Empezando a programar

PID_00268298

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

Segunda edición: julio 2020

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

Contenidos

Empezando a programar

¿Qué es programar?

Antes de entrar en materia hay que saber de qué estamos hablando.

Un **programa** es un conjunto de instrucciones que un ordenador sabe cómo ejecutar para realizar una tarea. Un ordenador puede parecer una máquina muy inteligente pero, de hecho, no sabe "pensar" en el sentido en que lo hacemos las personas, solo sabe seguir (ejecutar) un conjunto concreto de instrucciones simples, eso sí, de un modo increíblemente rápido y con gran precisión. Somos los programadores que los pensamos cuáles son estas instrucciones para hay que seguir para realizar una tarea concreta.

Y, de hecho, pensar estas instrucciones o pasos a seguir es **pensar computacionalmente** y, escribirlas de modo que el ordenador las entienda, es **programar**.

Así pues, pensar computacionalmente no es lo mismo que programar. Tampoco es pensar como un ordenador, ya que los ordenadores no saben pensar, saben ejecutar instrucciones. El **pensamiento computacional** es lo que nos permite a las personas determinar de qué modo, con qué pasos se puede hacer una tarea o resolver un problema.

Ejemplo

Imaginemos que se nos ha estropeado el horno y nos llama un técnico para venir a casa a arreglarlo. Nos pregunta cómo llegar desde su taller. Tenemos que pensar los caminos posibles y decidir cuál es la mejor para llegar a casa. El mejor puede ser el más rápido, el más corto o el más fácil, al fin y al cabo, el camino con el que tiene menos posibilidades de perderse. Y estas tres condiciones pueden no coincidir en la misma opción. Una vez decidido el mejor camino, tienes que darle las indicaciones para llegar del modo más claro y conciso posible.

Cuando piensas en las rutas que hay y cuál sería la mejor estás usando el pensamiento computacional, y cuando le das al técnico las indicaciones para llegar a tu casa, estás diseñando el algoritmo de la solución. Si, en lugar de dar las instrucciones para llegar a una persona, las tuviésemos que dar a un robot, por ejemplo, entonces deberíamos hacer un programa en algún lenguaje de programación que pudiera entender el robot. Hacer el programa no sería más que codificar el algoritmo con indicaciones escribiéndolo en el lenguaje concreto.

Un **algoritmo** es, pues, un conjunto de pasos para hacer una tarea y un **programa** es un algoritmo que se ha escrito en un lenguaje que puede entender un ordenador.

> "An algorithm is a set of rules for getting a specific output from a specific input. Each step must be so precisely defined that it can be translated into computer language and executed by machine." — Donald Knuth, un dels referents de l'algorítimica (1977)

Un algoritmo sería, por ejemplo, una receta de cocina o las instrucciones para montar un mueble paso a paso. Si la secuencia de instrucciones es correcta y las sigues a rajatabla, alcanzarás el resultado esperado, hacer una tortilla de patatas o montar una estantería. Vivimos, pues, rodeados de algoritmos.

Ejemplo

Por ejemplo, este es posiblemente el algoritmo que ejecuta un programa de la lavadora:

- Coger agua.
- Coger el jabón.
- Calentar el agua a 40 grados.
- Repetir durante 20 minutos. → Lavar.
  - Dar vueltas al tambor.
- Expulsar el agua.
- Repetir cuatro veces.
  - Coger agua.
  - Repetir durante 10 minutos. → Cada vuelta es un aclarado de ropa.
    - Dar vueltas al tambor.
  - Expulsar el agua.
- Repetir durante 1 minuto. → Centrifugado.
  - Dar vueltas al tambor muy rápidamente.

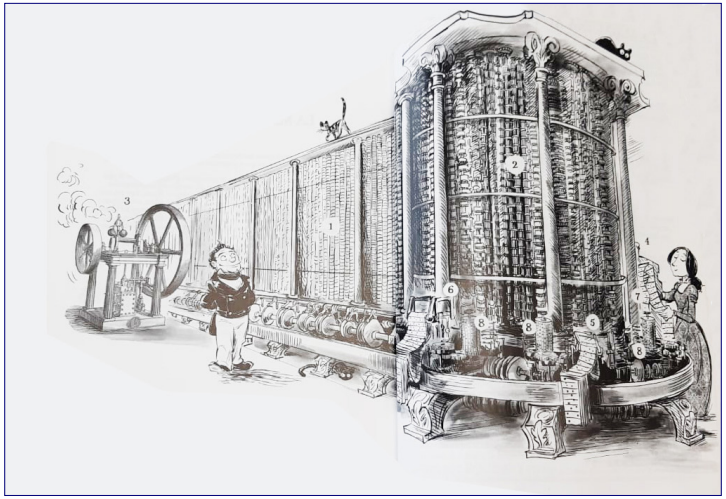
Eso sí, las instrucciones del algoritmo deben ser precisas, claras y completas. También correctas y eficientes.

Va muy bien trabajar con el concepto de algoritmo porque todo lo que explicamos relacionado con la algorítmica es válido y, en general, podemos aplicarlo más tarde a cualquier lenguaje de programación. También, porque expresar las instrucciones en notación algorítmica es más sencillo y no es necesario estar pendientes de la sintaxis más estricta de los lenguajes de programación. Nos permite, en primer lugar, centrarnos en cómo resolver el problema sin tenernos que preocupar de cómo expresarlos exactamente en un lenguaje de programación concreto..

Un poco de historia

De hecho, los primeros algoritmos se desarrollaron hace miles de años, mucho antes de que existieran los ordenadores. Uno de los algoritmos más célebres es el **algoritmo de Euclides**, para calcular los divisores de un número, que data del 300 a. C.

Así que los algoritmos nos han acompañado siempre en forma de secuencias de instrucciones que las personas seguimos para realizar una tarea. Hasta que en 1837, **Charles Babbage** imaginó una máquina capaz de ejecutar algoritmos (de seguir instrucciones) mecánicamente. Él pensaba en una máquina capaz de realizar cálculos matemáticos. Más tarde, **Ada Lovelace**, hija de Lord Byron y matemática brillante, se maravilló con la idea de la **máquina analítica** de Babbage y escribió en 1842 un programa para la máquina Babbage, y también pensó en darle las instrucciones a través de tarjetas perforadas. Por eso, se la considera la primera programadora de la historia. Ya imaginó en su día que los ordenadores (que aún ni existían) se utilizarían un día para crear música y arte. ¡Una visionaria, sin duda! Y, con ella, llega la programación.



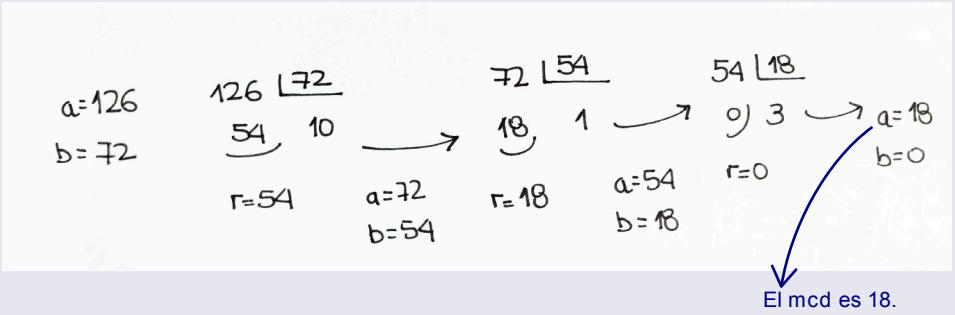
Ejemplo

El algoritmo de Euclides es el método que se utiliza para calcular el máximo común divisor (mcd) entre dos números distintos de cero.

- Datos de entrada **a** y **b**.
- El algoritmo:
  - si a < b, intercambiar a y b (a ↔ b)
  - mientras b ≠ 0 repetid las tres instrucciones siguientes:
    - r ← Residuo (el resto de la división entera) de **a** por **b** (dad a **r** el valor del residuo de **a** por **b**).
    - a ← b (el nuevo valor de **a** es el antiguo valor de **b**).
    - b ← r (el nuevo valor de **b** es el valor de **r**).
- El resultado es **a** (su último valor).

Veamos para qué nos sirve calcular el máximo común divisor entre dos números con un ejemplo extraído de un libro de matemáticas del 1.º de la ESO. Imaginemos que tenemos una colección de vehículos en miniatura (72 coches y 126 motocicletas) y acabamos de comprar unas estanterías para guardarlos. Pero queremos colocarlos todos de tal modo que haya la misma cantidad de vehículos en cada estante y, además, para no utilizar demasiado espacio, el máximo número de vehículos en cada uno. Y sin mezclar motocicletas y coches en una misma estantería. Los coleccionistas somos así, tenemos nuestras “manías”.

Calculamos el mcd (126.72):



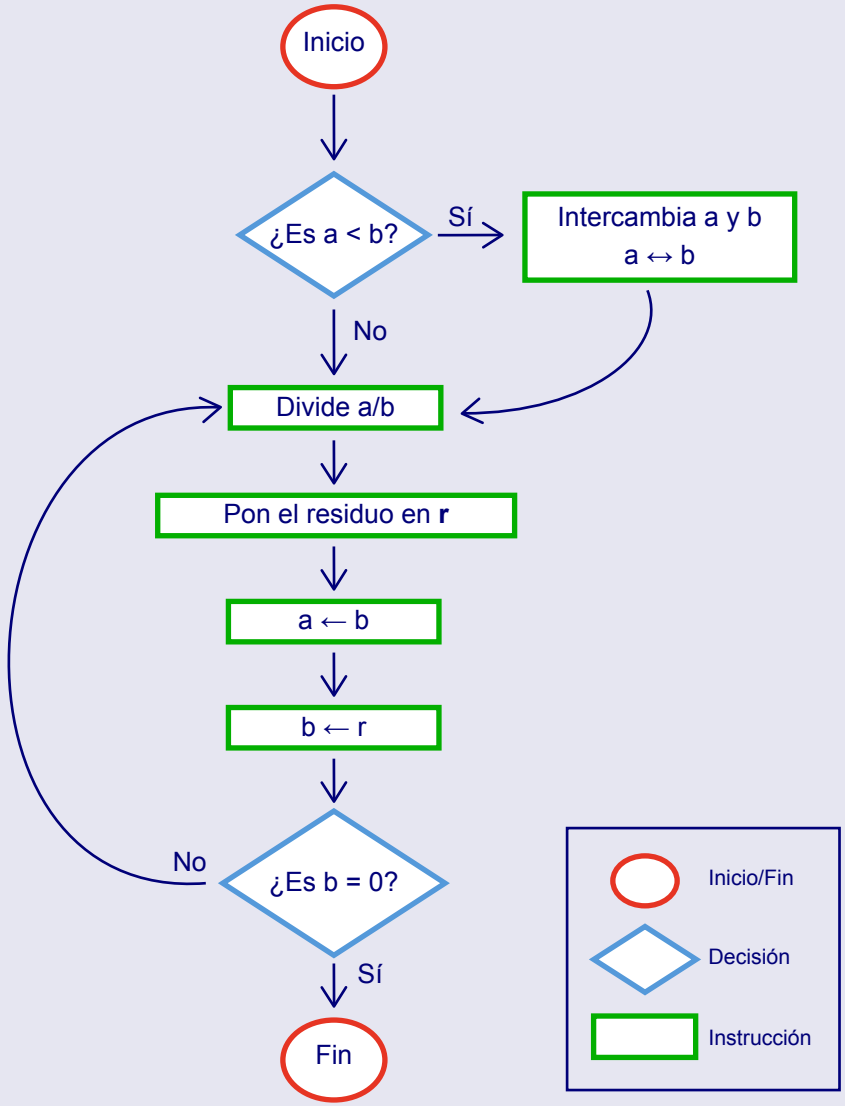
Por lo tanto, el mcd (126, 72) = 18.

Debemos poner 18 vehículos (coches o motocicletas) en cada estante. Y, ya que tenemos 126 + 72 = 198 vehículos en total, necesitaremos 198/18 = 11 estanterías, 7 estantes para coches y 4 estantes para motocicletas.

Los algoritmos se pueden describir textualmente, siguiendo alguna notación concreta como veremos más adelante y también gráficamente, por ejemplo con un **diagrama de flujo.**

Ejemplo

El algoritmo de Euclides representado como diagrama de flujo:

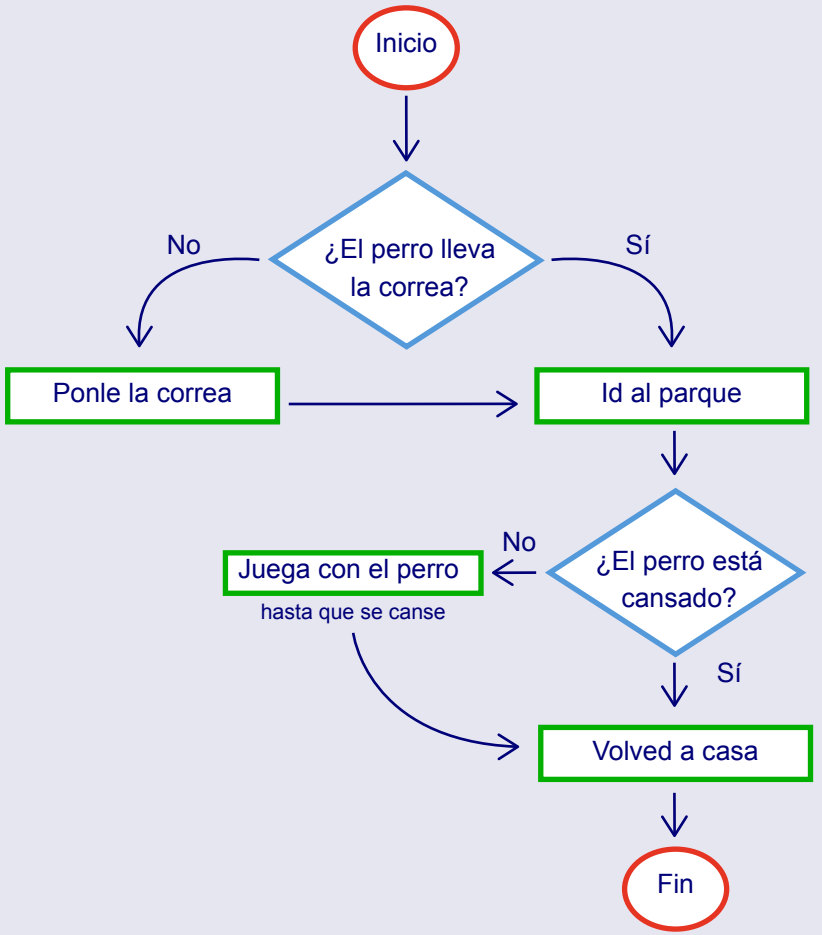


Ejemplo

Imaginad que tenéis que pedir a vuestro hijo que saque a pasear al perro. Es la primera vez que lo hace. Le daréis unas instrucciones:

"Si el perro no lleva la correa, pónsela. Id al parque. Juega con el perro hasta que se canse. Después, regresad a casa."

Que, representado como diagrama de flujo, sería:



Empezando a programar

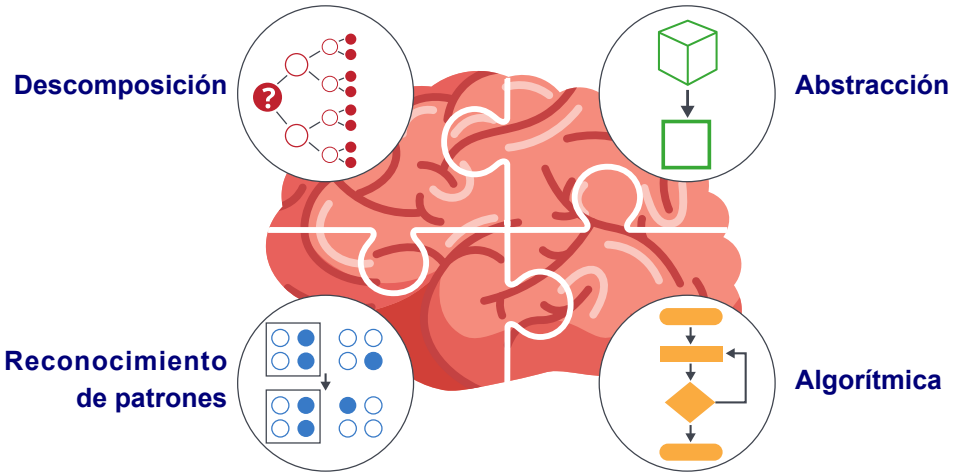
Pensamiento computacional

Los ordenadores nos ayudan a resolver problemas. Pero para resolver un problema primero hay que entenderlo y pensar posibles soluciones. A esto nos ayuda a aplicar el pensamiento computacional, que nos permite primero entender un problema complejo y, después, resolverlo paso a paso de un modo suficientemente claro y conciso para que lo pueda entender otra persona y también un ordenador.

Se fundamenta en cuatro técnicas, básicamente. Todas ellas ayudan a gestionar la complejidad de los problemas:

Descomposición	Dividir un problema complicado, que no sabemos cómo resolver directamente, en partes más pequeñas y más fácilmente solucionables.
Reconocimiento de patrones	Buscar similitudes entre diferentes problemas para aprovechar lo que se sabe para resolver uno y aplicarlo al otro.
Abstracción	Simplificar, ignorando los detalles innecesarios para quedarnos con la esencia y fijarnos solo en lo relevante en relación con el problema.
Algorítmica	Diseñar los pasos uno a uno para resolver un problema. Por muy mágico que parezca, todo lo que queramos que haga un ordenador se puede escribir a partir de unas instrucciones concretas y sencillas que se combinan entre ellas en forma de secuencia, repetición y selección.

Tener destreza en estas técnicas ayuda a programar porque facilita dividir un problema complejo en una serie de problemas más simples y manejables (descomposición) que se pueden resolver uno a uno individualmente, teniendo en cuenta cómo se han solucionado problemas similares anteriormente (reconocimiento de patrones) y centrándonos solo en los detalles relevantes del problema (abstracción). A continuación, se puede definir el plan de acción, entendido como pasos a seguir para resolver cada uno de los problemas (hacer el algoritmo). Por último, estos pasos serán el esquema de base para codificar el programa en un lenguaje de programación concreto con el que un ordenador resolverá el problema.



El **pensamiento computacional** se hizo popular después de un artículo de Jeannette Wing de 2006, que lo postulaba como una habilidad fundamental para todas las disciplinas, más allá de la programación. El término, sin embargo, fue usado por primera vez por Seymour Papert ya en 1980, casi tres décadas antes.

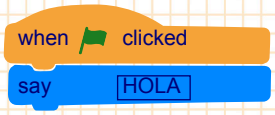
Las técnicas de pensamiento computacional son útiles también en muchas otras disciplinas y de hecho en nuestra vida diaria siempre que sea necesario resolver problemas complejos. De ahí que se considere una de las competencias del siglo XXI.

Empezando a programar

Lenguajes de programación

A la hora de programar, podemos escoger entre muchos lenguajes, algunos más adecuados que otros para según qué tipos de programas o aplicaciones, algunos más sofisticados y complicados de aprender y otros más eficientes para según qué.

C	Uno de los más utilizados. Es muy potente por ejemplo para hacer *software* que necesita funcionar rápido, como los sistemas operativos. Pero también es complejo de aprender.	`#include <stdio.h>`
		`main()`
		`{`
		`  printf("HOLA");`
		`}`

BASIC	Uno de los más antiguos, ya no se utiliza.	`10 PRINT "HOLA"` `20 GOTO 10`
SCRATCH	Es visual, en vez de escribir instrucciones hay que ir combinando diferentes bloques de código para construirlas. Por esto es muy adecuado para aprender a programar, especialmente para los niños.	

JAVA	Es muy versátil ya que permite que un programa funcione con diferentes sistemas operativos y pueda ejecutarse tanto en ordenadores, teléfonos móviles y tabletas sin ningún cambio.	`class Hola, WorldApp{`
		`  public static void main (String{}args){`
		`  System.out.println("HOLA");`
		`  }`
		`}`

JAVASCRIPT	Es muy útil crear código que funcione dentro de un HTML, por eso se utiliza para dar interactividad a las páginas web y solicitar información al usuario, por ejemplo. Actualmente también se pueden crear servidores, aplicaciones web y de escritorio que funcionan con JavaScript.	`console.log('HOLA');`
PYTHON	Es más reciente, muy popular y versátil. Se utiliza para diferentes propósitos. Es simple y relativamente fácil de aprender.	`print("HOLA")`

PHP	Se utiliza junto con el HTML para diseñar páginas web.	`<?php`
		`echo "HOLA";`
		`?>`

Los lenguajes de programación, al igual que los idiomas, están "vivos", evolucionan constantemente. Así que hay que saber siempre dónde encontrar la documentación de referencia, las librerías existentes, los manuales más actuales, y los foros de dudas, para estar al corriente de los últimos cambios.

Empezando a programar

Ser un buen programador

Convertirse en un buen programador, en un experto, requiere tiempo, como sucede con cualquier disciplina. Y se necesita mucha práctica y entrenamiento. A programar se aprende programando: probando, equivocándose, corrigiendo errores, preguntando, curioseando y analizando otros programas, pensando nuevas soluciones, reescribiendo código, buscando información, aprendiendo nuevos lenguajes, etc.

Esto no significa que la programación sea un tipo de arte que dependa de la inspiración del momento, y de probar y probar hasta que la cosa funcione. No. Es cierto que hay que husmear, tener curiosidad, ganas de probar, dedicarle tiempo. También que la creatividad y la persistencia aceleran e incrementan el aprendizaje y, si además se disfruta programando y cada programa se afronta como un reto, se alcanza un nivel de destreza importante mucho más rápido.

Pero también es cierto que es necesario aprender y consolidar las técnicas de programación básicas, la algorítmica, en definitiva. Aprenderlas bien desde el principio y también consolidarlas, y esto se hace con la práctica y con el tiempo.

En concreto, dado que un programa combina instrucciones y maneja datos para resolver un problema determinado, los elementos básicos de la programación que necesitamos son dos:

Esto lo trabajaremos en las unidades 2, 5 y 6.

1. Saber cómo representar los datos que tenemos que manejar.
2. Saber qué instrucciones resolverán el problema y cómo las tenemos que combinar.

Esto lo aprenderemos en las unidades 3, 4 y 7.

Pero, además de aprender los elementos y las técnicas algorítmicas y saberlas utilizar convenientemente, debemos incorporar buenas prácticas en nuestro quehacer que nos ayudarán a lograr buenos programas: correctos, claros y eficientes.

Y con esto tampoco será suficiente, tendremos que conocer muy bien la sintaxis y los intringulis del lenguaje de programación que utilizaremos, saber consultar los manuales de referencia adecuados en caso de duda y, finalmente, probar bien el programa para asegurarnos y comprobar que hace lo que tiene que hacer.

En definitiva, tenemos que poner en juego a partes iguales una buena técnica algorítmica, una serie de buenas prácticas, la inspiración, el pensamiento computacional, el conocimiento fino del lenguaje (en este caso el JS) y la persistencia. La buena notícia es que todo esto se puede aprender y mejorar.

Buenas prácticas

Para obtener un buen programa, antes hay que entender qué es lo que se debe resolver, a continuación, antes de lanzarse a escribir instrucciones sin ton ni son y probar si por casualidad la cosa funciona, pensar cómo resolverlo, cómo llegaremos a calcular u obtener lo que nos piden, y finalmente programarlo en JavaScript de modo muy preciso (ya sabéis que las computadoras no piensan y solo entienden exactamente lo que escribimos, cualquier pequeño lapsus no lo sabrán interpretar). Finalmente, tenemos que probar lo que hemos hecho para ver si hace lo que tiene que hacer. Lógico, ¿no?

Así, un buen hábito para incorporar buenas dinámicas de programación será seguir siempre estos pasos:
1. Entender el problema, 2. Pensar como resolverlo, 3. Hacer un borrador de la solución (algoritmo), 4. Implementarlo en un lenguaje de programación y 5. Probar el código.

1 Entender el problema | 2 Pensar cómo resolverlo | 3 Hacer un borrador de la solución (algoritmo) | 4 Implementarlo en un lenguaje de programación | 5 Probar el código

1 Entender el problema

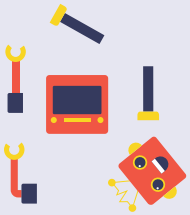
Es esencial asegurarnos de que entendemos el encargo que nos dan (el enunciado), en definitiva, que entendamos qué problema tiene que resolver el programa. Y, aunque parece muy evidente, este paso no es tan simple. A veces, el enunciado no es lo suficientemente clao, es ambiguo o no nos da toda la información que necesitamos. Si este es el caso, tenemos que repreguntar hasta que no haya ninguna duda. Otras veces no lo interpretamos bien, por lo que también es una buena estrategia comprobar que lo hemos entendido, contrastándolo con quien nos facilita el enunciado.

¿Cómo podemos hacerlo?
Leyendo el enunciado y preguntándonos:
1. ¿cuáles son los datos que tenemos de partida?, y
2. ¿qué datos debemos obtener como solución?

En definitiva, un programa toma algunos datos iniciales, los transforma y obtiene unos resultados, así que teniendo claras estas preguntas ya tenemos mucho ganado.

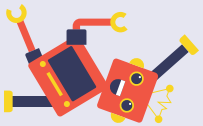
Ejemplo

Enunciado:
"Se nos ha desmontado el robot.
Tenéis que pegar todas las piezas":



Este sería un ejemplo de enunciado poco preciso. Podemos deducir que significa montarlo:

Pero también podemos no interpretarlo bien:



Este resultado también respondería al enunciado, ¿no? El robot está montado, pero mal, ¿lo veis?

Un enunciado más correcto sería: "Se nos ha desmontado el robot. Tenéis que montar cada pieza en su lugar correspondiente". Aquí ya sí que la primera solución sería correcta pero la segunda no.

Estos son los datos que nos da el enunciado.

Datos de entrada: cada pieza del robot por separado.
Datos de salida: un robot con todas las piezas conectadas en el lugar que corresponde.

Es la salida esperada de nuestro programa.

Ejemplo

Enunciado:
Calcula la suma de los cinco primeros números enteros.

Este enunciado es claro y conciso. No admite dudas ni confusiones, ¿no?
Datos de entrada: 1, 2, 3, 4, 5 (los cinco primeros números enteros).
Datos de salida: la suma de estos cinco números.

En definitiva, si el enunciado no está claro, o no lo entendemos bien, se tiene que aclarar antes de proseguir.

2 Pensar cómo resolverlo

Una vez tenemos claro lo que hay que resolver, hay que pensar en cómo hacerlo. Buscar la solución, en definitiva, pensar la estrategia (pasos a seguir) para llegar a obtener los datos que nos pide el enunciado a partir de los que nos facilita. Y de estrategias puede haber más de una. Si se da el caso, elegimos el más sencilla y clara.

Ejemplo

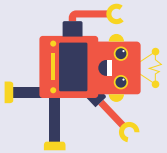
¿Cómo montamos el robot?

Una estrategia posible sería:
"Primero pegar la cabeza al cuerpo, luego pegarle los brazos y finalmente pegar las piernas".

Otro modo sería:
"Primero pegar las piernas en el cuerpo, segundo pegar los brazos y finalmente pegar la cabeza."

De los dos modos montaremos el robot, el resultado será el mismo. Sin embargo, ¿lo montaré correctamente? Quizá no.

Ambas soluciones son poco precisas y pueden dar lugar a:



Precisemos un poco más:
"Pegar la cabeza sobre el cuerpo, luego pegar los brazos, uno a cada lado del cuerpo y finalmente pegar las piernas una a cada lado, debajo del cuerpo."

Ahora sí que tenemos claro cómo montar el robot correctamente.

Ejemplo

Enunciado:
Calcula la suma de los cinco primeros números enteros.

Datos de entrada: 1, 2, 3, 4, 5 (los cinco primeros números enteros).
Datos de salida: la suma de estos cinco números.

Una estrategia sería calcular la suma de 1 + 2 + 3 + 4 + 5.

Otra estrategia posible sería sumar 5 + 4 + 3 + 2 + 1. Ya sabemos que la suma es conmutativa, por lo que el orden de los sumandos no tiene importancia. Pero es más clara la primera opción, ¿no?

Así que, si tenemos varias maneras de resolver el problema, elegiremos siempre la más sencilla y clara. Además, conviene recordar que los ordenadores no "piensan", siguen instrucciones específicas, que tienen que ser precisas y simples. Así que el programa solo hará lo que se indique explícitamente en las instrucciones.

3 Hacer un borrador de la solución (algoritmo)

Una vez hemos pensado cómo hallar la solución, debemos hacer un borrador de las instrucciones detalladas que nos conducirán a la salida esperada partiendo de los datos de entrada. Y para ello solo podemos usar instrucciones que una máquina pueda entender (recordad que estamos haciendo un programa), esto significa que solo podemos utilizar los elementos de la algorítmica. Y de ahí la importancia de conocer la algorítmica a fondo.

Ejemplo

Aunque aún no conocéis los elementos de la algorítmica, veamos cómo sería el algoritmo que resuelve la suma de los cinco primeros números:

```
algoritmo sumaPrimerosCinco

var
suma : entero;
fvar

suma := 1+2+3+4+5;

escribirPantalla (suma);

falgoritmo
```

Preparamos los datos con los que trabajará el algoritmo.

Este algoritmo tiene dos instrucciones que se ejecutan una tras otra (secuencialmente):
1) calcular la suma; y
2) mostrar el resultado por pantalla.

Se entiende lo que hace, ¿verdad? Un algoritmo es, pues, un conjunto de instrucciones expresadas de modo que se puedan seguir claramente para realizar una tarea, en este caso sumar los cinco primeros números enteros.

Esta sería la solución al problema ya expresada en notación algorítmica.

¿Por qué tenemos que realizar este paso? ¿Con el anterior no sería suficiente? No, tenemos que encontrar una solución al problema, pero tiene que ser una **solución algorítmica**, sino, no la podremos codificar posteriormente en un lenguaje de programación. Es por eso que se debe poder expresar esta solución en términos algorítmicos, hacer el algoritmo, en definitiva. Eso sí, la solución se expresa en una notación sencilla que permite escribir las instrucciones de manera simple y muy natural, en castellano en este caso.

Escribir el algoritmo es un buen modo de planificar la solución antes de codificarla en un u otro lenguaje de programación concreto.

Para escribir algoritmos, definiremos una **notación algorítmica**, que será un modo sencillo de describir conjuntos de instrucciones sin utilizar una sintaxis específica de ningún lenguaje de programación en concreto. Es una notación absolutamente arbitraria pero que es parecida a la de un lenguaje de programación. Esta notación nos permitirá "entender" entre nosotros los algoritmos que hacemos, ya que todos los expresamos del mismo modo. Revisando la bibliografía encontraréis que hay tantas notaciones algorítmicas como referencias, pero todas son muy parecidas porque tienen los mismos componentes, los elementos fundamentales de la algorítmica.

Conocer la notación algorítmica nos permite, además, representar la solución al problema en pseudocódigo sin preocuparnos por la sintaxis y las particularidades del lenguaje en el que tendremos que hacer el programa después, y, lo que es más importante, separar la etapa de pensar en la solución de la de codificarla.

4 Implementarlo en un lenguaje de programación

Ya sabéis que existen muchos lenguajes de programación. Así que, finalmente, para poder ejecutar el programa deberemos traducirlo a un lenguaje de programación en concreto. Habremos elegido uno u otro dependiendo de varios motivos. Algunos lenguajes son más adecuados que otros para según qué tareas. Muchas veces, sin embargo, el lenguaje ya nos vendrá determinado. En nuestro caso, trabajaremos con el lenguaje **JavaScript**, que es un lenguaje que se utiliza para dar interactividad a las páginas web, por ejemplo.

Así que tendremos que conocer muy bien la sintaxis y la semántica del lenguaje para hacer nuestros códigos **JS**.

Ejemplo

Este sería un programa JS que soluciona el problema de la suma:

```
1  var suma;
2  suma = 1+2+3+4+5;
3  console.log(suma);
```

Se parece al algoritmo, ¿verdad?

Y este sería el mismo algoritmo en lenguaje C:

```
#include <stdio.h>

int main()

{
 int suma;

 suma = 1+2+3+4+5;
 printf(suma);
}
```

Son similares, pero tienen una sintaxis algo diferente. Además, cada lenguaje tiene sus propias características para poder hacer los códigos más eficientes en términos de uso de memoria y tiempo de ejecución. Ahora esto no es relevante porque los programas que escribiremos son muy sencillos.

Pero todos los lenguajes comparten, eso sí, los mismos principios algorítmicos.

Por eso que es una buena práctica pensar la solución independientemente del lenguaje. Así separamos el problema en dos partes: primero, **pensar en la solución** y, después, preocuparnos por las **cuestiones técnicas del lenguaje** en concreto.

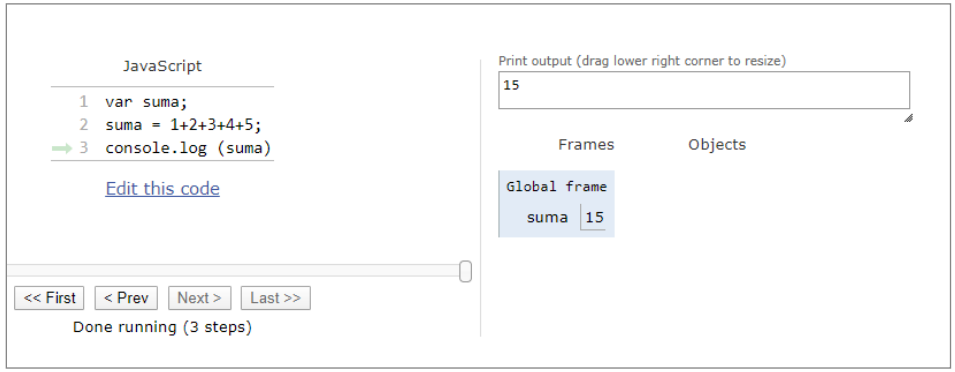
Es parecido a lo que sucede cuando escribimos un texto: primero pensamos qué queremos decir, en qué apartados lo distribuiremos y hacemos un borrador poniendo las ideas en el texto. Después, hacemos una relectura para fijarnos en la ortografía, la gramática, etc. A medida en que ganamos práctica como escritores, la segunda vuelta se va reduciendo porque ya directamente en la primera pasada sabemos escribir sin faltas y sin tener que pensar en las normas de ortografía. Con la programación sucede algo parecido: a medida que ganamos experiencia en un determinado lenguaje, ya no nos cuesta escribir directamente el código en el lenguaje sin que esto interfiera en su diseño.

5 Probar el código

Finalmente, cuando el programa funciona, hay que comprobar que la solución resuelve el problema. Esto es **probar el programa**. Este puede ser un paso complicado, especialmente en programas grandes y sofisticados. Por eso existen distintas técnicas de prueba.

En el caso de nuestro código JS será práctico ver qué sucede si lo ejecutamos desde el PythonTutor.

Lo podéis probar en la web de PythonTutor:



En este caso es sencillo porque solo hay una solución posible, así que, con una vez que lo ejecutemos y veamos que el resultado es 15, ¡ya lo tenemos!

Normalmente, sin embargo, hay que pensar en distintos "juegos de pruebas" para validar que un programa es correcto, y este paso es más complejo. Pero es importantísimo, eso sí, para validar nuestra solución. En términos profesionales, las pruebas son imprescindibles y representan un paso delicado que permite decidir cuándo una aplicación (o una nueva versión de una ya existente) se distribuye a los usuarios.

Qué pasaría si...

....la tentación (y la mala praxis) nos hacen ir al paso 4: a **codificar directamente**. ¿Y si, a base de prueba y error, vamos puliendo el código hasta que funcione más o menos? **Esto, además de claramente ineficiente es poco seguro**.

¿Y si resulta que no hemos entendido bien lo que se nos pedía porque no hemos pensado lo suficiente en el encargo o el enunciado (**paso 1**)? Habremos trabajado en balde porque, cuando le mostremos el resultado al usuario, nos dirá que no era eso lo que necesitaba.

¿Y si nos hemos puesto a escribir instrucciones sin pensar demasiado cómo hacerlo (**paso 2** y **paso 3**) y terminamos con un lío de programa que no hay quien lo entienda? Tal vez funciona, sin embargo, ¿alguien podrá modificarlo más adelante? ¿Alguien más lo entenderá? ¿Tiene una estructura clara? Si los programadores trabajan en equipo, ¿no es mejor hacer programas claros e inteligibles? Y, ¿qué sucedería si el profesor que os tiene que corregir no entiende lo que habéis hecho? Ya puede funcionar, que la cosa no terminará bien.

Y si desconocemos la sintaxis y las características del lenguaje (**paso 4**), costará que funcione y gastaremos muchas horas depurando errores y más errores. Seremos poco eficientes y se nos hará pesado. Antes de codificar la solución, revisamos la sintaxis, preguntamos, buscamos en manuales. Cuando escribimos un texto y tenemos una duda ortográfica, ¿verdad que consultamos el diccionario?

¿Y si finalmente no probamos a fondo el programa? (**paso 5**). A saber si hace lo que se esperaba, tal vez sí en algunos casos, pero no en todos. Antes de poner en circulación un nuevo código, estará bien que lo probemos a fondo. A este paso a menudo no se le dedica el tiempo necesario, desgraciadamente.

Y si, de paso, lo comentamos bien (incluyendo explicaciones y aclaraciones en el propio código), mejor que mejor. Será fácil de entender y, si es necesario, podremos modificarlo después. Esta es una muy buena práctica que a menudo se obvia por falta de tiempo, por las prisas, aunque está ampliamente documentado y es sobradamente conocido el alto coste que tiene más tarde cuando, por ejemplo, ya nadie es capaz de entender ese código y debe reescribirse de nuevo.

Cuanto más sistemáticamente lo hagamos, menos cosas dejaremos al azar y lograremos buenos programas, y seremos, en definitiva, buenos programadores.

Empezando a programar

Concepto clave

Un **algoritmo** es una secuencia de pasos para realizar una tarea. Un p**rograma** es la codificación de un algoritmo expresado en un lenguaje de programación concreto que un ordenador puede interpretar.

La programación estructurada se fundamenta en la **algorítmica**, que presenta los elementos fundamentales de la programación para cualquier lenguaje y que son: los tipos de datos para representar la realidad y los tres principios de combinación (secuencia, selección y repetición). Junto con las funciones para agrupar código y los esquemas para aprovechar ideas que ya funcionan, y ganar eficiencia. Puede parecer sorprendente, incluso mágico, pero combinando estos elementos se puede solucionar cualquier problema que se pueda resolver mediante un procedimiento (siguiendo unas instrucciones). Lo que se denomina, en términos informáticos, un problema computable. Dominar la algorítmica abre las puertas al poderoso mundo de la computación.

Hay otros paradigmas de programación: la programación funcional o la programación lógica por ejemplo, pero no se utilizan de manera habitual.

El **pensamiento computacional** aglutina una serie de técnicas que facilitan la programación: ayudan a dividir un problema complejo en problemas más sencillos y tratables (**descomposición**), que se pueden ir resolviendo uno a uno individualmente, teniendo en cuenta cómo se han resuelto problemas similares (**reconocimiento de patrones**) y prestando atención únicamente a los detalles relevantes del problema (**abstracción**). A continuación, permite definir el plan de acción, un conjunto de pasos a seguir, para resolver cada uno de los problemas (**algorítmica**) para terminar codificando estos pasos en un lenguaje de programación como instrucciones de un programa que el ordenador entenderá para resolver el problema.

Hay muchos tipos de **lenguajes de programación**, entre los textuales más comunes en el ámbito profesional, se encuentran el C, el Java y el Python. También el JavaScript (JS), que es muy adecuado para programar la interacción en las páginas web. Cada lenguaje tiene una sintaxis específica y diferente (como sucede con los idiomas que hablamos), y es más adecuado que otros para unos usos determinados, así que se elige uno u otro según lo que haya que programar. Pero todos comparten los elementos fundamentales de la algorítmica y del pensamiento computacional: la representación de los datos, las instrucciones elementales y los posibles modos de combinarlos (en secuencia, repitiendo o seleccionando).

Que un programa “funcione”, es decir, que haga lo que se solicita (esto es, que resuelva el problema o haga la tarea descrita en el enunciado o en el encargo) no es suficiente para ser considerado un buen programa; es necesario, además, que sea eficiente (que utilice los mínimos recursos de tiempo y memoria) y claro (que se entienda lo que hace y se pueda modificar fácilmente).

Un buen programador tiene destreza en pensamiento computacional y también conoce a fondo la sintaxis y la semántica del lenguaje en el que programa y pone en juego un conjunto de buenas prácticas: empieza por tener claro cuál es el encargo (la funcionalidad que debe tener el programa), piensa cuál es la mejor estrategia para lograrlo, define claramente los pasos o las instrucciones para conseguirlo, lo codifica sacando el máximo provecho al lenguaje de programación en el que está programando, y comprueba que el funcionamiento final responda a lo que se solicitaba con un buen juego de pruebas. Este modo de actuar se sintetiza en los siguientes pasos:

1. Entender el problema.
2. Pensar cómo resolverlo.
3. Hacer un borrador de la solución (algoritmo).
4. Implementarlo en un lenguaje de programación.
5. Probar el código.