Tratamiento secuencial

Esquemas de recorrido y búsqueda

Jordi Àlvarez Canal
Xavier Burgués Illa

PID_00149891

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

En el módulo anterior hemos aprendido a especificar un problema, hemos introducido la noción de algoritmo que lo resuelve y hemos introducido el lenguaje algorítmico, que nosotros utilizaremos para formular algoritmos. Todavía no hemos hablado de cómo llegar desde la especificación de un problema hasta el algoritmo que lo resuelve. En este módulo nos ocuparemos de este tema. No introduciremos aquí ningún elemento nuevo del lenguaje algorítmico, sino que, utilizando los elementos que ya conocemos, proporcionaremos una metodología que nos permita diseñar algoritmos de la forma más sistemática posible.

Ved el módulo "Introducción a la algorítmica" de este curso.

La parte más importante que trabajaremos en este módulo será, sobre todo, el planteamiento de la solución, que, como hemos visto en el módulo "Introducción a la algorítmica", es una de las etapas más dificultosas en el diseño de un algoritmo.

Ciertamente, podríamos intentar diseñar algoritmos de un nivel de complejidad similar a los ejemplos que aparecen en el módulo 1. Sin embargo, las únicas herramientas de las que disponemos hasta el momento son nuestra capacidad creativa para combinar adecuadamente las construcciones del lenguaje algorítmico y nuestra experiencia (que, por el momento, es poca), como ayuda inestimable para la primera.

El estudio de un conjunto de casos concretos que incremente nuestra experiencia podría ser una solución, pero el aprendizaje por esta vía puede ser bastante costoso y, a un tiempo, incompleto. Por el contrario, en este curso proponemos una metodología lo más sistemática posible para diseñar algoritmos. A pesar de todo, igualmente seguiremos necesitando nuestra capacidad creativa; aunque una vez asimilada esta **metodología**, el agujero que debemos llenar con nuestra creatividad será bastante menor.

Esta metodología se basa en la aplicación de unos algoritmos genéricos que nos permitirán solucionar problemas que se puedan modelar utilizando una secuencia de elementos, algo que, como veremos, es bastante habitual. Estos algoritmos genéricos los denominaremos ***esquemas***, y funcionan como una especie de plantillas, de forma que su aplicación (que llamaremos ***refinamiento***) consistirá en sustituir alguna de sus partes por "trocitos" de algoritmo correspondientes al problema concreto que estamos intentando solucionar en aquel momento.

Objetivos

Después de haber estudiado este módulo, tendréis que haber alcanzado los siguientes objetivos:

1. Entender la noción de secuencia como elemento básico para el diseño de algoritmos.

2. Conocer los esquemas básicos de tratamiento secuencial y ser capaz de identificar los problemas a los que se pueden aplicar.

3. Dado un problema al que podemos aplicar los esquemas, saber averiguar la secuencia subyacente. En ocasiones, esto resulta bastante evidente, como veremos por ejemplo en el apartado correspondiente a las secuencias de entrada y salida. Sin embargo, otras veces no lo es tanto.

4. Saber elegir adecuadamente el esquema que hay que aplicar para resolver un problema concreto.

5. Obtener práctica refinando esquemas para problemas concretos. De este modo, una vez hayáis averiguado la secuencia y decidido el esquema que hay que aplicar, el refinamiento debe ser una tarea bastante mecánica.

6. Identificar los problemas en los que hay que hacer varios tratamientos secuenciales para obtener un algoritmo que los solucione. Ser capaces de combinar los esquemas adecuadamente para hacerlo con éxito.

1. Algoritmos y secuencias

Todos tenemos en mente el concepto de secuencia: un conjunto de elementos ordenados de una forma determinada. Por ejemplo, sabemos que la siguiente secuencia: <1, 2, 3, 5, 7, 11, 13, 17, 19> es la secuencia de los números primos entre 1 y 20. También podemos encontrar ejemplos de secuencia fuera del mundo de las matemáticas y de la algorítmica: los automóviles que se encuentran en una cinta mecanizada a la espera de ser pintados por un brazo robot también suponen en realidad una secuencia.

Sin embargo, ¿por qué es importante para nosotros el concepto de secuencia? Pues resulta que la inmensa mayoría de los algoritmos mínimamente complejos consiste en repetir un conjunto de cálculos en una secuencia de datos. Estos algoritmos pueden ser modelizados como algoritmos que tratan secuencias.

De este modo, por ejemplo, un posible algoritmo del brazo robot podría ser el siguiente:

- Al principio de la jornada de trabajo hace falta que algún operario lo ponga en marcha, lo cargue de pintura si no tiene la suficiente y compruebe que el brazo funciona correctamente.

- Una vez hecho esto, el brazo robot se pone a funcionar. Su trabajo es muy simple. Espera a que llegue un automóvil y lo pinta. Entonces se detiene y espera a que llegue el siguiente automóvil.

- Finalmente, cuando la secuencia de automóviles se termina, el brazo robot se para.

Fijémonos en que el brazo robot repite las mismas operaciones para cada automóvil. La cinta mecanizada se encarga de ir poniéndole los automóviles delante para que el brazo robot actúe. Entonces, la dificultad del algoritmo del brazo robot se limita a pintar un automóvil, mientras que la cinta mecanizada se encarga de gestionar la secuencia de automóviles.

Notad que esta organización supone dos grandes ventajas: en primer lugar, el algoritmo del brazo robot es más sencillo que si él mismo tuviese que gestionar la secuencia de automóviles (seguro que el brazo robot tendría que ser más complejo; por ejemplo, debería tener capacidad de movimiento). En segundo lugar, podemos volver a utilizar la cinta mecanizada para otras cosas distintas. Por ejemplo, junto al robot pintor podemos situar otro que ponga los intermitentes del lado izquierdo. La estructura secuencial de algoritmo es la misma, y

lo único que cambia es el tratamiento que le damos a cada automóvil (un robot pinta mientras el otro pone intermitentes).

Lo que pretendemos en este módulo es trasladar todo esto al mundo de la algorítmica. En los siguientes temas proporcionaremos algoritmos genéricos que nos permitirán trabajar con secuencias. Estos algoritmos genéricos se denominan *esquemas*, y llevarán a cabo un trabajo equivalente al de la cinta mecanizada.

Entonces, dado un problema que queremos solucionar, si somos capaces de plantear un algoritmo que lo solucione en términos de tratamiento de una secuencia, sólo tendremos que aplicar el esquema adecuado a partir de este planteamiento. Esto nos permitirá centrar nuestro trabajo creativo única y exclusivamente en dos puntos (de forma similar al problema de pintar automóviles):

a) El planteamiento inicial del algoritmo como un algoritmo de tratamiento secuencial.

b) El tratamiento de un elemento de la secuencia.

Este hecho simplificará de modo importante nuestro trabajo. Además, el hecho de partir de unos algoritmos genéricos que nos sirven para muchas situaciones y que ya sabemos que son correctos hará que equivocarnos sea bastante más difícil.

Antes de plantear los esquemas, veamos cómo podemos modelizar un algoritmo de tratamiento de una secuencia. Partamos de un ejemplo que ya conocemos: el cálculo del factorial. Recordemos su algoritmo:

```
algoritmo factorial
   var
      i, n, fact := entero;
   fvar
   n := leerEntero();
   { Pre: n = N y N ≥ 0 }
   fact := 1;
   i := 1;
   mientras i ≤ n hacer
      fact := fact * i;
      i := i + 1;
   fmientras
   { Post: n = N, N > 0, fact es el factorial de i – 1 e i = n + 1, por lo tanto, fact es el factorial de N }
   escribirEntero(fact);
falgoritmo
```

Recordad que ya hemos visto el diseño de este algoritmo en el módulo "Introducción a la algorítmica".

¿Cuál es aquí la secuencia de elementos tratados? La que está formada por los números enteros entre 1 y *n*. Notad, sin embargo, que mientras el algoritmo del brazo robot ya se encuentra con una secuencia predeterminada de automóviles (aquellos que están en la cinta mecanizada), el algoritmo del factorial genera por sí mismo la secuencia. Para hacerlo, hemos utilizado la variable *i*, que irá "visitando" cada uno de los elementos de la secuencia, de forma que:

- Al principio nos situamos en el primer elemento de la secuencia (*i* := 1).

- Al final del cuerpo del **mientras**, la instrucción i := i + 1 nos sirve para pasar del elemento de la secuencia que ya hemos tratado al siguiente elemento (que trataremos en la siguiente iteración si no hemos llegado ya al final de la secuencia). Así pues, con esta instrucción pasamos del primer elemento (la variable *i* tiene por valor 1) al segundo elemento (*i* pasa a valer 2). Posteriormente pasaremos del segundo elemento al tercero (*i* vale 3). Y así hasta llegar al final de la secuencia.

- Finalmente, la condición del **mientras** ($i \leq n$) será verdadera mientras la variable *i* se corresponda a un elemento de la secuencia que hay que tratar. La condición será verdadera para todos los valores de *i* entre 1 y *n*; es decir, para todos los elementos de la secuencia.

Notad que...

... *i* llega a valer $n + 1$. De todos modos, la condición del **mientras** evalúa a **falso** para este valor, ya que no es un elemento de la secuencia que hay que tratar. Por lo tanto, el último elemento tratado se corresponde a *n*.

¿Y cuál es el tratamiento que le damos a los elementos de la secuencia? En este caso consiste en calcular el producto de todos ellos. Por eso nos hace falta una variable (*fact*) en la que vayamos acumulando todos los productos intermedios.

Podemos ver que realizamos dos operaciones diferentes con *fact*:

- En primer lugar, inicializamos *fact* a 1. Esto es necesario ya que, como nos sucederá en la inmensa mayoría de los casos, el tratamiento es acumulativo y necesitamos partir de unos valores iniciales (en este caso, damos a *fact* el valor del elemento neutro del producto, que nos permitirá ir "acumulando" los elementos que hay que tratar). Esta parte se corresponde a lo que llamaremos *inicializar el tratamiento*.

- En segundo lugar, ya dentro del cuerpo del **mientras**, multiplicamos *fact* por *i*; es decir: tratamos el elemento actual (correspondiente al valor de la *i*). Finalmente, cuando los hayamos tratado todos, *fact* contendrá el factorial de *n*, que está definido como el producto de los números enteros entre 1 y *n* (siendo *n* mayor que 0).

Fijaos en que la forma de trabajar es la misma que habíamos visto al principio con el brazo robot:

1) Hay un conjunto de acciones correspondientes al inicio del tratamiento y la preparación de la secuencia.

2) Hay otro grupo de acciones (en el caso del factorial, es una única acción) que nos sirve para tratar un elemento de la secuencia.

3) Una vez tratado un elemento, se pasa al siguiente elemento de la secuencia (o bien llegamos al final de la secuencia si no quedan elementos por tratar).

Debemos tener en cuenta, por un lado, que los elementos se tratan de forma ordenada (en primer lugar, se trata el primer elemento de la secuencia, después el segundo, el tercero, etc. hasta llegar al último). Por otro lado, y esto es bas-

tante importante, sólo tenemos acceso a un elemento de la secuencia a la vez. Este elemento es el mismo al que hemos hecho referencia en párrafos anteriores con el nombre de *elemento actual*.

Notad que tanto en el algoritmo del factorial como en el ejemplo del brazo robot los elementos de la secuencia son todos del mismo tipo. De este modo, en el ejemplo del brazo robot, los elementos son automóviles; y en el del factorial son números enteros. Esta propiedad será una constante a lo largo de todos los problemas de tratamiento secuencial.

Hasta ahora habíamos utilizado el término secuencia como "tira" de elementos. Sin embargo, para describir el funcionamiento de un algoritmo en términos de secuencias, es importante poder hablar del elemento que está "visitando" el algoritmo en un momento determinado (el elemento actual).

Por ello, introducimos la noción de **cabezal**, que nos indica la posición de la secuencia en la que está situado un algoritmo en un momento dado. Es necesario notar que el cabezal no se corresponde a ninguna construcción del lenguaje algorítmico. Simplemente es una noción que nos sirve de ayuda para hablar con mayor facilidad del elemento de la secuencia que estamos tratando en un momento dado.

Se trata justamente...

... de la misma idea de los cabezales de los reproductores de cintas magnéticas (video, casete, etc.); el cabezal está situado en el fragmento de cinta que estamos viendo, escuchando, etc. en un momento determinado.

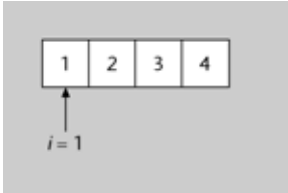
El cabezal puede estar situado encima de cualquier elemento de la secuencia, con las restricciones que ya hemos comentado anteriormente: sólo un elemento de la secuencia está disponible a la vez, y los elementos son visitados secuencialmente (en primer lugar el primero, después el segundo, el tercero, etc.), y de aquí el nombre secuencia. Esto implica que una vez hechas las inicializaciones que sitúan el cabezal en el primer elemento, éste sólo puede avanzar hacia adelante, posición a posición.

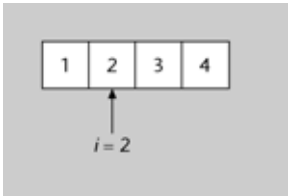
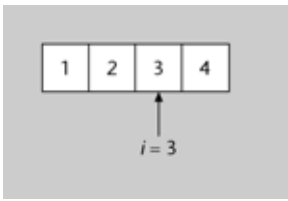
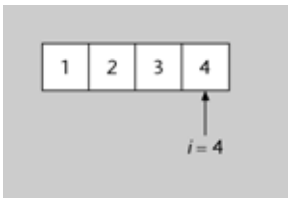
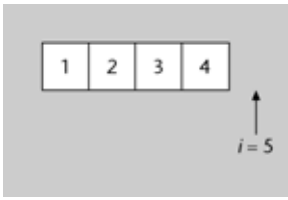
Así pues, a partir de ahora hablaremos muchas veces de la *parte de la secuencia que está a la izquierda del cabezal*, refiriéndonos a la parte correspondiente a aquellos elementos por los que el cabezal ya ha pasado. Y también hablaremos de la *parte de la secuencia que está a la derecha del cabezal*, refiriéndonos a aquellos elementos por los cuales el cabezal todavía no ha pasado (incluyendo el elemento donde está situado el cabezal).

Por otro lado, el cabezal tiene una posición extra al final de la secuencia, más allá de su último elemento. Cuando el cabezal se encuentra en esta posición, ya habremos tratado todos los elementos de la secuencia. En el caso del algoritmo del factorial, se llega a esta posición extra cuando i vale $n + 1$, que recordemos que no forma parte de la secuencia que hay que tratar.

Veamos gráficamente un ejemplo de ejecución del algoritmo del factorial para $n = 4$. En esta figura podemos observar cómo va progresando el cabezal (representado como una flechita) por la secuencia a medida que el algoritmo también avanza.

Algoritmo	Estado	Secuencia y cabezal	Variables del tratamiento

i :=1;
fact :=1;
→ **mientras** i ≤ n **hacer**
	Después de las inicializaciones
	(el cabezal apunta al primer elemento)
			*fact* = 1

mientras i ≤ n **hacer** → fact := fact ∗ i; i := i + 1 **fmientras**	Después de tratar al primer elemento (el cabezal apunta al segundo elemento)		*fact* = 1
	Después de tratar al segundo elemento (el cabezal apunta al tercer elemento)		*fact* = 2
	Después de tratar al tercer elemento (el cabezal apunta al cuarto elemento)		*fact* = 6
... **fmientras** → escribirEntero(fact);	Habiendo tratado toda la secuencia (el cabezal está en la posición extra de final de la sec.)		*fact* = 24

2. Esquema de recorrido de una secuencia

En el apartado anterior hemos visto que podemos modelizar un algoritmo sencillo que realiza cálculos repetitivos con un algoritmo de tratamiento de secuencias. A pesar de que el ejemplo utilizado (el factorial) es bastante sencillo, cualquier algoritmo que haga cálculos repetitivos puede ser modelizado como un algoritmo de tratamiento de secuencias.

Lo que pretendemos, sin embargo, es justamente el paso inverso: dado un problema, ser capaces de plantear un algoritmo que lo solucione en términos de tratamiento de una secuencia. Y entonces, a partir de este planteamiento, obtener sistemáticamente el algoritmo que soluciona el problema.

Para este último paso necesitamos el equivalente a la cinta mecanizada del brazo robot, pero en lenguaje algorítmico; es decir, los esquemas. La cinta mecanizada se va encargando de poner los automóviles delante del brazo del robot para que éste los pinte uno a uno. De la misma forma, los esquemas nos permitirán centrarnos en la parte del algoritmo correspondiente al tratamiento de un elemento de la secuencia.

En este apartado planteamos el esquema más general de todos los que veremos: el de recorrido de una secuencia. Este esquema nos permitirá recorrer una secuencia de elementos e irlos tratando uno a uno. La secuencia no puede ser infinita, a pesar de que no es necesario conocer con anterioridad el número de elementos que contiene.

Posteriormente,...

... en el apartado de secuencias de entrada y salida veréis ejemplos de tratamiento de secuencias donde al principio no conoceremos su número de elementos.

2.1. Planteamiento del esquema

Entendemos un esquema como un algoritmo muy general que no está expresado directamente en lenguaje algorítmico. Un esquema es, de hecho, un "esqueleto de algoritmos", un tipo de "plantilla" que tiene algunas partes que nosotros tendremos que sustituir por acciones (posiblemente más de una) o expresiones del lenguaje algorítmico. Estas partes aparecen en el esquema en cursiva. Su sustitución para aplicar el esquema a un problema concreto se denomina *refinamiento*.

Veamos a continuación el esquema de recorrido:

algoritmo EsquemaRecorrido

prepararSecuencia
inicioTratamiento
{ El cabezal está al principio de la secuencia. No hemos tratado ningún elemento de ésta }

```
    mientras no finSecuencia hacer
      { Hemos tratado la parte de secuencia que está a la izquierda del ca-
      bezal. Además, el cabezal no está en la posición extra del final }
        tratarElemento
        avanzarSecuencia
      { Hemos tratado la parte de la secuencia que está a la izquierda del
      cabezal }
    fmientras
      { Hemos tratado toda la secuencia. El cabezal está al final de la se-
      cuencia }
    tratamientoFinal
falgoritmo
```

En el esquema hemos añadido de forma orientativa, con comentarios (entre llaves), la descripción de los estados en que se encuentra el algoritmo en diferentes momentos del proceso. Estas descripciones son bastante generales y equivalen a la especificación del esquema; evidentemente, se pueden concretar para cada problema específico al que apliquemos el esquema.

2.2. Refinamiento

Las partes del esquema que deben ser sustituidas cuando lo refinamos para resolver un problema concreto son las siguientes:

- *prepararSecuencia*: en el caso en que la secuencia tenga al menos un elemento, debemos obtener el primero. Si la secuencia no tiene ningún elemento, situamos el cabezal en la posición extra del final (y entonces, *finSecuencia* será **cierto** de entrada). En el caso del factorial, el *prepararSecuencia* corresponde a inicializar i a 1. Notad que esto nos sirve tanto en el caso de que la secuencia tenga elementos (si calculamos el factorial de un número mayor o igual que 1), como en el caso de que no los tenga (si calculamos el factorial de 0, que por definición es 1). En este último caso, el cabezal ya estará situado de entrada en la posición extra del final de la secuencia, y el *finSecuencia* (ver más adelante) será **cierto** también de entrada.

- *inicioTratamiento*: equivale al conjunto de acciones de inicialización con las que damos un valor inicial a las variables que nos servirán para hacer los cálculos correspondientes al tratamiento. En el caso del factorial, esto se corresponde a inicializar *fact* a 1.

- *finSecuencia*: corresponde a una expresión que es cierta cuando el cabezal está en la posición especial que hay más allá del último elemento de la secuencia; es decir, al final de la secuencia. En el caso del factorial, esta expresión se corresponde a $i > n$, que negada nos queda $i \leq n$ (notad que la

condición del mientras es *no finSecuencia*), que es lo que aparece en el algoritmo. Fijaos en que cuando *n* es 0 (la secuencia no tiene elementos) *finSecuencia* ya es cierto la primera vez que se evalúa.

- *tratarElemento*: corresponde al conjunto de acciones que llevan a cabo el tratamiento de un elemento de la secuencia. En el caso del factorial, esto consiste en guardar en *fact* el producto del mismo *fact* por el elemento tratado.

- *avanzarSecuencia*: corresponde a un conjunto de acciones que avanzan el cabezal una posición. Es decir, o bien pasan al siguiente elemento o, en el caso en que el cabezal apunte al último elemento de la secuencia, sitúan el cabezal al final de la secuencia (recordad que hay una posición especial más allá del último elemento). En el caso del factorial, dado que la secuencia tratada es una secuencia de enteros donde los elementos corresponden a los diferentes valores que va tomando la variable *i*, se incrementa el valor de *i* en 1.

Tened en cuenta que...

... al igual que en el caso del factorial, en muchas otras ocasiones, la obtención del siguiente elemento se hace a partir del anterior.

- *tratamientoFinal*: corresponde a un conjunto de acciones que nos permite obtener el resultado del algoritmo a partir del tratamiento realizado. En algunas ocasiones, este conjunto de acciones será vacío; es decir, el tratamiento realizado ya nos dará directamente el resultado obtenido y, por lo tanto, no será necesario efectuar tratamiento final. En el caso del factorial, no tenemos tratamiento final.

Como ejemplo...

... de algoritmo con tratamiento final, pensad por ejemplo en un algoritmo que calcula la media de una secuencia de números. Lo más directo sería calcular la suma de los números (al mismo tiempo que los contamos); y una vez hecho esto (que sería el tratamiento), dividir la suma por la cantidad de números (tratamiento final).

2.3. Especificación

Reflexionemos un poco ahora sobre la corrección del esquema que acabamos de presentar estudiando su especificación. Este apartado queda fuera del ámbito de la asignatura, pero os puede ayudar bastante en la asimilación del esquema de recorrido.

El invariante del **mientras** (y también del recorrido) corresponde a: "hemos tratado la parte de la secuencia que está a la izquierda del cabezal". El caso concreto del invariante antes de empezar a iterar es: "el cabezal está al principio de la secuencia"; y después de hacer todo el tratamiento: "hemos tratado toda la secuencia y el cabezal está al final", algo que nos debe permitir, junto con el tratamiento final, asegurar que la postcondición se cumple.

Al igual que las diferentes partes del esquema, el invariante también se debe refinar para cada problema concreto al que apliquemos el esquema de recorrido. Es decir, el invariante general que aparece en el esquema se debe concretar y debe describir el tratamiento realizado sobre la parte de la secuencia que está a la izquierda del cabezal.

De este modo, por ejemplo, en el caso del factorial, el invariante se concretaría como: "*fact* contiene el producto de los números tratados hasta el momento". Si quisiéramos concretar más aún, añadiendo el hecho de que *i* se corresponde

al elemento actual de la secuencia (al que apunta el cabezal) nos quedaría: "*fact* es el producto de los números enteros entre 1 e $i - 1$".

El invariante, junto con el hecho de que hemos llegado al final de la secuencia, nos debe permitir asegurar que después de la ejecución del tratamiento final, la postcondición del algoritmo es cierta. Del mismo modo, la precondición de nuestro algoritmo nos debe asegurar que, después de ejecutar el tratamiento inicial, se cumple el invariante.

Existen métodos formales que permiten verificar, haciendo uso del mismo algoritmo, que las descripciones de los diferentes estados son ciertas. También permiten desarrollar automáticamente invariantes a partir de la especificación de un problema; y desde aquí, generar un algoritmo que lo cumpla (y que solucione el problema), pero no los estudiaremos en esta asignatura.

Por otro lado, como hemos visto por encima en el módulo "Introducción a la algorítmica", garantizamos que el algoritmo acaba proporcionando una función de cota. En el caso del esquema de recorrido, la función de cota se puede hacer corresponder al número de elementos que quedan por tratar. Siempre que la secuencia sea finita, este número se va reduciendo hasta llegar a 0, algo que garantizará la finalización del algoritmo.

Éste es el motivo...

... por el que no es necesario saber cuántos elementos tiene la secuencia para garantizar la finalización del algoritmo. En cambio, sí es necesario asegurar que la secuencia es finita.

2.4. Metodología

La metodología que utilizaremos para diseñar algoritmos empleando esquemas de tratamiento secuencial es la siguiente:

1) Especificar el problema que queremos resolver.

2) Modelizar el problema como un problema de tratamiento secuencial. Descubrir cuál es la secuencia que hay que tratar para resolver el problema.

3) Elegir el esquema adecuado que debemos aplicar. En este momento, sólo conocemos uno, pero en el siguiente apartado estudiaremos el esquema de búsqueda. A veces este paso y el anterior pueden estar bastante relacionados.

4) Decidir el conjunto de variables que nos debe permitir solucionar el problema (es fácil que, aparte de las variables de entrada y de salida, nos hagan falta algunas más para efectuar los cálculos necesarios y para "visitar" los elementos de la secuencia). Este paso se relaciona con el siguiente, refinamiento del esquema. Algunas de las variables dependerán de cuál sea el tratamiento que debamos dar a los elementos de la secuencia.

5) Refinar el esquema. Consiste en sustituir cada una de las partes generales del esquema por conjuntos de acciones y expresiones del lenguaje algorítmico, de modo que obtengamos un algoritmo que solucione el problema concreto especificado en el primer paso.

De entrada...

... nos podría parecer que aplicar esta metodología para resolver problemas como el del factorial puede ser como matar moscas a cañonazos. La utilidad del uso de los esquemas se va haciendo más evidente a medida que la complejidad de los problemas que hay que resolver va aumentando.

Como ya habéis estudiado...

... en el módulo "Introducción a la programación", puede haber muchas formas de resolver un problema. Del mismo modo, también pueden existir muchas formas de modelizar un problema como tratamiento de una secuencia de elementos. Algunas de estas formas pueden complicar los pasos que van del 3 al 5.

Siguiendo este método para diseñar algoritmos, el paso creativo se reduce a descubrir la secuencia a tratar. Posteriormente, si la secuencia es la adecuada, refinar el esquema debe resultar algo bastante directo. Si no es así, es altamente probable que no hayamos elegido la secuencia adecuada en el paso número 2, y entonces el resto de pasos propuestos pueden resultar bastante más complicados (podéis ver un caso de este tipo en el ejemplo de los divisores, a continuación).

2.5. Ejemplos

2.5.1. Divisores de un número

Nos piden diseñar un algoritmo que, dado un número entero positivo (que leeremos del dispositivo de entrada), escriba en el dispositivo de salida sus divisores (sin contarlo ni a éste ni al 1).

De este modo, por ejemplo, si el número que leemos del dispositivo de entrada es el 28, el algoritmo deberá escribir en el dispositivo de salida los números 2, 4, 7 y 14. Si el número leído es el 23, el algoritmo no deberá escribir ningún número (el 23 es un número primo).

Veamos cómo utilizamos la metodología antes presentada para diseñar el algoritmo. En primer lugar, especificamos el algoritmo:

n: **entero**
{ Pre: $n = N$ **y** $N \geq 0$ }
divisores
{ Post: en el dispositivo de salida se ha escrito la secuencia de divisores de N }

Ahora planteamos nuestro algoritmo. En primer lugar, averiguamos cuál es la secuencia que debemos tratar. Puesto que el problema habla de divisores, una secuencia que parece adecuada es precisamente la secuencia de divisores del número. De este modo, para el número 14, la secuencia que hay que tratar sería: <2, 7, 14>.

En el apartado 4 de este módulo veréis secuencias que ya nos vendrán predeterminadas y que no deberemos generar nosotros mismos en el algoritmo.

Tened en cuenta que, al igual que en el algoritmo del factorial (y también en los demás ejemplos de este apartado y del siguiente), los elementos de la secuencia no los tenemos guardados en ningún lugar, y los debemos generar nosotros mismos en el propio algoritmo. Esta generación se lleva a cabo en la parte correspondiente al *avanzarSecuencia*, donde debemos obtener el siguiente elemento de la secuencia a partir del elemento actual.

Sin embargo, antes de ir más allá pensemos cómo refinaríamos el esquema. El *tratarElemento* es bastante sencillo y evidente: consiste únicamente en escribir el elemento actual (que será uno de los divisores) por el dispositivo de salida.

En cambio, pensemos en el *avanzarSecuencia*. Esta parte del esquema, dado un divisor del número en cuestión, debe encontrar (o generar) el siguiente divisor. No se trata de una tarea fácil, pues no es posible hacerla de forma sencilla

y directa con una o varias operaciones (tal y como hacíamos con el factorial para pasar de *i* a *i + 1*).

De hecho, para encontrar el siguiente divisor de un número dado otro divisor (correspondiente al elemento actual, el apuntado por el cabezal) deberíamos aplicar un esquema de tratamiento secuencial. En este caso, lo más adecuado sería el esquema de búsqueda, que veremos en el siguiente apartado, refinado para encontrar el próximo divisor. Notad que finalmente estaríamos aplicando dos esquemas diferentes, uno dentro del otro.

Podríamos solucionar el problema con este modelo de secuencia. Pero también podemos modelizarlo como el tratamiento de una secuencia diferente que haga que el refinamiento del esquema sea bastante más sencillo y no nos haga falta el esquema de búsqueda.

Como podéis ver,...

... elegir una secuencia no adecuada puede complicar después el refinamiento de forma innecesaria. No os debe importar pasar más tiempo intentando averiguar cuál es la secuencia adecuada. Después, en el momento de refinar el esquema lo agradeceréis.

¿Cuál es, entonces, esta secuencia? Simplemente, la secuencia de números enteros entre 2 y $N - 1$. Ahora, por ejemplo, para el número 14 la secuencia que hay que tratar será <2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 >.

En este caso, encontrar el siguiente elemento de la secuencia es trivial (de hecho, es lo mismo que en el algoritmo del factorial). En cambio, *tratarElemento* es un poco más complicado que con la secuencia de divisores, ya que debemos comprobar si el elemento actual es divisor de *N*, y sólo en este caso deberemos escribirlo por el dispositivo de salida. Pero esto corresponde a un simple condicional que compruebe si el resto de dividir *N* por el elemento actual es 0 (operación módulo).

Por lo tanto, en conjunto, esta solución es bastante más sencilla. Entonces, solucionemos el problema con este segundo modelo.

En primer lugar, nos hará falta una variable que recorra los distintos elementos de la secuencia (utilizaremos *i*, igual que en el factorial), aparte de la variable de entrada (utilizaremos *n*), que leeremos del dispositivo de entrada.

Pasemos a comentar cómo refinamos cada una de las partes del esquema:

Como hemos comentado...

... en la sección 2.2, el refinamiento del esquema también nos lleva a refinar el invariante del **mientras** que aparece en el esquema. En este caso, como el tratamiento de un elemento corresponde a escribir el número si es divisor y a no hacer nada si no lo es, el invariante seria: "Se han escrito por el dispositivo de salida los divisores de *N* menores que *i*".

- *prepararSecuencia*: al igual que en el caso del factorial, inicializamos la variable *i* en el primer elemento de la secuencia, que en este caso es 2.

- *inicioTratamiento*: no es necesario hacer nada (notad que, a diferencia del algoritmo del factorial, aquí no hacemos ningún cálculo acumulativo. En todo caso, la acumulación está en el dispositivo de salida. Pero eso es otra historia).

- *finSecuencia*: el cabezal estará en la posición extra cuando se cumpla $i = n$ (el último elemento de la secuencia es $n - 1$). Notad, sin embargo, que en los casos $n = 0$, $n = 1$ y $n = 2$, se trata de una secuencia vacía. También debemos tener en cuenta estos casos en la expresión *finSecuencia*. Para tener todos los casos en cuenta, *i* se debe iniciar en 2 y la condición *finsecuencia* debe ser $i \geq n$.

– *tratarElemento*: debemos comprobar si el elemento *i* es divisor de *n*; y en este caso escribirlo por el dispositivo de salida. Lo podemos hacer con el operador módulo:

```
si n mod i = 0 entonces
    escribirEntero(i)
fsi
```

– *avanzarSecuencia*: consiste únicamente en incrementar *i*.

– *tratamientoFinal*: no lo hay.

Tened en cuenta que...

... en muchos problemas (como por ejemplo éste), la secuencia que hay que tratar puede estar vacía. En este caso, la secuencia está constituida por los enteros entre 2 y $n-1$. Por lo tanto, para los valores $n=0$, $n=1$ y $n=2$ disponemos de una secuencia que no tiene ningún elemento (algo que denotaremos por secuencia vacía). El esquema se puede aplicar sin ninguna dificultad a problemas en los que se den casos en los que la secuencia a tratar no tenga ningún elemento.

Veamos cómo queda con todo esto el algoritmo:

```
algoritmo divisores

    var
        n: entero;
        i: entero;
    fvar

    n := leerEntero();
    { Pre: n = N y N ≥ 0 }
    i := 2;
    mientras i < n hacer
        { Se han escrito por el dispositivo de salida
         los divisores de N menores que i }
        si n mod i = 0 entonces
            escribirEntero(i)
        fsi
        i := i + 1
    fmientras
    { Post: se ha escrito por el dispositivo de salida
    la secuencia de todos los divisores de N }
falgoritmo
```

2.5.2. Suma de las cifras de un número

Queremos diseñar un algoritmo que, dado un número entero (que leeremos del dispositivo de entrada), escriba por el dispositivo de salida otro número entero correspondiente a la suma de las cifras del primero. Para simplificar, al igual que en el ejemplo anterior, nos limitaremos a números positivos.

De este modo, por ejemplo, si el número leído es el 7.463, el algoritmo deberá escribir 20 (7 + 4 + 6 + 3).

Vamos a solucionar el problema. En primer lugar debemos especificar el algoritmo:

```
n: entero
{ Pre: n = N y N ≥ 0 }
sumaDigitos
{ Post: por el dispositivo de salida se ha escrito la suma de los dígitos de N }
```

A continuación debemos encontrar cuál es la secuencia que trataremos. En este caso, la secuencia más adecuada parece la secuencia de dígitos del núme-

ro. Tenemos dos opciones: recorrer los dígitos de un número de izquierda a derecha o hacerlo de derecha a izquierda (para sumar los dígitos no importa cómo lo hagamos, ya que la suma es conmutativa y asociativa). Elegimos la segunda opción porque nos resultará un poco más sencillo recorrer los dígitos de menos significativo a más significativo (como ejercicio podéis intentar resolver el problema de la otra forma). Así, por ejemplo, la secuencia correspondiente al número 7.463 será: <3, 6, 4, 7>.

Para ir generando la secuencia de dígitos sólo tenemos que empezar con el número original e ir dividiéndolo por diez. Entonces, cada vez que queramos acceder a un dígito (que se corresponderá al dígito menos significativo) sólo tenemos que calcular el número módulo 10.

Utilizaremos una variable para guardar el dígito en cada iteración (por ejemplo, *d*) y otra para guardar la suma de los dígitos que ya hemos tratado (*suma*), aparte de la variable que leeremos de la entrada (utilizaremos *n*), y que iremos dividiendo por 10 a medida que vayamos avanzando el cabezal por la secuencia de dígitos.

Pasando ya al refinamiento, tenemos que:

– *prepararSecuencia*: el número *n* se corresponde en realidad a la secuencia de dígitos. El cabezal debe ubicarse en su dígito menos significativo. Esto lo podemos hacer así:

```
d := n mod 10;
```

– *inicioTratamiento*: consiste en inicializar la variable *suma* en 0 (el elemento neutro de la suma).

– *tratarElemento*: debemos sumar el dígito menos significativo (que tenemos en la variable *d*):

```
suma := suma + d;
```

– *avanzarSecuencia*: debemos pasar al siguiente dígito. Esto lo hacemos dividiendo *n* por 10. De esta forma eliminamos el dígito menos significativo de *n* de modo que el nuevo dígito menos significativo pasa a ser el segundo dígito menos significativo. Después, debemos encontrar el nuevo dígito menos significativo:

```
n := n div 10;
d := n mod 10;
```

– *finSecuencia*: habremos recorrido (y tratado) todos los dígitos del número inicial cuando el número *n* (que hemos ido dividiendo por 10) sea 0.

– *tratamientoFinal*: el algoritmo debe escribir la suma de los dígitos por el dispositivo de salida.

Con todo esto, el algoritmo queda como sigue:

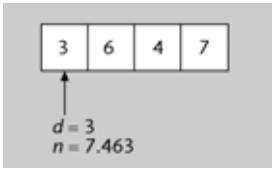
```
algoritmo sumaDigitos

    var
        n, suma, d: entero;
    fvar

    n := leerEntero();
    { Pre: n = N y N ≥ 0 }
    d := n mod 10;
    suma := 0;
    mientras no(n = 0) hacer
       { suma es la suma de dígitos que hay en N que ya han sido
         tratados. El número d es el dígito que se está tratando }
         suma := suma + d;
         n := n div 10;
         d := n mod 10
    fmientras
    { suma es la suma de dígitos que hay en N }
    escribirEntero(suma)
    { Post: por el dispositivo de salida se ha escrito
    la suma de los dígitos de N }
falgoritmo
```

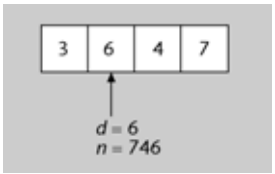
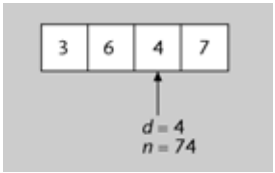
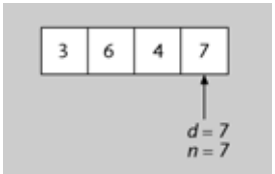
La condición del mientras,...

... en lugar de ser no ($n = 0$), podría ser perfectamente $n \neq 0$ (que es lo mismo); o bien $n > 0$.

En esta figura podemos observar gráficamente cómo se comporta el algoritmo para el caso concreto en el que $n = 7.463$.

Algoritmo	Estado	Secuencia y cabezal	Variables del tratamiento

d := n **mod** 10;
suma := 0;
→ **mientras no**(n = 0) **hacer**
	Después de *prepararSecuencia* e *inicioTratamiento*		*suma* = 0

mientras no(n = 0) **hacer** → suma := suma + d; n := n **div** 10; d := n **mod** 10 **fmientras**	Después de tratar el primer elemento		*suma* = 3
	Después de tratar el segundo elemento		*suma* = 9
	Después de tratar el tercer elemento		*suma* = 13
... → **fmientras** escribirEntero(suma);	Habiendo tratado toda la secuencia		*suma* = 20

Como vemos,...

... en este ejemplo hay dos variables que intervienen en la generación de la secuencia. En este caso nos podríamos haber ahorrado fácilmente la variable *d*.

Sin embargo, a medida que vayamos trabajando con algoritmos más complejos, es probable que nos encontremos frecuentemente con casos en los que nos haga falta más de una variable para generar la secuencia.

3. Esquema de búsqueda en una secuencia

3.1. Planteamiento del esquema

Introducimos en este apartado el segundo esquema de tratamiento secuencial de los que veremos en este curso: el esquema de búsqueda. Este esquema nos permitirá buscar un elemento en una secuencia. A diferencia del esquema de recorrido, donde recorremos y tratamos todos los elementos de la secuencia, con este nuevo esquema recorreremos la secuencia únicamente hasta que encontremos un elemento que cumpla una condición determinada. Esta condición no tiene por qué referirse a un elemento con un valor concreto (como por ejemplo, cuando queremos encontrar la primera letra 'a' en una frase), sino que puede ser una condición más general (encontrar el primer número par de una secuencia de enteros, la primera palabra capicúa de un texto, etc.).

El esquema de búsqueda es como sigue:

```
algoritmo EsquemaBusqueda

    var
      encontrado: booleano;
    fvar

    prepararSecuencia
    encontrado := falso;
    inicioTratamiento
    { El cabezal está al principio de la secuencia. No hemos tratado ningún elemento de ésta }
    mientras no finSecuencia y no encontrado hacer
        { Hemos tratado la parte de la secuencia que está a la izquierda del
        cabezal. El elemento buscado no está en esta parte de la secuencia.
        Además, el cabezal no está al final y encontrado es falso. }
        actualizarEncontrado
        si no encontrado entonces
            tratarElemento
            avanzarSecuencia
        fsi
        { Hemos tratado la parte de la secuencia que está a la izquierda del cabezal. El elemento buscado no está en esta parte de la secuencia. Si encontrado es cierto, el elemento actual es el elemento buscado. }
    fmientras
```

```
        { Hemos tratado la parte de la secuencia que está a la izquierda del
        cabezal. El elemento buscado no está en esta parte de la secuencia.
        Además, si encontrado es cierto, el cabezal apunta al elemento
        buscado; y si es falso, el cabezal está al final de la secuencia. }
        tratamientoFinal
falgoritmo
```

El esquema está preparado para el caso en que la secuencia no contenga el elemento buscado. En este caso, se llega al final de la secuencia y se acaba la ejecución del bloque **mientras** con la variable *encontrado* a **falso**.

En el caso de que la secuencia sí contenga el elemento buscado, el condicional interno del bloque **mientras** hace que el cabezal se sitúe sobre el elemento buscado (que no se trata). Se sale del bucle **mientras** con la variable *encontrado* con valor **cierto**.

Posteriormente, podemos utilizar el valor de *encontrado* en *tratamientoFinal* para determinar si hemos encontrado o no el elemento y emprender las acciones pertinentes en cada caso (como por ejemplo, tratar el elemento *encontrado*, si es necesario darle tratamiento).

Tal y como hemos hecho en el esquema de recorrido, en el esquema de búsqueda hemos añadido de forma orientativa la descripción de los estados en que se encuentra el algoritmo en diferentes momentos del proceso.

3.2. Refinamiento

Pasemos a comentar cada una de las partes del esquema, que igual que en el esquema de recorrido deben ser sustituidas al refinarlo:

- *prepararSecuencia*: del mismo modo que en el esquema de recorrido, si la secuencia tiene al menos un elemento (no está vacía), se procede a encontrar el primer elemento de la secuencia. Y si la secuencia está vacía, situamos el cabezal en la posición extra del final de la secuencia.

- *inicioTratamiento*: equivale a inicializar las variables correspondientes al tratamiento. Sólo es necesario en caso de que sea una búsqueda con tratamiento (muchas búsquedas tienen como único objetivo encontrar un elemento determinado, sin hacer ningún tratamiento a los elementos que se recorren).

- *finSecuencia*: corresponde a una expresión que evaluará a **cierto** cuando el cabezal esté al final de la secuencia.

- *actualizarEncontrar:* corresponde a un conjunto de acciones que tienen como objetivo determinar si el elemento actual se corresponde con el elemento bus-

cado o no. En caso de serlo, la variable *encontrado* debe tener como valor **cierto** después de ejecutar este conjunto de acciones. Y, en caso de no serlo, la variable *encontrado* debe tener como valor **falso**.

Es habitual que...

... en las búsquedas no nos haga falta tratamiento, a pesar de que, evidentemente, no siempre será así.

- *tratarElemento*: corresponde al conjunto de acciones que llevan a cabo el tratamiento de un elemento de la secuencia. En una búsqueda sin tratamiento, este conjunto de acciones estará vacío.

- *avanzarSecuencia*: corresponde a un conjunto de acciones que hacen avanzar el cabezal una posición.

- *tratamientoFinal*: corresponde a un conjunto de acciones que nos debe permitir obtener el resultado del algoritmo a partir de la búsqueda realizada.

3.3. Especificación

Volvamos otra vez a reflexionar sobre el comportamiento del esquema haciendo un estudio de su especificación.

Como antes,...

... este apartado queda fuera del ámbito de esta asignatura, pero os puede resultar bastante útil.

Las descripciones de los estados que aparecen en el esquema están totalmente marcadas por el invariante del **mientras** y de la búsqueda, que corresponde a: "Hemos tratado la parte de la secuencia que está a la izquierda del cabezal. El elemento buscado no está en esta parte de la secuencia. Si *encontrado* es cierto, el elemento actual es el elemento buscado."

El invariante, junto con la negación de la condición del **mientras**, nos debe permitir, después de ejecutar el tratamiento final, asegurar que la postcondición del algoritmo se cumple.

El invariante proporcionado es bastante general y se debe concretar para cada aplicación del esquema (igual que concretamos o refinamos los fragmentos del esquema en conjuntos de acciones o bien, expresiones del lenguaje algorítmico).

En cuanto a la función de cota, la podemos hacer corresponder con el número de elementos que quedan por tratar. Sin embargo, a diferencia del esquema de recorrido, este número no corresponde al número de elementos de la secuencia. Es posible que encontremos el elemento buscado, y en este caso el número de elementos tratados será inferior al número de elementos de la secuencia.

Entonces, lo único que hay que asegurar es que el número de elementos tratados es finito; para ello tenemos suficiente con asegurar que, o bien la secuencia es finita, o bien siempre encontraremos el elemento buscado. Es decir, podemos aplicar el esquema de búsqueda sobre secuencias infinitas siempre que estemos seguros de que encontraremos el elemento buscado. De este modo, por ejemplo, podríamos aplicar el esquema de búsqueda para encontrar un número primo que sea capicúa, siempre que el propio enunciado nos asegure su existencia (o bien nosotros lo hayamos deducido de otra forma).

3.4. Metodología

La metodología que debemos utilizar para aplicar el esquema de búsqueda es la misma que empleamos para el esquema de recorrido (ya ha sido introducida en el punto 2.4). Evidentemente, se da una pequeña diferencia en el paso número 5, donde los fragmentos de esquema que hay que refinar son diferentes en cada caso.

3.5. Ejemplos

3.5.1. Números de Fibonacci

Nos piden diseñar un algoritmo que determine si entre los 1.000 primeros números de Fibonacci hay algún número que acabe en 9. También nos piden que en caso de que el número exista, lo escribamos por el dispositivo de salida; y si no existe, que escribamos el entero negativo –1.

Los números de Fibonacci están definidos de la siguiente forma:

$fib_1 = 0$
$fib_2 = 1$
$fib_n = fib_{n-1} + fib_{n-2}$, para $n > 2$

Vamos a solucionar el problema. En primer lugar, como siempre, especificamos el algoritmo:

{ Pre: **cierto** }
buscarFibonacci9
{ Post: en el caso de que este número exista, por el dispositivo de salida se habrá escrito uno de los primeros 1.000 números de Fibonacci, que, además acaba en 9. Si este número no existe, se habrá escrito un –1 }

Notad que...

... este algoritmo no tiene ningún dato de entrada. Por lo tanto, en la precondición no es necesario establecer ninguna restricción sobre los datos de entrada. Por este motivo, la precondición siempre se cumplirá y, por lo tanto, corresponde a **cierto**.

Ahora planteamos el algoritmo en términos de secuencias. La secuencia de los 1.000 primeros números de Fibonacci; es decir: $<fib_1, fib_2, fib_3, ..., fib_{999}, fib_{1.000}>$. Fijémonos en que no tenemos suficiente con los valores de los números; necesitamos también los subíndices para saber cuándo hemos llegado al final de la secuencia (cuando el subíndice sea igual a 1.001, esto querrá decir que el cabezal está en la posición especial que sigue después del último elemento de la secuencia).

Recordad el ejemplo de los divisores visto en el apartado 2.5.1. En este apartado se dice que la secuencia de los divisores no es la más adecuada, ya que encontrar el siguiente divisor implica hacer una búsqueda (algo que nos podíamos ahorrar utilizando una simple secuencia de números enteros). En este caso nos sucede justo lo contrario: dados los dos números de Fibonacci anteriores, es fácil encontrar el siguiente. En cambio, dado un entero cualquiera, determinar si es el siguiente número de Fibonacci parece bastante complicado. Así pues, aquí la secuencia adecuada es la de los mismos números de Fibonacci.

Construir la secuencia de los números de Fibonacci no resulta nada complicado: los dos primeros los conocemos, el tercero lo podemos construir con la suma de los dos primeros, el cuarto como la suma del segundo y el tercero, etc., así hasta llegar a $fib_{1.000}$.

El enunciado nos pide que escribamos uno de estos números que acabe en 9. No nos pide ninguno en concreto, ni el primero, ni el último, ni ningún otro. Así pues, elegiremos el menor que podamos encontrar, ya que, si de forma natural recorreremos la secuencia de los números de Fibonacci de menor a mayor, será el primero que encontremos (si es que existe, claro).

Recordad que...

... al dar un algoritmo para resolver un problema, se trata, como siempre, de satisfacer la especificación. Si ésta nos ofrece varias posibilidades, nosotros somos libres de elegir la que más nos convenga (normalmente eligiremos aquella que consuma menos recursos, tanto en tiempo como en espacio de memoria).

Así pues, nuestro problema consistirá en buscar en la secuencia constituida por los 1.000 primeros números de Fibonacci el primero que acabe en 9. Debemos tener en cuenta que, de entrada, no sabemos si este número existe o no.

Utilizaremos tres variables para ir calculando los números de Fibonacci (*actual, siguiente* y *siguiente2*). Y además, nos hará falta una variable índice para saber cuándo hemos llegado ya a $fib_{1.000}$ (utilizaremos *i*, que estará asociada al índice del número de Fibonacci correspondiente a *actual*).

Pasemos ahora a refinar el esquema. Veamos a qué corresponde cada una de las partes:

Observad que...

... la secuencia es un poco más compleja que las que encontrábamos en los ejemplos que habíamos visto hasta ahora. Un elemento de la secuencia está asociado a los valores de dos variables: *actual* (que es el número de Fibonacci en cuestión) y el índice *i* (que es el subíndice correspondiente). Además, como para calcular un número de Fibonacci nos hacen falta los dos anteriores, debemos tener siempre calculados dos números de Fibonacci consecutivos a la vez (*actual* y *siguiente*).

– *prepararSecuencia*: corresponde a encontrar el primer elemento de la secuencia. Dado que para calcular un número de Fibonacci necesitamos los dos anteriores, inicializaremos *actual* con el primer número de Fibonacci y *siguiente* con el segundo. Al mismo tiempo, debemos mantener el índice *i* asociado a *actual* (es decir, lo inicializamos en 1, pues *actual* se corresponde a fib_1).

```
actual := 0;
siguiente := 1;
i := 1;
```

– *inicioTratamiento*: no hacemos ningún tratamiento; por lo tanto, esta parte del esquema no corresponde a ninguna acción.

– *finSecuencia*: habremos llegado al final de la secuencia cuando se cumpla $i = 1.001$; o también, $i > 1.000$.

– *actualizarEncontrado*: debemos comprobar si el elemento actual (indicado por la variable *actual*) acaba en 9, y en este caso poner *encontrado* en *verdadero*. Esto lo podemos hacer con un condicional, del siguiente modo:

```
si actual mod 10 = 9 entonces
    encontrado := cierto;
fsi
```

O también lo podemos hacer con una asignación así:

```
encontrado := actual mod 10 = 9;
```

– *tratarElemento*: dado que no hay tratamiento, se corresponde al conjunto de instrucciones vacío.

Notad que,...

... en realidad, el condicional no es equivalente a la asignación sin condicional. Mientras en el primero sólo se establece el valor de *encontrado* cuando el número acaba en 9, en el segundo se establece el valor de *encontrado* siempre (bien a **cierto** si el número acaba en 9, o bien a **falso** en caso contrario). Sin embargo, dado que cuando *encontrado* se hace **cierto** la búsqueda termina en ese preciso momento, las dos versiones nos sirven.

Tened en cuenta también que actual **mod** 10 = 9 es una expresión booleana que tiene como resultado **cierto** si actual **mod** 10 es 9, y **falso** en caso contrario.

– *avanzarSecuencia*: aquí avanzar es un poco más complicado que en los ejemplos que hemos visto hasta ahora. El motivo es que intervienen distintas variables a causa de que para calcular el siguiente número de Fibonacci nos hacen falta los anteriores. La forma de proceder será la siguiente: necesitamos hacer correr *actual* y *siguiente* una posición en la secuencia. Para hacerlo, debemos conocer el número de Fibonacci que sigue a *siguiente*. Lo calculamos y lo guardamos en *siguiente2*. Una vez hecho esto, ya podemos hacer correr *actual* y *siguiente* una posición; y al mismo tiempo, también debemos incrementar el índice. Veámoslo:

```
{ Calculamos el siguiente número de Fibonacci }
siguiente2 := actual + siguiente;
{ ahora ya podemos hacer correr actual y siguiente una posición }
actual := siguiente;
siguiente := siguiente2;
{ finalmente, actualizamos el índice de actual }
i := i + 1;
```

– *tratamientoFinal*: consiste en comprobar si la búsqueda ha tenido éxito, y en escribir entonces el número encontrado por el dispositivo de salida. En caso contrario, escribiríamos un –1.

```
si encontrado entonces
   escribirEntero(actual)
sino
   escribirEntero(–1)
fsi
```

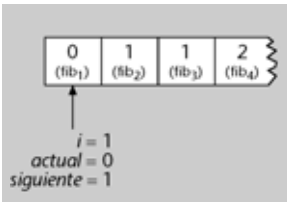
Veamos cómo queda el algoritmo juntando todos los fragmentos:

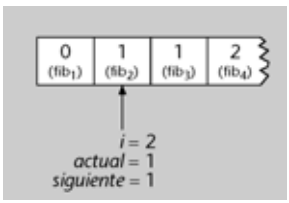
```
algoritmo buscarFibonacci9

  var
    actual, siguiente, siguiente2 : entero;
    i: entero;
    encontrado: booleano;
  fvar

  { Pre: cierto }
  encontrado := falso;
  actual := 0;
  siguiente := 1;
  i := 1;
  mientras i ≤ 1000 y no encontrado hacer
    encontrado := actual mod 10 = 9;
    si no encontrado entonces
      { calculamos el siguiente número de Fibonacci }
      siguiente2 := actual + siguiente;
      { ahora ya podemos hacer correr actual y siguiente una posición }
      actual := siguiente;
      siguiente := siguiente2;
      { finalmente, actualizamos el índice de actual }
      i := i + 1
    fsi
  fmientras
```

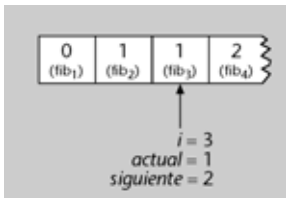
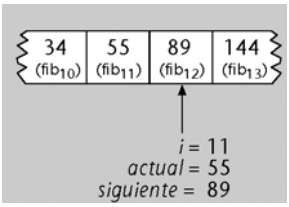
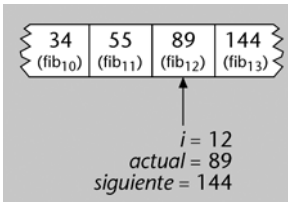
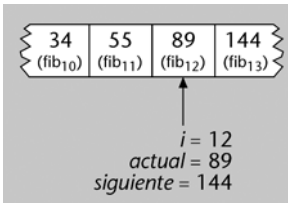
```
    si encontrado entonces
      escribirEntero(actual)
    sino
      escribirEntero(–1)
    fsi
    { Post: Por el dispositivo de salida se ha escrito uno de los primeros 1.000 números de
    Fibonacci que, además, acaba en 9, en el caso de que este número exista. Si este número
    no existe, se ha escrito un –1 }
falgoritmo
```

A continuación podemos ver de forma gráfica cómo se va realizando la búsqueda:

Algoritmo	Estado	Secuencia y cabezal

encontrado := **falso**;
actual := 0;
siguiente := 1;
i := 1;
mientras i ≤ 1.000 **y no** encontrado **hacer**
...
	Después de *prepararSecuencia* e *inicioTratamiento*	

mientras i ≤ 1.000 **y no** encontrado **hacer**
encontrado := actual **mod** 10 = 9;
si no encontrado **entonces**
siguiente2 := actual + siguiente;
actual := siguiente;
siguiente := siguiente2;
i := i + 1
fsi
fmientras
	Después de descartar el primer elemento	

	Después de descartar el segundo elemento	
	...	.....
	Después de descartar el decimoséptimo elemento	
	Después de descartar el decimoctavo elemento (*actualizarEcontrado* pone *encontrado* a **cierto**)	
... **fmientras** **si** encontrado **entonces** escribirEntero(actual) **sino** escribirEntero(–1) **fsi**	Habiendo encontrado el elemento buscado	

Como en este caso...

... la búsqueda no tiene tratamiento, nos ahorramos la columna correspondiente a las variables del tratamiento.

3.5.2. Números primos

Nos piden diseñar una función que determine si un número entero positivo es primo o no, retornando cierto en caso afirmativo y falso en caso contrario.

La especificación de la función (ya nos la da el enunciado) es la siguiente:

funcion esPrimo(n: **entero**): **booleano**
{ Pre: *n* = N y N > 0 }
{ Post: El valor retornado por la función esPrimo será **cierto** si *N* es primo, y **falso** en caso contrario }

Solucionemos el problema. Como ya tenemos la especificación, lo primero que debemos hacer es plantear el algoritmo en términos de tratamiento secuencial.

¿Cómo reformulamos el problema en función de una búsqueda en una secuencia? Pues pensemos en primer lugar qué significa que un número sea primo: si un número *N* es primo, quiere decir que no hay ningún otro número entre 2 y *N* – 1 que lo divida.

Para comprobar si un número *N* es primo o no, tendremos que verificar que no hay ningún número entre 2 y *N* – 1 que lo divida. Es decir, debemos verificar que todos los elementos de la secuencia <2, 3, 4, ..., *N* – 1> no dividen *N*.

En realidad,...

... tenemos suficiente con verificar esto para los elementos de la secuencia <2, 3, 4, ..., *K*>, donde *K* es el mayor número tal que $K^2 \leq N$. Siempre que exista un divisor entre *K* y *N*, seguro que habrá otro entre 2 y *K* (de forma que, multiplicándolos los dos, nos dé *N*). De la misma manera, si *N* no es divisible por 2, tampoco lo será por 4, 6, ...

En el momento en que encontremos un elemento de la secuencia que divida *N* ya podemos acabar la búsqueda, no se cumple que "todos los elementos de la secuencia no son divisores de *N*". En cambio, si no lo encontramos, concluiremos que todos los elementos de la secuencia no son divisores de *N* (es decir, ninguno de ellos es divisor); y por lo tanto, que *N* es un número primo.

Notad que el hecho de comprobar que todos los elementos de una secuencia cumplen una propiedad se puede reformular como una búsqueda de un elemento de la secuencia que no cumple dicha propiedad.

Es habitual encontrarse con problemas que corresponden a la aplicación del esquema de búsqueda sin que la palabra *búsqueda* aparezca por ningún lado. Este caso, en el que debemos comprobar que todos los elementos de la secuencia cumplen una propiedad, es uno de los más habituales.

De este modo, el problema queda reformulado como una búsqueda en la secuencia <2, 3, 4, ..., *N* – 1> de un divisor de *N*. Nos hace falta una variable (utilizaremos *i*) para ir visitando los diferentes elementos de la secuencia. Aparte, tendremos el dato de entrada que corresponde al número que queremos comprobar si es primo, y que en este caso recibimos como un parámetro de entrada de la función (lo denominamos *n*). Por último, tenemos una variable booleana (que denominaremos *primo*) que se corresponderá con el resultado que hay que retornar. Así pues, refinemos el esquema:

– *prepararSecuencia*: consiste en inicializar *i* a 2.

– *inicioTratamiento*: no hay tratamiento.

– *finSecuencia*: el cabezal habrá llegado a la posición que hay más allá del último elemento cuando se cumple que $i = n$. O si quisiéramos hacer una versión más eficiente (tal y como se ha comentado antes), *finSecuencia* se podría corresponder a $i * i > n$.

– *actualizarEncontrado*: no debemos olvidar que buscamos un divisor de *n*; por lo tanto, aquí debemos poner *encontrado* en **cierto** si el elemento actual es divisor de *n*. Del mismo modo que en el ejemplo anterior, esto se puede conseguir tanto con un condicional como con una simple asignación. De este modo:

```
si n mod i = 0 entonces
    encontrado := cierto
fsi
```

O bien así:

```
encontrado := n mod i = 0;
```

– *avanzarSecuencia*: simplemente debemos incrementar la variable *i*.

– *tratamientoFinal*: una vez hemos llevado a cabo la búsqueda, veremos que la variable *encontrado* tendrá como valor **cierto** si *n* tiene algún divisor y **falso** si, por el contrario, no tiene ninguno. De este modo, el número será primo si no hemos encontrado ningún divisor, y no primo si hemos encontrado alguno. Esto corresponde a la siguiente asignación:

```
primo := no encontrado;
```

Fácilmente nos podríamos haber ahorrado esta variable retornando directamente como resultado de la función la expresión **no** *encontrado*. No lo hemos hecho así con la única intención de mostrar explícitamente que existe un tratamiento final que consiste en negar el booleano encontrado.

Notad que...

... muchas veces podremos fusionar construcciones algorítmicas adjuntas. Aquí, por ejemplo, podemos fusionar los dos **si** en:

```
si n mod i = 0 entonces
   encontrado := cierto
sino
   i := i + 1
fsi
```

Este proceso de fusión siempre debe ser posterior al refinamiento del esquema y nunca debe ser una excusa para no aplicarlo correctamente.

Veamos, entonces, cómo queda la función *esPrimo*:

```
funcion esPrimo(n: entero): booleano
  { Pre: n = N y N > 0 }
  var
    encontrado, primo: booleano;
    i : entero;
  fvar
  i := 2;
  encontrado := falso;
  mientras i < n y no encontrado hacer
    si n mod i = 0 entonces
      encontrado := cierto
    fsi
    si no encontrado entonces
      i := i + 1
    fsi
  fmientras
  primo := no encontrado;
  retorna primo;
  { Post: primo es cierto si N es primo y falso en caso contrario }
ffuncion
```

4. Esquemas aplicados a secuencias de entrada/salida

Como ha quedado dicho, las secuencias sirven para modelizar muchos de los problemas que podemos resolver con un algoritmo. Como caso particular muy habitual, encontramos el procesamiento secuencial de la entrada de un algoritmo (datos que lee del teclado, un fichero, etc.). Esto amplía el tipo de problemas que podremos resolver: ahora ya no nos tendremos que limitar a tratar secuencias generadas por el propio algoritmo, sino que seremos capaces de procesar secuencias que nos vienen dadas. Teniendo en cuenta la frecuencia con que se dan estas situaciones, refinaremos los esquemas introducidos en las secciones anteriores para obtener de los mismos una adaptación al tratamiento de la entrada.

Utilizaremos las operaciones de lectura que se presentaron en el módulo anterior (las funciones *leerEntero*, *leerReal* y *leerCaracter*). Eso sí: en cada tratamiento secuencial sólo se podrá utilizar una de las tres funciones, teniendo en cuenta que todos los elementos de una secuencia deben ser del mismo tipo.

En todo tratamiento secuencial es indispensable un mecanismo de detección del final de la secuencia. En el caso de la entrada, para hacer esta detección se suele añadir un elemento adicional detrás del último elemento relevante. Este elemento no se debe tratar, sólo hace el papel de aviso de final de secuencia. Denominamos este elemento *marca*, y las secuencias que tienen marca las denominamos, como es natural, *secuencias marcadas*. De este modo, la entrada del programa será una secuencia marcada. Esto quiere decir que no tendremos nunca la secuencia vacía: deberemos leer –aunque no la tratemos–, como mínimo, la marca.

4.1. Esquema de recorrido aplicado a la entrada

4.1.1. Planteamiento

El refinamiento del esquema presentado en la sección 2 para adaptarlo como acabamos de explicar queda de este modo:

```
algoritmo RecorridoEntrada
    var elem: T;
    fvar    { donde T es el tipo de datos de los elementos de la secuencia }
    elem := leer(); { Función de lectura del tipo de datos de los elementos
                      de la secuencia. Dependerá del tipo T que no es rele-
                      vante frente a la formulación de este esquema }
    inicioTratamiento
```

```
    mientras no (elem = marca) hacer { Hemos tratado todos los elementos leídos a excepción del último, que se encuentra en elem y no es la marca }
    tratarElemento(elem)
    elem := leer()        { La misma función de lectura que antes }
    fmientras             { Hemos tratado todos los elementos leídos a excepción del último, que es la marca }
    tratamientoFinal
falgoritmo
```

Notad que...

... no hemos hecho más que copiar el esquema de recorrido cambiando *preparar-Secuencia* por una lectura, *finSecuencia* por una comparación con la marca y *avanzarSecuencia* por una lectura. En este refinamiento, el cabezal corresponde a la variable *elem*, donde se encuentra el último elemento leído, y la posición final de la secuencia corresponde a la marca.

4.1.2. Especificación

De forma similar a lo que hemos hecho en las secciones anteriores, y a pesar de que no se incluya en los objetivos de la asignatura, reflexionemos un poco sobre la corrección del algoritmo que acabamos de presentar.

En el esquema hemos mantenido la descripción de los estados en los que se encuentra el algoritmo en diferentes momentos del proceso, adaptándola teniendo en cuenta que la secuencia es el dispositivo de entrada. El invariante del **mientras** "hemos tratado la parte de la secuencia que está antes del cabezal" se ha refinado como "hemos tratado todos los elementos leídos a excepción del último". Por otro lado, sabemos que el último leído (el cabezal) está siempre guardado en la variable *elem*. Además, dentro del bucle, *elem* no es la marca, porque de otro modo no hubiésemos entrado. El caso concreto del invariante antes de empezar a iterar ("el cabezal está al principio de la secuencia" en el original) es "*elem* tiene el primer valor de la entrada", y después de hacer todo el tratamiento ("hemos tratado toda la secuencia y el cabezal está al final" en el original) es "hemos tratado toda la entrada anterior a la marca y *elem* contiene la marca".

Recordad que, al igual que habéis hecho con las diferentes partes del esquema, la invariante también se debe refinar para cada problema concreto al que apliquemos el esquema de recorrido.

Finalmente, la noción de cota se convierte ahora en el número de elementos de la entrada que quedan por leer.

4.1.3. Ejemplo

Como ejemplo de aplicación de este esquema, diseñaremos un algoritmo que cuente las veces que la letra *a* aparece en una frase acabada con un punto e introducida por el teclado. La especificación es la siguiente:

```
{ Pre: en la entrada leeremos una secuencia de caracteres que no contiene ningún punto seguida de un punto }
cuentaLetraA
{ Post: escribe el número de veces que aparece la letra 'a' en la entrada }
```

Podemos asimilar este algoritmo a un recorrido de la entrada en el que el tratamiento de cada elemento consiste en incrementar o no un contador según el elemento sea o no sea una letra *a*. El punto final de la frase actúa como marca de la secuencia. No nos es preciso ningún tratamiento final, y el tratamiento inicial se refina al inicializar el contador a 0, mientras que el tratamiento de un elemento se convierte en una instrucción alternativa (**si** ... **fsi**).

Para mayor claridad,...

... también conviene generalmente, como ya hemos dicho, refinar los predicados informales que describen el estado al comienzo de cada iteración y después del bucle.

Con estos refinamientos obtenemos este algoritmo:

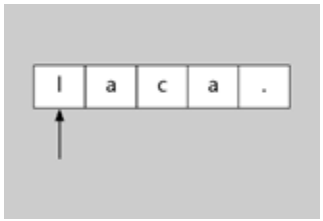
```
algoritmo cuentaLetraA

  var car: caracter; n: entero;
  fvar
  car := leerCaracter();
  n := 0;
  mientras no car = '.' hacer
    { n es el número de letras a que había entre los caracteres leídos excepto el último, que se
    encuentra en car y no es el punto final }
    si car = 'a' entonces n := n + 1
    fsi
    car := LeerCaracter()
  fmientras
  { n es el número de letras a que había entre los caracteres leídos excepto el último, que es el
  punto final }
   escribirEntero(n)
falgoritmo
```

Para facilitar la identificación del algoritmo con el esquema de recorrido original, veamos una representación gráfica de la evolución de su ejecución en la tabla siguiente. Supongamos que en la entrada encontraremos los caracteres 'l', 'a', 'c', 'a' y '.'. La segunda columna representa la secuencia del esquema y la tercera presenta el valor de las variables:

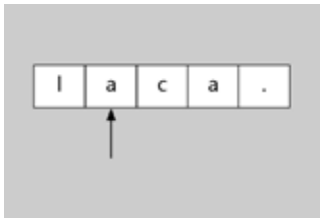
- La variable *car* refleja el estado de la secuencia y contiene el último elemento leído que, como hemos dicho, corresponde al cabezal de la secuencia.

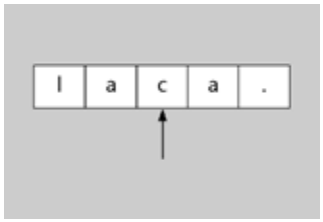
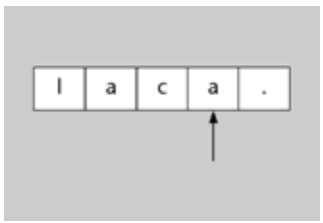
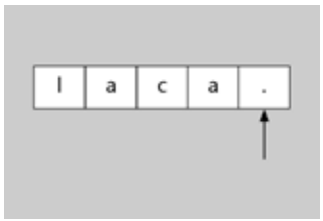
- La variable *n*, tal como indican los predicados del algoritmo, contiene el número de letras *a* que hay antes del cabezal.

La primera fila representa el estado después de ejecutar las inicializaciones (preparar la secuencia e inicializar el tratamiento). Las filas siguientes representan el estado al final de cada iteración del bucle. Una flecha recuerda a qué punto concreto del algoritmo corresponde el valor de las variables que se muestra.

Algoritmo	Secuencia	Variables
car := leerCaracter(); → n := 0;		*car* = 'l' *n* = 0

Algoritmo	Secuencia	Variables

mientras no car = '.' **hacer**
  si car = 'a' **entonces**
  n := n + 1
  fsi
  car := leerCaracter()
→ **fmientras**
		*car* = 'a'
		n = 0

		*car* = 'c' *n* = 1
		*car* = 'a' *n* = 1
		*car* = '.' *n* = 2

4.2. Esquema de búsqueda aplicado a la entrada

4.2.1. Planteamiento

Pasemos ahora a refinar el esquema de búsqueda original que hemos visto anteriormente en este módulo para obtener un esquema de búsqueda en el que la secuencia explorada es un dato de entrada del algoritmo.

Como en el apartado anterior, tratamos la entrada como una secuencia con una marca que actuará como la posición final del cabezal. También tendremos una variable *elem* que contendrá el valor correspondiente a la posición del cabezal del esquema de búsqueda original.

De este modo, como antes, hacemos estos refinamientos:

- *prepararSecuencia*: poner el cabezal en la primera posición equivale a hacer que *elem* contenga el primer valor.

```
elem := leer();
```

- *finSecuencia*: comprobar si el cabezal está al final.

  elem = marca

- *avanzarSecuencia*

  elem := leer();

  Como antes, según el tipo de elem, la función de lectura será *leerCaracter*(), *leerEntero*() *o leerReal*().

El esquema resultante es el siguiente:

```
algoritmo BúsquedaEntrada

  var
    encontrado: booleano;
    elem: T; { Tipos de datos de los elementos de la secuencia }
  fvar

  elem := leer(); { Función de lectura del tipo de datos de los elementos
                    de la secuencia. Dependerá del tipo T que no
                    es relevante para la formulación de este esquema }
  encontrado := falso;
  inicioTratamiento
  mientras no (elem = marca) y no encontrado hacer
    { Hemos tratado todos los elementos leídos a excepción del último,
    que se encuentra en elem y no es la marca. Ninguno de los elementos
    tratados es el que buscamos }
    actualizarEncontrado
    si no encontrado entonces
      tratarElemento
      elem := leer() { La misma función de lectura que antes }
    fsi
  fmientras
    { Hemos tratado todos los elementos leídos a excepción del último,
    que se encuentra en elem. Ninguno de los elementos tratados es el que
    buscamos. Si encontrado es cierto, elem es el elemento buscado; si es
    falso, elem es la marca }
    tratamientoFinal
falgoritmo
```

4.2.2. Especificación

De forma similar a lo que hemos hecho en las secciones anteriores, reflexionemos un poco sobre la corrección del esquema que acabamos de presentar. Tal como hemos hecho en la sección anterior, examinemos la conexión que se establece entre el esquema de búsqueda general y el esquema adaptado para procesar el dispositivo de entrada.

El predicado del inicio del **mientras** "hemos tratado la parte de la secuencia que está antes del cabezal. El elemento buscado no está en esta parte de la secuencia. Además, el cabezal no está al final" se ha reformulado en "hemos tratado todos los elementos leídos a excepción del último, que se encuentra en *elem* y no es la marca. Ninguno de los elementos tratados es el que buscamos".

Una vez más, el invariante proporcionado es bastante general y se debe concretar para cada aplicación del esquema (igual que concretamos o refinamos los fragmentos del esquema en conjuntos de acciones o en expresiones del lenguaje algorítmico).

En lo que respecta a la función de cota, vuelve a corresponder al número de elementos que quedan por leer.

4.2.3. Ejemplo

Un ejemplo sencillo de aplicación de la búsqueda a la entrada consiste en averiguar si en una frase acabada en punto aparece alguna letra *a*.

```
{ Pre: en la entrada leeremos una secuencia de caracteres que no contiene ningún punto seguida de un punto }
saleLetraA
{ Post: por el dispositivo de salida se escribe 'S' si en la entrada hay alguna letra 'a' y 'N' en caso contrario }
```

A pesar de la similitud de este ejemplo con el anterior (contar el número de letras *a*), ahora no lo debemos resolver con un recorrido sino con una búsqueda, porque no es necesario continuar procesando la secuencia a partir del momento en que encontramos una letra *a*. De este modo, a partir del esquema de búsqueda en la entrada, haciendo los refinamientos necesarios obtenemos el algoritmo siguiente:

```
algoritmo hayLetraA

    var
        encontrado: booleano;
        car: caracter;
    fvar

    car := leerCaracter();
    encontrado := falso;
    mientras no (car = '.') y no encontrado hacer
        { Ninguno de los elementos leídos anteriores al último es una letra "a".
          El último está en car }
        encontrado := car = 'a';
        si no encontrado entonces
            car := leerCaracter()
        fsi
    fmientras
    { Ninguno de los elementos leídos anteriores al último es una letra "a".
      El último está en car. Si encontrado es cierto, car es una letra "a"; si es falso, car es el
      punto final }
    si encontrado entonces escribirCaracter('S')
    si no escribirCaracter('N')
    fsi
falgoritmo
```

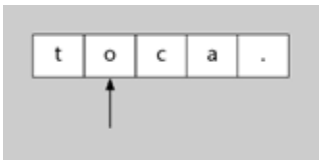
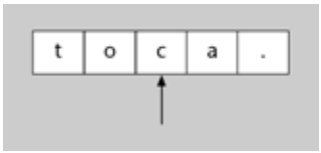
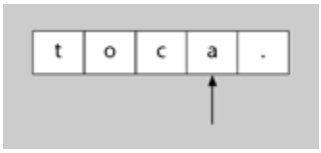
Es una búsqueda...

... sin tratamiento, de forma que no hay ni *InicioTratamiento* ni *TratarElemento* ni *TratamientoFinal*. *El refinamiento de ActualizarEncontrado* es, sencillamente:
"encontrado := car = 'a'".

Veamos, como antes, la evolución de la ejecución con una tabla para intentar identificar el algoritmo con el esquema de búsqueda original. Supongamos ahora que en la entrada encontraremos los caracteres 't', 'o', 'c', 'a' y '.'. De nuevo, la segunda columna representa la secuencia del esquema y la tercera presenta el valor de las variables:

- *car* refleja el estado de la secuencia y contiene el último elemento leído que, como hemos dicho, corresponde al cabezal de la secuencia lógica.
- *encontrado* se asignará a **cierto** si el elemento del cabezal –es decir, *car*– es una 'a'.

Recordemos que la primera fila representa el estado después de ejecutar las inicializaciones (preparar la secuencia e inicializar *encontrado*) y que cada una de las filas siguientes representa el estado al final de cada iteración del bucle.

Algoritmo	Secuencia	Variables
car := leerCaracter(); encontrado := **falso**;		*car* = 't' *encontrado* = **falso**
mientras no car = '.' **y no** encontrado **hacer** encontrado := car= 'a'; **si no** encontrado **entonces** car := leerCaracter() **fsi** **fmientras**		*car* = 'o' *encontrado* = **falso**
		*car* = 'c' *encontrado* = **falso**
		*car* = 'a' *encontrado* = **cierto**

4.3. Tres ejemplos

4.3.1. Media aritmética

Nos piden un algoritmo que calcule la media aritmética de una serie de números reales que tenemos que leer de la entrada. Nos aseguran que habrá por lo menos un número en la serie, que todos los números son diferentes de 0.0 y que detrás del último elemento de la serie habrá un 0.0.

```
{ Pre: en la entrada leeremos uno o más reales diferentes de 0.0 seguidos de un 0.0 }
media
{ Post: se ha escrito por el dispositivo de salida la media de los números leídos sin contar
el 0.0 }
```

Este problema se resuelve claramente con un recorrido de la entrada. Queremos que el bucle cuente y sume los elementos de la entrada. Por eso, antes de cada iteración queremos que un contador *n* diga cuántos elementos hemos tratado (los que hemos leído excepto el último) y que un acumulador *sum* sea la suma de estos *n* números. De este modo, al final de la entrada, cuando hayamos leído la marca, *n* será el número de elementos relevantes que había y *sum* la suma de estos elementos. Para cumplir la condición al principio, cuando hemos leído un elemento es necesario que *n* y *sum* equivalgan a cero.

Hemos refinado...

... el esquema de recorrido de la entrada de forma que la marca es el 0.0, el tratamiento inicial es la inicialización a cero del número de elementos y la suma y el tratamiento de un elemento consiste en sumarlo con la suma anterior e incrementar el contador de elementos El tratamiento final realiza la división para obtener el resultado pedido.

```
algoritmo media

    var
        x, sum: real;
        n: entero
    fvar

    x := leerReal();
    n := 0;
    sum := 0.0;
    mientras no (x = 0.0) hacer
        { sum = suma de todos los elementos leídos a excepción del último, que se encuentra
          en x y no es la marca y n = número de elementos sumados }
        sum := sum + x;
        n := n + 1;
        x := leerReal()
    fmientras
    { sum = suma de todos los elementos leídos a excepción del último, que es la marca, y
      n = número de elementos sumados }
    escribirReal(sum/enteroAReal(n)).
falgoritmo
```

Observad que para poder dividir *sum* entre *n*, se debe hacer la conversión de tipo.

4.3.2. ¿Aprueban todos?

Nos piden un algoritmo que nos indique si una serie de notas que debemos leer de la entrada está formada exclusivamente por aprobados. Nos aseguramos de que las notas de la serie serán caracteres del conjunto {“A”, “B”, “C”, “c”, “D”} y que detrás del último elemento de la serie habrá una zeta. Las calificaciones aprobadas son “A”, “B” y “C” (en la UOC conocemos esta última nota como C+). En cambio, la “c” (que representa la C–) y la “D” son suspensos.

```
{ Pre: en la entrada leeremos una secuencia de caracteres del conjunto {“A”, “B”, “C”, “c”,
“D”} seguida por una “Z” }
todosAprueban
{Post: por el dispositivo de salida se escribe “S” si en la entrada sólo había “A”, “B” y “C”,
y “N” en caso contrario }
```

Este problema puede replantearse como la búsqueda de un suspenso en una entrada marcada con el valor “Z”.

Es una búsqueda...

... sin tratamiento, de forma que no hay ni *InicioTratamiento ni TratarElemento* ni *TratamientoFinal. El refinamiento de ActualizarEncontrado* es, sencillamente, encontrado := (car = ‘c’) **o** (car = ‘D’);

```
algoritmo todosAprueban

    var
        encontrado: booleano;
        car: caracter;
    fvar
    car := leerCaracter();
    encontrado := falso;
```

```
    mientras no (car = 'Z') y no encontrado hacer
        { Ninguna de las notas leídas anteriores a la última es un suspenso.
        La última está en car }
        encontrado := (car = 'c') o (car = 'D');
        si no encontrado entonces
            car := leerCaracter()
        fsi
    fmientras
    { Ninguna de las notas leídas anteriores a la última es un suspenso.
      La última está en car. Si encontrado es cierto, car es un suspenso; si es falso, car es 'Z' }
    si encontrado entonces
        escribirCaracter('N')
    sino
        escribirCaracter('S')
    fsi
falgoritmo
```

4.3.3. Número de aprobados

Nos piden un algoritmo que nos indique si el número de aprobados de una serie de notas ordenada de forma descendente que debemos leer de la entrada supera una determinada cota, que también debemos leer. Nos aseguran que las notas de la serie serán caracteres del conjunto {"A", "B", "C", "c", "D"} y que detrás del último elemento de la serie habrá una "Z".

```
{ Pre: en la entrada leeremos un entero y una tira de caracteres del conjunto {"A", "B", "C",
"c", "D" seguida por una "Z" }
cotaAprobados
{ Post: por el dispositivo de salida se escribe 'N' si el número de aprobados no es inferior a la
cota y 'S' en caso contrario }
```

Este problema puede replantearse como la búsqueda de una nota que nos permite obtener la siguiente respuesta: un suspenso antes de *cota* notas o un apobado después de *cota* notas. Necesitamos, pués, hacer un tratamiento de recuento de los elementos que se van leyendo.

```
algoritmo cotaAprobados

    var
        encontrado: booleano;
        car: caracter;
        cota, n: entero;
    fvar

    cota := leerEntero();
    car := leerCaracter();
    encontrado := falso;
    n := 0;
    mientras no (car = 'Z') y no encontrado hacer
        { Ninguna de las notas leídas anteriores a la última es un suspenso.
          La última está en car, n és el número de notas leídas menos 1 }
          encontrado := (car = 'c') o (car = 'D') o (n ≥ cota);
            si no encontrado entonces
                n := n + 1;
                car := leerCaracter()
            fsi
    fmientras
    { Niguna de las notas leídas anteriores a la última es un suspenso.
      La última está en car. Si encontrado es cierto, car es un suspenso; si es falso, car es 'Z'.
      n es el número de notas leídas menos 1 }
```

Es una búsqueda...

... con estos refinamientos:

- *InicioTratamiento* es la inicialización de *n*.
- *TratarElemento* es el incremento de *n*.
- No hay *TratamientoFinal*.

El refinamiento de *ActualizarEncontrado* es:

encontrado := (car = 'c') **o** (car = 'D') **o** (n ≥ cota);

```
    si n ≥ cota entonces
        escribirCaracter('N')
    sino
        escribirCaracter('S')
    fsi
falgoritmo
```

5. Combinación de esquemas

5.1. Planteamiento

Hemos estudiado en este módulo dos esquemas de diseño de algoritmos –recorrido y búsqueda– muy útiles. Sin embargo, todavía podemos aumentar su utilidad si tenemos en cuenta la posibilidad de combinarlos, de forma que un algoritmo de proceso de una secuencia no sea refinamiento de un esquema sino de distintos esquemas dispuestos uno tras otro. De hecho, si lo observamos con mayor detenimiento, veremos que este tipo de combinación deber ser de una de estas dos formas:

a) *búsqueda*$_1$; *búsqueda*$_2$,..., *búsqueda*$_n$
b) *búsqueda*$_1$; *búsqueda*$_2$,..., *búsqueda*$_n$; *recorrido*

La razón es que, después de un recorrido, ya hemos leído toda la secuencia, de modo que ya no tiene sentido plantear el hecho de continuar haciendo en ella ningún proceso. En cambio, después de una búsqueda, es posible que la secuencia haya sido parcialmente procesada, por lo que puede ser útil continuar el proceso del resto de la secuencia con otra búsqueda o un recorrido. Los apartados siguientes presentan ejemplos de aplicación de la combinación de esquemas.

5.2. Ejemplos

5.2.1. Media de los suspensos

Queremos diseñar un algoritmo que lea una serie de notas entre 0.0 y 10.0 ordenada de forma descendente y marcada con un –1.0; el algoritmo nos debe dar la nota media de los suspensos (o 0.0, si no hay suspensos). Un suspenso es cualquier nota inferior a 5.0.

```
{ Pre: en la entrada leeremos una secuencia de reales entre 0.0 y 10.0 ordenada descendientemente seguida por un –1.0 }
mediaSuspensos
{ Post: escribe la media de los números x tales que 0 ≤ x < 5.0 o bien 0.0 si no había ningún número que cumpliese esta condición }
```

Una forma de resolver el problema consiste en efectuar un recorrido donde el tratamiento sea el siguiente: si la nota actual es un suspenso, sumamos su valor a un acumulador e incrementamos un contador. Al final, dividimos el acumulador por el contador (si es diferente de 0).

Hay otra forma de resolver el problema que consiste en leer primero todos los aprobados (recordemos que las notas nos las dan ordenadas de forma descen-

dente); a continuación, todas las notas que quedan son suspensos, de modo que las podemos procesar haciendo un recorrido que las acumule y las cuente todas (ya no es necesario comprobar si es suspenso o no). Después, como antes, efectuamos la división:

```
algoritmo mediaSuspensos

    var
        x, sum, result: real;
        n: entero;
        encontrado: booleano;
    fvar

    x := leerReal() ;
    encontrado := falso;
    mientras no (x = –1.0) y no encontrado hacer
        { Ninguna de las notas leídas anteriores a la última es un suspenso
          La última está en x }
        encontrado := x < 5.0;
        si no encontrado entonces
            x := leerReal()
        fsi
    fmientras
    { Ninguna de las notas leídas anteriores a la última es un suspenso.
      La última está en x. Si encontrado es cierto, x es un suspenso; si es falso, x es –1.0 }
    n := 0; sum := 0.0;
    mientras no (x = –1.0) hacer
        { sum = suma de todos los valores leídos a partir del primer suspenso a excepción del
        último, que se encuentra en x y no es la marca n = número de elementos sumados }
        sum := sum + x; n := n + 1;
        x := leerReal()
    fmientras;
    { sum = suma de todos los valores leídos a partir del primer suspenso a excepción del úl-
    timo, que es la marca n = número de elementos sumados }
    si n > 0 entonces
        result := sum/enteroAReal(n)
    sino
        result := 0.0
    fsi
    escribirReal(result)
falgoritmo
```

Esta primera parte...

... (la búsqueda) corresponde al algoritmo *todosAprueban* que hemos visto en la sección 4.3.2.

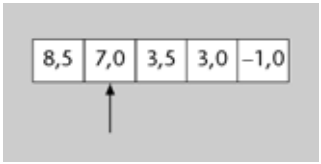
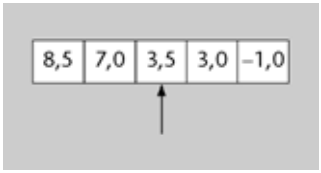
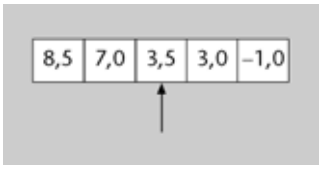
Esta segunda parte...

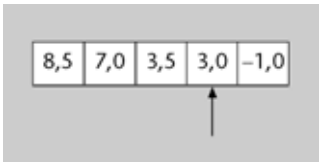
... (el recorrido) corresponde al algoritmo *media* que hemos visto en la sección 4.3.1.

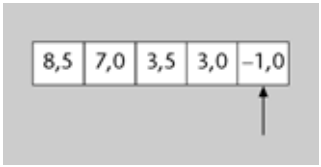
Sin embargo, no aparece la lectura previa al bucle porque en la variable *x* se encuentra el último valor leído en la búsqueda que hay que procesar en este recorrido (si no es la marca).

A continuación visualizamos con una tabla la ejecución del algoritmo cuando en la entrada encontramos los valores 8,5, 7,0, 3,5, 3,0 y –1,0. Seguimos el mismo convenio que en las tablas similares que hemos utilizado antes (ejemplos *cuentaLetraA* y *hayLetraA*). Hay una línea gruesa que separa las filas según correspondan a la búsqueda (primera parte del algoritmo, filas superiores de la tabla) o al recorrido (segunda parte, filas inferiores). La tercera fila representa el estado justo después del primer bucle, y la cuarta fila corresponde al estado justo antes del segundo bucle. En la columna de variables sólo aparecen las que son relevantes en cada parte (*x* en las dos, *encontrado* sólo en la primera y *sum* y *n* sólo en la segunda).

Algoritmo	Secuencia	Variables
x := leerReal(); encontrado := **falso**;		*x* = 8,5 encontrado = **falso**

Algoritmo	Secuencia	Variables
mientras no *x* := –1.0 **y no** encontrado **hacer** encontrado := *x* < 5.0; **si no** encontrado **entonces** *x* := leerReal() **fsi** **fmientras**		*x* = 7,0 encontrado = **falso**
		*x* = 3,5 encontrado = **cierto**
n := 0; sum := 0.0;		*x* = 3,5 *n* = 0 sum = 0,0

mientras no x = –1.0 **hacer**
sum := sum + x;
n := n + 1;
x := leerReal();
fmientras
		*x* = 3,0
		n = 1
		sum = 3,5

		*x* = –1,0 *n* = 2 sum = 6,5

5.2.2. Longitud de la primera palabra

Ahora nos plantearemos un problema que hace referencia a un texto: sabemos que en la entrada leeremos una frase marcada con un punto y formada por letras mayúsculas y otros caracteres que consideraremos separadores de palabras. Delante y detrás de cada palabra puede haber uno o más separadores. Queremos diseñar un algoritmo que nos indique la longitud de la primera palabra (si no hay ninguna palabra, la respuesta debe ser 0).

{ Pre: en la entrada leeremos una secuencia de caracteres que no contiene ningún punto seguida de un punto }
longPrimeraPal
{ Post: escribe la longitud *long* de la primera palabra (secuencia de letras mayúsculas seguidas) o 0 si no había ninguna letra }

Podemos obtener la solución combinando dos búsquedas: la primera lee caracteres separadores hasta encontrar la primera letra de la primera palabra; la segunda búsqueda lee caracteres hasta encontrar el primer separador posterior a la primera palabra y va contando las letras leídas. El valor de este contador es la respuesta pedida (incluso en el caso de que no haya ninguna palabra, porque, en esta situación, la primera búsqueda fracasa y queda al final de la secuencia; a continuación, la segunda búsqueda acaba inmediatamente sin entrar en el bucle, dejando el contador a 0).

```
algoritmo longPrimeraPal

    var
        car: caracter;
        n: entero;
        encontrado: booleano;
    fvar

    car := leerCaracter();
    encontrado := falso;
    mientras no (car = '.') y no encontrado hacer
        { Ninguno de los caracteres leídos anteriores al último es una letra. El último está
        en car }
        encontrado := (car ≤ 'Z') y (car ≥ 'A');
        si no encontrado entonces
            car := leerCaracter()
        fsi
    fmientras
    { Ninguno de los caracteres leídos anteriores al último es una letra.
      El último está en car.
      Si encontrado es cierto, car es una letra; si es falso, car es el punto final }
    encontrado := falso;

    n := 0;
    mientras no (car = '.') y no encontrado hacer
        { Ninguno de los caracteres leídos anteriores al último después de la primera letra es
          un separador. El último está en x, n es el número de letras leídas }
        encontrado := (car) > 'Z' o (car < 'A');
        si no encontrado entonces
            n := n + 1;
            car := leerCaracter()
        fsi
    fmientras
    { Ninguno de los caracteres leídos anteriores al último después de la primera letra es un
      separador. El último está en car. Si encontrado es cierto, car es un separador; si es falso,
      car es el punto final, n es el número de letras leídas }
    escribirEntero(n)
falgoritmo
```

Esta primera parte...

... es una búsqueda simple sin tratamiento.

Esta segunda parte...

... es una búsqueda con tratamiento de recuento.

Resumen

En este módulo hemos aprendido a diseñar algoritmos mediante la aplicación de esquemas de tratamiento secuencial.

Por ello, en primer lugar hemos introducido la noción de **secuencia**. Hemos visto cómo se pueden modelizar muchos algoritmos como algoritmos de tratamiento secuencial (en algunos casos, la secuencia será generada por el mismo algoritmo, mientras que en otros casos nos vendrá dada, normalmente, en el dispositivo de entrada).

Hemos estudiado los dos esquemas básicos de tratamiento secuencial: el de **recorrido** y el de **búsqueda**. Hemos aprendido a refinar estos esquemas para obtener algoritmos que solucionen problemas concretos.

– El esquema de recorrido nos permite tratar todos los elementos de una secuencia.

– En cambio, el esquema de búsqueda nos permite tratar únicamente una parte de la secuencia en cuestión, dejando el cabezal apuntando a un elemento que cumple una propiedad determinada.

Hemos presentado el caso particular de las **secuencias de entrada y salida**, en las que los datos se obtienen de forma secuencial del dispositivo de entrada (o enviados al de salida).

Hemos visto cómo se adaptan los dos esquemas de tratamiento secuencial para el caso en que la secuencia sometida a tratamiento proviene del dispositivo de entrada. Estas secuencias tienen una importancia especial, ya que hacen mucho más útiles los dispositivos de entrada y de salida utilizados en esta asignatura. En muchos problemas deberemos obtener o enviar los datos a estos dispositivos.

Hemos aprendido a combinar más de un esquema para resolver problemas en los que la solución consiste en hacer más de un tratamiento de forma encadenada.

Así pues, este módulo es un módulo de contenido metodológico, en el que hemos aprendido a diseñar algoritmos no triviales de una forma lo más sistemática posible. Ya que el objetivo consiste en obtener algoritmos correctos, es muy importante (prácticamente imprescindible y obligatorio en esta asignatura) la aplicación de la metodología y los esquemas de tratamiento secuencial aquí expuestos y tratados. 

Para diseñar algoritmos será necesario que sigáis las pautas que os hemos recomendado y que ya aparecían en la sección homónima del módulo "Introducción

a la programación": entender el enunciado, plantear la solución, formularla y, finalmente, evaluarla. Concretaremos un poco más en qué consiste cada uno de estos pasos ahora que ya habéis estudiado este módulo.

1. Entender el enunciado

En primer lugar, debemos saber de una forma clara y precisa lo que queremos conseguir. Por ello, lo que debemos hacer es especificar nuestro problema. Respecto a este paso no ha cambiado nada. Una vez tenemos descrito claramente **qué** debe hacer nuestro algoritmo (el "qué" se corresponde a la especificación), ya podemos pasar al **cómo**, es decir, podemos pasar a plantear el algoritmo.

2. Plantear el problema

En este punto debemos evaluar si nuestro problema puede resolverse mediante tratamiento secuencial. Si es así (casi siempre ocurre así con los problemas mímimamente complejos), debemos seguir los siguientes pasos:

a) Averiguar cuál es la secuencia que hay que tratar para resolver el problema. En muchos casos, la secuencia es bastante evidente, como aquellos problemas en los que la secuencia se obtiene de la entrada estándar. Trivial o no, decidir la secuencia que hay que tratar es un paso crucial en el planteamiento del problema.

b) Decidir si con un único tratamiento tendremos suficiente para resolver el problema o si nos hará falta aplicar varios esquemas (una o más búsquedas seguidas de un posible recorrido). Si necesitamos más de un tratamiento, repetiremos todos los pasos para cada uno.

c) Elegir el esquema adecuado que debemos aplicar (el de búsqueda o el de recorrido).

Cualquier problema que...

... se pueda resolver mediante el uso de una estructura iterativa puede ser modelado en realidad como un algoritmo de tratamiento secuencial. De todos modos, si el problema presenta una cierta complejidad, nos pueden hacer falta otras técnicas como las explicadas en el último módulo. Por otro lado, existen también otros tipos de problemas para los que resulta más adecuado aplicar otros esquemas que no veremos en esta asignatura.

3. Formular la solución

Una vez ya tenemos el planteamiento, se trata de formular la solución utilizando el lenguaje algorítmico que conocemos. Para hacerlo, deberemos seguir los siguientes pasos, íntimamente relacionados entre sí (tal y como ya se ha comentado en el tema del esquema de recorrido):

a) Decidir el conjunto de variables que necesitaremos para resolver el problema.

b) Refinar el esquema que hayamos elegido en el punto c del planteamiento. Es decir, sustituir cada una de las partes generales del esquema elegido por conjuntos de acciones y expresiones del lenguaje algorítmico, de modo que obtengamos un algoritmo que solucione nuestro problema (o uno de los tratamientos en el caso de que la solución consista en combinar varios tratamientos).

4. Evaluar la solución

Los esquemas son algoritmos genéricos bien construidos y correctos. Su uso, por lo tanto, nos libera de una gran cantidad de errores que podríamos cometer si tuviésemos que construir el algoritmo partiendo de cero cada vez (aparte de la mayor dificultad de hacer esto). Sin embargo, no nos liberaremos de cualquier error. Podemos cometer errores:

– En el planteamiento: cuando decidimos si el problema es solucionable mediante uno o más tratamientos, en el momento de elegir el esquema, o incluso cuando seleccionamos la secuencia que hay que tratar.

– En la formulación de la solución: cuando refinamos el esquema.

Los errores en el planteamiento son más difíciles de solucionar. También son costosos, pues muy probablemente implican que la formulación de la solución para el planteamiento equivocado se deba repetir. Por este motivo, no nos debe importar pasar un buen rato para plantear correctamente el problema.

Los errores de la formulación se deben a que hemos refinado alguna de las partes del esquema de forma incorrecta. Para detectar estos errores, debemos ser capaces de clasificar mentalmente las posibles secuencias que nuestro algoritmo debe poder tratar en diferentes tipos. Normalmente nos encontraremos con un caso general en el que tendremos una secuencia con unos cuantos elementos, pero también con casos especiales, como por ejemplo la secuencia vacía (siempre que la precondición admita estos casos especiales, evidentemente).

Debemos prestar atención a estos casos especiales y observar si la solución que hemos obtenido los trata correctamente. Tal vez hayamos hecho el refinamiento pensando únicamente en el caso general; entonces, en alguno de estos casos especiales, es probable que el algoritmo no se comporte correctamente. Una parte especialmente delicada es el *finSecuencia*. Por ejemplo, prestad atención al comentario correspondiente a esta parte en el ejemplo de los números de Fibonacci del tema del esquema de búsqueda. Sin embargo, podéis haber introducido problemas sin daros cuenta en cualquiera de las partes del esquema que deben refinarse.

Para hacer todo esto tendréis que razonar sobre el comportamiento del algoritmo resultante, utilizando el significado de las construcciones del lenguaje algorítmico que aparecen en el mismo (y que ya conocéis).

Por otro lado, se debe resaltar que el uso de los esquemas de tratamiento secuencial y de la metodología introducida en este módulo minimiza el esfuerzo del desarrollador para obtener una solución correcta a problemas que se puedan modelar mediante el tratamiento secuencial (que, como ya hemos indicado, son casi todos).

Por este motivo, el uso de estos esquemas y la experiencia planteando problemas como problemas de tratamiento secuencial son puntos clave para mejorar de una forma importante la eficacia y la eficiencia de cualquier programador en el momento de desempeñar su tarea.

Ejercicios de autoevaluación

1. Diseñad una acción que descomponga un número natural en cifras. Es necesario escribir el número dado dígito a dígito, siendo cada dígito un entero entre 0 y 9. Suponed que tenéis disponibles las funciones:

funcion cuantasCifras(n: **entero**): **entero**

{ Pre: $n \geq 0$ }

{ Post: restituye el número de cifras significativas de n }

funcion potencia(n: **entero**): **entero**

{ Pre $n \geq 0$ }

{ Post: retorna el valor 10 elevado a n }

Por ejemplo, 1234 se escribiría 1, 2, 3, 4.

2. Diseñad un algoritmo que nos muestre el valor máximo de una tira de reales marcada con 0.0 disponible para ser leída.

3. Diseñad una acción que indique si una secuencia de reales marcada con 0.0 leída de la entrada está ordenada de forma creciente.

4. Decimos que un número es perfecto si la suma de sus divisores excepto él mismo es igual al propio número.

Por ejemplo, 6 es perfecto porque los divisores de 6 son 1, 2, 3 y 6: 1 + 2 + 3 = 6.

Diseñad un algoritmo que indique si un número es perfecto.

5. Encontrad una acción que encuentre la parte entera de la raíz cuadrada de un entero positivo sin que tengamos ninguna función que calcule raíces cuadradas (sqr, raíz cuadrada...).

Solucionario

1. Este problema no consiste en procesar una secuencia sino en generarla. Sin embargo, podemos construir la acción basándonos en el esquema de recorrido, siendo la secuencia recorrida la que vamos generando. Empecemos especificando el problema para asegurarnos de su correcta comprensión.

accion cifras(**ent** n: **entero**)
{ Pre: $n \geq 0$ }
{ Post: escribe por el canal de salida el entero *n* dígito a dígito }

Ahora intentamos plantear la solución sobre la base de algún esquema. Lo que nos conviene es hacer un recorrido de la secuencia, que en el caso que nos ocupa es la secuencia de dígitos del número. Para obtener la solución tenemos que refinar el esquema que hemos elegido; si conocemos inicialmente la longitud de la secuencia, es decir, el número de cifras, podremos obtener los elementos dividiendo *n* por potencias de 10, empezando precisamente por 10 elevado al número de cifras menos 1. En cada iteración generamos una nueva cifra dividiendo el número por una potencia de 10 inferior a la anterior. Por ejemplo, si tenemos la cifra 9728, cuatroCifras (9728) = 4, potencia (4–1) = 1000, y por tanto, la primera cifra es (9728 **div** 1000) **mod** 10) = 9 mod 10 = 9

```
accion cifras(n: entero)
    {Pre: n ≥ 0 }
    var
    nx, pot10: entero
    fvar
    nx := cuantasCifras (n);
    pot10 := Potencia(nx – 1);
    mientras nx > 0 hacer
        { Quedan nx cifras por escribir }
        escribirEntero((n div pot10) mod 10);
        nx := nx – 1;
        pot10 := pot10 div 10
    fmientras
    { Post: escribe por el canal de salida el entero n dígito a dígito }
faccion
```

2. Primero especificamos el algoritmo (con una precondición que requiere por lo menos un elemento, porque no se puede definir el máximo de un conjunto vacío; para evitar una repetición, nos limitamos a anotar la precondición y la postcondición en el propio algoritmo). Ahora podemos solucionar el problema examinando toda la secuencia con un recorrido que, en cada momento, recuerda el valor más alto de los que se han leído. Refinando el esquema de la forma adecuada obtenemos este algoritmo:

```
algoritmo valMax
    { En la entrada hay una secuencia no vacía de reales que no contiene ningún 0.0 seguida
    de 0.0 }
    var
        x, max: real;
    fvar
    x := leerReal();
    inicioTratamiento
    mientras x ≠ 0.0 hacer
        { max es el mayor de los que se han leído antes de x }
        si x > max entonces max := x
        fsi
        x := leerReal()
    fmientras
    {max es el mayor real de los que se han leído antes de x y x es la marca}
    escribirReal(max)
falgoritmo
```

Hemos refinado el esquema de recorrido de la entrada excepto la parte *inicioTratamien*to, porque requiere pensarla un poco más que las otras. Debemos llevar a cabo las inicializaciones que sea necesario para que { *max* es el mayor real de los que se han leído antes de *x* }. Sin embargo, dado que antes de *x* no se ha leído nada, no está definido cuál debe ser el valor de *max*. Lo que debemos hacer es tratar el primer elemento (la precondición nos garantiza que existe) antes del bucle y el resto de los elementos en el bucle. Entonces tenemos:

```
algoritmo valMax
    { En la entrada hay una tira no vacía de reales que no contiene ningún 0.0 seguida de 0.0 }
    var
```

```
        x, max: real;
    fvar
    x := leerReal(); max := x; x := leerReal();
    mientras x ≠ 0.0 hacer
        { max es el mayor real de los que se han leído antes de x }
        si x > max entonces
            max := x
        fsi
        x := leerReal()
    fmientras
    { max es el mayor real de los que se han leído antes de x y x es la marca }
      escribirReal(max)
falgoritmo
```

3. La especificación es sencilla, y la podéis ver reflejada con predicados en la propia acción. Por lo que respecta a la estrategia de solución, la podemos enfocar como una búsqueda: buscamos si existe un elemento que es menor que el anterior. Si esta búsqueda tiene éxito, la secuencia no es creciente. Si la búsqueda fracasa, la secuencia es creciente.

Notad que la búsqueda se debe realizar a partir del segundo elemento, porque el primer elemento no tiene anterior. Notad también que si la secuencia es vacía, no habrá primer elemento (consideraremos que una secuencia de 0 ó 1 elementos es creciente).

```
accion esCrec ()
    { En la entrada hay una secuencia no vacía de reales que no contiene ningún 0.0 seguida
    de 0.0 }
    var
        x, ant: real;
        encontrado: booleano;
    fvar
    x := leerReal(); encontrado := falso;
    si x ≠ 0.0 entonces
        ant := x; x := leerReal();
        mientras no encontrado y x ≠ 0.0 hacer
            { Los elementos leídos excepto el último, que se encuentra en x y no es la marca,
            forman una serie creciente, y en ant está el elemento anterior a x }
            encontrado := x < ant;
            si no encontrado entonces
                ant := x;
                x := leerReal()
            fsi
        fmientras
    fsi
    { encontrado indica si la secuencia no es creciente }
    si encontrado entonces escribirCaracter('N')
    sino escribirCaracter('S')
    fsi
faccion
```

4. Resolveremos el problema calculando primero la suma de los divisores y, a continuación, la compararemos con el número. Para calcular la suma de los divisores de *N*, podemos hacer un recorrido de una secuencia imaginaria de todos los números mayores que 0 y menores que *N*; el tratamiento consiste en acumular el número actual si es divisor de *N*.

Utilizaremos una variable que irá tomando los valores de la secuencia. De este modo, *preparar-Secuencia* es asignar 1 a *i;* el final de la secuencia lo detectamos cuando *i* = *N* y *avanzarSecuencia* se refina como un incremento de *i*.

```
algoritmo esPerfecto
    { En la entrada hay un número entero mayor que 0 }
    var
        x, i, sum: entero;
    fvar
    x := leerEntero();
    i := 1; sum := 0;
    mientras i < x hacer
        { sum es la suma de los divisores de x menores que i }
        si x mod i = 0 entonces
            sum := sum + i
        fsi
        i := i + 1
```

```
    fmientras
    { sum es la suma de los divisores de x menores que x }
    si sum = x entonces escribirCaracter('S')
    sino escribirCaracter('N')
    fsi
falgoritmo
```

5. La parte entera de la raíz cuadrada de un entero positivo *n* la podemos encontrar mediante una búsqueda en la secuencia de todos los naturales. Buscamos el primer elemento x que cumple la propiedad "*x* al cuadrado > *n*"; el valor que nos piden es *x* – 1. Por ejemplo:

La raíz cuadrada de 0 es 0, que es igual a 1 –1, siendo 1 el primer valor que, elevado al cuadrado, da un resultado mayor que 0.

La parte entera de la raíz cuadrada de 5 es 2, que es igual a 3 –1, siendo 3 el primer valor que, elevado al cuadrado, da un resultado mayor que 5.

Refinamos *prepararSecuencia* con *i* := 0 y *avanzarSecuencia* con *i* := *i* + 1. *tratamientoFinal* no lo refinamos porque estamos haciendo una búsqueda en una secuencia infinita. Sin embargo, tenemos la garantía de que la búsqueda acabará porque siempre encontraremos un elemento que cumpla la propiedad buscada.

```
accion raizCuadr(ent x: entero)
    { x ≥ 0 }
    var
        i: entero; encontrado: booleano;
    fvar
    encontrado := falso; i := 0;
    mientras no encontrado hacer
        { Ninguno de los números menores que i elevado al cuadrado da un resultado mayor
        que x }
        encontrado := i * i > x;
        si no encontrado entonces
        i := i + 1
        fsi
    fmientras
    { Ninguno de los números menores que i elevado al cuadrado da un resultado mayor que
      x, pero i al cuadrado es mayor que x }
    escribirEntero(i – 1)
faccion
```

Glosario

búsqueda
Proceso que tiene como objetivo localizar un elemento que cumple una propiedad determinada.

cabezal
Noción abstracta que permite determinar en cada momento el elemento actual de la secuencia (aquel al que tenemos acceso en un momento dado). El nombre de cabezal se debe a la analogía con el cabezal de los aparatos reproductores de casete, vídeo u otros.

esquema algorítmico
Algoritmo genérico que sirve para solucionar un conjunto de problemas de un tipo determinado. No está expresado totalmente en lenguaje algorítmico, sino que es un tipo de plantilla que contiene unas partes que deben ser sustituidas por acciones y expresiones en lenguaje algorítmico. Estas acciones y expresiones dependerán del problema concreto que estemos solucionando en aquel momento.

marca
Elemento especial que nos determina el final de la secuencia. Este elemento, normalmente, no se debe tratar. Las marcas son especialmente útiles en las secuencias que leemos del dispositivo de entrada, donde representan una forma bastante cómoda de determinar el final de la secuencia.

recorrido
Proceso que tiene como objetivo hacer un tratamiento a todos los elementos de una secuencia.

refinar (un esquema)
Sustituir las partes del esquema algorítmico correspondiente por conjuntos de acciones o expresiones (según lo que convenga). De esta forma convertimos un esquema algorítimico en un algoritmo que soluciona un problema concreto.

secuencia
Conjunto de elementos del mismo tipo que están en un orden determinado (hay un primero, un segundo, un tercero, etc.) Sólo podemos acceder a sus elementos uno a uno, empezando por el primero y siguiendo en el orden en que están dispuestos los elementos en la secuencia.

secuencia vacía
Secuencia que no tiene ningún elemento.

tratamiento secuencial
Proceso que consiste en recorrer una secuencia (o bien una de sus partes) con el objetivo de realizar algún cálculo sobre sus elementos.

Bibliografía

Burgués, X.; Franch, X. (1998). "Tratamiento secuencial". En: P. Botella; M. Bofill; X. Burgués; R. Lagonigro; J. Vancells (1998). *Fundamentos de Programación I* (módulo 3). Barcelona: Ediuoc.
Son los apuntes anteriores de esta asignatura. Utilizan una especificación formal y un modelo físico de secuencias que aquí no tenemos.

Castro, J.; Cucker, F.; Messeguer, X.; Rubio, A.; Solano, L.; Valles, B. (1992). *Curs de programació*. Madrid, etc.: McGraw-Hill.

Scholl, P. C.; Peurin, J.P. (1991). *Esquemas algorítmicos fundamentales. Secuencias e iteración*. Manuales de informática Masson. Masson.

Vancells, J.; López E. (1992). *Programació: introducció a l'algorísmica*. Vic: Eumo Editorial (Tecno-Ciència).