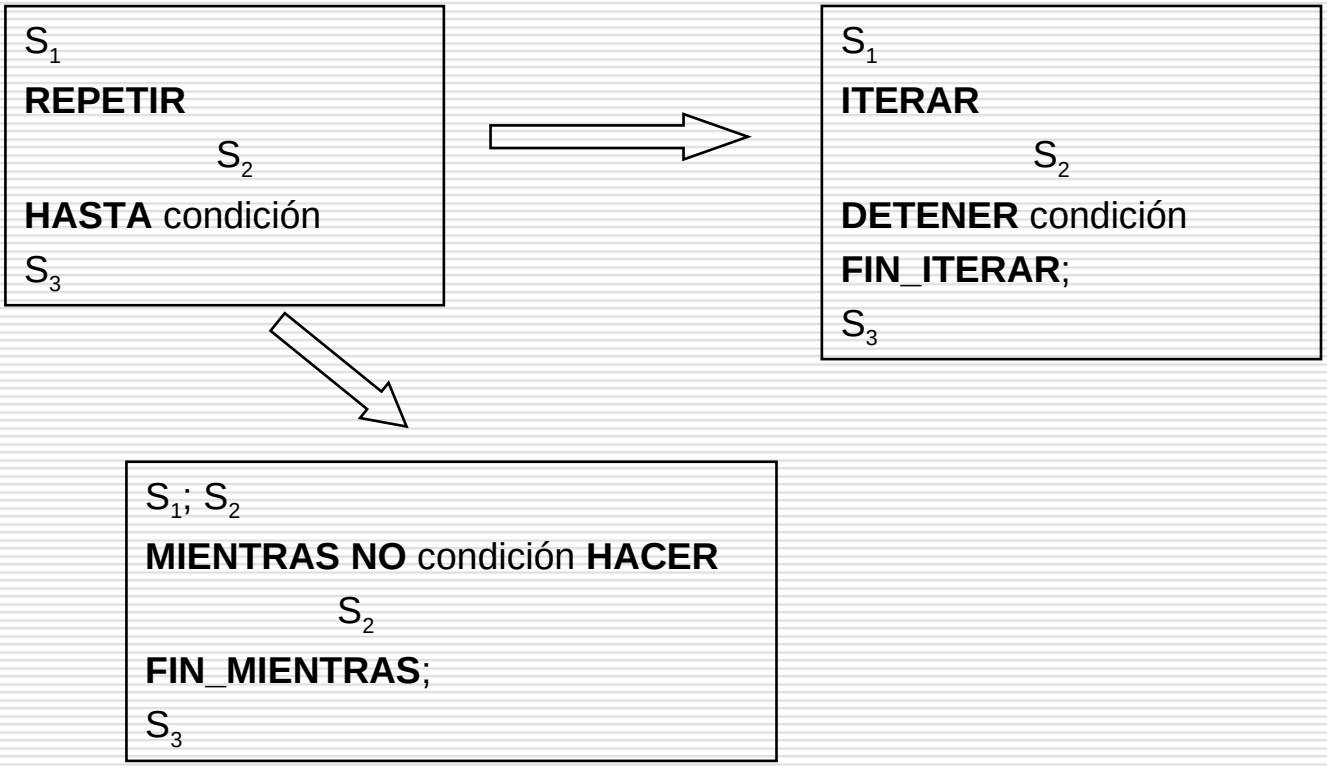
3.2 Composiciones iterativas: Equivalencias

- Cualquiera se puede expresar en términos de cualquier otra. No obstante, hay que saber elegir la adecuada en cada caso. Equivalencia del **MIENTRAS**:



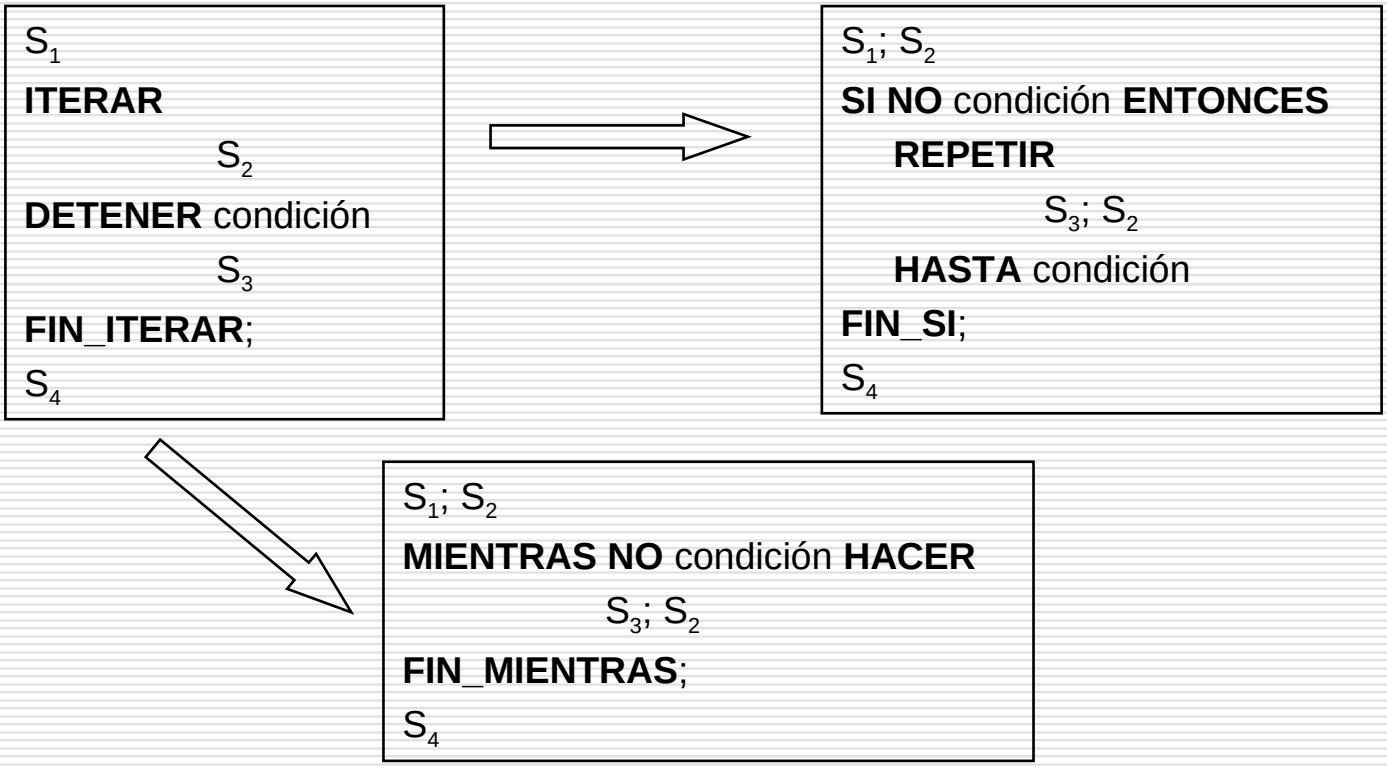
3.2 Composiciones iterativas: Equivalencias

- Equivalencia del **REPETIR**:



3.2 Composiciones iterativas: Equivalencias

- Equivalencia del **ITERAR**:



3.2 Composiciones iterativas: Composición RECORRIENDO

- Es una composición algo distinta de las anteriores.
- Su inicio y parada están implícitas en sus elementos constituyentes.
- Omitiremos los tratamientos de inicialización y final.
- **Esquema general:**

k **RECORRIENDO** [a, b] **HACER**
S
FIN_RECORRIENDO;

- Donde:
  - k es una variable de un tipo ordinal declarada en el léxico
  - a y b son expresiones del mismo tipo ordinal de k que definen un intervalo de valores (a es el límite inferior y b el superior)
  - S es una sucesión de sentencias que no alteran el valor de k

3.2 Composiciones iterativas: Composición RECORRIENDO

Semántica:

- Ejecutar el bloque de sentencias S tantas veces como elementos haya en el intervalor [a, b] haciendo en cada una de ellas variar el valor de k tomando sucesivamente los valores de este intervalo en orden ascendente:

Ejemplo:

```
i RECORRIENDO [1, 10] HACER
    Escribir (i, i*i)
FIN_RECORRIENDO
```

Escribe los 10 primeros números naturales y sus cuadrados.

El bloque de sentencias puede no ejecutarse ni una sola vez. Esto ocurre si el intervalo [a, b] es vacío (si el límite inferior 'a' está por encima del superior 'b').

3.2 Composiciones iterativas: Composición RECORRIENDO

- **Observaciones respecto a la composición RECORRIENDO:**
- El bloque de sentencias S no debe modificar el valor de la variable de control (la variable que aparece antes de la palabra **RECORRIENDO**).
- No debe suponerse la reevaluación de los límites del recorrido (a y b) en cada iteración.
- De hecho, es una precaución “sana” que estos límites no varíen durante la ejecución.
- El valor de la variable de control al final de esta iteración no está definido. Esta variable quedaría en una situación equivalente a no inicializada.

3.2 Composiciones iterativas: Composición RECORRIENDO

- Existe una variante de la iteración **RECORRIENDO** que altera el orden en el que se produce el recorrido:

k **RECORRIENDO** [a, b] **EN SENTIDO INVERSO HACER**
S
FIN_RECORRIENDO;

- Su significado es el mismo que en la anterior variante, con la diferencia de que la variable k recorre el intervalo [a, b] desde b hasta a.
- Las mismas consideraciones anteriores se aplican también a esta variante.

3.3 Ejemplos de problemas iterativos

- Calcular el producto de dos enteros no negativos a y b realizando únicamente sumas.
- Podemos resolverlo iterando sobre los múltiplos de a: 0, a, 2*a, 3*a, ..., hasta llegar a b*a.
- En cada paso de la iteración sumaremos el valor de a al resultado actual para pasar al siguiente múltiplo.
- La iteración se detiene cuando hayamos alcanzado el valor de b*a en el resultado.
- Puesto que el número de iteraciones es conocido de antemano (en tiempo de ejecución, cuando sabemos los valores de a y b) podemos utilizar una iteración de tipo **RECORRIENDO**.

3.3 Ejemplos de problemas iterativos

```
LEXICO
        a, b: Entero ≥ 0; // Datos de entrada
        p: Entero ≥ 0;     // Resultado
        i: Entero ≥ 0;     // Variable de la iteración
ALGORITMO
        Leer (a, b);
        p ← 0;       // p=0*a
        i RECORRIENDO [1, b] HACER
                p ← p+a  // Pasamos al siguiente múltiplo
        FIN_RECORRIENDO;
        Escribir (p)
FIN.
```

3.3 Ejemplos de problemas iterativos

- Cálculo del factorial de un número natural.
- El factorial de un número natural n es el producto de todos los números naturales desde el 1 hasta n.
- El factorial de n se denota como n!
- El factorial de 0 se define axiomáticamente como 1.
- El factorial puede definirse recurrentemente de la siguiente forma:
  - 0! = 1
  - n! = n * (n-1)! para n > 0
- Podemos resolver este problema iterando desde 1 hasta n y multiplicando todos los enteros de ese intervalo.

3.3 Ejemplos de problemas iterativos

- Dos formas distintas de calcular el factorial (se omite el léxico por legibilidad):

```
ALGORITMO
  Leer(n);
  fact ← 1; i ← 0;
  MIENTRAS i<n HACER
        i ← i+1;
        fact ← fact * i
  FIN_MIENTRAS;
  Escribir(fact)
FIN.
```

```
ALGORITMO
Leer (n);
fact ← 1; i ← 1;
MIENTRAS i ≤ n HACER
        fact ← fact * i;
        i ← i + 1
FIN_MIENTRAS;
Escribir (fact)
FIN.
```

3.3 Ejemplos de problemas iterativos

- Una forma más:

ALGORITMO

Leer (n)

fact ← 1;

i **RECORRIENDO** [2, n] **HACER**

fact ← fact * i

FIN_RECORRIENDO;

Escribir (fact)

FIN.

¿Cuál de las tres te parece más apropiada y por qué?

3.3 Ejemplos de problemas iterativos

Cálculo del máximo común divisor de dos enteros, a>0 y b>0, aplicando las siguientes propiedades del máximo común divisor (mcd):

- **mcd (a, a) = a**
- **mcd (a, b) = mcd (a-b, b) si a>b**
- **mcd (a, b) = mcd (a, b-a) si a<b**

La estrategia de cálculo sería realizar una iteración en la que en cada paso vayamos restándole al mayor de los valores el menor.

El proceso se repite mientras ambos valores sean distintos, y al final esa es la solución del problema.

3.3 Ejemplos de problemas iterativos

```
LEXICO
   a, b : Entero > 0;  // Datos de entrada
   u, v : Entero > 0;  // Datos intermedios y resultado
ALGORITMO
   Leer (a, b);
   u ← a; v ← b;
   MIENTRAS u ≠ v HACER
         SEGÚN u, v
            v < u : u ← u – v
            v > u : v ← v – u
         FIN_SEGÚN
   FIN_MIENTRAS;
   Escribir (u)  // Lo mismo valdría Escribir (v)
FIN.
```

3.3 Ejemplos de problemas iterativos

Obtención y cuenta de los divisores propios de un entero positivo a:

SOLUCIÓN: Recorreremos la secuencia de valores candidatos (intervalo [2, a-1]), comprobando si cada valor es divisor o no.

LÉXICO

a, n, i : Entero ≥ 0;

ALGORITMO

Leer (a); n ← 0;
i **RECORRIENDO** [2, a-1] **HACER**
SI a **MOD** i = 0 **ENTONCES**
Escribir (i); n ← n+1
FIN_SI
FIN_RECORRIENDO;
Escribir (n)

FIN.

¿Habría alguna forma mejor de hacer esto mismo?

3.3 Ejemplos de problemas iterativos

Problema: ¿Un número n es primo?

SOLUCIÓN: Haremos una búsqueda de un divisor empezando en el 2. Si el primero que encontramos es el propio n, el número es primo.

```
LEXICO
    n: Entero > 0; j: Entero > 1;
ALGORITMO
    Leer (n);
    SI n = 1 ENTONCES Escribir ("1 es primo")
    SI_NO   j ← 2;
            MIENTRAS n MOD j ≠ 0 HACER
                j ← j + 1
            FIN_MIENTRAS;
            SI  j=n ENTONCES Escribir (n, " es primo")
            SI_NO Escribir (n, " no es primo") FIN_SI
            FIN_SI
FIN.
```

¿Habría alguna forma mejor de hacer esto mismo?

3.3 Ejemplos de problemas iterativos

Dados dos enteros positivos x e y, 0<x<y, encontrar las potencias de 2 en el intervalo [x, y]: 2^x, 2^{x+1}, ..., 2^y

Solución 1: recorrido o enumeración de los enteros de x a y, calculando la potencia de 2 para cada elemento del recorrido, para ello construimos la función **P2** que calcula la potencia de 2 de un número.

```
LÉXICO
  x, y, i: Entero;
P2: función (n: Entero)→ Entero;
LÉXICO
  p, i :Entero;
ALGORITMO // de P2
   p ← 1;
   i RECORRIENDO [1, n] HACER
          p ← p*2
   FIN_RECORRIENDO;
   P2 ← p
FIN; // de P2
```

```
ALGORITMO // principal
    Leer (x, y);
    i RECORRIENDO [x, y] HACER
          Escribir (P2 (i))
    FIN_RECORRIENDO
FIN.  // del algoritmo
```

¿Habría alguna forma mejor de hacer esto mismo?

3.3 Ejemplos de problemas iterativos

Dado un entero x>0 obtener la parte entera de $\log_2 x$, es decir el enero *r* tal que $2^r \leq x < 2^{r+1}$

- Se trata de encontrar la primera potencia de 2 que sea mayor que x, esquema de búsqueda sobre la secuencia de potencias de 2.
- Recorreremos las potencias de dos y nos detendremos cuando una supere el valor de x. El valor de exponente anterior será el buscado.
- Ej: Si x = 40, r = 5, $2^5 = \mathbf{32} \leq \mathbf{40} < \mathbf{64} = 2^6$

r	0	1	2	3	4	5
2^r	1	2	4	8	16	**32**
2^{r+1}	2	4	8	16	32	**64**

3.3 Ejemplos de problemas iterativos

```
LÉXICO
    x : Entero > 0;
    r : Entero ≥ 0;
    p2 : Entero > 0;
ALGORITMO
    Leer (x);
    r ← 0; p2 ← 2;  // p2 = 2^(r+1) en cada paso
    MIENTRAS x ≥ p2 HACER
        r ← r + 1;
        p2 ← p2 * 2
    FIN_MIENTRAS;
    Escribir (r)
FIN.
```

Es posible demostrar esto por inducción sobre el número de iteraciones

3.3 Ejemplos de problemas iterativos

Dado un entero n ≥ 0 calcular la parte entera de su raíz cuadrada

- Se resuelve de una forma muy semejante al caso anterior, solo que en lugar de recorrer las potencias de dos recorreremos los cuadrados de los números naturales.
- Calcularemos los cuadrados como la suma de la serie parcial de los números impares (propiedad descubierta por los pitagóricos):

i	**1**	**2**	**3**	**4**	**5**	**…**
$impar_i$	1	3	5	7	9	…
$\Sigma\ impar_i$	1	4	9	16	25	…

3.3 Ejemplos de problemas iterativos

```
LÉXICO
    n, rc, imp, c: Entero ≥ 0;
ALGORITMO
    Leer (n);
    rc ← 0; c ← 1; imp ← 1;
    // (rc + 1)² = c en cada paso
    MIENTRAS n ≥ c HACER
            imp ← imp + 2;
            c ← c + imp;
            rc ← rc + 1
    FIN_MIENTRAS;
    Escribir (rc)
FIN.
```

Es posible demostrar esto por inducción sobre el número de iteraciones

3.3 Ejemplos de problemas iterativos

- **Problema:** escribir un algoritmo que calcule el término f_n de la sucesión de Fibonacci, $n \geq 0$. Esta sucesión se define de modo recurrente de la siguiente manera:

  $f_0 = 1$

  $f_1 = 1$

  $f_i = f_{i-1} + f_{i-2}$ (para $i \geq 2$)

- Por tanto, los primeros diez términos de la sucesión de Fibonacci son: 1, 1, 2, 3, 5, 8, 13, 21, 34 y 55.

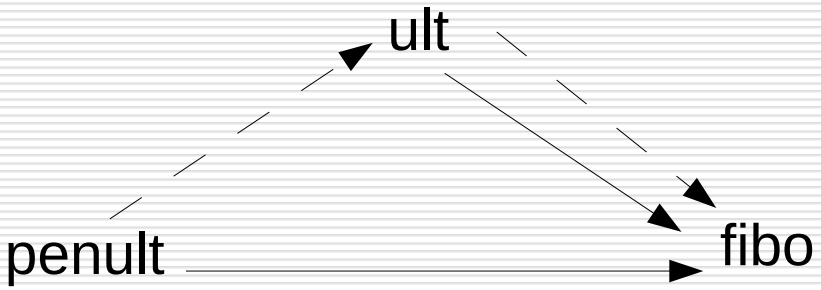
- La solución consiste en recorrer el intervalo [2, *n*], calculando en cada paso el término enésimo de la sucesión mediante la suma de los dos anteriores. Se definen tres variables: **fibo**, para almacenar el valor del término actual de la sucesión, **ult** y **penult**, para llevar cuenta de los valores de los dos términos anteriores.

3.3 Ejemplos de problemas iterativos

- La evolución, por tanto, de estas variables de la iteración i a la i+1 sería la siguiente:

	i	**i+1**
fibo	f_i	$f_{i+1} = f_i + f_{i-1} = ult_{i+1} + penult_{i+1}$
ult	f_{i-1}	$f_i = fibo_i$
penult	f_{i-2}	$f_{i-1} = ult_i$

- Y la relación precedencia entre ellas en los cálculos:



3.3 Ejemplos de problemas iterativos

```
LÉXICO
    ult, penult, fibo : Entero > 0;
    n : Entero ≥ 0;
ALGORITMO
    Leer (n);
    SEGÚN n
        n < 2 :   Escribir (1);
        n > 1 :   penult ← 1; ult ← 1; fibo ← 2;
                  i RECORRIENDO [3, n] HACER
                          penult ←  ult;
                          ult ← fibo;
                          fibo ← ult + penult;
        // { (fibo = f_i) Y (ult = f_i-1) Y (penult = f_i-2), 3 ≤ i ≤ n }
                  FIN_RECORRIENDO;
                  Escribir (fibo)
    FIN_SEGÚN
FIN.
```

3.3 Ejemplos de problemas iterativos

El problema de las potencias de 2 en [x, y] revisitado:

```
LÉXICO
        x : Entero ≥ 0;
        y, potencia2 : Entero > 0;
        i : Entero > 0;
ALGORITMO
    Leer (x, y);
    potencia2 ← 1;
    i RECORRIENDO [1, x] HACER
        potencia2 ← potencia2 * 2
    FIN_RECORRIENDO;
    { e1 : potencia2 = 2^x }
    i RECORRIENDO [x, y] HACER
        Escribir ("La potencia ", i, "-ésima de 2 es ", potencia2);
        potencia2 ← potencia2 * 2
    FIN_RECORRIENDO
FIN.
```

¿Cuál es fundamentalmente la diferencia entre este algoritmo y la versión anterior que vimos que resuelve el mismo problema?

3.4 Esquema general de una iteración

- Vamos a tratar de visualizar una “iteración” como el recorrido de una secuencia de valores: la de los conjuntos de valores que en cada paso van tomando las variables involucradas en la iteración.

- Abordaremos un problema fundamental en muchos procesos iterativos: el problema de la parada. ¿Es posible garantizar que un proceso iterativo termina siempre?

3.4 Esquema general de una iteración

$V \leftarrow V_0$;

MIENTRAS P(V) HACER

$V \leftarrow f(V)$

FIN_MIENTRAS

- **V**: conjunto de variables implicadas
- **V_0**: conjunto de valores iniciales de las variables de V
- **P**: condición de continuación
- **f**: función que representa el conjunto de operaciones que en cada paso modifican los valores de las variables de V

3.4 Esquema general de una iteración

- Si la iteración consta de *n* pasos y v_i representa los valores del conjunto de variables v en el paso i-ésimo, se cumple que:

1. $v_i = f(v_{i-1})$ para todo $0 < i \leq n$
2. $v_i \neq v_j$ para todo $i \neq j$
3. $P(v_i)$ = verdadero, para todo $i < n$
4. $P(v_n)$ = falso

3.4 Esquema general de una iteración

- Ejemplo del cálculo del factorial:
- V = {i, fact, n}
- V_0={i ← 0; fact ← 1; n = *valor leído de la entrada*}
- f={i ← i+1; fact ← fact*i}
- P(V)={i<n}
- Nótese que en para i=n-1, la función f, actualiza i, y a continuación se calcula el factorial de n. Cuando i=n, no se entra en la iteración.

```
LÉXICO
  n, i, fact : Entero ≥ 0;
ALGORITMO
  Leer(n); i ← 0; fact ← 1;
  MIENTRAS i<n HACER          // En cada paso: fact = i !
        i ← i+1; fact ← fact * i
  FIN_MIENTRAS;
  Escribir(fact)
FIN.
```

3.4 Esquema general de una iteración

- Ejemplo de la búsqueda de si un número n es primo:

```
LÉXICO
   n: Entero > 0; j: Entero > 1;
ALGORITMO
   Leer (n);
   SI n = 1 ENTONCES Escribir ("1 es primo")
   SI_NO       j ← 2;
      MIENTRAS n MOD j ≠ 0 HACER
                  j ← j + 1
      FIN_MIENTRAS;
      SI  j=n ENTONCES Escribir (n, " es primo")
      SI_NO Escribir (n, " no es primo") FIN_SI
   FIN_SI
FIN.
```

COMPONENTES DE LA ITERACIÓN:

- V = { n, j }
- V_0 = { j ← 2; n *valor leído* }
- f = { j ← j + 1 }
- P(V) = { n **MOD** j ≠ 0 }

3.4 Esquema general de una iteración

- **EL PROBLEMA DE LA PARADA:** ¿Cómo sabemos que la iteración finaliza después de un número finito de pasos?
- Que una iteración no se detenga nunca es una de las causas más frecuentes de los “cuelgues” de los programas.
- En general, salvo en el caso de la iteración RECORRIENDO, no tenemos garantía de que una iteración llegue a un punto en el que se detenga.
- Analizar una iteración y garantizar su parada tras un número finito de pasos es en general un problema **muy** difícil.
- En iteraciones de cálculo, la parada está relacionada con la convergencia de los algoritmos utilizados.
- Veremos una forma simple de estudiar este problema que en muchos casos puede ser útil.

3.4 Esquema general de una iteración

- Definimos una función T que va del conjunto de las variables involucradas en la iteración (o un subconjunto de éste) a los enteros:

$$T : V \longrightarrow \mathbb{Z}$$

- Esta función debe cumplir las siguientes propiedades:

1) Cuando la condición de continuación P(V) se satisface (es verdadera), T(V) toma un valor estrictamente positivo.
2) En cada paso de la iteración el valor de T(V) es estrictamente decreciente.
3) Cuando T(V) $\leq$ 0 la condición de continuación P(V) es falsa.

La función T puede entenderse como una “función de energía”, o medida de “entropía inversa” del proceso iterativo. Se le suele llamar “función de cota”, porque su valor inicial acota el número de iteraciones.

3.4 Esquema general de una iteración

- ¿Qué función de cota podríamos definir para la búsqueda de si un número es primo?

```
LÉXICO
  n: Entero > 0; j: Entero > 1;
ALGORITMO
  Leer (n);
  SI n = 1 ENTONCES Escribir ("1 es primo")
  SI_NO      j ← 2;
    MIENTRAS n MOD j ≠ 0 HACER
                j ← j + 1
    FIN_MIENTRAS;
    SI  j=n ENTONCES Escribir (n, " es primo")
    SI_NO Escribir (n, " no es primo") FIN_SI
  FIN_SI
FIN.
```

POR EJEMPLO:

$$T(V) = n - j$$

- Es estrictamente decreciente
- SI n **MOD** j ≠ 0, T(V) es positiva
- Cuando T(V) llega a ser cero, n **MOD** j = 0

NOTA:
P(V) puede ser falsa antes de que T(V) llegue a ser ≤ 0.

3.4 Esquema general de una iteración

- ¿Y en el caso del cálculo del máximo común divisor?

```
LEXICO
   a, b, u, v : Entero > 0;
ALGORITMO
   Leer (a, b);
   u ← a; v ← b;
   MIENTRAS u ≠ v HACER
      SEGÚN u, v
         v < u : u ← u – v
         v > u : v ← v – u
      FIN_SEGÚN
   FIN_MIENTRAS;
   Escribir (u)
FIN.
```

EJEMPLO DE FUNCIÓN DE COTA:

$$T(V) = u + v - 2 \cdot mcd(a, b)$$

Ejercicio: demostrar que esta función cumple todas las propiedades requeridas para una función de cota.