Organizando el código. Estructuras algorítmicas

PID_00275247

Autores que han participado colectivamente en esta obra
Lluís Beltrà
Kenneth Capseta
Maria Jesús Marco Galindo
Antonio Ponce

Idea y dirección de la obra
Maria Jesús Marco Galindo

Diseño y edición gráfica
Asunción Muñoz

Material docente de la UOC

Universitat Oberta de Catalunya

Tercera edición: febrero 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Contenidos

Organizando el código
Estructuras algorítmicas

Cuando ya se tienen claros los datos que tiene que tratar un algoritmo (o un programa) y cómo representarlos, el siguiente paso es determinar los pasos a seguir para resolver el problema que nos plantea el enunciado (estrategia), y concretarlos en las instrucciones y las sentencias del algoritmo.

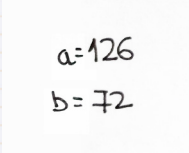
Así podemos completar el paso 3: **diseñar el algoritmo**. Pero ¿cómo se construye un algoritmo?

Ejemplo

Retomamos el algoritmo de Euclides para **calcular el máximo común divisor** entre dos números diferentes de cero.

1 Entendemos el problema

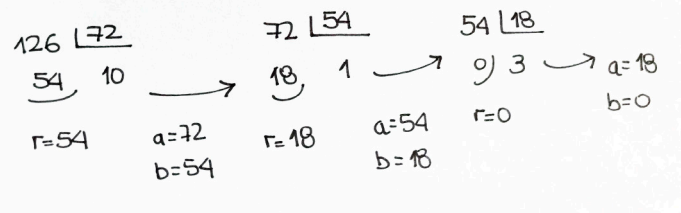
De entrada, tenemos dos números positivos y tendremos que calcular un número también entero positivo que será el máximo común divisor entre los dos de la entrada.



Ya tenemos claro el problema a resolver.

2 Pensamos cómo resolverlo

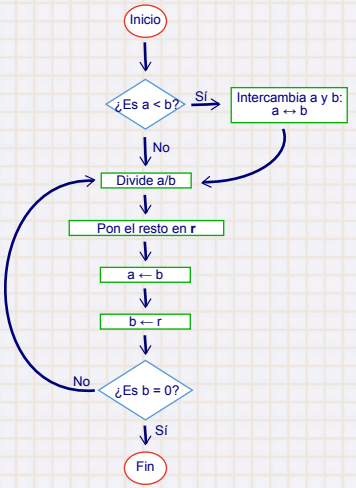
La estrategia de solución ya la tenemos detallada en la unidad anterior, siguiendo el algoritmo de Euclides. Primero, debemos comprobar cuál de los dos números de entrada es mayor, que será el dividendo, el otro será el divisor y luego, recordemos:



Ya tenemos la estrategia para calcular el mcd.

3 Diseñamos el algoritmo

En la unidad 1, hemos realizado el diagrama de flujo que representa la estrategia de solución. Veamos ahora cómo hacer el algoritmo, que será el paso previo a poder programar la solución.



3 Diseñamos el algoritmo

Primera cuestión: ¿cómo representamos y obtenemos los datos?
Con lo que hemos visto en la unidad anterior, tendríamos:

```
var
   dividendo : entero;
   divisor : entero;
   residuo : entero;
fvar

dividendo := leerTeclado();
divisor := leerTeclado();
```

Los datos de entrada.

Segunda cuestión: indicar las instrucciones para hacer los cálculos. Algunas expresiones ya sabemos indicarlas. Por ejemplo, cómo comprobar si el dividendo es menor que el divisor (ya que, en este caso, deberíamos intercambiarlos para poder empezar las divisiones). La expresión sería:

```
(dividendo < divisor)
```

Si esta expresión resultase cierta, siguiendo el algoritmo de Euclides deberíamos intercambiar los valores. Por esta razón necesitamos definir una **variable intermedia** que llamaremos "aux" (variable auxiliar):

```
aux: entero;

aux := dividendo;
dividendo := divisor;
divisor := aux;
```

Añadimos entre **var** y **fvar** la declaración de esta nueva variable auxiliar.

Es la misma estrategia que seguiríamos intercambiando el agua entre dos vasos: tomar un tercero, pasar el líquido del primero al tercero, el del segundo al primero y el del tercero, al segundo.

Recordad esta estrategia siempre que necesitéis intercambiar los valores de dos variables.

Hemos utilizado la asignación **:=** y ya tenemos las tres primeras acciones del algoritmo (que finalmente seran las instrucciones del programa).

La **asignación** es la acción más básica y fundamental sin la que no se puede hacer ningún algoritmo.

Y también podemos escribir la expresión que comprueba si el residuo es cero:

```
residuo = 0
```

Hasta aquí, pues, ya hemos combinado diferentes elementos de la algorítmica que conocemos:

- variables,
- expresiones,
- acción elemental de asignación.

También hemos utilizado funciones de entrada/salida predefinidas para leer los datos del problema.

Además podríamos hacer la división: lo que nos interesa es calcular el residuo, esto podemos hacerlo con el operador **mod** que calcula el módulo o residuo de una división:

```
residuo := dividendo mod divisor;
```

Hemos visto este operador en la unidad anterior.

Pero ya no podemos continuar. Para ello necesitamos saber cómo combinar estas acciones o frases, por ejemplo para poder indicar que, si el dividendo es menor que el divisor, es necesario intercambiarlos, pero si no lo es no, o que es necesario ir repitiendo las divisiones hasta llegar al residuo 0.

Por eso necesitamos conocer las estructuras algorítmicas: secuencial, alternativa o iterativa.

Antes de seguir, fijaos, ¿qué ha sucedido? Hemos definido las variables, y hemos empezado a leer los datos y hacer los cálculos, uno tras otro. Sin embargo, ha llegado un momento en el que se ha necesitado una variable auxiliar para intercambiar los dos valores. Entonces hemos tenido que retroceder, ir a la zona de definición de variables y añadir esta variable *aux*. Por lo tanto, los algoritmos no se escriben desde la primera línea a la última, todo seguido, sino que **se va avanzando y retrocediendo para ir definiendo y completando el código.**

Estructuras algorítmicas: composición de acciones

La asignación

Como ya se ha visto, la **asignación** es la acción fundamental de la algorítmica. Consiste en asignar (dar, guardar) un valor específico dentro de una variable.

Sintaxis de la acción fundamental de asignación

```
nombreVariable := valor;
```

Puede ser un valor o una expresión que dé como resultado un valor.

El valor debe ser del mismo tipo de datos que el de la variable.

Composición de acciones

Solo hay tres maneras de combinar las instrucciones en un algoritmo: en forma de **secuencia**, **de alternativa o de repetición**.

1. **Secuencial**
2. **Alternativa o condicional**
3. **Iterativa o repetitiva**

1. Estructura secuencial

La primera de las opciones, la combinación secuencial, ya la hemos utilizado.

Un algoritmo es, de hecho, una secuencia de acciones (sentencias o instrucciones elementales). Esto es, cada instrucción se ejecuta una tras otra de manera ordenada. Los pasos para resolver el problema se realizan uno tras otro en el orden en el que se escriben en el algoritmo. Por lo tanto, aunque parezca una estructura muy simple, es primordial en programación.

Ejemplo

Vemos, en esta parte del algoritmo que ya hemos resuelto, cómo se aplica la estructura secuencial.

```
dividendo := leerTeclado();
divisor := leerTeclado();
```

Las dos acciones de lectura se realizan secuencialmente una tras la otra.

```
aux := dividendo;
dividendo := divisor;
divisor := aux;
```

Estas tres instrucciones que intercambian los valores también se realizan secuencialmente según el orden en el que están escritas.

Para mejorar la legibilidad del algoritmo, fijaos en que c**ada acción se sitúa en una línea diferente y termina con punto y coma**. Son convenciones que facilitan la elaboración y la comprensión del algoritmo, y que es importante respetar. Lo mismo sucede en los lenguajes de programación, como podéis ver en los códigos JavaScript del final de la unidad.

Ejemplo

Diseñamos un algoritmo que, dado el salario bruto de un trabajador, calcule el salario neto y los impuestos que tiene que pagar sabiendo que paga un 20 % de impuestos.

1 Entendemos el problema

Tenemos dos datos de entrada: el salario bruto y el porcentaje de impuestos que paga. Y piden dos informaciones de salida: Por un lado el salario neto y por otro los impuestos a pagar.

2 Pensamos cómo resolverlo

Decidimos los cálculos a realizar para obtener lo que pide el enunciado

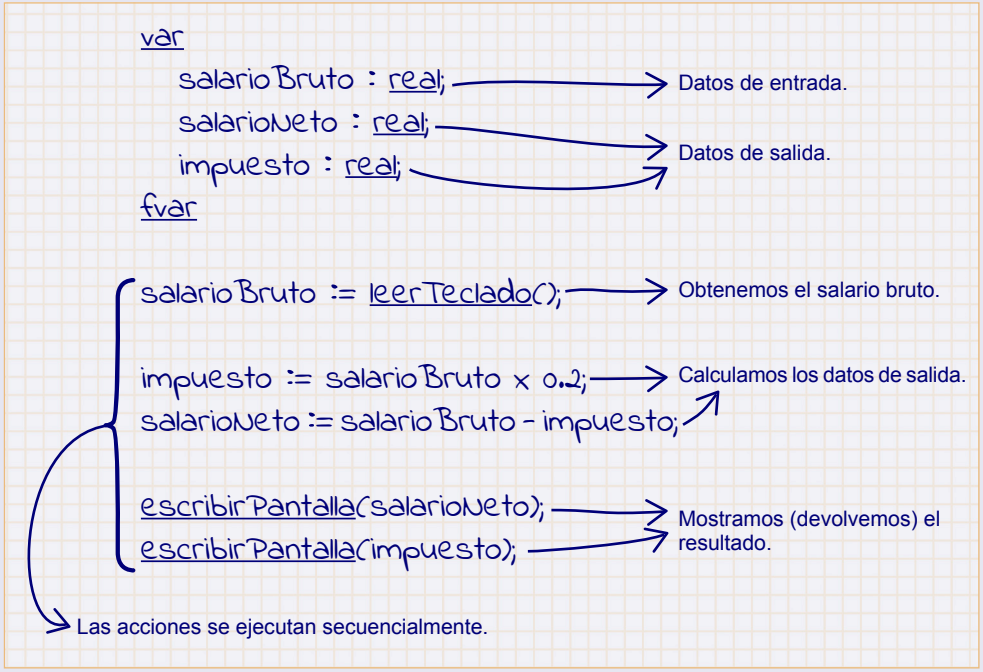
- Salario neto = salario bruto x 0.8
- Impuestos = salario bruto x 0.2

Otra alternativa (estrategia de liquidación) sería

- Impuestos = salario bruto x 0.2
- Salario neto = salario bruto - impuestos

3 Hacemos el algoritmo

Elegimos, por ejemplo, la segunda opción.



2. Estructura alternativa

La estructura secuencial nos permite definir (seguir) un camino (un flujo de ejecución) para resolver el problema, en el que se ejecuta la última instrucción tras otra, un camino único, podríamos decir. A veces, sin embargo, necesitamos que un algoritmo realice ciertas acciones si se cumple una determinada condición, o que haga unas acciones si se cumple una condición y, si no se cumple, que haga otras. Para esto **se utiliza la estructura alternativa o condicional**.

Este concepto de acciones condicionadas se ilustra muy bien con el algoritmo que seguimos para cruzar un paso con semáforo: si está en verde, cruzamos, si está en rojo, esperamos. Y, así, podríamos pensar en muchos otros ejemplos.

Usaremos la composición alternativa cuando queramos ejecutar una acción u otra dependiendo de si se cumple o no una determinada condición (por eso también se denomina composición condicional), es decir, cuando queremos variar el flujo de ejecución de nuestro algoritmo para definir varios caminos alternativos.

Sintaxis de la estructura alternativa o condicional

Algorítmicamente, la **estructura alternativa** (elegir entre HACER UNA COSA o HACER OTRA) se representa de este modo:

```
si condición entonces
        acciones1;
sino
        acciones2;
fsi
```

Las palabras en negrita son palabras clave y, por lo tanto, reservadas (no se pueden utilizar como nombres de variables, por ejemplo).

Acción o acciones que se realizarán si se cumple la condición.

Acción o acciones que se realizarán si NO se cumple la condición.

Y su variante simple (elegir entre HACER o NO HACER):

```
si condición entonces
        acciones;
fsi
```

Ejemplo

Ahora que conocemos la sintaxis de la estructura condicional podemos seguir avanzando en nuestro algoritmo de Euclides para calcular el mcd entre dos números.

El primer paso del cálculo dice que hay que comprobar si el dividendo es menor que el divisor y, si es así, intercambiarlos antes de seguir con las operaciones.

Ya tenemos la expresión para realizar la comprobación:

```
(dividendo < divisor)
```

Expresión que evalúa la condición.

Y las acciones para hacer el intercambio si fuese necesario:

```
aux := dividendo;
dividendo := divisor;
divisor := aux;
```

Acciones a realizar si se cumple la condición.

Ahora solo tenemos que componerlo todo en una estructura iterativa:

```
si (dividendo < divisor) entonces
   aux := dividendo;
   dividendo := divisor;
   divisor := aux;
fsi
```

Expresión que evalúa la condición.

Acciones a realizar si se cumple la condición.

¡Ya teníamos la composición alternativa hecha! Fijaos en que es una versión simplificada de la estructura porque no necesitamos, en este caso, la parte alternativa ya que, si el dividendo es mayor que el divisor, no es necesario hacer nada. Por eso prescindimos de esta parte de la estructura.

La estructura condicional representa una parte del algoritmo (de código, diríamos, si hablásemos de un lenguaje de programación) que necesita comprobar si alguna condición es cierta o falsa antes de ejecutar las acciones de esta parte. Nos permite hacer una selección.

Las decisiones en los algoritmos se toman haciendo comparaciones entre variables, números, caracteres o utilizando expresiones booleanas, en definitiva, expresiones que dan como resultado cierto o falso.

Recordemos los operadores de comparación que hemos visto anteriormente: **=** (igual), **≠** (diferente), **<** (menor), **>** (mayor), **≤** (menor o igual) y **≥** (mayor o igual), operadores que se aplican principalmente a los números pero que también se pueden aplicar a caracteres y tipos booleanos, como ya hemos visto anteriormente. Son operadores que siempre dan como resultado un valor booleano: **cierto** o **falso**.

Las **expresiones booleanas** se utilizan para determinar qué flujo de ejecución (qué "ruta") debe seguir el algoritmo en función de si el resultado de la expresión es **cierto** o **falso**. De aquí que tanto las expresiones algebraicas como el tipo booleano sean tan importantes en la algorítmica.

Ejemplo

Han empezado las rebajas en nuestra tienda de ropa y hace falta ajustar el sistema de cálculo del precio final de cada pieza. ¡Un montón de trabajo! Por suerte podemos hacer un algoritmo simple que nos simplificará el trabajo. Nuestra promoción es la siguiente: a todos los artículos les aplicamos un 25 % de descuento, pero si se compran más de tres piezas, entonces se aplica el 50 % a todas ellas. ¡Una ganga!

1 Entendemos el problema

Tenemos dos datos de entrada: el precio de compra y el número de artículos totales de la compra. Y, como salida, nos piden que calculemos el importe tras aplicar las rebajas.

2 Pensamos cómo resolverlo

Decidimos los cálculos a realizar para obtener lo que pide el enunciado:

- El tipo de descuento es del 25 % si el número de productos es de 1 a 3, y del 50 % si es de más de 3.
- Importe descontado = importe x tipo de descuento.
- Importe final: importe – importe descontado.

3 Hacemos el algoritmo

Detallamos los pasos a seguir en notación algorítmica:

```
const
    tipoDescuento1 : entero := 25;
    tipoDescuento2 : entero := 50;
fconst

var
    importeInicial, importeDescontado, importeFinal : real;
    numPiezas : entero;
fvar

importeInicial := leerTeclado();
numPiezas := leerTeclado();

si numPiezas ≤ 3 entonces
    importeDescontado := importeInicial x (enteroAReal(tipoDescuento1)/100.0);
sino
    importeDescontado := importeInicial x (enteroAReal(tipoDescuento2)/100.0);
fsi

importeFinal := importeInicial - importeDescontado;
escribirPantalla(importeFinal);
```

Dato inicial.

Datos salida.

Podemos agrupar todas las variables reales en una sola línea.

Obtenemos el importe inicial (sin el descuento aplicado) de la compra.

Obtenemos el número de piezas vendidas.

Estructura alternativa que aplica un descuento u otro según el número de piezas vendidas.

¿Qué pasaría si...

...no hubiésemos definido los dos tipos de descuento como constantes? El cálculo del descuento habría tenido que ser:

```
si numPiezas ≤ 3 entonces
    importeDescuento := importeInicial x (25/100);
sino
    importeDescuento := importeInicial x (50/100);
fsi
```

Estructura alternativa que aplica un descuento u otro según el número de piezas vendidas.

Esta opción es correcta, pero no facilita la modificación posterior del algoritmo. Porque, ¿y si en unos días decidimos mejorar la oferta y que los descuentos sean del 30 % y el 70 % respectivamente? Si tenemos definidos los descuentos como constantes, solo hay que cambiar el valor de las constantes, sin revisar ninguna otra línea de código, De lo contrario, habrá que revisar cuidadosamente todo el algoritmo. Esto puede parecer poco importante, porque el algoritmo en este caso tiene muy pocas instrucciones, pero los programas suelen tener miles y miles de líneas. Así que esta medida se convierte en una buena práctica casi imprescindible si queremos trabajar de modo eficiente.

Por lo tanto, un algoritmo aún mejor hecho sería:

```
const
    tipoDescuento1 : entero := 25;
    tipoDescuento2 : entero := 50;
    límitePiezas : entero := 3;
fconst

var
importeInicial, importeDescontado, importeFinal : real;
numPiezas : entero;
fvar

importeInicial := leerTeclado();
numPiezas := leerTeclado();

si numPiezas ≤ límitePiezas entonces
    importeDescontado := importeInicial x (enteroAReal(tipoDescuento1)/100.0);
sino
    importeDescontado := importInicial x (enteroAReal(tipoDescuento2)/100.0);
fsi

importeFinal := importeInicial - importeDescontado;
escribirPantalla(importeFinal);
```

Así, si decidimos que ahora aplicaremos el descuento a partir de 2 piezas, cambiarlo será mucho más sencillo.

...si fuera necesario definir un sistema de reducción más granulado de rebajas como, "entre una y dos piezas aplicar un 10 %, entre 3 y 5 piezas aplicar un 20 % y para más de 6 piezas aplicar un 30 %", sería:

```
si numPiezas ≤ 2 entonces
    importeDescontado := importeInicial x (10/100);
sino
        si (numPiezas ≤ 5) entonces
            importeDescontado := importeInicial x (20/100);
        sino
            importeDescontado := importeInicial x (30/100);
        fsi
fsi
```

Como ha sido necesario hacer en este ejemplo, las estructuras se pueden anidar una dentro de otra tantas veces como sea necesario.

En este caso, para facilitar la legibilidad del algoritmo es muy importante identar correctamente las estructuras, de lo contrario costaría entender el flujo de ejecución. Esto puede parecer poco relevante, pero es muy importante. Por esto, los editores de los lenguajes de programación respetan siempre los colores que indican las palabras clave y la identación de las estructuras.

O aún mejor si utilizamos constantes para los diferentes tipos de descuento y así los podremos modificar fácilmente si hay que variar más adelante las condiciones de las rebajas:

```
const
    tipoDescuento1 : entero := 10;
    tipoDescuento2 : entero := 20;
    tipoDescuento3 : entero := 30;
    límitePiezas1 : entero := 2;
    límitePiezas2 : entero := 5;
fconst

var
importeInicial, importeDescontado, importeFinal : real;
numPiezas : entero;
fvar

importeInicial := leerTeclado();
numPiezas := leerTeclado();

si numPiezas ≤ límitePiezas1 entonces
    importeDescontado := importeInicial x (enteroAReal(tipoDescuento1)/100.0);
sino

si numPiezas ≤ límitePiezas2 entonces
    importeDescontado := importeInicial x (enteroAReal(tipoDescuento2)/100.0);
sino
    importeDescontado := importInicial x (enteroAReal(tipoDescuento3)/100.0);
fsi
fsi

importeFinal := importeInicial - importeDescontado;
escribirPantalla(importeFinal);
```

Estructuras alternativas más elaboradas

Hemos visto dos variaciones de la estructura alternativa:

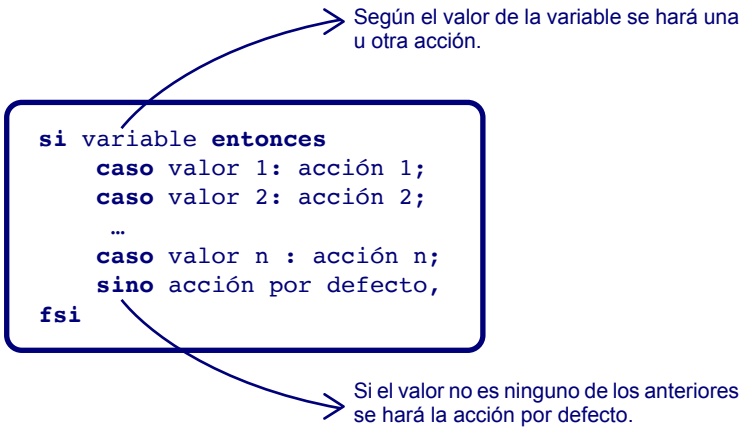
- Hacer o no hacer (la versión simplificada que nos permite realizar una desviación en el camino): **Si**.
- Hacer una cosa o hacer otra (la versión completa que nos permite dos caminos alternativos): **SI-SINO**.

También, que la estructura alternativa da juego a muchas combinaciones de la ejecución del flujo de un programa cuando realizar una acción depende de una determinada condición, como ya hemos visto. A veces incluso necesitaremos usar una (o varias) estructuras alternativas una dentro de la otra. Es lo que se conoce como estructuras **imbricadas** o **anidadas** que nos permiten definir varios caminos alternativos.

Existe, sin embargo, otra variante que sirve solo para casos muy concretos en los que se puede ahorrar tener que anidar varias estructuras alternativas, como sería el caso del ejemplo anterior. Se trata de elegir entre diferentes alternativas posibles dependiendo del valor de una variable de tipo entero o caracter:

- Realizar una de estas acciones según el caso (selección de diferentes opciones): **SI-CASO-SINO.**

La sintaxis de esta estructura es la siguiente:



Ejemplo

Queremos hacer un algoritmo que simule una calculadora que realiza las operaciones aritméticas elementales (suma, resta, multiplicación y división) a partir de dos números dados a y b enteros positivos.

```
var
   a, b : entero
   r : entero;
   operación: carácter;
fvar

a := leerTeclado();
b := leerTeclado();
operación := leerTeclado();

si (operación) entonces
   caso 'S': r = a + b;
   caso 'R': r = a - b;
   caso 'M': r = a x b;
   caso 'D': r = a div b;
   sino
fsi
```

Según el valor de la variable se realizará una u otra acción.

Si la operación que nos piden no es ninguna de estas, no hay que hacer nada.

Esta estructura “busca” qué acción se debe realizar entre las diferentes opciones (casos) que se describen. La opción **sino** indica qué hacer si el valor de la variable no coincide con ninguno de los casos detallados.

Cuando se utiliza esta estructura, el flujo del programa deja de comprobar más opciones tan pronto como encuentra una respuesta positiva. Es importante pues relacionar los casos en el orden más adecuado para que el algoritmo sea lo más eficiente posible.

3. Estructura iterativa

Con la estructura condicional se puede resolver un mayor número de problemas, pero aún tenemos que ir más allá. A menudo, se requiere que un algoritmo repita una acción (o un conjunto de acciones) varias veces hasta que se cumpla una determinada condición. Para ello, se utiliza la estructura **iterativa** o **repetitiva**.

Este concepto de acciones repetidas se representa con la idea de **bucle**. Y se ilustra muy bien con un montón de algoritmos que seguimos de un modo cotidiano: beber agua mientras tenemos sed, hacer la PAC hasta que hayamos terminado, subir la montaña hasta llegar a la cima, repetir la serie de abdominales tres veces al día, correr hasta llegar a la línea de meta, bajar escalones hasta llegar al pie de la escalera; y así, muchos otros.

Utilizaremos la composición iterativa cuando queramos repetir una acción que ya se ha hecho (es decir, volver a una acción anterior) y ejecutarla de nuevo, por eso también se llama composición repetitiva. O, dicho de otro modo, cuando queramos variar el flujo de ejecución de nuestro algoritmo para poder retroceder y ejecutar varias veces (**iteraciones**) una parte del código.

Sintaxis de la estructura iterativa o repetitiva

Algorítmicamente, la **estructura iterativa** (repetir varias veces una parte del código) se representa de este modo:

```
mientras condición hacer
   acción;
fmientras
```

Condición que hay que comprobar (evaluar) para saber si tenemos que hacer o no las acciones de dentro del bucle.

Acción que hay que repetir mientras se cumpla la condición.

Cada "vuelta" que hacemos en el bucle la llamamos **iteración**. La estructura iterativa permite repetir la ejecución del bucle mientras se cumpla la condición lógica. Esta condición se verifica al principio de cada iteración. Si la primera vez ya no se cumple la condición, no se ejecuta ninguna iteración y se sigue el flujo del algoritmo por la siguiente acción tras el bucle, en el caso del ejemplo, la acción **escribirPantalla**. Una vez que se deja de cumplir la condición "se sale" del bucle y se sigue el flujo del algoritmo por la acción que tras el bucle (a continuación del **fmientras**).

Ejemplo

Ahora que conocemos la sintaxis de la estructura iterativa, finalmente podemos acabar de diseñar el algoritmo de Euclides para calcular el mcd entre dos números.

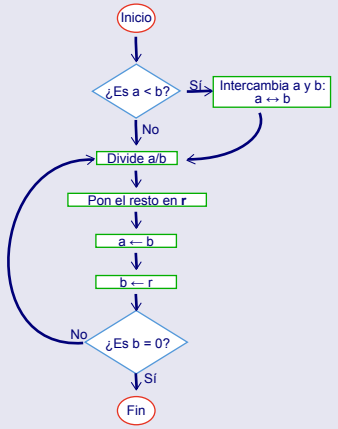
Nos habíamos quedado en este punto:

```
var
   dividendo: entero;
   divisor: entero;
   residuo: entero;
   aux: entero;
fvar

dividendo := leerTeclado();
divisor := leerTeclado();

si (dividendo < divisor) entonces
   aux := dividendo;
   dividendo := divisor;
   divisor := aux;
fsi
```

Variables de entrada y salida y variables auxiliares definidas.

Intercambio de valores, si es necesario al principio para empezar a calcular.

Nos queda hacer el cálculo para obtener el mcd. Recordemos cómo se hace:



Hacemos los cálculos que deberán repetirse:

```
residuo := dividendo mod divisor;
dividendo := divisor;
divisor := residuo;
```

Y la expresión que regulará el número de veces que se repetirá:

```
divisor ≠ 0
```

Así que ya podemos completar el bucle:

```
mientras divisor ≠ 0 hacer
   residuo := dividendo mod divisor;
   dividendo := divisor;
   divisor := residuo;
fmientras
```

Condición de repetición.

Acciones que se van repitiendo.

Fijaos en que el número de iteraciones que realizará el bucle dependen del valor de los dos números de los que tengamos que calcular el mcd.

Cuando el divisor sea 0 ya no hay que seguir dividiendo, el mcd habrá quedado en la variable *dividendo*. Este será el resultado que mostraremos:

```
escribirPantalla(dividendo);
```

Y ahora será el algoritmo completo para calcular el MCD entre dos enteros positivos:

```
algoritmo
var
   dividendo : entero;
   divisor : entero;
   residuo : entero;
fvar

dividendo := leerTeclado();
divisor := leerTeclado();

si (dividendo < divisor) entonces
   aux := dividendo;
   dividendo := divisor;
   divisor := aux;
fsi
mientras divisor ≠ 0 hacer
   residuo := dividendo mod divisor;
   dividendo := divisor;
   divisor := residuo;
fmientras
escribirPantalla(dividendo);
falgoritmo
```

Estructura secuencial.

Estructura condicional.

Estructura repetitiva.

La estructura iterativa representa una parte del algoritmo (de código, diríamos, si estamos hablando de un lenguaje de programación) que se debe ir repitiendo mientras una condición sea cierta o bien un número determinado de veces.

Con la estructura iterativa **mientras** podemos representar cualquier repetición que necesitemos en un algoritmo, es la estructura iterativa por antonomasia.

El formato es similar al de la estructura condicional simple. Y, del mismo modo, también necesita evaluar en primer lugar una condición lógica, esto es una expresión que necesariamente tiene que dar como resultado un valor booleano. Si el valor es cierto, se ejecutan las acciones dentro del bucle, si no, se continúa a partir de la siguiente acción después del **fmientras**. De nuevo, se confirma la importancia de las expresiones algebraicas en las estructuras algorítmicas.

Ejemplo

Otro "juego" matemático es contar las cifras de un número entero. Así sabremos, por ejemplo, cuántos caracteres (cuánto espacio) serán necesarios si queremos escribirlo en un formulario:

1 Entendemos el problema

El dato de entrada es el número entero a escribir. La salida es el número de dígitos (caracteres) que ocupará cuando lo escribamos en pantalla.

2 Pensamos cómo resolverlo

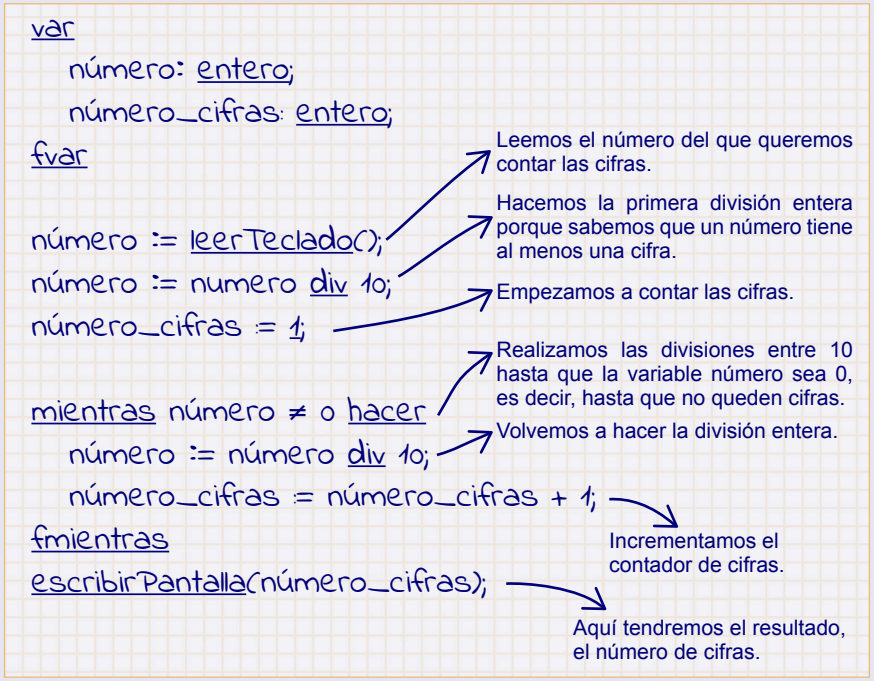
Decidimos los cálculos a realizar para obtener lo que pide el enunciado.

La división entera de un número entero cualquiera entre 10 da el mismo número sin la cifra de las unidades (la de más a la derecha). Por ejemplo: 3943 div 10 = 394.

Este hecho nos permite obtener la solución repitiendo las divisiones hasta que el resultado sea 0. En efecto, llegará un momento en el que solo tendremos la división 3 div 10 = 0.

3 Hacemos el algoritmo

Ya que debemos repetir la división entera varias veces hasta que el número sea cero, usaremos la composición iterativa **mientras**.



Simulando el algoritmo para el número 3984, se puede verificar que hacemos en total 10 acciones, ya que las dos acciones del bucle se repetirán 3 veces cada una.

En este ejemplo se observan algunos aspectos interesantes de la estructura iterativa. Y es que, en muchos casos, se necesita una variable que hace de **contador** (en nuestro ejemplo es la variable *número_cifras*). Si es el caso, hay que tener en cuenta estos aspectos:

- El **nombre de la variable contador**, procurar que sea apropiado y claro.
- Hay que **inicializar el contador**, normalmente antes de empezar la estructura iterativa.
- Es necesario **ir incrementando el contador** dentro de la estructura iterativa.

¿Qué pasaría si...

...la condición del bucle nunca es cierta?
En este caso, las acciones dentro del bucle nunca se realizarán, y esto no tiene sentido. Sería como dibujar un camino que no haya manera de transitar.
Por lo tanto, ¡es muy importante asegurar que al menos la condición del bucle se cumplirá una vez!

Y si la condición del bucle nunca es falsa?
En este caso, tendríamos un bucle infinito y el algoritmo nunca terminaría. Así que, ¡cuidado! Cuando construimos composiciones iterativas es imprescindible comprobar que el bucle terminará en algún momento. Asegurar la **condición final de la iteración** es un aspecto que conlleva muchos quebraderos de cabeza a los programadores por los efectos tan nefastos que tiene si no se maneja bien, y que pueden hacer que el programa "se cuelgue".

Otra estructura iterativa

La estructura **mientras** permite organizar prácticamente cualquier conjunto de instrucciones que haya que repetir, pero no siempre es la estructura más adecuada:

- Repite unas acciones mientras se cumpla una cierta condición (bucle controlado por una condición): **MIENTRAS**.

En algunos casos, es óptimo utilizar otra estructura iterativa llamada **PARA**:

- Repetir unas acciones un número determinado de veces (bucle controlado por un contador): **PARA**.

Ejemplo

El profesor de matemáticas está repartiendo los exámenes del segundo semestre que ya ha corregido a los estudiantes de la clase. Son 25, por lo tanto, sabe seguro que tendrá que repartir 25 exámenes.

Se podría resolver con una estructura **mientras**:

```
i := 1
mientras i ≤ 25  hacer
   Leer el nombre de la posición i de la lista;
   llamar al alumno con ese nombre;
   darle su examen;
   i := i+1;
fmientras
```

Pero, dado que en este caso sabemos el número de iteraciones que se van a realizar, es más sencillo utilizar la estructura **para** ya que no es necesario inicializar ni aumentar el contador.

```
para i := 1 hasta 25 hacer
   Leer el nombre de la posición i de la lista;
   llamar al alumno con ese nombre;
   darle su examen;
fpara
```

El bucle se va repitiendo desde i=1 hasta i=25, el contador i, se incrementa automáticamente.

Se usa la construcción **para** cuando se sabe el número de veces que se tiene que repetir el código. Es una estructura muy útil para trabajar con secuencias de datos, como que se verá más adelante.

Una construcción **para** siempre se puede hacer usando una composición **mientras**, pero al revés no sucede. Por eso decimos que la construcción **mientras** es más general que la construcción **para**.

La forma que hemos visto de la composición iterativa **para** (que en cada vuelta incrementa el contador en una unidad) es la más habitual. Pero no responde a todas las situaciones con las que nos podemos encontrar. Por ejemplo, si queremos simular una cuenta atrás desde un valor determinado hasta 0 o sumar todos los números pares entre dos números iniciales.

Claro que ambos problemas los podríamos resolver con una estructura **mientras** porque para poder hacerlo con una estructura **para**, a pesar de conocer la cantidad de vueltas que hay que realizar, es necesario poder indicar a la composición que, si hacemos una cuenta atrás, nos interesaría ir bajando el valor del contador y no subirlo y, si queremos sumar pares, habría que incrementar el contador de dos en dos unidades.

La composición **para** permite también indicar cómo queremos que varíe el contador en cada iteración.

La sintaxis de la estructura completa **PARA** es la siguiente:

Esta parte es opcional, si no se indica se entiende que el índice se incrementa de uno en uno.

```
para índice := valor inicial hasta valor final (incremento índice) hacer
acciones;
fpara
```

En las acciones no es necesario incrementar el índice.

Ejemplo

Queremos programar una cuenta atrás. Una opción sería la siguiente:

```
var
   i: entero;
   contadorInicial: entero;
fvar
   contadorInicial := leerTeclado();

para i := contadorInicial hasta 0 (-1) hacer
   escribirPantalla(i);
fpara
```

Indicamos que hay que ir decrementando el contador de uno en uno.

Por último, las estructuras iterativas (como sucedía con las condicionales) se pueden combinar con otras estructuras iterativas y alternativas, utilizando una (o varias) estructuras una dentro de la otra. Esto se conoce como estructuras **imbricadas** o **anidadas** que nos permiten definir muchísimas combinaciones de flujos de ejecución.

Estructura de un algoritmo

Un algoritmo (y un programa también) sigue una estructura predeterminada para ir escribiendo todos los elementos de modo ordenado, asignando un lugar concreto a cada uno de ellos. Por lo tanto, todos siguen la misma estructura y esto los hace más fácilmente comprensibles. Los elementos que incluye un algoritmo ya sabemos que son de dos tipos, básicamente: relacionados con los datos y relacionados con las instrucciones.

En primer lugar, lógicamente, siempre se describe la parte relacionada con los datos (declaración de variables, constantes y tipos). De este modo, el algoritmo sabe qué datos podrá utilizar en las instrucciones, que se detallan siempre posteriormente y que forman el **cuerpo** del algoritmo:

```
algoritmo
    Bloque de definición de datos
    Bloque de instrucciones
falgoritmo
```

A esta parte se la llama "cuerpo" del algoritmo.

Además, para indicar el inicio y el final de un bloque, la notación algorítmica lo denota con determinadas palabras clave. En el caso del inicio y el final de un algoritmo lo denota así:

```
algoritmo nombreAlgoritmo
falgoritmo
```

Esto es la **cabecera** del algoritmo.

Así se indica que se ha llegado al final del algoritmo.

Es importante que el nombre que damos al algoritmo sea significativo de lo que lo hace. Así, se facilita su comprensión y posterior modificación si es necesario. Es una buena práctica aconsejable aplicar las mismas recomendaciones de la nomenclatura de las variables.

Sintaxis de la estructura de un algoritmo

```
algoritmo nombre          → Cabecera.

const
fconst                    → Declaración de constantes.

var
fvar                      → Declaración de variables.

acción1;
acción2;
…
acción n:                 → Cuerpo del algoritmo.

falgoritmo                → Indicación de final del algoritmo.
```

Conceptos clave

Un algoritmo es una secuencia de acciones que se suceden para resolver un problema a partir de unos datos iniciales. Estas acciones se organizan de modo **secuencial** (una tras otra), **alternativo** (unas u otras dependiendo de una determinada condición), o **iterativo** (se van repitiendo un número concreto de veces o mientras se produce una determinada circunstancia). Así pues, tenemos tres estructuras fundamentales con las que podemos construir el flujo de ejecución del programa.

La programación estructurada es un paradigma de programación que se basa en definir un algoritmo como un número de pasos finito estructurado únicamente usando estas tres estructuras, de aquí su nombre.

La estructura alternativa tiene tres formas básicas:

- Hacer o no hacer (la versión simplificada que nos permite hacer una desviación en el camino): **Si**.
- Hacer una cosa u otra (la versión completa que nos permite dos caminos alternativos): **SI-SINO**.
- Realizar una de estas acciones según el caso (selección entre diferentes opciones): **SI-CASO-SINO**.

La estructura iterativa tiene una forma por excelencia que cubre prácticamente todos los casos:

- Repetir unas acciones mientras se cumpla una determinada condición (bucle controlado por una condición): **MIENTRAS**.

Sin embargo, cuando conocemos seguro el número de iteraciones que deberán hacerse, es más práctico utilizar esta otra variante:

- Repetir unas acciones un número determinado de veces (bucle controlado por un contador): **PARA**.

En ambas estructuras (alternativa e iterativa) las **expresiones** y los valores booleanos desempeñan un papel imprescindible, ya que las usamos para definir las condiciones de ejecución de una rama u otra en el caso de la estructura condicional y las repeticiones que se deben realizar en el caso de la estructura iterativa.

En el caso de la estructura iterativa es muy importante asegurar que el bucle se ejecuta al menos una vez y también que termina de algún modo (**condición de final de iteración**), de lo contrario entraríamos en un bucle infinito y el programa se colgaría. Esto se logra teniendo cuidado en cómo se define la condición que controla la ejecución del bucle.

Con estas estructuras podemos indicar múltiples modos de ejecución del flujo de un programa porque, además, podemos incluir unas dentro de otras según convenga (un **mientras** dentro de un **si**, un **mientras** dentro de otro **mientras**, etc.). Es lo que se conoce como estructuras **imbricadas** o **anidadas**.

Para facilitar la comprensión de los algoritmos, se identifican con dos palabras clave que marcan el inicio y el final (**algoritmo** y **falgoritmo**). Así mismo, un algoritmo sigue siempre la misma **estructura** de bloques; se empieza por el bloque de datos: primero se declaran las constantes y luego las variables. De este modo, el algoritmo ya "sabe" con qué elementos puede operar (las variables y las constantes). Y, después, se incluye el cuerpo del algoritmo con todas las acciones que debe hacer combinadas convenientemente a partir de las tres estructuras. Los lenguajes de programación también siguen unas pautas de formato muy similares. Pautas que, en el caso de los programas, hay que seguir estrictamente para que después se puedan ejecutar.

Las acciones pueden ser asignaciones de valores a variables (la **acción elemental**), operaciones de datos (operaciones aritméticas en el caso de los números, por ejemplo) o llamadas a funciones ya existentes, como por ejemplo la escritura por pantalla.

¿Cómo trabajamos las estructuras en JavaScript?

Una vez se ha decidido cuál será la estructura del algoritmo que resuelve un problema determinado, sabemos ya qué estructuras utilizaremos para indicar el flujo de ejecución. Será necesario, después, especificarlo en un lenguaje de programación.

Ejemplo

Continuamos con el desarrollo del programa para calcular el mcd. En la página 14 tenemos el algoritmo ya diseñado como resultado de los tres primeros pasos:

1 Entender el problema

2 Pensar cómo resolverlo

3 Diseñar un algoritmo que detalle los pasos a seguir

```
algoritmo
var
    dividendo : entero;
    divisor : entero;
    residuo : entero;
fvar

leerTeclado(dividendo);
leerTeclado(divisor);

si (dividendo < divisor) entonces
    aux := dividendo;
    dividendo := divisor;
    divisor := aux;
fsi
mientras divisor ≠ 0 hacer
    residuo := dividendo mod divisor;
    dividendo := divisor;
    divisor := residuo;
fmientras
escribirPantalla(dividendo);
falgoritmo
```

Estructura secuencial.

Estructura condicional.

Estructura repetitiva.

4 Implementarlo en lenguaje JavaScript

El lenguaje JavaScript permite representar todas las estructuras algorítmicas que hemos visto, cada una de las cuales tiene su propia sintaxis que es importante conocer y aplicar correctamente en el momento de programar. Aun así, los principios implicados algorítmicamente son plenamente válidos y no conviene olvidarlos en ningún momento.

El código JS que seguiría la estrategia indicada en el algoritmo anterior es el siguiente:

```
1  // definimos las variables de entrada
2  // y les asignamos un valor para probar
3  var dividendo = 126;
4  var divisor = 72;
5  var aux, residuo;
6  // si el dividendo es menor que el divisor,
7  // los intercambiamos
8  if (dividendo < divisor) {
9    aux = dividendo;
10   dividendo = divisor;
11   divisor = aux;
12 }
13 // calculamos el mcd
14 while (divisor !=0) {
15   residuo = dividendo % divisor;
16   dividendo = divisor;
17   divisor = residuo;
18 }
19 //mostramos el resultado
20 console.log(dividendo);
21
```

La estructura de un programa JS también es bueno que siga este orden: primero declarar las constantes y las variables y, a continuación, añadir las instrucciones. No hay, en este caso, ningún delimitador de inicio y fin.

Pero el lenguaje JS es más flexible y permite definir y dar valor tanto a constantes como a variables en la misma línea de código. En cualquier caso, lógicamente es imprescindible que las variables y las constantes se definan antes de ser utilizadas. De lo contrario, el programa dará un error de ejecución.

Sintaxis de las estructuras alternativas en JS

La sintaxis es muy similar a la que usamos en notación algorítmica.

Hacer o no hacer (condicional simple):

```
if  (condición) {
    acción;
};
```

Haga una cosa u otra (condicional):

```
if  (condición) {
    acción1;
 }
else
{
    acción2;
};
```

Utilizamos las claves para delinear las acciones a realizar.

Hacer una cosa u otra (condicionales imbricados):

```
if  (condición1) {
    acción1;
 }
else if (condición2)
{
    acción2;
};
```

Elegir entre varias opciones:

```
switch (condición) {
    case valor :
           acción1;
           break;
    case valor :
           acción2;
           break;
    default:
           acciónDefault;
           break;
}
```

Ejemplo

```
1  const tipoDescuento1 = 25;
2  const tipoDescuento2 = 50;
3  const límitePiezas = 3;
4  //variables de entrada, les asignamos el valor
5  //directamente
6  var importeInicial = 1000;
7  var numPiezas = 4;
8  //variables intermedia y final
9  var importeDescontado, importeFinal;
10 if (numPiezas <= límitePiezas){
11 importeDescontado = importeInicial * (tipoDescuento1/100);
12 }
13 else {
14 importeDescontado = importeInicial * (tipoDescuento2/100);
15 }
16 importeFinal = importeInicial - importeDescontado;
17 console.log(importeFinal);
18
```

```
1  var a, b;
2  var resultado;
3  var operación;
4  // damos valor a las variables de entrada
5  var a = 2;
6  var b = 5;
7  var operación = "S";
8  //simulamos el funcionamiento de la calculadora
9  switch (operación){
10 case "S":
11     resultado = a + b;
12     break;
13 case "R":
14     resultado = a - b;
15     break;
16 case "M":
17     resultado = a * b;
18     break;
19 case "D":
20     resultado = a / b;
21     break;
22 default:
23     console.log("Operación no permitida");
24     break;
25 }
26 console.log(resultado);
27
```

Sintaxis de las estructuras iterativas en JS

La sintaxis también es muy sencilla:

Repetir mientras se cumpla una determinada condición:

```
while (condición) {
        acciones;
}
```

Aquí las claves también sirven para delimitar las acciones a realizar.

Ejemplo

```
1  var numero; //valor de entrada
2  var numero_cifras; // valor resultante
3  numero= 3984; // damos valor a la entrada
4  // dividimos y nos quedamos con la parte entera
5  numero = parseInt(numero/10);
6  //Inicializaremos el valor resultante
7  numero_cifras = 1;
8  while (numero > 0) {
9        numero = parseInt(numero/10);
10       numero_cifras = numero_cifras + 1;
11 }
12 console.log (numero_cifras);
13
```

Repetir mientras se cumpla una determinada condición:

```
for (índice = valor inicial; índice < valor final; incremento índice) {
   acciones;
 }
```

Ejemplo

```
1  var i; // índice
2  var contadorInicial;
3  // damos valor a la variable de entrada
4  contadorInicial = 3;
5  //simulamos la cuenta atrás
6  for (i = contadorInicial; i>0; i--){
7    console.log (i);
8  }
9  console.log(i);
10
```

Decrementa el contador en una unidad.

Encontraréis una explicación detallada del funcionamiento de todas estas estructuras en JavaScript en la guía de JavaScript de Mozilla.

Probar el código en JS

El último paso que hacemos cuando programamos es probar el código resultante para asegurar de que hace lo que realmente tiene que hacer.

Por eso, es importante pensar en una serie de **juegos de prueba**. Un juego de prueba es un conjunto de valores, uno para cada variable inicial que, ejecutando el programa, ofrece el resultado final esperado y correcto. Se pueden necesitar muchos casos de prueba para determinar si un programa es correcto o no.

Ejemplo

5 Probar el código

¿Cómo podemos probar si el código del programa JS que calcula el mcd aplicando el algoritmo de Euclides es correcto? Si retrocedemos al paso 1 (entender el problema) recordaremos cuáles eran las variables de entrada: dos números enteros positivos *a* y *b*. Y como salida del programa esperamos el resultado de calcular su mcd, otro entero positivo.

Hay muchos tipos de pruebas, las que se realizan en una parte de código concreta y pequeña se denominan **pruebas unitarias**.

Podemos hacer una tabla y pensar algunos valores para probar que funciona el código. Hay que pensar en valores "posibles" de *a* y *b* según lo que detalla el enunciado. En este caso dice que son enteros positivos, por lo tanto no es necesario probar casos como a=10 o b=0 que ya sabemos que no se producirán.

Juegos de prueba	a	b	Resultado
Caso 1	126	72	18
Caso 2	72	126	18
Caso 3	60	36	12
Caso 4	36	60	12
...			
Caso n			

Tenemos que dar valores diferentes a *a* y *b* y calcular cuál es el resultado que esperamos del código.

El número de casos a probar dependerá de la complejidad del código del programa y deben ser representativos de los casos más importantes y extremos que se puedan dar durante la ejecución del programa, para así poder verificar el comportamiento del programa en el máximo número de situaciones posibles.

Puede parecer que los casos 1 y 2 son redundantes, pero no. En el caso 2, *a es menor que b* y, por lo tanto, se ejecutará la sentencia alternativa y se intercambiarán los valores de *a* y *b*. En el primer caso, esto no era necesario. Así nos aseguramos de probar todas las ramas (los flujos de posible ejecución) del código.

Cuando alguna prueba falla (no da el resultado esperado), se puede ejecutar el código con el PythonTutor e ir ejecutando línea a línea hasta localixar dónde está el error y corregirlo.

Podéis ver un ejemplo en este vídeo.

A veces el error puede ser sintáctico (hemos escrito un "wile" en lugar de "while", o hemos olvidado un corchete para cerrar una sentencia **if**), otras semántico (hemos usado el símbolo **=** para comparar en lugar del **==**, por ejemplo).

Las pruebas, aunque puedan parecer una cuestión menor, sencilla o ya final y más rutinaria, son una parte fundamental de la programación. Si una prueba falla, hay que retroceder a los pasos anteriores y ver dónde está el problema: ¿está mal codificado?, ¿el algoritmo está mal concebido y por eso no hace lo que debe hacer?, ¿no se ha entendido el problema inicial? Cuanto más haya que retroceder para corregir el error, más costoso será corregirlo. Este es otro de los motivos por los que es importante realizar todos los pasos, uno a uno, cuando estamos programando: omitir alguno (codificar sin pensar la estrategia de solución o sin entender el problema, por ejemplo) suele dar malos resultados, al final.

A medida que los programas van creciendo y tienen más y más líneas de código, las pruebas no podrán descubrir todos los posibles errores del código desarrollado.

¿Qué pasaría si...

...hacéis las mismas pruebas con el siguiente código?

```
1  // definimos las variables de entrada y les
2  // asignamos un valor para probar
3  var dividendo = 72;
4  var divisor = 126;
5  var aux, residuo;
6  //calculamos el mcd
7  while (divisor !=0) {
8    residuo = dividendo % divisor;
9    dividendo = divisor;
10   divisor = residuo;
11 }
12 //mostramos el resultado
13 console.log(dividendo);
14
```

¡Que daría los mismos resultados! De hecho, el algoritmo de Euclides se puede simplificar, ya que si el dividendo es menor que el divisor, la primera iteración del bucle ya sirve para intercambiar los valores. Pero la primera versión del código era más clara y comprensible, ¿verdad? Esta segunda opción sería más eficiente. A veces hay que elegir entre claridad y eficiencia.

Podéis probar estos códigos en la web de PythonTutor.