Capítulo 1

Especificación de algoritmos [1]

Las matemáticas son el alfabeto con el cual Dios ha escrito el Universo.

Galileo Galilei (1564-1642)

Resumen: En este tema se enseña a distinguir entre especificar e implementar algoritmos, se repasa la lógica de predicados, y se establecen convenios y notaciones para especificar funciones y procedimientos utilizando dicha lógica.

1. Introducción

⋆ **Especificar** un algoritmo consiste en contestar a la pregunta "**qué** hace el algoritmo", sin dar detalles sobre cómo consigue dicho efecto. Una actitud correcta consiste en contemplar el algoritmo como una **caja negra** de la que solo podemos observar sus entradas y sus salidas.

⋆ **Implementar**, por el contrario, consiste en contestar a este segundo aspecto, es decir "**cómo** se consigue la funcionalidad pretendida", suponiendo que dicha funcionalidad está perfectamente clara en la especificación.

⋆ Muchos programadores noveles tienen dificultades para distinguir entre ambas actividades, produciendo documentos donde se mezclan contínuamente los dos conceptos.

⋆ La especificación de un algoritmo tiene un doble destinatario:

- Los **usuarios** del algoritmo: debe recoger todo lo necesario para un uso correcto del mismo. En particular, ha de detallar las obligaciones del usuario al invocar dicho algoritmo y el resultado producido si es invocado correctamente.
- El **implementador** del algoritmo: define los requisitos que cualquier implementación válida debe satisfacer, es decir las obligaciones del implementador. Ha de dejar suficiente **libertad** para que éste elija la implementación que estime más adecuada con los recursos disponibles.

[1] Ricardo Peña es el autor principal de este tema.

⋆ Una vez establecida la especificación, los trabajos del usuario y del implementador pueden proseguir **por separado**. La especificación actúa pues como una **barrera** o **contrato** que permite independizar los razonamientos de corrección de los distintos componentes de un programa grande. Así, se descompone la tarea de razonar sobre el mismo en **unidades manejables** que corresponden a su estructura modular.

⋆ 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. Los lenguajes formales ofrecen a la vez precisión y brevedad.

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

⋆ En este capítulo se presenta una técnica de especificación formal de funciones y procedimientos basada en el uso de la **lógica de predicados**, también conocida como técnica **pre/post**. Más adelante se presentará otra técnica formal diferente, las llamadas **especificaciones algebraicas**, más apropiadas para especificar tipos de datos.

⋆ Las ventajas de una **especificación formal** frente a una informal son bastantes:

- **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.
- Son las únicas que permiten realizar una **verificación formal** del algoritmo. Este tema se tratará en los capítulos **3** y **4**, y consiste en esencia 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 en alguna parte el siguiente tipo de datos:

$$\textbf{typedef int } \textit{vect } [1000]; \tag{1.1}$$

Queremos una función que, dado un vector a de tipo *vect* y un entero n, devuelva 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:

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

⋆ Esta especificación informal presenta algunas 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*.

⋆ La siguiente especificación formal contesta a estas preguntas:

$$Q \equiv \{0 \leq n \leq 1000\}$$
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 son correctas llamadas con valores negativos de n, ni con $n > 1000$; sí son correctas llamadas con $n = 0$ y en ese caso ha de devolver **false**; las llamadas con $n \geq 1$ y $a[0] = 0$ han de devolver $b =$ **true**. Para saber por qué, seguir leyendo.

⋆ La técnica pre/post, debida a C.A.R. Hoare (1969), se basa en considerar que un algoritmo es una función de estados en estados: comienza su ejecución en un **estado inicial** válido descrito por el valor de los parámetros de entrada, y tras un cierto tiempo cuya duración no es relevante a efectos de la especificación, termina en un **estado final** en el que los parámetros de salida contienen los resultados esperados.

⋆ Si S representa la función a especificar, Q es un predicado que tiene como variables libres los **parámetros de entrada** de S, y R es un predicado que tiene como variables libres **los parámetros de entrada y de salida** de S, la notación

$$\{Q\}S\{R\}$$

es la **especificación formal** de S y ha de leerse: *"Si S comienza su ejecución en un estado descrito por Q, S termina y lo hace en un estado descrito por R".*

⋆ Q recibe el nombre de **precondición** y caracteriza el conjunto de estados iniciales para los que está garantizado que el algoritmo funciona correctamente. Describe las **obligaciones del usuario**. Si éste llama al algoritmo en un estado no definido por Q, no es posible afirmar qué sucederá.

⋆ R recibe el nombre de **postcondición** y caracteriza la relación entre cada estado inicial, supuesto éste válido, y el estado final correspondiente. Describe las **obligaciones del implementador**, supuesto que el usuario ha cumplido las suyas.

2. Predicados

⋆ Usaremos las letras mayúsculas P, Q, R, ..., para denotar predicados de la lógica de primer orden. La sintaxis se define inductivamente mediante las siguientes reglas:

1. Una expresión e de tipo **bool** es un predicado.
2. Si P es un predicado, $\neg P$, leído "no P", también lo es.
3. Si P y Q son predicados, P & Q también lo es, siendo & cualquiera de las conectivas lógicas $\{\wedge, \vee, \rightarrow, \leftrightarrow\}$, leídas respectivamente "y", "o", "entonces", "si y solo si".

4. Si P y Q son predicados, $(\forall w : Q(w) : P(w))$ y $(\exists w : Q(w) : P(w))$ también lo son, denominados respectivamente **cuantificación universal** y **cuantificación existencial**.

En lógica de primer orden, estas cuantificaciones se escriben con una sintaxis más simple (respectivamente $\forall w \,.\, R(w)$ y $\exists w \,.\, R(w)$, siendo R un predicado). La sintaxis edulcorada propuesta aquí se escribiría entonces $(\forall w \,.\, Q(w) \rightarrow P(w))$ y $(\exists w \,.\, Q(w) \wedge P(w))$ respectivamente. La hemos elegido porque facilitará la escritura de asertos sobre programas.

⋆ Algunos ejemplos de predicados:

$$
\begin{array}{ll}
n \geq 0 & \\
x > 0 \wedge x \neq y & \\
0 \leq i < n & \{\text{abreviatura de } \ 0 \leq i \wedge i < n\} \\
\forall w : 0 \leq w < n : a[w] \geq 0 & \\
\exists w : 0 \leq w < n : a[w] = x & \\
\forall w : 0 \leq w < n : \mathit{impar}(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u) &
\end{array}
$$

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

⋆ Es **muy importante** saber 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. Por ejemplo, el ámbito de $(\forall w : Q(w) : P(w))$ son los predicados Q y P. Toda aparición de w en ellos que no esté ligada a otro cuantificador más interno, está ligada al $\forall w$ externo.

Variables libres Son el resto de las variables que aparecen en el predicado. La intención es que estas variables se refieran a variables o parámetros del algoritmo que se está especificando.

⋆ En el predicado $\forall w : 0 \leq w < n : \mathit{impar}(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u)$, las variables n y a son **libres**, mientras que w y u son **ligadas**.

⋆ 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:

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

⋆ Dado el uso tan diferente de las variables libres y ligadas y que el nombre de estas últimas siempre se puede cambiar, utilizaremos en este curso la siguiente regla:

> *Las variables ligadas de un predicado* **nunca tendrán el mismo nombre** *que una variable del programa que esta siendo especificado o verificado. Siempre que sea posible, usaremos las letras* $u, v, w, \ldots$ *para nombrar variables ligadas.*

⋆ Utilizaremos frecuentemente otras expresiones cuantificadas, **de tipo entero** en lugar de booleano, que se han demostrado útiles en la escritura de predicados breves y legibles. Son las siguientes (Q y P son predicados y e es una expresión entera):

$\sum w : Q(w) : e(w)$	suma de las $e(w)$ tales que $Q(w)$
$\prod w : Q(w) : e(w)$	producto de las $e(w)$ tales que $Q(w)$
máx $w : Q(w) : e(w)$	máximo de las $e(w)$ tales que $Q(w)$
mín $w : Q(w) : e(w)$	mínimo de las $e(w)$ tales que $Q(w)$
$\# \; w : Q(w) : P(w)$	número de veces que se cumple $Q(w) \wedge P(w)$

3. Significado

⋆ En el contexto de este curso, utilizaremos predicados para definir **conjuntos de estados**.

⋆ Un **estado** es simplemente una asociación de las variables del algoritmo con valores compatibles con su tipo. Por ejemplo, si x e y son variables de tipo entero, $\sigma = \{x \mapsto 3,\ y \mapsto 7\}$ representa un estado en el que la variable x vale 3, y la variable y vale 7. Un estado modeliza intuitivamente una "fotografía" de las variables de un algoritmo en un instante concreto.

⋆ Diremos que 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**. Por ejemplo, el estado σ anterior satisface $P \equiv y - x > 0$, pero no $Q \equiv x \textbf{ mod } 2 = 0$.

⋆ Dado un predicado P, queda determinado con precisión el conjunto de todos los estados que satisfacen P. Adoptaremos entonces el punto de vista de **identificar** un predicado con el **conjunto de estados que lo satisfacen**. Este conjunto frecuentemente es infinito y a veces es vacío.

⋆ Suponiendo que x e y sean las únicas variables del algoritmo, $P \equiv y - x > 0$ define los infinitos pares (x, y) en los y es mayor que x (es decir el octante encima de la diagonal principal del plano), mientras que $Q \equiv x \textbf{ mod } 2 = 0$ define todos los pares (x, y) en los x es un número par.

⋆ En particular, el predicado **true** define el conjunto de **todos** los estados posibles (i.e. cualquier variable del algoritmo puede tener cualquier valor de su tipo), y el predicado **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 $\forall w : Q(w) : P(w) \equiv$ **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 $\exists w : Q(w) : P(w) \equiv$ **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. $\exists w : 0 < w < 1 : a[w] = 8$). 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:

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

⋆ 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, **false** $\Rightarrow P$. Simétricamente, el predicado más débil posible es **true**, es decir para cualquier predicado P, $P \Rightarrow$ **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.

4. Ejemplos de especificación

⋆ Identificaremos un algoritmo con la noción de **función** en C++. En este lenguaje, las funciones tienen un tipo que corresponde al del resultado devuelto. Si no devuelven ningún valor, se usa el tipo **void**. Los mecanismos de paso de parámetros distinguen entre **paso por valor**, **paso por referencia** y **paso por referencia constante**. En esencia especifican si los parámetros formales de la función llamada son disjuntos o son los mismos que los parámetros reales del llamante. Se trata de un aspecto más relacionado con la eficiencia que con el flujo de información.

⋆ A efectos de especificación es más ilustrativo saber si los parámetros son 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.

⋆ Por ello, usaremos en la especificación dos tipos de **cabeceras virtuales** que el programador habrá de traducir después a la cabecera de función C++ que le parezca más apropiada.

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

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.

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

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

⋆ 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:

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

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:

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

- 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:

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

- ¿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:

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

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

- Obsérvese que el uso del predicado auxiliar $\textit{primo}(x)$ 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: $\textit{primo}(x)$ 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:

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

  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:

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

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:

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

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

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

Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando el Capítulo 2 de (Peña, 2005) en el cual se han basado, si bien la notación para predicados es ligeramente diferente. También la Sección 1.1 de (Rodriguez Artalejo et al., 2011).

El Capítulo 1 de Martí Oliet et al. (2012) utiliza la misma notación de predicados empleada aquí y contiene numerosos ejemplos resueltos.

Ejercicios

1. Representar gráficamente la relación $P \Rightarrow Q$ para el siguiente conjunto de predicados:

   a) $P_1 \equiv x > 0$

   b) $P_2 \equiv (x > 0) \wedge (y > 0)$

   c) $P_3 \equiv (x > 0) \vee (y > 0)$

   d) $P_4 \equiv y \geq 0$

   e) $P_5 \equiv (x \geq 0) \wedge (y \geq 0)$

   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:

    a) El vector a es estrictamente creciente.

b) Todos los valores de a son distintos.

c) Todos los valores de a son iguales.

d) Si a contiene un 0, entonces a también contendrá un 1.

e) No hay dos elementos contiguos de a que sean iguales.

f) El máximo de a sólo aparece una vez en a.

g) El resultado l es la longitud máxima de un segmento constante en a.

h) Todos los valores de a son números primos.

i) El número de elementos pares de a es igual al número de elementos impares.

j) El resultado p es el producto de todos los valores positivos de a.

k) 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:

a) $b = (\exists w : 0 \leq w < n : a[w] = 2 \times w)$

b) $m = (\#\ w : 1 \leq w < n : a[w-1] < a[w])$

c) $m = (\text{máx}\, u, v : 0 \leq u < v < n : a[u] + a[v])$

d) $l = (\text{máx}\, u, v : 0 \leq u \leq v < n \wedge (\forall w : u \leq w \leq v : a[w] = 0) : v - u + 1)$

e) $r = (\text{máx}\, u, v : 0 \leq u \leq v < n : (\sum w : u \leq w \leq v : a[w]))$

f) $p = (\prod u, v : 0 \leq u < v < n : a[v] - a[u])$

14. (Martí Oliet et al. (2012)) 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. (Martí Oliet et al. (2012)) Comparar la fuerza lógica de los siguiente pares de predicados:

a) $P = x \geq 0, \quad Q = x \geq 0 \rightarrow y \geq 0.$

b) $P = x \geq 0 \vee y \geq 0, \quad Q = x + y = 0.$

c) $P = x < 0, \quad Q = x^2 + y^2 = 9.$

d) $P = x \geq 1 \rightarrow x \geq 0, \quad Q = x \geq 1.$

16. (Martí Oliet et al. (2012)) Especificar funciones que resuelvan los siguientes problemas:

a) Decidir si un entero es el factorial de algún número natural.

b) Calcular el número de ceros que aparecen en un vector de enteros.

c) Calcular el producto escalar de dos vectores de reales.

d) Calcular la posición del máximo del vector no vacío de enteros $v[0..n)$.

e) Calcular la primera posición en la que aparece el máximo del vector no vacío $v[0..n)$.

f) Calcular la moda de un vector no vacío de enteros $v[0..n)$.

17. (Martí Oliet et al. (2012)) 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. (Martí Oliet et al. (2012)) 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.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

Brassard, G. y Bratley, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.

Martí Oliet, N., Segura Díaz, C. M. y Verdejo López, J. A. *Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios*. Ibergarceta Publicaciones, 2012.

Martí Oliet, N., Ortega Mallén, Y. y Verdejo López, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.

Peña, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.

Rodriguez Artalejo, M., González Calero, P. A. y Gómez Martín, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.

Stroustrup, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201889544.