Especificación de algoritmos

Ricardo Peña es el autor principal de este tema

Facultad de Informática - UCM

7 de octubre de 2013

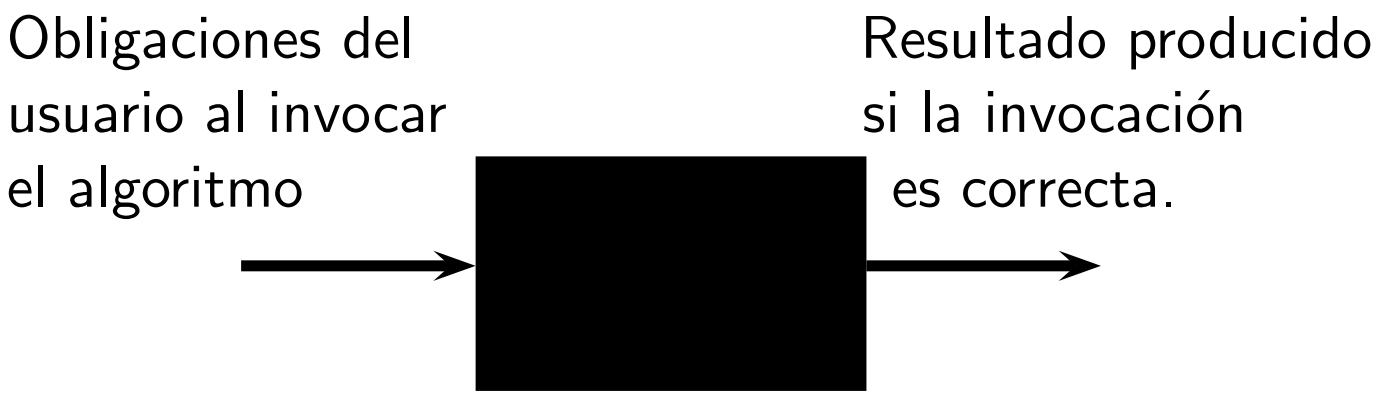
Biliografía Recomendada

- **Diseño de Programas: Formalismo y Abstracción**. *Ricardo Peña*. Tercera edición, Pearson Prentice-Hall, 2005
- **Estructuras de datos: un enfoque moderno**. *Mario Rodríguez Artalejo, Pedro Antonio González Calero y Marco Antonio Gómez Martín*. Editorial Complutense, 2011
- **Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos**. *Narciso Martí Oliet, Clara María Segura Díaz y Jose Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006

Introducción

- Especificar un algoritmo.

"**qué** hace el algoritmo".



- Implementar,

"**cómo** se consigue la funcionalidad pretendida".

Introducción

Especificación 1: Dado un vector de números enteros A ordenado de menor a mayor y un número entero x, devolver un valor booleano b que indique si alguno de los elementos del vector $A[0], \ldots, A[n-1]$ es igual al valor dado x.

Implementación: Algoritmo de búsqueda secuencial o algoritmo de búsqueda binaria.

Especificación 2: Ordenar de menor a mayor un vector de números enteros A.

Implementación: Ordenación rápida (quicksort), ordenación por mezclas (mergesort), ordenación por inserción (insertion sort)...

Usos de una especificación:

- Los **usuarios** del algoritmo:
  la especificación debe contener la información necesaria para utilizar el algoritmo.
- El **implementador** del algoritmo:
  - la especificación debe definir los requisitos que cualquier implementación válida debe satisfacer.
  - Ha de dejar suficiente **libertad** para que se elija la implementación que se estime más adecuada con los recursos disponibles.

Introducción

Propiedades de una buena especificación:

Precisión Ha de responder a cualquier pregunta sobre el uso del algoritmo sin tener que acudir a la implementación del mismo. El lenguaje natural no permite la precisión requerida si no es sacrificando la brevedad.

Brevedad Ha de ser significativamente más breve que el código que especifica.

Claridad Ha de transmitir la intuición de lo que se pretende. A veces el lenguaje formal ha de ser complementado con explicaciones informales.

Los lenguajes formales ofrecen a la vez precisión y brevedad. A veces deben ser complementado con explicaciones informales para obtener claridad.

Introducción

Ventajas de una **especificación formal** frente a una informal:

- Eliminan ambigüedades que el usuario y el implementador podrían resolver de formas distintas, dando lugar a errores de uso o de implementación que aparecerían en ejecución.
- Permiten realizar una verificación formal del algoritmo. Este tema se tratará en los capítulos **3** y **4**, y consiste en razonar sobre la corrección del algoritmo mediante el uso de la **lógica**.
- Permite generar automáticamente un conjunto de casos de prueba que permitirán probar la corrección de la implementación. La especificación formal se usa para **predecir** y **comprobar** el resultado esperado.

- Supongamos declarado el siguiente tipo de datos:

$$\textbf{typedef int } vect\ [1000]; \tag{1}$$

Especificación: dado un vector a de tipo $vect$ y un entero n, devolver un valor booleano b que indique si el valor de alguno de los elementos $a[0], \ldots, a[n-1]$ es igual a la suma de todos los elementos que le preceden en el vector.
Utilizaremos la siguiente cabecera sintáctica:

$$\textbf{fun } essuma\ (vect\ a,\ \textbf{int } n)\ \textbf{return } (\textbf{bool } b)$$

Introducción

Ambigüedades:

- No quedan claras todas las obligaciones del usuario: (1) ¿serían admisibles llamadas con valores negativos de n?; (2) ¿y con $n = 0$?; (3) ¿y con $n > 1000$?
- En caso afirmativo, ¿cuál ha de ser el valor devuelto por la función?
- Tampoco están claras las obligaciones del implementador. Por ejemplo, si $n \geq 1$ y $a[0] = 0$ la función, ¿ha de devolver **true** o **false**?

Tratar de explicar en lenguaje natural todas estas situaciones llevaría a una especificación más extensa que el propio código de la función *essuma*.

Especificación formal:

$$Q \equiv \{0 \leq n \leq 1000\}$$
fun $essuma\ (vect\ a,$ **int** $n)$ **return bool** b
$$R \equiv \{b = \exists w : 0 \leq w < n : a[w] = (\sum u : 0 \leq u < w : a[u])\}$$

No es ambigua:

1. No son correctas llamadas con valores negativos de n, ni con $n > 1000$;
2. sí son correctas llamadas con $n = 0$ y en ese caso ha de devolver **false**;
3. las llamadas con $n \geq 1$ y $a[0] = 0$ han de devolver $b = \mathbf{true}$.

Introducción

Técnica de especificación de algoritmos pre/post. Un algoritmo es una función de estados en estados.

$$Q \equiv \texttt{Precondicion}.$$
fun
$$R \equiv \texttt{Postcondicion}$$

- Precondición: Es un predicado lógico que describe los estados iniciales válidos. Las variables libres deben ser parámetros de entrada al algoritmo.
- Postcondición: Es un predicado lógico que describe los estados finales posibles del algoritmo. Indica la relación entre los parámetros de salida y el valor de retorno, con los parámetros de entrada. Las variables libres deben ser parámetros del algoritmos o el valor de retorno.

Si S comienza su ejecución en un estado descrito por Q, S termina y lo hace en un estado descrito por R.

Introducción

Sea la especificación:

$$Q \equiv \{0 \leq n \leq 1000\}$$
$$S : \textbf{fun}\ essuma\ (vect\ a,\ \textbf{int}\ n)\ \textbf{return bool}\ b$$
$$R \equiv \{b = \exists w : 0 \leq w < n : a[w] = (\sum u : 0 \leq u < w : a[u])\}$$

- S representa la función a especificar,
- Precondición: Q. Solo se garantiza que el algoritmo termina en el estado descrito por la postcondicion para valores de n mayores o iguales que 0 y menores o iguales que 1000.
- Postcondición: R. El resultado es un valor booleano que indica si existe un índice del vector tal que.....

Predicados

Sintaxis de la lógica de primer orden que utilizamos:

Un predicado es una expresión e de tipo **bool**.

Si P y Q son predicados, también lo son:

1. $\neg P$, (no P).
2. $P \wedge Q$, (P y Q).
3. $P \vee Q$, (P o Q).
4. $P \rightarrow Q$, (P entonces Q).
5. $P \leftrightarrow Q$, (P si y solo si Q).
6. Cuantificación universal: $(\forall w : Q(w) : P(w))$, (para todo valor perteneciente al rango Q se cumple P).
7. Cuantificación existencial: $(\exists w : Q(w) : P(w))$, (existe un valor perteneciente al rango Q para el que se cumple P) . Estos dos predicados en la lógica de primer orden equivalen a : $(\forall w \,.\, Q(w) \rightarrow P(w))$ y $(\exists w \,.\, Q(w) \wedge P(w))$.

- Eejemplos de predicados:

$n \geq 0$

$x > 0 \wedge x \neq y$

$0 \leq i < n$ {abreviatura de $0 \leq i \wedge i < n$}

$\forall w : 0 \leq w < n : a[w] \geq 0$

$\exists w : 0 \leq w < n : a[w] = x$

$\forall w : 0 \leq w < n : impar(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u)$

donde $impar(x)$ es un predicado que dice si su argumento x es impar o no.

- Es **muy importante** distinguir los dos tipos de variables que pueden aparecer en un predicado:

  Variables ligadas Son las que están afectadas por un cuantificador, es decir se encuentran dentro de su ámbito.

  Variables libres Son las variables que no se encuentran afectadas por ningún cuantificador.

- Por ejemplo en el predicado
  $\forall w : 0 \leq w < n : impar(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u)$,
  - las variables **libres** son n y a,
  - las variables **ligadas** son w y u.

  Cuando se anidan cuantificadores las variables están ligadas al cuantificador más interno.

- Una variable ligada se puede **renombrar** de forma consistente sin cambiar el significado del predicado. Formalmente, si v no aparece en Q ni en P, entonces:

$$(\partial\, w : Q(w) : P(w)) \equiv (\partial\, v : Q(v) : P(v))$$

  donde ∂ representa un cuantificador cualquiera (por ahora hemos presentado $\forall$ y $\exists$, pero aparecerán más). Por ejemplo:

$$(\exists w : 0 \leq w < n : a[w] = x) \equiv (\exists v : 0 \leq v < n : a[v] = x)$$

- Durante este curso, para evitar confusión, las variables ligadas de un predicado **nunca tendrán el mismo nombre** que una variable del programa que esta siendo especificado.
- Siempre que sea posible, usaremos las letras $u, v, w, \ldots$ para nombrar variables ligadas.

- Otras expresiones cuantificadas que utilizaremos.
- Son expresiones **de tipo entero** en lugar de booleano.

$\sum w : Q(w) : e(w)$	suma de las $e(w)$ tales que $Q(w)$
$\prod w : Q(w) : e(w)$	producto de las $e(w)$ tales que $Q(w)$
$\text{máx}\, w : Q(w) : e(w)$	máximo de las $e(w)$ tales que $Q(w)$
$\text{mín}\, w : Q(w) : e(w)$	mínimo de las $e(w)$ tales que $Q(w)$
$\# w : Q(w) : P(w)$	De los valores que cumplen $R(w)$ cuantos verifican $P(w)$

donde Q y P son predicados y e es una expresión entera:

Significado

El **estado de ejecución** de un algoritmo es una asociación de las variables del algoritmo con valores compatibles con su tipo.
Ejemplo Linear Search

```
1 bool LinearSearch (int a[], int n, int x)
2 {
3     int i = 0;
4     while (i < n && a[i] != x) i++;
5     return i < n;
6 }
```

LLamada al algoritmo: `LinearSearch(a,5,3)` Siendo el vector $a = \{2, 5, 6, 1, 3\}$
Estado después de ejecutar la linea 3:
$a = \{2, 5, 6, 1, 3\}$; $n = 5$; $x = 3$; $i = 0$

Significado

- Utilizamos predicados para definir **estados**.
- Un estado σ **satisface** un predicado P si al sustituir en P las variables libres por los valores que esas variables tienen en σ, el predicado se evalúa a **true**.
- Al utilizar variables en los predicados podemos definir conjuntos de estados.
  Por ejemplo:
  - $Q \equiv \{a = \{2, 5, 6, 1, 3\} \wedge n = 5 \wedge x = 3$
  - $Q :\equiv \{a = A \wedge n > 0\}$
  - $Q :\equiv \{\forall i : 0 \leq i \leq n : a[i] \leq 0 \wedge n \geq 0 \wedge x \leq 0\}$
- **Identificaremos** un predicado con el **conjunto de estados que lo satisfacen**. Puede ser infinito o vacío.

Ejemplos de Predicados

- Sean x e y las únicas variables del algoritmo,
  1. $P \equiv y - x > 0$ define los infinitos pares (x, y) en los y es mayor que x (es decir el octante encima de la diagonal principal del plano),
  2. $Q \equiv x \ \mathbf{mod}\ 2 = 0$ define todos los pares (x, y) en los x es un número par.
  3. **true** define el conjunto de **todos** los estados posibles (i.e. cualquier variable del algoritmo puede tener cualquier valor de su tipo),
  4. **false** define el conjunto **vacío**.

- El significado del predicado $\forall w : Q(w) : P(w)$ es equivalente al de la **conjunción** $P(w_1) \wedge P(w_2) \wedge \ldots$, donde $w_1, w_2, \ldots$ son todos los valores de w que hacen cierto $Q(w)$. Si este conjunto es **vacío**, entonces $\forall w : Q(w) : P(w) \equiv \mathbf{true}$.
- El significado del predicado $\exists w : Q(w) : P(w)$ es equivalente al de la **disyunción** $P(w_1) \vee P(w_2) \vee \ldots$, donde $w_1, w_2, \ldots$ son todos los valores de w que hacen cierto $Q(w)$. Si este conjunto es **vacío**, entonces $\exists w : Q(w) : P(w) \equiv \mathbf{false}$.
- Hay predicados que se satisfacen en todos los estados (e.g. $x > 0 \vee x \leq 0$). Equivalen a **true**. Hay otros que no se satisfacen en ninguno (e.g. $\exists w : 0 \leq w < 1 : a[w] = 8$). Equivalen a **false**.

- Por definición, el significado del resto de los cuantificadores cuando el rango $Q(w)$ al que se extiende la variable cuantificada es **vacío**, es el siguiente:

$$
\begin{array}{ll}
(\sum w : \mathbf{false} : e(w)) = 0 & \\
(\prod w : \mathbf{false} : e(w)) = 1 & \\
(\text{máx}\, w : \mathbf{false} : e(w)) & \text{indefinido} \\
(\text{mín}\, w : \mathbf{false} : e(w)) & \text{indefinido} \\
(\#\ w : \mathbf{false} : P(w)) = 0 &
\end{array}
$$

- Diremos que un predicado P es **más fuerte** (resp. **más débil**) que otro Q, y lo expresaremos $P \Rightarrow Q$ (resp. $P \Leftarrow Q$), cuando en términos de estados se cumpla $P \subseteq Q$ (resp. $P \supseteq Q$), es decir todo estado que satisface P también satisface Q (resp. todo estado que satisface Q también satisface P).

$$
\begin{array}{lcl}
x > 0 & \Rightarrow & x \geq 0 \\
P \wedge Q & \Rightarrow & P \\
P \wedge Q & \Rightarrow & Q \\
P & \Rightarrow & P \vee Q \\
\forall w : 0 \leq w < 10 : a[w] \neq 0 & \Rightarrow & a[3] \neq 0
\end{array}
$$

- El predicado más fuerte posible es **false**, es decir para cualquier predicado P, $\mathbf{false} \Rightarrow P$. Simétricamente, el predicado más débil posible es **true**, es decir para cualquier predicado P, $P \Rightarrow \mathbf{true}$.
- La relación "ser más fuerte que" coincide con la noción lógica de **deducción**. Leeremos con frecuencia $P \Rightarrow Q$ como "de P se deduce Q", o "P implica Q".
- Si $P \Rightarrow Q$ y $Q \Rightarrow P$, diremos que son igual de fuertes, o igual de débiles, o simplemente que son **equivalentes**, y lo expresaremos como $P \equiv Q$. Dos predicados equivalentes definen el mismo conjunto de estados.

Ejemplos de especificaciones

- Identificamos un algoritmo con una **función** en C++.
- La sintaxis utilizada en las especificaciones varía ligeramente de la sintaxis de C++ para ofrecer mas información.
- Los parámetros pueden ser de:

entrada	Su valor inicial es relevante para el algoritmo y éste **no debe** modificarlo.
salida	Su valor inicial es irrelevante para el algoritmo y éste **debe** almacenar algún valor en él.
entrada/salida	Su valor inicial es relevante para el algoritmo, y además este **puede** modificarlo.

- Tipos de **cabeceras** que utilizamos en la especificación:

  fun $nombre$ $(\mathbf{tipo}_1\ p_1, \ldots, \mathbf{tipo}_n\ p_n)$ **return** **tipo** r
  proc $nombre$ $(cualif\ \mathbf{tipo}_1\ p_1, \ldots, cualif\ \mathbf{tipo}_n\ p_n)$

  donde $cualif$ será vacío si el parámetro es de entrada, **out** si es de salida, o **inout** si es de entrada/salida. Los parámetros de una cabecera **fun** se entienden siempre de entrada.
- La primera se usará para algoritmos que devuelvan un solo valor, y la segunda para los que no devuelvan nada, devuelvan más de un valor, o/y modifiquen sus parámetros.
- El programador habrá de traducir la especificación a la cabecera de función C++ que le parezca más apropiada.

- Especificar un algoritmo que calcule el cociente por defecto y el resto de la división de naturales. Primer intento:

$$\{a \geq 0 \wedge b > 0\}$$
$$\textbf{proc}\ divide\ (\textbf{int}\ a,\ \textbf{int}\ b,\ \textbf{out int}\ q,\ \textbf{out int}\ r)$$
$$\{a = q \times b + r\}$$

- El especificador ha de imaginar que el implementador es un ser **malévolo** que trata de satisfacer la especificación del modo más simple posible, respetando la "letra" pero no el "espíritu" de la especificación.

- El siguiente programa sería correcto:

$$\{q = 0;\ r = a;\}$$

  El problema es que la postcondición es demasiado **débil**.

- Segundo intento:

$$\{a \geq 0 \wedge b > 0\}$$
$$\textbf{proc}\ \mathit{divide}\ (\textbf{int}\ a,\ \textbf{int}\ b,\ \textbf{out int}\ q,\ \textbf{out int}\ r)$$
$$\{a = q \times b + r \wedge 0 \leq r < b\}$$

  Por conocimientos elementales de matemáticas sabemos que sólo existen dos números naturales que satisfacen lo que exigimos a q y r.

- El máximo de las primeras n posiciones de un vector:

$\{n > 0 \wedge longitud(a) \geq n\}$
fun $maximo$ (**int** $a[\,]$, **int** n) **return int** m
$\{(\forall w : 0 \leq w < n : m \geq a[w]) \wedge (\exists w : 0 \leq w < n : m = a[w])\}$

  - La precondición $n > 0$ es necesaria para asegurar que el rango del existencial no es vacío. Una postcondición **false** solo la satisfacen los algoritmos que no terminan.
  - Nótese que la segunda conjunción de la postcondición es necesaria. Si no, el implementador malévolo podría devolver un numero muy grande pero no necesariamente en el vector. Una postcondición más sencilla sería.

$$\{m = \text{máx } w : 0 \leq w < n : a[w]\}$$

  - La segunda conjunción de la precondicion requiere que el vector parámetro real tenga una longitud de al menos n.

- Devolver un número primo mayor o igual que un cierto valor:

$$\{n > 1\}$$
$$\textbf{fun } unPrimo \ (\textbf{int } n) \textbf{ return int } p$$
$$\{p \geq n \wedge (\forall w : 1 < w < p : p \mod w \neq 0)\}$$

  - ¿Sería correcto devolver $p = 2$?
  - Nótese que la postcondición no determina un único p. El implementador tiene la libertad de devolver cualquier primo mayor o igual que n.

- Devolver el **menor** número primo mayor o igual que un cierto valor:

  $\{n > 1\}$
  fun $menorPrimo$ (**int** n) **return int** p
  $\{p \geq n \wedge primo(p) \wedge (\forall w : w \geq n \wedge primo(w) : p \leq w)\}$

  donde $primo(x) \equiv (\forall w : 1 < w < x : x \mod w \neq 0)$

  - Obsérvese que el uso del predicado auxiliar $primo(x)$ hace la especificación a la vez más modular y legible.
  - Nótese que un predicado **no** es una implementación. Por tanto no le son aplicables criterios de eficiencia: $primo(x)$ es una propiedad y **no** sugiere que la comprobación haya de hacerse dividiendo por todos los números menores que x.
  - La postcondición podría expresarse de un modo más conciso:

    $$p = (\text{mín}\, w : w \geq n \wedge primo(w) : w)$$

    sin que de nuevo esta especificación sugiera una forma de implementación.

- Especificar un procedimiento que *positiviza* un vector. Ello consiste en reemplazar los valores negativos por ceros. Primer intento:

$$\{n \geq 0 \wedge longitud(a) = n\}$$
$$\textbf{proc } positivizar \ (\textbf{inout int } a[\,], \ \textbf{int } n)$$
$$\{\forall w : 0 \leq w < n : a[w] < 0 \rightarrow a[w] = 0\}$$

  Donde lo que conseguimos es una postcondición equivalente a **false**.

- Añadimos a la precondición la condición $a = A$ que nos sirve para poder dar un nombre al valor del vector a **antes** de ejecutar el procedimiento, ya que a en la postcondición se refiere a su valor **después** de ejecutarlo. Segundo intento:

$$\{n \geq 0 \wedge longitud(a) = n \wedge a = A\}$$
$$\textbf{proc } positivizar \ (\textbf{inout int } a[\,], \ \textbf{int } n)$$
$$\{\forall w : 0 \leq w < n : A[w] < 0 \rightarrow a[w] = 0\}$$

- El implementador malévolo podría modificar también el resto de valores. Tercer intento:

$$\{n \geq 0 \wedge longitud(a) = n \wedge a = A\}$$
$$\textbf{proc } positivizar \ (\textbf{inout int } a[\,], \ \textbf{int } n)$$
$$\{\forall w : 0 \leq w < n : (A[w] < 0 \rightarrow a[w] = 0) \wedge (A[w] \geq 0 \rightarrow a[w] = A[w])\}$$

Ejercicios

1. Representar gráficamente la relación $P \Rightarrow Q$ para el siguiente conjunto de predicados:
   1. $P_1 \equiv x > 0$
   2. $P_2 \equiv (x > 0) \wedge (y > 0)$
   3. $P_3 \equiv (x > 0) \vee (y > 0)$
   4. $P_4 \equiv y \geq 0$
   5. $P_5 \equiv (x \geq 0) \wedge (y \geq 0)$

   Indicar cuáles de dichos predicados son *incomparables* (P es incomparable con Q si $P \not\Rightarrow Q$ y $Q \not\Rightarrow P$).
2. Construir un predicado $ord(a, n)$ que exprese que el vector **int** $a[n]$ está ordenado crecientemente.

3. Generalizar el predicado anterior a otro $ord(a, i, j)$ que exprese que el subvector $a[i..j]$ de **int** $a[n]$ está ordenado crecientemente. ¿Tiene sentido $ord(a, 7, 6)$?
4. Construir un predicado $perm(a, b, n)$, donde a y b son vectores de longitud n, que exprese que el vector b contiene una permutación de los elementos de a.
5. Especificar un procedimiento o función que ordene un vector de longitud n crecientemente.
6. Especificar un procedimiento o función que sustituya en un vector **int** $a[n]$ todas las apariciones del valor x por el valor y.

7. Dada una colección de valores, se denomina *moda* al valor que más veces aparece repetido en dicha colección. Queremos especificar una función que, dado un vector $a[n]$ de enteros con $n \geq 1$, devuelva la moda del vector.
8. Un vector **int** $a[n]$, con $n \geq 0$, se dice que es *gaspariforme* si todas sus sumas parciales son no negativas y la suma total es igual a cero. Se llama suma parcial a toda suma $a[0] + \cdots + a[i]$, con $0 \leq i < n$. Especificar una función que, dados a y n, decida si a es o no gaspariforme. ¿Qué debe devolver la función cuando $n = 0$?
9. Especificar un algoritmo que calcule la imagen especular de un vector. Precisando, el valor que estaba en $a[0]$ pasa a estar en $a[n-1]$, el que estaba en $a[1]$ pasa a estar en $a[n-2]$, etc. Permitir el caso $n = 0$.

10. Dado un vector **int** $a[n]$, con $n \geq 0$, formalizar cada una de las siguientes afirmaciones:
    1. El vector a es estrictamente creciente.
    2. Todos los valores de a son distintos.
    3. Todos los valores de a son iguales.
    4. Si a contiene un 0, entonces a también contendrá un 1.
    5. No hay dos elementos contiguos de a que sean iguales.
    6. El máximo de a sólo aparece una vez en a.
    7. El resultado l es la longitud máxima de un segmento constante en a.
    8. Todos los valores de a son números primos.
    9. El número de elementos pares de a es igual al número de elementos impares.
    10. El resultado p es el producto de todos los valores positivos de a.
    11. El vector a contiene un cuadrado perfecto.

11. Especificar una función que dado un natural n, devuelva la raíz cuadrada entera de n.
12. Especificar una función que dados $a > 0$ y $b > 1$, devuelva el logaritmo entero en base b de a.
13. Dado un vector **int** $a[n]$, con $n \geq 1$, expresar en lenguaje natural las siguientes postcondiciones:
    1. $b = (\exists w : 0 \leq w < n : a[w] = 2 \times w)$
    2. $m = (\#\ w : 1 \leq 0 < n : a[w-1] < a[w])$
    3. $m = (\text{máx}\, u, v : 0 \leq u < v < n : a[u] + a[v])$
    4. $l = (\text{máx}\, u, v : 0 \leq u \leq v < n \wedge (\forall w : u \leq w \leq v : a[w] = 0) : v - u + 1)$
    5. $r = (\text{máx}\, u, v : 0 \leq u \leq v < n : (\sum w : u \leq w \leq v : a[w]))$
    6. $p = (\prod u, v :: 0 \leq u < v < n : a[v] - a[u])$

14. ([?]) Dado x un vector de enteros, r y m dos enteros, y $n \geq 1$ un natural, formalizar cada una de las siguientes afirmaciones:
    - $x[0..n)$ contiene el cuadrado de un número.
    - r es el producto de todos los elementos positivos en $x[0..n)$.
    - m es el resultado de sumar los valores en las posiciones pares de $x[0..n)$ y restar los valores en las posiciones impares.
    - r es el número de veces que m aparece en $x[0..n)$.
15. ([?]) Comparar la fuerza lógica de los siguiente pares de predicados:
    1. $P = x \geq 0, \quad Q = x \geq 0 \to y \geq 0.$
    2. $P = x \geq 0 \vee y \geq 0, \quad Q = x + y = 0.$
    3. $P = x < 0, \quad Q = x^2 + y^2 = 9.$
    4. $P = x \geq 1 \to x \geq 0, \quad Q = x \geq 1.$

16. ([?]) Especificar funciones que resuelvan los siguientes problemas:
    1. Decidir si un entero es el factorial de algún número natural.
    2. Calcular el número de ceros que aparecen en un vector de enteros.
    3. Calcular el producto escalar de dos vectores de reales.
    4. Calcular la posición del máximo del vector no vacío de enteros $v[0..n)$.
    5. Calcular la primera posición en la que aparece el máximo del vector no vacío $v[0..n)$.
    6. Calcular la moda de un vector no vacío de enteros $v[0..n)$.
17. ([?]) Especificar una función que, dado un vector de números enteros, devuelva un valor booleano indicando si el valor de alguno de los elementos del vector es igual a la suma de todos los elementos que le preceden en el vector.

18. ([?]) La fórmula de Taylor-Maclaurin proporciona la siguiente serie convergente para calcular el seno de un ángulo x expresado en radianes:

$$seno(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}\ x^{2n+1}.$$

Por esta razón, para aproximar el seno de x con un error menor que un número real positivo ϵ basta encontrar un número natural k tal que $\left|\frac{(-1)^k}{(2k+1)!}\ x^{2k+1}\right| < \epsilon$ y entonces

$$senoAprox(x) = \sum_{n=0}^{k-1} \frac{(-1)^n}{(2n+1)!}\ x^{2n+1}.$$

Especificar formalmente una función que, dados los valores de x y $\epsilon > 0$, calcule el valor aproximado del seno de x con un error menor que ϵ.

19. Diremos que una posición de un vector de enteros es *fija* si el valor contenido en dicha posición coincide con la posición. Especificar una función $hayFija$ que, dado un vector $v[0..n)$ de enteros estrictamente creciente, indique si hay alguna posición *fija* en el vector.