

## Tema 3: Diseño de algoritmos iterativos

precondición más débil

Invariante.

Clara María Segura Díaz  
Fundamentos de Algoritmia, Curso 2020-21

Tema muy importante de  
algoritmos. Enfocado en algoritmos  
iterativos

Dpto. de Sistemas Informáticos y Computación  
Facultad de Informática  
Universidad Complutense de Madrid

- **Diseño de programas. Formalismo y abstracción;** R. Peña. Tercera edición. Prentice Hall, 2005.  
**Secciones 4.1 y 4.2**
- **Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios.** N. Martí, C. Segura, J.A. Verdejo. Ibergarceta Publicaciones, 2012.  
**Capítulo 4**
- **The Derivation of algorithms;** A. Kaldewaij; Prentice Hall, 1990.  
**Capítulo 2**

- **Verificación de algoritmos**
  - Reglas de verificación básicas
  - Verificación de instrucciones
  - Ejemplos de verificación
- **Derivación de algoritmos**
  - Algoritmos numéricos
  - Problemas de búsqueda
  - Variables acumuladoras
  - Segmento más largo
  - Modificación de vectores

## Verificación de algoritmos

Verificar consiste en demostrar que un algoritmo cumple con su especificación. Es decir, que la precondición está bien definida y que la postcondición se cumple.

# Verificación

- **Verificar** consiste en demostrar con un razonamiento suficientemente claro que un algoritmo cumple su especificación.

- 1 • **Verificación**: “a posteriori”, cuando el algoritmo está terminado se razona sobre él.
- 2 • **Derivación**: el algoritmo se diseña y se verifica a la vez.

Verificación se realiza después de haber hecho el algoritmo y la derivación es que el algoritmo se diseña y verifica a la misma vez

Nosotros nos encargamos de hacer derivaciones

# Verificación

## Idea de por qué nuestro programa es correcto

- **Verificar** consiste en demostrar con un razonamiento suficientemente claro que un algoritmo cumple su especificación. cumple lo que queremos hacer.

1 • **Verificación:** "a posteriori", cuando el algoritmo está terminado se razona sobre él. No, solo si el programa no es nuestro.

2 • **Derivación:** el algoritmo se diseña y se verifica a la vez. Esta si.

- Las reglas de verificación sirven para demostrar la corrección de un algoritmo con respecto a su especificación. Corregir un algoritmo solo si no es nuestro
- Necesitamos reglas para decidir si una instrucción elemental  $A$  satisface una especificación dada  $\{P\} A \{Q\}$ .

Nuestros algoritmos debemos derivarlos ya que tendremos que diseñarlo y verificarlo a la vez

Premisas

Conclusión

si cumple premisas  
cumple conclusión

- Dados  $P$ ,  $A$ , y  $Q$ , las reglas permiten decidir si se satisface  $\{P\} A \{Q\}$ .
- Conocer dichas reglas para cada instrucción del lenguaje es equivalente a conocer la **semántica** del mismo (semántica axiomática). con axiomas

Acordarnos que  $\{p\}$  y  $\{q\}$  son precondición y postcondición respectivamente.

## Verificación de algoritmos

### Reglas de verificación básicas

Las reglas de verificación y de derivación son las mismas solo que se realizan en distinto tiempo.

Verificación: después de haber diseñado el algoritmo (con su especificación y todo)  
Derivación: a la vez que se diseña el algoritmo

## Fortalecimiento de la precondition

Siempre se puede fortalecer la precondition.

Supongamos que  $A$  cumple la siguiente especificación donde  $x$  es de tipo entero:

$$\{x \leq 5\}$$

$A$

$$\{x \leq 10\}$$

¿Qué ocurre si cambiamos la precondition por un predicado que la implique?

Como  $x \leq -7 \Rightarrow x \leq 5$ , se cumple también la especificación

$$\{x \leq -7\}$$

$A$

$$\{x \leq 10\}$$

satisface

Mejor esta que  $-5$  porque es más restrictivo.

Es más restrictivo  $x \leq -7$  que  $x \leq 5$ . Eso es FORTALECER la precondition.

Porque abarca menos casos, por lo tanto es más restrictivo.

## Fortalecimiento de la precondition

Podemos escribirlo como regla de verificación de la siguiente manera:

$$\frac{P' \Rightarrow P \quad \{P\} A \{Q\}}{\{P'\} A \{Q\}}$$

$$x \leq -7 \Rightarrow x \leq 5$$

$$\{x \leq 5\} A \{x \geq 10\}$$

---

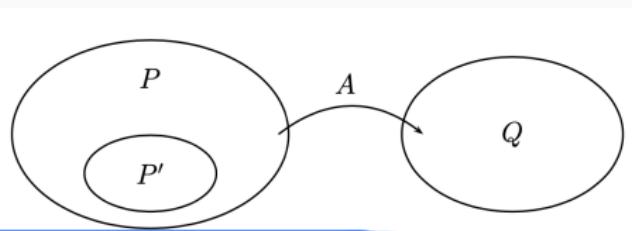
$$\{x \leq -7\} A \{x \geq 10\}$$

## Fortalecimiento de la precondition

Podemos escribirlo como regla de verificación de la siguiente manera:

$$\frac{P' \Rightarrow P \quad \{P\} A \{Q\}}{\{P'\} A \{Q\}}$$

Lo ilustramos por medio de un dibujo, viendo los predicados como conjuntos de estados.



Si  $P'$  satisface  $P$  y  $P$  satisface  $Q$ , entonces  $P'$  satisface  $Q$

## Debilitamiento de la postcondición

Siempre se va a poder debilitar.

Supongamos que  $A$  cumple la siguiente especificación, donde  $V[100]$  es un vector de enteros: hacer la precondition más débil

{true} Para cualquier entrada

$A$

$\{(\exists j : 0 \leq j \leq 50 : v[j] \bmod 2 = 0)\}$

Existe un indice  $j < 50$  t.q el valor en una posición de  $v$  vector  $j$  es par. Si para 50 hay una posición, también la habrá para 100, 200...

¿Qué ocurre si cambiamos la postcondición por un predicado implicado por ella?

Como

$(\exists j : 0 \leq j \leq 50 : v[j] \bmod 2 = 0) \Rightarrow (\exists j : 0 \leq j < 100 : v[j] \bmod 2 = 0)$

se cumple la especificación

{true}

$A$

$\{(\exists j : 0 \leq j < 100 : V[j] \bmod 2 = 0)\}$

si de las 50 primeras posiciones del vector hay una posición par, para 100 también habrá una posición par.

## Debilitamiento de la postcondición

Lo escribimos como regla de verificación de la siguiente manera:

$$\frac{\{P\} A \{Q\} \quad Q \Rightarrow Q'}{\{P\} A \{Q'\}}$$

Si hay una precondición que satisface una postcondición, entonces si debilitamos la postcondición, la postcondición primitiva satisface la debilitada, de forma que la precondición inicial también satisface la postcondición final.

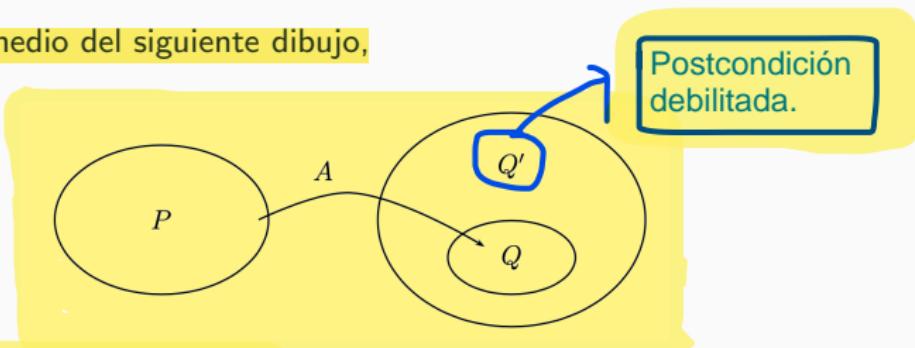
## Debilitamiento de la postcondición

Lo mismo que ocurría con la precondition ocurre con la postcondición.

Lo escribimos como regla de verificación de la siguiente manera:

$$\frac{\{P\} A \{Q\} \quad Q \Rightarrow Q'}{\{P\} A \{Q'\}}$$

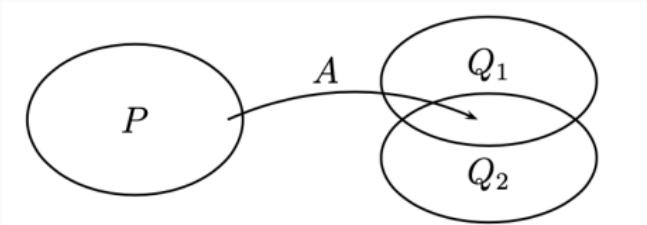
Y lo ilustramos por medio del siguiente dibujo,



Entender con la explicación anterior.

- Conjunción en la postcondición

$$\begin{array}{c} \{P\} A \{Q_1\} \quad \{P\} A \{Q_2\} \\ \hline \{P\} A \{Q_1 \wedge Q_2\} \end{array}$$



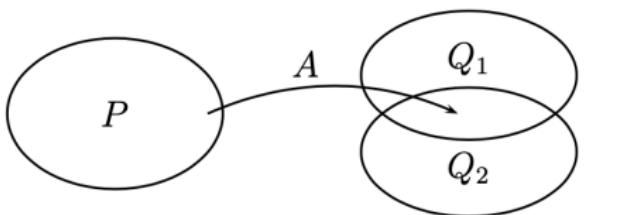
Si una precondición  $P$  satisface dos postcondiciones  $Q_1$  y  $Q_2$  entonces  $P$  satisface a la intersección de las postcondiciones

# Conjunción y disyunción

Mismo conjunto pero cambiado de orden = permutación.

- **Conjunción en la postcondición**

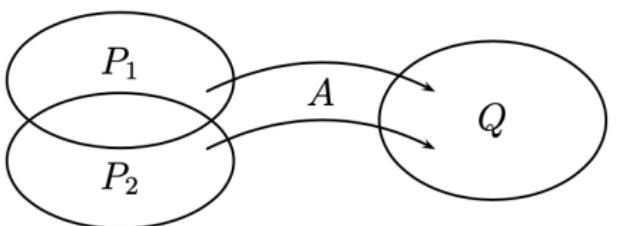
$$\frac{\{P\} A \{Q_1\} \quad \{P\} A \{Q_2\}}{\{P\} A \{Q_1 \wedge Q_2\}}$$



- **Disjunción en la precondición**

en distinciones de casos es muy común

$$\frac{\{P_1\} A \{Q\} \quad \{P_2\} A \{Q\}}{\{P_1 \vee P_2\} A \{Q\}}$$



verificar dos cosas por un lado

## Precondición más débil

- Dado un algoritmo  $A$  y una postcondición  $Q$ , queremos hallar la precondición  $P$  lo más débil posible, tal que se cumpla  $\{P\} A \{Q\}$ .
- La **precondición más débil** del algoritmo  $A$  con respecto a la postcondición  $Q$ ,  $pmd(A, Q)$ , se define como el predicado que cumple:
  - ①  $\{pmd(A, Q)\} A \{Q\}$ . pmd de  $A$  y  $Q$  que satisface  $Q$  para  $A$
  - ② Si  $P'$  cumple  $\{P'\} A \{Q\}$ , entonces  $P' \Rightarrow pmd(A, Q)$ .

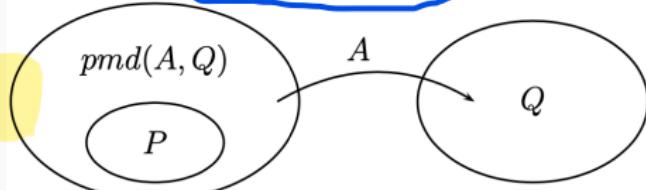
## Precondición más débil

Necesitamos reglas para cada instrucción: if, while, fors. Vamos a ver dos versiones.

- Dado un algoritmo  $A$  y una postcondición  $Q$ , queremos hallar la precondición  $P$  lo más débil posible, tal que se cumpla  $\{P\} A \{Q\}$ .
- La **precondición más débil** del algoritmo  $A$  con respecto a la postcondición  $Q$ ,  $pmd(A, Q)$ , se define como el predicado que cumple:
  - ①  $\{pmd(A, Q)\} A \{Q\}$ .
  - ② Si  $P'$  cumple  $\{P'\} A \{Q\}$ , entonces  $P' \Rightarrow pmd(A, Q)$ .
- Regla para utilizar la  $pmd$

Precondición más débil.

$$\frac{P \Rightarrow pmd(A, Q)}{\{P\} A \{Q\}}$$



Sobre todo los de los tests.

Partimos de la postcondición. Porque es lo que queremos que se cumpla. Muy importante acordarnos de esto para los ejercicios.

## **Verificación de algoritmos**

---

### **Verificación de instrucciones**

Precondición más débil de cada instrucción

Hallarlo a partir de una postcondición.

## Instrucción nada

El axioma que define la instrucción nada, cuyo efecto intuitivo es no realizar acción alguna, es el siguiente:

$$\{P\} \text{ nada } \{P\}$$

Los conjuntos de estados inicial y final son el mismo, lo que nos indica que la instrucción siempre termina y que su efecto sobre el estado del cómputo es nulo.

## Instrucción nada

El axioma que define la instrucción nada, cuyo efecto intuitivo es no realizar acción alguna, es el siguiente:

$$\{P\} \text{ nada } \{P\}$$

Los conjuntos de estados inicial y final son el mismo, lo que nos indica que la instrucción siempre termina y que su efecto sobre el estado del cómputo es nulo.

La precondition más débil para la instrucción nada se obtiene de la siguiente manera:

$$pmd(\text{nada}, Q) \Leftrightarrow Q.$$

Combinando esta regla con las reglas básicas, obtenemos esta regla de verificación:

$$\frac{P \Rightarrow Q}{\{P\} \text{ nada } \{Q\}}$$

## Instrucción nada

pmd: precondición más débil

El axioma que define la instrucción nada, cuyo efecto intuitivo es no realizar acción alguna, es el siguiente:

$$\{P\} \text{ nada } \{P\}$$

Los conjuntos de estados inicial y final son el mismo, lo que nos indica que la instrucción siempre termina y que su efecto sobre el estado del cómputo es nulo.

La precondición más débil para la instrucción nada se obtiene de la siguiente manera:

$$pmd(\text{nada}, Q) \Leftrightarrow Q.$$

Combinando esta regla con las reglas básicas, obtenemos esta regla de verificación:

$$\frac{P \Rightarrow Q}{\{P\} \text{ nada } \{Q\}}$$

Ejemplo: Para comprobar

$$\{x > 2\} \text{ nada } \{x > 0\}$$

hay que comprobar que  $x > 2 \Rightarrow x > 0$ , lo cual es cierto.

se cumple  $x > 2$  después de no hacer nada se cumple que  $x > 0$ ? Si.

## Predicado de definitud

Si una expresión está bien definida.

- Al admitir en las expresiones tipos de datos y funciones cualesquiera, dichas funciones pueden ser **parciales**, es decir, no estar definidas para ciertos valores de sus argumentos.  
No podemos acceder al elemento 888 de un vector de 200 elementos.
- Por ejemplo, no están definidas  $9 \text{ div } 0$ ,  $v[888]$  si  $v$  es un vector **int**  $v[200]$ , o  $15 \text{ mod } 0$ . **claro porque no se puede dividir entre 0**
- Utilizamos un **predicado de definición**  $\text{def}(e)$ , que devuelve **true** si  $e$  es una expresión definida y **false** en caso contrario.
- Por ejemplo,
  - $\text{def}(a \text{ mod } b) \Leftrightarrow b \neq 0$ ,
  - $\text{def}(a + b) \Leftrightarrow \text{true}$ , porque si  $a$  y  $b$  son iguales  $x/(a-b)$  es imposible
  - $\text{def}(x \text{ div } (a - b)) \Leftrightarrow a \neq b$ , porque no se puede dividir entre 0.
  - $\text{def}(x \text{ div } y + y \text{ div } x) \Leftrightarrow y \neq 0 \wedge x \neq 0$ .

Admitiremos que las operaciones son **estrictas**, es decir,

$$\neg \text{def}(e_i) \Rightarrow \neg \text{def}(f(e_1, \dots, e_n)).$$

## Instrucción de asignación

- La instrucción de asignación es  $x = e$ .
- El axioma para la asignación es el siguiente:

$$\{\text{def}(e) \wedge Q_x^e\}$$
$$x = e$$
$$\{Q\}$$

## Instrucción de asignación

- La instrucción de asignación es  $x = e$ .
- El axioma para la asignación es el siguiente:

$$\begin{array}{l} \{\text{def}(e) \wedge Q_x^e\} \\ x = e \\ \{Q\} \end{array}$$

- La precondition más débil para la instrucción de asignación se puede calcular como

$$pmd(x = e, Q) \Leftrightarrow \text{def}(e) \wedge Q_x^e$$

## Instrucción de asignación

- La instrucción de asignación es  $x = e$ .
- El axioma para la asignación es el siguiente:

$$\begin{array}{c} \{\text{def}(e) \wedge Q_x^e\} \\ x = e \\ \{Q\} \end{array}$$

- La precondition más débil para la instrucción de asignación se puede calcular como

$$pmd(x = e, Q) \Leftrightarrow \text{def}(e) \wedge Q_x^e$$

coge la Q y sustituye la expresión "x" por la expresión "e"

- Combinando con las reglas que conocemos para la  $pmd$ , obtenemos la siguiente regla de verificación:

$$\frac{P \Rightarrow \text{def}(e) \wedge Q_x^e}{\{P\} x = e \{Q\}}$$

## Ejemplos

- ¿Cuál es la precondition más débil para la siguiente especificación?

$$\{?\} x = 7 \{x > 0\}$$

Tenemos que hallar la pmd

$$(x > 0)_x^7 \Leftrightarrow 7 > 0 \Leftrightarrow \text{true}$$

Sustituimos lo de entre medias en la postcondición

7>0 se cumple siempre, luego la precondition más débil es true, puesto que se cumple siempre.

## Ejemplos

- ¿Cuál es la precondition más débil para la siguiente especificación?

$\{?\} x = 7 \{x > 0\}$   
pmd = true

$$(x > 0)_x^7 \Leftrightarrow 7 > 0 \Leftrightarrow \text{true}$$

- $\{?\} x = x + 1 \{x > 0\}$

pmd =  $x \geq 0$

$$(x > 0)_x^{x+1} \Leftrightarrow x + 1 > 0 \Leftrightarrow x > -1 \Leftrightarrow x \geq 0$$

sustituir ( $x$  por  $x+1$ ) $>0 \implies x+1>0 \implies x>-1 \implies x \geq 0$

$x > -1$

## Ejemplos

Vamos a verificar

$$\{x \geq 2 \wedge 2 * y > 5\} \boxed{x = 2*x + y - 1} \{x - 3 > 2\}$$

Utilizando la regla de la asignación, primero calculamos la precondición más débil de la asignación respecto de la postcondición dada.

$$\begin{aligned} (x - 3 > 2)_{x}^{2*x+y-1} &\Leftrightarrow \{ \text{sustitución} \} \\ y > 6 - 2x & \qquad \qquad \qquad (2*x + y - 1) - 3 > 2 \\ &\Leftrightarrow \{ \text{aritmética} \} \\ & \qquad \qquad \qquad 2*x + y > 6 \\ &\Leftarrow x \geq 2 \wedge 2*y > 5 \end{aligned}$$

Donde el último paso se puede dar ya que  $2*y > 5 \Rightarrow y > 2$  y

$$x \geq 2 \wedge y > 2 \Rightarrow 2*x + y > 6$$

## Asignación múltiple

Útil para intercambiar cosas.      aux =x; x= y; y= aux; ===== <x,y>; E<y,x>

- Modifica el valor de varias variables *simultáneamente*.

$$\langle x_1, \dots, x_n \rangle = \langle e_1, \dots, e_n \rangle$$

- El axioma para este tipo de asignación es:

$$\left\{ \bigwedge_{i=1}^n \text{def}(e_i) \wedge Q_{x_1, \dots, x_n}^{e_1, \dots, e_n} \right\} \langle x_1, \dots, x_n \rangle = \langle e_1, \dots, e_n \rangle \{ Q \}$$

- Y la precondición más débil

$$pmd(\langle x_1, \dots, x_n \rangle = \langle e_1, \dots, e_n \rangle, Q) \Leftrightarrow \bigwedge_{i=1}^n \text{def}(e_i) \wedge Q_{x_1, \dots, x_n}^{e_1, \dots, e_n}$$

- La regla de verificación es

$$\frac{P \Rightarrow \bigwedge_{i=1}^n \text{def}(e_i) \wedge Q_{x_1, \dots, x_n}^{e_1, \dots, e_n}}{\{P\} \langle x_1, \dots, x_n \rangle = \langle e_1, \dots, e_n \rangle \{Q\}}$$

## Ejemplo

Vamos a verificar la siguiente especificación

$$\{x = X \wedge y = Y\} \langle x, y \rangle = \langle y, x \rangle \{x = Y \wedge y = X\}$$

Calculamos la precondición más débil de la asignación múltiple respecto de la postcondición.

$$pmd(\langle x, y \rangle = \langle y, x \rangle, x = Y \wedge y = X) \Leftrightarrow (x = Y \wedge y = X)^{y,x}_{x,y}$$

$\Leftrightarrow \{\text{sustitución}\}$

$$y = Y \wedge x = X$$

$$\Leftrightarrow \{\text{lógica}\}$$

$$x = X \wedge y = Y$$

Hacerlo de golpe.

## Composición secuencial

- Composición secuencial de dos instrucciones  $A_1$  y  $A_2$

$$A_1 ; A_2$$

- La siguiente regla expresa formalmente la idea de secuenciación:

$$\frac{\{P\} A_1 \{R\} \quad \{R\} A_2 \{Q\}}{\{P\} A_1 ; A_2 \{Q\}}$$

## Composición secuencial

Para muchas instrucciones.

- Composición secuencial de dos instrucciones  $A_1$  y  $A_2$

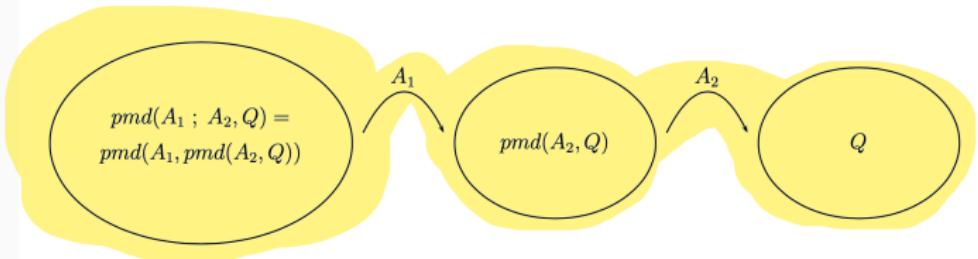
$$A_1 ; A_2$$

- La siguiente regla expresa formalmente la idea de secuenciación:

$$\frac{\{P\} A_1 \{R\} \quad \{R\} A_2 \{Q\}}{\{P\} A_1 ; A_2 \{Q\}}$$

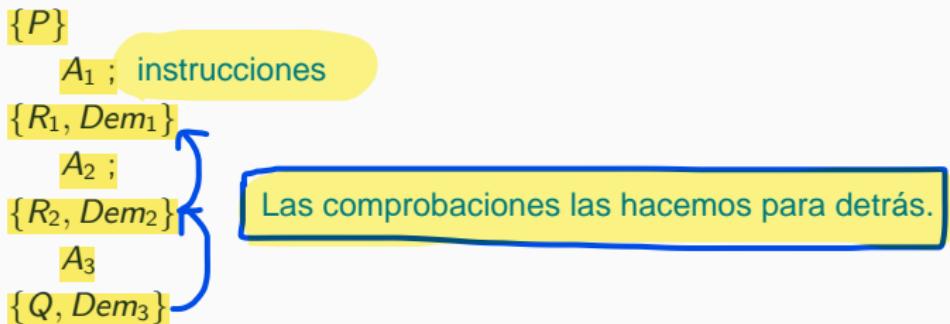
- En la mayoría de los casos vamos a tomar como  $R$  la  $pmd(A_2, Q)$ , para solo demostrar  $\{P\} A_1 \{R\}$  calculando a su vez la  $pmd(A_1, R)$  y viendo que  $P$  es más fuerte, es decir,  $P \Rightarrow pmd(A_1, R)$ .
- Regla para calcular la precondition más débil de una composición secuencial:

$$pmd(A_1 ; A_2, Q) \Leftrightarrow pmd(A_1, pmd(A_2, Q))$$



## Composición secuencial

Cuando tengamos más de dos instrucciones utilizaremos la misma idea, determinando aserciones intermedias y construyendo demostraciones para cada paso siguiendo el siguiente esquema de demostración (para el caso de tres instrucciones):



donde  $Dem_1$  demuestra  $\{P\}A_1\{R_1\}$ ,  $Dem_2$  demuestra  $\{R_1\}A_2\{R_2\}$ , y  $Dem_3$  demuestra  $\{R_2\}A_3\{Q\}$ .

Generalmente iremos hacia atrás, probando primero  $Dem_3$ , luego  $Dem_2$  y luego  $Dem_1$ .

## Horas, minutos y segundos

Diseñar y verificar un algoritmo que convierta una cantidad entera no negativa de segundos a horas, minutos y segundos.

La especificación es

$\{s = S \wedge s \geq 0\}$  te dan un número de segundos  $S$  mayor que 0

**proc** convertir(**inout int**  $s$ , **out int**  $m$ , **out int**  $h$ )

$\{S = 3600 * h + 60 * m + s \wedge 0 \leq s < 60 \wedge 0 \leq m < 60\}$

## Horas, minutos y segundos

Diseñar y verificar un algoritmo que convierta una cantidad entera no negativa de segundos a horas, minutos y segundos.

La especificación es

$$\{s = S \wedge s \geq 0\}$$

**proc convertir(inout int s, out int m, out int h)**

$$\{S = 3600 * h + 60 * m + s \wedge 0 \leq s < 60 \wedge 0 \leq m < 60\}$$

Tema 2

especificación.

Una hora tiene 3600 segundos, 1 minuto tiene 60 segundos. El número de segundos a de ser menor que 60 porque si no sería 1 minuto, y el número de minutos también tendría que ser menor que 60, porque si no sería 1 hora.

Y la implementación, anotada con aserciones es

$$\{P \equiv s = S \wedge s \geq 0\} \text{ demostramos que las horas son segundos/3600 y los nuevos segundos son el resto de la división entre 3600}$$
$$\langle h, s \rangle = \langle s \text{ div } 3600, s \text{ mod } 3600 \rangle ;$$

$$\{R \equiv S = 3600 * h + s \wedge 0 \leq s < 3600, Dem_1\}$$

$$\langle m, s \rangle = \langle s \text{ div } 60, s \text{ mod } 60 \rangle \text{ y los minutos lo mismo pero entre 60.}$$

$$\{Q \equiv S = 3600 * h + 60 * m + s \wedge 0 \leq s < 60 \wedge 0 \leq m < 60, Dem_2\}$$

## Horas, minutos y segundos

Como tenemos la condición intermedia, podemos probar  $Dem_1$ .

$$\begin{aligned} & R_{h,s}^{s \text{ div } 3600, s \text{ mod } 3600} \\ \Leftrightarrow & S = 3600 * (s \text{ div } 3600) + (s \text{ mod } 3600) \wedge 0 \leq s \text{ mod } 3600 < 3600 \\ \Leftrightarrow & \{ \text{propiedades de div y mod} \} \\ & S = s \\ \Leftarrow & P \end{aligned}$$

## Horas, minutos y segundos

Como tenemos la condición intermedia, podemos probar  $Dem_1$ .

$$\begin{aligned} R_{h,s}^{s \text{ div } 3600, s \text{ mod } 3600} &\Leftrightarrow S = 3600 * (s \text{ div } 3600) + (s \text{ mod } 3600) \wedge 0 \leq s \text{ mod } 3600 < 3600 \\ &\Leftrightarrow \{ \text{propiedades de div y mod} \} \\ S &= s \\ &\Leftarrow P \end{aligned}$$

Y  $Dem_2$  consiste en

$$\begin{aligned} Q_{m,s}^{s \text{ div } 60, s \text{ mod } 60} &\Leftrightarrow S = 3600 * h + 60 * (s \text{ div } 60) + (s \text{ mod } 60) \wedge \\ &\quad 0 \leq s \text{ mod } 60 < 60 \wedge 0 \leq s \text{ div } 60 < 60 \\ &\Leftrightarrow S = 3600 * h + s \wedge 0 \leq s \text{ div } 60 < 60 \\ &\Leftarrow R \end{aligned}$$

LEER Y PREGUNTAR DUDAS.

NO ENTIENDO LA DEMOSTRACIÓN DE ESTE EJEMPLO

## Intercambio de variables

Algoritmo que intercambia el valor de dos variables:

$$\begin{aligned} \{P \equiv x = X \wedge y = Y\} \\ z = x ; \quad x = y ; \quad y = z \quad z=\text{aux}; \text{aux}=y; y=z; \\ \{Q \equiv x = Y \wedge y = X\} \end{aligned}$$

Demostración:

$$\begin{aligned} \{P\} \\ z = x ; \\ \{R_1, Dem_1\} \\ x = y ; \\ \{R_2, Dem_2\} \\ y = z ; \\ \{Q, Dem_3\} \end{aligned}$$

Empezamos desde la postcondición siempre porque es lo que queremos demostrar

Y ahora vamos haciendo las pequeñas demostraciones de abajo arriba.

LEER Y PREGUNTAR SI TENEMOS DUDAS. HACERLO CON LA PRECONDICIÓN MÁS DÉBIL

## Intercambio de variables

- $Dem_3: \{R_2\} [y = z] \{Q\}$

$$\begin{aligned} R_2 \equiv pmd(y = z, Q) &\Leftrightarrow Q_y^z \\ &\Leftrightarrow x = Y \wedge z = X \end{aligned}$$

Sustituimos en la y por la z. Ya que  $y=z$ . En la POSTCONDICIÓN

Esta será la postcondición de la siguiente expresión

## Intercambio de variables

- $Dem_3: \{R_2\} y = z \{Q\}$

$$\begin{aligned} R_2 \equiv pmd(y = z, Q) &\Leftrightarrow Q_y^z \\ &\Leftrightarrow x = Y \wedge z = X \end{aligned}$$

- $Dem_2: \{R_1\} x = y \{R_2\}$

Es la postcondición de la anterior

$$\begin{aligned} R_1 \equiv pmd(x = y, R_2) &\Leftrightarrow (R_2)_x^y \\ &\Leftrightarrow y = Y \wedge z = X \end{aligned}$$

ahora en la siguiente donde hay una z pondrá x x= X

## Intercambio de variables

- $Dem_3: \{R_2\} y = z \{Q\}$

$$R_2 \equiv pmd(y = z, Q)$$

 $\Leftrightarrow$ 

$$Q_y^z$$

 $\Leftrightarrow$ 

$$x = Y \wedge z = X$$

Variable auxiliar para intercambiar valores.

- $Dem_2: \{R_1\} x = y \{R_2\}$

$$R_1 \equiv pmd(x = y, R_2) \Leftrightarrow (R_2)_x^y$$

 $\Leftrightarrow$ 

$$y = Y \wedge z = X$$

- $Dem_1: \{P\} z = x \{R_1\}$

$$pmd(z = x, R_1) \Leftrightarrow (R_1)_z^x$$

 $\Leftrightarrow$ 

$$y = Y \wedge x = X$$

 $\Leftarrow$ 

$$P$$

Este ejemplo si lo he entendido. Primero lo demostramos para la postcondición. z aquí sería como la variable auxiliar.

IDEM.

## Composición alternativa



La distinción de casos if/else

- La distinción de casos o composición alternativa

`if B A1 else A2`

B es la condición del if, si se cumple B si no negación de B

- La regla de verificación para este caso es:

$$\frac{P \Rightarrow \text{def}(B) \\ \{P \wedge B\} A_1 \{Q\} \\ \{P \wedge \neg B\} A_2 \{Q\}}{\{P\} \text{ if } B A_1 \text{ else } A_2 \{Q\}}$$

## Composición alternativa

- La distinción de casos o composición alternativa

`if B A1 else A2`

Esta es la condición del if/else.

- La regla de verificación para este caso es:

$$\frac{\begin{array}{c} P \Rightarrow \text{def}(B) \\ \{P \wedge B\} A_1 \{Q\} \\ \{P \wedge \neg B\} A_2 \{Q\} \end{array}}{\{P\} \text{ if } B A_1 \text{ else } A_2 \{Q\}}$$

- Y la precondition más débil

$pmd(\text{if } B A_1 \text{ else } A_2, Q)$

$$\Leftrightarrow \text{def}(B) \wedge ((B \wedge pmd(A_1, Q)) \vee (\neg B \wedge pmd(A_2, Q)))$$

## Composición alternativa

A la hora de demostrar una distinción de casos utilizaremos el siguiente esquema:

```
{P}
  if B
    {P ∧ B} A1 {Q, Dem1}
  else
    {P ∧ ¬B} A2 {Q, Dem2}
{Q}
```

## Ejemplo 1

```
{true}
if (x >= 0)
    { x = x + 1; }
else
    { x = x - 1; }
{x ≠ 0}
```

## Ejemplo 1

```
{true}
  if (x >= 0)
    { $x \geq 0$ } { x = x + 1; } { $x \neq 0, Dem_1$ }
  else
    { $x < 0$ } { x = x - 1; } { $x \neq 0, Dem_2$ }
{x \neq 0}
```

sustituimos la x de Q por x-1 y tenemos

## Ejemplo 1

```
{true} P
  if (x >= 0) B
    {x ≥ 0} {x = x + 1;} {x ≠ 0, Dem1}
  else A1
    {x < 0} {x = x - 1;} {x ≠ 0, Dem2}
{x ≠ 0} Q
```

A2  
pmd(A1, Q) x+1 != 0 => x != -1 => x >= 0 bien  
pmd(A2, Q) x-1 != 0 => x != 1 => x < 0 bien

ambos están bien definidos.

- $Dem_1$

$$pmd(x = x + 1, x \neq 0) \Leftrightarrow x \neq -1 \Leftrightarrow x \geq 0$$

- $Dem_2$

$$pmd(x = x - 1, x \neq 0) \Leftrightarrow x \neq 1 \Leftrightarrow x < 0$$

## Ejemplo 2

```
{true} P  
 $\langle x, y \rangle = \langle y * y, x * x \rangle;$   
if (x >= y) B  
{ x = x - y; } A1  
else (x < y)  
{ y = y - x; } A2  
 $\{x \geq 0 \wedge y \geq 0\}$   
Q
```

Escribimos el algoritmo con condiciones intermedias anotadas:

```
{P ≡ true}  
 $\langle x, y \rangle = \langle y * y, x * x \rangle;$   
{R}  
if (x >= y)  
{R1} x = x - y; {Q}  
else  
{R2} y = y - x; {Q}  
 $\{Q \equiv x \geq 0 \wedge y \geq 0\}$  Postcondicion, se tiene que cumplir en ambas ramas.
```

## Ejemplo 2

Primero calculamos  $R_1$  y  $R_2$ :

**R = rama.**

$$\begin{aligned} R_1 &\equiv \text{pmd}(x = x - y, Q) \\ &\Leftrightarrow (x \geq 0 \wedge y \geq 0)_x^{x-y} \Leftrightarrow x - y \geq 0 \wedge y \geq 0 \end{aligned}$$

$$\begin{aligned} R_2 &\equiv \text{pmd}(y = y - x, Q) \\ &\Leftrightarrow (x \geq 0 \wedge y \geq 0)_y^{y-x} \Leftrightarrow x \geq 0 \wedge y - x \geq 0 \end{aligned}$$

y escribimos  $R$  como

$$\begin{aligned} R &\equiv (x \geq y \wedge R_1) \vee (x < y \wedge R_2) \\ &\Leftrightarrow (x \geq y \wedge y \geq 0) \vee (x < y \wedge x \geq 0) \end{aligned}$$

## Ejemplo 2

Primero calculamos  $R_1$  y  $R_2$ :

$$\begin{aligned} R_1 &\equiv \text{pmd}(x = x - y, Q) \\ &\Leftrightarrow (x \geq 0 \wedge y \geq 0)_x^{x-y} \Leftrightarrow x - y \geq 0 \wedge y \geq 0 \end{aligned}$$

$$\begin{aligned} R_2 &\equiv \text{pmd}(y = y - x, Q) \\ &\Leftrightarrow (x \geq 0 \wedge y \geq 0)_y^{y-x} \Leftrightarrow x \geq 0 \wedge y - x \geq 0 \end{aligned}$$

y escribimos  $R$  como

$$\begin{aligned} R &\equiv (x \geq y \wedge R_1) \vee (x < y \wedge R_2) \\ &\Leftrightarrow (x \geq y \wedge y \geq 0) \vee (x < y \wedge x \geq 0) \end{aligned}$$

Ahora podemos calcular la precondition más débil de la asignación múltiple (primera instrucción) respecto de  $R$ :

$$\begin{aligned} \text{pmd}(\langle x, y \rangle = \langle y * y, x * x \rangle, R) &\Leftrightarrow R_{x,y}^{y*y,x*x} \\ &\Leftrightarrow (y * y \geq x * x \wedge x * x \geq 0) \vee (y * y < x * x \wedge y * y \geq 0) \\ &\Leftrightarrow y * y \geq x * x \vee y * y < x * x \quad \text{es trivial.} \\ &\Leftrightarrow \mathbf{true} \end{aligned}$$

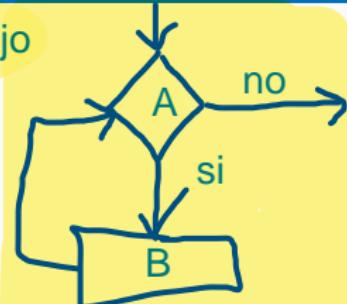
## Instrucción iterativa

A partir de aquí viene lo más importante de este tema que es el invariante y demás

- La **instrucción de iteración** tiene sintaxis

while *B*  
    *A*

d. flujo



- Para la verificación de un bucle necesitamos un predicado llamado **invariante** que describe los distintos estados por los que pasa el bucle dando la relación existente entre las variables que intervienen en este:

{Pre. *P*}  
while *B*

{Inv. *I*}

*A*  
{Post. *Q*}

En el examen nos pide el invariante del bucle y la función de cota.

cuerpo(secuencia de instrucciones). Se vuelve a evaluar *B*.

NO escribir breaks dentro del bucle para evitar problemas de legibilidad.

Vamos a tener que comprobar 3 propiedades

## Instrucción iterativa

TODO esto es la corrección parcial del bucle

Estás expresiones son importantes.

El invariante  $I$  tiene que cumplir las siguientes condiciones:

(i.1) Se satisface antes de empezar el bucle, es decir, antes de la primera iteración:

**PRECONDICIÓN ES MÁS FUERTE QUE INVARIANTE**

$$P \Rightarrow I.$$

Precondición satisface al invariante

(i.2) Se mantiene al ejecutar el cuerpo A del bucle:

Después de ejecutarse el cuerpo del bucle (A) también se cumple el invariante.

$$\{I \wedge B\} A \{I\}.$$

(i.3) Se cumple al salir del bucle, cuando  $B$  se hace falsa:

$$I \wedge \neg B \Rightarrow Q.$$

Cuando la condición ya no se cumple, esto nos tiene que llevar a la postcondición == Q. Sólo se cumple si el bucle termina.

Es decir el invariante se mantiene tanto antes de entrar en el bucle, como durante el bucle, como al final del bucle

## Instrucción iterativa

Se tiene que cumplir esto para que se cumpla de manera total.

Además tendremos que probar que la instrucción iterativa **termina**. Para lo cual

tendremos que buscar una **función C** que dependa de las variables del bucle y que tome valores enteros, de tal forma que se cumpla que:

**La idea es dar funciones de cota**

(c.1) Es mayor que cero cuando se cumple la condición B

$$I \wedge B \Rightarrow C \geq 0.$$

Tiene que ser positiva  
depende de las variables  
del programa == libres

(c.2) Decrece al ejecutar el cuerpo A del bucle:

$$\{I \wedge B \wedge C = T\} A \{C < T\}.$$

Esta función es una cota superior del número de iteraciones que quedan por realizar y por eso nos referiremos a ella como **función de cota**.

Aprendernos las funciones de cota.

Es importante, debemos de dar los costes de nuestros algoritmos, hacer su especificación y hallar el invariante y la función de cota. TODO suma

## Instrucción iterativa

Además tendremos que probar que la instrucción iterativa **termina**. Para lo cual tendremos que buscar una función  $C$  que dependa de las variables del bucle y que tome valores enteros, de tal forma que se cumpla que:

(c.1) Es mayor que cero cuando se cumple la condición  $B$ :

$$I \wedge B \Rightarrow C \geq 0.$$

(c.2) Decrece al ejecutar el cuerpo  $A$  del bucle:

$$\{I \wedge B \wedge C = T\} A \{C < T\}.$$

Esta función es una cota superior del número de iteraciones que quedan por realizar y por eso nos referiremos a ella como **función de cota**.

La regla de verificación para la instrucción iterativa es:

el invariante se satisface antes de entrar a la función  
el invariante se mantiene mientras estamos dentro del bucle  
el invariante se cumple al salir del bucle

$$\frac{\begin{array}{l} i1) P \Rightarrow I \\ i2) \{I \wedge B\} A \{I\} \\ i3) I \wedge \neg B \Rightarrow Q \\ c1) I \wedge B \Rightarrow C \geq 0 \\ c2) \{I \wedge B \wedge C = T\} A \{C < T\} \end{array}}{\{P\} \text{ while } B A \{Q\}}$$

En el examen NO nos va a pedir todos esos pasos. Nos pedirá la función de cota y el invariante.

VAMOS A VER EJEMPLOS DE ESTO DEL INVARIANTE DE LA FUNCIÓN DE COTA, SUMADO SI NO ME EQUIVOCO A LA ESPECIFICACIÓN Y ANALISIS DE COSTE DE LOS TEMAS ANTERIORES.

## Verificación de algoritmos

### Ejemplos de verificación

Muy importante entender todos los ejemplos que veamos aquí porque los ejercicios nos piden todo esto.

## Multiplicación mediante suma

En este ejemplo queremos realizar la multiplicación en forma de suma. Es decir Si hacemos 5 por ocho se trata de sumar 8 veces 5.

Verificamos el siguiente algoritmo de multiplicación:

$$\{P \equiv x = X \wedge y = Y \wedge y \geq 0\}$$

p = 0;

```
while (y != 0) {  
    {Inv. ?; Cota ?}  
    p = p + x;  
    y = y - 1; A  
}
```

$$\{Q \equiv p = X * Y\}$$

Coste del algoritmo es del orden de y. Y todas las condiciones del bucle son constantes.

Sumamos x y veces.

Coste:  $\Theta(Y)$

Entrará las Y veces hasta que y sea 0.

Para hacer la verificación primero tenemos que encontrar el invariante y la función de cota.

no puede ser un booleano.

La cota del bucle es: y (se mantiene positiva). Luego se cumple c1. Y decrece al ejecutar el cuerpo del bucle, se cumple c2.

El invariante tenemos que decir:  $I ::= y \geq 0 \wedge (y)$  vamos a ver las siguientes diapos

## Multiplicación mediante suma

Vamos a ejecutar varias veces el cuerpo del bucle para descubrir cuál es la relación que se mantiene invariante entre las variables:

estado	x	y	p
$\sigma_0$	X	Y	0



## Multiplicación mediante suma

Vamos a ejecutar varias veces el cuerpo del bucle para descubrir cuál es la relación que se mantiene invariante entre las variables:

estado	x	y	p
$\sigma_0$	X	Y	0
$\sigma_1$	X	$Y - 1$	X

## Multiplicación mediante suma

Vamos a ejecutar varias veces el cuerpo del bucle para descubrir cuál es la relación que se mantiene invariante entre las variables:

estado	x	y	p
$\sigma_0$	X	Y	0
$\sigma_1$	X	$Y - 1$	X
$\sigma_2$	X	$Y - 2$	$2 * X$

## Multiplicación mediante suma

Vamos a ejecutar varias veces el cuerpo del bucle para descubrir cuál es la relación que se mantiene invariante entre las variables:

estado	x	y	p
$\sigma_0$	X	Y	0
$\sigma_1$	X	Y - 1	X
$\sigma_2$	X	Y - 2	2 * X
:			
$\sigma_i$	X	Y - i	$i * X$

## Multiplicación mediante suma

Vamos a ejecutar varias veces el cuerpo del bucle para descubrir cuál es la relación que se mantiene invariante entre las variables:

La  $x$  no varia (invariante).

La  $y$  si cambia y todo el tiempo está entre 0 y su valor inicial (prop invariante)

estado	$x$	$y$	$p$
$\sigma_0$	$X$	$Y$	0
$\sigma_1$	$X$	$Y - 1$	$X$
$\sigma_2$	$X$	$Y - 2$	$2 * X$
$\vdots$			
$\sigma_i$	$X$	$Y - i$	$i * X$

$$X * Y = p + y * x$$

$$\begin{aligned} p &= X * Y - y * X \\ &= X(Y - y) \end{aligned}$$

Vemos que la variable  $x$  no se modifica, que  $y$  lleva cuenta de las veces que nos falta sumar  $x$  al resultado para obtener  $X * Y$ , y que  $p$  guarda el resultado parcial, es decir, cuántas veces ya hemos sumado  $x$ . Son invariantes:

$$x = X$$

$$0 \leq y \leq Y$$

true

$$\begin{aligned} \exists k : k \in \text{nat} : p &= k * X \\ X * Y &= p + x * y \end{aligned}$$

## Multiplicación mediante suma

Vamos a ejecutar varias veces el cuerpo del bucle para descubrir cuál es la relación que se mantiene invariante entre las variables:

estado	x	y	p
$\sigma_0$	X	Y	0
$\sigma_1$	X	Y - 1	X
$\sigma_2$	X	Y - 2	2 * X
:			
$\sigma_i$	X	Y - i	$i * X$

Proponemos como invariante  $I \equiv x = X \wedge (X * Y = p + x * y) \wedge y \geq 0$

La  $x = X$  se mantiene todo el rato, y la  $y$  siempre va a ser  $\geq 0$ . Luego eso lo tenemos fijo.  $p$  empieza siendo 0 y sabemos que  $p = X * Y$  luego  $X * Y = p + x * y$

## Multiplicación mediante suma

antes de entrar al bucle p=0

- (i.1) En este caso, como tenemos la asignación  $p = 0$  entre la precondition que nos dan y el invariante, sustituimos en la regla de verificación la parte  $P \Rightarrow I$  por otra adecuada a esta situación, que es

$$\{P\} \ p = 0 \ \{I\}$$

Por la regla de la asignación:

$$\begin{aligned} pmd(p = 0, I) &\Leftrightarrow (x = X \wedge X * Y = p + x * y \wedge y \geq 0)_p^0 \\ &\Leftrightarrow x = X \wedge X * Y = x * y \wedge y \geq 0 \\ &\Leftrightarrow x = X \wedge y = Y \wedge y \geq 0 \\ &\Leftrightarrow P \end{aligned}$$

## Multiplicación mediante suma

- (i.1) En este caso, como tenemos la asignación  $p = 0$  entre la precondition que nos dan y el invariante, sustituimos en la regla de verificación la parte  $P \Rightarrow I$  por otra adecuada a esta situación, que es

$$\{P\} p = 0 \{I\}$$

Por la regla de la asignación:

$$\begin{aligned} pmd(p = 0, I) &\Leftrightarrow (x = X \wedge X * Y = p + x * y \wedge y \geq 0)_p^0 \\ &\Leftrightarrow x = X \wedge X * Y = x * y \wedge y \geq 0 \\ &\Leftarrow x = X \wedge y = Y \wedge y \geq 0 \\ &\Leftrightarrow P \end{aligned}$$

Este paso de aquí consiste en demostrar que el invariante se cumple antes de B siendo B el bucle.

- (i.2) En este punto tenemos que verificar

$\{I \wedge y \neq 0\}$       mirar si precondition es más fuerte que la precondition más débil  
 $p = p + x;$   
 $y = y - 1;$   
 $\{I\}$

## Multiplicación mediante suma

TODO LO QUE VAMOS A VER EN LAS PRÓXIMAS DIAPOS SON DEMOSTRACIONES.

Por la regla de la asignación y de la composición secuencial:

$$pmd(p = p + x; y = y - 1; I)$$

sust y por y-1 e p=p+x

$$\Leftrightarrow (x = X \wedge X * Y = p + x * y \wedge y \geq 0)_{y,p}^{y-1, p+x}$$

$$\Leftrightarrow x = X \wedge X * Y = (p + x) + x * (y - 1) \wedge y - 1 \geq 0$$

$$\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y - 1 \geq 0$$

$$\Leftarrow I \wedge y \neq 0$$

$$\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y > 0$$

## Multiplicación mediante suma

Por la regla de la asignación y de la composición secuencial:

$$pmd(p = p + y; y = y - 1;, I)$$

$$\Leftrightarrow (x = X \wedge X * Y = p + x * y \wedge y \geq 0)^{y-1, p+x}_{y,p}$$

$$\Leftrightarrow x = X \wedge X * Y = (p + x) + x * (y - 1) \wedge y - 1 \geq 0$$

$$\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y - 1 \geq 0$$

$$\Leftarrow I \wedge y \neq 0$$

$$\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y > 0$$

(i.3) Suponiendo que se termina (luego se demostrará), hemos de verificar  $I \wedge \neg B \Rightarrow Q$ .

$$I \wedge \neg B \Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y \geq 0 \wedge y = 0 \rightarrow B$$

$$\Leftrightarrow x = X \wedge X * Y = p + x * 0 \wedge y = 0$$

$$\Leftrightarrow x = X \wedge X * Y = p \wedge y = 0$$

$$\Rightarrow Q$$

$$\Leftrightarrow X * Y = p$$

Para terminar la verificación tenemos que **demostrar que el bucle termina**. Para ello tenemos que **buscar una función de cota** y probar los dos puntos que nos faltan de la regla de verificación.

Las demostraciones no se suelen pedir explícitamente en los ejercicios, pero es muy importante entenderlas porque es lo que luego tenemos que pensar haciendo los ejjs.

## Multiplicación mediante suma

En este caso podemos tomar como cota la variable  $y$ :  $C = y$ .

Ahora probamos los dos puntos:

- (c.1) La cota es positiva cuando entramos en el bucle:  $I \wedge B \Rightarrow y \geq 0$ . Lo cual es cierto ya que:

$$\begin{aligned} I \wedge B &\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y \geq 0 \wedge y \neq 0 \\ &\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y > 0 \\ &\Rightarrow y \geq 0 \end{aligned}$$

## Multiplicación mediante suma

En este caso podemos tomar como cota la variable  $y$ :  $C = y$ .

Ahora probamos los dos puntos:

- (c.1) La cota es positiva cuando entramos en el bucle:  $I \wedge B \Rightarrow y \geq 0$ . Lo cual es cierto ya que:

$$\begin{aligned} I \wedge B &\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y \geq 0 \wedge y \neq 0 \\ &\Leftrightarrow x = X \wedge X * Y = p + x * y \wedge y > 0 \\ &\Rightarrow y \geq 0 \end{aligned}$$

- (c.2) La cota decrece al pasar por el cuerpo del bucle. Tenemos que verificar:

$$\begin{aligned} &\{I \wedge B \wedge y = T\} \\ &\quad p = p + y; \quad y = y - 1; \\ &\{y < T\} \end{aligned}$$

Por la regla de la asignación y la composición secuencial:

$$\begin{aligned} pmd(p = p + y; \quad y = y - 1; \quad y < T) &\Leftrightarrow (y < T)_{y,p}^{y-1,p+x} \\ &\Leftrightarrow y - 1 < T \\ &\Leftarrow y = T \\ &\Leftarrow I \wedge B \wedge y = T \end{aligned}$$

Y de este modo ya hemos verificado todo el algoritmo.

Lo difícil de los ejercicios es decir el invariante

## Cuadrado mediante suma

Ejemplo MUY parecido al anterior. Leerlo en casa atento

```
{P ≡ N ≥ 0}  q=0; p =1  
i = 0; q = 0; p = 1;  
while (i < N) B {  
    {Inv. ?; Cota ?}  
    i = i + 1;  
    q = q + p; A  
    p = p + 2;  
}  
{Q ≡ q = N2} Q
```

El coste del algoritmo es del orden exacto de N  
y la cota es N-i.

Coste:  $\Theta(N)$  El coste del bucle y la función de cota del bucle, son cosas totalmente distintas.

Pero al final pueden llegar a ser muy parecidos.

$$\begin{aligned}q_1 &= q_0 + p + 2 \\q_2 &= q_1 + p + 2 = q_0 + 2p + 4 \\q_3 &= q_2 + p + 2 = q_0 + 3p + 6;\end{aligned}$$

## Cuadrado mediante suma

$$\{P \equiv N \geq 0\}$$

```
i = 0; q = 0; p = 1;
```

```
while (i < N) {
```

```
{Inv. ?; Cota ?}
```

```
    i = i + 1;
```

```
    q = q + p;
```

```
    p = p + 2;
```

```
}
```

$$\{Q \equiv q = N^2\}$$

Coste:  $\Theta(N)$

$i$	$q$	$p$
0	0	1

## Cuadrado mediante suma

$$\{P \equiv N \geq 0\}$$

```
i = 0; q = 0; p = 1;
```

```
while (i < N) {
```

```
{Inv. ?; Cota ?}
```

```
    i = i + 1;
```

```
    q = q + p;
```

```
    p = p + 2;
```

```
}
```

$$\{Q \equiv q = N^2\}$$

Coste:  $\Theta(N)$

$i$	$q$	$p$
0	0	1
1	1	3

## Cuadrado mediante suma

$\{P \equiv N \geq 0\}$

i = 0; q = 0; p = 1;

**while** (i < N) {

{Inv. ?; Cota ?}

i = i + 1;

q = q + p;

p = p + 2;

}

$\{Q \equiv q = N^2\}$

Coste:  $\Theta(N)$

i	q	p
0	0	1
1	1	3
2	4	5

## Cuadrado mediante suma

$\{P \equiv N \geq 0\}$

i = 0; q = 0; p = 1;

**while** (i < N) {

{*Inv.* ?; *Cota* ?}

i = i + 1;

q = q + p;

p = p + 2;

}

$\{Q \equiv q = N^2\}$

Coste:  $\Theta(N)$

$i$	$q$	$p$
0	0	1
1	1	3
2	4	5
3	9	7

## Cuadrado mediante suma

$\{P \equiv N \geq 0\}$

i = 0; q = 0; p = 1;

**while** (i < N) {

{*Inv.* ?; *Cota* ?}

i = i + 1;

q = q + p;

p = p + 2;

}

$\{Q \equiv q = N^2\}$

Coste:  $\Theta(N)$

$i$	$q$	$p$
0	0	1
1	1	3
2	4	5
3	9	7
4	16	9

## Cuadrado mediante suma

$\{P \equiv N \geq 0\}$

```
i = 0; q = 0; p = 1;
```

```
while (i < N) {
```

```
{Inv. ?; Cota ?}
```

```
i = i + 1;
```

```
q = q + p;
```

```
p = p + 2;
```

```
}
```

$\{Q \equiv q = N^2\}$

Coste:  $\Theta(N)$

$i$	$q$	$p$
0	0	1
1	1	3
2	4	5
3	9	7
4	16	9

$$I \equiv q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N$$

## Cuadrado mediante suma

(i.1)  $\{P\} \ i = 0; \ q = 0; \ p = 1; \ \{I\}$

$$\begin{aligned} & (((q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_p^1)_q^0)_i^0 \\ \Leftrightarrow & ((q = i^2 \wedge 1 = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_q^0)_i^0 \\ \Leftrightarrow & (0 = i^2 \wedge 1 = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_i^0 \\ \Leftrightarrow & 0 = 0^2 \wedge 1 = 2 \cdot 0 + 1 \wedge 0 \leq 0 \leq N \\ \Leftrightarrow & 0 \leq N \end{aligned}$$

## Cuadrado mediante suma

(i.1)  $\{P\} \ i = 0; \ q = 0; \ p = 1; \ \{I\}$

Las demostraciones

no las tendremos que

realizar explícitamente

en el examen pero nos sirve

para saber si nuestras de-

mostraciones son co-

rrectas.

$$(((q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_p^1)_q^0)_i^0$$

$$\Leftrightarrow (((q = i^2 \wedge 1 = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_q^0)_i^0$$

$$\Leftrightarrow (0 = i^2 \wedge 1 = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_i^0$$

$$\Leftrightarrow 0 = 0^2 \wedge 1 = 2 \cdot 0 + 1 \wedge 0 \leq 0 \leq N$$

$$\Leftrightarrow 0 \leq N$$

B

(i.2)  $\{I \wedge (i < N)\} \ i = i + 1; \ q = q + p; \ p = p + 2; \ \{I\}$

$$(((q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_p^{p+2})_q^{q+p})_i^{i+1}$$

$$\Leftrightarrow ((q = i^2 \wedge p + 2 = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_q^{q+p})_i^{i+1}$$

$$\Leftrightarrow (q + p = i^2 \wedge p + 2 = 2 \cdot i + 1 \wedge 0 \leq i \leq N)_i^{i+1}$$

$$\Leftrightarrow q + p = (i + 1)^2 \wedge p + 2 = 2 \cdot (i + 1) + 1 \wedge 0 \leq (i + 1) \leq N$$

$$\Leftrightarrow q + p = i^2 + 1 + 2 \cdot i \wedge p = 2 \cdot i + 1 \wedge 0 \leq (i + 1) \leq N$$

$$\Leftrightarrow q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i < N$$

$p = 2i + 1$

## Cuadrado mediante suma

(i.3)

$$q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N \wedge i \geq N \Rightarrow q = N^2$$

## Cuadrado mediante suma

(i.3)

$$q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N \wedge i \geq N \Rightarrow q = N^2$$

(c.1) Tomamos como cota  $N - i$

$$q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N \wedge i < N \Rightarrow N - i \geq 0$$

## Cuadrado mediante suma

(i.3)

$$q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N \wedge i \geq N \Rightarrow q = N^2$$

(c.1) Tomamos como cota  $N - i$

$$q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i \leq N \wedge i < N \Rightarrow N - i \geq 0$$

(c.2)  $\{I \wedge i < N \wedge N - i = T\} i = i + 1; q = q + p; p = p + 2 \{N - i < T\}$

$$(((N - i < T)_p^{p+2})_q^{q+p})_i^{i+1}$$

$$\Leftrightarrow N - (i + 1) < T$$

$$\Leftrightarrow N - i - 1 < T$$

$$\Leftarrow q = i^2 \wedge p = 2 \cdot i + 1 \wedge 0 \leq i < N \wedge N - i = T$$

## Suma de un vector

Ejercicio muy sencillo según ella.

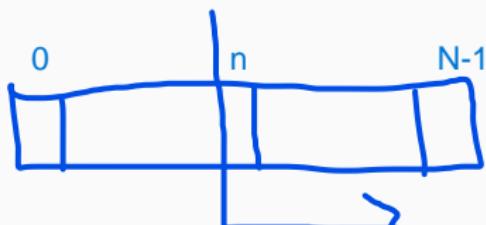
$$\{P \equiv 0 \leq N \leq \text{longitud}(v)\}$$

```
//fun  suma(int v[ ], int N) dev int x
int suma(v[ ], int N) {
    int n = N; x = 0;
    while (n != 0) B {
        x = x + v[n-1];
        n = n - 1;
    }
    return x;
}
```

$$\{Q \equiv x = (\sum i : 0 \leq i < N : v[i])\}$$

$$S = \sum_{z=n}^{N-1} v[z]$$

cuando  $n=0$  Tenemos la suma de todos. Se cumple.



1º rango

utilizando como invariante  $I \equiv 0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : v[i]).$

Coste:  $\Theta(N)$

La función de cota  $t=n$ ; (esto es lo más fácil de calcular)  
Muchas veces el invariante es parecido a la postcondición.

final  $n=0$ .

### PASO 1, COMPROBAR SE CUMPLE AL PRINCIPIO

]

- (i.1) El invariante se cumple justo antes de comenzar el bucle.

$$\{P\} \ n = N; \ x = 0; \ \{I\}$$

Para ello utilizamos las reglas de verificación de la composición secuencial y de la asignación.

$$pmd(n = N; \ x = 0;, I)$$

$$\Leftrightarrow (I_x^0)_n^N$$

$$\Leftrightarrow 0 \leq N \leq N \wedge 0 = (\sum i : N \leq i < N : v[i])$$

$$\Leftarrow P \quad \text{Esto sí se cumple.}$$

## Suma de un vector

]

### COMPROBAR INVARIANTE SE CUMPLE EN EL CUERPO DEL BUCLE

- (i.2) El invariante se mantiene al ejecutar el cuerpo del bucle.

$$\{I \wedge n \neq 0\} \quad x = x + v[n-1]; \quad n = n - 1; \quad \{I\}$$

Utilizando las reglas de verificación de la composición secuencial y de la asignación, tenemos

$$\begin{aligned} & pmd(x = x + v[n-1]; \quad n = n - 1; \quad I) \\ \Leftrightarrow & (I_n^{n-1})_x^{x+v[n-1]} \\ \Leftrightarrow & 0 \leq n - 1 \leq N \wedge x + v[n - 1] = (\sum i : n - 1 \leq i < N : v[i]) \\ \Leftrightarrow & 0 \leq n - 1 \leq N \wedge x + v[n - 1] = v[n - 1] + (\sum i : n \leq i < N : v[i]) \\ \Leftarrow & I \wedge n \neq 0 \end{aligned}$$

donde el último paso es correcto por las siguientes implicaciones:

$$0 \leq n \leq N \wedge n \neq 0 \Rightarrow 0 \leq n - 1 \leq N$$

$$x = (\sum i : n \leq i < N : v[i]) \Rightarrow x + v[n - 1] = v[n - 1] + (\sum i : n \leq i < N : v[i])$$

## Suma de un vector

]

(i.3) Al salir del bucle se cumple la postcondición.

$$I \wedge \neg(n \neq 0)$$

$$\Leftrightarrow 0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : V[i]) \wedge n = 0$$

$$\Rightarrow x = (\sum i : 0 \leq i < N : V[i])$$

]

(i.3) Al salir del bucle se cumple la postcondición.

$$I \wedge \neg(n \neq 0)$$

$$\Leftrightarrow 0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : V[i]) \wedge n = 0$$

$$\Rightarrow x = (\sum i : 0 \leq i < N : V[i])$$

(c.1) Existe una función de cota que es mayor o igual que 0 cuando el bucle da una vuelta más y decrece al darla. Elegimos como función de cota  $n$ .

$$I \wedge n \neq 0 \Rightarrow n \geq 0$$

## Suma de un vector

]

(i.3) Al salir del bucle se cumple la postcondición.

$$I \wedge \neg(n \neq 0)$$

$$\Leftrightarrow 0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : V[i]) \wedge n = 0$$

$$\Rightarrow x = (\sum i : 0 \leq i < N : V[i])$$

(c.1) Existe una función de cota que es mayor o igual que 0 cuando el bucle da una vuelta más y decrece al darla. Elegimos como función de cota  $n$ .

$$I \wedge n \neq 0 \Rightarrow n \geq 0$$

(c.2) La cota decrece al ejecutar el cuerpo del bucle.

$$\{I \wedge n \neq 0 \wedge n = T\} \quad x = x + V[n-1]; \quad n = n - 1; \quad \{n < T\}$$

$$pmd(x = x + V[n-1]; \quad n = n - 1; \quad n < T)$$

$$\Leftrightarrow ((n < T)_n^{n-1})_x^{x+V[n-1]}$$

$$\Leftrightarrow n - 1 < T$$

$$\Leftarrow I \wedge n \neq 0 \wedge n = T$$

Entonces lo más importante es hallar el invariante y hallar la función de cota. El invariante siempre debe mantenerse antes de entrar el bucle, durante la ejecución del bucle y al final del bucle cuando termina y se debe cumplir la postcondición

## Derivación

Ahora vamos a pasar de verificar a derivar. Recordamos que verificar consistía en comprobar si el algoritmo era correcto UNA VEZ ya esté diseñado o implementado el código del algoritmo.

Derivar consiste en realizar el proceso de verificación en paralelo al diseño de un algoritmo. Creo que es lo que nosotros vamos a tener que hacer en nuestros ejercicios.

- **Derivar:** construir las instrucciones a partir de la especificación asegurando su corrección. Realizar la verificación a la vez que diseño de algoritmo
- La postcondición dirige el proceso de verificación.

- **Derivar:** construir las instrucciones a partir de la especificación asegurando su corrección.
- La postcondición dirige el proceso de verificación.
- Las igualdades de la postcondición se intentan satisfacer mediante las asignaciones correspondientes:
  - Si la precondición es más fuerte que el predicado más débil de estas asignaciones y la postcondición → Proceso finalizado.
  - En caso contrario utilizaremos instrucciones iterativas. Intentaremos ceñirnos al siguiente esquema:

```
{P}
A0      // Inicializacion: I se cumple al principio
{I, Cota}
while (B) {
    {I ∧ B}
    A1      // Restablecer: mantiene el invariante I
    {R}
    A2      // Avanzar: hace que la Cota decrezca
    {I}
}
{Q}
```

Si nos acordamos bien el proceso de derivación consiste en corregir el algoritmo a la misma vez que lo vamos a implementar.

## Pasos

Estos son los pasos para conseguir un buen invariante y una buena condición de bucle a la misma vez que hacemos el proceso de derivación (verificación).

Primero diseñar el invariante ( $I$ ) y la condición del bucle ( $B$ ).

① Diseñar  $I$  y  $B$  a partir de  $Q$ . Probar  $I \wedge \neg B \Rightarrow Q$ . Comprobar si post se cumple.

② Diseñar  $A_0$ . Probar  $\{P\} A_0 \{I\}$ . Diseñar func  $A_0$  (lo del principio cumple con invariante)

③ Diseñar  $C$ . Probar  $I \wedge B \Rightarrow C \geq 0$ . ¿Cuál es la FUNCIÓN DE COTA.

④ Diseñar  $A_2$ . Construir  $R \equiv \text{pmd}(A_2, I)$ . Precondición más débil.

⑤ Diseñar  $A_1$  comparando  $I \wedge B$  con  $R$ . comprobar que el invariante se mantiene en el cuerpo del bucle  
Probar  $\{I \wedge B\} A_1 \{R\}$ .

⑥ Probar  $\{I \wedge B \wedge C = T\} A_1 ; A_2 \{C < T\}$ . Comprobar que la cota decrece al ejecutar el cuerpo del bucle.

## **Derivación**

---

### **Algoritmos numéricos**

Cómo realizar el proceso de derivación en algoritmos numéricos.

## División entera

Precondición: solo podemos dividir números positivos y no podemos dividir entre 0

- Deriva un algoritmo que verifique la siguiente especificación:

pre $\{a \geq 0 \wedge b > 0\}$  el divisor no puede ser 0.

proc divide(int a, b; out int c, r)

pos $\{a = b * c + r \wedge 0 \leq r < b\}$  el resto debe ser menor que el divisor.

- Solución: La postcondición tiene forma conjuntiva:

- $I \equiv a = b * c + r \wedge 0 \leq r$

- Condición del bucle:  $r \geq b$

Mientras que el resto sea mayor que el divisor se seguirá ejecutando el bucle.

- Tenemos que asignar valores a  $c$  y  $r$  para que el invariante se cumpla:

$$\text{AD} \xrightarrow{\quad} \{P\} \quad \boxed{r=a; \quad c=0;} \quad \{I\}$$

Comprobamos:  $((a = b * c + r \wedge 0 \leq r)_c^0)^a_r \Leftrightarrow a = b * 0 + a \wedge 0 \leq 0 \Leftrightarrow a \geq 0 \wedge b > 0$

- Tenemos que asignar valores a  $c$  y  $r$  para que el invariante se cumpla:

$$\{P\} \quad r=a; \quad c=0; \quad \{I\}$$

Comprobamos:  $((a = b * c + r \wedge 0 \leq r)_c^0)^a_r \Leftrightarrow a = b * 0 + a \wedge 0 \leq 0 \Leftrightarrow a \geq 0 \wedge b > 0$

- Tomamos como función de cota=r:  $I \wedge B \Rightarrow r \geq 0$ .

- Tenemos que asignar valores a  $c$  y  $r$  para que el invariante se cumpla:

$$\{P\} \quad r=a; \quad c=0; \quad \{I\}$$

Comprobamos:  $((a = b * c + r \wedge 0 \leq r)_c^0)^a_r \Leftrightarrow a = b * 0 + a \wedge 0 \leq 0 \Leftrightarrow a \geq 0 \wedge b > 0$

- Tomamos como función de cota= $r$ :  $I \wedge B \Rightarrow r \geq 0$ .
- Podríamos pensar en tomar como instrucción para avanzar la asignación  $r = r - 1$ . Pero, ¿podemos restar más? Sabemos que  $r \geq b$ .  
Tomamos  $r = r - b$ ; y comprobamos:

$$\begin{aligned} R \equiv I_r^{r-b} &\Leftrightarrow a = b * c + (r - b) \wedge 0 \leq (r - b) \\ &\Leftrightarrow a = b * (c - 1) + r \wedge 0 \leq (r - b) \\ &\stackrel{?}{\Leftarrow} I \wedge B \end{aligned}$$

## División entera

- Para restablecer el invariante, incrementamos el cociente  $c$  en 1:

$$\{I \wedge B\} \ c = c+1; \{R\}$$

Comprobamos:  $R_c^{c+1} \Leftrightarrow a = b * (c + 1) + (r - b) \Leftarrow I \wedge B.$

- Para restablecer el invariante, incrementamos el cociente  $c$  en 1:

$$\{I \wedge B\} \ c = c+1; \ {R}\}$$

Comprobamos:  $R_c^{c+1} \Leftrightarrow a = b * (c + 1) + (r - b) \Leftarrow I \wedge B.$

- Veamos que la cota,  $r$ , decrece en cada vuelta del bucle:

$$\{I \wedge B \wedge r = T\} \ c = c+1; \ r = r-b; \ {r < T}\}$$

Comprobamos:  $((r < T)_r^{r-b})_c^{c+1} \Leftrightarrow r - b < T \Leftarrow I \wedge B \wedge r = T$

## División entera

Creo que nosotros mejor vamos a realizar los algoritmos y después a realizar la especificación.

- El algoritmo resultante:

Por ejemplo: 14/6

```
{b ≠ 0}
//proc divide(int a, int b, out int c, out int r)
void divide(int a, int b, int &c, int &r){
    c = 0;
    r = a;
    while (r) >= b {
        {l ≡ a = b * c + r ∧ 0 ≤ r, Cota : r}
        c = c + 1;
        r = r - b;
    }
    {a = b * c + r ∧ 0 ≤ r < b}
```

14 = 6\*2+2 luego se cumple

Esto es lo que decrece.

La r es lo que decrece

Luego el resto es dos, el cociente es 2

Pero también se cumple para el resto de iteraciones. Es decir se mantiene siempre y es por eso que es el invariante

Coste:  $\Theta(a \text{ div } b)$

Claro, porque el coste será igual al número de veces que tengamos que dividir. Por lo tanto será del orden de a entre b

## Raíz cuadrada entera

la raíz de cero existe. Y la raíz CUADRADA se puede realizar siempre que el número sea mayor que 0, luego esa es nuestra precondición

$\{P \equiv n \geq 0\}$  solo se puede hacer de números positivos.

fun raizEnt(int n) dev int r

$\{Q \equiv r^2 \leq n < (r + 1)^2\}$  La raíz cuadrada ENTERA de un número será un número que esté entre la raíz cuadrada de un número exacto y del siguiente exacto.  
 $3^2 \leq 12 < (4)^2$

Podemos elegir distintos invariantes:

$$\neg B \equiv r^2 \leq n \quad I \equiv n < (r + 1)^2$$

$$\neg B \equiv n < (r + 1)^2 \quad I \equiv r^2 \leq n$$

Entonces, nuestra condición del bucle será: mientras que  $n > (r+1)^2$ ...

## Raíz cuadrada entera

$$\{P \equiv n \geq 0\}$$

```
fun raizEnt(int n) dev int r
```

$$\{Q \equiv r^2 \leq n < (r+1)^2\}$$

Podemos elegir distintos invariantes:

$$\neg B \equiv r^2 \leq n \quad I \equiv n < (r+1)^2$$

$$\neg B \equiv n < (r+1)^2 \quad I \equiv r^2 \leq n$$

Tomamos como inicialización  $r = 0$ ;

$$\{ P \} \quad r = 0; \quad \{ I \}$$

Comprobamos:  $(r^2 \leq n)_r^0 \Leftrightarrow 0 \leq n \Leftrightarrow n \geq 0$       Se cumple I1)

## Raíz cuadrada entera

Tomamos como función de cota  $C = n - r^2 \geq 0$  y como instrucción para avanzar la asignación  $r = r + 1$ ;

¿Cómo afecta avanzar al invariante?

$$R \equiv I_r^{r+1} \Leftrightarrow (r+1)^2 \leq n \stackrel{?}{\Leftarrow} I \wedge B$$

## Raíz cuadrada entera

Tomamos como función de cota  $C = n - r^2 \geq 0$  y como instrucción para avanzar la asignación  $r = r + 1$ ;

¿Cómo afecta avanzar al invariante?

$$R \equiv I_r^{r+1} \Leftrightarrow (r+1)^2 \leq n \stackrel{?}{\Leftarrow} I \wedge B$$

Probamos la terminación

$$\{ I \wedge B \wedge n - r^2 = T \} \quad r = r + 1; \quad \{ n - r^2 < T \}$$

$$\begin{aligned} (n - r^2 < T)_r^{r+1} &\Leftrightarrow n - (r+1)^2 < T \\ &\Leftrightarrow n - r^2 - 2r - 1 < T \\ &\stackrel{?}{\Leftarrow} I \wedge B \wedge n - r^2 = T \end{aligned}$$

Es necesario añadir  $r \geq 0$  al invariante.

## Raíz cuadrada entera

```
{n ≥ 0}
//fun raizEnt(int n) dev int r
int raizEnt(int n){
    int r = 0;
    while (n >= (r+1)*(r+1)){
        {I ≡ r2 ≤ n, Cota : n - r2}
        r = r + 1;
    };
    return r;
}
{r2 ≤ n < (r + 1)2}
```

condición del bucle

Coste:  $\Theta(\sqrt{n})$

## Raíz cuadrada entera

```
{n ≥ 0}  
//fun raizEnt(int n) dev int r  
int raizEnt(int n){  
    int r = 0;  
    while (n >= (r+1)*(r+1) ){  
        {I ≡ r² ≤ n, Cota : n - r²}  
        r = r + 1;  
    };  
    return r;  
}
```

$$\{r^2 \leq n < (r+1)^2\} \quad 3^2 \leq 10 < 4^2$$

Coste:  $\Theta(\sqrt{n})$  primer algoritmo con este coste, ver en casa.

**Ejercicio:** Probar con  $B \equiv r^2 \leq n$  y  $I \equiv n < (r+1)^2$

## precondición.

$$\{P \equiv a > 0 \wedge b \geq 0\}$$

fun potencia(int a, int b) dev int r

$$\{Q \equiv r = a^b\}$$

## postcondición

No hay conjunciones. Añadimos nuevas variables que sustituyen a constantes (parámetros de entrada). Varias posibilidades:

- $r = a^x \wedge x = b$
- $r = y^b \wedge y = a$
- $r = y^x \wedge y = a \wedge x = b$

Elegimos  $\neg B \equiv x = b$ ,  $I \equiv r = a^x$ .

$$2^3$$

multiplicar 2 por si mismo 3 veces.

Mientras que b sea  $\geq 0$   
b función de cota.

## Potencia

- Inicialización:  $x=0$ ;  $r=1$ ;

$$((r = a^x)_r)^0_x \Leftrightarrow 1 = a^0 \Leftrightarrow \text{true}$$

## Potencia

- Inicialización:  $x=0; r=1;$

$$((r = a^x)_r)_x^0 \Leftrightarrow 1 = a^0 \Leftrightarrow \text{true}$$

- Avanzar:  $x = x + 1;$

Añadimos al invariante:  $0 \leq x \leq b$

Cota:  $b - x. I \wedge B \Rightarrow b - x \geq 0$

Calculamos

$$R \equiv I_x^{x+1} \Leftrightarrow r = a^{x+1} \wedge 0 \leq x + 1 \leq b \stackrel{?}{\Leftrightarrow} I \wedge B$$

## Potencia

- Inicialización:  $x=0; r=1;$

$$((r = a^x)_r)_x^0 \Leftrightarrow 1 = a^0 \Leftrightarrow \text{true}$$

- Avanzar:  $x = x + 1;$

Añadimos al invariante:  $0 \leq x \leq b$

Cota:  $b - x. I \wedge B \Rightarrow b - x \geq 0$

Calculamos

$$R \equiv I_x^{x+1} \Leftrightarrow r = a^{x+1} \wedge 0 \leq x + 1 \leq b \stackrel{?}{\Leftrightarrow} I \wedge B$$

- Restablecer:  $r = a * r;$

$$(I_x^{x+1})_r^{a * r} \Leftrightarrow a * r = a^{x+1} \wedge 0 \leq x + 1 \leq b \Leftrightarrow I \wedge B$$

## Potencia

- Inicialización:  $x=0; r=1;$

$$((r = a^x)_r)_x^0 \Leftrightarrow 1 = a^0 \Leftrightarrow \text{true}$$

- Avanzar:  $x = x + 1;$

Añadimos al invariante:  $0 \leq x \leq b$

Cota:  $b - x. I \wedge B \Rightarrow b - x \geq 0$

Calculamos

$$R \equiv I_x^{x+1} \Leftrightarrow r = a^{x+1} \wedge 0 \leq x + 1 \leq b \stackrel{?}{\Leftrightarrow} I \wedge B$$

- Restablecer:  $r = a * r;$

$$(I_x^{x+1})_r^{a * r} \Leftrightarrow a * r = a^{x+1} \wedge 0 \leq x + 1 \leq b \Leftrightarrow I \wedge B$$

- Terminación:

$$\{I \wedge B \wedge b - x = T\} \quad r = a * r; \quad x = x + 1; \quad \{b - x < T\}$$

## Potencia

```
{ $a > 0 \wedge b \geq 0\}$ 
//fun potencia(int a, int b) dev int r
int potencia(int a, int b) 3^5 = 243
{
    int x = 0; int r = 0;      1           r tiene que ser 1.
    while (x != b) {          en este caso b-x será la cota
        {I  $\equiv 0 \leq x \leq b \wedge r = a^x$ , Cota :  $b - x\}$ 
        r = a * r; 3, 9, 27, 81, 243
        x = x + 1; 1, 2, 3, 4, 5
    };
    return r;
}
{ $r = a^b\}$ 
```

Coste:  $\Theta(b)$  Claro, ya que el bucle se repite el orden de la potencia veces.

I:  $0 \leq x < b \wedge r = a^x$

## Derivación

Problemas de búsqueda

para algoritmos de búsqueda..

Dado cierto algoritmo debemos decir el invariante.

Ejemplo;

{true} -> pre

func buscar( vector<> v, x) dev(bool b, int x)

{

b= existe i t.q  $0 \leq i < v.size()$  con  $v[i] = x \ \&\& \ b \Rightarrow 0 \leq p < v.size()$ ...

## Problemas de búsqueda

Muchos problemas de programación pueden verse como **problemas de búsqueda**.

Por ejemplo, el problema de la raíz cuadrada entera de  $n$  puede formularse como "buscar el mayor natural  $i$  tal que  $i^2 \leq n$ "

$$r = (\max i : 0 \leq i \wedge i^2 \leq n : i)$$

O también puede formularse como

$$r = (\min i : 0 \leq i \wedge (i + 1)^2 > n : i)$$

es decir, buscamos el menor natural  $i$  tal que  $(i + 1)^2 > n$ .

- Búsqueda lineal
- Búsqueda lineal acotada
- Búsqueda binaria

Problemas de divide y vencerás del tema 5.

Acordarnos de FP2 que la búsqueda binaria se realizaba en el caso de que los elementos del vector ESTÉN ORDENADOS (mirar apuntes año pasado)

## Búsqueda lineal acotada

Dado un vector de booleanos  $b[N]$ , con  $N \geq 0$ , queremos asignar a  $x$  el menor valor  $i$ , con  $0 \leq i < N$ , tal que  $b[i]$  sea cierto. Si no existe,  $x$  debe valer  $N$ .

$\{0 \leq N \leq \text{longitud}(b)\}$  precondición correcta.

fun busquedaLinealAcotada(bool b[], int N) dev int x

$\{x = (\max i : 0 \leq i \leq N \wedge (\forall j : 0 \leq j < i : \neg b[j]) : i)\}$

todos los anteriores sean false. Desde la posición i hasta el principio son todos false porque nosotros queremos el primer true.

## Búsqueda lineal acotada

Dado un vector de booleanos  $b[N]$ , con  $N \geq 0$ , queremos asignar a  $x$  el menor valor  $i$ , con  $0 \leq i < N$ , tal que  $B[i]$  sea cierto. Si no existe,  $x$  debe valer  $N$ .

$$\{0 \leq N \leq \text{longitud}(b)\}$$

fun busquedaLinealAcotada(bool b[ ], int N) dev int x

$$\{x = (\max i : 0 \leq i \leq N \wedge (\forall j : 0 \leq j < i : \neg b[j]) : i)\}$$

desde esa posición j hasta la posición i todos son false. Y la i es el primer verdadero

Podemos escribir la postcondición como

$$0 \leq x \leq N \wedge (\forall j : 0 \leq j < x : \neg b[j]) \wedge \mathcal{B}(x)$$

donde

$$\mathcal{B}(x) = (0 \leq x < N \wedge_c b[x]) \vee (x = N)$$

Invariante:

$$I \equiv 0 \leq x \leq N \wedge (\forall j : 0 \leq j < x : \neg b[j])$$

## Búsqueda lineal acotada

- Si tomamos  $I \equiv 0 \leq x \leq N \wedge (\forall j : 0 \leq j < x : \neg b[j])$  entonces la condición del bucle será  $\neg \mathcal{B}(x) = x \neq N \text{ } \&\& \text{ } !b[x]$  puesto que  $0 \leq x \leq N$  en el invariante.
- Inicialización:  $x = 0;$

## Búsqueda lineal acotada

- Si tomamos  $I \equiv 0 \leq x \leq N \wedge (\forall j : 0 \leq j < x : \neg b[j])$  entonces la condición del bucle será  $\neg \mathcal{B}(x) = x \neq N \wedge \neg b[x]$  puesto que  $0 \leq x \leq N$  en el invariante.
- Inicialización:  $x = 0;$
- Cota:  $N - x$
- Avanzar:  $x = x + 1;$

$$\begin{aligned} I_x^{x+1} &\Leftrightarrow 0 \leq x + 1 \leq N \wedge (\forall j : 0 \leq j < x + 1 : \neg b[j]) \\ &\Leftrightarrow I \wedge (x \neq N \wedge_c \neg b[x]) \end{aligned}$$

## Búsqueda lineal acotada

```
{ $0 \leq N \leq longitud(b)$ }  
//fun busquedaLinealAcotada(bool b[ ], int N) dev int x  
int busquedaLinealAcotada(bool b[ ], int N){  
    int x = 0;  
    while (x != N && !b[x]) { Cota es N-x.  
        {I  $\equiv 0 \leq x \leq N \wedge (\forall j : 0 \leq j < x : \neg b[j])$ , Cota :N - x}  
        x = x + 1;  
    };  
    return x;  
}  
{x = ( $\max i : 0 \leq i \leq N \wedge (\forall j : 0 \leq j < i : \neg b[j]) : i$ )}
```

Coste:  $\Theta(N)$  en el caso peor (cuando no hay ninguno en el vector que sea cierto).

**Ejercicio:** Haz la búsqueda de derecha a izquierda.

Sería lo mismo pero desde N hasta 0 por lo que la función de cota sería n.

## Búsqueda lineal acotada

Otro ejercicio de búsqueda de un elemento.

Dado un vector  $v[N]$  de enteros devolver en  $x$  el menor índice del vector en el que haya un número par. Si no hay ninguno devolver  $N$ .

Ajustamos el esquema a nuestro problema:

```
{ $0 \leq N \leq longitud(v)$ }
```

```
x = 0;
```

```
while (x != N && v[x] % 2 != 0){  
    { $I \equiv 0 \leq x \leq N \wedge (\forall j : 0 \leq j < x : v[j] \bmod 2 \neq 0)$ ,  
     x = x + 1;  
}
```

```
{ $x = (\max i : 0 \leq i \leq N) \wedge (\forall j : 0 \leq j < i : v[j] \bmod 2 \neq 0) : i$ }
```

Todos los anteriores al primer par deben ser impares.

Cota :  $N - x$

Esto es menor o igual porque puede devolver N

Coste:  $O(N)$  en el caso peor (cuando no hay ninguno en el vector que sea par).

**Ejercicio:** Dado un vector de enteros, se dice que un índice es *pastoso* si el valor del vector en dicha posición es igual a la suma de los valores de todas las posiciones que le siguen. Buscar el mayor índice pastoso.

Si no me equivoco esto es un ejercicio del juez.

## **Derivación**

---

### **Variables acumuladoras**

## Suma de elementos buenos

Queremos un algoritmo que satisfaga la siguiente especificación:

$$\{0 \leq N \leq \text{longitud}(v)\}$$

fun sumaBuenos(int  $v[ ]$ , int  $N$ ) dev int  $s$

$$\{s = (\sum i : 0 \leq i < N \wedge \text{bueno}(i, v) : v[i])\}$$

donde  $\text{bueno}(i, v) \equiv (v[i] = 2^i)$ , sin utilizar (en el algoritmo) ninguna operación que calcule potencias.

## Suma de elementos buenos

Queremos un algoritmo que satisfaga la siguiente especificación:

$$\{0 \leq N \leq longitud(v)\}$$

fun sumaBuenos(int  $v[ ]$ , int  $N$ ) dev int  $s$

$$\{s = (\sum i : 0 \leq i < N \wedge bueno(i, v) : v[i])\}$$

donde  $bueno(i, v) \equiv (v[i] = 2^i)$ , sin utilizar (en el algoritmo) ninguna operación que calcule potencias.

v[0] es bueno si vale 1. v[1] es bueno si vale 2. Es decir si son potencias de 2

$$I \equiv s = (\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]) \wedge 0 \leq n \leq N$$

## Suma de elementos buenos

Queremos un algoritmo que satisfaga la siguiente especificación:

$$\{0 \leq N \leq longitud(v)\}$$

fun sumaBuenos(int  $v[ ]$ , int  $N$ ) dev int  $s$   
 $\{s = (\sum i : 0 \leq i < N \wedge bueno(i, v) : v[i])\}$

donde  $bueno(i, v) \equiv (v[i] = 2^i)$ , sin utilizar (en el algoritmo) ninguna operación que calcule potencias.

$$I \equiv s = (\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]) \wedge 0 \leq n \leq N$$

Tomamos:

- $B \equiv n \neq N$ .
- Inicialización:  $n = 0$ ;  $s = 0$ ;
- Instrucción para avanzar  $n = n + 1$ ;

Cota:  $N - n$

## Suma de elementos buenos

Vemos cuál es el efecto que sobre el invariante tiene avanzar.

$$\begin{aligned} I_n^{n+1} &\equiv s = (\sum i : 0 \leq i < n + 1 \wedge \text{bueno}(i, v) : v[i]) \wedge 0 \leq n + 1 \leq N \\ &\stackrel{?}{\Leftarrow} I \wedge B \end{aligned}$$

La última parte no plantea problemas, ya que

$$0 \leq n \leq N \wedge n \neq N \Rightarrow 0 \leq n + 1 \leq N.$$

Veamos cómo hacer cierta la primera igualdad.

$$\begin{aligned} s &= (\sum i : 0 \leq i < n + 1 \wedge \text{bueno}(i, v) : v[i]) \\ \Leftrightarrow s &= (\sum i : 0 \leq i < n \wedge \text{bueno}(i, v) : v[i]) + \begin{cases} v[n] & \text{si } \text{bueno}(n, v) \\ 0 & \text{si } \neg \text{bueno}(n, v) \end{cases} \end{aligned}$$

## Suma de elementos buenos

```
n = 0; s = 0;
while (n != N) {
    {I ≡ s = ( $\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]$ )  $\wedge 0 \leq n \leq N$ , Cota :  $N - n$ }
    if (v[n] es bueno)
        s = s + v[n];
    n = n + 1;
}
```

## Suma de elementos buenos

```
n = 0; s = 0;
while (n != N) {
    {I ≡ s = ( $\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]$ )  $\wedge 0 \leq n \leq N$ , Cota :  $N - n$ }
    if (v[n] es bueno)
        s = s + v[n];
    n = n + 1;
}
```



Para hacer de manera eficiente la comprobación de que  $v[n]$  es bueno, introducimos en el invariante una nueva variable  $p$  que lleva calculada la potencia de 2 que nos hace falta.

Añadimos  $p = 2^n$ . De esta forma para comprobar que es bueno basta con  $v[n] = p$ .

Inicialización de la nueva variable:  $p = 1$ ;

## Suma de elementos buenos

```
n = 0; s = 0; p = 0;
while (n != N) {
    {I ≡ s = ( $\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]$ )  $\wedge 0 \leq n \leq N \wedge p = 2^n$ , Cota :  $N - n$ }
    if (v[n] == p)
        s = s + v[n];
    p = ?";
    {I_n^{n+1} ≡ s = ( $\sum i : 0 \leq i < n + 1 \wedge bueno(i, v) : v[i]$ )  $\wedge 0 \leq n + 1 \leq N \wedge p = 2^{n+1}$ }
    n = n + 1;
}
```

## Suma de elementos buenos

```
n = 0; s = 0; p = 0;
while (n != N) {
    {I ≡ s = ( $\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]$ )  $\wedge 0 \leq n \leq N \wedge p = 2^n$ , Cota : N - n}
    if (v[n] == p)
        s = s + v[n];
    p = ?;
    {I_n^{n+1} ≡ s = ( $\sum i : 0 \leq i < n + 1 \wedge bueno(i, v) : v[i]$ )  $\wedge 0 \leq n + 1 \leq N \wedge p = 2^{n+1}$ }
    n = n + 1;
}
```

Falta restablecer  $p$ :

$$p = 2^{n+1} \Leftrightarrow p = 2 * 2^n.$$

Luego debemos hacer  $p = 2 * p$ ; para restablecerla.

## Suma de elementos buenos

```
{ $0 \leq N \leq longitud(v)$ }  
//fun sumaBuenos(int v[ ], int N) dev int s  
int sumaBuenos(int v[ ], int N) {  
    int n = 0; int s = 0; int p = 0; Esto daría problemas CREO.  
    while (n != N) {  
        { $I \equiv s = (\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i]) \wedge 0 \leq n \leq N \wedge p = 2^n$ , Cota :  $N - n$ }  
        if (v[n] == p) esto es si es bueno.  
            s = s + v[n];  
        p = 2 * p;  
        n = n + 1;  
    };  
    return s;  
}  
{ $s = (\sum i : 0 \leq i < n \wedge bueno(i, v) : v[i])$ }
```

Coste:  $\Theta(N)$

Debería de ser p inicializado a 1 porque en la primera iteración  $v[n]$  debería ser = 1 ya que  $2^0=1$

## Ejercicios con variables acumuladoras

- Suma de picos (Ejercicio 16)
- Número de índices heroicos (Ejercicio 17)
- Ejercicios de matrices (31-35)

Realizar estos ejercicios del juez es importante.

Estos ejercicios son de elementos y no de segmentos. Se suelen hacer de coste lineal como ya hemos visto.

## Segmento de suma máxima

Muy importantes también estos ejercicios de segmentos

Queremos una función de coste lineal (con respecto a la longitud del vector) que calcule, dado un vector no vacío de enteros, la suma del segmento no vacío de suma máxima.

Un par  $p, q$  representa el segmento  $[p, q]$ .

La  $q$  sin incluir siempre; acordarnos.

La especificación es la siguiente:

$\{0 < N \leq \text{longitud}(v)\}$  → No vacío

fun segSumaMax(int v[ ], int N) dev int r  
 $\{r = (\max p, q : 0 \leq p < q \leq N : S(p, q))\}$

dónde  $S(p, q) = (\sum i : p \leq i < q : X[i])$ .

Esto es debido a que cogemos  $[p, q)$

SEGMENTO-> REP  $[p, q)$   
menor solo porque no incluimos el de la derecha (lo dijo en el tema 2).

EN C++ nosotros lo haríamos cubico u cuadrático, pero nosotros queremos conseguirlo lineal.

Procesamos el vector de izquierda a derecha.

## Segmento de suma máxima

Queremos una función de coste lineal (con respecto a la longitud del vector) que calcule, dado un vector no vacío de enteros, la suma del segmento no vacío de suma máxima.

Un par  $p, q$  representa el segmento  $[p, q]$ .

La especificación es la siguiente:

$\{0 < N \leq \text{longitud}(v)\}$  → Estrictamente mayor que 0. Porque el vector NO PUEDE SER vacío  
fun segSumaMax(int v[ ], int N) dev int r  
 $\{r = (\max p, q : 0 \leq p < q \leq N : S(p, q))\}$

donde  $S(p, q) = (\sum i : p \leq i < q : X[i])$ .

Parte del invariante:

$$I \equiv 1 \leq n \leq N \wedge r = (\max p, q : 0 \leq p < q \leq n : S(p, q))$$

Condición del bucle:  $B \equiv n != N$

Nuestro bucle será algo así:

```
int i=0;  
while(i< v.size());
```

## Segmento de suma máxima

- Inicialización:  $n = 1; r = v[0];$

SI lo inicializamos a 0 el invariante no se cumple desde el principio.

Si es 1:  $0 \leq p < q \leq 1$  tendríamos el siguiente intervalo -----> [0,1)

## Segmento de suma máxima

Para que no sea el segmento vacío

- Inicialización:  $n = 1; r = v[0];$
- Avanzar:  $n = n + 1; .$  Cota:  $N - n$

$$1 \leq n \leq N \wedge n \neq N \Rightarrow 1 \leq n + 1 \leq N.$$

Veamos qué pasa con la parte correspondiente a  $r$ :

$$\begin{aligned} & (\max p, q : 0 \leq p < q \leq n + 1 : \mathcal{S}(p, q)) \\ = & (\max p, q : 0 \leq p < q \leq n : \mathcal{S}(p, q)) \\ & \max \\ & (\max p : 0 \leq p < n + 1 : \mathcal{S}(p, n + 1)) \\ = & r \max (\max p : 0 \leq p < n + 1 : \mathcal{S}(p, n + 1)). \end{aligned}$$

## Segmento de suma máxima

- Inicialización:  $n = 1; r = v[0];$
- Avanzar:  $n = n + 1; .$  Cota:  $N - n$

$$1 \leq n \leq N \wedge n \neq N \Rightarrow 1 \leq n + 1 \leq N.$$

Veamos qué pasa con la parte correspondiente a  $r$ :

$$\begin{aligned} & (\max p, q : 0 \leq p < q \leq n + 1 : S(p, q)) \\ = & (\max p, q : 0 \leq p < q \leq n : S(p, q)) \\ & \max \\ & (\max p : 0 \leq p < n + 1 : S(p, n + 1)) \\ = & r \max (\max p : 0 \leq p < n + 1 : S(p, n + 1)). \end{aligned}$$

Añadimos al invariante la misma expresión con  $n$  en vez de  $n + 1$ , pues si  $n = N$   $S(p, n + 1)$  no está bien definido:

$$S \equiv s = (\max p : 0 \leq p < n : S(p, n)).$$

## Segmento de suma máxima

Cuando  $n = 1$ , el único posible valor para  $p$  es 0, y entonces  $s = S(0, 1) = v[0]$ .

Hasta aquí, hemos obtenido el esquema siguiente:

```
int n = 1; int r = v[0]; int s = v[0];
while (n != N) {
    {I ∧ n ≠ N}
    s = ?;
    {I ∧ Snn+1}
    r = max ( r, s );
    {Inn+1 ∧ Snn+1}
    n = n + 1;
}
```

## Segmento de suma máxima

Desarrollando la expresión del máximo en  $S_n^{n+1}$ :

$$\begin{aligned}& (\max p : 0 \leq p < n + 1 : S(p, n + 1)) \\&= (\max p : 0 \leq p < n : S(p, n + 1)) \max S(n, n + 1) \\&= (\max p : 0 \leq p < n : (S(p, n) + v[n])) \max X[n] \\&= ((\max p : 0 \leq p < n : S(p, n)) + v[n]) \max v[n] \\&= (s + v[n]) \max v[n]\end{aligned}$$

## Segmento de suma máxima

El algoritmo final es:

```
{0 < N ≤ longitud(v)}  
//fun segSumaMaxima(int v[ ], int N) dev int r  
int segSumaMax(int v[ ], int N)  
{  
    int n = 1; int r = v[0]; int s = v[0];  
    while (n != N) {  
        s = max(s + v[n], v[n]); v[n] es toda la suma  
        r = max (r, s); que lleva  
        n = n + 1;  
    };  
    return r;  
}  
{r = (max p, q : 0 ≤ p < q ≤ N : S(p, q))}
```

r va llevando el máximo

La cota es  $v.size() - i$ , el  $i$  avanza de 1 en 1 y por eso decrece.

Y hay una variable que se encarga de almacenar el máximo. Sumar el max anterior más  $v[i]$  y comprobar si ese nuevo valor es más grande que el maximo anterior

El coste en tiempo está en  $\Theta(N)$ . coste lineal que es lo que queríamos conseguir.

**Ejercicio:** devolver los extremos del segmento de suma máxima

No se si es que se guarden los extremos  $p$  y  $q$  o los valores  $v[p]$  y  $v[q]$ . Podríamos añadir otra variable y un if de si es mayor entonces guardar los extremos.

## Derivación

---

**Segmento más largo**

## Problemas de segmentos más largos

son del estilo:  $\max p,q: \dots q-p$ . Recordamos: NO INCLUÍMOS EL Q

Dado un vector  $v$  estamos interesados en la longitud del **segmento más largo  $[p..q]$**  que cumpla cierta propiedad.

Cual es el segmento más largo + condiciones. Hay muchos de estos pero son parecidos. Hay uno que es distinto.

## Problemas de segmentos más largos

Dado un vector  $v$  estamos interesados en la longitud del **segmento más largo**  $[p..q]$  que cumpla cierta propiedad.

Segmento más largo para el cual todos los elementos son 0

**Ejemplo:** todos los elementos del segmento son 0

$\{0 \leq N \leq \text{longitud}(v)\}$  precondición

fun todosCero(int v[], int N) dev int r

Se permite el segmento vacío.

$\{r = (\max p, q : 0 \leq p \leq q \leq N \wedge \mathcal{A}(p, q) : q - p)\}$

Longitud es  $q - p$ . Los segmentos no son todos, son todos los que cumplen una propiedad donde  $\mathcal{A}(p, q) = (\forall i : p \leq i < q : v[i] = 0)$ .

Da igual en este caso recorrerlo de dcha a izq que viceversa. Importaría en el caso de que nos dijeran "el primer segmento" / "último segmento".

La longitud del segmento  $[0,0]$  es 0

## Problemas de segmentos más largos

Dado un vector  $v$  estamos interesados en la longitud del **segmento más largo**  $[p..q]$  que cumpla cierta propiedad.

**Ejemplo:** todos los elementos del segmento son 0

$$\{0 \leq N \leq \text{longitud}(v)\}$$

fun todosCero(int v[], int N) dev int r

$$\{r = (\max p, q : 0 \leq p \leq q \leq N \wedge \mathcal{A}(p, q) : q - p)\}$$

donde  $\mathcal{A}(p, q) = (\forall i : p \leq i < q : v[i] = 0)$ .

¿Qué propiedades cumple  $\mathcal{A}$ ?

Si el número de segmentos es vacío entonces es true

- Cierta para **segmentos vacíos**:  $\mathcal{A}(p, p) = \text{true}$ . Esta propiedad es importante por dos razones:
  - El vector vacío ( $N = 0$ ) solo tiene segmentos vacíos.
  - Puede suceder que ningún segmento no vacío cumpla la propiedad.
- **Cerrada bajo prefijos** (por la izquierda)  $\mathcal{A}(p, q) \Rightarrow (\forall i : p \leq i \leq q : \mathcal{A}(p, i))$
- **Cerrada bajo sufijos** (por la derecha)  $\mathcal{A}(p, q) \Rightarrow (\forall i : p \leq i \leq q : \mathcal{A}(i, q))$

## Todos cero

Invariante:  $I \equiv 0 \leq n \leq N \wedge r = (\max p, q : 0 \leq p \leq q \leq n \wedge \mathcal{A}(p, q) : q - p)$

## Todos cero

Invariante:  $I \equiv 0 \leq n \leq N \wedge r = (\max p, q : 0 \leq p \leq q \leq n \wedge \mathcal{A}(p, q) : q - p)$

Inicialización:  $n = 0$ ; y

$$\begin{aligned} r &= (\max p, q : 0 \leq p \leq q \leq 0 \wedge \mathcal{A}(p, q) : q - p) \\ &= (\max p, q : p = 0 \wedge q = 0 \wedge \mathcal{A}(p, q) : q - p) \\ &= 0 - 0 = 0 \end{aligned}$$

## Todos cero

Invariante:  $I \equiv 0 \leq n \leq N \wedge r = (\max p, q : 0 \leq p \leq q \leq n \wedge \mathcal{A}(p, q) : q - p)$

Inicialización:  $n = 0$ ; y

$$\begin{aligned} r &= (\max p, q : 0 \leq p \leq q \leq 0 \wedge \mathcal{A}(p, q) : q - p) \\ &= (\max p, q : p = 0 \wedge q = 0 \wedge \mathcal{A}(p, q) : q - p) \\ &= 0 - 0 = 0 \end{aligned}$$

Avanzar:  $n = n + 1$ ; Cota:  $N - n$

$$\begin{aligned} &(\max p, q : 0 \leq p \leq q \leq n + 1 \wedge \mathcal{A}(p, q) : q - p) \\ &= (\max p, q : 0 \leq p \leq q \leq n \wedge \mathcal{A}(p, q) : q - p) \\ &\quad \max \\ &\quad (\max p : 0 \leq p \leq n + 1 \wedge \mathcal{A}(p, n + 1) : (n + 1) - p) \\ &= r \max (n + 1 + (\max p : 0 \leq p \leq n + 1 \wedge \mathcal{A}(p, n + 1) : -p)) \\ &= r \max (n + 1 - (\min p : 0 \leq p \leq n + 1 \wedge \mathcal{A}(p, n + 1) : p)) \end{aligned}$$

## Todos cero

Añadimos al invariante

$$Q \equiv s = (\min p : 0 \leq p \leq n \wedge \mathcal{A}(p, n) : p)$$

Iniciar:  $s = 0;$

Restablecer:  $r = \max(r, n + 1 - s);$

Por ahora tenemos

```
n = 0; r = 0; s = 0;
while (n != N) {
    {Q}
    ???
    {Q_n^{n+1}}
    r = max(r, (n+1-s));
    n = n + 1;
}
```

## Todos cero

$$Q \equiv s = (\min p : 0 \leq p \leq n \wedge \mathcal{A}(p, n) : p)$$

podemos escribirlo también como

$$\underbrace{0 \leq s \leq n}_{Q_0} \wedge \underbrace{\mathcal{A}(s, n)}_{Q_1} \wedge \underbrace{(\forall p : 0 \leq p < s : \neg \mathcal{A}(p, n))}_{Q_2}$$

$$Q_n^{n+1} \equiv 0 \leq s \leq n + 1 \wedge \mathcal{A}(s, n + 1) \wedge (\forall p : 0 \leq p < s : \neg \mathcal{A}(p, n + 1))$$

## Todos cero

$$Q \equiv s = (\min p : 0 \leq p \leq n \wedge \mathcal{A}(p, n) : p)$$

podemos escribirlo también como

$$\underbrace{0 \leq s \leq n}_{Q_0} \wedge \underbrace{\mathcal{A}(s, n)}_{Q_1} \wedge \underbrace{(\forall p : 0 \leq p < s : \neg \mathcal{A}(p, n))}_{Q_2}$$

$$Q_n^{n+1} \equiv 0 \leq s \leq n + 1 \wedge \mathcal{A}(s, n + 1) \wedge (\forall p : 0 \leq p < s : \neg \mathcal{A}(p, n + 1))$$

Queremos comprobar si  $Q_0 \wedge Q_1 \wedge Q_2 \Rightarrow Q_n^{n+1}$ :

- $Q_0 \Rightarrow 0 \leq s \leq n + 1$
  - $Q_2 \Rightarrow (\forall p : 0 \leq p < s : \neg \mathcal{A}(p, n + 1))$  por ser  $\mathcal{A}$  cerrada bajo prefijos.
  - Nos falta  $\mathcal{A}(s, n + 1)$ :
    - Si  $\mathcal{A}(s, n + 1)$ , no hay que hacer nada.
    - Si  $\neg \mathcal{A}(s, n + 1)$ , debemos buscar un nuevo valor a  $s$  tal que  $\mathcal{A}(s, n + 1)$ . Para ello podremos usar el hecho de que  $\mathcal{A}(s, n)$ .
- Además:  $Q_2 \Rightarrow (\min p : 0 \leq p \leq n + 1 \wedge \mathcal{A}(p, n + 1) : p) \geq s$
- Luego **basta con investigar valores de  $p$  tales que  $s \leq p \leq n + 1$**

## Todos cero

Ahora usamos la definición del problema concreto:  $\mathcal{A}(p, q) = (\forall i : p \leq i < q : v[i] = 0)$

$$\begin{aligned}\mathcal{A}(p, n+1) &\Leftrightarrow (\forall i : p \leq i < n+1 : v[i] = 0) \\ &\Leftrightarrow (\forall i : p \leq i < n : v[i] = 0) \wedge (v[n] = 0) \\ &\Leftrightarrow \mathcal{A}(p, n) \wedge (v[n] = 0)\end{aligned}$$

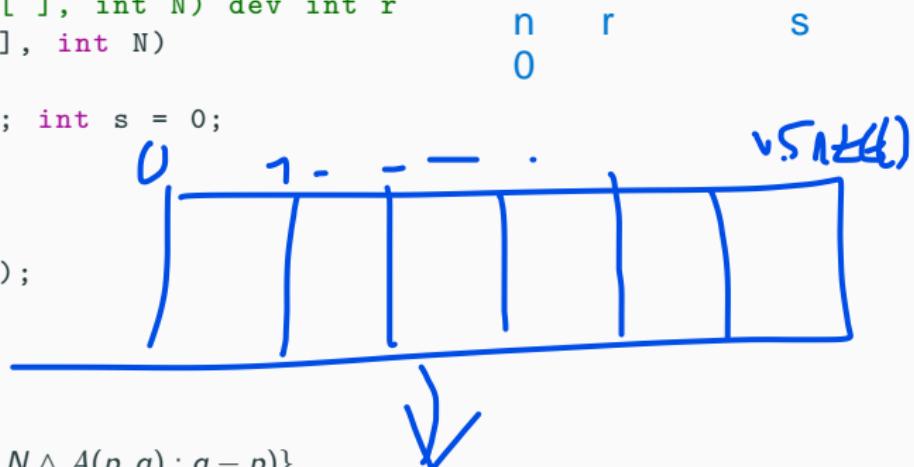
①  $Q \wedge v[n] = 0 \Rightarrow Q_n^{n+1}$

②  $Q \wedge v[n] \neq 0 \Rightarrow (Q_n^{n+1})_s^{n+1}$

Pues si  $v[n] \neq 0$  entonces  $(\forall p : s \leq p \leq n : \neg \mathcal{A}(p, n+1))$

## Todos cero

```
{ $0 \leq N \leq longitud(v)$ }  
//fun todosCero(int v[ ], int N) dev int r  
int todosCero(int v[ ], int N)  
{  
    int n = 0; int r = 0; int s = 0;  
    while (n != N) {  
        if (v[n] != 0)  
            s = n + 1;  
        r = max(r, (n+1-s));  
        n = n + 1;  
    };  
    return r;  
}  
{ $r = (\max p, q : 0 \leq p \leq q \leq N \wedge A(p, q) : q - p)$ }
```



Coste:  $\Theta(N)$

La s indicará el principio del vector

## Ejercicios de segmentos más largos

- Segmento más largo de elementos crecientes.
- Segmento más largo tal que el extremo izquierdo es el menor del segmento.
- Segmento más largo con a lo sumo 10 ceros.

## Derivación

---

### Modificación de vectores

Tenemos dos tipos de problemas:

Los de particion de vectores.

Los de mezcla.

Se ven algoritmos de otro tema.

# Partición

Tenemos un vector

$$\{0 \leq a \leq b < v.size() \wedge v = V\}$$

proc particion(inout vector<int> v, int a, int b, out int p) (a,b)

$$\{0 \leq a \leq p \leq b < v.size() \wedge \text{multiset}(V[a..b]) = \text{multiset}(v[a..b]) \wedge v[p] = V[a]$$

$$V[0..a-1] = v[0..a-1] \wedge V[b+1..v.size()-1] = v[b+1..v.size()-1] \wedge$$

$$(\forall x : a \leq x \leq p-1 : v[x] \leq v[p]) \wedge (\forall y : p+1 \leq y \leq b : v[y] \geq v[p])\}$$

donde van a ser menores que el pivot multiconjunto, permutacion el 7 hace de pivote y hace intercambios en el segmento. dos multiconjuntos son el mismo ==> que uno es la permutación de otro.

- $\text{multiset}(V[a..b]) = \text{multiset}(v[a..b])$  expresa que los valores contenidos en las posiciones  $[a..b]$  de  $v$  solamente han cambiado de lugar dentro del vector: no se ha perdido ninguno ni ha aparecido ninguno que no estuviera antes.
- $V[0..a-1] = v[0..a-1] \wedge V[b+1..v.size()-1] = v[b+1..v.size()-1]$  expresa que el algoritmo no modifica las partes del vector fuera del intervalo  $[a..b]$ .

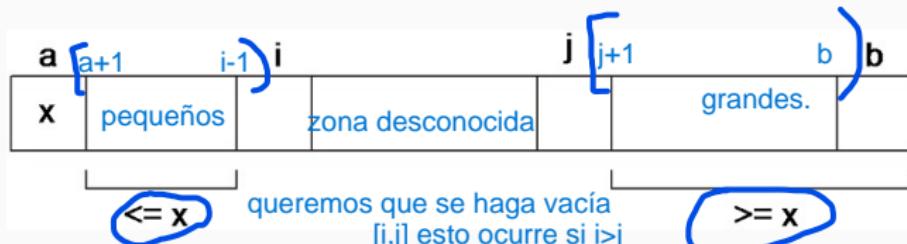
Estas dos propiedades formarán parte del invariante ya que solamente se realizarán intercambios de elementos en el intervalo  $[a..b]$ .

menores



## Partición

- El invariante se obtiene generalizando la postcondición con la introducción de dos variables nuevas,  $i, j$ , que indican el avance por los dos extremos del subvector hasta cruzarse.



$$\begin{aligned} a + 1 \leq i \leq j + 1 \leq b + 1 \wedge \text{multiset}(V[a..b]) = \text{multiset}(v[a..b]) \wedge \\ V[0..a] = v[0..a] \wedge V[b + 1..v.size() - 1] = v[b + 1..v.size() - 1] \wedge \\ (\forall x : a + 1 \leq x \leq i - 1 : v[x] \leq v[a]) \wedge \\ (\forall y : j + 1 \leq y \leq b : v[y] \geq v[a]) \end{aligned}$$

- El bucle termina cuando se cruzan los índices  $i$  y  $j$ , es decir, cuando se cumple  $i = j + 1$ , y, por lo tanto, la condición del bucle es  $i \leq j$
- A la salida del bucle, el vector estará particionado salvo por el pivote  $v[a]$ , que aún no ha cambiado. Para terminar el proceso basta con intercambiar los elementos de las posiciones  $a$  y  $j$ , quedando la partición en la posición  $j$ .

Cuales la función de cota? Qué es lo que va a decrecer? La zona desconocida-> Cota =  $j-i+1$

- Expresión de acotación:  $C : j - i + 1$
- Acción de inicialización:  $i = a + 1; j = b;$   
Esta acción hace trivialmente cierto el invariante porque  $v[(a + 1)..(i - 1)]$  y  $v[(j + 1)..b]$  se convierten en subvectores vacíos.
- Acción de avance: el objetivo del bucle es conseguir que  $i$  y  $j$  se vayan acercando, y además se mantenga el invariante en cada iteración. Para ello, se hace un análisis de casos comparando las componentes  $v[i]$  y  $v[j]$  con  $v[a]$ :
  - $v[i] > v[a] \wedge v[j] < v[a] \rightarrow$  intercambiamos  $v[i]$  con  $v[j]$ , incrementamos  $i$  y decrementamos  $j$
  - $v[i] \leq v[a] \rightarrow$  incrementamos  $i$
  - $v[j] \geq v[a] \rightarrow$  decrementamos  $j$

## Partición

```
void particion( vector<int> &v, int a, int b, int & p) {  
    int aux;  
    int i = a+1; int j = b; int aux;  
  
    while ( i <= j ) {  
        if ( (v[i] > v[a]) && (v[j] < v[a]) ) {  
            aux = v[i]; v[i] = v[j]; v[j] = aux;  
            i = i + 1; j = j - 1;  
        }  
        else {  
            if ( v[i] <= v[a] )  
                i = i + 1;  
            if ( v[j] >= v[a] )  
                j = j - 1;  
        }  
    }  
  
    p = j;  
    aux = v[a]; v[a] = v[p]; v[p] = aux;  
}
```

NO es `v.size()`, es muy poco preciso.

Lo utilizaremos en el contexto de Quicksort

Coste:  $\Theta(b - a)$  El coste es lineal del orden del tamaño del trozo. En este caso  $b-a$ . Depende del  $a$  y del  $b$  con el que lo llame.

# Mezcla de vectores ordenados

Aquí hay algunas erratas.

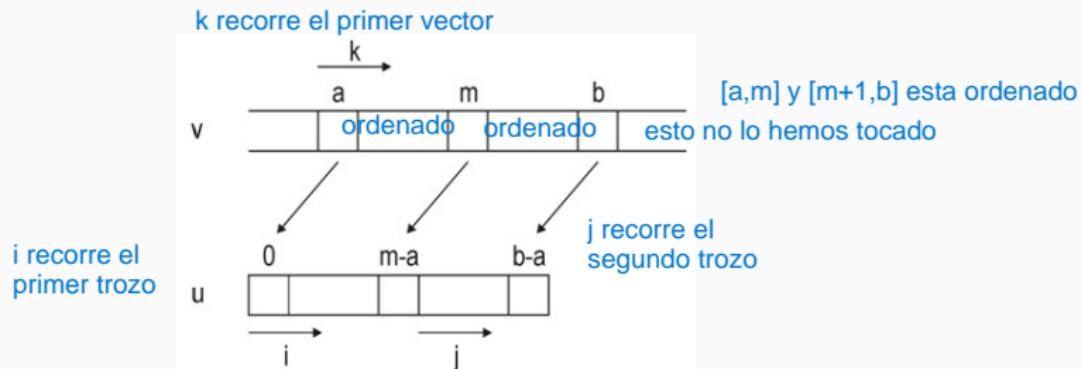
Asimilar con barajas. Varias barajas. Dados 2 vectores ordenados, las mezcla ORDENADAMENTE, se utiliza en el MerchShort.

$$\{0 \leq a \leq m \leq b < v.size() \wedge v = V \wedge ord(v, a, m) \wedge ord(v, m + 1, b)\}$$

```
proc mezcla(inout vector<int> v, int a, int m, int b)
```

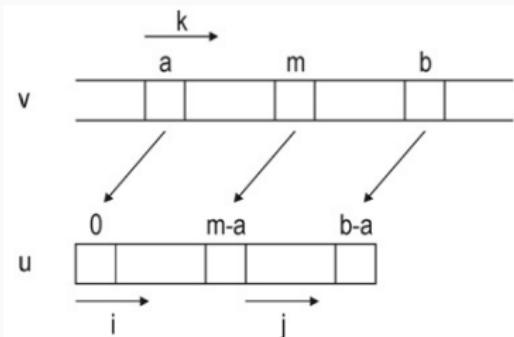
$$\{ord(v, a, b) \wedge multiset(V[a..b]) = multiset(v[a..b]) \wedge$$

$$V[0..a - 1] = v[0..a - 1] \wedge V[b + 1..v.size() - 1] = v[b + 1..v.size() - 1]\}$$



Tenemos dos trozos ordenados----> queremos obtener TODO el conjunto ORDENADO.

## Mezcla de vectores ordenados



$$k = a + i + (j - (m-a+1))$$

Invariante del bucle principal de mezcla:

$$k = a + i + (j - m - 1) \wedge a \leq i \leq m \leq j \leq b \wedge \text{X} \quad \text{Los indices están mal.}$$

$$\text{ord}(v, a, k) \wedge v[a..k-1] \leq u[i..m-a] \wedge v[a..k-1] \leq u[j..b-a] \wedge$$

$$\text{multiset}(v[a..k-1]) = \text{multiset}(u[0..i-1]) + \text{multiset}(u[m-a+1..j-1]) \wedge$$

$$\text{multiset}(u[0..b-a]) = \text{multiset}(V[a..b]) \wedge$$

$$v[0..a-1] = V[0..a-1] \wedge v[m+1..b] = V[m+1..b]$$

$$0 \leq i \leq m-a+1$$

$$m-a+1 \leq j \leq b-a+1$$

Esto es lo que está mal en las transparencias.

## Mezcla de vectores ordenados

```
void mezcla(vector<int> & v, int a, int m, int b) {
    int *u = new int[b-a+1];
    int i, j, k;

    //Copia en el vector auxiliar
    for (k = a; k <= b; k++) u[k-a] = v[k]; creamos un nuevo vector.

    //Bucle principal de la mezcla
    i = 0; j = m-a+1; k = a;
    while ((i <= m-a) && (j <= b-a)) { cuando el 1er trozo y el 2o no sean vacíos.
        if (u[i] <= u[j]){
            v[k] = u[i];
            i = i + 1;
        } else {
            v[k] = u[j];
            j = j + 1;
        }
        k = k + 1;
    }
}
```

## Mezcla de vectores ordenados

```
while ( i <= m-a ) {  
    v[k] = u[i];  
    i = i+1;  
    k = k+1;  
}  
while ( j <= b-a ) {  
    v[k] = u[j];  
    j = j+1;  
    k = k+1;  
}  
delete [] u;  
}
```

Coste:  $\Theta(b - a)$  El coste vuelve a ser del orden del tamaño del trozo.