2

DISEÑO DE ALGORITMOS RECURSIVOS

- INTRODUCCIÓN
- PRINCIPIO DE INDUCCIÓN
- DEFINICIÓN RECURSIVA DE UNA FUNCIÓN
- ESPECIFICACIÓN FORMAL (CUANTIFICADORES)
- OTROS PRINCIPIOS DE INDUCCIÓN (NOETHERIANA Y ESTRUCTURAL)
- INMERSIÓN
- VERIFICACIÓN FORMAL

INTRODUCCIÓN

La **recursividad**, junto a la **iteración**, son los dos mecanismos que suministran los lenguajes de programación para describir cálculos que, con pequeñas variaciones, han de repetirse un cierto número de veces.

¿Qué es la recursividad?

Es una característica de la mayoría de los lenguajes de programación, por la cual se permite que un procedimiento, o función, haga referencia a sí mismo dentro de su definición.

INTRODUCCIÓN

- El concepto de recursividad no es nuevo. En el campo de las matemáticas podemos encontrar muchos ejemplos relacionados con él.
- En primer lugar muchas **definiciones matemáticas se realizan en términos de sí mismas**. Considérese el clásico ejemplo de la función factorial:

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

donde la definición se realiza por un lado para un **caso** que denominamos **base (n=0)** y por otro, para un **caso general (n>0)** cuya formulación es claramente recursiva.

INTRODUCCIÓN

- Consideremos por otro lado la conocida **demostración por inducción** que generalmente sólo necesita realizar una simple demostración para un **caso base** y una segunda para un **caso general** de tamaño n (con la ventaja de suponerlo demostrado para cualquier caso menor que n) de forma que queda demostrado para todos los casos.

- De forma similar, **cuando diseñamos una función recursiva podemos suponer que está resuelta para un tamaño menor**.

- De esta forma, la recursividad constituye una de las herramientas mas potentes en programación.

INTRODUCCIÓN

- La **recursividad tiene la misma potencia expresiva que la iteración**, en el sentido de que permite describir los mismos cómputos y las mismas funciones que pueden describirse mediante algoritmos iterativos.

- El uso de la recursividad genera **algoritmos más compactos** que sus correspondientes versiones iterativas.

INTRODUCCIÓN

- En este tema vamos a presentar la **recursividad** como una **herramienta fundamental** para el **diseño de algoritmos** y para el **razonamiento formal sobre su corrección**.

- Veremos que el **esfuerzo de razonamiento es menor** pensando en recursivo que en iterativo.

- Veremos también como estos algoritmos recursivos pueden ser transformados a su versión iterativa sin merma de corrección ni de eficiencia.

INTRODUCCIÓN

Resolver un problema ***P*** sobre unos datos ***D***

¿ problema ***P***  datos ***D*** ?

INTRODUCCIÓN

Calcular la potencia n-ésima de a, siendo $a \geq 0$ y $n \geq 0$

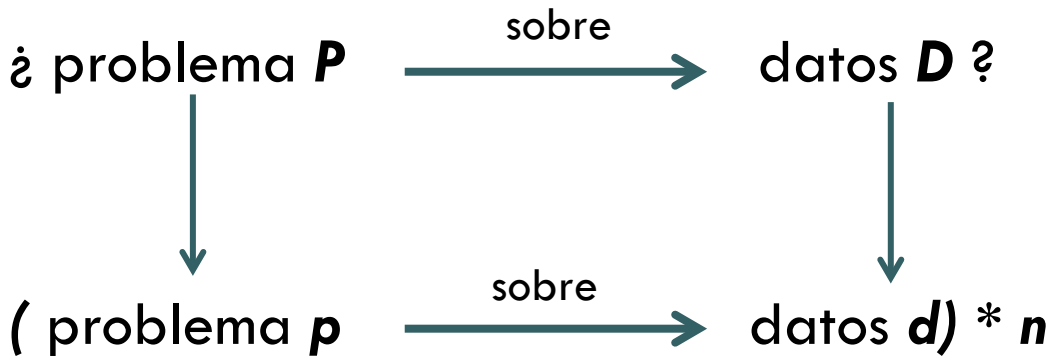
PRECONDICIÓN: caracteriza el conjunto de estados iniciales para los que el algoritmo funciona correctamente

$$\mathbf{Q \equiv \{\ a \geq 0 \wedge n \geq 0\ \}}$$

Función **POTENCIA** (a:entero, n:entero) retorna (p : entero)

INTRODUCCIÓN

Planteamiento **ITERATIVO** al diseñar la solución



INTRODUCCIÓN

Calcular la potencia n-ésima de a, siendo a≥0 y n≥0

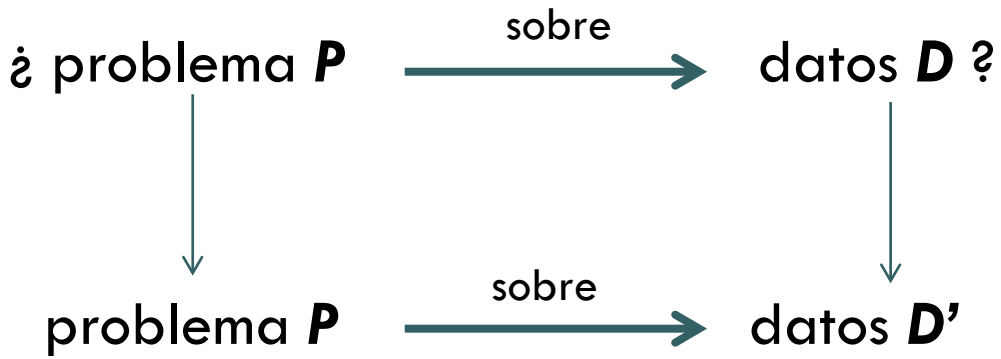
Q ≡ { a ≥ 0 ∧ n ≥ 0 }

Función **POTENCIA** (a:entero, n:entero) retorna (p : entero)

```
    var x:entero fvar
    x = 1;
    para i = 1 hasta n hacer        /* se repite un número n de veces */
          x = a * x;                /* problema p = multiplicar dos números */
    fpara
    retorna x;
ffunción
```

INTRODUCCIÓN

Planteamiento **RECURSIVO** al diseñar la solución



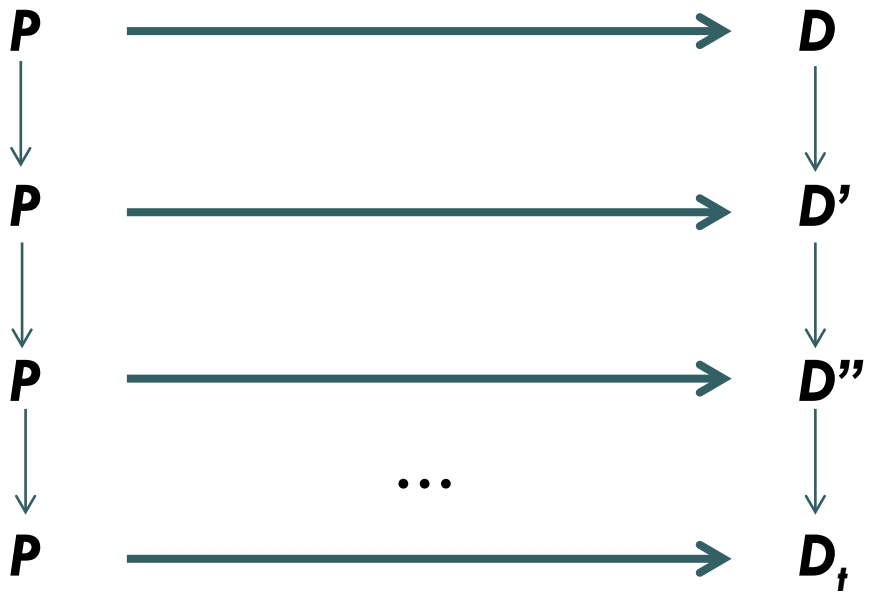
INTRODUCCIÓN

- El mismo razonamiento que conduce a resolver el problema **P** sobre los datos **D** a partir del problema **P** sobre los datos **D´** se aplica a su vez para resolver el problema **P** sobre los datos **D'** a partir del problema **P** sobre los datos **D''**, los cuales son aún más pequeños.

- Se forma así, una sucesión **D > D' > D'' > ... > Dt** de datos cada vez más pequeños.

INTRODUCCIÓN

Planteamiento **RECURSIVO** al diseñar la solución



INTRODUCCIÓN

Q ≡ { a ≥ 0 ∧ n ≥ 0 }

Función **POTENCIA** (a:entero, n:entero) retorna (p : entero)

¿ **POTENCIA(a,n)** ?, dicho de otro modo, ¿ a^n ?

Vamos a suponer conocido **POTENCIA(a,n-1)**, esto es, a^{n-1}

INTRODUCCIÓN

¿Cómo podemos calcular **POTENCIA(a,n)** a partir de **POTENCIA(a,n-1)**?

o, dicho de otra forma,

¿Cómo podemos calcular a^n partir de a^{n-1}?

$$a^n = a^{n-1} * a$$

↓

POTENCIA(a,n) = POTENCIA(a,n-1) * a

INTRODUCCIÓN

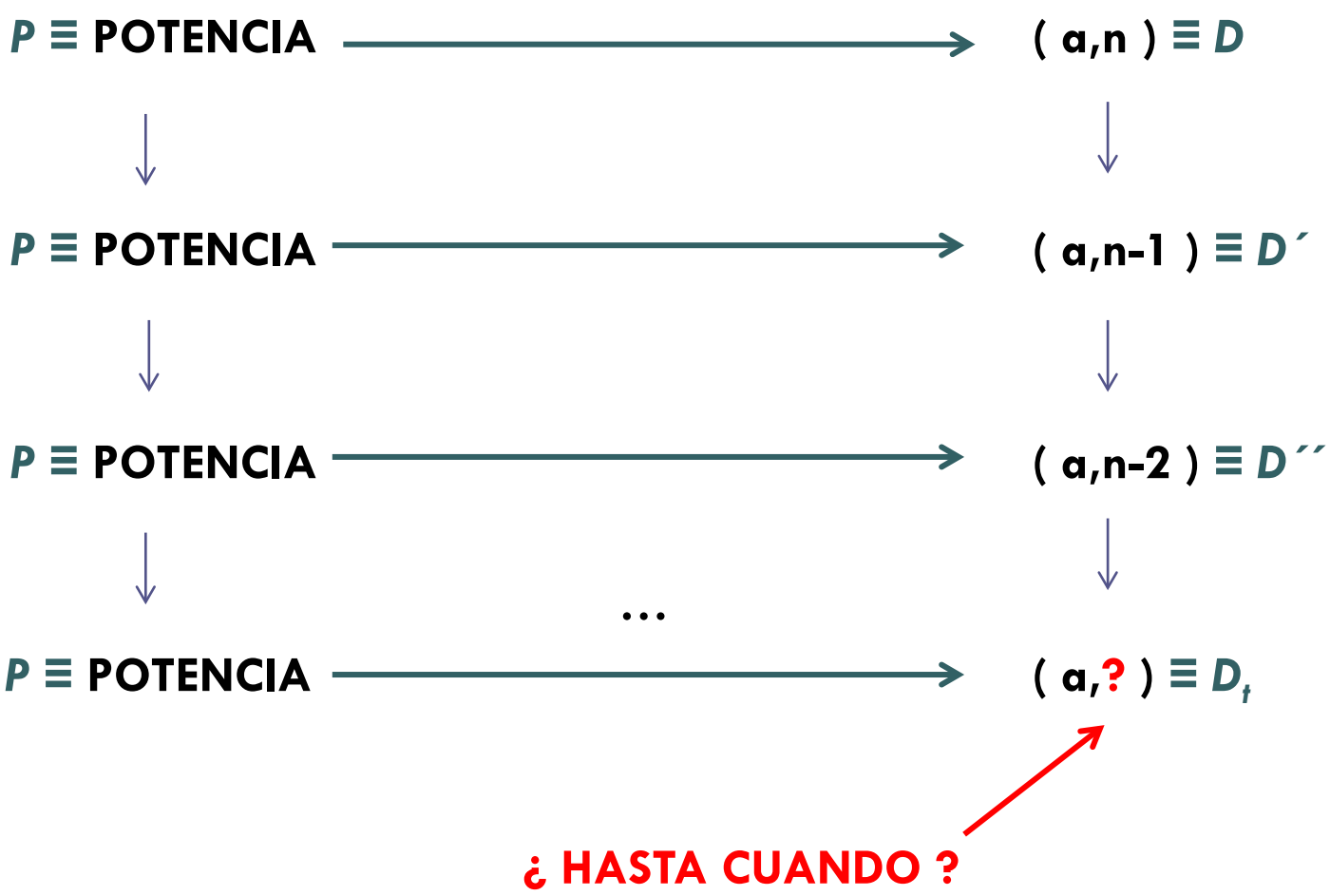
¿ POTENCIA(a,n-1) ?

Suponemos conocido **POTENCIA(a,n-2)**

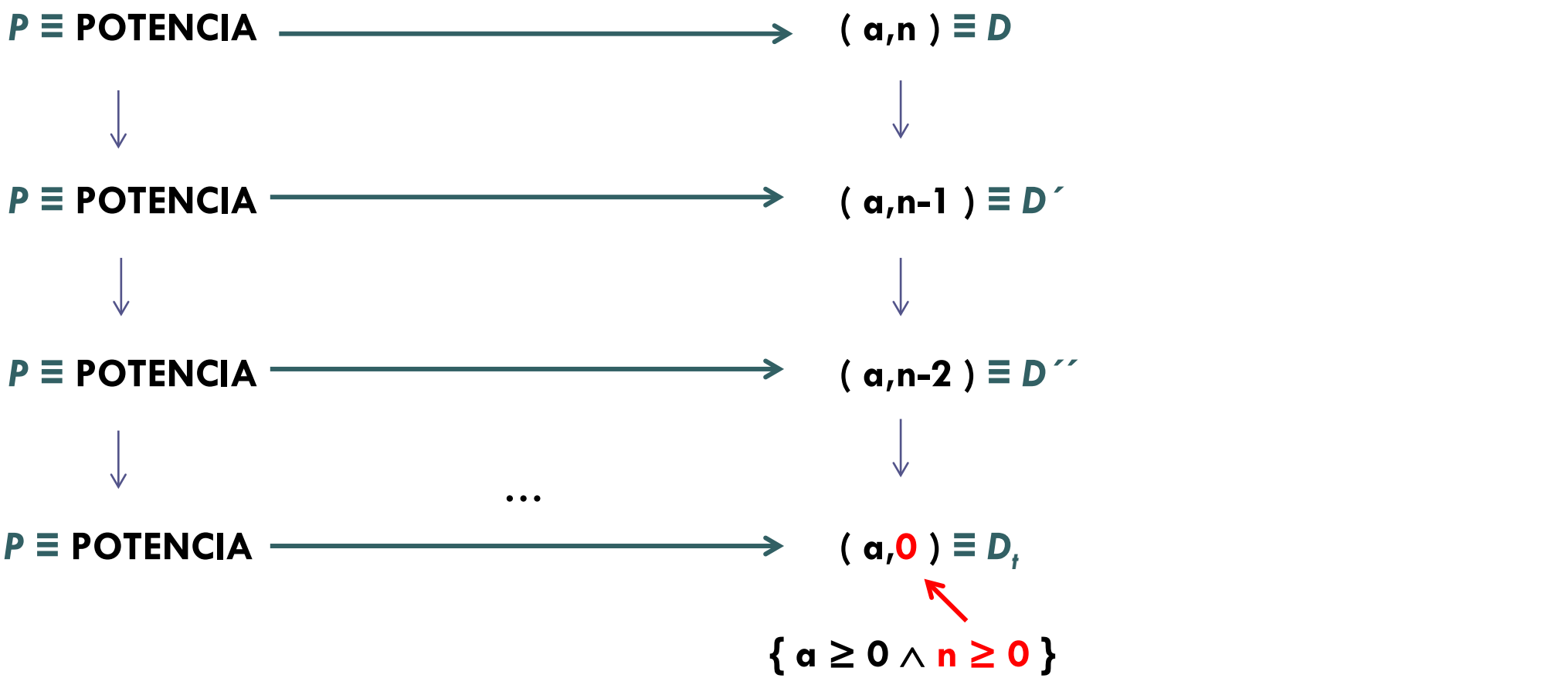
Por tanto,

POTENCIA(a,n-1) = POTENCIA(a,n-2) * a

INTRODUCCIÓN



INTRODUCCIÓN



Función **POTENCIA** (a:entero, n:entero) retorna (p : entero)

INTRODUCCIÓN

Q ≡ {a ≥ 0 ∧ n ≥ 0}

Funcion **POTENCIA** (a:entero, n:entero) retorna (p:entero)

si n = 0 entonces retorna 1

sino retorna **POTENCIA**(a, n-1) * a

fsi

ffunción

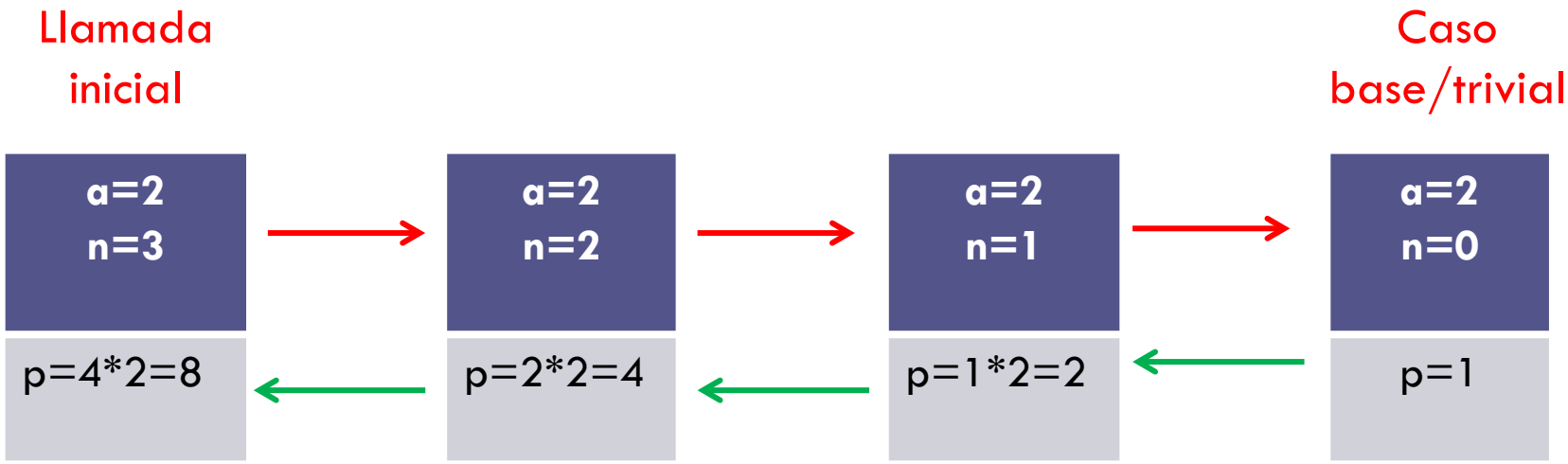
CADENA DE INVOCACIONES.-

(a,n) → (a,n-1) → (a,n-2) → …. →(a,2) → (a,1) → (a,0)

¡¡ Sucesión estrictamente decreciente y finita !!

INTRODUCCIÓN

Funcionamiento de la función POTENCIA.-



PRINCIPIO DE INDUCCIÓN

- La garantía de que el proceso visto se hace correctamente y que el algoritmo anterior produce la potencia de a para cualquier valor de n, viene de la mano del **principio de inducción**, que es la herramienta teórica en la que nos apoyaremos para dar sustento al planteamiento recursivo.
- El principio de inducción se establece sobre el conjunto de los números naturales (**IN**) en tres fases o etapas, a las que denominaremos **Base**, **Recurrencia** y **Conclusión**.
- Nos permite demostrar que una propiedad P es cierta para cualquier natural. Por ejemplo, demostrar que $\forall n \in IN$, a^n se puede calcular como

$$a^{n-1} * a$$

- Existen dos tipos fundamentales de inducción: **débil** (al cual corresponde el ejemplo anterior) y **fuerte**.

INDUCCIÓN DÉBIL

Objetivo.- Demostrar que la propiedad **P** se cumple para todos los naturales mayores o iguales que **b**

Base de la recurrencia

Demostrar que **P** es cierta para un número natural **b**

Recurrencia o paso inductivo: demostrar que la propiedad es hereditaria

Hipótesis: Se supone que **P** es cierta un natural cualquiera **n,** siendo **n ≥ b**

Tesis: Se demuestra entonces que **P** se cumple también para el natural **n+1**

Conclusión

El resultado satisfactorio en cada una de las dos etapas anteriores, permite entonces establecer que **P** es cierta para todos los naturales mayores o iguales que **b.**

INDUCCIÓN DÉBIL: ejemplo

Demostrar la siguiente propiedad **P**: "**10^n-1 es divisible por 9, $\forall$ n $\in$ IN**"

Base

P se cumple para **n=0**. Efectivamente, $10^0-1=0=0*9$

Recurrencia o paso inductivo

Hipótesis: Se supone que **10^n-1** es divisible por **9**.

Tesis: Se demuestra entonces que **10^{n+1}-1** es también divisible por **9**.

INDUCCIÓN DÉBIL: ejemplo (continuación)

Demostración:

La hipótesis garantiza que existe un valor **a** $\in$ **IN** tal que $\mathbf{10^n-1 = a*9}$.

Por lo tanto,

$10^{n+1}-1 = 10*10^n-1 = 10*(a*9+1)-1 = 90*a+10-1 =$

$10^n = a*9+1$ **(H.I.)**

$= 90*a+9 = (10*a+1) * 9$ que resulta ser múltiplo de 9.

Conclusión

El principio de inducción nos permite concluir que la propiedad **P** es cierta $\forall$ **n** $\in$ **IN**.

INDUCCIÓN FUERTE

Objetivo.- Demostrar que la propiedad **P** se cumple para todos los naturales mayores o iguales que **b**

Base de la recurrencia

Demostrar que **P** es cierta para un número natural **b**

Recurrencia o paso inductivo: demostrar la herencia

Hipótesis: Se supone que **P** es cierta para todo natural del intervalo **[b,n]** con **n≥b**

Tesis: Se demuestra entonces que **P** se cumple también para el natural **n+1**

Conclusión

El resultado satisfactorio en cada una de las dos etapas anteriores, permite entonces establecer que **P** es cierta para todos los naturales mayores o iguales que **b**

INDUCCIÓN FUERTE: ejemplo

Demostrar la siguiente propiedad **P**: "**Todo numero natural n (n ≥ 2) puede descomponerse en un producto de números primos**"

Base

P se cumple para **n=2**

Basta considerar que **2** es el producto **2*1**, y que **el 2 y el 1 son primos**

INDUCCIÓN FUERTE: ejemplo (continuación)

Recurrencia o paso inductivo

Hipótesis: La propiedad **P** se cumple para todos los naturales del intervalo **[2,n]** con **$n \geq 2$**

Tesis: Entonces **P** se cumple también para el natural **n+1**

Para demostrarlo, basta tener en cuenta que **n+1** debe cumplir una de las dos condiciones siguientes:

INDUCCIÓN FUERTE: ejemplo (continuación)

- **n+1 es primo:** Entonces la tesis está demostrada.

- **n+1 no es primo:** decir que **n+1** no es primo permite afirmar que existen dos enteros **a** y **b** tales que

$$n+1 = a * b \quad y \quad 1 < a < n+1 \quad y \quad 1 < b < n+1$$

Por hipótesis de inducción **a** y **b** satisfacen la propiedad. Se deduce entonces la propiedad para **n+1**

Conclusión

La propiedad **P** se cumple para todos los **naturales mayores o iguales que 2**

DEFINICIÓN RECURSIVA DE UNA FUNCIÓN

Especificar formalmente la función.-

- Nombre de la función
- Nombre y tipo del conjunto inicial (dominio)
- Nombre y tipo del conjunto final (imagen)
- Precondición y postcondición.

Describir el principio de inducción utilizado.-

Estudiar cómo se pueden descomponer recursivamente los datos del problema, de forma que podamos calcular la solución pedida a partir de una o más soluciones del propio problema para datos más pequeños.

DEFINICIÓN RECURSIVA DE UNA FUNCIÓN

Realizar el Análisis por casos (valores de la función).-

Analizar los casos que se puedan presentar a la función, identificando bajo qué condiciones el problema ha de considerarse **TRIVIAL** y cuál ha de ser la solución a aplicar en ese caso, y bajo qué condiciones el problema ha de considerarse **NO TRIVIAL** y cómo ha de resolverse entonces.

Escribir el algoritmo en pseudocódigo

Verificar formalmente la corrección del algoritmo

DEFINICIÓN RECURSIVA DE UNA FUNCIÓN

El siguiente esquema describe el **modelo general** en el que deben encuadrarse las funciones recursivas:

$\{ Q(\bar{x}) \}$
función $f(\bar{x}: T_1)$ retorna $(\bar{y}: T_2)$
  caso
    $Bt(\bar{x}) \rightarrow triv(\bar{x})$
    $Bnt(\bar{x}) \rightarrow c(f(s(\bar{x})), \bar{x})$
  fcaso
ffunción
$\{ R(\bar{x},\bar{y}) \}$

donde los parámetros formales $\bar{x}$ e $\bar{y}$ han de entenderse como tuplas $\{ x_1, x_2, ..., x_n \}$ e $\{y_1, y_2, ..., y_m \}$, respectivamente.

DEFINICIÓN RECURSIVA DE UNA FUNCIÓN

- Las expresiones booleanas Bt y Bnt distinguen, respectivamente si el problema $\bar{x}$ es trivial o no. Obviamente ha de cumplirse

$$Bt(\bar{x}) \wedge Bnt(\bar{x}) \equiv falso,$$

para todo $\bar{x}$ que satisfaga la precondición $\{Q(\bar{x})\}$

- triv: $T_1 \rightarrow T_2$ es una función que calcula la solución de f cuando $\bar{x}$ es trivial

- s: $T_1 \rightarrow T_1$ es la FUNCIÓN SUCESOR y realiza la descomposición recursiva de los datos $\bar{x}$, calculando el subproblema $\bar{x}'$ de $\bar{x}$, al cual se le aplica la invocación recursiva de f

- Denotaremos $s(\bar{x})$ como $\bar{x}'$ y $f(s(\bar{x}))$ como $\bar{y}'$

DEFINICIÓN RECURSIVA DE UNA FUNCIÓN

- c: T_2 x $T_1 \rightarrow T_2$ es la FUNCIÓN DE COMBINACIÓN y combina el resultado devuelto por f para $\bar{x}'$ con (todos o parte de) los propios parámetros de entrada $\bar{x}$

- R es la postcondición de f que establece la relación que ha de cumplirse entre los parámetros de entrada $\bar{x}$ y los resultados $\bar{y}$

- Q es la precondición de f y establece los estados válidos para invocar a la función f

ESPECIFICACIÓN FORMAL DE ALGORITMOS

Vamos a escribir especificaciones claras y precisas. Para ello es necesario un lenguaje formal, llamado así porque tanto su sintaxis como su semántica están perfectamente definidas. Vamos a utilizar una técnica de especificación formal de algoritmos basada en lógica de predicados denominada **especificación pre/post**.

Sea **A** un algoritmo del que conocemos sus parámetros de entrada y sus resultados. Una especificación pre/post utiliza un **predicado Q**, denominado **precondición**, que incluye como variables libres los parámetros de entrada de A y describe los estados en los que el algoritmo puede ejecutarse.

ESPECIFICACIÓN FORMAL DE ALGORITMOS

Además, se utiliza otro **predicado R**, denominado **postcondición**, que incluye como variables libres los parámetros de entrada y los resultados de A. Este predicado R describe los estados finales que el algoritmo alcanza tras su ejecución dando la relación entre la entrada y la salida. La notación

$$\{ Q \} A \{ R \}$$

representa la especificación formal de A y ha de leerse como:

"si A comienza su ejecución en un estado que satisface Q, entonces la ejecución de A termina y lo hace en un estado que cumple R".

ESPECIFICACIÓN FORMAL DE ALGORITMOS

Para escribir la precondición y la postcondición utilizaremos lógica de primer orden. Así, un predicado se podrá construir a partir de fórmulas atómicas (las constantes cierto y falso, variables booleanas, expresiones aritméticas relacionales como las construidas con las operaciones $=$, $\neq$, $\leq$, etc.), las conectivas lógicas de negación ($\neg P$), conjunción ($P \wedge Q$), disyunción ($P \vee Q$) e implicación ($P \rightarrow Q$) y los cuantificadores, universal ($\forall$) y existencial ($\exists$).

Utilizaremos cuantificadores en los que se indica el rango de la variable cuantificada, r(i), y la expresión a evaluar, E(i), según el siguiente formato:

(Cuantificador i)(E(i) : r(i))

ESPECIFICACIÓN FORMAL DE ALGORITMOS

Ejemplos:

$$(\forall i)\ (A[\ i\ ] = 3 : 1 \leq i \leq n)$$

$$(\exists i)\ (A[\ i\ ] = 3 : 1 \leq i \leq n)$$

Si el rango de un cuantificador universal es vacío, esto es, no hay ningún valor de i que satisfaga el predicado, entonces el predicado es igual a cierto y en el caso de una cuantificación existencial, el valor es falso. Como se puede ver, en ambos casos, corresponde al elemento neutro de la operación representada.

ESPECIFICACIÓN FORMAL DE ALGORITMOS

Para escribir expresiones también utilizaremos otros cuantificadores, siempre indicando el rango de la variable ligada:

Cuantificador	Ejemplo
Sumatorio (Σ)	$(\Sigma\ i)\ (A[\ i\ ] : 1 \leq i \leq n)$
Producto (Π)	$(\Pi\ i)\ (A[\ i\ ] : 1 \leq i \leq n)$
Máximo de una serie de valores (MAX)	$(MAX\ i)\ (A[\ i\ ] : 1 \leq i \leq n)$
Mínimo de una serie de valores (MIN)	$(MIN\ i)\ (A[\ i\ ] : 1 \leq i \leq n)$
Conteo del número de ocurrencias (N)	$(N\ i)\ (A[\ i\ ] = 3 : 1 \leq i \leq n)$

Si el rango de un cuantificador es vacío, esto es, no hay ningún valor de i que satisfaga el predicado, entonces el predicado es igual al elemento neutro de la operación representada.

Ejemplo

$Q \equiv \{a \geq 0 \wedge n \geq 0\}$

Funcion **POTENCIA** (a:entero, n:entero) retorna (p:entero)

    si n = 0 entonces retorna 1

    sino retorna **POTENCIA**(a, n-1) * a

    fsi

ffunción

$R \equiv \{p = a^n\}$

$\overline{x} \equiv (a,n)$	$B_t(a,n) \equiv (n=0)$
$\overline{y} \equiv p$	$B_{nt}(a,n) \equiv (n>0)$
$Q(a,n) \equiv (a \geq 0 \wedge n \geq 0)$	$s(a,n) \equiv (a,n-1)$
$R((a,n),p) \equiv (p = a^n)$	$triv(a,n) \equiv 1$
$c(p',(a,n)) \equiv p' * a$	

Ejemplo: FACTORIAL

Dado un número n, siendo n≥0, diseñar un función recursiva que retorne el factorial de dicho número n.

Q ≡ { n ≥ 0 }

Función **FACTORIAL** (n : entero) retorna (p : entero)

…

Ejemplo: NUMERO_CIFRAS

Dado un número n, siendo n≥0, diseñar un función recursiva que retorne el número de cifras de dicho número n.

Q ≡ { n ≥ 0 }

Función **NUMERO_CIFRAS** (n : entero) retorna (p : entero)

…

Ejemplo: SUMA_CIFRAS

Dado un número n, siendo n≥0, diseñar un función recursiva que retorne la suma de las cifras de dicho número n.

Q ≡ { n ≥ 0 }

Función **SUMA_CIFRAS** (n : entero) retorna (p : entero)

…

Ejemplo: FIBONACCI

Dado un número n, siendo n≥0, diseñar un función recursiva que retorne el número que ocupa la posición n en la **sucesión de Fibonacci**.

Nota.- La **sucesión de Fibonacci** es la siguiente sucesión infinita de números naturales:

"La sucesión comienza con los números 0 y 1, y a partir de éstos, cada término es la suma de los dos anteriores."

Q ≡ { n ≥ 0 }

Función **FIBONACCI** (n : entero) retorna (p : entero)

…

Ejemplo: SEMIFACTORIAL

Dado un número n, siendo n≥1, diseñar un función recursiva que retorne el **semifactorial**, o **doble factorial**, del entero n.

Nota.- el producto de todos los enteros desde el 1 hasta un entero no-negativo *n* que tiene la misma paridad (pares o impares) que *n* se llama **doble factorial** o **semifactorial** de *n* y se representa como *n*!!

Q ≡ { n ≥ 1 }

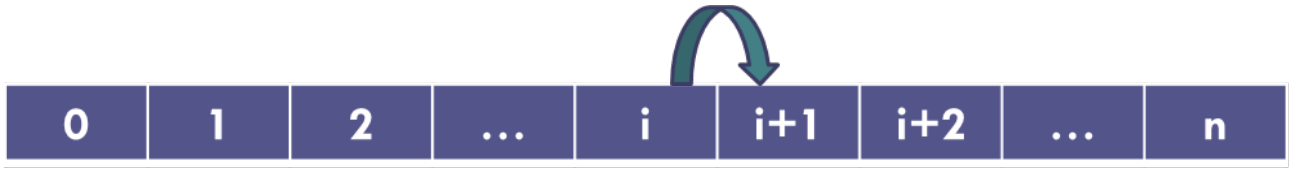
Función **SEMIFACTORIAL** (n : entero) retorna (p : entero)

…

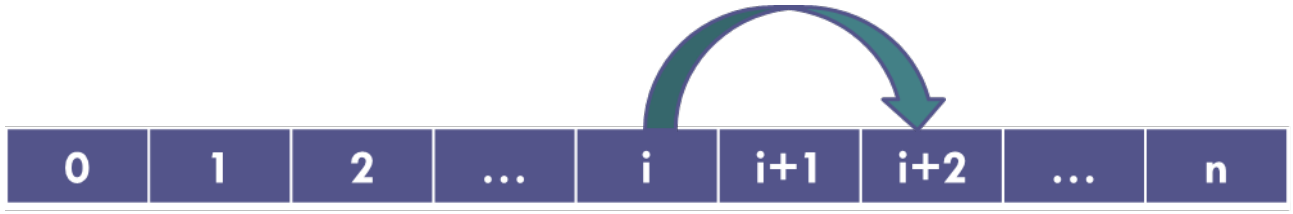
Ejemplo: RAYUELA

Dados unos cuadros alineados (como si se tratara del juego de la rayuela) y numerados desde el 0, el juego comienza en el cuadro 0 y consiste en ir saltando hasta el cuadro que ocupa la posición n, siendo n>0, permitiéndose para ello dos posibles movimientos:

Movimiento de longitud 1.-



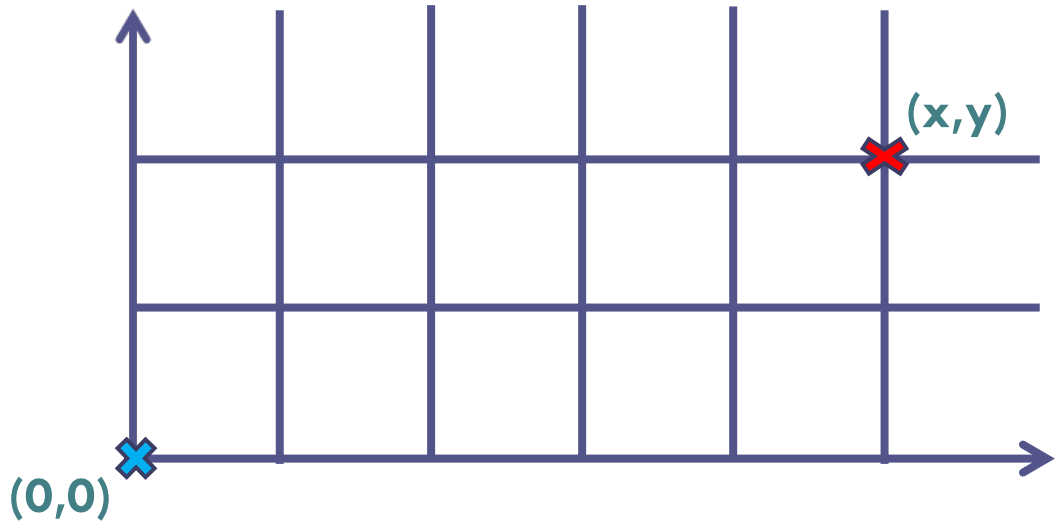
Movimiento de longitud 2.-



Se desea diseñar una función recursiva que calcule de cuántas maneras diferentes se puede llegar al n-ésimo cuadro desde el cuadro de partida (numerado con el 0)

Ejercicio: PLANO

Dado un plano n x n y dado un punto (x,y), con $x>0 \vee y>0$, en el plano, se desea saber cuántos caminos diferentes pueden llevarnos desde el origen (0,0) al punto (x,y) teniendo en cuenta que no se admiten retrocesos en el desplazamiento. Los únicos movimientos permitidos son a la derecha y hacia arriba en vertical, siendo dichos movimientos de longitud 1.



OTROS PRINCIPIOS DE INDUCCIÓN

- Hemos visto el principio de inducción matemática, que nos permite verificar si una propiedad se cumple sobre IN.

- Pero nuestro interés es más amplio. Queremos comprobar, apoyándonos en la inducción, una propiedad P muy importante: que un algoritmo cumple la postcondición para cualquier entrada que satisfaga la precondición. O sea, necesitamos aplicar la inducción a dominios que no son necesariamente los números naturales.

INDUCCIÓN NOETHERIANA

- Los métodos de demostración recordados se pueden utilizar directamente para establecer propiedades sobre dominios distintos del de los enteros naturales.

- Una técnica frecuentemente utilizada consiste en asociar un natural a cada elemento del dominio considerado y examinar la validez de la propiedad de los elementos del dominio en el orden de los naturales que le son asociados.

- El siguiente ejemplo clásico ilustra esta técnica:

INDUCCIÓN NOETHERIANA: ejemplo

Se quiere demostrar la siguiente propiedad **P**:

"**Todo conjunto de cardinal n contiene 2^n partes**".

De acuerdo con el proceso a seguir en la inducción noetheriana, necesitamos asociar a cada conjunto un entero (su cardinal). Definimos así la relación $\leq_{\mathbf{card}}$ (efectuamos "**una inducción sobre el cardinal del conjunto**")

INDUCCIÓN NOETHERIANA: ejemplo (continuación)

Base:

El conjunto de cardinal 0 es el conjunto vacío. Sólo hay una parte, él mismo.

Como $\mathbf{2^0 = 1}$, la propiedad queda demostrada.

Recurrencia:

Hipótesis: Todo conjunto de cardinal **n** tiene $\mathbf{2^n}$ **partes.**

Tesis: sea **E** un conjunto de cardinal **n+1**

INDUCCIÓN NOETHERIANA: ejemplo (continuación)

Demostración:

Sea **E** un conjunto de cardinal **n+1**.

Sea **a** un elemento cualquiera de **E** y sea **F** el subconjunto **E - {a}**.

Entonces, **E = {a} ∪ F** y **F** tiene **n** elementos.

Toda parte de **E** puede ser caracterizada por la presencia o ausencia del elemento **a**.

INDUCCIÓN NOETHERIANA: ejemplo (continuación)

- El conjunto de las partes de **E** que **no contienen al elemento a** es el conjunto de las partes de **F**. Por la hipótesis de recurrencia, hay entonces $\mathbf{2^n}$ partes de **E** que no contienen al elemento **a**.

- Por otro lado, el conjunto de las partes de **E** que **contienen al elemento a** se puede construir simplemente añadiendo a cada parte de **F**, el elemento **a**. Por lo tanto, hay también $\mathbf{2^n}$ partes de **E** que contienen el elemento **a**.

En conclusión, **E** tiene $\mathbf{2^n + 2^n}$ partes, es decir, $\mathbf{2^{n+1}}$ partes cqd.

Conclusión:

Todo conjunto de cardinal **n** contiene $\mathbf{2^n}$ partes.

INDUCCIÓN NOETHERIANA: ejemplo (continuación)

$$E = \{ a, b, c \} \qquad F = \{ b, c \}$$

$$E = \{a\} \cup F$$

Conjunto de las partes de E.-

- partes de **E** que **NO CONTIENEN al elemento a** son las partes de **F**, es decir,

$$\mathcal{P}(F) = \{ \emptyset, \{b\}, \{ c\}, \{b,c\} \}$$

- partes de **E** que **CONTIENEN al elemento a**, se construyen añadiendo el elemento **a** cada parte de **F**, esto es,

$$\{ \{a\}, \{a,b\}, \{a,c\}, \{a,b,c\} \}$$

En conclusión $\mathcal{P}(E) = \{ \emptyset, \{b\}, \{c\}, \{b,c\}, \{a\}, \{a,b\}, \{a,c\}, \{a,b,c\} \}$

INDUCCIÓN ESTRUCTURAL

Supongamos que pretendemos demostrar que una propiedad **P** se cumple para todos los elementos de un dominio **D definido por recurrencia**.

Base

Demostrar que **P** es cierta para todos los elementos de **D** que constituyen la **base** de la definición recurrente del dominio.

Recurrencia o paso inductivo

Hipótesis

Sea **e**∈**D** un elemento cualquiera del dominio. Suponemos que **P** es cierta para todos los elementos del dominio que sirven para la creación de **e**.

Tesis

Demostrar que **P** se cumple para **e**.

Conclusión

P se cumple para todo elemento de **D**.

INDUCCIÓN ESTRUCTURAL: ejemplo

Sea $X = \{X_1, X_2, ..., X_i, X_{i+1}, ...\} = \{ 1, 3, 5, ..., 15, 17, ...\}$

la sucesión de **números impares.**

Queremos demostrar que sobre **X** se cumple la siguiente propiedad:

$$X_i = 2\,i - 1 \quad \forall i \geq 1$$

De acuerdo con el proceso que debemos seguir en la inducción estructural, nos apoyaremos en una **definición recurrente de X.** En este caso, es la siguiente:

Base de la definición del dominio: $1 \in X$

Recurrencia de la definición del dominio: si $y \in X$ entonces $y + 2 \in X$

INDUCCIÓN ESTRUCTURAL: ejemplo (continuación)

Base:

Demostrar que la propiedad es cierta para el **1**.

En efecto, puesto que $X_1 = 2 * i - 1 = 2 * 1 - 1 = 1$

Recurrencia:

Hipótesis: Suponemos que la propiedad es cierta para X_i, o sea, $X_i = 2i-1$

Tesis: Entonces hay que demostrar $X_{i+1} = 2(i+1) - 1$

Demostración: $X_{i+1} = X_i + 2 = (2\,i - 1) + 2 = 2(i+1) - 1$ cqd.

H.I.

Conclusión:

La propiedad se cumple para todos los elementos de **X**.

INMERSIÓN

Sucede con frecuencia que no es posible abordar directamente el diseño recursivo de una función f porque no se encuentra una descomposición adecuada de los datos.

En esos casos, una técnica que puede solucionar el diseño consiste en definir una función **g**, más general que ***f***, con más parámetros y/o más resultados, que para ciertos valores de los nuevos parámetros calcula lo mismo que ***f***.

INMERSIÓN

Diremos que hemos aplicado una inmersión de ***f*** en ***g***. La función más general, ***g***, se denomina ***función inmersora***. Y la función original, ***f***, se denomina ***función sumergida***.

Para calcular la función original basta establecer el valor inicial de los nuevos parámetros que hacen que la función ***g*** se comporte como la función ***f***.

$$\begin{array}{l}\{\ Q(\bar{x})\ \}\\ \text{Función } f(\bar{x}) \text{ retorna } \bar{y}\\ \{\ R(\bar{x}\ \bar{y})\}\end{array} \quad \Longrightarrow \quad \begin{array}{l}\{\ Q'(\bar{x},\bar{w})\ \}\\ \text{Función } g(\bar{x},\bar{w}) \text{ retorna } \bar{y}\\ \{\ R'(\bar{x},\bar{w},\bar{y})\}\end{array}$$

INMERSIÓN

Vamos a ilustrarlo a través de un ejemplo sencillo.

"Dado un vector A[1..n], con n>0, se pide diseñar una función recursiva que calcule la suma de los elementos del vector"

Su especificación formal es la siguiente.-

$Q \equiv \{n > 0\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

INMERSIÓN NO FINAL

El diseño recursivo de **SUMA** no parece abordable directamente ya que tendríamos que "descomponer" el vector A en un vector más pequeño y del mismo tipo, lo cual es contradictorio. Al hacerlo más pequeño, claramente estamos cambiando su tipo, con lo que no podríamos llamar recursivamente a **SUMA**.

Lo que sí que podemos añadir es un parámetro entero j y definir una función que calcule "la parte" de **SUMA** que se extiende de 1 a j.

Para llevar a cabo el diseño recursivo pedido vamos a aplicar la técnica de inmersión no final.

INMERSIÓN NO FINAL

1.- Consiste en obtener **R´** a partir de **R** a través de un proceso de sustitución simbólica, el cual consiste en sustituir expresiones o constantes por nuevas variables.

$Q \equiv \{n > 0\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$\boldsymbol{R} \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq \boldsymbol{n})\}$

/* Sustituimos n por j */

$Q' \equiv \{\, ¿\, ?\, \}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

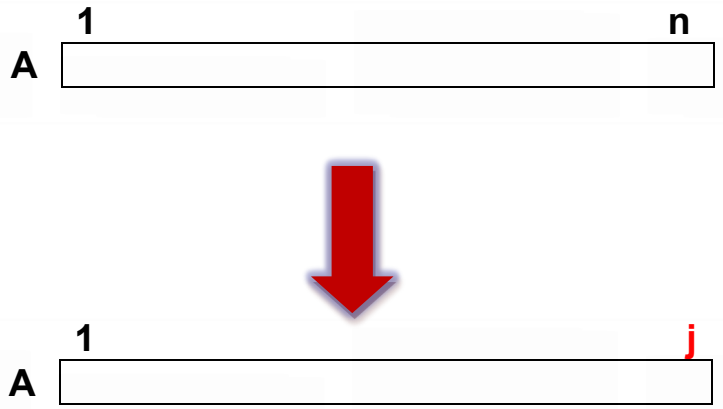
$\boldsymbol{R'} \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq \boldsymbol{j})\}$

$$\boldsymbol{R' \wedge (j = n) \equiv R}$$

INMERSIÓN NO FINAL

2.- La precondición **Q´** se obtiene mediante la conjunción de **Q** y el dominio de la nueva variable, en nuestro ejemplo, j.

$$Q' \equiv Q \wedge \text{Dominio (j)}$$



$1 \leq j \leq n$

INMERSIÓN NO FINAL

$Q \equiv \{n > 0\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

$\boldsymbol{Q' \equiv \{1 \leq j \leq n\}}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

$\boldsymbol{R' \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq j)\}}$

INMERSIÓN NO FINAL

3.- Valor que debemos dar a j en la invocación a iSUMA para conseguir que iSUMA devuelva el mismo valor que SUMA.

$Q \equiv \{n > 0\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

$\boldsymbol{Q' \equiv \{1 \leq j \leq n\}}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

$\boldsymbol{R' \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq j)\}}$

$\boldsymbol{j_{ini} = n \rightarrow SUMA(A) = iSUMA(A, n)}$

lo que nos garantiza que: $\boldsymbol{R' \wedge (j = n) \rightarrow R}$

INMERSIÓN NO FINAL

Procedemos a continuación a realizar el análisis por casos de la función iSUMA.

$Q' \equiv \{1 \leq j \leq n\}$

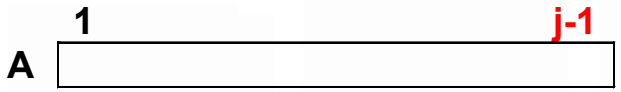
Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

$R' \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq j\ )\}$

INMERSIÓN NO FINAL

Queremos calcular la suma de la sección del vector que va de 1 a j (¿e? o ¿iSUMA(A,j)?)

Suponemos conocida la suma de la sección del vector que se extiende de 1 a j-1, esto es, e´ o iSUMA(A,j-1)



En conclusión, e = e´ + A[j] o iSUMA(A,j) = iSUMA(A,j-1) + A[j]

El caso trivial corresponde a j=1, esto es, a la suma de la sección del vector [1..1], siendo su valor asociado A[1].

INMERSIÓN NO FINAL

$Q' \equiv \{1 \leq j \leq n\}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

caso

j = 1 → A[1]

j > 1 → iSUMA(A, j-1) + A[j]

fcaso

ffunción

$R' \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq j)\}$

INMERSIÓN NO FINAL: Sustituir constante por variable

[1 .. n] → [1 .. j]

[1 .. n] → [j .. n]

[1 .. n] → [i .. j]

INMERSIÓN NO FINAL

Dado un vector A[1..n], con n>0, se pide diseñar una función recursiva que calcule la suma de los elementos del vector

$Q \equiv \{n > 0\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

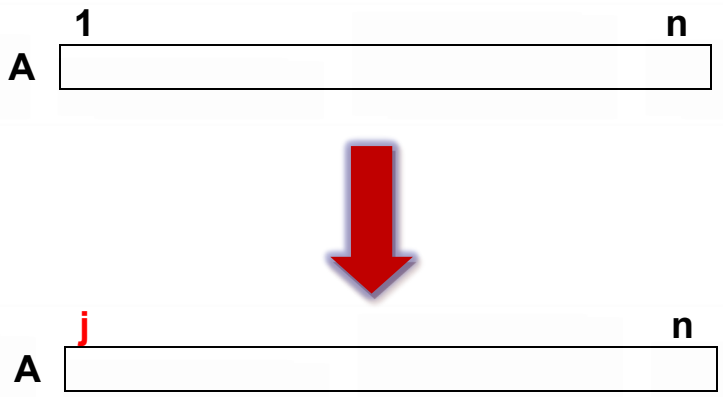
$\boldsymbol{Q'} \equiv \{¿\,?\}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

$\boldsymbol{R'} \equiv \{e = (\sum i)(A[i]: \boldsymbol{j} \leq i \leq n)\}$

$$\boldsymbol{R' \wedge (j = 1) \rightarrow R}$$

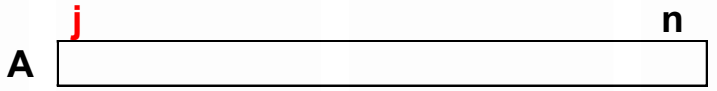
INMERSIÓN NO FINAL



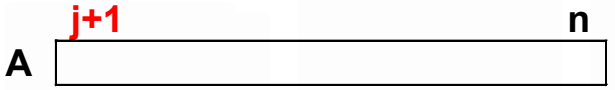
$1 \leq j \leq n$

INMERSIÓN NO FINAL

Queremos calcular la suma de la sección del vector que va de j a n (¿e? o ¿ iSUMA(A,j) ?)



Supongamos conocida la suma de la sección del vector que se extiende de j+1 a n, la denominaremos e´ o iSUMA(A,j+1)



En conclusión, e = e´ + A[j] o iSUMA(A,j) = iSUMA(A,j+1)+A[j])

El caso trivial corresponde a j=n, esto es, a la suma de la sección del vector [n..n], siendo su valor asociado A[n].

INMERSIÓN NO FINAL

$Q \equiv \{n > 0\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

$\boldsymbol{Q' \equiv \{1 \leq j \leq n\}}$

Función **iSUMA** (A[1..n]:vector de enteros, j:entero) retorna (e:entero)

caso

j = n → A[n]

j < n → iSUMA(A, j+1) + A[j]

fcaso

ffunción

$\boldsymbol{R' \equiv \{e = (\sum i)(A[i]: j \leq i \leq n)\}}$

j_{ini} = 1 ➔ SUMA(A) = iSUMA(A,1)

Ejemplo: SUMA VECTOR (rango vacío)

Dado un vector A[1..n], con n≥0, se pide diseñar una función recursiva que calcule la suma de los elementos del vector

$Q \equiv \{\boldsymbol{n \geq 0}\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

$\boldsymbol{Q' \equiv \{0 \leq j \leq n\}}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

caso

j = 0 → 0

j > 0 → iSUMA(A, j-1) + A[j]

fcaso

ffunción

$R' \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq \boldsymbol{j})\}$

Ejemplo: SUMA VECTOR (rango vacío)

Dado un vector A[1..n], con n≥0, se pide diseñar una función recursiva que calcule la suma de los elementos del vector

$Q \equiv \{\boldsymbol{n \geq 0}\}$

Función **SUMA** (A[1..n]:vector de enteros) retorna (e:entero)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)\}$

$\boldsymbol{Q' \equiv \{1 \leq j \leq n+1\}}$

Función **iSUMA** (A[1..n]:vector de enteros, **j:entero**) retorna (e:entero)

caso

j = n+1 → 0

j < n+1 → iSUMA(A, j+1) + A[j]

fcaso

ffunción

$\boldsymbol{R'} \equiv \{e = (\sum i)(A[i]: \boldsymbol{j} \leq i \leq n)\}$

j_{ini} = 1 ➔ SUMA(A) = iSUMA(A,1)

Ejemplo: VECTOR CAPICÚA

Dado un vector A[1..n], con n>0, se pide diseñar una función recursiva que determine si dicho vector es o no capicúa

Su especificación formal es la siguiente.-

$Q \equiv \{n > 0\}$

Función **CAPICUA** (A[1..n]:vector de enteros) retorna (s:booleano)

$R \equiv \{s = (\forall k)(A[k] = A[n-k+1]: 1 \leq k \leq n)\}$

El diseño recursivo de CAPICÚA no parece abordable directamente ya que tendríamos que "descomponer" el vector A en un vector más pequeño y del mismo tipo, lo cual es contradictorio. Al hacerlo más pequeño, claramente estamos cambiando su tipo, con lo que no podríamos llamar recursivamente a CAPICÚA .

Ejemplo: VECTOR CAPICÚA

Lo que sí que podemos añadir son dos parámetros enteros i y j y definir una función que calcule "la parte" de **CAPICUA** que se extiende de i a j.

Para llevar a cabo el diseño recursivo pedido vamos a aplicar la técnica de inmersión no final: Sustituimos 1 por i y n por j, determinamos el dominio de i y de j, así como los valores que debemos dar a i y j en la invocación a **iCAPICUA** para conseguir que **iCAPICUA** devuelva el mismo valor que **CAPICUA**.

$\boldsymbol{Q' \equiv \{1 \leq i \leq j \leq n\}}$

Función **iCAPICUA** (A[1..n]:vector de enteros, **i,j:entero**) retorna (s:booleano)

$\boldsymbol{R' \equiv \{s = (\forall k)(A[k] = A[j - k + i]: i \leq k \leq j)\}}$

Ejemplo: VECTOR CAPICÚA

Tras realizar el análisis por casos resulta:

$$Q' \equiv \{1 \leq i \leq j \leq n\}$$

Función **iCAPICUA** (A[1..n]: vector de enteros, **i, j:entero**) retorna (s:booleano)
caso
    i = j → VERDADERO
    i = j-1 → (A[i] = A[j])
    i < j-1 → iCAPICUA(A, i+1, j-1) AND (A[i] = A[j])
fcaso
ffuncion

$$R' \equiv \{s = (\forall k)(A[k] = A[j - k + i]: i \leq k \leq j)\}$$

La llamada inicial a la función se realizará con i=1 y j=n, de ese modo.-

i_{ini}= 1 y j_{ini}= n → CAPICUA(A) = iCAPICUA(A, 1, n)

Ejemplo: VECTOR CAPICÚA (rango vacío)

Dado un vector A[1..n], con n≥0, se pide diseñar una función recursiva que determine si dicho vector es o no capicúa

Su especificación formal es la siguiente.-

$Q \equiv \{\boldsymbol{n \geq 0}\}$

Función **CAPICUA** (A[1..n]:vector de enteros) retorna (s:booleano)

$R \equiv \{s = (\forall k)(A[k] = A[n-k+1]: 1 \leq k \leq n)\}$

Ejemplo: VECTOR CAPICÚA (rango vacío)

Vamos a añadir dos parámetros enteros i y j y definir una función que calcule "la parte" de **CAPICUA** que se extiende de i a j.

Para llevar a cabo el diseño recursivo pedido vamos a aplicar la técnica de inmersión no final: Sustituimos 1 por i y n por j, determinamos el dominio de i y de j, así como los valores que debemos dar a i y j en la invocación a **iCAPICUA** para conseguir que **iCAPICUA** devuelva el mismo valor que **CAPICUA**.

$\boldsymbol{Q' \equiv \{1 \leq i \leq j+1 \leq n+1\}}$

Función **iCAPICUA** (A[1..n]:vector de enteros, **i,j:entero**) retorna (s:booleano)

$\boldsymbol{R' \equiv \{s = (\forall k)(A[k] = A[j-k+i]: i \leq k \leq j)\}}$

Ejemplo: VECTOR CAPICÚA (rango vacío)

Tras realizar el análisis por casos resulta:

$$Q' \equiv \{1 \leq i \leq j+1 \leq n+1\}$$

Función **iCAPICUA** (A[1..n]: vector de enteros, **i, j:entero**) retorna (s:booleano)
caso
    i > j → VERDADERO
    i ≤ j-1 → iCAPICUA(A, i+1, j-1) AND (A[i] = A[j])
fcaso
ffuncion

$$R' \equiv \{s = (\forall k)(A[k] = A[j-k+i] : i \leq k \leq j)\}$$

La llamada inicial a la función se realizará con i=1 y j=n, de ese modo.-

i_{ini}= 1 y j_{ini}= n → CAPICUA(A) = iCAPICUA(A, 1, n)

Ejemplo: SIMETRÍA DE UNA MATRIZ

Dada una matriz A[1..n][1..n], con n>0, se pide diseñar una función recursiva que determine si dicha matriz es simétrica o no

Su especificación formal es la siguiente.-

$Q \equiv \{n > 0\}$

Función **SIMETRICA** (A[1..n][1..n]:matriz de enteros) retorna (b:booleano)

$R \equiv \{b = (\forall i)((\forall j)(A[i][j] = A[j][i]: 1 \leq j \leq n): 1 \leq i \leq n)\}$

El diseño recursivo de **SIMETRICA** no parece abordable directamente ya que tendríamos que "descomponer" la matriz A en una matriz más pequeña y del mismo tipo, lo cual es contradictorio.

Ejemplo: SIMETRÍA DE UNA MATRIZ

Vamos a añadir un parámetro entero k y definir una función que calcule "la parte" de **SIMETRICA** que se extiende desde la fila 1 a la fila k y de la columna 1 a la columna k.

Para llevar a cabo el diseño recursivo pedido vamos a aplicar la técnica de inmersión no final: Sustituimos n por k, determinamos el dominio de k, así como los valores que debemos dar a k en la invocación a **iSIMETRICA** para conseguir que **iSIMETRICA** devuelva el mismo valor que **SIMETRICA.**

$\boldsymbol{Q' \equiv \{1 \leq k \leq n\}}$

Función **iSIMETRICA** (A[1..n][1..n]:matriz de enteros, **k:entero**) retorna (b:booleano)

$\boldsymbol{R'} \equiv \{b = (\forall i)((\forall j)(A[i][j] = A[j][i]: 1 \leq j \leq \boldsymbol{k}): 1 \leq i \leq \boldsymbol{k})\}$

Ejemplo: SIMETRÍA DE UNA MATRIZ

Tras realizar el análisis por casos resulta:

$Q' \equiv \{1 \leq k \leq n\}$

Función **iSIMETRICA** (A[1..n][1..n]:matriz de enteros, **k:entero**) retorna (b:booleano)
    caso
        k=1 → VERDADERO
        k>1 → iSIMETRICA(A,k-1) AND banda_simetrica(A,k)
    fcaso
ffuncion

$R' \equiv \{b = (\forall i)((\forall j)(A[i][j] = A[j][i]: 1 \leq j \leq k): 1 \leq i \leq k)\}$

donde banda_simétrica(A,k) $= (\forall i)(A[i][k] = A[k][i]: 1 \leq i \leq k-1)$

$k_{ini} = n$ → **SIMETRICA(A) = iSIMETRICA(A,n)**

Ejemplo: SIMETRÍA DE UNA MATRIZ

donde la función banda_simétrica es:

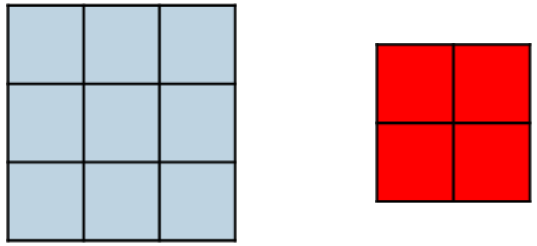
$Q \equiv \{2 \leq k \leq n\}$

```
Función banda_simetrica (A[1..n][1..n]:matriz de enteros, k:entero) retorna (b:booleano)

  var i:entero, iguales:booleano fvar

  i = 0;
  iguales = VERDADERO;

  mientras (i < k-1 AND iguales ) hacer
       i = i + 1;
       si ( A[ i ][ k ] != A[ k ][ i ] ) entonces iguales = FALSO fsi
  fmientras

  retorna iguales;

ffuncion
```

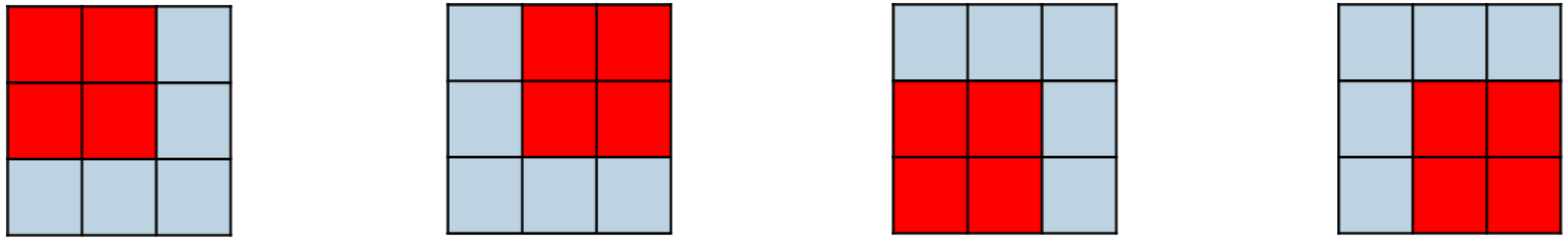
$R \equiv \{b = (\forall i)(A[i][k] = A[k][i]: 1 \leq i \leq k-1)\}$

Ejemplo: TABLEROS

Calcular el número de formas diferentes de colocar un tablero de m x m sobre otro de n x n donde $n \geq m$, $n > 0$ y $m > 0$. Ejemplo: n=3 y m=2



Solución: 4



Ejemplo: MEDIA ARITMÉTICA

Dado un vector A[1..n] de reales, con n>0, se pide diseñar una función recursiva que calcule la media aritmética de los elementos del vector

$Q \equiv \{n > 0\}$

Función **MEDIA** (A[1..n]:vector de reales) retorna (e:real)

$R \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq n)/n\}$

$Q' \equiv \{\ \mathbf{1 \leq k \leq n}\ \}$

Función **iMEDIA** (A[1..n]:vector de reales, **k:entero**) retorna (e:real)

$R' \equiv \{\ e = (\sum i)(A[i]: 1 \leq i \leq \mathbf{k})/\mathbf{k}\}$

Ejemplo: MEDIA ARITMÉTICA

Tras realizar el análisis por casos resulta:

$Q' \equiv \{1 \leq k \leq n\}$

Función iMEDIA (A[1..n]:vector de reales, k:entero) retorna (e:real)

caso

k = 1 → A[1]
k > 1 → (iMEDIA(A,k-1) * (k – 1) + A[k]) / k
fcaso

ffuncion

$R' \equiv \{e = (\sum i)(A[i]: 1 \leq i \leq k) \ / \ k\}$

La llamada inicial a la función se realizará con k=n, de ese modo.-

k_{ini} = n → MEDIA(A) = iMEDIA (A, n)

VERIFICACIÓN FORMAL

Con el fin de dar rigor y formalizar el proceso de verificación, enunciamos a continuación las 6 propiedades que necesitamos demostrar para tener garantía de que un algoritmo es correcto.

1. $Q(\overline{x}) \Rightarrow B_t(\overline{x}) \vee Bnt(\overline{x})$
2. $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \Rightarrow Q\big(s(\overline{x})\big)$
3. $Q(\overline{x}) \wedge B_t(\overline{x}) \Rightarrow R\big(\overline{x}, triv(\overline{x})\big)$
4. $Q(\overline{x}) \wedge Bnt(\overline{x}) \wedge R(s(\overline{x}), \overline{y}') \Rightarrow R\big(\overline{x}, c(\overline{y}', \overline{x})\big)$
5. Encontrar una función t : D $\rightarrow$ Z tal que $Q(\overline{x}) \Rightarrow t(\overline{x}) \geq 0$
6. $Q(\overline{x}) \wedge Bnt(\overline{x}) \Rightarrow t\big(s(\overline{x})\big) < t(\overline{x})$

La demostración de estas propiedades es, en general, bastante sencilla, y nos asegura la corrección del algoritmo. Obsérvese que todos los símbolos que intervienen en las propiedades anteriores aparecen en el esquema general de las funciones recursivas.

VERIFICACIÓN FORMAL

Los puntos 1 y 2 son condiciones que expresan que la función f está bien definida:

$$\mathbf{1.}\ \boldsymbol{Q}(\overline{\boldsymbol{x}}) \Rightarrow \boldsymbol{B_t}(\overline{\boldsymbol{x}}) \vee \boldsymbol{Bnt}(\overline{\boldsymbol{x}})$$

La función f ha de estar bien definida dentro del dominio, es decir, la instrucción condicional no aborta en los estados que satisfacen Q.

$$\mathbf{2.}\ \boldsymbol{Q}(\overline{\boldsymbol{x}}) \wedge \boldsymbol{B_{nt}}(\overline{\boldsymbol{x}}) \Rightarrow \boldsymbol{Q}\big(\boldsymbol{s}(\overline{\boldsymbol{x}})\big)$$

La función f se invoca siempre en estados que satisfacen su precondición.

VERIFICACIÓN FORMAL

$$Q(\overline{x}) \Rightarrow R\left(\overline{x}, f(\overline{x})\right) \quad \forall \overline{x} \in D$$

3. $Q(\overline{x}) \wedge Bt(\overline{x}) \Rightarrow R\left(\overline{x}, triv(\overline{x})\right)$

Base de la inducción

4. $Q(\overline{x}) \wedge Bnt(\overline{x}) \wedge R(s(\overline{x}), \overline{y}') \Rightarrow R\left(\overline{x}, c(\overline{y}', \overline{x})\right)$

Paso de la inducción donde $R(s(\overline{x}), \overline{y}')$

representa la hipótesis de inducción

VERIFICACIÓN FORMAL

Los puntos 5 y 6 respectivamente definen la estructura del pbf y demuestran que las llamadas se hacen sobre datos que forman una secuencia estrictamente decreciente:

5. Encontrar una función t : D → Z tal que $Q(\bar{x}) \Rightarrow t(\bar{x}) \geq 0$

6. $Q(\bar{x}) \wedge Bnt(\bar{x}) \Rightarrow t\big(s(\bar{x})\big) < t(\bar{x})$

VERIFICACIÓN FORMAL: ejemplo 1

Vamos a estudiar la corrección del ejemplo de la potencia n-ésima de a:

$\bar{x} \equiv (a,n)$	$B_t(a,n) \equiv (n=0)$
$\bar{y} \equiv p$	$B_{nt}(a,n) \equiv (n>0)$
$Q(a,n) \equiv (a \geq 0 \wedge n \geq 0)$	$s(a,n) \equiv (a,n-1)$
$R((a,n),p) \equiv (p=a^n)$	$triv(a,n) \equiv 1$
$c(p',(a,n)) \equiv p' * a$	

Sustituyendo estas equivalencias en los 6 puntos de demostración tenemos:

1. $Q(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$

$n \geq 0 \Rightarrow n = 0 \vee n > 0$

2. $Q(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow Q(s(\bar{x}))$

$n \geq 0 \wedge n > 0 \Rightarrow (n-1) \geq 0$

VERIFICACIÓN FORMAL: ejemplo 1 (continuación)

3. $Q(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow R(\bar{x}, triv(\bar{x}))$

$n \geq 0 \wedge n = 0 \Rightarrow 1 = a^n$

¿ $p = a^n$ cuando $n = 0$? Sí, ya que $p=1$ y $a^0=1$, por lo que $1=1$.

4. $Q(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge R(s(\bar{x}), \bar{y}') \Rightarrow R(\bar{x}, c(\bar{y}', \bar{x}))$

$n \geq 0 \wedge n > 0 \wedge (p' = a^{n-1}) \Rightarrow a * p' = a^n$

¿ $p = a^n$?

$p = a * p' = a * a^{n-1} = a^n$

VERIFICACIÓN FORMAL: ejemplo 1 (continuación)

5. **Encontrar una función $t:D \rightarrow Z$, tal que $Q(\bar{x}) \Rightarrow t(\bar{x}) \geq 0$**

$t(a,n) \overset{def}{=} n$

$n \geq 0 \Rightarrow n \geq 0$

6. **$Q(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow t(s(\bar{x})) < t(\bar{x})$**

$n \geq 0 \wedge n > 0 \Rightarrow n-1 < n$

Ejemplo propuesto para verificar formalmente

{n ≥ 0}
Funcion **FACTORIAL(n:entero)** retorna (f:entero)

caso
n = 0 → 1
n > 0 → **FACTORIAL(n-1)** * n
fcaso

función
{ f = (Π i) (i: 1≤ i ≤ n)}

VERIFICACIÓN FORMAL: ejemplo 2

{ n ≥ 0 }
Funcion **F (n:entero)** retorna (p:entero)

```
caso
        n = 0 → 1
        n = 1 → 3
        n > 1 → 2 * F(n - 1) + 3 * F(n - 2)
fcaso

ffuncion
```

{ p = 3^n }

VERIFICACIÓN FORMAL: ejemplo 2 (continuación)

{ n ≥ 0 }
Funcion **F (n:entero)** retorna (p:entero)

caso
n = 0 → 1
n = 1 → 3
n > 1 → 2 * **F(n - 1)** + 3 * **F(n - 2)**
fcaso

ffuncion
{ p = 3^n}

1. $Q(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$

$n \geq 0 \Rightarrow n = 0 \vee n = 1 \vee n > 1$

2. $Q(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow Q(s(\bar{x}))$

$n \geq 0 \wedge n > 1 \Rightarrow (n-1) \geq 0$
$n \geq 0 \wedge n > 1 \Rightarrow (n-2) \geq 0$

VERIFICACIÓN FORMAL: ejemplo 2 (continuación)

3. $Q(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow R(\bar{x}, triv(\bar{x}))$

$n \geq 0 \wedge n = 0 \Rightarrow 1 = 3^n$

¿ $p = 3^n$ cuando $n = 0$? Sí, ya que $p=1$ y $3^0=1$, por lo que $1=1$.

$n \geq 0 \wedge n = 1 \Rightarrow 3 = 3^n$

¿ $p = 3^n$ cuando $n = 1$? Sí, ya que $p=3$ y $3^1=3$, por lo que $3=3$.

VERIFICACIÓN FORMAL: ejemplo 2 (continuación)

4. $Q(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge R(s(\bar{x}), \bar{y}') \Rightarrow R(\bar{x}, c(\bar{y}', \bar{x}))$

$n \geq 0 \wedge n > 1 \wedge (p_1 = 3^{n-1}) \wedge (p_2 = 3^{n-2}) \Rightarrow 2 * p_1 + 3 * p_2 = 3^n$

¿ $p = 3^n$?

$p = 2 * p_1 + 3 * p_2 = 2 * 3^{n-1} + 3 * 3^{n-2} = 2 * 3^{n-1} + 3^{n-1} = (2+1) * 3^{n-1} =$

$= 3 * 3^{n-1} = 3^n$

VERIFICACIÓN FORMAL: ejemplo 2 (continuación)

5. Encontrar una función $t:D \to Z$, tal que $Q(\bar{x}) \Rightarrow t(\bar{x}) \geq 0$

$t(n) \overset{def}{=} n$

$n \geq 0 \Rightarrow n \geq 0$

6. $Q(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow t(s(\bar{x})) < t(\bar{x})$

$n \geq 0 \wedge n > 1 \Rightarrow n-1 < n$

$n \geq 0 \wedge n > 1 \Rightarrow n-2 < n$