Metodología de la Programación

Transparencias de la asignatura

Joaquín Ezpeleta

Departamento de Informática
e Ingeniería de Sistemas

Centro Politécnico Superior

Universidad de Zaragoza

Metodología de la Programación
La asignatura

- Marco general:
  - metodología y tecnología de la Programación
    - » IP, MP, EA
  - estructuras de datos y de la información
    - » EDA, FBD
- Créditos:
  - teóricos: 4.5
  - prácticos: 1.5+1.5

adquirir madurez en Programación

OBJETIVOS

*** evaluar la eficiencia de un algoritmo y poder compararlo con otros que resuelvan el mismo problema**

*** razonar sobre la corrección de un algoritmo**

*** habituarse a documentar formalmente los programas**

*** diseñar algoritmos recursivos e iterativos, demostrar su corrección y evaluar su eficiencia**

*** conocer y saber aplicar un conjunto de técnicas algorítmicas fundamentales**

Metodología de la Programación
La asignatura

Tema 1: Análisis de la eficiencia

- Nociones sobre eficiencia de algoritmos
- Notaciones para medir la eficiencia de algoritmos
- Jerarquía de eficiencias
- Cálculo de la eficiencia de un algoritmo

Tema 2: Introducción a la especificación y verificación de algoritmos

- Especificación de algoritmos mediante predicados
- El transformador de predicados "`pmd`"
- Semántica de un lenguaje imperativo
- Introducción a la derivación de algoritmos

Metodología de la Programación
La asignatura

Tema 3: Diseño de algoritmos recursivos

- Introducción a la recursividad
- El método de inducción
- Demostración de propiedades por inducción
- Ejemplos de planteamientos recursivos
- Inducción Noetheriana
- Algoritmos recursivos: diseño, verificación y cálculo de la complejidad
- Técnicas de inmersión
  - transformación de algoritmos por inmersión
    - inmersión por cuestiones de eficiencia
    - técnicas de plegado y desplegado
  - diseño de algoritmos por inmersión
    - por debilitamiento de Post
    - por reforzamiento de la Pre

Metodología de la Programación
La asignatura

<u>Tema 4</u>: Diseño de algoritmos iterativos

- Introducción
- Recursividad final con Post constante y solución iterativa
- Corrección de programas iterativos
- Transformación recursivo-iterativo
- Derivación de algoritmos iterativos

<u>Tema 5</u>: Esquemas algorítmicos

- El esquema "divide y vencerás"

- **Programa de prácticas**:
  - Construcción de módulos y medida experimental de la complejidad
  - Especificación y anotación de programas
  - Diseño de programas recursivos
  - Diseño de programas iterativos
  - Transformación recursivo/iterativo

Metodología de la Programación
Bibliografía

- "Programación metódica"
  J.L. Balcázar, McGraw-Hill, 1993
  [Balc 93]

- "Algorítmica: concepción y análisis"
  G. Brassard, P. Bratley, Ed. Masson, 1990, [BrBr 90]

- "Verificación y desarrollo de programas"
  R. Cardoso, Ediciones Uniandes, 1991, [Card 91]

- "The science of programming"

  D. Gries, Texts and Monographs in Computer Science, Springer-Verlag, 1981, [Grie 81]

- "Diseño de programas. Formalismo y abstracción"
  R. Peña, Prentice-Hall, 1993, [Peña 93]
  segunda edición de 1997 [Peña 97]

- Algún buen libro de Matemática Discreta

Metodología de la Programación
Sobre Prácticas

- Sobre las prácticas:
  - realización obligatoria
  - asistencia no obligatoria
  - precondición para presentarse a examen
  - inscribirse en lista de prácticas
  - al igual que la teoría, se guarda nota las tres posibles convocatorias
  - por parejas
  - se entregarán con tiempo para preparar
    - » trabajo en casa
    - » clase: dudas, implementación, evaluación
  - Fechas:
  - Lugar:
  - Nota: hasta 1.5 puntos

TEMA 1: Análisis de la eficiencia de algoritmos

1) Nociones sobre eficiencia de algoritmos

2) Notaciones para medir la eficiencia de algoritmos

3) Jerarquía de eficiencias

4) Cálculo de la eficiencia de un algoritmo

5) Ejemplos y ejercicios

Nociones sobre eficiencia de algoritmos

Algoritmo

Conjunto de **operaciones elementales** organizadas de acuerdo a **reglas** precisas, y cuyo objetivo de resolver un problema dado.

Para cada dato del problema (entrada) el algoritmo debe dar **respuesta** en un número **finito** de pasos

- Para un mismo problema, el algoritmo no es único
  - ¿Qué algoritmo elegir?
  - ¿Con qué criterios se determina si un algoritmo "me conviene más o menos"?

Nociones sobre eficiencia de algoritmos

Eficiencia de un algoritmo

medida de los recursos necesarios para su ejecución

- Aspectos que deben considerarse:
  - espacio
  - simplicidad
  - adecuación a los datos
  - tiempo

¿Dónde está el tiempo?

Eficiencia en tiempo

número de operaciones elementales a realizar, en función del tamaño de los datos de entrada

Operación elemental

operación cuyo tiempo de ejecución está acotado superiormente por una constante

  » no depende del tamaño de los datos
    - coste unitario

Nociones sobre eficiencia de algoritmos

- Normalmente, tres métodos:

  empírico/teórico/híbrido

¡Trabajoso!

- Método empírico:
  - implementar los distintos algoritmos, y mediante pruebas, tomar una determinación
- Método teórico:
  - determinar, teóricamente, la cantidad de recursos que cada algoritmo necesitaría, en función al tamaño del problema
    - » **tamaño**: número de bytes, pero, habitualmente, número de datos
  - Ventajas claras:
    - » no depende del computador
    - » no depende del lenguaje ni del compilador
    - » eficiencia “medible” para cualquier tamaño de datos

Nociones sobre eficiencia de algoritmos

- Método híbrido:
  - cálculo de la eficiencia teórica
  - implementación específica para ajustar ciertos parámetros (constantes) a la máquina específica
- ¿Cómo medir la eficiencia de un algoritmo?

`Principio de invarianza`

`la eficiencia de distintas implementaciones de un mismo algoritmo difiere únicamente en una constante multiplicativa`

  - notación "del orden de": establece cómo es el crecimiento de las necesidades de recursos en función al tamaño de los datos a procesar
    - » notación asintótica: crecimiento lineal, cuadrático, exponencial,....

Nociones sobre eficiencia de algoritmos

- ¿Merece la pena complicarse tanto la vida?
  - Un ejemplo: cálculo del determinante
    - método de los adjuntos

      n=10 → t=600 sg.

      n=20 → t=10 millones de años
    - método de Gauss-Jordan:

      n=10 → t=0.01 sg.

      n=20 → t=0.05 sg.

Nociones sobre eficiencia de algoritmos

- Ejemplo 1: Producto escalar de dos vectores.

```
Constantes n = 30
Tipos   vect = vector[1..n] de real

Función prodEscalar(E v1,v2:vect)
                         dev (pE:real)
--Q:
--R: pE=producto escalar de v1 y v2

Variables i:entero
Principio
   pE := 0
   Para i := 1 hasta n
      pE := pE + v1[i]*v2[i]
   FPara
   devuelve(pE)
Fin
```

- ¿Tamaño del problema?
- Nº de operaciones elementales
  - 3n+2 (más o menos)
  - tiempo $\approx \alpha n + \beta$
  - ¿Espacio?

Ejemplo 2: Ordenación de un vector por selección

```
Constantes n = 30
Tipos vect = vector[1..n] de real

Algoritmo ordenaSelección(ES v:vect)
--Q:
--R: v queda ordenado “<=”
Variables i,posMin:entero; x:real
Principio
    Para i := 1 hasta n-1
      --posMin: pos. del mínimo en v[i],...,v[n]
        posMin := i
        Para j := i+1 hasta n
            Si v[j]<v[posMin] ent
                posMin := j
            FSi
        FPara
        x := v[i]; v[i] := v[posMin]
        v[posMin] := x
    FPara
Fin
```

- ¿Tamaño del problema?
- Nº. de operac. elementales
- ¿Cuándo funciona mejor y/o peor?

Nociones sobre eficiencia de algoritmos

- El tiempo de ejecución depende de:
  - 1- el tamaño de los datos
  - 2- el valor de los datos
  - 3- la eficiencia del compilador
  - 4- el computador
- Normas de “sentido común” para medir la eficiencia
      - » de manera independiente de la máquina (3 y 4 no se considerarán)
  - – considerar 2 implica conocer distribuciones probabilísticas de los datos
  - – sólo consideraremos 1
    - » coste en el peor caso
    - » coste en el mejor caso

`cota superior`

`cota inferior`

Notaciones para medir la eficiencia de algoritmos

- Análisis teórico de la eficiencia
  - no constantes multiplicativas
    - » abstraerse de implementación
    - » abstraerse de lenguaje
    - » abstraerse de computador
  - notación "*del orden de*" (asintótica)
- Objetivo: encontrar
  - cota superior para el crecimiento, como f(n)
    - » Ejemplo: coste inferior a n^3
  - cota inferior para el crecimiento, como f(n)
    - » Ejemplo: coste superior a n

Notaciones para medir la eficiencia de algoritmos

O(g(n))

$R^* = R^+ \cup \{0\}$

Sea g: $\mathbb{N}$ **--->** $\mathbb{R}^*$**. Denotamos**

$$\mathbf{O}(\mathbf{g}(\mathbf{n})) = \{f : \mathbb{N} \dashrightarrow \mathbb{R}^* \mid \exists\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{N}, \forall\, n >= n_0,\ f(n) <= cg(n)\ \}$$

- O(g(n)):
  - » conjunto de funciones <u>del orden de</u> g
  - » conjunto de las funciones <u>mayoradas</u> por g(n)

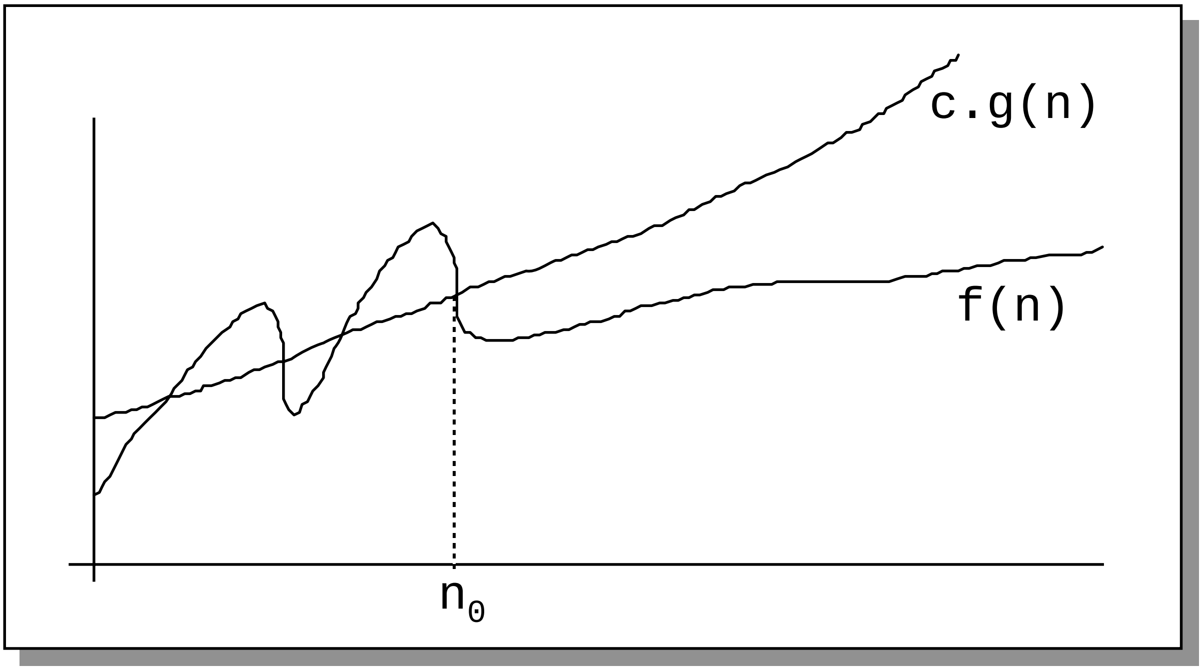
- Se suele denotar como:

$$f = \mathbf{O}(g(n)) \quad ó \quad f \in \mathbf{O}(g(n))$$

» Introducida por Bachmann en 1894,
y "popularizada" por Landau

Notaciones para medir la eficiencia de algoritmos

- Fundamental: significado de `c,n`$_0$



$$\boxed{f(n) \in O(g(n))}$$

- Significado: si el tiempo de ejecución de una implementación concreta es **f(n)∈O(g(n))**, cualquier otra implementación **f'(n)**, que difiera sólo en la máquina/compilador/lenguaje, será también del orden **O(g(n))**

máquinas distintas

$\Rightarrow$

`c,n`$_0$ distintos

Notaciones para medir la eficiencia de algoritmos

- ¿Qué significa que un algoritmo sea $\mathtt{O(an^2+bn+c)}$?

- Pero, ¡No es la panacea!

$$\boxed{f(n)\in O(n^2) \Rightarrow f(n)\in O(n^k), k>=2}$$

- Más general:

$$\boxed{\begin{array}{c} f(n)\in O(g(n)), g(n)\in O(h(n)) \\ \Rightarrow \\ f(n)\in O(h(n)) \end{array}}$$

- Ejercicio: Probar que $\boxed{n^3\notin O(n^2)}$

Notaciones para medir la eficiencia de algoritmos

- En conclusión, para una `f(n)`dada, deberíamos encontrar la función `g(n)` "más pequeña" que verifique

$$\boxed{f(n) \in O(g(n))}$$

- También nos interesa acotar "por debajo"

  objetivo: "emparedar"

Notaciones para medir la eficiencia de algoritmos

$\Omega(g(n))$

Sea g:$\mathbb{N}$ **--->** $\mathbb{R}^*$**. Denotamos**

$$\Omega(g(n)) = \{ f: \mathbb{N} \dashrightarrow \mathbb{R}^* \mid \exists\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{N}, \forall\, n >= n_0,\ f(n) >= cg(n) \}$$

- $\Omega(g(n))$:
  - » conjunto de las funciones minorantes de g(n)

- Se suele denotar como:

  $f = \Omega(g(n))$ ó $\boxed{f \in \Omega(g(n))}$

- <u>Proposición</u>:

$$\boxed{f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))}$$

Notaciones para medir la eficiencia de algoritmos

- Y uniendo ambas notaciones

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

- $\Theta(f(n))$:**orden <u>exacto</u>** de f
- Ejercicio:

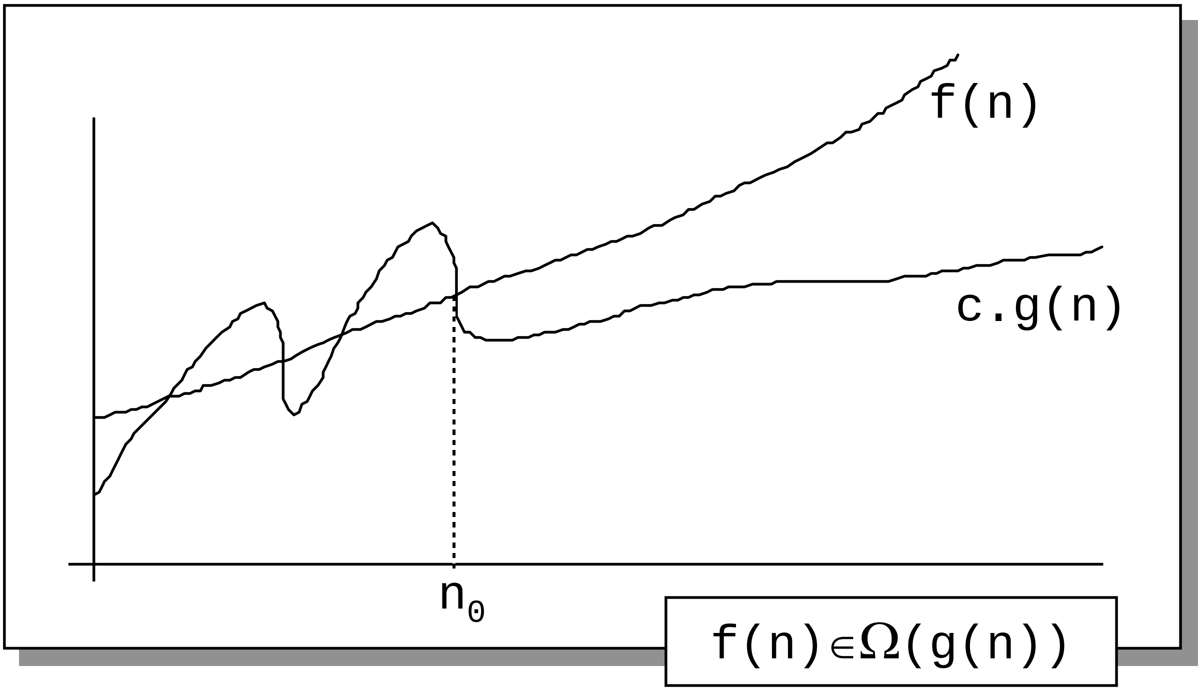
$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists\ c,d \in R^{+}, \exists\ n_0 \in N \ni \forall n >= n_0\ cg(n) <= f(n) <= dg(n)$$

- Proposición:

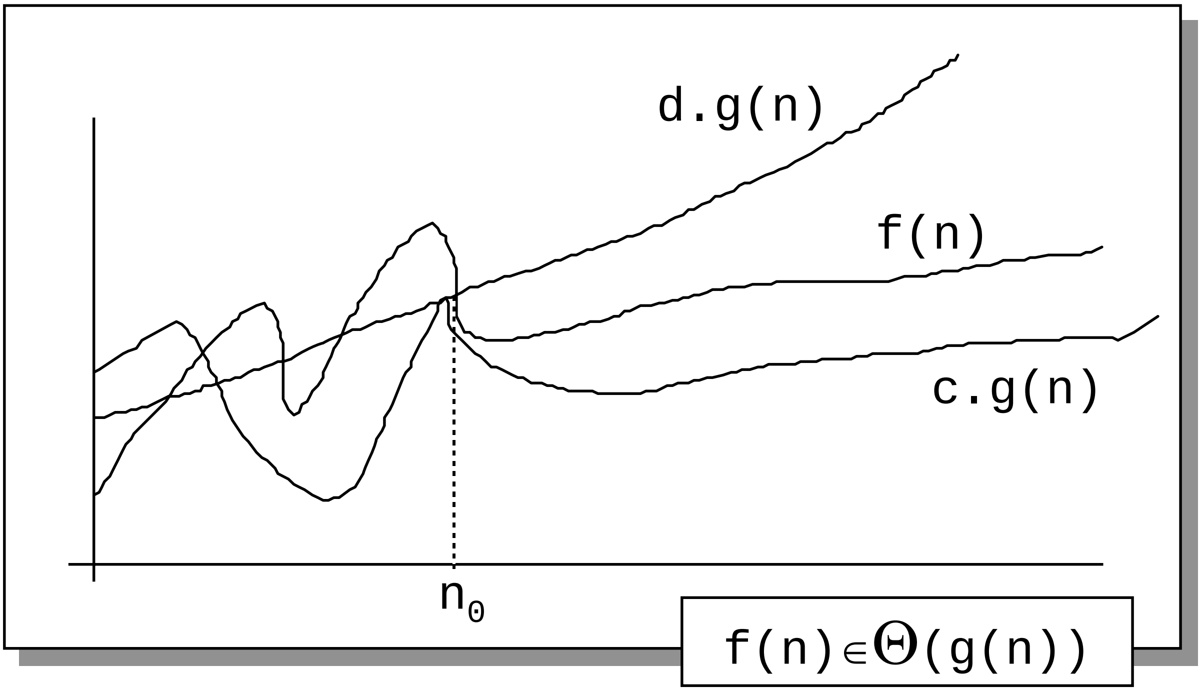
$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ y } g(n) \in O(f(n))$$

Notaciones para medir la eficiencia de algoritmos

- Gráficamente:



$f(n) \in \Omega(g(n))$



$f(n) \in \Theta(g(n))$

Notaciones para medir la eficiencia de algoritmos

- Suma de órdenes de eficiencia

$$\begin{array}{l} O(f(n))+O(g(n))=\{h:N\text{-->}R^{+} \cup \{0\} \mid \\ \quad \exists f' \in O(f(n)), \exists g' \in O(g(n)), \exists n_0 \in N \\ \qquad \forall n>=n_0,\ h(n)=f'(n)+g'(n) \\ \} \end{array}$$

- Producto de órdenes de eficiencia

$$\begin{array}{l} O(f(n)).\cdot O(g(n))=\{h:N\text{-->}R^{+} \cup \{0\} \mid \\ \quad \exists f' \in O(f(n)), \exists g' \in O(g(n)), \exists n_0 \in N \\ \qquad \forall n>=n_0,\ h(n)=f'(n)\cdot g'(n) \\ \} \end{array}$$

Notaciones para medir la eficiencia de algoritmos

- Propiedades interesantes:

1) $g(n) \in O(g(n))$

2) $c\Theta(g(n)) = \Theta(g(n)), \forall\ c \in R^+$

3) $\Theta(g(n)) + \Theta(g(n)) = \Theta(g(n))$

regla de la suma

4) $\Theta(g_1(n)) + \Theta(g_2(n)) = \Theta(g_1(n)+g_2(n)) =$
$\Theta(\max(g_1(n),g_2(n)))$

regla del producto

5) $\Theta(g_1(n))\ \Theta(g_2(n)) =$
$g_1(n)\ \Theta(g_2(n)) = \Theta(g_1(n))g_2(n)$

6) $f(n) \in \Theta\ (g(n)) \Rightarrow f(n) \in O(g(n))$

7) El recípro de 6) no es cierto

Notaciones para medir la eficiencia de algoritmos

- Lo mismo vale para k funciones, en lugar de dos

- Ejercicios interesantes:

1) Si $f(n) \in O(n)$ ¿⇒?

- $(f(n))^2 \in O(n^2)$
- $2^{f(n)} \in O(2^n)$

2) $\forall\ k \in N,\ f(n) = \sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$

3) Si $g_1, g_2 : N \to R^+$,

$O(g_1(n)+g_2(n))$ ¿=? $g_1(n)+O(g_2(n))$

Jerarquía de eficiencias

- Jerarquías de órdenes

$$O(1) \subset O(\log(n)) \subset O(n) \subset O(n\log(n)) \subset \ldots \subset O(n^a) \subset \ldots \subset O(a^n) \subset O(n!)$$

- Denominaremos:

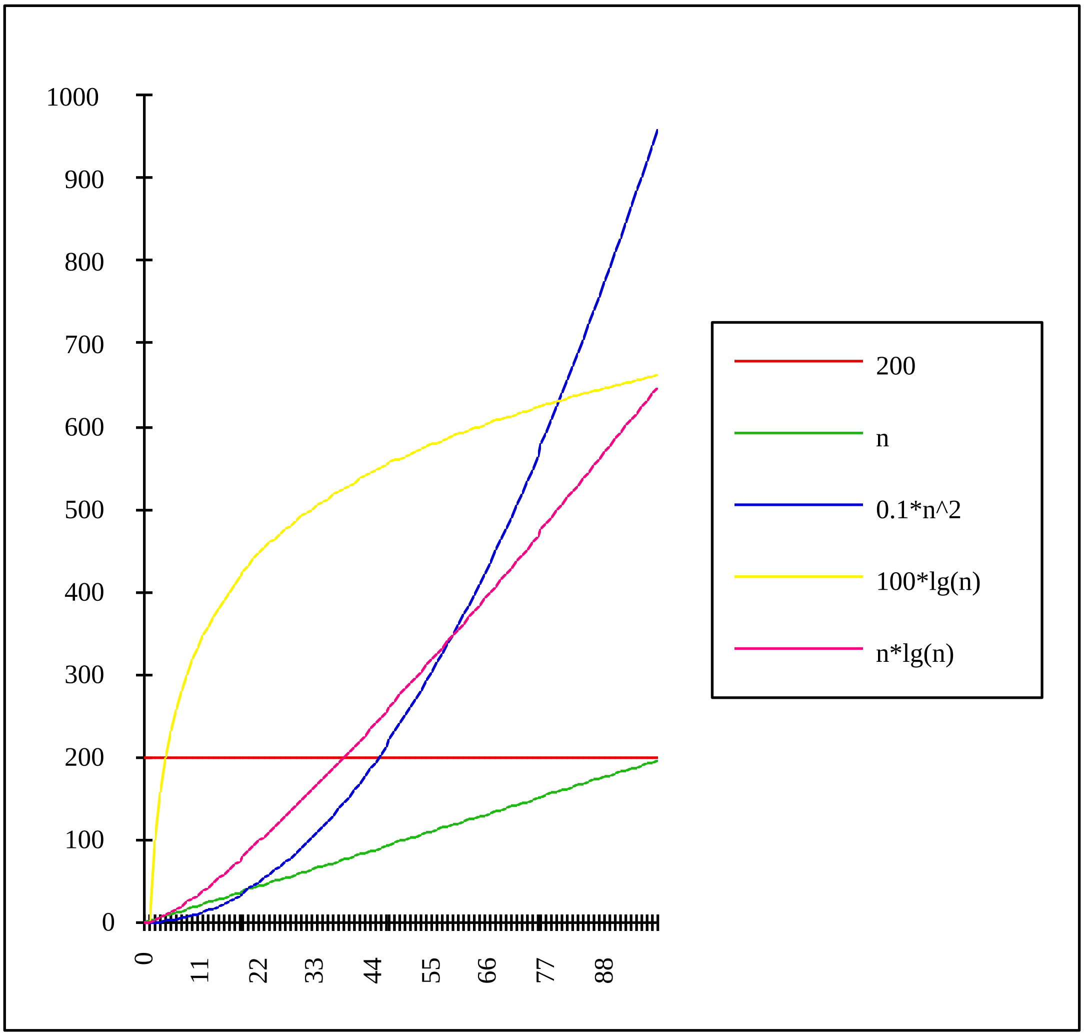
	complejidad
$O(1)$	**constante**
$O(\log n)$	**logarítmica**
$O(n)$	**lineal**
$O(n\log(n))$	**"enelogene"**
$O(n^a)$	**polinomial**
$O(a^n)$	**exponencial**
$O(n!)$	**factorial**

¡INTRATABLES!

¡Pero necesarios!

Jerarquía de eficiencias

- Las distintas curvas tienen el siguiente aspecto



Jerarquía de eficiencias

- Proposición:

1) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \overset{\not\Leftarrow}{\Rightarrow} f(n) \in \Theta(g(n))$$

a <> 0

2) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n))$$
$$\not\Leftarrow$$
$$f(n) \notin \Theta(g(n))$$

3) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow g(n) \in O(f(n))$$
$$\not\Leftarrow$$
$$g(n) \notin \Theta(f(n))$$

- Consideraciones sobre jerarquías:
  - dan una idea del comportamiento ASINTOTICO del algoritmo
  - pero también puede ocurrir que, para los tamaños de datos que vayamos a manejar, sea mejor n^3 que $10000n^2$
  - en la “bondad” también son factores a considerar el trabajo de desarrollo y de uso

Cálculo de la eficiencia de un algoritmo

- Objetivo: medir la eficiencia de un algoritmo
- Formas "comunes":
  - ver que es equivalente a uno de complejidad conocida
    - no siempre es posible
  - la cuenta de la vieja
    - posible, con astucia
- Algunas reglas para calcular la eficiencia
  - regla de coste constante
  - regla de composición secuencial
  - regla de composición condicional
  - regla de composión iterativa
  - regla de invocación a proc. y func.

Cálculo de la eficiencia de un algoritmo

- Regla del Coste constante

asignación (indep.de n)
E/S (indep.de n)
evaluación de expresiones (indep.de n)

$\boxed{\texttt{coste= } \Theta(1)}$

- Regla de la Composición secuencial:

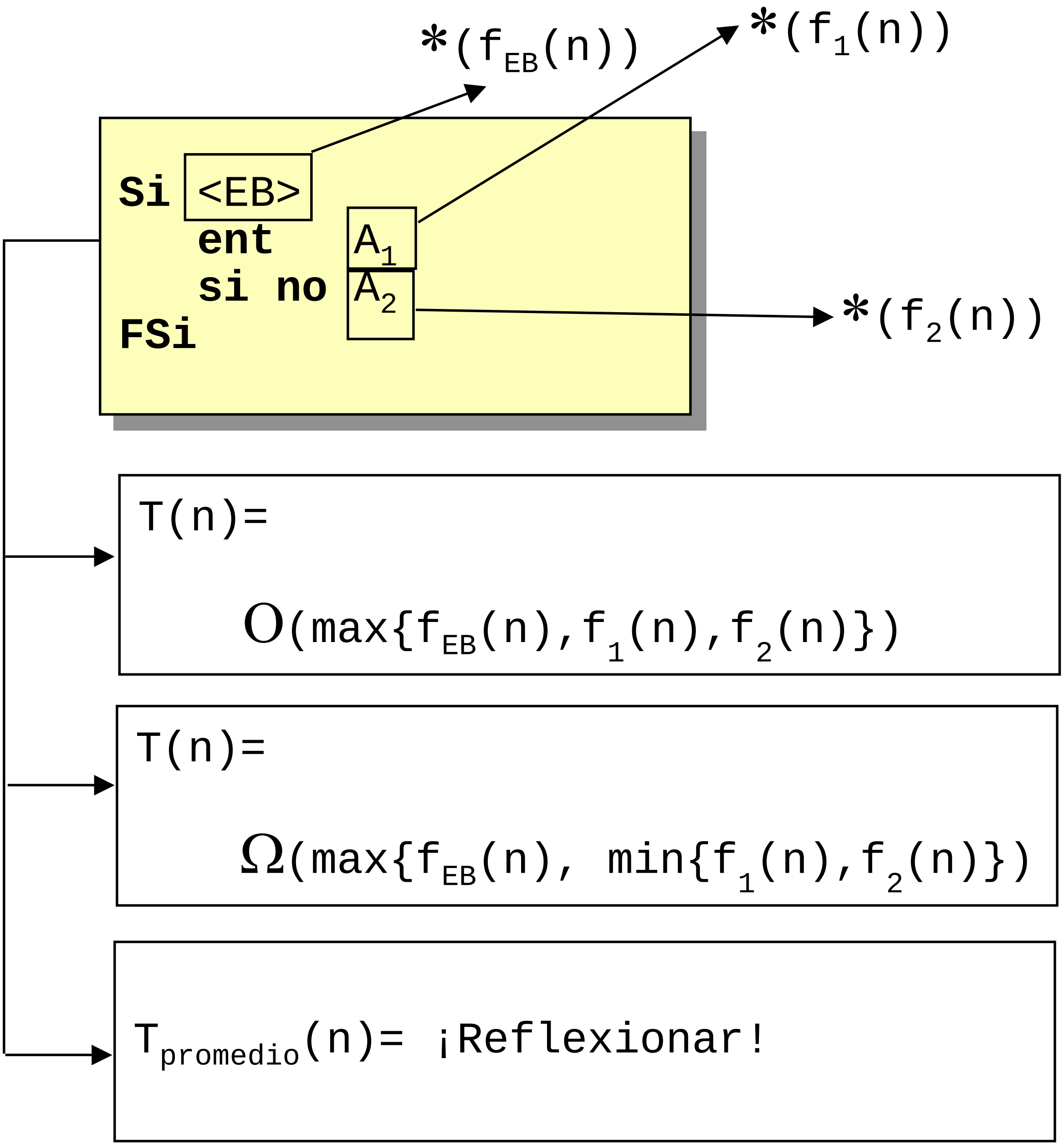
$A_1;A_2$

$A_i \in \Theta(f_i(n))$

$$\texttt{coste= } \Theta(f_1(n))+\Theta(f_2(n))= \Theta(\max\{f_1(n),f_2(n)\})$$

Cálculo de la eficiencia de un algoritmo

- Regla de la composición condicional:



```
Si <EB>
   ent   A1
   si no A2
FSi
```

$*(f_{EB}(n))$

$*(f_1(n))$

$*(f_2(n))$

T(n)=

$O(\max\{f_{EB}(n),f_1(n),f_2(n)\})$

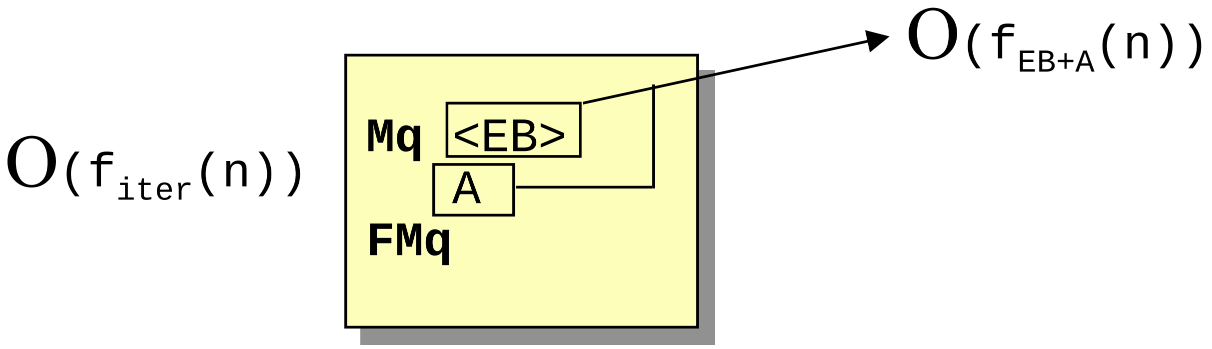
T(n)=

$\Omega(\max\{f_{EB}(n), \min\{f_1(n),f_2(n)\})$

$T_{promedio}(n)=$ ¡Reflexionar!

Cálculo de la eficiencia de un algoritmo

- Regla de la composición iterativa



  - se aplica regla del producto

$$\boxed{O(f(n)).O(g(n))=O(f(n).g(n))}$$

  - Si coste de cada iteración el mismo:

$$\boxed{T(n)= O(f_{EB+A}(n).f_{iter}(n))}$$

  - Si coste iteraciones distinto:

$$\boxed{T(n)= \sum_{i=1}^{f_{iter}(n)} O(\text{coste iteración i})}$$

Cálculo de la eficiencia de un algoritmo

- Regla de la invocación a un procedimiento o función:
  - tiempo total:
    1) tiempo de llamada
    2) tiempo de ejecución
    3) tiempo de retorno
  - en general: 1) y 3) es $\Theta(1)$

¡OJO! No siempre. Es preciso tener presente paso por valor de vectores, evaluación de expresiones, ...

Ejemplos y ejercicios

1

```
x := x+1
```

2

```
Para i:=1 hasta n
   Para j:=1 hasta n
      Para k:=1 hasta n
          x:=x+1
      FPara
   FPara
FPara
```

3

```
x:=0
Para i:=1 hasta n
   x:=x+i
FPara
```

4

```
Si n MOD 2=0 entonces
   Para i:=1 hasta n
      x:=ROUND(x+1/i)
   FPara
FSi
```

5

```
i:=1
Mq i<=n
   x:=x+1
   i:=i+2
FMq
```

6

```
Para i:=1 hasta n
   Para j:=1 hasta i
        x:=x+1
   FPara
FPara
```

7

```
x:=1
Mq x<n
   x:=2x
FMq
```

Ejemplos y ejercicios

- Ejemplo:

```
Constantes n = ????
Tipos vect = vector[1..n] de real

Algoritmo ordenaSelección(ES v:vect)
--Q:
--R: v queda ordenado no decrecientemente

Variables i,posMin:entero; x:real
Principio
    Para i := 1 hasta n-1
      --posMin: pos. del mínimo en v[i],...,v[n]
        posMin := i
        Para j := i+1 hasta n
            Si v[j]<v[posMin] ent
                posMin := j
            FSi
        FPara
        x := v[i]; v[i] := a[posMin]
        a[posMin] := x
    FPara
Fin
```

Ejemplos y ejercicios

- Ejemplo: producto de matrices cuadradas

```
Constantes n=...
Tipos M=vector[1..n,1..n] de real

Algoritmo mulMat(E m1,m2:M;S m3:M)
--Q:
--R: m3=m1*m2

Variables i,j:entero
Principio
   Para i := 1 hasta n
      Para j := 1 hasta n
        m3[i,j]:=0
        Para k := 1 hasta n
           m3[i,j]:=m3[i,j]+
               m1[i,k]*m2[k,j]
        FPara
      FPara
   FPara
Fin
```

Ejemplos y ejercicios

- Ejemplo: ordenación por inserción directa

(v[0] es el centinela)

```
Constantes n = ?????
Tipos vect = vector[0..n] de real

Algoritmo ordenaInserDir(ES v:vect)
--Q:
--R: v[1],...,v[n] queda ordenado “<=”

Variables i,j:entero; x:real
Principio
    Para i := 2 hasta n
        x:=v[i]
        v[0]:=x
        j:=i
        Mq x<v[j-1]
            v[j]:=v[j-1]
            j:=j-1
        FMq
            --como ¬(x<v[j-1]),
             --x debe estar en v[j]
        v[j]:=x
    FPara
Fin
```

Ejemplos y ejercicios

- Ejemplo: ordenación por intercambio (burbuja)

```
Constantes n = ?????
Tipos vect = vector[1..n] de real

Algoritmo ordenaSelección(ES v:vect)
--Q:
--R: v[1],...,v[n] queda ordenado “<=”

Variables i,j:entero; temporal,x:real
Principio
    Para i:=1 hasta n-1
        Para j:=n descend hasta i+1
            Si v[j-1]>v[j] ent
                temporal:=v[j-1]
                v[j-1]:=v[j]
                v[j]:=temporal
            FSi
        FPara
    FPara
Fin
```

Ejemplos y ejercicios

```
algoritmo busqSecOrd(ES v:tipoVec
                     E miClave:entero
                     S éxito:booleano...)

--Pre:  v[1]<=v[2]<=..<=v[n]
--Post: Si i es el índice menor de manera que
--    miClave=v[i], entonces exito el valor
--    Cierto. Si no existe índic que coincida,
--    exito toma el valor falso. Además...

variables  índice:entero;
principio
   índice:=1
   Mq (índice<nEl) ∧ (v[índice]<miClave)
      índice:=índice+1
   FMq
       --¿Qué condición de la guarda
       --ha sido violada?
   exito:=miClave=v[índice]
   .........
fin
```

¿Hubiera cambiado algo que en lugar de “**ES** v:tipoVec“ apareciera “**E** v:tipoVec“ ?

Ejemplos y ejercicios

```
algoritmo busqDicotómica(ES v:tipoVec
                E miClave:entero
                S éxito: booleano...)
--Pre:  v[1]<=v[2]<=..<=v[n]
--Post: Si i es el índice menor de manera que
--   v[i]=miClave, entonces exito toma el valor
--   Cierto. En caso contrario (no existe índice
--   que coincida), exito toma el valor falso.
--   Además....

variables I,S:entero
               --Acotan spacio de búsq.
          M:entero
             --Punto medio espacio de búsq.
principio
   I:=1;  S:=n  --Buscar entre todos
   Mq   I<>S
      M:=(I+S) div 2
      Si miclave<=v[M]
         ent    S:=M    --Está en [I,M]
         sino   I:=M+1  --Está en (M,S]
      FSi
   FMq
   éxito:=miClave=v[I]
   ........
fin
```

Ejemplos y ejercicios

- Ejemplo: ordenación por inserción binaria

```
Constantes n = ?????
Tipos vect = vector[1..n] de real

Algoritmo ordenaInserBin(ES v:vect)
--Pre:
--Post: v[1],...,v[n] queda ordenado “<=”

Variables i,j,iz,de,m:entero; x:real
Principio
    Para i := 2 hasta n
        x:=v[i];iz:=1;de:=i-1
        Mq iz<=de
            m:= (iz+de) DIV 2
            Si x<=v[m]
               ent  de:=m-1
               sino iz:=m+1
            FSi
        FMq
        Para j:=i-1 descend iz
            v[j+1]:=v[j]
        FPara
        v[iz]:=x
    FPara
Fin
```

Ejemplos y ejercicios

```
Funcion contarPalabras(ES f:texto)
                              dev (nP:entero)
--Pre: f abierto para lectura y al principio
--Post: nP=num. palabras del fichero

Variables  numPal:entero
           elCar : caracter
Principio
  numPal:=0
  Mq ¬finFichero(f)
    Si finLinea(f)
       ent  leerLinea(f)
       si_no
          leer(f,elCar)
          Si elCar<>‘ ‘
            ent numPal:=numPal+1
                Mq ¬finLinea(f)∧elCar<>‘ ‘
                    leer(f,elCar)
                FMq
          FSi
    FSi
  FMq
  dev(numPal)
Fin
```

TEMA 2: Introducción a la especificación y verificación de algoritmos.

1) Introducción

2) Especificación de algoritmos mediante predicados

- Fundamentos para la especificación
- LPPO y especificación de algoritmos
- Ejemplos y ejercicios

3) El transformador de predicados “`pmd`“

4) Semántica de un lenguaje imperativo

- Las instrucciones “**`seguir`**“ y “**`abortar`**“
- La asignación: simple y múltiple
- La composición secuencial
- La composición condicional: simple y múltiple
- La invocación a procedimientos
- La invocación a funciones
- Operadores sobre ficheros secuenciales

5) Introducción a la derivación de algoritmos

Especificación y verificación de algoritmos

- Cuando se hace un programa

  ¿Cómo se sabe realmente que es correcto?
- Hasta ahora, DEPURACION
  - búsqueda y eliminación de errores
- Pero
  - no asegura CORRECCION
    - salvo ¡¡Prueba exhaustiva!!
  - no ayuda al DISEÑO
  - aunque importante (baterías de test)
- En esta asignatura buscamos
  - especificar “formalmente” los programas: LO QUE DEBE HACER
  - verificar formalmente que REALMENTE hace lo que debe
  - derivar : usar la especificación para guiar la escritura del algoritmo

Especificación y verificación de algoritmos

- Distinguir:
  - **especificar**: qué hace
  - **implementar**: cómo lo hace
- ¿Cómo podemos especificar ?
  - lenguaje natural
    - » ambíguo
    - » “verbose”
  - fórmulas lógicas expresando las propiedades que han de verificar las variables y/o parámetros antes y después
- Especificaciones PRE/POST:
  - Precondición/Postcondición

Introducción a la especificación

- Sea A una acción

```
--Pre
  A
--Post
```

siendo Pre y Post **predicados** (¡¡Lógica!!) que dicen algo (¿?) sobre el **estado** (¿?)

- Aclaremos cosas:

`Estado de un programa`

`tupla de los valores de sus variables`

`Especificación sobre un estado`

`aserto sobre los valores de las variables`

Introducción a la especificación

- Ejemplo:

```
vars x,y:entero
principio
   .........
   .........
        --(x<0) ∨ (x>y²)
   .........
   .........
fin
```

- Algoritmo como caja negra: sus efectos SOLO se ven a través de los parámetros

```
algoritmo niSeSabe(E  x:....
                   S  y:....
                   ES z:....)
--Pre:  Q(x,z)
--Post: R(x,y,z)
```

Introducción a la especificación

- Significado

```
--Pre
   A
--Post
```

Pre toma el valor Cierto

Si

* al invocar la acción el estado verifica Pre

entonces

* la acción TERMINA

* el estado al que se llega la acción verifica el predicado Post

Post toma el valor Cierto

- Importante: no sabemos **cómo** llega al estado final

Un ejemplo de especificación

- Asumamos el entorno

```
tipos vect=vector[1..1000] de entero
```

- Se nos pide

```
funcion haySuma(E v:vect;E n:entero)
                         dev (hS:booleano)
```

  - tal que, dado un vector y el número de datos ocupados, determine si existe algún elemento igual a la suma de todos los anteriores
- Pero:
  - ¿Asumimos que n>=0?
  - ¿Asumimos que n<=1000?
  - ¿Qué pasa si hay más de un dato verificando la propiedad?
  - Si v[1]=0, ¿La propiedad es verificada?

Un ejemplo de especificación

- Especifiquemos mejor:

  Diseñar una función que tomando un dato de tipo vect y un entero en el rango 1..1000 determine si hay un elemento tal que es la suma de todos los anteriores.

  Si no hay ningún elemento que cumpla la propiedad, el primer elemento debe ser 0 para que la propiedad sea verificada.

  Por otra parte, si en el vector no hay elementos, entendemos que la propiedad no puede ser verificada por ningún elemento, por lo que la función debe devolver Falso.....

- Problema: es muy duro hacer las especificaciones así, por lo que buscamos otro “lenguaje”
  - más conciso
  - no ambíguo

Un ejemplo de especificación

- Más o menos, lo haremos así:

```
funcion haySuma(E v:vect;E n:entero)
                    dev (hS:booleano)
--Pre:  (0<=n)∧(n<=1000)
--Post: hS= (∃ α ∈{1..n}.
              v[α]= Σ β ∈{1..α-1}.v[β])
```

- Significado de la especificación
  - quien invoca a la función debe suministrarle los parámetros. Asumamos que se invoca

    `miRes := haySuma(miV,miN)`
  - Si al invocar se verifica que

    `0 <= miN <= 1000`
  - la invocación se termina de ejecutar, y el valor de miRes es equivalente al hecho de que en miV haya algún dato verificando la propiedad

Un ejemplo de verificación

- La especificación es fundamental para la verificación
- Ejemplo (intuitivo, de momento):

```
algoritmo swap(ES x,y:entero)
--Pre:  x=X ∧ y=Y
--Post: x=Y ∧ y=X
Principio
           --Pre:  x=X ∧ y=Y
  x := x+y
           --Q1:  x=X+Y ∧ y=Y
  y := x-y
           --Q2:  x=X+Y ∧ y=X+Y-Y
  x := x-y
           --Post: x=X+Y-X ∧ y=X
Fin
```

programa anotado ó esquema de demostración

- En lo que sigue:
  - herramientas para especificar
  - herramientas para verificar
  - ayudas para diseñar

Ejemplo de especificación y verificación

```
funcion modulo(E x:entero)
                   dev (vA:entero)
--Pre:  x=X
--Post: vA=|X|
Principio
    Si x<0
      ent                --P1: x=X ∧ x<0
          vA:=-x
                         --P2: x<0 ∧vA=-x
                         --P3:  vA=|x|
      si no              --P4: x=X ∧ x>=0
          vA:=x
                         --P3

    FSi
    dev(vA)              --P3
Fin
```

- Algún convenio:
  - daremos nombres a las aserciones para referirnos a ellas
  - dos aserciones seguidas, sin instrucción entre ellas, implica la deducción de la segunda a partir de la primera

Especificación de algoritmos mediante predicados

- Usaremos LPPO*
  - Lógica Proposicional de Primer Orden
  - lógica proposicional + cuantificadores*
- Especificación: FBF
  - fórmula bien formada

Las FBF

cualquier fórmula atómica es un FBF

* True,False

* una variable booleana

* cualquier expresión booleana

(P) siendo P una FBF

$\neg P$ siendo P una FBF

$P \wedge Q$ siendo P,Q FBF

$P \vee Q$ siendo P,Q FBF

$P \rightarrow Q$ siendo P,Q FBF

$P \leftrightarrow Q$ siendo P,Q FBF

$\forall \alpha \in D.P$ siendo P FBF

$\exists \alpha \in D.P$ siendo P FBF

D: dominio

Especificación de algoritmos mediante predicados

- Ejemplos de FBF
  - T
  - $(x>7.0) \wedge (x <= 26.9)$
  - $\exists\ i \in \{1..n\}.v[i]=0$
  - $(x \text{ MOD } 2=0) \wedge (x < 25)$
  - $(j=0) \rightarrow (x \text{ MOD } 2 =0)$

♦ representa un cuantificador

- Prioridades: $\neg\ \wedge\ \vee\ \rightarrow\ \leftrightarrow$ ♦
- Paréntesis: máxima prioridad
- Dos proposiciones E1 y E2 son **equivalentes** cuando E1=E2 es una tautología
  - tautología: siempre es cierto
  - lo denotaremos mediante “=“
- En lo que sigue, algunas equivalencias fundamentales

Especificación de algoritmos mediante predicados

- Leyes conmutativas

  $(E1 \wedge E2) = (E2 \wedge E1)$

  $(E1 \vee E2) = (E2 \vee E1)$

  $(E1 \leftrightarrow E2) = (E2 \leftrightarrow E1)$

- Leyes asociativas

  $E1 \wedge (E2 \wedge E3) = (E1 \wedge E2) \wedge E3$

- Leyes distributivas

  $E1 \wedge (E2 \vee E3) = (E1 \wedge E2) \vee (E1 \wedge E3)$

  $E1 \vee (E2 \wedge E3) = (E1 \vee E2) \wedge (E1 \vee E3)$

- Leyes de Morgan

  $\neg(E1 \wedge E2) = \neg E1 \vee \neg E2$

  $\neg(E1 \vee E2) = \neg E1 \wedge \neg E2$

- Ley de la negación

  $\neg(\neg E)=E$

- Ley del “medio” imposible

  $E \vee \neg E = T$

- Ley de la contradicción

  $E \wedge \neg E = F$

Especificación de algoritmos mediante predicados

- Ley de la implicación

  $E1 \rightarrow E2 = \neg E1 \vee E2$

- Ley de la igualdad

  $(E1 \leftrightarrow E2) = (E1 \rightarrow E2) \wedge (E2 \rightarrow E1)$

- Leyes de simplificación del OR

  $E1 \vee E1 = E1$

  $E1 \vee T = T$

  $E1 \vee F = E1$

  $E1 \vee (E1 \wedge E2) = E1$

- Leyes de simplificación del AND

  $E1 \wedge E1 = E1$

  $E1 \wedge T = E1$

  $E1 \wedge F = F$

  $E1 \wedge (E1 \vee E2) = E1$

Especificación de algoritmos mediante predicados

- Leyes de ∀ y ∃

$(\forall \alpha \in D.P)=T$ si $D=\emptyset$

$(\exists \alpha \in D.P)=F$ si $D=\emptyset$

$\neg(\forall \alpha \in D.P)=(\exists \alpha \in D.\neg P)$

$\neg(\exists \alpha \in D.P)=(\forall \alpha \in D.\neg P)$

♦ representa un cuantificador

$(\blacklozenge \alpha \in D.P)=(\blacklozenge \beta \in D.P[\beta/\alpha])$ Si $\beta \notin$variables(P)

$\blacklozenge \alpha \in D_1.\ \blacklozenge \beta \in D_2.P=\blacklozenge \beta \in D_2.\ \blacklozenge \alpha \in D_1.P$

- <u>Reglas de inferencia</u>: generan nuevas equivalencias
- Forma:

$$\boxed{\frac{E_1,\ldots,E_n}{E}}$$

Especificación de algoritmos mediante predicados

- Regla 1: de la **transitividad**

$$\frac{P_1=P_2, P_2=P_3}{P_1=P_3}$$

- Regla 2: de **sustitución**

$$\frac{P_1=P_2}{Q(P_1)=Q(P_2)}$$

- En resumen:

LEYES

+

REGLAS INFERENCIA

=

Cálculo de Predicados

Especificación de algoritmos mediante predicados

- En LPPO* aparecen dos clases de variables:
  - no controladas por un cuantificador
    - **libres**
    - asociadas a variables de programa
  - controladas por un cuantificador
    - **ligadas** (**mudas**)
    - asociadas a “recorridos” de posibles valores de variables de programas
- Siendo E una FBF, denotamos
  - **`libres(E)={variables libres de E}`**
  - **`ligadas(E)={variables ligadas de E}`**
- Ejemplo:

$$E=\forall i\in\{n..m\}.x*i>0$$

  - **`libres(E)={m,n,x}`**
  - **`ligadas(E)={i}`**

Especificación de algoritmos mediante predicados

- Más formalmente [Peña 93] :

  ligadas(E) y libres(E)

  - libres(E)=variables(E),
    ligadas(E)=∅ si E es un átomo
  - libres(¬E)=libres(E),
    ligadas(¬E)= ligadas(E

♥ representa un conector binario

  - libres(E1♥E2)=libres(E1) ∪
    libres(E2)
    ligadas(E1♥E2)=ligadas(E1) ∪
    ligadas(E2)

♦ representa un cuantificador

  - libres(♦α∈D.E)=libres(E)\{α}

    ligadas(♦α∈D.E)=ligadas(E)∪{α}

Especificación de algoritmos mediante predicados

- Ejemplo:

$$\underbrace{\forall\alpha\in\{1..n\}.\underbrace{\exists\beta\in\{1..\beta\}.\underbrace{(\alpha\neq\beta\wedge v(\alpha)=v(\beta))}_{R}}_{Q}}_{P}$$

  - libres(R)={v,α,β}, ligadas(R)={}
  - libres(Q)={v,α}, ligadas(Q)={β}
  - libres(P)={v,n}, ligadas(P)={α,β}

- Propiedad [Gries 78]: Sea E un predicado
  - libres(E) ∩ ligadas(E)=∅
  - una misma variable NO puede aparecer simultáneamente en dos cuantificadores

NO	$(\forall\alpha\in\{m..n\}.x*\alpha>0)\wedge(\forall\alpha\in\{1..5\}.y*\alpha<0)$
SI	$(\forall\alpha\in\{m..n\}.x*\alpha>0)\wedge(\forall\beta\in\{1..5\}.y*\beta<0)$

Especificación de algoritmos mediante predicados

- Determinar si son válidas o no los siguientes predicados. Establecer las variables libres y ligadas en cada caso

1) $(2 \leq m) \wedge (m \leq n-1) \wedge (\forall \alpha \in \{2..m-1\}.m \text{ DIV } \alpha \neq 0)$

2) $(2 \leq m) \wedge (m \leq n-1) \wedge (\forall n \in \{2..m-1\}.m \text{ DIV } n \neq 0)$

3) $(\exists \alpha \in \{1..24\}.25 \text{ DIV } \alpha \neq 0) \wedge (\exists \alpha \in \{1..24\}.25 \text{ MOD } \alpha = 0)$

5) $\forall \alpha \in \{n+1..n+5\}.\exists \beta \in \{2..m-1\}.m \text{ DIV } \beta = 0$

6) $\forall \alpha \in \{n+1..n+5\}.\exists n \in \{2..m-1\}.m \text{ DIV } n = 0$

Especificación de algoritmos mediante predicados

- Buscamos mayor flexibilidad para escribir predicados.
- Sea P un predicado:
  - **renombramiento**: sea y∉variables(P)

$$P[y/x]$$

todas las apariciones de x se sustituyen SIMULTANEAMENTE por y

  - **sustitución**: sea x libre en P; sea E una expresión del mismo tipo que x

$$P_x^E$$

En E sólo libres de P

denota el predicado resultante de sustituir SIMULTANEAMENTE todas las apariciones de x por la expresión E

Especificación de algoritmos mediante predicados

`x`$_1$`,...,x`$_n$ $\in$ `libres(P)`

– **sustitución múltiple:**

$$\boxed{P^{E1,\dots,En}_{x1,\dots,xn}}$$

- Ejemplo 1: $E=(x<y) \wedge (\forall\alpha\in\{6..20\}.y \text{ DIV } \alpha=0)$

1) $E^{z}_{x}=(z<y) \wedge (\forall\alpha\in\{6..20\}.y \text{ DIV } \alpha=0)$

2) $E^{x+y}_{y}=(x<(x+y)) \wedge (\forall\alpha\in\{6..20\}.(x+y) \text{ DIV } \alpha=0)$

3) $E^{\beta}_{\alpha}=E$ (lo definimos, pues sólo para libres)

4) $(E^{wz}_{y})^{a+u}_{z}=((x<wz) \wedge (\forall\alpha\in\{6..20\}.wz \text{ DIV } \alpha=0))^{a+u}_{z}=$

$(x<w(a+u)) \wedge (\forall\alpha\in\{6..20\}.w(a+u) \text{ DIV } \alpha=0)$

- Ejemplo 2: $P=x + x + y < 7$

$P^{x+y,z}_{x,y} = x+y+x+y+z<7 \neq$

$(P^{x+y}_{x})^{z}_{y} = x+z+x+z+z$

LPPO y especificación de algoritmos

- **Objetivo**: aplicar los conceptos presentados a la especificación de algoritmos
- **Nomenclatura**: Sea un programa
  - **ID** es el conjunto de identificadores de las variables/parámetros declarados en el algoritmo
  - sea $x \in$ **ID**. $\mathbf{D}_x$ denota el "dominio semántico" de x, al que se ha añadido un valor $\perp_x$ para el caso en que el valor de x esté indeterminado
- **Espacio de estados** del programa:

$$\boxed{\varepsilon = \Pi_{x \in ID} D_x}$$

- **Estado** de un programa:

tupla/vector de valores

$$\boxed{\sigma \in \varepsilon}$$

LPPO y especificación de algoritmos

- Para hablar de propiedades de programas, hablaremos de predicados verificados por sus estados
- Sea σ un estado y P un predicado

P está bien definido

$\forall x \in variables(P).x \neq \bot_x$

evaluación de P en σ

sustitución de átomos por valores, aplicación de prioridades, evaluación de funciones invocadas,...

Lo denotaremos como: $\| P \| (\sigma)$ $(\in\{F,T\})$

σ satisface P cuando

$\| P \| (\sigma) = T$

Lo denotatemos como: $\sigma \models P$

del estado σ se deduce **P**

LPPO y especificación de algoritmos

- Alguna definición más (¡!)
  - P es <u>universalmente válido</u> ssi

$$\forall \sigma \in \mathcal{E},\ \ \sigma \models P$$

  - P es una <u>contradicción</u> ssi

$$\forall \sigma \in \mathcal{E},\ \ \sigma \not\models P$$

  - Q es <u>consecuencia lógica</u> de P ssi

$$\forall \sigma \in \mathcal{E},\ \ \sigma \models P$$

implica que

$$\sigma \models Q$$

Se denota como $P \models Q$

LPPO y especificación de algoritmos

- Sea P un predicado. Definimos:

`estados(P)={ σ∈Ɛ | σ |= P }`

- Entonces:

`P es más fuerte que Q`
`“restringe” más`

`P |= Q   ssi  estados(P) ⊆ estados(Q)`

- Ejemplo. Asumamos el entorno
  - `var x,y:integer`
  - $P = (x > 0) \wedge (x-y > 27)$
  - $Q = (x > -3) \wedge (x-y > 27)$
  - Tenemos: $\mathcal{E} = Z^2$ (¡implementación!)
  - Todo estado que haga cierto P hace también cierto Q: P es más fuerte
  - Luego P |= Q

Especificación de algoritmos

- Ya casi estamos listos para especificar algoritmos. Falta un poquito....
- Algunas convenciones y "cuantificadores" específicos (no son de la LPPO)
  - $\boxed{\{a..b\}}$ denota
    - $\emptyset$ si a > b
    - $\{a,a+1,a+2,\ldots,b-1,b\}$ si a <= b
  - $\boxed{\sum \alpha \in \{a..b\}.E(\alpha)}$ denota
    - 0 si a > b
    - $\sum_{\alpha=a}^{b} E(\alpha)$ si $a \leq b$
  - $\boxed{\prod \alpha \in \{a..b\}.E(\alpha)}$ denota
    - 1 si a > b
    - $\prod_{\alpha=a}^{b} E(\alpha)$ si $a \leq b$

E es una expresión

Especificación de algoritmos

P es un predicado

- El último:
  - $\boxed{N\ \alpha \in \{a..b\}.P(\alpha)}$ denota
    - » 0 si $a > b$
    - » $\#(\ \{ \beta \in \{a..b\} \mid P(\beta)\}\ )$ si $a <= b$

Especificación de un algoritmo A

A
--Q
--R
donde:

También
{Q} A {R}

* A es la cabecera de un procedimiento o función (con sus parámetros E,S,ES)

* Q es un predicado (Precondición)
Sus únicas variables libres son los parámetros de tipo **E,ES, ~~S~~**

* R es un predicado (Postcondición)
Sus únicas variables libres son los parámetros (de cualquier clase)

Especificación de algoritmos
Ejemplos

```
Constantes dim=....       -->=1
Tipos vect=vector[1..dim]de entero
```

1)
```
funcion dividir (E dividendo,
                   divisor:entero)
                        dev (coc,rest:entero)
--coc: cociente división entera
--rest: resto división entera
```

2)
```
funcion raizCuadEntera (E n:entero)
                             dev (r:entero)
--r: raíz cuadrada entera
```

3)
```
funcion maxV(E v:vect) dev (max:entero)
--max: valor máximo de v
```

4)
```
funcion posPriMax(E v:vect) dev (pPM:entero)
--pPM: posición de la primera aparición del
--     máximo valor del vector
```

5)
```
funcion max(E a,b:entero) dev (eM:entero)
--eM: el máximo de {a,b}
```

6)
```
funcion estaEnV(E v:vect;E n:entero)
                            dev (eV:booleano)
--eV: ¿Es n uno de los elementos de v?
```

Especificación de algoritmos
Ejemplos

```
Constantes dim=....      -->=1
Tipos vect=vector[1..dim]de entero
```

7)
```
funcion esGuay(E n:entero) dev (eG:booleano)
--eG: ¿Es n “guay”? Un número es guay cuando
--    es la suma 1+2+...+k para cierto k
```

8)
```
funcion esPermutación(E v:vect)
                           dev(eP:booleano)
--eP: ¿Contiene v una permutación de los
--    los números 1,2,...,dim?
```

9)
```
funcion mcd(E a,b:entero) dev (gcd:entero)
--gcd=máximo común divisor de (a,b)
```

10)
```
algoritmo sustit(ES v:vect;E x,y:entero)
--Sustituye las apariciones de x por y
```

11)
```
algoritmo invierte(ES v:vect)
--Invierte el vector v
```

12)
```
algoritmo insertaOrd(ES v:vect;E nD,x:entero)
--Suponiendo v con nD datos ordenados no
--dec., y cabiendo uno más, inserta x en la
--posición que le corresponde
```

Especificación de algoritmos
Ejemplos

```
Tipos dato=Registro
           codigo:      entero
           transaccion: real
      Freg
      fichDato=fichero de dato
```

13)
```
funcion estadoC(E cod:entero;E fD:fichDat)
                               dev (e:real)
--Q: se asume “fD” abierto para lectura y el
--   el buffer sobre el primer dato
--R: “e” da el estado de la cuenta del
--   cliente cuyo código es “cod”
```

14)
```
funcion estadoC2(E cod:entero;
                 E nF:string) dev (e:real)
--Q: se asume “nF” como el nombre de un
--   fichero físico con datos de tipo “dato”
--R: “e” da el estado de la cuenta del
--   cliente cuyo código es “cod”
```

Especificación de algoritmos
Ejemplos

- Considerar el entorno de datos

```
Constantes m=.....     --m >= 2
          n=.....     --1 <= n < m
Tipos  vect=vector[1..n] de entero
       fichEnt=fichero de entero
```

- Especificar formalmente y diseñar el siguiente algoritmo, teniendo en cuenta que la complejidad asintótica en tiempo ha de ser **O(m)**

```
algoritmo escribeFallos(E v:vect;ES f:fichEnt)
--Pre: f=(<>,1,E) y v contiene n enteros,
--     distintos, del conjunto {1.. m}
--Post: f=(<d1,..., dm-n >,m-n+1,E)
--      y d1,...,dm-n son los enteros
--      del conjunto {1..m} que no
--      están en el vector v
```

- ¿Es el algoritmo propuesto Θ(m)?

Especificación de algoritmos Ejemplos

- Mismo entorno de datos
- Especificar y diseñar:

```
funcion mayorCola(E v,v':vect)
                          dev (pM,s:entero)
--Pre: True
--Post: pM es la primera posición del vector
--      v, empezando por 1, tal que todos
--      los elementos de v y v', desde
--      pM hasta n, coinciden. Por otra
--      parte, s es la suma de dichos
--      elementos. Tener presente que puede
--      no haber ninguna coincidencia
```

- Calcular su complejidad

Especificación de algoritmos
Ejemplos

- Se representan vuelos entre ciudades como sigue

Vuelos directos

Destino

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

Número de vuelos con una escala

Destino

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

```
Constantes n=.....   --n>=2
Tipos matVuelos=vector[1..n,1..n] de entero
      matPuentes=vector[1..n,1..n] de entero
          --vuelos con una escala intermedia}
```

Especificación de algoritmos
Ejemplos

- Especificar formalmente y diseñar:

```
funcion hayPuente(E m:matVuelos;
      cOr,cInt,cDes:entero) dev (hV:booleano)
--Pre: 1<=cOr<=n ∧ 1<=cInt<=n ∧ 1<=cDes<=n
--Post: hV indica si hay un vuelo de cOr a
--      cDes haciendo transbordo en cInt
```

```
funcion numPuentes(E m:matVuelos;
         E cOr,cDes:entero) dev (nP:entero)
--Pre: 1<=cOr<=n ∧ 1<=cDes<=n
--Post: nP=número de formas de llegar de cOr
--       a cDes con una escala intermedia
```

```
algoritmo numPuentesGlobal(E m:matVuelos;
                           S mP:matPuentes)
--Pre: True
--Post:cada elemento mP[i,j] indica el
--     número de formas de llegar de i a j
--     haciendo una escala intermedia
```

El transformador de predicados “pmd”

- Primer objetivo: definir un lenguaje de programación que permita razonar sobre programas
  - Tomaremos el lenguaje algorítmico manejado hasta ahora
- Fundamental: establecer claramente su **semántica**
  - ¿Qué hacen sus instrucciones? ó
  - ¿Cómo actúan sus instrucciones? ó
  - ¿**Cómo transforman el estado del programa sus instrucciones**?
- Se definirá a través del transformador de predicados “pmd”
  - “Precondición Más Débil”

El transformador de predicados “pmd”

pmd(A,R)

* **A**: acción
* **R**: predicado sobre el estado

* “**pmd(A,R)**“ es el conjunto de todos los estados tal que, si la ejecución de **A** comienza en uno de ellos, **A** termina de ejecutarse en un tiempo finito y, 1además, termina en un estado que verifica **R**.

- ¿Por qué el nombre de “pmd”?
  - recordar el significado de

```
--Q
   A
--R
```

  - notar que de las propias definiciones se deduce inmediatamente que si {Q} A {R}, entonces

Q |= pmd(A,R)

El transformador de predicados "pmd"

- Sea A la acción "**`i:= i+1`**" y sea **`R=(i<=1)`**
  - *`pmd(S,R)=(i<=0)`*
  - Además, si **`Q=(i<=-4)`**, entonces

    `{Q} A {R}`

    pues Q es más fuerte que *`pmd(A,R)=(i<=0)`*
- Sea **`R= (z=máx(x,y))`**; sea A=

```
Si x>=y
    ent   z:=x
    si_no z:=y
FSi
```

`pmd(A,R)=True`

- Mismo A, **`R= (z=y)`**

`pmd(A,R)= x<=y`

- Mismo A, **`R= (z=y-1)`**

`pmd(A,R)=F`

- Mismo A, **`R= (z=y+1)`**

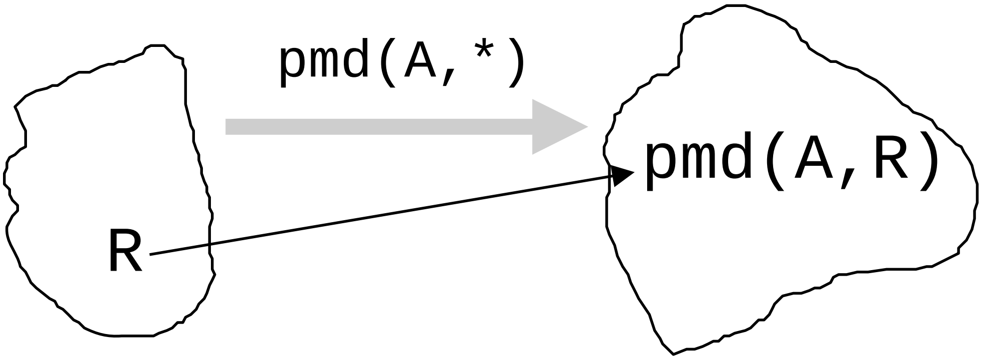
`pmd(A,R)=(x=y+1)`

- A cualquiera, **`R= True`**

  ¿Qué conjunto de estados denota "**`pmd(A,R)`**"?

El transformador de predicados “pmd”

- Fijemos una acción “A”:



“<u>transformador de predicados</u>”

- Importante:
  - En general, las especificaciones serán
    `{Q} A {R}`
  - La corrección se asegura probando que
    $$\mathbf{Q \Rightarrow pmd(A,R)}$$
  - Pero no siempre será necesario obtener **`pmd(A,R)`**, pues en muchos casos se podrá probar directamente
  - Sin embargo, en otros casos será necesario establecer el propio Q

El transformador de predicados "pmd"

- Leyes fundamentales del "**pmd**"
  - Ley del milagro imposible

$$\boxed{\mathtt{pmd(S,F)=F}}$$

  - Distributividad de la conjunción

$$\boxed{\mathtt{pmd(S,Q)} \wedge \mathtt{pmd(S,R)=pmd(S,Q} \wedge \mathtt{R)}}$$

  - Distributividad de la disyunción

$$\boxed{\mathtt{pmd(S,Q)} \vee \mathtt{pmd(S,R)=pmd(S,Q} \vee \mathtt{R)}}$$

  - Ley de monotonía

$$\boxed{\begin{array}{l}\mathtt{Si\ Q} \Rightarrow \mathtt{R\ entonces} \\ \qquad \mathtt{pmd(S,Q)} \Rightarrow \mathtt{pmd(S,R)}\end{array}}$$

Las instrucciones "seguir" y "abortar"

- Objetivo: dar la semántica del lenguaje algorítmico que manejamos
- La instrucción "**`seguir`**"
  - no hace nada (inst. vacía de Pascal, p.e.)
  - transformador identidad
  - Sintaxis: `seguir`
  - Semántica: `pmd(seguir,R)=R`
- La instrucción "**`abortar`**"
  - nunca es ejecutable (no hay estado que verifique la precondición)
  - representa una interrupción del cálculo o un tiempo infinito de cálculo
  - útil para casos de errores en tiempo de ejecución
  - Sintaxis: `abortar`
  - Semántica `pmd(abortar,R)=F`

La instrucción de asignación

- La asignación
  - `x` variable, `E` expresión del mismo tipo
  - `Dom(E)` conjunto de estados en que `E` está definida
  - Sintaxis: $\boxed{\mathtt{x\ :=\ E}}$
  - Semántica:

$$\boxed{\mathtt{pmd(x\ :=\ E,R)=Dom(E)} \wedge \mathtt{R}_{\mathtt{x}}^{\mathtt{E}}}$$

  - Recordar significado de $\mathtt{R}_{\mathtt{x}}^{\mathtt{E}}$
  - Principio de limpieza: cuando `E` sea una expresión parcial (hay estados en los que no está definida) hay que poner el dominio de la expresión en la precondición
  - Por comodidad, y si no hay confusión, se omitirá por lo general `Dom(E)`

La instrucción de asignación

- <u>Ejemplos</u>

¿Dominios?

1) $pmd(x:=5, x=5) = (5=5) = T$

2) $pmd(x:=5, x \neq 5) = (5 \neq 5) = F$

3) $pmd(x:=x+1, x<0) = (x+1<0) = (x<-1)$

4) $pmd(x:=x*x, x^4=10) = (x^8=10)$

5) $pmd(x:=A \text{ DIV } B, P(x)) =$

$$(B \neq 0) \wedge (P(A \text{ DIV } B))$$

6) $pmd(x:=e, x=c) = (e=c)$

7) $pmd(x:=e, y=c) = (y=c)$

La instrucción de asignación múltiple

- Usaremos asignación múltiple
  - Sintaxis

$$\boxed{\texttt{<}x_1,\ldots,x_n\texttt{>:=<}E_1,\ldots,E_n\texttt{>}}$$

  - Semántica

$$\boxed{\begin{array}{l}\mathbf{pmd(}\texttt{<}x_1,\ldots,x_n\texttt{>:=<}E_1,\ldots,E_n\texttt{>}\mathbf{,R)=} \\ \quad \mathrm{Dom}(E_1)\wedge\ldots\wedge\mathrm{Dom}(E_n)\wedge\ \mathbf{R}_{x_1,\ldots,x_n}^{E_1,\ldots,E_n}\end{array}}$$

- ¡OJO! No es lo mismo

	`<x,y>:=<y,x>`
que	`x:=y ; y:=x`
ni que	`y:=x ; x:=y`

- El funcionamiento es como sigue:
  - evaluar simultáneamente las expresiones de la derecha
  - realizar las **n** asignaciones simultáneamente

La instrucción de asignación

- Ejemplos

1) `{Q= x=a}` **`q:=0`** `{R= qy+x=a}`

```
2)  --Q= qy+x=a
          x:=x-y
    --R= qy+x+y=a
```

```
3)  --Q= qy+x+y=a
          q:=q+1
    --R= qy+x=a
```

- Ejercicios: obtener (un) Q en

`{Q}` **`x:=x+1`** `{R}`

para R=

1) $x=7$
2) $x+y>0$
3) $y=2^k$
4) $\exists\ x\geq 0.\ y=2^x$

La composición secuencial

- La asignación será la única instruccion básica para modificar los estados
- Los programas serán composiciones de asignaciones
- Recordar esquemas de composición:
  - secuencial
  - alternativos (simple o múltiple)
  - iterativo
- <u>Siguiente paso</u>: estudiar los “`pmd`” para cada una de las composiciones
- Para cada esquema de composición recordaremos su “sintaxis” (relajada) y daremos una semántica formal

La composición secuencial

- Se utiliza para encadenar cálculos
- Sean S_1 y S_2 dos acciones
  - Sintaxis:

$$\boxed{S_1;S_2} \qquad \boxed{\begin{array}{c}S_1\\ S_2\end{array}}$$

  - Semántica:

$$\boxed{\mathbf{pmd}(S_1;S_2,\mathbf{R})=\mathbf{pmd}(S_1,\mathbf{pmd}(S_2,\mathbf{R}))}$$

- Por asociatividad de predicados, se puede generalizar a tres acciones, cuatro acciones, etc.

$$\boxed{\begin{array}{l}\mathbf{pmd}(S_1;S_2;S_3,\mathbf{R})\ \mathbf{=}\\ \mathbf{pmd}((S_1;S_2);S_3,\mathbf{R})=\\ \mathbf{pmd}(S_1;(S_2;S_3),\mathbf{R})\end{array}}$$

La composición secuencial

- Aplicación a la corrección:

```
algoritmo swap(ES x,y:entero)
--Pre:  x=X ∧ y=Y
--Post: x=Y ∧ y=X
Principio
                     --Pre:  x=X ∧ y=Y
    x := x+y
    y := x-y
    x := x-y
                     --Post: x=Y ∧ y=X
Fin
```

Demostración:

- $pmd(x:=x-y, x=Y \wedge y=X)=(x-y=Y) \wedge (y=X)$
- $pmd(y:=x-y, x-y=Y \wedge y=X) =$

  $(x-x-y=Y) \wedge (x-y=X) = (y=Y) \wedge (x-y=X)$
- $pmd(x:=x+y, (y=Y) \wedge (x-y=X)) =$
  $(y=Y) \wedge (x+y-y=X)=(y=Y) \wedge (x=X)$
- Como Pre ⇒ el último predicado (es igualdad) se deduce la corrección del algoritmo

La composición secuencial

- Para verificaciones, es útil hacerlas por partes
- Para ello, se aplica la regla de la comp. secuencial:

$$\frac{\{P\}S_1\{Q\},\{Q\}S_2\{R\}}{\{P\}\ S_1;S_2\ \{R\}}$$

interpretación

Si S_1 se ejecuta bajo la precondición **P**, se garantiza **Q** al terminar.
Si además ejecutando S_2 cuando **Q** es cierto se asegura al terminar **R**,
entonces la ejecución de $S_1;S_2$ cuando **P** es cierto asegura la terminación en un estado que verifica **R**

La composición secuencial

- Ejercicio: Probar la corrección de

```
            --Q: x=X ∧ y=Y
t:=x
x:=y
y:=t
            --R: x=Y ∧ y=X
```

- Forma 1: Calcular la “`pmd`” y probar que es deducible de Q
- Forma 2: “Aventurar” predicados intermedios y aplicar la regla de la composición secuencial
- En cualquier caso:
  - Escribirlo en forma de “programa anotado”

La composición condicional

- En función a una expresión booleana, ejecuta una acción u otra
- Sean A_1 y A_2 dos acciones, y sea `E` una expresión booleana
  - Sintaxis:

```
Si E
   ent   A₁
   si_no A₂
FSi
```

  - Semántica:

$$\mathbf{pmd(Si}\ E\ \mathbf{ent}\ A_1\ \mathbf{si\_no}\ A_2\ \mathbf{FSi,R)=}\\ \quad \mathrm{Dom}(E)\wedge\\ \quad ((E\rightarrow \mathrm{pmd}(A_1,R))\wedge(\neg E\rightarrow \mathrm{pmd}(A_2,R)))$$

  - o, lo que es lo mismo

$$\mathbf{pmd(Si}\ E\ \mathbf{ent}\ A_1\ \mathbf{si\_no}\ A_2\ \mathbf{FSi,R)=}\\ \quad \mathrm{Dom}(E)\wedge\\ \quad ((E\wedge \mathrm{pmd}(A_1,R))\vee(\neg E\wedge \mathrm{pmd}(A_2,R)))$$

La composición condicional

- En términos de algoritmos

```
--$Dom(E) \wedge ((E \rightarrow pmd(A_1,R))$
--          $\wedge (\neg E \rightarrow pmd(A_2,R) )$

Si E
   ent      --$E \rightarrow pmd(A_1,R)$
       $A_1$
            --R
   si_no    --$\neg E \rightarrow pmd(A_2,R)$
       $A_2$
            --R
FSi
--R
```

```
--$Dom(E) \wedge (  (E \wedge pmd(A_1,R))$
--               $\vee (\neg E \wedge pmd(A_2,R))$

Si E
   ent      --$E \wedge pmd(A_1,R)$
       $A_1$
            --R
   si_no    --$\neg E \wedge pmd(A_2,R)$
       $A_2$
            --R
FSi
--R
```

La composición condicional

- <u>Ejemplo</u> 1: diseñar y verificar la siguiente función:

```
funcion valAbs(E n:entero) dev (vA:entero)
--Pre: True
--Post: vA=|n|
```

- <u>Ejemplo</u> 2: Encontrar <u>una</u> precondición Q para el siguiente trozo de algoritmo

```
--¿Q?
Si x<0
   ent     x:=x+1
   si_no   x:=x-1
FSi
--R: x >= 0
```

La composición condicional

- Asumiendo una precondición Q dada, la semántica se puede establecer mediante la siguiente regla

$$\frac{\{Q\wedge E\}A_1\{R\},\{Q\wedge\neg E\}A_2\{R\}}{\{Q\wedge Dom(E)\}\ \mathbf{Si}\ E\ \mathbf{ent}\ A_1\ \mathbf{si\_no}\ A_2\ \mathbf{FSi}\ \{R\}}$$

- <u>Ejemplo</u> 3: Siendo
  `v:vector[1..n] de entero`
  probar la corrección de

```
--Q:a>0
Si a>b
   ent      m:=a
   si_no    m:=b
FSi
--R: m>0 ∧ m=máx{a,b}
```

La composición condicional

- Ejemplo 4: Verificar la corrección del siguiente algoritmo

```
--Q: y>0 ∧ z+xy=A*B
Si impar(x)
   ent    <z,x>:=<z+y,x-1>
   si_no  seguir
FSi
<y,x>:=<2*y,x DIV 2>
--R: y>=0 ∧ z+xy=A*B
```

- A propósito, ¿Semántica del condicional "degenerado" ?

La composición condicional múltiple

– Sintaxis:

Sel
E_1: S_1
......
E_n: S_n
FSel

– Semántica:

$BB=E_1 \vee....\vee E_n$
$UNO(BB) = (\mathbf{N}\alpha\in\{1,..,n\}.E_\alpha)=1$

pmd(Sel E_1:S_1 ...E_n:S_n **FSel,R)=**
$Dom(BB) \wedge BB \wedge UNO(BB) \wedge ($
$(E_1 \rightarrow pmd(S_1,R)) \wedge ... \wedge (E_n \rightarrow pmd(S_n,R))$
$)$

- ¿Qué significa exactamente?
- Hay otras semánticas alternativas

Aunque, en realidad,
$UNO(BB) \rightarrow BB$

La composición condicional múltiple

- Ejemplo. `Se sabe que v1 y v2 están ordenados ↑ y tienen al menos un valor común X. Obtener un Q que haga correcto`

```
v1,v2:vector[1..n] de entero
--Q
Sel
 v1[i]<v2[j]: i:=i+1
 v1[i]=v2[j]: seguir
 v1[i]>v2[j]: j:=j+1
FSel
--R: ordenado(v1)∧ordenado(v2)∧
--               v1[i]<=X ∧v2[j]<=X
```

- Asumiendo una precondición Q, la semántica se puede establecer mediante

$$\frac{\{Q\wedge E_1\}S_1\{R\},\ldots,\{Q\wedge E_n\}S_n\{R\}}{\{Q\wedge Dom(BB)\wedge UNA(BB)\}\,\mathbf{Sel}\ E_1{:}S_1\ \ldots E_n{:}S_n\ \mathbf{FSel}\{R\}}$$

Invocación a procedimientos

- Un procedimiento es la abstracción de una acción: **acción virtual**
- Sintaxis de la declaración

```
Algoritmo miAlg(E x:tX;ES y:tY;S z:tZ)
--Pre:  Q(x,y)
--Post: R(x,y,z)
      A
```

  - Barras: representan listas de parámetros
  - Por simplicidad, a veces manejaremos sólo parámetros de E y de ES

```
Algoritmo miAlg(E x:tX;ES y:tY)
--Pre:  Q(x,y)
--Post: R(x,y,z)
      A
```

- Invocación:

```
--....

miAlg(a,b,c)

-- R(a,b,c)
```

Invocación a procedimientos

- Semántica

$$
\begin{array}{l}
\texttt{pmd(miAlg(}\underline{\texttt{a}}\texttt{,}\underline{\texttt{b}}\texttt{,}\underline{\texttt{c}}\texttt{),R)=}\\
\quad\texttt{pmd(}\quad \texttt{<}\underline{\texttt{x}}\texttt{,}\underline{\texttt{y}}\texttt{>:=<}\underline{\texttt{a}}\texttt{,}\underline{\texttt{b}}\texttt{>;}\\
\qquad\qquad\quad \texttt{A ;}\\
\qquad\qquad\quad \texttt{<}\underline{\texttt{b}}\texttt{,}\underline{\texttt{c}}\texttt{>:=<}\underline{\texttt{y}}\texttt{,}\underline{\texttt{z}}\texttt{>,}\\
\qquad\qquad\quad \texttt{R}\\
\quad\texttt{)}
\end{array}
$$

- Lo mismo, de otra forma:

$$
\texttt{pmd(miAlg(}\underline{\texttt{a}}\texttt{,}\underline{\texttt{b}}\texttt{,}\underline{\texttt{c}}\texttt{),R)=}\\
\texttt{(pmd((A,R}^{\underline{\texttt{y}},\underline{\texttt{z}}}_{\underline{\texttt{b}},\underline{\texttt{c}}}\texttt{)))}^{\underline{\texttt{a}},\underline{\texttt{b}}}_{\underline{\texttt{x}},\underline{\texttt{y}}}
$$

- Ejemplo: Calcular

pmd(raro(a,b),a>=**0** ∧ **b**>=**0)**

```
algoritmo raro(E x:entero; ES y:entero)
Vars z:entero
Principio
   z:=2*x
   y:=y-3*z
Fin
```

Invocación a procedimientos

- La invocación no introduce complejidad :
  - bastaría con sustituir la invocación por el código que genera y asignaciones de parámetros
- El siguiente teorema es de ayuda para la verificación cuando hay invocación

Si

“**algoritmo** miAlg(**E** $\underline{x}$:$\underline{tX}$;**ES** $\underline{y}$:$\underline{tY}$;**S** $\underline{z}$:$\underline{tZ}$)

--Pre: Q($\underline{x}$,$\underline{y}$)

--Post:R($\underline{x}$,$\underline{y}$,$\underline{z}$)}

A

“

miAlg($\underline{a}$,$\underline{b}$,$\underline{c}$)

--S($\underline{a}$,$\underline{b}$,$\underline{c}$)

es correcto (es decir, $Q \rightarrow pmd(A,R)$)

Entonces

$$Q^{\underline{a},\underline{b}}_{\underline{x},\underline{y}} \wedge (\forall u,v. R^{\underline{u},\underline{v}}_{\underline{y},\underline{z}} \rightarrow S^{\underline{u},\underline{v}}_{\underline{b},\underline{c}}\ ) \rightarrow pmd(miAlg(\underline{a},\underline{b},\underline{c}),S)$$

Invocación a procedimientos

- Asumamos

```
algoritmo miAlg(E x:tX;ES y:tY;S z:tZ)
--Pre:  Q(x,y)
--Post: R(x,y,z)
       A
```

Teorema

Considerar **miAlg($\underline{a}$,$\underline{b}$,$\underline{c}$)** ($\underline{b}$,$\underline{c}$ distintos))
Sea **I** un predicado que no depende de $\underline{b}$,$\underline{c}$.
Entonces

invariante

$$\{Q^{\underline{a},\underline{b}}_{\underline{x},\underline{y}} \wedge I\}\ \mathbf{miAlg}(\underline{a},\underline{b},\underline{c})\ \{R^{\underline{a},\underline{b},\underline{c}}_{\underline{x},\underline{y},\underline{z}} \wedge I\}$$

OJO: no podemos asignar nada a $\underline{x}$

Invocación a procedimientos

- Ejercicio 1: Verificar

```
algoritmo cambia(ES s,t:entero)
--QC: s=S ∧ t=T
--RC: s=T ∧ t=S
```

```
--Q: s1=A ∧ s2=B ∧ s3=A+B

cambia(s1,s2)

--R: s1=B ∧ s2=A ∧ s3=A+B
```

Ejercicio 2: Verificar

```
algoritmo eleva(E x,n:entero;S e:entero)
--QE: n>=0
--RE: e=xⁿ
```

```
--Q: x₀=A

   s:=1

   eleva(x₀,8,s')

   s:=s + s'

--R: s= A⁸ +1
```

Invocación a funciones

- Algunas normas de “buena educación”
  - no usar variables globales
  - en una invocación, los parámetros actuales han de tener distintos nombres...
  - además, parámetros act/formales distintos
- Sintaxis declaración de función

```
Funcion miFunc(E x:tX) dev (r:tR)
{Pre(x)} A  {Post(x,r)}
```

- Semántica
  - Asumamos un proc. asociado a la función:

```
Algoritmo miProc_f(E x:tX; S r:tR)
{Pre(x)} A  {Post(x,r)}
```

  - y sea una instrucción `I(f(a))`

```
pmd(I(f(a)),R)=
            pmd(miProc_f(a,TEMP);I(TEMP),R)
```

Invocación a funciones

- Sea

función F(**E** $\underline{x}$:$\underline{tX}$) **dev** ($\underline{z}$:$\underline{tZ}$)
--Pre$_f$($\underline{x}$)
--Post$_f$($\underline{x}$,$\underline{z}$)

Teorema 1

$$\frac{Q \rightarrow \mathrm{Pre}_f{}^{\underline{E}}_{\underline{x}},\ \mathrm{Post}_f{}^{\underline{E},\underline{b}}_{\underline{x},\underline{z}} \rightarrow R}{\{Q\}\ b:= f(E)\ \{R\}}$$

Teorema 2

Caso en que f(E) aparece en una expresión I,

e:= I(f(E))

es equivalente a

TEMP := f(E)
e:= I(TEMP)

Operadores sobre ficheros secuenciales

- Vamos a denotar un fichero secuencial como:

$$\boxed{\texttt{f = (<d}_1\texttt{,d}_2\texttt{,...,d}_n\texttt{>,pB,M)}}$$

  - **$<d_1,d_2,...,d_n>$** es la secuencia de datos
  - **pB** $\in$ **{1..n+1}** representa la posición del buffer
    - “**pB=n+1**” significa que **finDeFichero(f)** toma el valor cierto
  - **M** $\in$ **{L,E}** representa el modo en que está abierto:
    - L: sólo lectura
    - M: sólo escritura

Operadores sobre ficheros secuenciales

- Sea **`f=(<d`**$_1$**`,d`**$_2$**`,...,d`**$_n$**`>,pB,M`**`)`un fichero

  `f: fichero de T`

  y sea **`R`** un predicado

`pmd(iniciarLectura(f),R)=` $R^{1,L}_{pB,M}$

`pmd(iniciarEscritura(f),R)=`

$$R^{<>,1,E}_{<d_1,\ldots,d_n>,pB,M}$$

Operadores sobre ficheros secuenciales

v es una variable de tipo T

`pmd(leer(f,v),R)=`

$$(pB<n+1)\wedge(M=L)\wedge R^{pB+1,d_{pB}}_{pB,v}$$

Exp es una expresión de tipo T

`pmd(escribir(f,Exp),R)=`

$$(pB=n+1)\wedge(M=E)\wedge R^{<d_1,\ldots,d_n,Exp>,n+2}_{<d_1,\ldots,d_n>,pB}$$

fF es una variable de tipo booleano

`pmd(fF:=finFichero(f),R)=` $R^{pB=n+1}_{fF}$

Operadores sobre ficheros secuenciales

- También podemos establecer reglas alternativas para los operadores anteriores

I: predicado que no depende ni de pB ni de M

$\{f=(<d_1,..,d_n>,pB,M) \wedge I\}$

`iniciarLectura(f)`

$\{f=(<d_1,..,d_n>,1,L) \wedge I\}$

I: predicado que no depende de f

$\{f=(<d_1,..,d_n>,pB,M) \wedge I\}$

`iniciarEscritura(f)`

$\{f=(<>,1,E) \wedge I\}$

Operadores sobre ficheros secuenciales

v: variable de tipo T
I: predicado que no depende ni de pB ni de v

$\{f=(<d_1,..,d_n>,pB,L) \wedge pB<n+1 \wedge I\}$

leer(f,v)

$\{f=(<d_1,..,d_n>,pB+1,L) \wedge v=d_{pB} \wedge I\}$

Exp: expresión de tipo T
I: predicado que no depende de n

$\{f=(<d_1,..,d_n>,n+1,E) \wedge I\}$

escribir(f,Exp)

$\{f=(<d_1,..,d_n,Exp>,n+2,E) \wedge I\}$

Operadores sobre ficheros secuenciales

fF: variable de tipo booleano
I: predicado que no depende de fF

$\{f=(<d_1,..,d_n>,pB,M) \wedge I\}$

`fF := finFichero(f)`

$\{f=(<d_1,..,d_n>,pB,M) \wedge fF=(pB=n+1) \wedge I\}$

Derivación de algoritmos

- Hasta ahora, hemos hablado de cómo verificar (validar) la corrección de un algoritmo
- *Gries*: la programación es una actividad dirigida por objetivos
  - se conoce la Post
  - la Pre es menos importante

`¡¡Incluso desconocida!!`

`derivación`

`deducir instrucciones a partir de su especificación`

Derivación de algoritmos

- Proceso para la derivación:
  - establecer muy claramente la Post
  - a partir de ella, tratar de derivar cuáles pueden ser las instrucciones que lleven a la Post
  - El proceso es mixto: se construyen simultáneamente el algoritmo (sus instrucciones) y su prueba (argumentación de la corrección)
  - el proceso es "retroalimentado": conforme se avanza en la derivación, se modifican inst. anteriores, prueba de la corrección
  - Util: la verificación ayuda al desarrollo en cuanto que determina claramente algunos elementos "peligrosos": dominio de los valores iniciales, conjunción/disjunción de las guardas, comparación "<" o "<=",...
- No es un proceso "mecanizado": hay que darle al "coco"

Derivación de algoritmos

- Sentido común:
  - Si en la Post aparecen igualdades entre variables y expresiones
    - probar asignaciones
  - Si en la Post aparecen disyunciones:
    - probar selecciones
  - Si en la Post aparecen conjunciones
    - intentar satisfacerlas por separado
    - tratar que todas "encajen" conjuntamente
- Proceso heurístico: de prueba y error

- <u>Ejemplo</u>: Derivar la acción A

```
--Q: x+y=T ∧ z=Z
   A
--R: x+y=T ∧ x=z ∧ z=Z
```

Derivación de algoritmos

- Ejercicios:

  1) Derivar “Q” y “A” para:

```
--Q
   A
--R: z=máximo(x,y)
```

  2) Derivar “A” para:

```
--Q:   x=X ∧ y=Y∧ XY=u+xy
   A
--R: x=X DIV 2 ∧ y=2Y ∧ XY=u+xy
```

  3) Derivar “Q” y “A” para:

```
--Q
   A
--R: z=abs(x)
```

TEMA 3: Diseño de algoritmos recursivos

1) Introducción a la recursividad

2) El método de inducción

3) Demostración de propiedades por inducción

4) Ejemplos de planteamientos recursivos

5) Inducción Noetheriana

6) Algoritmos recursivos

- diseño
- verificación
- cálculo de la complejidad

7) Técnicas de inmersión

- transformación de algoritmos por inmersión
  - » inmersión por cuestiones de eficiencia
  - » técnicas de plegado y desplegado
- diseño de algoritmos por inmersión
  - » por debilitamiento de Post
  - » por reforzamiento de la Pre

Introducción a la Recursividad

- Definición recurrente de un conjunto: se compone de 3 partes
  - BASE: establece un subconjunto de elementos iniciales
    - » Sirven, junto con la recurrencia, para construir nuevos elementos
  - RECURRENCIA: permite construir nuevos elementos del conjunto a partir de otros
    - » ya sean del conjunto base o construídos por aplicación de las reglas
  - CONCLUSION: es la “afirmación” de que los elementos así definidos son todos los del conjunto

Introducción a la Recursividad

- Ejemplo: las “`fbf`” de la LPPO
  - cualquier fórmula atómica es un FBF
    - T,F
    - una variable booleana
    - cualquier expresión booleana
  - (P) siendo P una FBF
  - $\neg$P, siendo P una FBF
  - $P \wedge Q$ siendo P,Q FBF
  - $P \vee Q$ siendo P,Q FBF
  - $P \rightarrow Q$ siendo P,Q FBF
  - $P \leftrightarrow Q$ siendo P,Q FBF
  - $\forall\ \alpha \in D.P$ siendo P FBF (D: dominio)
  - $\exists\ \alpha \in D.P$ siendo P FBF

Introducción a la Recursividad

- Ejemplo: Secuencia de datos de tipo T
  - **Base**: B={< >}
    - » < > se denomina "secuencia vacía", y está compuesta por el conjunto vacío de elementos de tipo T
  - **Recurrencia**: Si σ=<d$_1$, . . . , d$_n$> es una secuencia de datos de tipo T, y d$\in$T, entonces la "inserción por la derecha" de d a T, denotada como "$\sigma \bullet$d=<d$_1$,...,d$_n$,d>" , es también una secuencia de datos de tipo T.
  - **Conclusión**: s es una secuencia de datos de tipo T ssi o es la secuencia vacía o se ha formado usando la ley de recurrencia.
  - Comentarios:
    - » La regla de recurrencia es siempre "constructora"
    - » Esto estará ligado a la idea de "tipo abstracto de dato" (asignatura de EDA)

Demostración de propiedades por inducción

conjunto definido por recurrencia

demostración de propiedades por recurrencia

- Prueba en tres fases:
  - BASE: probar la propiedad para cada elemento del conjunto base
  - RECURRENCIA: si los generadores de una regla verifican la propiedad, probar que también los generados la cumplen
  - CONCLUSION: se afirma la generalidad de la propiedad sobre el total de los elementos
    - » generalmente obviaremos esta última etapa
- Generalización de la inducción sobre los Naturales

El método de inducción

- Conocemos inducción sobre los Naturales

Primer principio de Inducción

Sea P(n) un predicado (propiedad) que depende de n∈N. Si se verifican las condiciones:

B) P(0) es cierto

I) $\forall n \in N . P(n) \rightarrow P(n+1)$

entonces $\forall n \in N . P(n)$

El método de inducción

`Segundo principio de Inducción`

Sea P(n) un predicado (propiedad) que depende de n∈N. Si se verifica que

I') $\forall n \in N.((\forall k<n.P(k)) \rightarrow P(n))$

entonces $\forall n \in N.P(n)$

- <u>Ejercicios</u>:
  - Probar que $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
  - Probar que todo natural n>=2 se puede poner como producto de primos

Demostración de propiedades por inducción

- Más ejercicios
  - Probar que

$$\forall n \in N.((n+1)^2-(n+2)^2-(n+3)^2+(n+4)^2=4)$$

  - Probar que

$$\forall m \in N.\exists n \in N \wedge \exists \beta_1,...,\beta_n \in \{-1,1\}. \quad m=\beta_1 1^2+\beta_2 2^2+\ldots+\beta_n n^2$$

    - » Pista: Probarlo primero para $m \in \{0,1,2,3\}$

- Para definir una función sobre un dominio definido recurrentemente:
  - para un elemento base: definir el valor “directamente”
  - para un elemento generado: definir su valor en función a los valores de los elementos generadores

FUNCION RECURRENTE (RECURSIVA)

Ejemplos de planteamientos recursivos

- Ejemplo 1: ¿Cuántas permutaciones se pueden hacer con n datos?
  - Caso 1: si n=1, sólo una permutación
  - Caso 2: supongamos que M es el número de permutaciones de (n-1) elementos

b
a
s
e

¡recurrencia!

$d\ \boxed{d_1\ d_2\ d_3 \ldots d_{n-1}}$

$\boxed{d_1}\ d\ \boxed{d_2\ d_3 \ldots d_{n-1}}$

$\boxed{d_1\ d_2}\ d\ \boxed{d_3 \ldots d_{n-1}}$

......

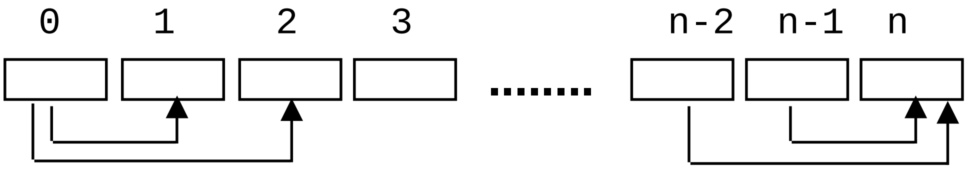
$\boxed{d_1\ d_2\ d_3 \ldots}\ d\ \boxed{d_{n-1}}$

$\boxed{d_1\ d_2\ d_3 \ldots d_{n-1}}\ d$

`nPerm(n) = n*M = n*nPerm(n-1) = n!`

Ejemplos de planteamientos recursivos

- Ejemplo 2: ¿De cuántas formas diferentes se puede alcanzar la losa `n`, partiendo de la losa 0 y sabiendo que hay dos movimientos posibles?
  - movimiento 1: saltar 1 losa a derecha
  - movimiento 2: saltar 2 losas a derecha



base

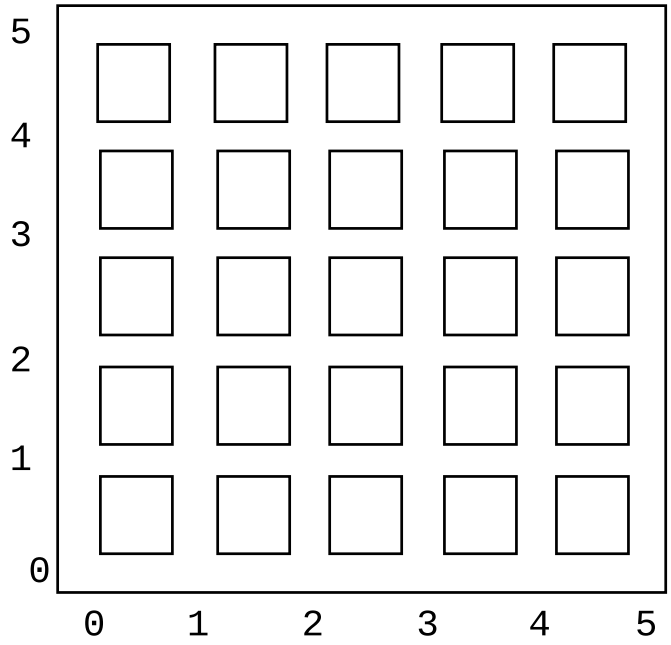
  - Caso 1: si n=1, sólo una forma
  - Caso 2: si n=2, hay dos formas
  - Caso 3: si n>2,

```
numFormas(n)=numFormas(n-1)+
             numFormas(n-2)
```

¡recurrencia!

Ejemplos de planteamientos recursivos

- Ejemplo 3: Caminos en una ciudad cuadri-culada



- ¿Cuántos caminos de longitud mínima hay para ir desde (0,0) hasta un (m,n) dados? (0<=m,n <= 5)

Algoritmos recursivos

```
Función factorial(E n:entero)
                        dev (r:entero)
--Pre: n≥0
--Post: r=n!
Principio
   Sel
     (n=0)∨(n=1): r:=1
     n>1: r:=n*factorial(n-1)
   FSel
   dev(r)
Fin
```

triv

recurrencia

```
Función caminosLosas(E n:entero)
                        dev (r:entero)
--Pre: n≥0
--Post: r=nº de caminos de 0 a n
Principio
   Sel
     (n=0)∨(n=1): r:=1
     n=2:  r:=2
     n>2:  r:= caminosLosas(n-1)+
                 caminosLosas(n-2)
   FSel
   dev(r)
Fin
```

triv

recurrencia

Algoritmos recursivos

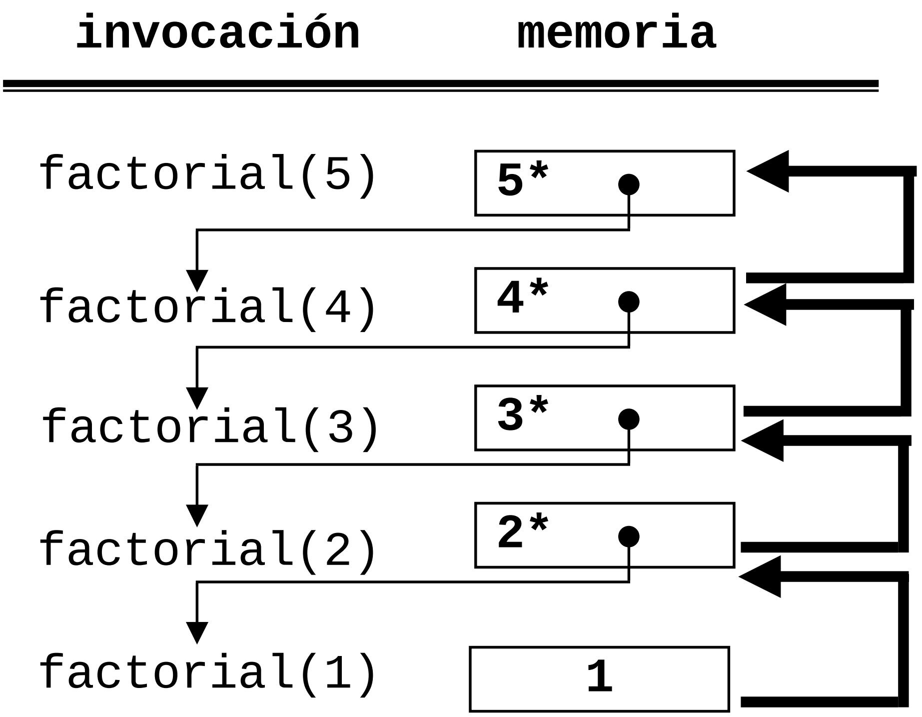
- Forma de una **función** recursiva:

```
Función funRec(E $\underline{x}$:$\underline{tX}$) dev ($\underline{r}$:$\underline{tR}$)
--Pre:  Q($\underline{x}$)
--Post: R($\underline{x}$,$\underline{r}$)
Principio
   ..........
   Sel
     .....
     $T_i$: $\underline{r}$:=$AT_i$
     .....
     $nT_j$: $\underline{r}$:=$AnT_j$(..funRec($transf_j$($\underline{x}$))..)
     .....
   FSel
   ..........
   dev($\underline{r}$)
Fin
```

  - AT_i : es la acción asociada a la guarda, que no genera invocaciones a la propia función
  - nT_j : representa un caso recurrente (no triv.)
  - AnT_j : contiene invocación a la propia función, pero los parámetros de la invocación son una transformación de los originales, y están “más cerca” de un caso básico

Algoritmos recursivos

- Intuitivamente: coste en tiempo y espacio del factorial rec.



invocación	**memoria**
`factorial(5)`	**5***
`factorial(4)`	**4***
`factorial(3)`	**3***
`factorial(2)`	**2***
`factorial(1)`	**1**

- Tenemos

$\boxed{T_{factRec(n)}=\Theta(n)}$ $\boxed{M_{factRec(n)}=\Theta(n)}$

$\boxed{T_{factIter(n)}=\Theta(n)}$ $\boxed{M_{factIter(n)}=\Theta(1)}$

- En general, serán menos eficientes

Verificación de algoritmos recursivos

- En muchos casos, cuando se diseña un algoritmo recursivo se sigue "lo que la intuición" dice
- Como siempre, esto es útil
- Pero, ¿Hace el algoritmo lo que realmente queremos que haga?
- ¿Cómo podemos demostrar que es así?

VERIFICACION-VALIDACION

- La verificación va a tratar de seguir la propia construcción recurrente:
  - verificar su corrección para los casos de base
  - supuesta correcta una invocación, demostrar corrección de la(s) invocaciones que la usan
  - asegurarse de que la invocación acaba

Verificación de algoritmos recursivos

- Recordemos: validar la función

```
Función funRec(E x:tX) dev (r:tR)
--Pre:  Q(x)
--Post: R(x,r)}
```

es demostrar que

$$\boxed{\forall \underline{x} \in D_{\underline{tX}} . Q(\underline{x}) \rightarrow R(\underline{x},\underline{r})}$$

- Los valores de salida verifican la Post siempre que los de entrada verifiquen la Pre
- Como hay que probarlo para todo dato del dominio, y habitualmente será muy grande, trataremos de razonar “por inducción”
- Cuando los parámetros son enteros (p.e.), es fácil aplicar inducción

Verificación de algoritmos recursivos

- Un razonamiento intuitivo

```
Función factorial(E n:entero) dev (r:entero)
--Q: n≥0
--R: r=n!
Principio
   Sel                  --n≥0∧((n=0)∨(n=1))→ n!=1

     (n=0)∨(n=1): r:=1
                         --n≥0∧n≥1→ n!=n(n-1)!

      n>1: r:=n*factorial(n-1)
   FSel
   dev(r)
Fin
```

- Además, es “obvio” que acaba
- Hay que tener cuidado con esto
  - Probar olvidando la precondición n≥0

Inducción Noetheriana

- Para dominios de otro tipo, vamos a hacer algo parecido: **inducción Noetheriana**
- Definición:

Dado un conjunto D, una relación binaria "$\leq$" $\subseteq$ DxD es un <u>PREORDEN</u> si es REFLEXIVA y TRANSITIVA

  - también llamamos preorden a (D, $\leq$)
  - si además antisimétrica, orden parcial
- Ejemplo:
  - D={cadenas finitas de caracteres}
  - c_1 " $\leq$" c_2 ssi long(c_1) $\leq$ long(c_2)
  - Probar que es preorden, pero no orden parcial
- Definición:

Dado un preorden(D,$\leq$), se define una relación "<", <u>PREORDEN ESTRICTO</u>, como

$$x<y \equiv (x\leq y) \wedge \neg(y\leq x)$$

Inducción Noetheriana

- Ejercicio: probar que es transitiva y anti-rreflexiva
- Definición:

Dado un preorden(D,≤), un elemento m∈D es MINIMO si no tiene predecesores estrictos:

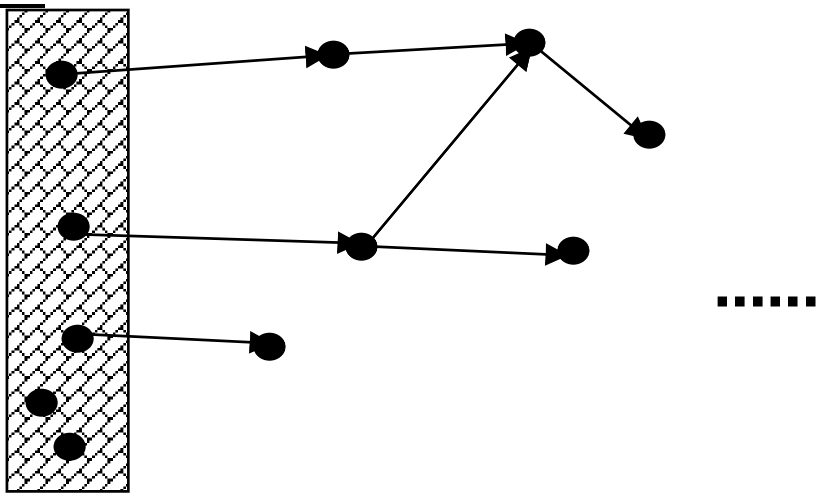
$$\texttt{m}\in\texttt{D mínimo} \equiv \neg(\exists \texttt{x}\in\texttt{D.x<m})$$

  - Nota: los elementos mínimos ni tienen por qué existir ni por qué ser únicos
- Ejemplos:
  - en (Z, ≤) no hay elementos mínimos
  - las cadenas finitas con la relación anterior tienen más de un mínimo
- Definición:

Un preorden(D,≤) se dice BIEN FUNDADO (PBF) si no existen en D sucesiones infinitas estrictamente decrecientes

Inducción Noetheriana

- Ejemplos de PBF:
  - (N ,≤)
  - cadenas de caracteres finitas siendo "≤" el orden lexicográfico
  - (P(A),⊆), siendo P(A) el conjunto de las partes del conjunto A
  - cualquier conjunto finito A con cualquier preorden "≤"
- Ejemplo de preórdenes que no son bien fundados
  - (Z , ≤)
  - ([0.0,1.0], ≤)
  - (N^2, ≤), siendo (a′,b′) ≤(a,b)≡(b′-a′) $\leq_Z$(b-a)
- Ejercicio: establecer un PBF en N^2



Inducción Noetheriana

- Proposición:

(D,≤) es un PBF ssi todo subconjunto A no vacío de D tiene al menos un mínimo (para A)

- A veces, la mejor manera de establecer un PBF es mediante una aplicación a un conjunto que ya tiene una definida
- Proposición:

Sea $(D_2,\leq_2)$ un PBF y sea $f:D_1 \rightarrow D_2$ una aplicación. Sean $a,b \in D_1$ y se define

$$a \leq_1 b \equiv f(a) \leq_2 f(b)$$

Entonces, $(D_1,\leq_1)$ es un PBF

- Ejercicio: Siendo A un conjunto finito, establecer un PBF en él

Inducción Noetheriana

- Ejemplo:N^2 se puede convertir en PFB mediante cualquier aplicación $f:N^2 \to N$
  1) $(a',b') \leq (a,b) \equiv a' \leq a$
  2) $(a',b') \leq (a,b) \equiv b' \leq b$
  3) $(a',b') \leq (a,b) \equiv (a'+b') \leq (a+b)$
  4) $(a',b') \leq (a,b) \equiv \text{máx}(a',b') \leq \text{máx}(a,b)$
  5) "inventar" alguno más

- Teorema: (principio de inducción completa sobre PBF)

Sea $(\mathbf{D},\leq)$ un PBF y sea P un predicado definido sobre los elementos de $\mathbf{D}$. Entonces

$$\frac{\forall a \in \mathbf{D}.(\forall b \in \mathbf{D}.b<a \to P(b)) \to P(a)}{\forall a \in \mathbf{D}.\ P(a)}$$

  - Notar parecido con segundo principio de inducción

Inducción Noetheriana

- El anterior principio es útil para probar propiedades (enunciadas como predicados) de PBF
- Por inducción Noetheriana
  - Base: probar la propiedad para cada elemento mínimo
  - Hipótesis de inducción: Asumir que, para un e cualquiera, la propiedad es cierta para todo predecesor suyo
  - Paso de inducción: probar que la propiedad se verifica para e
- Ejercicios:

  1) Considerando las cadenas de longitud finita con el preorden anterior, probar que toda cadena tiene longitud no negativa

  2) Considerando N^2 con cualquiera de los preórdenes anteriores, probar para todo `(a,b)` se verifica que

     `(a+b es par) ∨ (a+b es impar)`

Diseño y verificación de programas recursivos

- Una clasificación de programas recursivos:
  - lineales: cada invocación, genera una única invocación (excepto la última, claro)
    - » especial: recursividad final
  - múltiple: una misma invocación puede generar más de una invocación
    - » ejemplo: Fibonacci/camino en losas
- Otra clasificación:
  - finales: la actual invocación devuelve el mismo valor que la invocación que ella ha generado
  - no finales: el valor que dev. la actual invocación es el resultado de operar el valor de la invocación que ésta ha generado

Diseño y verificación de programas recursivos

```
Función maxCD(E a,b:entero) dev (m:entero)
--Pre: a>0 ∧ b>0
--Post: m=mcd(a,b)
Principio
   Sel
      a=b: m:=a
      a>b: m:=maxCD(a-b,b)
      a<b: m:=maxCD(a,b-a)
   FSel
   dev(m)
Fin
```

lineal
final

```
Función factorial(E n:entero) dev (r:entero)
--Pre: n≥0
--Post: r=n!
Principio
   Sel
      (n=0)∨(n=1): r:=1
       n>1: r:=n*factorial(n-1)
   FSel
   dev(r)
Fin
```

lineal
NO final

Diseño y verificación de programas recursivos

I) Especificación formal

II) Análisis de casos:

- identificar los posibles casos que pueden aparecer, de manera que se cubran todos los posibles estados que verifiquen la Pre
- habrá al menos un caso base (trivial) y un caso de invocación recursiva
- para identificar los casos base, habrá que "mirar" detenidamente la Post y la Pre

III) Composición

- generar el código de cada caso
- para los recursivos, hay que asegurarse de que la invocación se genera para datos más cercanos a los casos de base. En general, "más grande el salto", más eficiencia
  - » función de cota: hace corresponder a cada conj. de parámetros que verifican la Pre un elemento de un PBF
  - » será el elemento clave de la inducción

Diseño y verificación de programas recursivos

IV) Verificación formal de cada caso

V) Estudio de la eficiencia

- como siempre, tratar de encontrar una expresión del orden de la eficiencia
- en general, es “sencillo” como sigue:
  - obtener la recurrencia
  - resolver la recurrencia

Diseño y verificación de programas recursivos

- Asumamos, por simplificar, una función recursiva de la forma (D_f=dom.)

```
--Q(x)
Función f(E x:tX) dev (r:tR)
Principio
   Sel
     Bt(x): r:=triv(x)
     Bnt(x): r:=c(f(s(x)),x)
   FSel
   dev(r)
Fin
--R(x,r)
```

- Hay que probar lo siguiente:
  a) `f()` bien definida en el dominio.:

  $$\boxed{Q(\underline{x}) \rightarrow B_t(\underline{x}) \vee B_{nt}(\underline{x})}$$

  b) las su) se generan con parámetros “correctos”

  $$\boxed{B_{nt}(\underline{x}) \wedge Q(\underline{x}) \rightarrow Q(s(\underline{x}))}$$

  *Nota

  (Valores de tX que verifican la Pre)

Diseño y verificación de programas recursivos

c) la solución dada en el caso trivial es correcta

$$\boxed{Q(\underline{x}) \wedge B_t(\underline{x}) \rightarrow R(\underline{x}, triv(\underline{x}))}$$

d) asumiendo correcto el resultado de la invocación recursiva, probar la corrección del caso recursivo

$r' = f(s(\underline{x}))$

$$\boxed{Q(\underline{x}) \wedge B_{nt}(\underline{x}) \wedge \underbrace{R(s(\underline{x}), \underline{r}')} \rightarrow R(\underline{x}, c(\underline{x}, \underline{r}'))}$$

la invocación rec.
es correcta

Si hay varios casos recursivos, d) se prueba para cada uno de ellos

Diseño y verificación de programas recursivos

e) en cada llamada generada recursivamente, los parámetros son cada vez “más pequeños”

$$\boxed{Q(\underline{x}) \wedge B_{nt}(\underline{x}) \rightarrow s(\underline{x}) \text{ “<“ } \underline{x}}$$

f) los elementos mínimos del dominio de la función están incluídos en el caso no trivial y verifican la Pre.

<u>Nota</u>:

si el punto anterior es correcto, esta última comprobación es innecesaria, porque INEVITABLEMENTE llegaremos a tomar un valor mínimo

Diseño y verificación de programas recursivos

- Como ya se comentó
  - no dispondremos, en general, de un PBF para el dominio de la función, por lo que
    - » obtener uno a través de una aplicación de dicho dominio en uno que lo tenga (N, a ser posible)
  - Sea $t{:}D_f \rightarrow N$ (FUNCION DE COTA)
  - c) se sustituye por

$$\boxed{Q(\underline{x}) \wedge B_{nt}(\underline{x}) \rightarrow t(s(\underline{x})) < t(\underline{x})}$$

- A veces, se suele extender la función ‘`t`’ a todo tX como sigue:
  - como $\mathbf{D}_f \subseteq \mathbf{D}_{tX}$ se toma $\boxed{t{:}D_{tX} \rightarrow Z}$

  y se sustituye e) y f) por

e’) $$\boxed{\begin{array}{l} Q(\underline{x}) \rightarrow t(\underline{x}) \geq 0 \\ Q(\underline{x}) \wedge B_{nt}(\underline{x}) \rightarrow t(s(\underline{x})) < t(\underline{x}) \end{array}}$$

Diseño y verificación de programas recursivos

- En resumen, los puntos a verificar para asegurar la corrección de un algoritmo recursivo son:

1) $\boxed{Q(\underline{x}) \rightarrow B_t(\underline{x}) \vee B_{nt}(\underline{x})}$

2) $\boxed{B_{nt}(\underline{x}) \wedge Q(\underline{x}) \rightarrow Q(s(\underline{x}))}$

3) $\boxed{Q(\underline{x}) \wedge B_t(\underline{x}) \rightarrow R(\underline{x}, triv(\underline{x}))}$

$r' = f(s(\underline{x}))$

4) $\boxed{Q(\underline{x}) \wedge B_{nt}(\underline{x}) \wedge \underbrace{R(s(\underline{x}), \underline{r}')}_{} \rightarrow R(\underline{x}, c(\underline{x}, \underline{r}'))}$

la invocación rec.
es correcta

5) **Encontrar** $t : D_{tX} \rightarrow Z$ **t. q.**

$$Q(\underline{x}) \rightarrow t(\underline{x}) \geq 0$$

6) $\boxed{Q(\underline{x}) \wedge B_{nt}(\underline{x}) \rightarrow t(s(\underline{x})) < t(\underline{x})}$

Diseño y verificación de programas recursivos

- Un ejemplo:

```
--Q: $n \geq 0 \wedge a \neq 0$
Función pot(E a,n:entero) dev (p:entero)
Principio
   Sel
      n=0: p:=1
      n>0: p:=a*pot(a,n-1)
   FSel
   dev(p)
Fin
--R: $p=a^n$
```

- Para este caso:

$\underline{x}=(a,n)$
$\underline{r}=(p)$
$B_t=(n=0)$
$B_{nt}=(n>0)$
$s(a,n)=(a,n-1)$
$c(p',(a,n))=a*p'$

1) $n \geq 0 \wedge a \neq 0 \rightarrow (n=0) \vee (n>0)$
2) $n \geq 0 \wedge n>0 \wedge a \neq 0 \rightarrow n-1 \geq 0 \wedge a<>0$
3) $n \geq 0 \wedge n=0 \wedge a \neq 0 \rightarrow 1=a^n$
4) $n \geq 0 \wedge n>0 \wedge a \neq 0 \wedge (p'=a^{n-1}) \rightarrow a*p'=a^n$
5) tomamos $t(a,n)=n$
6) $n \geq 0 \wedge n>0 \rightarrow n-1<n$

Coste:???????

Diseño y verificación de programas recursivos

- Ejercicios: Probar la corrección de

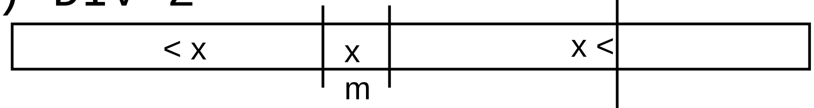
```
Función factorial(E n:entero) dev (r:entero)
--Pre: n≥0
--Post: r=n!
Principio
   Sel
     (n=0)∨(n=1): r:=1
     n>1: r:=n*factorial(n-1)
   FSel
   dev(r)
Fin
```

```
--Q: (a≥0)∧(b≥0)∧¬((a=0)∧(b=0))
--R: m=maximo común divisor (a,b)
Función mcd(E a,n:entero)
                    dev (m:entero)
Principio
   Sel
     a=0: m:=b
     b=0: m:=a
     (a≥b)∧(b>0): m:=mcd(b,a MOD b)
     (a>0)∧(b≥a): m:=mcd(a,b MOD a)
   FSel
   dev(m)
Fin
```

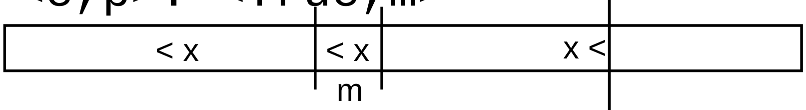
Diseño y verificación de programas recursivos

```
Constantes N=.....
Tipos vect=vector[1..N] de entero
```

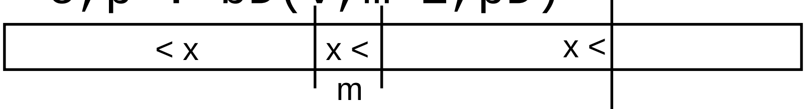
```
--Q: orden(v,pI,pD)∧(1≤pI≤pD+1<N)
Función bD(E v:vect;E x,pI,pD:entero)
           dev (e:booleano;p:entero)
--R: (e →(pI≤p≤pD)∧(x=v[p])) ∧
--   (¬e →(pI≤p≤pD+1) ∧
--         (v[pI..p-1]<x) ∧
--         (x<v[p..pD]))
Principio
   Sel
     pI>pD: <e,p>:=<False,pI>
     pI≤pD:
       m:=(pI+pD) DIV 2
       Sel
```



```
         x=v[m]: <e,p>:=<True,m>
```



```
         x>v[m]: <e,p>:=bD(v,m+1,pD)
```



```
         x<v[m]: <e,p>:=bD(v,pI,m-1)
       FSel
   FSel
   dev(e,p)
Fin
```

Diseño y verificación de programas recursivos

- Diseñar y verificar las siguientes funciones

```
Función sumaC(E v:vect;E pI,pD:entero)
                         dev (sC:entero)
--QsC: $1 \leq pI \leq pD \leq N$
--RsC: $sC= \sum \alpha\in\{pI..pD\}.v[\alpha]$
```

```
Algoritmo copia(E v:vect; S w:vect
                        E pI,pD:entero)

--Qc: $1 \leq pI \leq pD \leq N$
--Rc: $\forall\alpha\in\{pI..pD\}.w[\alpha]=v[\alpha]$
```

```
Función divide(E a,b:entero)
                     dev (q,r:entero)
--Qd: $a\geq 0 \wedge b>0$
--Rd: $a=bq+r \wedge 0\leq r<b$
```

  - Tratar de evaluar el coste de los ejemplos anteriores

Diseño y verificación de programas recursivos

- Ejercicios: Especificar formalmente, diseñar recursivamente y verificar los siguientes algoritmos

1) Función que devuelva el valor medio de los datos en un fichero de reales (parámetro)

2) Algoritmo que almacene el contenido de un fichero de enteros en otro, pero en sentido invertido

3) Función que determine si un string es palíndromo o no

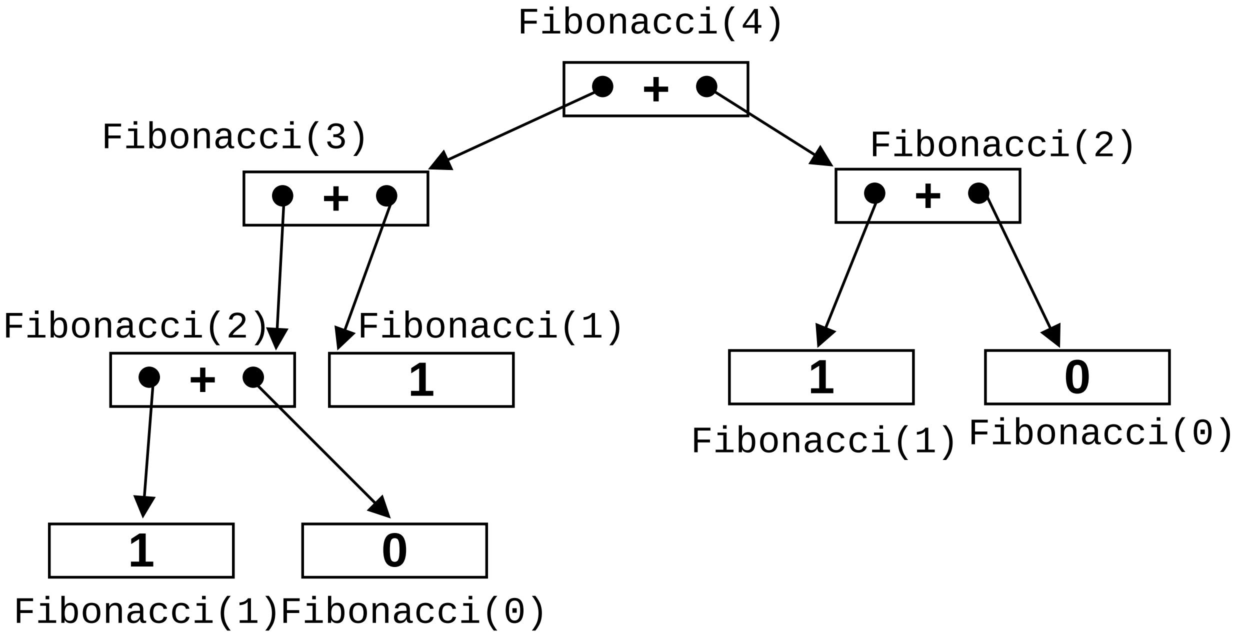
4) Función que determine si un vector de enteros está ordenado "≤"

5) Algoritmo que mezcle dos vectores de enteros "≤" en un tercero, quedando éste también "≤" (los tres vectores son parámetros)

6) Función que cuente el número de repeticiones de un entero (parámetro) en un fichero de enteros (cuyo nombre es también parámetro)

Diseño de algoritmos por inmersión

- Algunas veces, un diseño recursivo puede ser poco eficiente



- Otras, es imposible

```
Constantes n=.....
Tipos      vect=vector[1..n] de real

Función sumaC(E v:vect) dev (sC:real)
--Pre: True
--Post: sC=∑α∈{1..n}.v[α]
```

Diseño de algoritmos por inmersión

- Algunos de estos casos pueden resolverse mediante el mecanismo de la inmersión de algoritmos
- Inmersión de un algoritmo:
  - generalización del mismo
  - con más parámetros y/o resultados
  - para determinados valores de los parámetros se tiene la solución del problema inicial

```
--Q(x)
Función f(E x:tX) dev (r:tY)
--R(x,r)
```

```
--Q'(x,w)
Función g(E x:tX;E w:tW) dev (r:tY;z:tZ)
--R'(x,w,r,z)
```

- T. q. bajo determinadas condiciones, de la solución de `g` se obtenga la solución de `f`

$$\boxed{\mathbf{Q'(\underline{x},\underline{w})\wedge P(\underline{x},\underline{w})\wedge R'(\underline{x},\underline{w},\underline{r},\underline{z})\rightarrow R(\underline{x},\underline{r})}}$$

Diseño de algoritmos por inmersión

- Diferentes técnicas
- Inmersión por cuestiones de eficiencia
  - ya se dispone de una solución recursiva
  - buscamos una más eficiente
    - inmersión de parámetros
    - inmersión de resultados
    - técnicas de plegado y desplegado
- Inmersión de especificaciones
  - buscamos un algoritmo, pero su especificación impide una solución recursiva
  - por "debilitamiento de la Post"
    - dará solución recursiva no final
  - por "reforzamiento de la Pre"
    - dará solución recursiva final

Inmersión por cuestiones de eficiencia

- Recordar Fibonacci:

```
Función Fib(E n:entero) dev (f:entero)
--Q(n):   n≥0
--R(n,f): f=Fibonacci(n)}
Principio
   Sel
     n=0: f:=0
     n=1: f:=1
     n>1: f:= Fib(n-1)+ Fib(n-2)
   FSel
   dev(f)
Fin
```

“recalcula Fib(n-2)”

- Otro caso:

```
Función f(E a:entero;..)
                      dev (......)
{......}
Principio
   ....
   .... a² ....
   .... f(..a+1..)..
   ....
Fin
```

a^2 es de ayuda para obtener $(a+1)^2$

Inmersión por cuestiones de eficiencia

- Dos casos fundamentales
- Caso 1: la expresión “compleja” involucra sólo elementos de x

caso a^2

inmersión de parámetros

“parámetros acumuladores”

- Caso 2: la expresión “compleja” se evalúa después de la llamada recursiva, e involucra resultados de la misma

caso Fib()

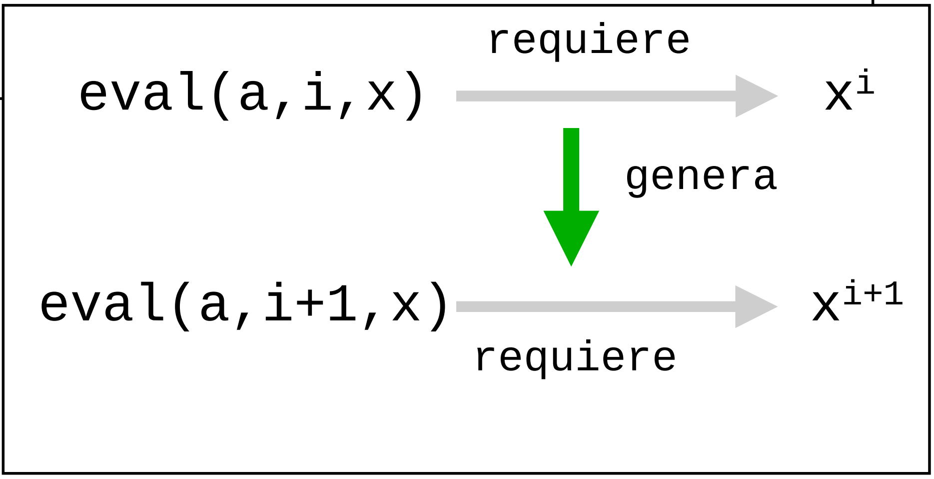
inmersión de resultados

“resultados acumuladores”

Inmersión de parámetros

- Caso 1: inmersión de parámetros

```
Constantes n=....
Tipos vect=vector[0..n] de real

Función eval(ES a:vect;E i:entero;
             E x:real) dev (v:real)
--Q(a,i,x):   $0 \le i \le n$
--R(a,i,x,v): $v=\sum \alpha \in \{i..n\}.a[\alpha]x^{\alpha}$
--eval(a,0,$x_0$) da el valor del pol.
--en el punto $x_0$
Principio
   Sel
      i=n: v:=a[n]*$x^n$
      i<n: v:=a[i]*$x^i$+eval(a,i+1,x)
   FSel
   dev(v)
Fin
```



eval(a,i,x) requiere x^i

genera

eval(a,i+1,x) requiere x^{i+1}

Inmersión de parámetros

- Demasiados cálculos innecesarios

```
Función eval2(ES a’:vect;E i’:entero;
       E x’:real;E w:real) dev (v’:real)
--Q(a,i,x,w) : $0 \leq i' \leq n \wedge w=x'^{i}$
--R(a,i,x,w,v) : $v=\sum_{\alpha \in \{i'..n\}} a[\alpha]x'^{\alpha}$
Principio
   Sel
     i’=n: v’:=a’[n]*w
     i’<n: v’:=a’[i’]*w +
                eval2(a’,i’+1,x’,x’*w)
   FSel
   dev(v’)
Fin
```

```
Función eval(ES a:vect;E i:entero;
             E x:real) dev (v:real)
Principio
   v:=eval2(a,i,x,$x^{i}$)
   dev(v)
Fin
```

Inmersión de parámetros

- Esquemáticamente:
  - añadir a Q($\underline{x}$) una conjunción de la forma

$$\underline{w}=\Phi(\underline{x})$$

¡Sólo depende de $\underline{x}$!

  - sustituir en **`f()`** toda aparición de $\Phi(\underline{x})$ por $\underline{w}$
  - calcular en la función sucesor de **`g()`** el nuevo valor $\underline{w}'$, de manera que la precondición siga invariante
  - el valor inicial $\underline{w}_{ini}$ se obtiene por la propia precondición:

$$\underline{w}_{ini}=\Phi(\underline{x}_{ini})$$

Inmersión de resultados

- Caso 2: inmersión de resultados

```
Función Fib(E n:entero) dev (f:entero)
--Q(n): n≥0
--R(n,f): f=Fibonacci(n)
Principio
   Sel
     n=0: f:=0
     n=1: f:=1
     n>1: f:= Fib(n-1)+Fib(n-2)
   FSel
   dev(f)
Fin
```

```
Función Fib2(E n:entero) dev
                (fN,fNM1:entero)
--Q(n):    n≥1
--R(n,fN,fNM1): fN=Fibonacci(n) ∧
--              fNM1=Fibonacci(n-1)
```

```
Función Fib(E n:entero) dev (f:entero)
variables fAux:entero
Principio
   Sel
     n=0: f:=0
     n≥1: <f,fAux>:=Fib2(n)
   FSel
   dev(f)
Fin
```

Inmersión de resultados

- Donde `Fib2()` es como sigue:

```
Función Fib2(E n:entero) dev
               (fN,fNM1:entero)
--Q(n): n≥1
--R(n,fN,fNM1): fN=Fibonacci(n) ∧
--              fNM1=Fibonacci(n-1)
Principio
   Sel
     n=1: <fN,fNM1>:=<1,0>
     n>1: <fN,fNM1>:= Fib2(n-1)
                  --fN=Fibonacci(n-1)
                  --fNM1=Fibonacci(n-2)
          <fN,fNM1>:=<fN+fNM1,fN>
   FSel
   dev(fN,fNM1)
Fin
```

- Estudiar las complejidades de ambas versiones de Fibonacci

Inmersión de resultados

- Esquemáticamente:
  - añadir la conjunción $\underline{z}=\Phi(\underline{x},\underline{r})$ a la post $R(\underline{x},\underline{r})$
    - » $\underline{z}=\Phi(\underline{x},\underline{r})$
      es el resultado acumulador
    - » $\Phi(\underline{x}',\underline{r}')$
      expresión que la llamada interna devuelve, calculada, a la actual
  - sustituir en `f` cada aparición de $\Phi(\underline{x}',\underline{r}')$ por $\underline{z}'$, resultado precalculado en la invocación recursiva
  - para el caso no trivial, rehacer la Post, calculando $\underline{z}$ a partir de $\underline{z}'$
  - para el caso trivial, rehacer la Post mediante un valor de $\underline{z}$ que la satisfaga

Técnicas de desplegado y plegado

- Se trata de una técnica de **transformación** de algoritmos
- **Transformación**: obtención de un algoritmo a partir de otro, pero con el mismo comportamiento
  - deseo de mayor eficiencia
  - imposibilidad de implementación directa
- Varias técnicas
  - desplegado/plegado (ahora)
  - transformación recursivo-iterativo (más adelante)
- **Interesante**: encadenamiento de transformaciones

Técnicas de desplegado y plegado

- Recordar

NO final

```
Función fact(E n:entero) dev (f:entero)
--Q(n): n≥0
--R(n,f): f=n!
Principio
   Sel
     n=0: f:=1
     n>0: f:=n*fact(n-1)
   FSel
   dev(f)
Fin
```

```
f:=fact(n-1)
f:=n*f
```

final

```
Tipos fichEnt=fichero de entero
Función buscaPorPos(E f:fichEnt;
         E p,pos:entero) dev (c:entero)
--Q(f,p,pos): f=(<$d_1$,...,$d_n$>,p,L) ∧
              1 ≤ p ≤ pos ≤ n
--R(f,p,pos,c): f=(<$d_1$,...,$d_n$>,pos+1,L)∧
                c=$d_{pos}$
Principio
   Sel
     p=pos: leer(f,c)
     p<pos: leer(f,c) --avanza a sig. pos
            c:=buscaPorPos(f,p+1,pos)
   FSel
   dev(c)
Fin
```

¿Cómo se puede usar para encontrar el dato que en un fichero ocupa determinada pos?

Técnicas de desplegado y plegado

- Partimos de una solución recursiva (lineal)
- Buscamos una solución recursiva final
  - más eficientes en tiempo y espacio
  - fácilmente transformable a una iterativa
- En resumen: (caso de funciones)

dada una función recursiva no final **f(..)**, encontrar una función recursiva final **g(..)** con el mismo comportamiento (para determinados casos)

Técnicas de desplegado y plegado

- Tres etapas (inmersión para f):
  - generalización
    - » construir una `g()` más general (más parámetros)
  - desplegado
    - » se "despliega" `f()` dentro de `g`() (manipulaciones `+/-` automáticas)
  - plegado
    - » sustituir la aparición de `f()` en `g()` por una expresión equivalente, pero sólo con `g()`

Técnicas de desplegado y plegado

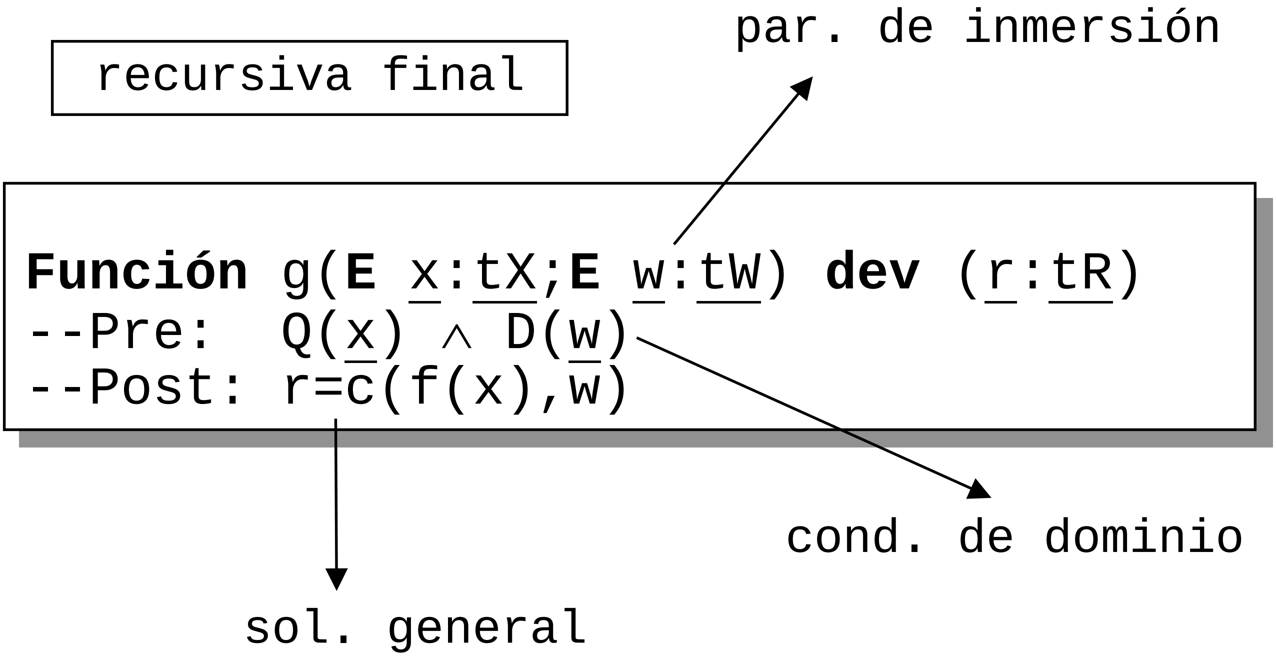
- Forma que trataremos

```
--Q(x)
Función f(E x:tX) dev (r:tR)
Principio
   Sel
      $B_t$(x): r:=triv(x)
      $B_{nt}$(x): r:=c(x,f(s(x)))
   FSel
   dev(r)
Fin
--R(x,r)
```

Técnicas de desplegado y plegado

- **Generalización**: buscamos

`recursiva final`



```
Función g(E x:tX;E w:tW) dev (r:tR)
--Pre:  Q(x) ∧ D(w)
--Post: r=c(f(x),w)
```

  - notar que si `c(..)` tiene **elemento neutro** $\underline{w}_0$

$$\boxed{\mathbf{g(\underline{x},\underline{w}_0)} = c(f(\underline{x}),\underline{w}_0) = \mathbf{f(\underline{x})}}$$

¡¡Tenemos f como un caso particular de g!!

Técnicas de desplegado y plegado

- **DESPLEGADO**

$\boxed{g(\underline{x},\underline{w}) \equiv c(f(\underline{x}),\underline{w})}$

=

c(**Sel**
    $B_t(\underline{x})$: $\underline{r}$:=triv($\underline{x}$)
    $B_{nt}(\underline{x})$: $\underline{r}$:=c(f(s($\underline{x}$)),$\underline{x}$)
  FSel,
  $\underline{w}$)

=

Sel
    $B_t(\underline{x})$: $\underline{r}$:=c(triv($\underline{x}$),$\underline{w}$)
    $B_{nt}(\underline{x})$: $\underline{r}$:=c(c(f(s($\underline{x}$)),$\underline{x}$),$\underline{w}$)
FSel

si **c** asociativa

=

Sel
    $B_t(\underline{x})$: $\underline{r}$:=c(triv($\underline{x}$),$\underline{w}$)
    $B_{nt}(\underline{x})$: $\underline{r}$:=c(f(s($\underline{x}$)),c($\underline{x}$,$\underline{w}$))
FSel

x'=s($\underline{x}$)
w'=c($\underline{x}$,$\underline{w}$)

g(x',w')=g(s($\underline{x}$),c($\underline{x}$,$\underline{w}$))
¡¡Forma de g!!

Técnicas de desplegado y plegado

- **PLEGADO**

```
g(x,w)
```

=

```
Sel
   Bt(x):  r:=c(triv(x),w)
   Bnt(x): r:=g(s(x),c(x,w))
FSel
```

- Importante: para llevar a cabo esta inmersión hemos impuesto que:
  - c tenga un elemento neutro
    - » dará el caso particular en que g se comporta como f
  - c sea asociativa
- No siempre es posible
- Hay formas más generales en que es posible

Técnicas de desplegado y plegado

- En resumen:

```
Función f(E x:tX) dev (r:tR)
--Pre:  Q(x)
--Post: R(x,r)
Principio
   r:= g(x,$w_0$)
   dev(r)
Fin
```

1) ∃ w_0 t.q. f(x)=g(x,w_0)

2) c asociativa

```
Función g(E x:tX;E w:tW) dev (r:tR)
--Pre:  Q(x) ∧ D(w)
--Post: r=c(f(x),w)
Principio
   Sel
      $B_t$(x):  r:=c(triv(x),w)
      $B_{nt}$(x): r:=g(s(x),c(x,w))
   FSel
   dev(r)
Fin
```

Técnicas de desplegado y plegado

- Ejemplo: a recursivo final

```
Constantes n=..... {n≥1}
Tipos      vect=vector[1..n] de entero

Función sumaC(E v:vect;E i:entero)
                                dev (sC:entero)
{Pre: 1 ≤ i ≤ n
 Post: sC=$\sum \alpha \in \{1..i\}.v[\alpha]$}
Principio
   Sel
     i=1: sC:=v[1]
     i>1: sC:=sumaC(v,i-1)+v[i]
   FSel
   dev(sC)
Fin
```

Inmersión por debilitamiento de la Post

- A veces, no podremos construir, a partir de la especificación, una solución recursiva
  - por ser imposible/por no saber hacerlo
- Ejemplo:

```
Constantes n=.....
Tipos      vect=vector[1..n] de real

Función prodEsc(E v1,v2:vect)
                             dev (pE:real)
--Pre: True
--Post: pE=∑α∈{1..n}.v1[α]*v2[α]
```

- Intentarlo "pidiéndonos menos"
  - debilitando la Post
    - » generaremos recursividad no final
  - reforzando la Pre
    - » generaremos recursividad final
- Inmersión de **especificaciones**, no de algoritmos

Inmersión por debilitamiento de la Post

- Partimos de

```
--Q(x)
Función f(E x:tX) dev (r:tR)
--R(x,r)
```

- Buscamos

```
--Q’(x,w)
Función g(E x:tX;E w:tW) dev (r:tR)
--R’(x,w,r)
```

tal que para cierto $\underline{w}_{ini}$

$$\boxed{f(\underline{x})=g(\underline{x},\underline{w}_{ini})}$$

- Sobre debilitamiento de asertos
  - ¿Qué es debilitar un aserto?
  - ¿Cómo se puede debilitar un aserto?

Inmersión por debilitamiento de la Post

- Proceso:
  - Sustituir en $R(\underline{x},\underline{r})$ ctes. o variables que dependan sólo de $\underline{x}$ por variables inmersoras

$$\boxed{R'(\underline{x},\underline{r},\underline{w})}$$

  - Llamando $\Psi(\underline{x})$ a lo sustituído, tenemos

$$\boxed{R'(\underline{x},\underline{r},\underline{w}) \xrightarrow[\underline{w}]{\Psi(\underline{x})} R(\underline{x},\underline{r})}$$

  - LLamando $P(\underline{x},\underline{w})= \underline{w}=\Psi(\underline{x})$ (ec. de sust.)

$$\boxed{R'(\underline{x},\underline{r},\underline{w})\wedge P(\underline{x},\underline{w}) \rightarrow R(\underline{x},\underline{r})}$$

  - Obtención de Q':

$$\boxed{Q'(\underline{x},\underline{w})=Q(\underline{x}) \wedge \underbrace{D(\underline{x},\underline{w})}}$$

elimina valores de w que hagan R' falso o indefinido

  - establecer el $\underline{w}_{ini}$ adecuado
  - diseñar el nuevo algoritmo
  - recursivo NO final

Inmersión por debilitamiento de la Post

- Encontrar dos soluciones de inmersión distintas por debilitamiento de la Post para:

```
Constantes n=.....
Tipos      vect=vector[1..n] de real

Función prodEsc(E v1,v2:vect)
                        dev (pE:real)
--Pre: True
--Post: pE=∑α∈{1..n}.v1[α]*v2[α]
```

- Lo mismo para

```
Función max(E v:vect) dev (m:real)
--Pre: True
--Post: ∃α∈{1..n}.v[α]=m ∧
--      ∀β∈{1..n}.v[β]≤m
```

Inmersión por reforzamiento de la Pre

- La forma más sencilla de reforzar un aserto: **<u>añadir conjunciones</u>**
- En la propia `Pre` del alg. inmersor se exigirá ya parte de la `Post`

  (mediante par. inmersores)

  - la nueva Pre será un debilitamiento de la `Post`, con condiciones de dominio para las variables de inmersión
- La <u>Post no ha de variar</u>, con lo que se conseguirá una solución <u>recursiva</u> <u>final</u>
- Trataremos de encontrar parámetros de inmersión

$$\boxed{\underline{w}=(\underline{w}_1,\underline{w}_2)}$$

- Poner el alg. `g()` de la forma:

```
Sel
    $B_t$: $\underline{r}$ := $\boxed{\underline{w}_1}$
    $B_{nt}$: $\underline{r}$ := g(...)
FSel
```

par. acumulador

Inmersión por reforzamiento de la Pre

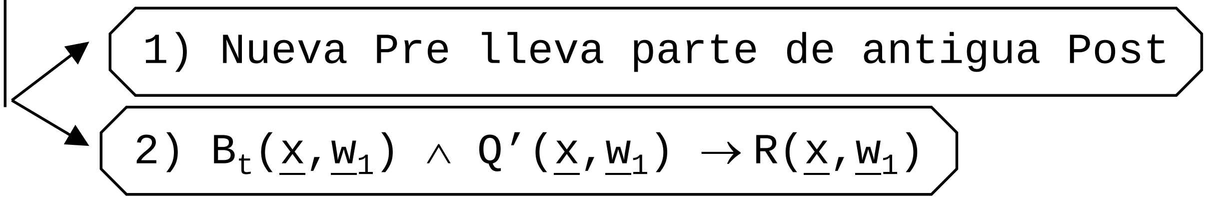
- Proceso:
  - renombrar en $R(\underline{x},\underline{r})$, $\underline{r}$ como $\underline{w}_1$
  - tratar de poner $R(\underline{x}, \underline{w}_1)$ como conjunción

$$\boxed{R(\underline{x},\underline{w}_1)=A(\underline{x},\underline{w}_1) \wedge C(\underline{x},\underline{w}_1)}$$

  - Caso 1: $R(\underline{x},\underline{r})$ como conjunción
    - » tomar $\underline{w}=\underline{w}_1$ como parámetro de inmersión
    - » meter A ó C como parte de Q'
    - » tomar B_t como la parte que queda
  - Diseñar el resto del algoritmo
  - Encontrar $\underline{w}_{ini}$ tal que

$$\boxed{Q(\underline{x}) \rightarrow Q'(\underline{x}, \underline{w}_{ini})}$$

  - Verificar



1) Nueva Pre lleva parte de antigua Post

2) $B_t(\underline{x},\underline{w}_1) \wedge Q'(\underline{x},\underline{w}_1) \rightarrow R(\underline{x},\underline{w}_1)$

Inmersión por reforzamiento de la Pre

```
Función f(E x:tX) dev (r:tR)
--Q(x)
--R(x,r)= A(x,r) ∧ C(x,r)
```

```
Función g(E x’:tX;E w:tW) dev (r’:tR)
--Q’(x’,w)= A(x’,w) ∧ D(w)
--R(x’,r’)= A(x’,r’) ∧ C(x’,r’)
Principio
   Sel
      C(x’,w’): r’:=w
      ¬C(x’,w’):  ..... --diseñar
   FSel
   dev(r’)
Fin
```

- También se llama
  <u>**inmersión con Post constante**</u>

Inmersión por reforzamiento de la Pre

- Caso 2: $R(\underline{x},\underline{r})$ NO es conjunción
  - debilitar $R(\underline{x},\underline{r})$

$$\boxed{R_{\text{débil}}(\underline{x},\underline{w}_1,\underline{w}_2)}$$

mediante sustitución $\boxed{\underline{w}_2=\Psi(\underline{x},\underline{w}_1)}$

de manera que

$$\boxed{\begin{array}{c} R_{\text{débil}}(\underline{x},\underline{w}_1,\underline{w}_2) \wedge \underline{w}_2=\Psi(\underline{x},\underline{w}_1) \\ \Rightarrow \\ R(\underline{x},\underline{w}_1) \end{array}}$$

- ¡CASO ANTERIOR!

Inmersión por reforzamiento de la Pre

- Ejemplos: Resolver mediante inmersión por reforzamiento de la Pre

```
Función raiz(E n:entero) dev (r:entero)
--Q: n≥0
--R: r² ≤ n < (r+1)²
```

```
Constantes n=.....
Tipos      vect=vector[1..n] de real

Función prodEsc(E v1,v2:vect)
                                 dev (pE:real)
--Q: True
--R: pE=∑α∈{1..n}.v1[α]*v2[α]
```

```
Función max(E v:vect) dev (m:real)
--Q: True
--R: ∃α∈{1..n}.v[α]=m ∧
     ∀β∈{1..n}.v[β]≤m
```

Inmersión por reforzamiento de la Pre

```
Constantes N=    --entero,>=1
Tipos vectN=vector(1..N) de entero
```

```
Función mayorCola(E v,v’:vectN)
                     dev (pM,s:entero)
--QmC: TRUE
--RmC: pM es la primera posición,empe-
--     zando por la derecha, tal que
--     las componentes de v y v’ desde
--     pM hasta N coinciden.
--     s es la suma de dichas
--     componentes
```

Inmersión por reforzamiento de la Pre

```
Constantes N=    --entero,>=1
Tipos vectN=vector(1..N) de entero
```

```
Función cuentaComunes(E v,v’:vectN)
                     dev (cC:entero)
--QcC: tanto “v” como “v’” están
--     ordenados en sentido
--     estrictamente creciente
--RcC: “cC” es el número de valores
--     comunes a ambos vectores
```

TEMA 3Bis: Resolución de ecuaciones recurrentes

1) La eficiencia de algoritmos recursivos

2) Clasificación de ecuaciones recurrentes

3) Recurrencias lineales homogéneas de coeficientes constantes

4) Recurrencias lineales NO homogéneas de coeficientes constantes

5) Recurrencias lineales de coeficientes variables

6) Recurrencias de partición y cambio de variable

7) ¿Y cuando todo falla?

Eficiencia de algoritmos recursivos

- Cálculo del tiempo de ejecución:
  - Factorial: $t_0=t_1=k_1$
    $t_n=t_{n-1}+k_2$
  - Fibonacci: $t_0=t_1=k_1$
    $t_n=t_{n-1}+t_{n-2}+k_2$
  - Luego, en ambos casos, la obtención de t_n pasa por la resolución de un sistema de ecuaciones recurrentes
    - » $t_0=t_1=k_1$ establece en ambos casos las condiciones iniciales
    - » $t_n=f(t_{n-1},t_{n-2},\ldots)$ establece la recurrencia
    - » k_1 y k_2 son datos del problema
- Luego necesitamos saber resolver ecuaciones recurrentes...

Ecuaciones recurrentes

- Forma general de una ecuación recurrente

$$\boxed{u_n=f(\{u_p | p<n\}),\ n \in J \subset N}$$

- `J` establece el rango en que `n` puede variar de forma que la ecuación esté bien definida. En general, será `N`
- Distintas clasificaciones de las ecuaciones recurrentes
  - Por la forma de **`f()`**
  - Por las u_p, $p<n$, que intervienen en la definición de u_n
  - Por el hecho de que en **`f()`**, además de las u_n, aparezcan otros parámetros

Ecuaciones recurrentes

- Por la forma de f:
  - <u>recurrencia lineal</u>: f es una c.l. de las u_p
    - » coeficientes constantes
    - » coeficientes variables
  - <u>recurrencia polinomial</u>: f es un polinomio en las u_p
- Por las u_p , $p<n$, que intervienen en la definición de u_n
  - <u>recurrencia completa</u>: aparecen todas
  - orden K: depende de $u_{n-1},\ldots,u_{n-k}$
    - » notar que para este caso,
      $J=\{n\in N \mid p\geq k\}$
  - <u>recurrencia de partición</u>: sólo interviene $u_{n/a}$, siendo $a\in N$ y $a>1$
    - » notar que $J=\{n\in N \mid a \text{ divide a } n\}$

Ecuaciones recurrentes

- Por parámetros que acompañen a las u_p en f
  - <u>recurrencia homogénea</u>: f sólo depende de las u_p
  - <u>recurrencia no homogénea</u>: f depende de más parámetros
    - » en general, trabajaremos con la forma

$$\boxed{u_n=f(\{u_p|p<n\})+\underbrace{\boxed{g(n)}}_{\texttt{segundo miembro}}}$$

Ecuaciones recurrentes

- Sea una ecuación recurrente

$$\boxed{u_n=f(\{u_p|p<n\}),\ n\in J\subset \mathbb{N}} \quad \textbf{(1)}$$

- Una solución consiste en encontrar una sucesión $(u_i)_{i\geq 0}$ que verifican (1)
- De entre las soluciones, nos interesará, en general, aquélla(s) que verifican determinadas condiciones iniciales

$$\boxed{\{a_i|i\in I\},\ I\subset \mathbb{N}}$$

- Significado: para algunos valores de subíndices i (aquéllos que pertenecen al conjunto I) los valores de las u_i están predeterminados
  - Factorial: $t_0=t_1=k_1$

    $t_n=t_{n-1}+k_2$

    $I=\{1,2\}$

Ecuaciones recurrentes

- Proposición 1:

Sea $u_n=f(\{u_p|p<n\})$, $n \in J \subset \mathbb{N}$, una relación de recurrencia. Entonces:

- * Si f completa y u_0 dado, existe al menos una solución
- * f lineal de orden k y $u_0,...,u_j$ dadas, $j<k$, hay al menos una solución
- * f de partición $u_n=f(u_{a/n})$ y se da $u_0,...,u_j$, $j<a$, al menos una solución

- Proposición 2:

Sea $u_n=f(\{u_p|p<n\})$, $n \in J \subset \mathbb{N}$, una relación de recurrencia. Entonces, la solución es única cuando:

- * f de orden k y $u_0,...,u_{k-1}$ dadas
- * f completa y u_0 dada
- * f de partición $u_n=f(u_{a/n})$ y todos los u_i tal que $(i<a)$ ó $((i \geq a) \wedge$ a no divide $i)$ dados

  - Notar que de la conjunción de las proposiciones se determinan los casos con una única solución

Recurrencias lineales homogéneas de coef. ctes.

- Forma:

$$\boxed{u_n=a_1u_{n-1}+...+a_ku_{n-k},\ n>k,a_i\in R} \quad \textbf{(1)}$$

- Método del polinomio característico
- **Definición**: ecuación característica

ec.característica

$$\boxed{r^k=a_1r^{k-1}+...+a_{k-1}r^1+a_k} \quad \textbf{(2)}$$

(3)

$$\boxed{P(r)=r^k-a_1r^{k-1}-...-a_{k-1}r^1-a_k}$$

pol.característico

Recurrencias lineales homogéneas de coef. ctes.

- Proposición 3:

Las soluciones de (1) forman un e.v.

- Proposición 4:

Si P(r) tiene **k raíces distintas**, $r_1, ..., r_k$, entonces, las k sucesiones

$$(r_1^n)_{n\geq 0}, ..., (r_k^n)_{n\geq 0}$$

forman una base del e.v. de soluciones. Toda solución es de la forma

$$u_n = \sum_{i=1}^{k} \lambda_i r_i^n$$

con los λ_i determinados por las condiciones iniciales

Recurrencias lineales homogéneas de coef. ctes.

- Ejemplo: resolver la recurrencia

$$\boxed{\begin{array}{l} u_0=0 \\ u_1=1 \\ u_n=3u_{n-1}+4u_{n-2}\ , \quad \text{si } n\geq 2 \end{array}}$$

  - Polinomio característico:

    $$P(r)=r^2-3r-4=(r-4)(r+1)$$

  - Raíces:

    $$r_1=4 \quad r_2=-1 \text{ (mult. 1)}$$

  - Solución:

    $$u_n= \lambda_1 r_1^n + \lambda_2 r_2^n= \lambda_1 4^n + \lambda_2(-1)^n$$

  - Aplicando condiciones iniciales:

    $$u_0= 0 = \lambda_1 + \lambda_2$$
    $$u_1= 1 = 4\lambda_1 - \lambda_2$$

  - La solución definitiva:

$$\boxed{u_n= \frac{1}{5} 4^n - \frac{1}{5} (-1)^n}$$

Recurrencias lineales homogéneas de coef. ctes.

- Ejercicio 1: resolver

```
u₀=0
u₁=1
uₙ = uₙ₋₁+ uₙ₋₂ ,    si n≥2
```

- Ejercicio 2: resolver

```
uₙ = 3uₙ₋₁ - uₙ₋₂ ,    si n≥2
```

- Ejercicio 3:

Sea $\Sigma=\{a,b\}$ y sea $B\subseteq\Sigma^*$ definido como

(B) $a\in B, b\in B$

(I) $w\in B \Rightarrow abw\in B$ y $baw\in B$

1) Escribir 6 elementos de B

2) Demostrar lo siguiente: $w\in B \Rightarrow |w|$ es impar

3) ¿Es cierto el recíproco?

4) Sea $u_n=\#(\{w \mid w\in B \text{ y } |w|=n\})$

4.1) Calcular u_1 y u_2

4.2) Dar la relación de recurrencia que genere u_n

4.3) Resolver la relación

Recurrencias lineales homogéneas de coef. ctes.

- Proposición 4:

Si P(r) tiene **p raíces distintas**, $r_1, \ldots, r_p$, de multiplicidades $m_1, \ldots, m_p$,

entonces, las k sucesiones

$$(r_1^n)_{n\geq 0}, (nr_1^n)_{n\geq 0}, \ldots, (n^{m_1-1}\, r_1^n)_{n\geq 0}$$

....

$$(r_p^n)_{n\geq 0}, (nr_p^n)_{n\geq 0}, \ldots, (n^{m_p-1}\, r_p^n)_{n\geq 0}$$

forman una base del e.v. de soluciones. Toda solución es de la forma

$$u_n = \sum_{j=1}^{p} P_j(n) r_j^n$$

siendo $P_j(n)$ un polinomio de grado m_j-1, a determinar por las condiciones iniciales

Recurrencias lineales homogéneas de coef. ctes.

- Ejemplo: resolver la recurrencia

$$\boxed{\begin{array}{l} u_0=1 \\ u_1=2 \\ u_2=5 \\ u_n= 4u_{n-1}- 5u_{n-2} + 2u_{n-3}\ , \quad \text{si } n\geq 3 \end{array}}$$

  - Polinomio característico:

$$P(r)=r^3-4r^2+5r-2=(r-1)^2(r-2)$$

  - Raíces:

$$r_1=1 \text{ (mult. 2) } \quad r_2=2 \text{ (mult. 1)}$$

  - Solución:

$$u_n= \underbrace{P_1(n)}_{\downarrow \atop \text{pol. grado 1}}1^n + \underbrace{P_2(n)}_{\downarrow \atop \text{pol. grado 0}}2^n$$

$$u_n= (\alpha n+\beta)1^n + \delta 2^n$$

  - Aplicando condiciones iniciales:

$$u_0= 1 = \beta+\delta$$
$$u_1= 2 = \alpha+\beta+2\delta$$
$$u_2= 5 = 2\alpha+\beta+4\delta \quad (\alpha =-1 ,\beta=-1,\delta=2)$$

  - Solución: $\boxed{u_n= \quad 2^{n+1} - n - 1}$

Recurrencias lineales NO homogéneas de coef. ctes.

- También llamadas con “segundo término”
- Forma general:

$$\boxed{u_n=a_1u_{n-1}+...+a_ku_{n-k}+\mathbf{b(n)},\ n>k, a_i\in R}$$

- Solución del caso general es compleja, y pasa por encontrar previamente una solución particular
- Estudiaremos casos particulares de b(n)
  - cubren la mayor parte de nuestros problemas
- Método del polinomio característico
  - Aplicable cuando

$$b(n)=\sum_{i=1}^{m} b_i^n\ P_i(n)$$

  - donde: $b_i\neq 0$
    P_i polinomio de grado d_i

Recurrencias lineales NO homogéneas de coef. ctes.

- Construir el polinomio característico

$$P(r)=(r^k-a_1r^{k-1}-...-a_{k-1}r^1-a_k)\prod_{i=1}^{m}(r-b_i)^{di+1}$$

- Proposición 5:

Si P(r) tiene **p raíces distintas**, $r_1,...,r_p$, de multiplicidades $m_1,...,m_p$,

entonces las soluciones son de la forma

$$u_n= \sum_{j=1}^{p} Q_j(n)r_j^n$$

siendo $Q_j(n)$ un polinomio de grado m_j-1

Recurrencias lineales NO homogéneas de coef. ctes.

- Ejemplo: resolver la recurrencia

$$\boxed{\begin{array}{l} u_0=1 \\ u_n= 2u_{n-1}+ 3^n, \quad \text{si } n\geq 1 \end{array}}$$

  - Polinomio característico:

    $$P(r)=(r-2)(r-3)$$

  - Raíces:

    $$r_1=2 \qquad r_2=3 \text{ (mult. 1)}$$

  - Solución:

    $$u_n= Q_1(n)2^n + Q_2(n)3^n$$

    como multip.=1 pol. de grado 0

    $$u_n= \alpha 2^n + \beta 3^n$$

  - Aplicando condiciones iniciales:

    $$u_0= 1 = \alpha+\beta$$

    $$u_1= 5 = 2\alpha+3\beta$$

    ec. para n=1

  - Solución:

    $$\boxed{u_n=-2^{n+1} + 3^{n+1}}$$

Recurrencias lineales NO homogéneas de coef. ctes.

- Un caso particular muy frecuente

$u_n = c.n^k$ cuando $0 \leq n < b$

$u_n = au_{n-b} + c.n^k$ cuando $b \leq n$

- La solución de la recurrencia da

$u_n \in \Theta \in (n^k)$ si $a<1$

$u_n \in \Theta(n^{k+1})$ si $a=1$

$u_n \in \Theta(a^{n \text{ div } b})$ si $a>1$

- Ejemplos:
  - factorial, potencia
  - suma componentes de un vector
  - muchos más de los vistos

Recurrencias lineales NO homogéneas de coef. ctes.

```
Función factorial(E n:entero)
                        dev (r:entero)
{Pre: n≥0
 Post: r=n!}
Principio
   Sel
     (n=0)∨(n=1): r:=1
     n≥1: r:=n*factorial(n-1)
   FSel
   dev(r)
Fin
```

```
Función divide(E a,b:entero)
                dev (q,r:entero)
{Pre: a≥0 ∧ b>0
 Post: a=q*b+r ∧ 0≤r<b}
Principio
 Sel
     a<b: <q,r>:=<0,a>
     a≥b: <q,r>:=divide(a-b,b)
          <q,r>:=<q+1,r>
   FSel
   dev(<q,r>)
Fin
```

Recurrencias lineales de coeficientes variables

- No se conoce una solución general. Estudiamos únicamente el caso de recurrencia lineal de coef. variables de orden 1
- Forma general $\boxed{A(n)u_n=B(n)u_{n-1}+C(n)}$
- Dividiendo por A(n) $\boxed{u_n=b(n)u_{n-1}+c(n)}$
- Tomando $\boxed{f(n)=\prod_{i=1}^{n} 1/b(i)}$
- se define $(v_n)_{n\geq 0}$ donde $\boxed{\begin{array}{l} v_n=f(n)u_n\text{ , } n>0 \\ v_0=u_0 \end{array}}$
- que satisface $\boxed{v_n=v_{n-1}+f(n)c(n)}$
- Resolviendo $\boxed{v_n=v_0+\sum_{k=1}^{n} f(k)c(k)}$
- De donde

$$\boxed{u_n=(1/f(n))v_n=\prod_{i=1}^{n} b(i)(u_0+\sum_{k=1}^{n} f(k)c(k))}$$

Recurrencias de partición y cambio de variable

- Aparecen frecuentemente en la aplicación de la técnica de “divide y vencerás”
- Forma general:

$$u_n=bu_{n/a}+c(n)$$

donde n se restringe a po

$$a^k=n, v_k=u_{a^k}$$

- Cambio de variable:

$$u_{a^k} = bu_{a^{k-1}}+c(a^k)$$

- Nueva recurrencia:

$$v_k = bv_{k-1} + c(a^k)$$

- O:

$$v_k=b^k(v_0+\sum_{j=1}^{k}c(a^j)/b^j)=v_0b^k+\sum_{j=1}^{k}b^jc(a^{k-j+1})$$

- Como $a^k=n$ $k=\log$ n

$$u_n=v_0b^{\log_a n}+\sum_{j=1}^{\log_a n} b^jc(n/a^{j-1})$$

- Resolver $u_n=4u_{n/2}+n^2$ para $n=2^k$

Recurrencias de partición y cambio de variable

- Un caso particular muy frecuente

$u_n= c.n^k$ cuando $1\le n<a$

$u_n= b.u_{n/a}+c.n^k$ cuando $a\le n$

- La solución de la recurrencia da

$u_n\in \Theta(n^k)$ si $b<a^k$

$u_n\in \Theta(n^k.\lg_a n)$ si $b=a^k$

$u_n\in \Theta(n^{\lg_a b})$ si $b>a^k$

- Ejemplos:
  - búsqueda dicotómica
  - “divide2” (solución más eficiente a divide)
  - “potencia2” (solución más eficiente a pot)

Recurrencias de partición y cambio de variable

```
Función bD(E v:vect;E x,pI,pD:entero)
           dev (e:booleano;p:entero)
--Pre: orden(v,pI,pD)∧(1 ≤ pI ≤ pD+1 < N)}
--Post: (e →(pI ≤ p ≤ pD)∧(x=v[p])) ∧
--       (¬e →(pI ≤ p ≤ D+1)  ∧
--          (v[pI..p-1]<x) ∧
--          (x<v[p..pD]))
Principio
   Sel
     pI > pD: <e,p>:=<False,pI>
     pI ≤ pD:
       m:=(pI+pD) DIV 2
       Sel
         x=v[m]: <e,p>:=<True,m>
         x>v[m]: <e,p>:=bD(v,m+1,pD)
         x<v[m]: <e,p>:=bD(v,pI,m-1)
       FSel
   FSel
   dev(e,p)
Fin
```

Recurrencias de partición y cambio de variable

```
Función divide2(E a,b:entero)
                dev (q,r:entero)
--Pre: a≥0 ∧ b>0
--Post: a=q*b+r ∧ 0 ≤ r < b
Principio
   Sel
     a<b: <q,r>:=<0,a>
     a≥b: <q,r>:=divide2(a,2b)
         -- a=q*2b+r ∧ 0 ≤ r < 2b
          Sel
              r<b: <q,r>:=<2q,r>
              r≥b: <q,r>:=<2q+1,r-b>
          FSel
   FSel
   dev(<q,r>)
Fin
```

Recurrencias y transformación de la imagen

- Ejemplo: Sea la recurrencia

NO lineal

$u_1=6$
$u_n=n(u_{n/2})^2+c(n)$, n potencia de 2

- Cambio de variable: $v_k=u_{2^k}$
- Queda:

$v_0=6$
$v_k=2^k(v_{k-1})^2$, $k>0$

  - No es lineal, y tiene un coef. no constante
- Cambio de imagen: $V_k=lg(v_k)$

$V_0=lg\ 6$
$V_k=k+2V_{k-1}$, $k>0$

- Ec. característica: $(x-2)(x-1)^2 = 0$
- Sol: $V_k=c_1 2^k+c_2 1^k+c_3 k 1^k$
- Como: $V_0=1+lg\ 3, V_1=3+2lg\ 3, V_2=8+4lg\ 3$

$V_k=(3+lg\ 3)2^k-k-2$

$u_n=2^{3n-2}\ 3^n\ /n$

¿Y si casi todo falla?

- A veces, no se puede aplicar ninguno de los métodos anteriores
- Podemos intentar usar la intuición (y el "ingenio")
  - "aventurar" una forma
  - utilizar los métodos de inducción
- Ejemplo: Resolver

$$\boxed{\begin{array}{l} u_0=a \\ u_n=u_{n-1}+b,\ n>0 \end{array}}$$

- Conjetura: u_n es un polinomio g=2

$$\boxed{u_n=an^2+bn+c}$$

- Demostración: por inducción

TEMA 4: Diseño de algoritmos iterativos

1) Introducción

2) Recursividad final con Post constante y solución iterativa

3) Corrección de programas iterativos

4) Transformación recursivo-iterativo

- Caso 1: recursividad final
- Caso 2: recursividad lineal con inversa de la “sucesora”
- Caso 3: recursividad lineal y uso de una pila de datos

5) Derivación de algoritmos iterativos

Introducción

- En general, soluciones recursivas
  - menos eficientes
  - más claras y sencillas

  que las iterativas

`Pensar en recursivo`
`Implementar en iterativo`

- A veces interesante/obligatorio:
  - diseño recursivo
  - implementación iterativa
- Para programas iterativos:
  - semántica de la iteración
  - diseño iterativo
  - transformación de recursivo a iterativo

Rec. Final+Post Cte y solución iterativa

- Consideremos la función

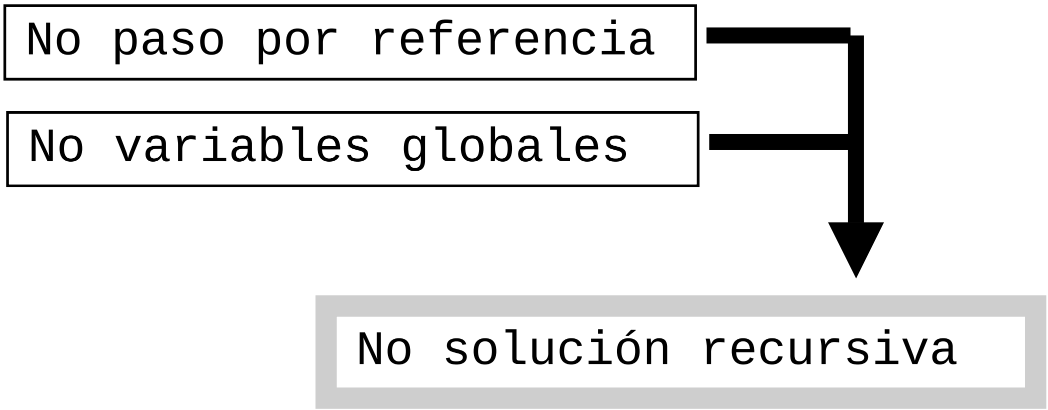
```
Función f(E X:tX;E y:tY) dev (r:tR)
--Pre(X,y)
--Post(X,r)
Principio
   Sel
      $B_t$(X,y): r:=triv(X,y)
      $B_{nt}$(X,y): r:=f(X,s(X,y))
   FSel
   dev(r)
Fin
```

- Donde:
  - recursividad final
  - X no varía entre llamadas
  - Post constante
- Notar que:
  - cada invocación almacena su valor y correspondiente (lo mismo para X)
  - cuando se invoca a la siguiente, el anterior y ya no es necesario para nada

Rec. Final+Post Cte y solución iterativa

- Tratamos de que

  todas la invocaciones compartan la misma `y` `(misma zona de memoria)`



`No paso por referencia`

`No variables globales`

`No solución recursiva`

- Habrá que buscar una solución iterativa

Rec. Final+Post Cte y solución iterativa

- Planteamos como solución iter.:

```
Función f(E X:tX;E y:tY) dev (r:tR)
--Q(X,y)
--R(X,r)
Principio
   Sel
      $B_t$(X,y): r:=triv(X,y)
      $B_{nt}$(X,y): r:=f(X,s(X,y))
   FSel
   dev(r)
Fin
```

```
Función fIter(E X:tX;E y:tY) dev (r:tR)
--Q(X,y)
--R(X,r)
Principio
   Mq $B_{nt}$(X,y)
      y:=s(X,y)
   FMq
                     --Q1
   r:=triv(X,y)
   dev(r)
Fin
```

Rec. Final+Post Cte y solución iterativa

- Correción: asumamos que la solución recursiva es correcta.
- ¿Lo es la iterativa?

antes de evaluar cada vez la guarda
$\mathtt{Pre}(\underline{X},\underline{y})$

tiempo finito en el bucle

$$\mathtt{Pre}(\underline{X},\underline{y}) \wedge \mathtt{B}_t(\underline{X},\underline{y}) \rightarrow \mathtt{Post}(\underline{X},\mathtt{triv}(\underline{X},\underline{y}))$$

- Bucle asociado al programa recursivo final con Post cte.
- Ya hemos manejado los aspectos fundamentales:
  - función de cota (limitadora)
  - invariante de bucle
- Sintaxis:

```
Mq B
      S
FMq
```

Corrección de programas iterativos

- Invariante de bucle:

```
Predicado tal que:
  - cierto antes de cada una de
    las iteraciones del bucle
  - cierto inmediatamente después
    del bucle
```

- Lo usaremos para "pasar información" entre los estados anterior y posterior del bucle

`¡¡Y seguir trabajando!!`

- Función de cota
  (variante del bucle):

t: $\mathcal{E} \rightarrow Z$ tal que:

- $\geq$ 0 durante ejecución del bucle

- estrictamente decreciente entre dos estados consecutivos

Corrección de programas iterativos

- Ejemplo:

```
--Ng≥0
   <w,j,n>:=<1,25,N>
   Mq n>0
      w:=w*n
      n:=n-1
   FMq
```

- `t(n,w,j)=n` es función de cota
- Son invariantes:

```
I1: w≤N!
I2: j<27
I3: w=∏α∈{n+1..N}.α
```

- No son invariantes:

```
I4: w<n
I5: w<174
I6: n>0
```

Corrección de programas iterativos

- Sean:
  - `I` un invariante para el bucle
  - `t` una función de cota para el bucle

Entonces, el bucle verifica

```
--I
  Mq B
     S
  FMq
--I ∧ ¬B
```

Asumamos que termina

- Por lo tanto, para verificar

```
1) Q → I
```

```
2) I ∧ ¬B → R
```

→

```
--Q
  Mq B
     S
  FMq
--R
```

- Se suele hablar de corrección total y corrección parcial

Corrección de programas iterativos

- La semántica se puede expresar por la siguiente regla:

$$\boxed{\frac{\{I\wedge B\}S\{I\},I\wedge B\rightarrow t\geq 0,\{I\wedge B\wedge t=T\}S\{t<T\}}{\{I\}\ \mathbf{Mq}\ B\ S\ \mathbf{FMq}\ \{I\wedge\neg B\}}}$$

- Si además $\boxed{Q\ \rightarrow I}$

- y $\boxed{I\wedge\neg B\ \rightarrow R}$

- entonces

```
--Q
     Mq B
        S
     FMq
--R
```

Corrección de programas iterativos

- La forma "normal" de especificación

```
--Q
  inicialización
  Mq B
       S
  FMq
--R
```

- Verificación:

`1) buscar:` **`t e I`**

`2) probar:` **$I \wedge \neg B \rightarrow R$**

`3) probar:` **`{Q} inicialización {I}`**

`4) probar:` **$\{I \wedge B\}\ S\ \{I\}$**

`5) probar:` **$I \wedge B \rightarrow t \geq 0$**

`6) probar:` **$\{I \wedge B \wedge t=T\}\ S\ \{t<T\}$**

Corrección de programas iterativos

- Lo difícil: encontrar el invariante
  - suficientemente fuerte para poder deducir la Post después del bucle
  - suficientemente débil para no restringir el dominio

- Ejemplo:

```
--N≥0
    <w,n>:=<1,N>
    Mq n>0
       w:=w*n
       n:=n-1
    FMq
--w=N!
```

Corrección de programas iterativos

- Ejercicio 1: Probar la corrección del siguiente algoritmo

```
Constantes n=....   --n>=1
Tipos vect=vector[1..n] de entero
Función suma(E b:vect) dev (s:entero)
--Pre: True
--Post: s=∑α∈{1..n}.b[α]
Variables i:entero
Principio
   <i,s>:=<n,0>
   Mq i≠0
      <i,s>:=<i-1,s+b[i]>
   FMq
   dev(s)
Fin
```

```
s:=s+b(i)
i:=i-1
```

Corrección de programas iterativos

- Como programa anotado, debemos escribirlo así:

```
Constantes n=....
Tipos vect=vector[1..n] de entero
Función suma(E b:vect) dev (s:entero)
--Pre: True
--Post: $s=\sum \alpha\in\{1..n\}.b[\alpha]$
Variables i:entero
Principio
   <i,s>:=<n,0>
          --I: $s=\sum \alpha\in\{i+1..n\}.b[\alpha]$
          --              $\wedge$ $0\leq i\leq n$
          --t(i)=i
   Mq $i\neq 0$
      <i,s>:=<i-1,s+b[i]>
   FMq
   dev(s)
Fin
```

Desde ahora, en cualquier etapa del diseño,
¡¡Todo programa anotado!!

Corrección de programas iterativos

- Ejercicio 2: ¿Es correcta la siguiente función?

```
Función mult(E a,b:entero)
                    dev (p:entero)
--Pre: a≥0 ∧ b≥0
--Post: p=a*b
Vars a’,b’:entero
Principio
   <a’,b’,p>:=<a,b,0>
   Mq a’>0
      Si impar(a’)
        ent  p:=p+b’
      FSi
      <a’,b’>:=<a’ DIV 2, 2b’>
   FMq
   dev(p)
Fin
```

Corrección de programas iterativos

- Ejercicio 3: ¿Es correcto el siguiente algoritmo?

```
Constantes n=....
Tipos vect=vector[1..n] de entero
Función bLA(E v:vect;E x:entero)
                  dev (i:entero)
--Pre: $\exists\alpha\in\{1..n\}.v[\alpha]=x$
--Post: $v[i]=x \wedge \forall\alpha\in\{1..i-1\}.v[\alpha]\neq x$
Principio
   i:=1
   Mq $v[i]\neq x$
      i:=i+1
   FMq
   dev(i)
Fin
```

Corrección de programas iterativos

- Ejercicio 4: Diseñar los siguientes algoritmos

```
Función pot(E a,b:entero)
                        dev (p:entero)
--Pre: $a \neq 0 \wedge b \geq 0$
--Post: $p=a^b$
```

```
Constantes n=....
Tipos vect=vector[1..n] de entero
Función esOrd(E a:vect)
                   dev (o:booleano)
--Pre: True
--Post: $o=\forall \alpha \in \{1..n-1\}.v[\alpha] \leq v[\alpha+1]$
```

```
Función suma(E n:entero) dev (s:real)
--Pre: $n \geq 1$
--Post: $s=\sum \alpha \in \{1..n\}.1/\alpha^2$
```

Corrección de programas iterativos

- Diseñar los siguientes algoritmos

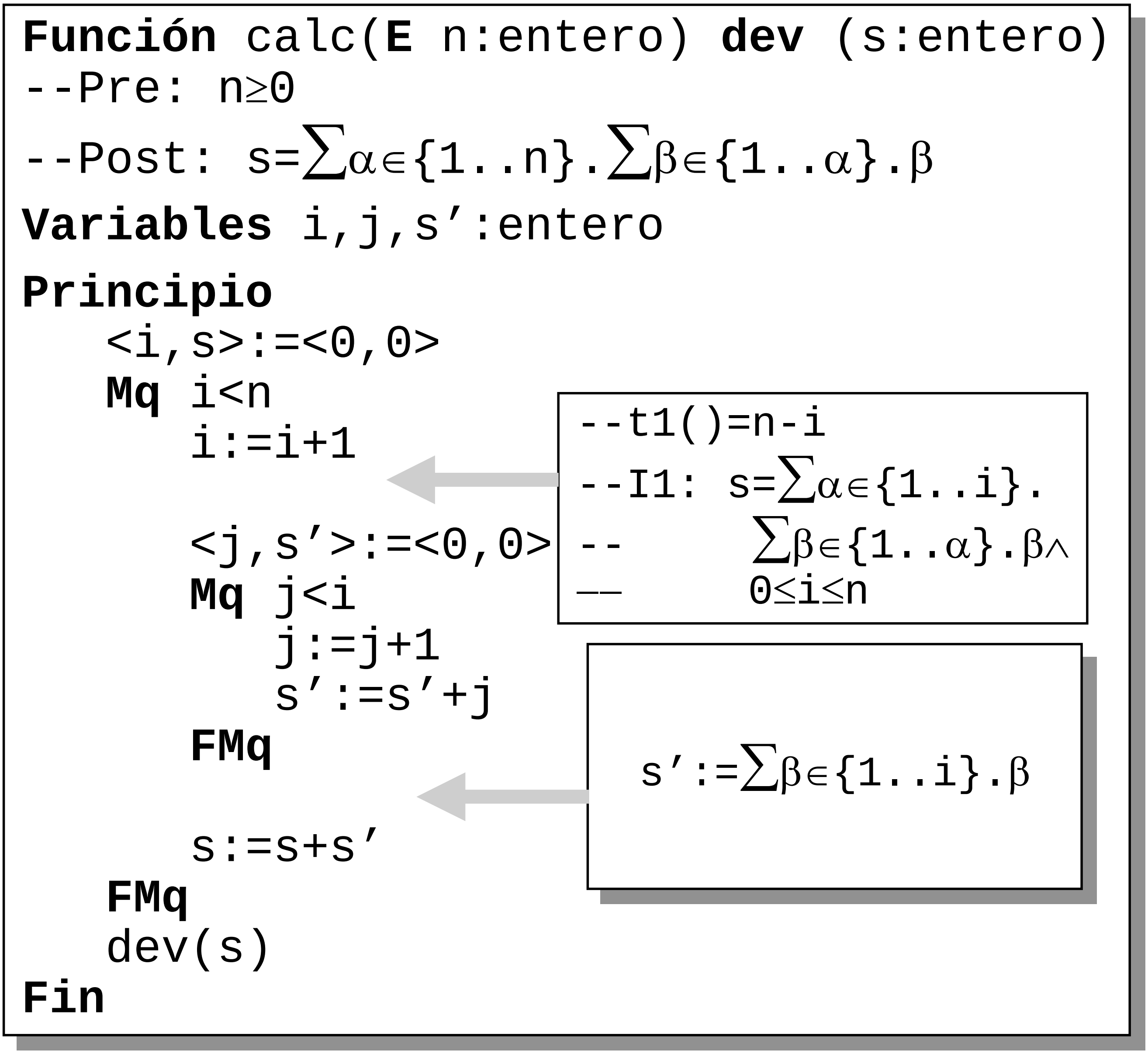
```
Función eAX(E x,$\varepsilon$:real)
                  dev (k:entero;s:real)

--Pre: $0.0<\varepsilon \wedge \varepsilon\leq 1.0 \wedge x>0.0$
--Post: $s=\sum\alpha\in\{0..k\}.x^{\alpha}/(\alpha!) \wedge$

          $x^{k}/(k!)\geq\varepsilon \wedge x^{k+1}/((k+1)!)<\varepsilon$
```

```
Función calc(E n:entero) dev (s:entero)
--Pre: $n\geq 0$
--Post: $s=\sum\alpha\in\{1..n\}.\sum\beta\in\{1..\alpha\}.\beta$
```

```
Función esCap(E n:entero)
                       dev (eC:booleano)
--Pre: $n\geq 0$
--Post: s=¿Es “s” un número capicúa?
```

Corrección de programas iterativos

- Una posible solución.



```
Función calc(E n:entero) dev (s:entero)
--Pre: n≥0
--Post: s=∑α∈{1..n}.∑β∈{1..α}.β
Variables i,j,s’:entero
Principio
   <i,s>:=<0,0>
   Mq i<n
      i:=i+1

      <j,s’>:=<0,0>
      Mq j<i
         j:=j+1
         s’:=s’+j
      FMq

      s:=s+s’
   FMq
   dev(s)
Fin
```

```
--t1()=n-i
--I1: s=∑α∈{1..i}.
--      ∑β∈{1..α}.β∧
——      0≤i≤n
```

```
s’:=∑β∈{1..i}.β
```

Corrección de programas iterativos

```
--Q': $0 \le i$
    <j,s'>:=<0,0>

    **Mq** j<i
      j:=j+1
      s':=s'+j
    **FMq**

--R': s':=$\sum \beta \in \{1..i\}.\beta$
```

```
--t2=i-j
--I2: s'=$\sum \alpha \in \{1..j\}.\alpha$
        $\wedge$ $0 \le j \le i$
```

Transformación recursivo-iterativo

- Puede haber distintas causas para tratar de obtener una versión iterativa de un algoritmo recursivo:
  - el lenguaje no soporta la recursividad
  - versión recursiva poco eficiente
- Sólo caso de recursividad lineal
- Distinguiremos tres casos:
  - final
  - lineal con inversa de la función “sucesora”
  - lineal utilizando una pila de datos

Transformación recursivo-iterativo: lineal final

- Transformación recursivo final
- Ya vimos el siguiente caso. ¿Correcto?

```
Función f(E x:tX) dev (r:tR)
--Q(x)
--R(x,r)
Principio
   Sel
      $B_t$(x): r:=triv(x)
      $B_{nt}$(x): r:=f(s(x))
   FSel
   dev(r)
Fin
```

```
Función fI(E x:tX) dev (r:tR)
--Q(x)
--R(x,r)
Variables y:tX
Principio
   y:=x
   Mq $B_{nt}$(y)
      y:=s(y)
   FMq
   r:=triv(y)
   dev(r)
Fin
```

I: $Q(y) \wedge f(x)=f(y)$

Transformación recursivo-iterativo: lineal final

```
Función f(E x:tX) dev (r:tR)
--Q(x)
--R(x,r)
Principio
   Sel
      B_t(x):  r:=triv(x)
      B_nt(x): r:=f(s(x))
   FSel
   dev(r)
Fin
```

$Q(\underline{x}) \rightarrow B_t(\underline{x}) \vee B_{nt}(\underline{x})$

$B_{nt}(\underline{x}) \wedge Q(\underline{x}) \rightarrow Q(s(\underline{x}))$

$Q(\underline{x}) \wedge B_t(\underline{x}) \rightarrow R(\underline{x},triv(\underline{x}))$

$Q(\underline{x}) \wedge B_{nt}(\underline{x}) \wedge R(s(\underline{x}),\underline{r}') \rightarrow R(\underline{x},c(\underline{x},\underline{r}'))$

Encontrar $t:D_{tX} \rightarrow \mathbf{Z}$

t.q. $Q(\underline{x}) \rightarrow t(\underline{x})\ 0$

$Q(\underline{x}) \wedge B_{nt}(\underline{x}) \rightarrow t(s(\underline{x}))<t(\underline{x})$

```
Función fIter(E x:tX) dev (r:tR)
--Q(x)
--R(x,r)
Variables xAux:tX
Principio
   xAux:=x
   Mq B_nt(xAux)
      xAux:=s(xAux)
   FMq
   r:=triv(xAux)
   dev(r)
Fin
```

I: $Q(\underline{xAux}) \wedge f(\underline{xAux})=f(\underline{x})$

buscar: I y $t:D \rightarrow \mathbf{Z}$

$I \wedge \neg B \rightarrow R$

$\{Q\}$ inic $\{I\}$

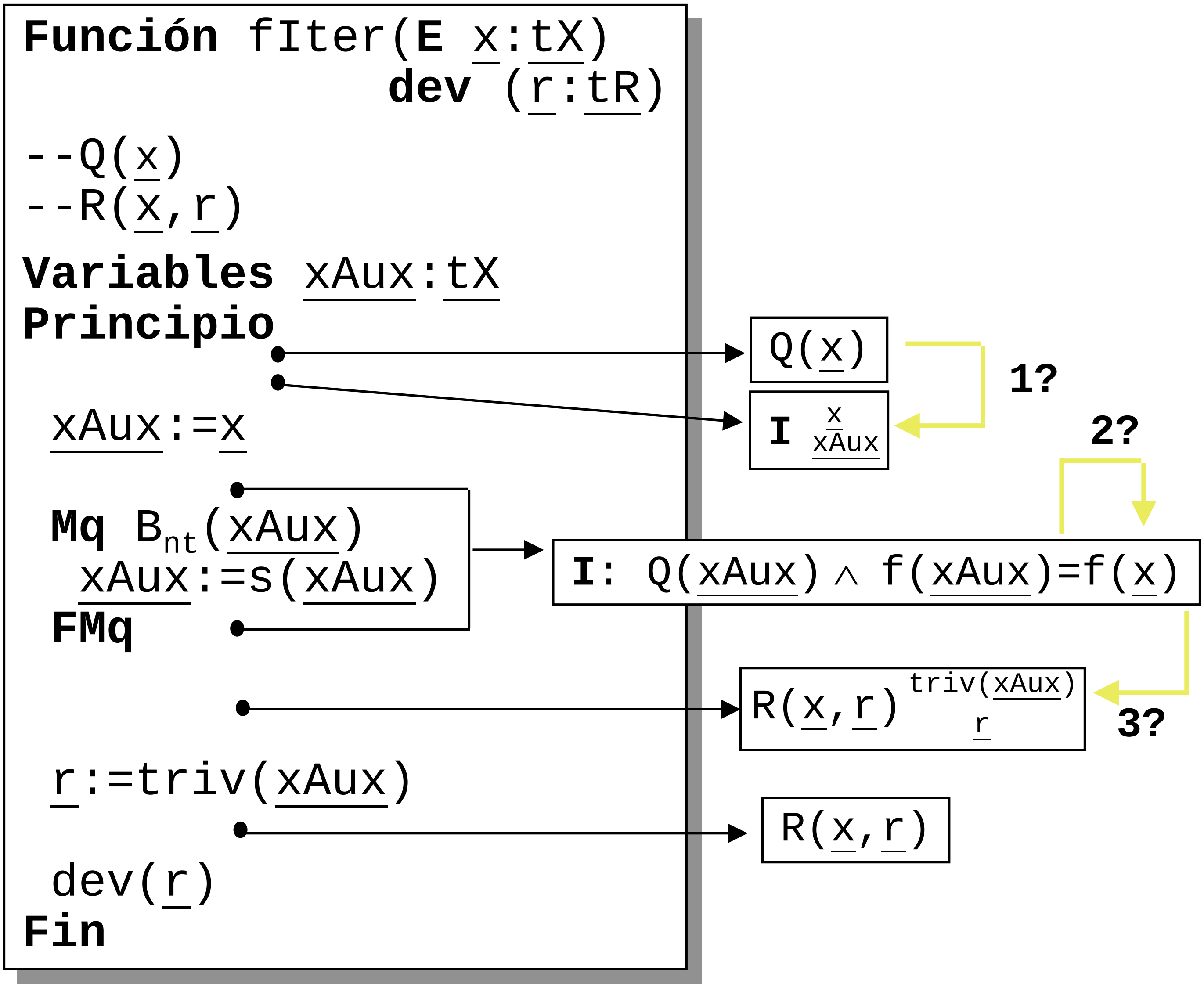
$\{I \wedge B\}$ S $\{I\}$

$I \wedge B \rightarrow t \geq 0$

$\{I \wedge B \wedge t=T\}$ S $\{t<T\}$

Transformación recursivo-iterativo: lineal final

- Esquema de verificación:



```
Función fIter(E x:tX)
                dev (r:tR)
--Q(x)
--R(x,r)
Variables xAux:tX
Principio

  xAux:=x

  Mq Bnt(xAux)
    xAux:=s(xAux)
  FMq

  r:=triv(xAux)

  dev(r)
Fin
```

$Q(\underline{x})$

$\mathbf{I}\ {}^{\underline{x}}_{\underline{xAux}}$

$\mathbf{I}\text{: } Q(\underline{xAux}) \wedge f(\underline{xAux})=f(\underline{x})$

$R(\underline{x},\underline{r})\ {}^{triv(\underline{xAux})}_{\underline{r}}$

$R(\underline{x},\underline{r})$

Transformación recursivo-iterativo: lineal final

1) $¿Q(\underline{x}) \rightarrow \mathbf{I}^{\underline{x}}_{\underline{xAux}}\ ?$

$$\boxed{Q(x) \rightarrow Q(\underline{x}) \wedge f(\underline{x})=f(\underline{x})}$$

2)

$$\boxed{\begin{array}{l} \text{--}B_{nt}(\underline{xAux}) \wedge I \\ \quad \underline{xAux}:=s(\underline{xAux}) \\ \text{--}I \end{array}}$$

$$\leftrightarrow$$

$$\boxed{B_{nt}(\underline{xAux}) \wedge I \rightarrow I^{s(\underline{xAux})}_{\underline{xAux}}}$$

$$\leftrightarrow$$

$$\boxed{\begin{array}{l} B_{nt}(\underline{xAux}) \wedge Q(\underline{xAux}) \wedge f(\underline{xAux})=f(\underline{x}) \rightarrow \\ \qquad Q(s(\underline{xAux})) \wedge f(s(\underline{xAux}))=f(\underline{x}) \end{array}}$$

Transformación recursivo-iterativo: lineal final

3) ¿$\neg B_{nt}(\underline{xAux}) \wedge I \rightarrow$
$R(\underline{x},triv(\underline{x}))$?

$$\boxed{\begin{array}{r}\neg B_{nt}(\underline{xAux}) \wedge Q(\underline{xAux}) \wedge f(\underline{xAux})=f(\underline{x}) \rightarrow \\ R(\underline{x},triv(\underline{x}))\end{array}}$$

4) ¿Terminación?

Transformación recursivo-iterativo: lineal final

- Ejercicio: Obtener dos versiones iterativas de

```
Constantes n=.....
Tipos      vect=vector[1..n] de entero

Función sC(E v:vect;E i,w:entero)
                                dev (s:entero)
--Pre: 0≤i≤n
--Post: s=w+∑α∈{1..i}.v[α]
Principio
   Sel
     i=0: s:=w
     i>0: s:=sC(v,i-1,v[i]+w)
   FSel
   dev(s)
Fin
```

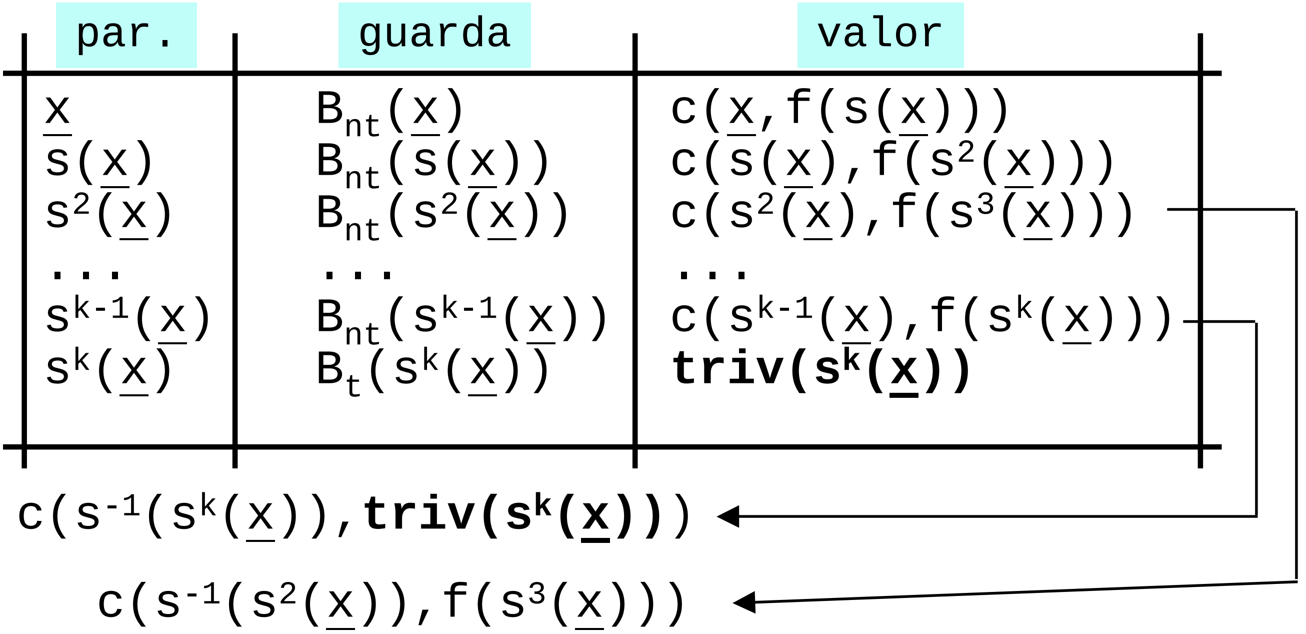
Transformación recursivo-iterativo: lineal final

- Ejercicio: Obtener una versión iterativa de

```
Tipos fichEnt=fichero de entero
Función comp(E f:fichEnt;
         E p,pos:entero) dev (c:entero)
--Pre: f=(<d_1,...,d_n>,p,L) ∧ 1≤p≤pos≤n
--Post: f=(<d_1,...,d_n>,pos+1,L)∧ c=d_pos
Principio
   Sel
     p=pos: leer(f,c)
     p<pos: leer(f,c) --por avanzar
            c:=comp(f,p+1,pos)
   FSel
   dev(c)
Fin
```

Transformación recursivo-iterativo: no final con inversa

```
Función f(E $\underline{x}$:$\underline{tX}$) dev ($\underline{r}$:$\underline{tR}$)
--Pre:  Q($\underline{x}$)
--Post: R($\underline{x}$,$\underline{r}$)
Principio
   Sel
      $B_t(\underline{x})$: $\underline{r}$:=triv($\underline{x}$)
      $B_{nt}(\underline{x})$: $\underline{r}$:=c($\underline{x}$,f(s($\underline{x}$)))
   FSel
   dev($\underline{r}$)
Fin
```



par.	guarda	valor
$\underline{x}$	$B_{nt}(\underline{x})$	$c(\underline{x},f(s(\underline{x})))$
$s(\underline{x})$	$B_{nt}(s(\underline{x}))$	$c(s(\underline{x}),f(s^2(\underline{x})))$
$s^2(\underline{x})$	$B_{nt}(s^2(\underline{x}))$	$c(s^2(\underline{x}),f(s^3(\underline{x})))$
...	...	...
$s^{k-1}(\underline{x})$	$B_{nt}(s^{k-1}(\underline{x}))$	$c(s^{k-1}(\underline{x}),f(s^k(\underline{x})))$
$s^k(\underline{x})$	$B_t(s^k(\underline{x}))$	$\mathbf{triv}(\mathbf{s}^{k}(\underline{\mathbf{x}}))$

$c(s^{-1}(s^k(\underline{x})),\mathbf{triv}(\mathbf{s}^{k}(\underline{\mathbf{x}})))$

$c(s^{-1}(s^2(\underline{x})),f(s^3(\underline{x})))$

Transformación recursivo-iterativo: no final con inversa

```
Función f(E x:tX) dev (r:tR)
--Pre:  Q(x)
--Post: R(x,r)
Principio
   Sel
      $B_t$(x):   r:=triv(x)
      $B_{nt}$(x): r:=c(x,f(s(x)))
   FSel
   dev(r)
Fin
```

```
Función fI(E x:tX)
          dev (r:tR)
--Pre:  Q(x)
--Post: R(x,r)
Variables xAux:tX

Principio
   xAux:=x
   Mq $B_{nt}$(xAux)
      xAux:=s(xAux)
   FMq
   r:=triv(xAux)

   Mq xAux$\neq$x
      xAux:=$s^{-1}$(xAux)
      r:=c(xAux,r)
   FMq
   dev(r)
Fin
```

I1:Q(xAux) $\wedge$ SUC(xAux,x)

I2:I1 $\wedge$ R(xAux,r)

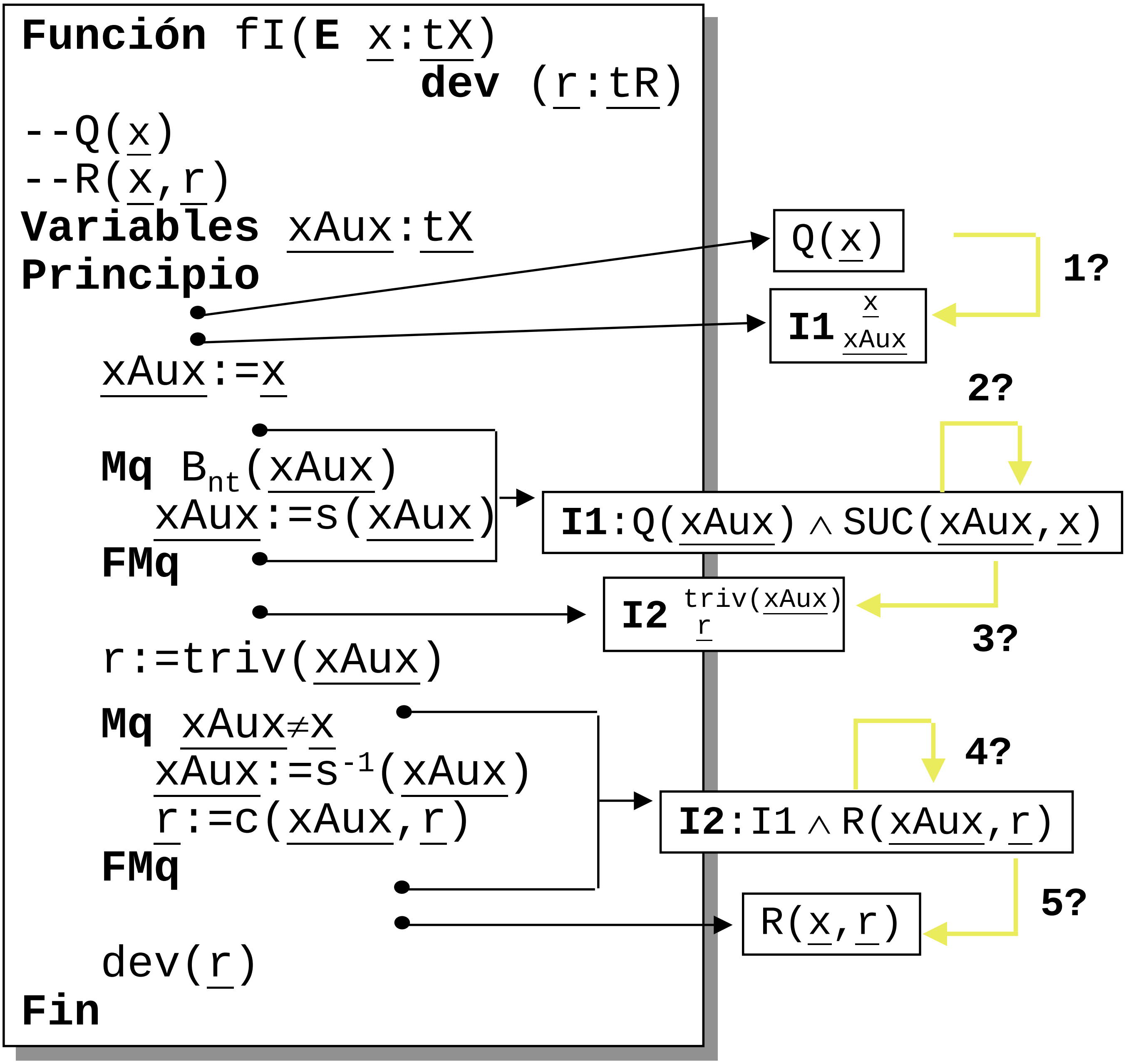
SUC(y,x)=

$\exists\alpha\in$**N**.y=s^{α}(x)$\wedge$

$\forall\beta\in$\{1..α-1\}.

(B_{nt}(s^{β}(x))$\wedge$Q(s^{β}(x)))

Transformación recursivo-iterativo: no final con inversa



```
Función fI(E x:tX)
               dev (r:tR)
--Q(x)
--R(x,r)
Variables xAux:tX
Principio
   xAux:=x
   Mq $B_{nt}$(xAux)
      xAux:=s(xAux)
   FMq
   r:=triv(xAux)
   Mq xAux≠x
      xAux:=$s^{-1}$(xAux)
      r:=c(xAux,r)
   FMq
   dev(r)
Fin
```

Q(x)

1?

$\mathbf{I1}\ \frac{x}{xAux}$

2?

I1:Q(xAux) ∧ SUC(xAux,x)

3?

$\mathbf{I2}\ \frac{triv(xAux)}{r}$

4?

I2:I1 ∧ R(xAux,r)

5?

R(x,r)

$$
\begin{aligned}
&SUC(y,x)= \\
&\quad \exists\alpha\in\mathbf{N}.\,y=s^{\alpha}(x)\wedge \\
&\qquad \forall\beta\in\{1..\alpha-1\}. \\
&\qquad\quad (B_{nt}(s^{\beta}(x))\wedge Q(s^{\beta}(x)))
\end{aligned}
$$

Transformación recursivo-iterativo: no final con inversa

1) $¿\mathtt{Q}(\underline{\mathtt{x}}) \rightarrow \mathbf{I1}^{\underline{\mathtt{x}}}_{\underline{\mathtt{xAux}}}\ ?$

$$\boxed{\mathtt{Q}(\underline{\mathtt{x}}) \rightarrow \mathtt{Q}(\underline{\mathtt{x}}) \wedge \mathtt{SUC}(\underline{\mathtt{x}},\underline{\mathtt{x}})}$$

2)

$$\boxed{\begin{array}{l} \mathtt{--B_{nt}}(\underline{\mathtt{xAux}}) \wedge \mathtt{I1} \\ \quad \underline{\mathtt{xAux}}\mathtt{:=s}(\underline{\mathtt{xAux}}) \\ \mathtt{--I1} \end{array}}$$

$$\leftrightarrow$$

$$\boxed{\mathtt{B_{nt}}(\underline{\mathtt{xAux}}) \wedge \mathtt{I1} \rightarrow \mathtt{I1}^{\mathtt{s}(\underline{\mathtt{xAux}})}_{\underline{\mathtt{xAux}}}}$$

$$\leftrightarrow$$

$$\boxed{\begin{array}{l} \mathtt{B_{nt}}(\underline{\mathtt{xAux}}) \wedge \mathtt{Q}(\underline{\mathtt{xAux}}) \wedge \mathtt{SUC}(\underline{\mathtt{xAux}},\underline{\mathtt{x}}) \rightarrow \\ \quad \mathtt{Q}(\mathtt{s}(\underline{\mathtt{xAux}})) \wedge \mathtt{SUC}(\mathtt{s}(\underline{\mathtt{xAux}}),\underline{\mathtt{x}}) \end{array}}$$

$$\boxed{\begin{array}{l} \mathtt{SUC}(\underline{\mathtt{y}},\underline{\mathtt{x}}) = \\ \quad \exists \alpha \in \mathbf{N}.\, \underline{\mathtt{y}} = \mathtt{s}^{\alpha}(\underline{\mathtt{x}}) \wedge \\ \qquad\qquad \forall \beta \in \{1..\alpha-1\}. \\ \qquad\qquad\quad (\mathtt{B_{nt}}(\mathtt{s}^{\beta}(\underline{\mathtt{x}})) \wedge \mathtt{Q}(\mathtt{s}^{\beta}(\underline{\mathtt{x}}))) \end{array}}$$

Transformación recursivo-iterativo: no final con inversa

3) $¿\neg B_{nt}(\underline{xAux}) \wedge I1 \rightarrow I2^{triv(\underline{xAux})}_{\underline{r}}\ ?$

$$\neg B_{nt}(\underline{xAux}) \wedge Q(\underline{xAux}) \wedge SUC(\underline{xAux},\underline{x}) \rightarrow$$
$$Q(\underline{xAux}) \wedge SUC(\underline{xAux},\underline{x}) \wedge R(\underline{xAux},triv(\underline{xAux}))$$

4)

```
--$\underline{xAux}°\underline{x} \wedge I2$
    $\underline{xAux}:=s^{-1}(\underline{xAux})$
    $\underline{r}:=c(\underline{xAux},\underline{r})$
--$I2$
```

$$\leftrightarrow$$

$$\underline{xAux}\neq\underline{x} \wedge I2 \rightarrow \left(I2^{c(\underline{xAux},\underline{r})}_{\underline{r}}\right)^{s^{-1}(\underline{xAux})}_{\underline{xAux}}$$

$$\leftrightarrow$$

$$\underline{xAux}\neq\underline{x} \wedge Q(\underline{xAux}) \wedge SUC(\underline{xAux},\underline{x}) \wedge R(\underline{xAux},\underline{r})$$
$$\rightarrow$$
$$Q(s^{-1}(\underline{xAux})) \wedge SUC(s^{-1}(\underline{xAux}),\underline{x}) \wedge$$
$$R(s^{-1}(\underline{xAux}),c(s^{-1}(\underline{xAux}),\underline{r}))$$

Transformación recursivo-iterativo: no final con inversa

5) ¿ $\neg(\underline{xAux} \neq \underline{x}) \wedge I2 \rightarrow R(\underline{x},\underline{r})$?

$$\boxed{\begin{array}{c} \neg(\underline{xAux} \neq \underline{x}) \wedge Q(\underline{xAux}) \wedge SUC(\underline{xAux},\underline{x}) \wedge R(\underline{xAux},\underline{r}) \\ \rightarrow \\ R(\underline{x},\underline{r}) \end{array}}$$

6) ¿Terminación?

Transformación recursivo-iterativo: no final con inversa

- Un ejemplo:

```
Const n=.....
Tipos vect=vector[1..n]
                  de entero

Función sC(E v:vect;
           E i:entero)
             dev (s:entero)
--Pre:  $0 \le i \le n$
--Post: s=$\sum \alpha \in \{1..i\}.v[\alpha]$
Principio
  Sel
    i=0: s:=0
    i>0: s:=sC(v,i-1)+v[i]
  FSel
  dev(s)
Fin
```

Transformación recursivo-iterativo: no final con inversa

- Su solución:

```
Función sCI(E v:vect;
            E i:entero)
             dev (s:entero)
--Pre:  0≤i≤n
--Post: s=∑α∈{1..i}.v[α]
Variables iAux:entero
Principio
  iAux:=i
  Mq iAux>0
    iAux:=iAux-1
  FMq
  s:=0
  Mq iAux≠i
    iAux:=iAux+1
    s:=s+v[iAux]
  FMq
  dev(s)
Fin
```

Transformación recursivo-iterativo: no final con inversa

- Ejercicios: dar una versión iterativa

```
Función Fib2(E n:entero)
                  dev (f,f1:entero)
--Pre: n≥1
--Post: f=Fibonacci(n)
              ∧ f1=Fibonacci(n-1)
Principio
   Sel
     n=1: <f,f1>:=<1,0>
     n>1: <f,f1>:=Fib2(n-1)
          <f,f1>:=<f+f1,f>
   FSel
   dev(f,f1)
Fin
```

```
Función divide2(E a,b:entero)
                dev (q,r:entero)
--Pre: a≥0 ∧ b>0
--Post: a=q*b+r ∧ 0≤r<b
Principio
   Sel
     a<b: <q,r>:=<0,a>
     a≥b: <q,r>:=divide2(a,2b)
          Sel
              r<b: <q,r>:=<2q,r>
              r≥b: <q,r>:=<2q+1,r-b>
          FSel
   FSel
   dev(q,r)
Fin
```

Transformación recursivo-iterativo: no final sin inversa

- El último caso ha sido posible por la existencia de la inversa de la función sucesora
- No siempre va a ser posible
  - no se conoce la inversa
  - un elemento tiene más de un inverso
- Una solución: usar una PILA
  - almacenar los sucesivos elementos

$$\boxed{x, s(\underline{x}), s^{2}(\underline{x}), \ldots, s^{k}(\underline{x}) \ldots.}$$

de manera que en cada momento se pueda obtener $\boxed{s^{-1}(\underline{xActual})}$

`¡Sólo necesitamos éste!`

Transformación recursivo-iterativo: no final sin inversa

- Breve introducción al “TAD pila”
- TAD Pila: estructura de datos + operadores+...

```
Tipos pila=pila de elemento

algoritmo creaVacía(S p:pila)
--Pre: True
--Post: p=[]

algoritmo apilar(ES p:pila;E e:elemento)
--Pre:  p=[$e_1$,...,$e_n$]
--Post: p=[$e_1$,...,$e_n$,e]

algoritmo desapilar(ES p:pila)
--Pre: p=[$e_1$,...,$e_n$] $\wedge$ n>0
--Post: p=[$e_1$,...,$e_{n-1}$]

función cima(E p:pila) dev (e:elemento)
--Pre: p=[$e_1$,...,$e_n$] $\wedge$ n>0
--Post: e=$e_n$

función esVacía(E p:pila)
                      dev (eV:booleano))
--Pre: p=[$e_1$,...,$e_n$]
--Post: eV= (n=0)
```

Transformación recursivo-iterativo: no final sin inversa

```
Función f(E x:tX) dev (r:tR)
{Pre(x)
 Post(x,r)}
Principio
   Sel
      $B_t$(x): r:=triv(x)
      $B_{nt}$(x): r:=c(x,f(s(x)))
   FSel
   dev(r)
Fin
```

```
Función fI(E x:tX) dev (r:tR)
--Pre(x)
--Post(x,r)
Variables xAux:tX
          p:pila de tX
Principio
   creaVacia(p)
   apilar(p,x)
   xAux:=x
   Mq $B_{nt}$(xAux)
      xAux:=s(xAux)
      apilar(p,xAux)
   FMq
   r:=triv(xAux)
   desapilar(p)

   Mq ¬esVacia(p)   --xAux≠x
      xAux :=cima(p)
      r:=c(xAux,r)
      desapilar(p)
   FMq
   dev(r)
Fin
```

Transformación recursivo-iterativo: no final sin inversa

```
Constantes n=....
Tipos vect=vector[0..n] de real

Función eval(E a:vect;E i:entero;
       E x:real) dev (v:real)
--Pre: $0 \leq i \leq n$
--Post: $v=\sum \alpha \in \{i..n\}.a[\alpha]x^{\alpha}$
Principio
   Sel
     i=n: v:=a[n]*$x^n$
     i<n: v:=a[i]*$x^i$+eval(a,i+1,x)
   FSel
   dev(v)
Fin
```

```
Función eval2(E a:vect;E i:entero;
                E x,xi:real) dev (v:real)
--Pre: $0 \leq i \leq n \wedge xi=x^i$
--Post: $v=\sum \alpha \in \{i..n\}.a[\alpha]x^{\alpha}$
Principio
   Sel
     i=n: v:=a[n]*xi
     i<n: v:=a[i]*xi+eval2(a,i+1,x,x*xi)
   FSel
   dev(v)
Fin
```

Derivación de algoritmos iterativos

- Ya vimos que se puede considerar la programación como una actividad dirigida por objetivos
- Derivación de algoritmos:

  `deducir instrucciones a`
  `partir de su especificación`

- Recordar pasos a seguir:
  - establecer muy claramente la Post
  - a partir de ella, tratar de derivar cuáles pueden ser las instrucciones que lleven a la `Post`
  - el proceso es mixto: se construyen simultáneamente el algoritmo (sus instrucciones) y su prueba (argumentación de la corrección)
  - el proceso es "retroalimentado": conforme se avanza en la derivación, se modifican inst. anteriores, prueba de la corrección

Derivación de algoritmos iterativos

- Según [Gries 81,Peña 93]:
  - derivación de instrucciones simples (secuencial y alternativas)
  - derivación de bucles
    - » precisar lo más detalladamente la Post (justo después del bucle)
    - » “derivar” el invariante a partir de la Post
    - » “derivar”, a partir del invariante, el resto del bucle
      - condición de terminación del bucle
      - inicialización de las variables que intervienen en el bucle
      - derivación del cuerpo del bucle...
- <u>Una vez más</u>: NO es un proceso automático
- Tenemos la especificación

```
{Q} .... {R}
```

- e <u>intuímos</u> un bucle

Derivación de algoritmos iterativos

- Derivación del bucles:

1) derivación del invariante `I`

2) conocidos `I` y `R`, buscar $\neg$`B` tal que

$$\mathtt{I} \land \neg \mathtt{B} \rightarrow \mathtt{R}$$

3) A partir de `I`, buscar `inicialización` tal que

`{Q} inicialización {I}`

4) Tenemos, de momento:

```
--Q
  inicialización


  Mq B


     ????
  FMq
--R
```

```
--I:
?
--
t():?
```

Derivación de algoritmos iterativos

5) Buscar `t` y añadir al bucle `avanzar`, que aproxima el estado a la salida del mismo

6) Puede ser necesario establecer algunas instrucciones para mantener cierto el invariante

```
--Q
   inicialización



   Mq B


      avanzar
      restablecer
   FMq
--R
```

```
--I:  ?
--t():?
```

- Pero notar que todo pasa por derivar el invariante previamente
  – debilitamiento de la Post es un buen principio

Derivación de algoritmos iterativos

- Ejercicio: Siguiendo, dentro de lo posible, el esquema propuesto, derivar los siguientes algoritmos

```
Constantes n=.....
Tipos      vect=vector[1..n] de real

Función sumaC(E v:vect) dev (sC:real)
--Pre: True
--Post: sC=∑α∈{1..n}.v[α]
```

```
Función raiz1(E n:entero)
                      dev (r:entero)
--Pre: 0≤n
--Post: r²≤n<(r+1)²
```

```
Algoritmo ordena(ES v:vect)
--Pre: v=V
--Post: ∀α∈{1..n-1}.v[α]≤v[α+1]} ∧
       PERMUTACION(v,V,1,n)
```

TEMA 5: El esquema de divide y vencerás

- Idea muy simple:

cuando un problema es demasiado grande,
es mejor resolverlo "por partes"
más sencillas

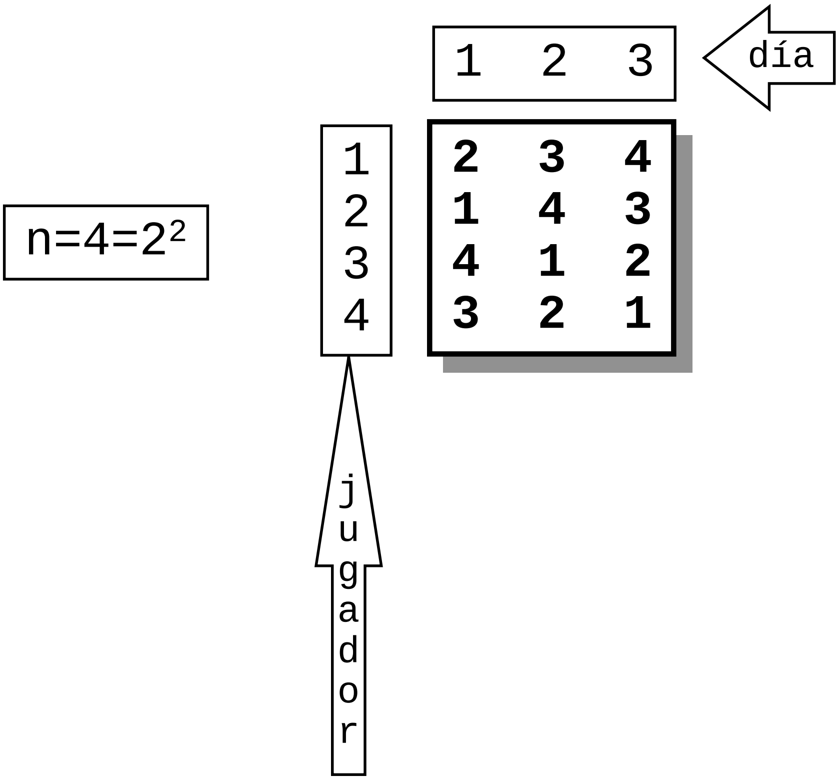
- Asumamos un problema de tamaño **n**:
  - resolver `k (1<k<=n)` problemas de tamaño menor
  - "componer" las soluciones parciales

- <u>Ejemplo</u>:

organizar los enfrentamientos en forma de liga para n participantes (asumir $n=2^k$). Los enfrentamientos son en días consecutivos, y cada día cada jugador sólo juega un partido

Esquemas algorítmicos: “Divide y vencerás”

- Dar la solución como una matriz



$n=4=2^2$

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array}$$

- Una solución “a lo bestia”
  (“fuerza bruta”)
  - para cada jugador `i`, obtener **`P({1..n}-{i})`**
  - rellenar cada fila `i` de la matriz con un elemento de **`P({1..n}-{i})`**, de manera que se cumplan las restricciones impuestas

Esquemas algorítmicos: "Divide y vencerás"

- Problema serio: la complejidad
  - obtener **`P({1..n}-{i})`** es **`(n-1)!`**
  - hay que hacerlo para **`i∈{1..n}`**
  - Por lo tanto

$$\boxed{O(n*(n-1)!)=O(n!)}$$

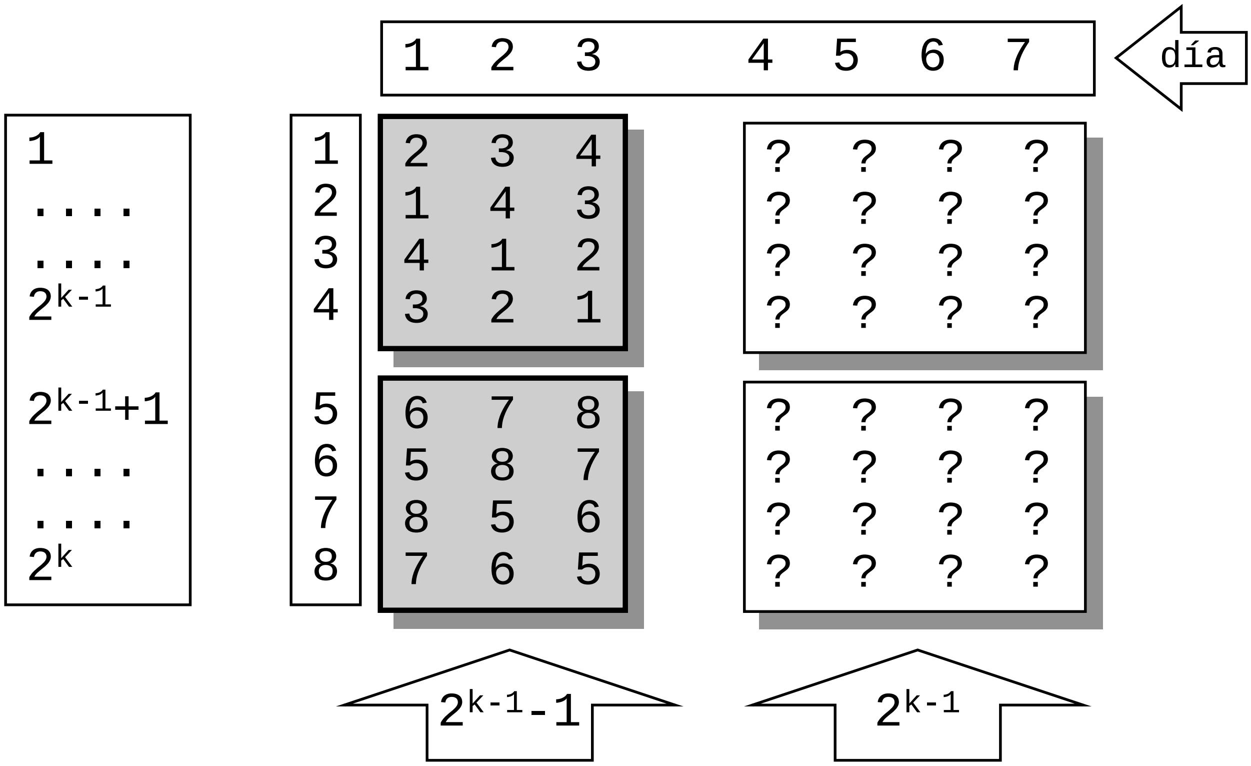
- Necesitamos algo mejor:
  - vamos a dividir el problema en problemas más sencillos
  - vamos a resolver cada uno de ellos
  - vamos a componer las soluciones

Esquemas algorítmicos: "Divide y vencerás"

- Si hay **2^k** jugadores, necesitamos obtener una matriz de **$2^k x 2^k-1$**
- En un primer paso, elaboremos, independientemente, los calendarios de los jugadores:

  $1, 2^{k-1}$ **y** $2^{k-1}+1, 2^k$

- Ejemplo para k=3



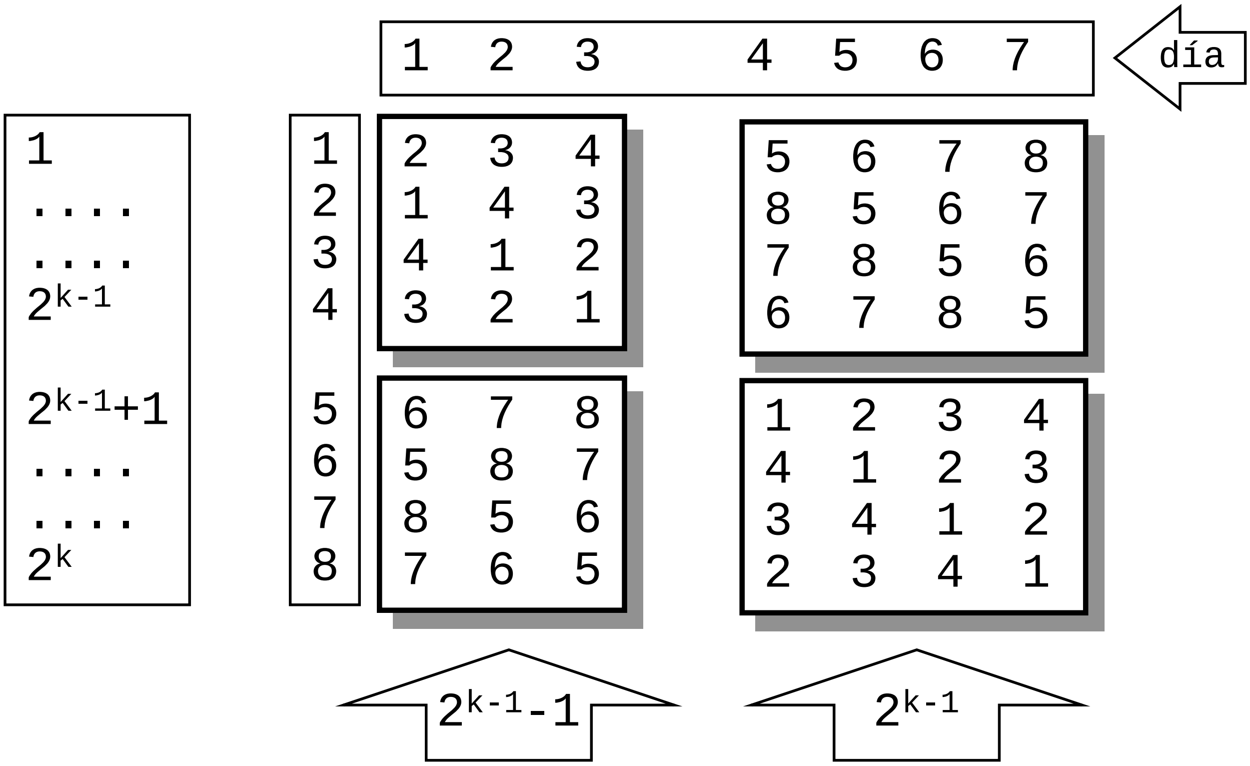
1
....
....
2^{k-1}

$2^{k-1}+1$
....
....
2^k

	1	2	3	4	5	6	7
1	2	3	4	?	?	?	?
2	1	4	3	?	?	?	?
3	4	1	2	?	?	?	?
4	3	2	1	?	?	?	?
5	6	7	8	?	?	?	?
6	5	8	7	?	?	?	?
7	8	5	6	?	?	?	?
8	7	6	5	?	?	?	?

Esquemas algorítmicos: "Divide y vencerás"

- Faltan todavía elementos por completar **componer las soluciones parciales**
- Lo que queda es muy fácil



día

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

1
....
....
2^{k-1}

$2^{k-1}+1$
....
....
2^k

$2^{k-1}-1$

2^{k-1}

Esquemas algorítmicos: "Divide y vencerás"

- Esquemáticamente, hemos hecho

```
problema(2^k)

   dos problema(2^(k-1))
   problemas de composición
```

- Siendo un caso trivial cuando n=2
- El sistema de ecuaciones recurrentes

$$
t_n = \begin{cases} k_1 & \text{si } n=2 \\ 2t_{n/2}+k_2n^2 & \text{si } n>2 \end{cases}
$$

- Recurrencia de partición, que da complejidad **n^2**
- ¡¡ Hemos mejorado mucho !!

Esquemas algorítmicos: "Divide y vencerás"

- En un caso general, para un problema de tamaño **n** el coste será:

`Asumimos descomposición en dos subproblemas`

```
T(n)=
    g(n)                  si n pequeño

    2T(n/2) + f(n)         resto de n
```

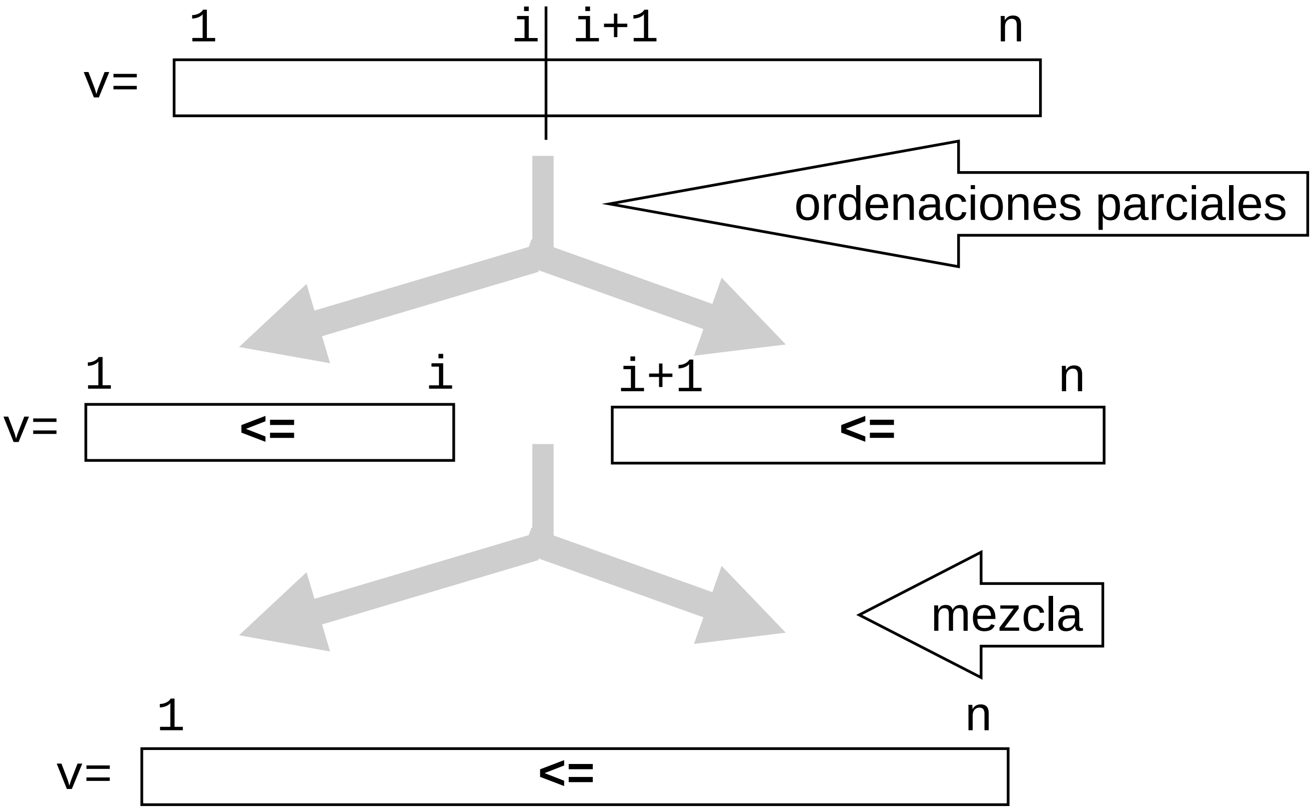
- donde:
  - g(n): será normalmente constante
  - f(n): coste de composición
- Por lo tanto:
  - lo que se gane en eficiencia depende en gran medida de la función f
  - será interesante que los sub-problemas sean de tamaño parecido

Esquemas algorítmicos: “Divide y vencerás”

- Ejemplo 1: ordenación por mezcla

```
Constantes n=.....
Tipos      vect=vector[1..n] de real

Algoritmo ordena(ES v:vect)
--Qo: v=V
--Ro: permutacion(v,V,1,n)∧
           ∀α∈{1..n-1}.v[α]<=v[α+1]
```

- ¿Cómo podemos dividirlo? **`i=n DIV 2`**



1 i i+1 n

v=

ordenaciones parciales

1 i i+1 n

v= **<=** **<=**

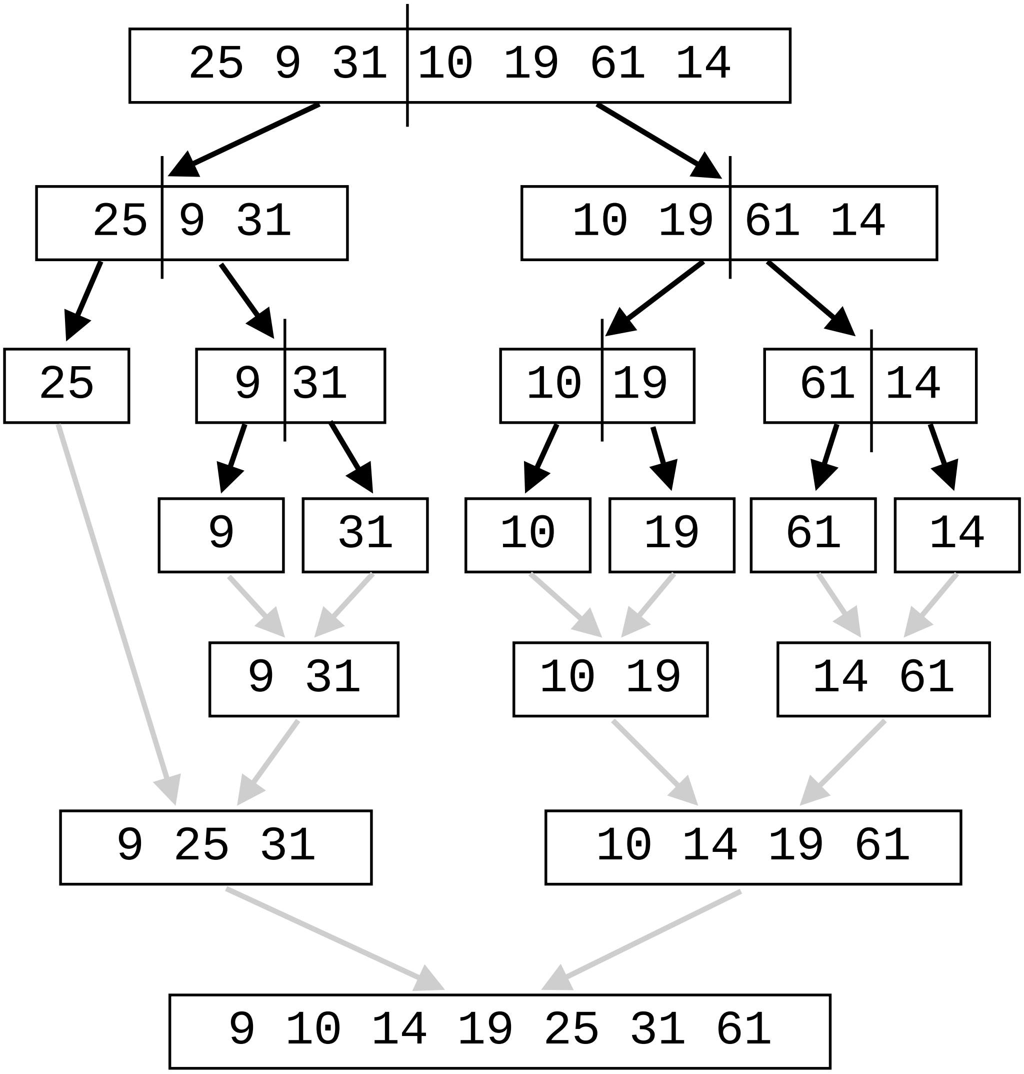
mezcla

1 n

v= **<=**

Esquemas algorítmicos:
“Divide y vencerás”

- Aplicando a un ejemplo:



25 9 31 | 10 19 61 14

25 | 9 31

10 19 | 61 14

25

9 | 31

10 | 19

61 | 14

9

31

10

19

61

14

9 31

10 19

14 61

9 25 31

10 14 19 61

9 10 14 19 25 31 61

Esquemas algorítmicos: “Divide y vencerás”

```
Algoritmo ordenMezcla(ES v:vect)
--QoM: v=V
--RoM : permutacion(v,V,1,n)∧
      ∀α∈{1..n-1}.v[α]<=v[α+1]
Principio
   mergeSort(v,1,n)
Fin
```

```
Algoritmo mergeSort(ES v:vect
                    E pI,pF:entero)
--QmS : 1<=pI<=pF<=n ∧ v=V
--RmS : PERMUTACION(v,V,pI,pF)∧
               ∀α∈{pI..pF-1}.v[α]<=v[α+1]
Variables medio:entero
Principio
   Si pI<pF ent
       medio:= (pF+pI)DIV 2
       mergeSort(v,pI,medio)
       mergeSort(v,medio+1,pF)
       merge(v,pI,medio,pF)
   FSi
Fin
```

Esquemas algorítmicos: "Divide y vencerás"

- Algoritmo de mezcla

```
Algoritmo merge(ES v:vect;E I,M,D:entero)
--Qm: 1<=I<=M<D<=n ∧ v=V ∧
       ORDENADO(V,I,M) ∧ ORDENADO(V,M+1,D)

--Rm: MEZCLA(V,I,M,V,M+1,D,v,I,D) ∧
                ORDENADO(v,I,D)
```

- Donde usamos los predicados

```
ORDENADO(v,pI,pD)=
   ∀α∈{pI..pD-1}.v[α]<=v[α+1] ∧ pI<=pD
```

```
MEZCLA(v1,i1,d1,v2,i2,d2,v3,i3,d3)=
   d3-i3+1=d2-i2+1+d1-i1+1 ∧
   ∀α∈{i3..d3}.
          (Nβ∈{i3..e3}.v3[β]=v3[α])  =
              Nβ1∈{i1..d1}.v1[β1]=v3[α]+
              Nβ2∈{i2..d2}.v2[β2]=v3[α]
```

Esquemas algorítmicos: "Divide y vencerás"

```
Algoritmo merge(ES v:vect;E I,M,D:entero)
Variables vAux:vect
          iI,iD,iNuevo,otroI:entero
Principio
   <iI,iD,iNuevo> := <I,M+1,I>
   Mq (iI<=M) ∧ (iD<=D)
      Si v[iI]<v[iD]
         entonces vAux[iNuevo]:=v[iI]
                  iI:=iI+1
         si no    vAux[iNuevo]:=v[iD]
                  iD:=iD+1
      FSi
      iNuevo:=iNuevo+1
   FMq

   Para otroI:=iI hasta M
      vAux[iNuevo]:=v[otroI]
      iNuevo:=iNuevo+1
   FPara
   Para otroI:=iD hasta D
      vAux[iNuevo]:=v[otroI]
      iNuevo:=iNuevo+1
   FPara
   v:=vAux    --asignación entre vectores
Fin
```

Esquemas algorítmicos: "Divide y vencerás"

- Complejidad de mergeSort:

$$t_n = k_1 \qquad \text{si } n=1$$

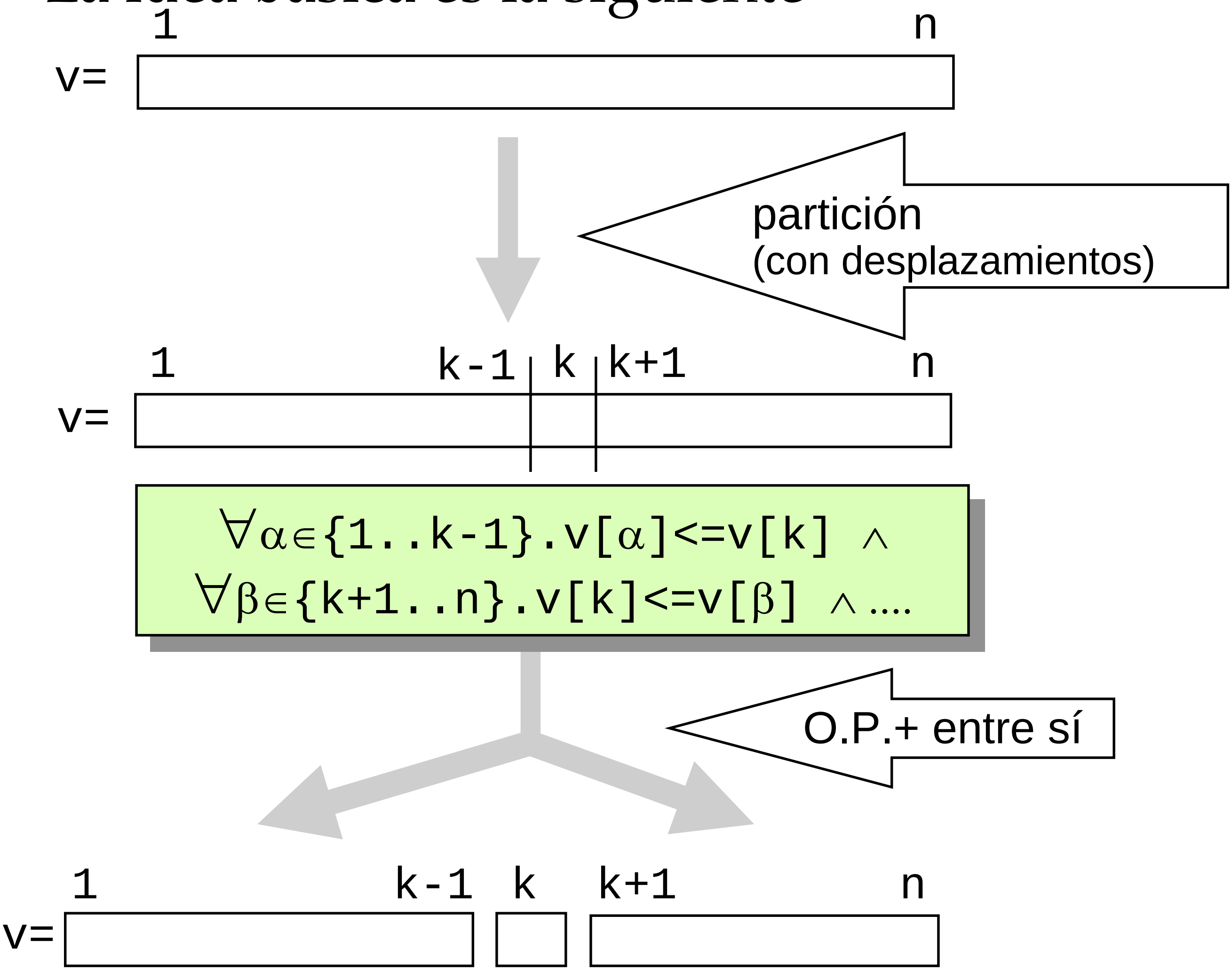
$$t_n = 2t_{n/2} + k_2 n \qquad \text{si } n>1$$

de manera que $t_n \in O(n \lg_2(n))$

- El método tiene un incoveniente importante: el uso de **vAux** multiplica por dos la memoria necesaria
- Esto es debido a que cada invocación recursiva ordena parte del vector, pero el orden de ambas partes no guarda relación entre sí

Esquemas algorítmicos: “Divide y vencerás”

- Un nuevo algoritmo: <u>Quicksort</u>
- La idea básica es la siguiente



$\forall \alpha \in$`{1..k-1}.v[`α`]<=v[k]` $\wedge$

$\forall \beta \in$`{k+1..n}.v[k]<=v[`β`]` $\wedge$ ....

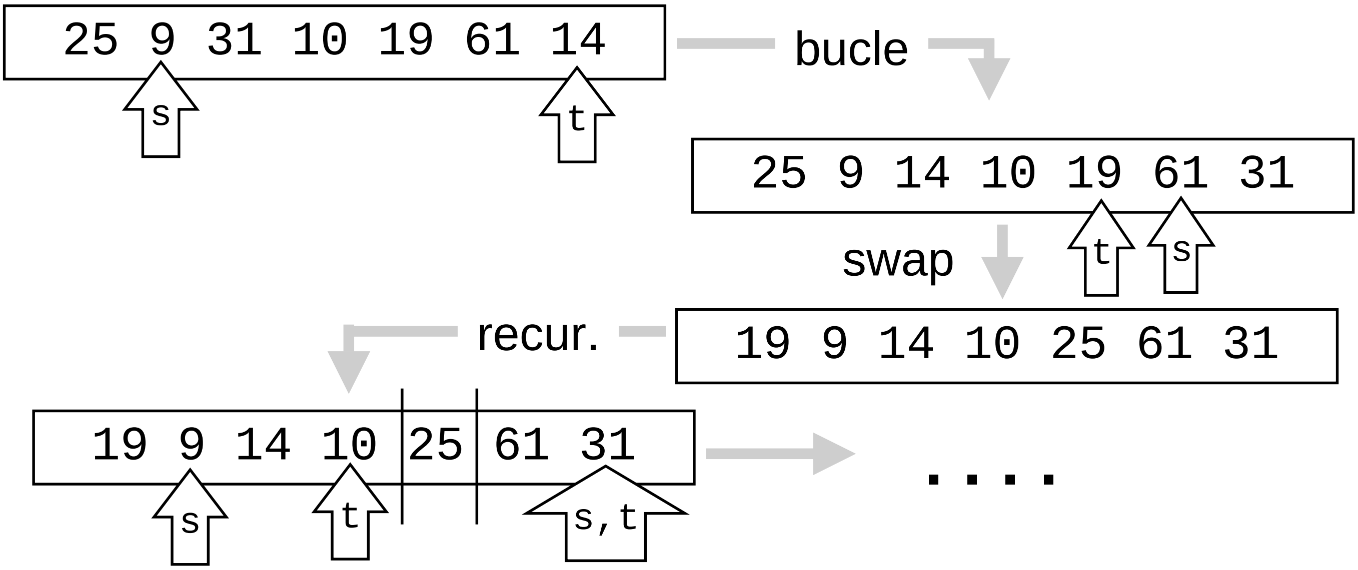
Esquemas algorítmicos: "Divide y vencerás"

```
Algoritmo ordenRápido(ES v:vect)
--QoR: v=V
--RoR: PERMUTACION(v,V,1,n)∧
       ∀α∈{1..n-1}.v[α]<=v[α+1]
Principio
   quickSort(v,1,n)
Fin
```

```
Algoritmo quickSort(ES a:vect;E i,j:entero)
--QqS: 1<=i ∧ j<=n ∧ a=A
--RqS: PERMUTACION(a,A,i,j)∧
             ∀α∈{i..j-1}.a[α]<=a[α+1]}
```

Esquemas algorítmicos:
“Divide y vencerás”

```
Algoritmo quickSort(ES a:vect;E i,j:entero)
Variables s,t:entero; x:real
Principio
  Si i<=j ent
    x:=a[i];<s,t>:=<i+1,j>
    Mq s<>t+1
       Sel
          a[s]<=x: s:=s+1
          a[s]>x ∧ a[t]>=x: t:=t-1
          a[s]>x ∧ a[t]<x : swap(a,s,t)
                             <s,t>:=<s+1,t-1>
       FSel
    FMq
    swap(a,i,t);quickSort(a,i,t-1)
                quickSort(a,t+1,j)
  FSi
Fin
```



25 9 31 10 19 61 14

bucle

25 9 14 10 19 61 31

swap

19 9 14 10 25 61 31

recur.

19 9 14 10 | 25 | 61 31

....

Esquemas algorítmicos: "Divide y vencerás"

- Complejidad:
  - caso más desfavorable: n^2
  - en media: $nlg_2(n)$
- Notar que NO hay etapa de composición