Iniciación a la Programación

Juan Antonio
Martínez-Castroverde Pérez

Iniciación a la programación

Juan Antonio Martínez-Castroverde Pérez

Índice

Introducción

El objetivo de este trabajo es mostrar, de forma muy rápida y sencilla, cómo trabajan los ordenadores y cómo se programa. Permite entender por qué se quedan colgados los ordenadores, o saber qué es depurar un programa.

Se comenzará dando "unas pinceladas" sobre algunos conceptos básicos relacionados con la programación y se terminará dando tan solo 6 ejemplos de programas (de los cuales, además, 2 son muy sencillos, 2 son sencillos y los otros 2 son relativamente fáciles de entender).

Tras este pequeño esfuerzo, tendremos una perspectiva bastante buena y completa del mundo de la programación, y veremos los ordenadores, y los programas, con otros ojos.

Tener las nociones básicas que aquí se ilustran debiera formar parte de la cultura general de cualquier usuario de los ordenadores, tanto por su interés en el mundo en el que vivimos, como por el bagaje cultural que dan, como por la sencillez de estos conocimientos.

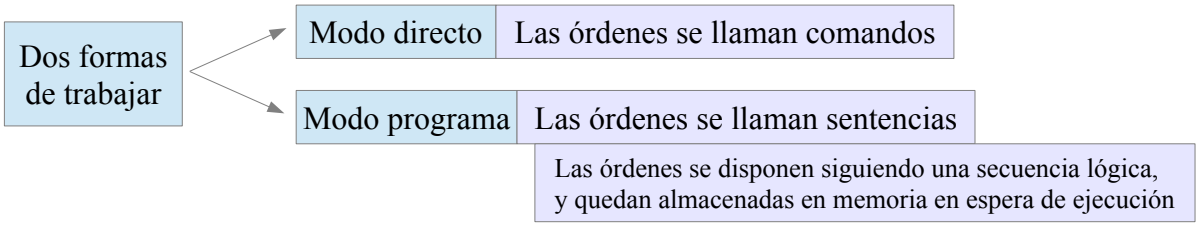
Lenguajes

Una forma rápida y práctica para introducirse en el mundo de los lenguajes de programación es considerar ejemplos de lenguajes de programación... y de otras cosas que no son lenguajes de programación, pero que muchas veces se confunden con ellos.

Como ejemplos de lenguajes de programación podemos citar el Fortan, Cobol, C, Pascal, C++, Python, Java, JavaScript, Ensamblador, Código Máquina...
Por el contrario el HTML no es propiamente un lenguaje de programación, así como tampoco lo es Windows (o su predecesor, el MS-DOS).

Sistemas operativos Vs lenguajes de programación

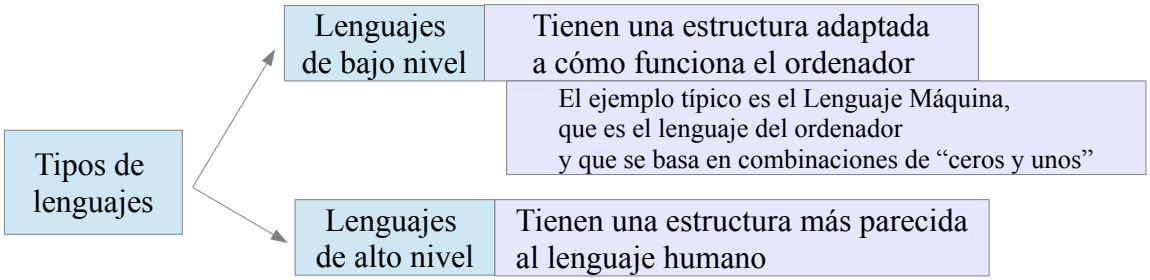
A la hora de trabajar con los ordenadores, podemos hacerlo de dos formas distintas:



Ejemplo: cuando en los actuales sistemas operativos (de interfaz gráfica), hacemos doble clic en el icono de una carpeta, estamos indicando al ordenador que nos muestre el contendido de la misma. En el sistema precursor de Windows, el MS-DOS (de interfaz de comandos), el equivalente es teclear el comando DIR, y el ordenador lista el contendido de un directorio (ahora llamado carpeta). En ambos casos estamos dándole órdenes al ordenador, y acto seguido las cumple (Modo directo).

¿Cómo podemos clasificar los lenguajes de programación? Lenguajes de alto nivel Vs lenguajes de bajo nivel

Los lenguajes de programación se pueden clasificar en atención a distintos criterios, pero la clasificación más básica es esta:



Como los ordenadores funcionan de forma muy diferente a como pensamos los humanos, resulta que trabajar con lenguajes de bajo nivel es más complicado. Por motivos que quedarán claros más adelante, además, trabajar con lenguajes de bajo nivel es mucho más laborioso.

Como contrapartida, si se trabaja con lenguajes de bajo nivel, podemos hacer con el ordenador todo lo que éste es capaz de hacer, cosa que no siempre es posible con los lenguajes de alto nivel.

Ejemplo: una orden típica en informática es CLS, que proviene de la expresión CLEAR SCREEN y cuando se le da al ordenador, éste borra la pantalla. Pero resulta que el ordenador no sabe lo que es la pantalla, ni sabe lo que es borrar... En realidad, cuando se da la orden CLS, lo que en el fondo se hace es llamar o ejecutar un programa, escrito en lenguaje de bajo nivel, que “pone a cero” los bits de una zona de la memoria del ordenador que es la que se utiliza para “construir” la imagen que vemos en el monitor.

Interpretes y compiladores

Lo que se quiere ilustrar con el último ejemplo visto, es que el ordenador siempre trabaja, en última instancia, en lenguaje de bajo nivel: CLS se utiliza tanto como comando, por ejemplo en el sistema operativo MS-DOS, como sentencia en múltiples lenguajes de programación de alto nivel. Cuando el usuario da la orden CLS, de una u otra forma, ésta se traduce a un programa en lenguaje de bajo nivel.

Un interprete es un programa que hace una “traducción simultánea” de una orden que el ordenador no entiende al lenguaje del ordenador. Esta traducción se hace tan pronto como se puede, una vez que se le da al ordenador esa orden. Se interpretan “o traducen” tanto las órdenes que se dan al ordenador desde un sistema operativo, como las órdenes que se le dan al ordenador durante la ejecución de algunos programas escritos en determinados lenguajes de alto nivel, como por ejemplo el lenguaje Java.

Un compilador es un programa que hace una “traducción previa” de la totalidad de un programa escrito en determinados lenguajes de alto nivel. Esta forma de trabajar tiene ventajas, ya que la “traducción simultánea” ralentiza la ejecución de un programa y, por tanto, merma el rendimiento de nuestra computadora. La “traducción previa” permite que el programador utilice un lenguaje de alto nivel, que le resulta más cómodo, sin el inconveniente mencionado de la “traducción simultánea”: una vez terminado el programa en lenguaje de alto nivel, un compilador lo “traduce” (compila) al lenguaje del ordenador, de forma que queda guardado en Lenguaje Máquina y cuando se tiene que ejecutar, el ordenador lo hace rápidamente porque no pierde tiempo en su “traducción”.

Nota: un lenguaje de bajo nivel también puede compilarse para traducirse al Lenguaje Máquina. Este es el caso del lenguaje Ensamblador. Ambos lenguajes son casi idénticos, y ambos están adaptados a como trabaja el ordenador. La diferencia es que en el Lenguaje Máquina todo son combinaciones de ceros y unos (tanto las órdenes, como las variables o los números...) por lo que a un humano le resulta muy difícil de manejar, mientras que en el Ensamblador se sustituye cada grupo de ceros y unos por una palabra, letra o número, de forma que a las personas nos quede más claro su significado. Hay, por tanto, una relación biunívoca entre ambos y podemos decir que, en el fondo, es lo mismo, pero el Ensamblador utiliza la capacidad nemotécnica humana para facilitar su uso.

Ventajas de saber programar

El software cada vez está más presente en nuestras vidas. Tener nociones básicas de programación se considera cada vez más importante. Las previsiones son que cada vez más empleos demanden trabajadores con conocimientos de programación, aunque el trabajo no sea específicamente para programar.

Programar ayuda a desarrollar el pensamiento creativo y también el sistematico, la lógica, la abstracción y la habilidad para resolver problemas. Programar fomenta el espíritu emprendedor.

Por todas estas razones, cada vez está cobrando más peso en la enseñanza el aprendizaje de la programación.

Mitch Resnick, conocido investigador del MIT MediaLab, se muestra escéptico ante esta generación de nativos digitales. Según Resnick, el problema de los nativos digitales no es la interacción con las nuevas tecnologías, sino su capacidad para crearlas.

No se trata de crear una generación de programadores, señala Resnick. Más bien, se trata de considerar la programación como una puerta a un mayor aprendizaje. "Cuando aprendemos a leer, podemos leer para aprender. Pasa lo mismo con la programación: si aprendes a escribir código, puedes codificar para aprender". Aprender a programar significa aprender a pensar creativamente, razonar sistemáticamente y trabajar en equipo. Y estas habilidades son aplicables a cualquier profesión, además de ser muy necesarias en nuestra vida personal.

Resnick y su equipo del MIT han desarrollado Scratch, un programa que permite crear y compartir fácilmente animaciones y juegos interactivos.

Extraído de la noticia: http://www.madrimasd.org/informacionidi/noticias/noticia.asp?id=55706&origen=notiweb&dia_suplemento=jueves

Pasos para hacer un programa

Hacer un programa es un caso particular de algo mucho más genérico, que es resolver un problema, y como en todo problema, buena parte del camino hacia su resolución es saber enunciar ese problema en sus justos términos, lo cual solo es posible cuando se conoce con profundidad el problema al que nos enfrentamos.

Ejemplo:
¿cómo debe ser un programa que enseñe al ordenador a reconocer letras, caras u otros objetos?

Aquí tenemos un ejemplo típico que muestra la diferencia entre nosotros y las computadoras: hay cosas que estas máquinas hacen mucho mejor que nosotros, como calcular con rapidez y memorizar rápidamente cantidades enormes de datos, sin embargo, nosotros hacemos con suma naturalidad cosas que a los ordenadores les resultan muy difíciles, sino imposibles. En este caso estamos hablando del problema de la percepción.

¿Cómo le podemos enseñar a una máquina para que sepa percibir cosas? Una primera forma de abordar un problema es tratar de inspirarse en la naturaleza. El problema que tenemos con la percepción es que todavía apenas estamos empezando a entender cómo trabaja el cerebro para percibir. Así que si queremos enseñar a un ordenador a percibir, tenemos que abordar el problema de otra forma. La cuestión es ¿cómo?

Como este es un problema muy complejo, lo que se ha hecho es enfrentarse a problemas concretos de percepción: reconocimiento óptico de caracteres, reconocimiento facial...

Para ilustrar una posible forma de abordar la identificación de rostros mediante un programa de ordenador, y dejar claro, al mismo tiempo, la forma tan distinta a la nuestra en la que los ordenadores trabajan, comentamos uno de estos posibles procedimientos:

Se hace un programa que le enseñe al ordenador a identificar algunos puntos característicos de la cara, como pueden ser los ojos, la punta de la nariz, los pómulos, la barbilla (todo esto, por cierto, ya es bastante complicado de por sí). Luego, otro programa une estos punto mediante segmentos y calcula ángulos entre segmentos que se cortan y proporciones entre distintos segmentos. A continuación se calcula la probabilidad de que los valores obtenidos correspondan con los valores asociados de algunas de la caras que tiene el ordenador en su base de datos.

Como se ve, nosotros no trabajamos de esta forma, y para nosotros sería muy pesado hacer todo este procedimiento, pero, como sabemos, los ordenadores calculan muy rápidamente, por lo que para ellos esta forma de trabajar no es ningún problema.

Una vez que se conoce en profundidad el problema que deseamos resolver y, como consecuencia, podemos enunciar el problema de forma precisa, debemos buscar un camino apropiado para resolverlo.

Cuando el problema que queremos resolver está ligado a la elaboración de un programa de ordenador, los pasos anteriores necesitan tener presente lo que sabe hacer un ordenador.

¿Qué es capaz de hacer un ordenador?

Para el que todavía no lo conoce, lo más sorprendente es que un ordenador apenas sabe hacer casi nada. Tan solo sabe hacer "cuatro cosas sencillas". Y todavía sorprende más cuando nos paramos a pensar que, a partir de esas "cuatro tonterías" que es capaz de hacer el ordenador, mediante un buen programa, somos capaces de conseguir que una computadora gane al mejor ajedrecista del mundo, sepa reconocer el habla, pilotar un avión o hacer diagnósticos médicos con más acierto que los especialistas en ese campo...

Un ordenador sabe:

- ✔ mover información
- ✔ guardar información
- ✔ sumar números
- ✔ comparar números

Es interesante resaltar que un ordenador ¡ni siquiera sabe multiplicar! Cuando un ordenador hace multiplicaciones, lo que realmente está haciendo es ejecutar un programa que le enseña a multiplicar.

Dentro de las capacidades del ordenador, como contrapartida a lo poco que sabe hacer, debemos señalar que lo que hace lo hace a una velocidad increíble. Las supercomputadoras ya han superado la frecuencia del petaflops (¡10^{15} operaciones lógicas por segundo!)

Pero pese a su velocidad, lo que realmente sabe hacer un ordenador es muy limitado, así que es lógico preguntarse: ¿cómo se consigue que un ordenador haga todo lo que hace?

Eso es precisamente lo que se quiere ilustrar en este trabajo: cómo utilizando el ingenio, y mediante un programa de ordenador, somos capaces de conseguir que un ordenador haga prácticamente cualquier cosa.

Vemos, pues, que la clave está en los programas... junto a la elevada velocidad de la computadora.

¿Qué es un programa?

Una buena aproximación, para aclarar qué es un programa, es resaltar que "todo" son programas:

- Un sistema operativo es un programa que nos ayuda a gestionar la computadora.
- Un procesador de texto es un programa.

- Un virus es un programa, igual que el antivirus.
- Un juego es un programa.
- También hay programas dentro de los programas: la herramienta buscar de un procesador de texto (o bien de un sistema operativo) también es un programa, al igual que el corrector ortográfico, que es otro programa y esto tan solo por poner unos ejemplos.

Esta idea era la que se recogía en la película de *Matrix*, donde se comenta que todo lo que aparece en Matrix son programas: los árboles, las hojas, los pájaros... y el agente Smith.

Entrando más a fondo, un programa es un algoritmo.

Un algoritmo es algo a lo que estamos muy acostumbrados, aunque a veces no se es consciente de esto, o no se conoce con este nombre.

Un algoritmo es un conjunto de operaciones ordenadas en una secuencia lógica.

Ejemplos de algoritmos:

- ✔ Cualquier receta de cocina (cualquier receta consiste en una serie de pasos u operaciones que se deben hacer en un orden determinado: no se puede freír un huevo si antes no se echa el aceite en la sartén)
- ✔ Decir si un número es par o no.
- ✔ Hacer rufini, o hacer una división (los alumnos, en el colegio, suelen aprender muchos algoritmos)

Para expresar algoritmos utilizaremos los organigramas.

Organigramas

Un organigrama es una forma de representar gráficamente los algoritmos.

En lo que sigue, pondremos ejemplos de programas de ordenador, y por tanto de algoritmos, utilizando organigramas. Una de las ventajas que tiene el utilizar organigramas es que podremos trabajar sin necesidad de hacer referencia a ningún lenguaje de programación concreto, y de esta forma podremos conseguir fácilmente nuestro objetivo, que es, como ya se dijo, dar una visión general de cómo se programan los ordenadores.

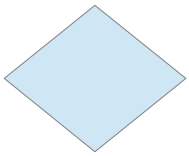
En nuestros organigramas, vamos a utilizar los siguientes elementos básicos:

Rectángulo. Representa una operación o cálculo.

Sirve tanto para indicar una operación matemática, como para asignar un valor a una variable.

Segmento. Indica el camino que sigue el ordenador y, por tanto, el orden en el que ejecuta los distintos pasos del algoritmo.

Por defecto el orden será hacia abajo. En ese caso el segmento es suficiente. En caso contrario de indicará el sentido en el que se recorre el segmento con flechas.

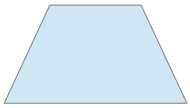


Rombo. Representa la toma de decisiones.

El camino que sigue el ordenador (indicado por los segmentos) solo puede tener bifurcaciones en este elemento.

Es interesante observar que su forma geométrica es apropiada ya que los segmentos llegan o salen desde los vértices. En un vértice estaría el segmento o camino de llegada y se utilizarían dos o tres vértices para los diversos caminos de salida.

Como hemos comentado, una de las pocas cosas que sabe hacer el ordenador es comparar números (o, en general, el valor de variables) y fruto de esta comparación los números serán iguales o distintos (en ese caso habrá dos salidas del rombo). El caso más complicado es cuando el resultado de la comparación es del tipo igual, menor o mayor (en este caso habrá tres salidas del rombo)



Trapecio. Representa la entrada y salida de datos al ordenador o desde el ordenador. El trapecio normal se utiliza para representar la entrada de datos y el invertido () para la salida.

En la práctica, para introducir datos en el ordenador se puede utilizar el teclado, un cable USB, el ratón...

Para la salida de datos sirve la pantalla del ordenador, la impresora, una conexión inalámbrica...

Puede sorprender que con estos cuatro elementos gráficos podamos expresar cualquier programa, pero hemos de recordar que ya se dijo que los ordenadores saben hacer “cuatro cosas”. La relación que hay entre lo que sabe hacer un ordenador y los elementos gráficos que utilizaremos es, sin ánimo de ser rigurosos, la siguiente:

- ✔ mover información: se producen en la entrada y salida de datos, y está vinculada también, de alguna forma, a los distintos elementos comentados de los organigramas.
- ✔ guardar información: se produce en el trapecio, ya que los datos que entran se guardan en alguna de los diferentes tipos de memoria que tiene el ordenador. También se guarda información en el rectángulo, cuando se asigna un valor a una variable.

- ✔ sumar números. Se produce en el rectángulo, cuando la operación considerada es una suma.
- ✔ comparar números. Se produce en el rombo.

Ejemplos de algoritmos

Es oportuno señalar que los ejemplos que vamos a ver a continuación son sencillos, pero eso no quita que haya quien encuentre todo esto de cierta complejidad. Hay que tener presente que, debido a la novedad, puede resultar esto un poco extraño, pero no debemos confundir lo nuevo con dificultad, porque no lo es.

Por otro lado, es mucho más complicado elaborar un algoritmo que entenderlo, y aquí se pide tan solo entender algoritmos.

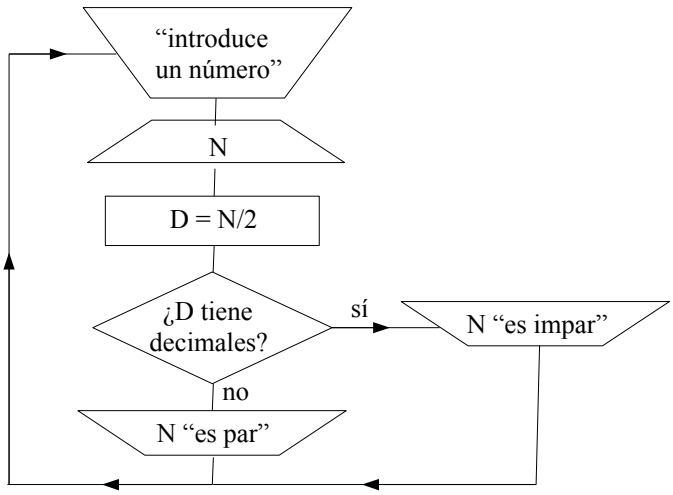
Sin embargo programar, una vez que se tiene práctica, es más fácil de lo que puede parecer, ya que los distintos procedimientos que vamos a ver acaban por ser de uso común en informática, y programar acaba siendo muchas veces repetitivo. Lo verdaderamente difícil suele ser desarrollar programas que permitan al ordenador emular algunas facetas de la "inteligencia", empezando por la percepción y la elaboración de conceptos.

Ejemplo 1: Par o impar

¿Cómo sabemos que 456 es par? Los humanos, en principio, nos fijaremos en el último dígito y si este es 0, 2, 4, 6 u 8 sabemos que el número es par.

Como hemos visto, los ordenadores saben hacer muy pocas cosas, y determinar cuál es el último dígito de un número no es algo tan obvio para una computadora, sobre todo teniendo presente que trabajan en el sistema binario...

Para un ordenador resulta mucho más sencillo todo lo relacionado con cálculos matemáticos, por eso nos podemos preguntar ¿hay algún procedimiento matemático que nos permita determinar si un número es par? La respuesta es fácil: dividiendo entre 2. Si la división tiene decimales, el número es impar, si no los tiene, es par.



Comentarios
Como hemos contemplado la división dentro del organigrama, eso significa que hemos elegido un lenguaje de alto nivel.

Podemos utilizar cualquier letra para la variable, pero lo aconsejable es utilizar una letra que nos recuerde su significado: N para número y D para división.

La finalidad de los dos primeros algoritmos es familiarizarnos tanto con la notación, como con algunos procedimientos básicos.

Lo entrecomillado aparecerá literalmente en pantalla. Cuando se da la orden de poner en pantalla N (sin comillas) no aparecerá N, sino su valor numérico.

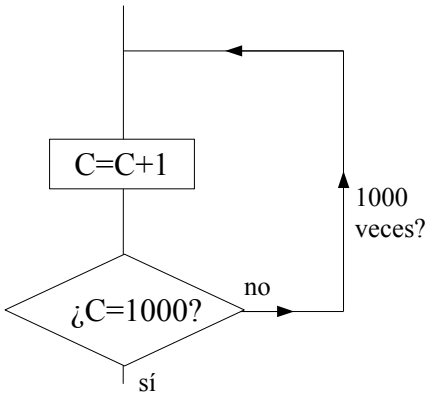
Ejemplo 2: Bucle de retardo

La rapidez que tienen los ordenadores, a veces, puede ser excesiva. Hay ocasiones en las que le tenemos que decir a la computadora que vaya más despacio. Esto se hace mediante los bucles de retardo.

Se llaman bucles porque se vuelve hacia atrás.

A la hora de construir un algoritmo empezaremos escribiendo la parte básica, y luego iremos corrigiendo errores y ajustando los detalles, para que funcione bien. Se tratará de reproducir el proceso habitual de construcción...

Borrador 1



Comentarios

Un ordenador cuenta tan rápido que tenemos que contar hasta valores altos...

Utilizamos la letra C por cumplir la variable la función de un contador.

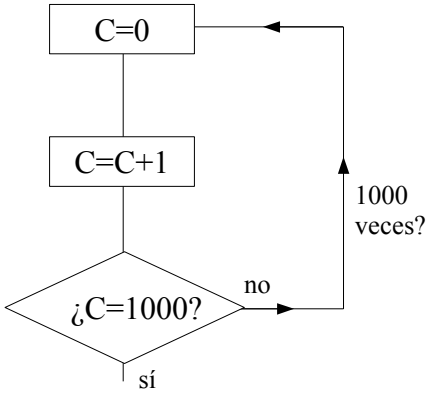
Matemáticamente, la ecuación C=C+1 no tiene sentido. Es la notación usada habitualmente en informática y debe interpretarse como C'=C+1 (el nuevo valor de C es el antiguo más uno)

Si se ejecutara el algoritmo del borrador 1 aparecería uno de los errores más frecuentes de la informática:
Variable NO definida.
La primera vez que el ordenador llega a la instrucción C'=C+1 tiene que utilizar un valor de C que NO conoce.

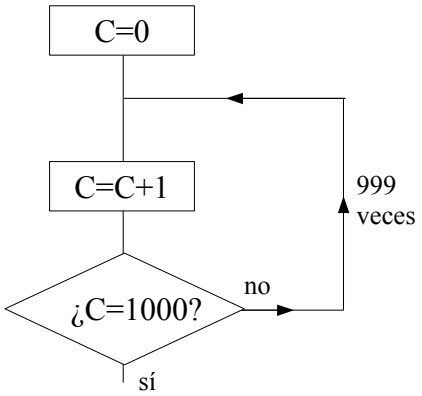
Este error se soluciona tal y como aparece en el Borrador 2

¿Qué ocurriría si un ordenador ejecuta el algoritmo del borrador 2?
Se quedaría colgado: entraría en un proceso sin fin donde, a pesar de estar continuamente haciendo operaciones, no iría a ninguna parte. Esto es lo que ocurre cuando un ordenador se queda colgado, aunque normalmente la causa de quedarse colgado no tiene un fallo tan evidente como este. Más adelante veremos motivos de cuelgue más frecuentes.

Borrador 2



solución:



La solución presenta lo que hay que hacer para evitar que se quede colgado.

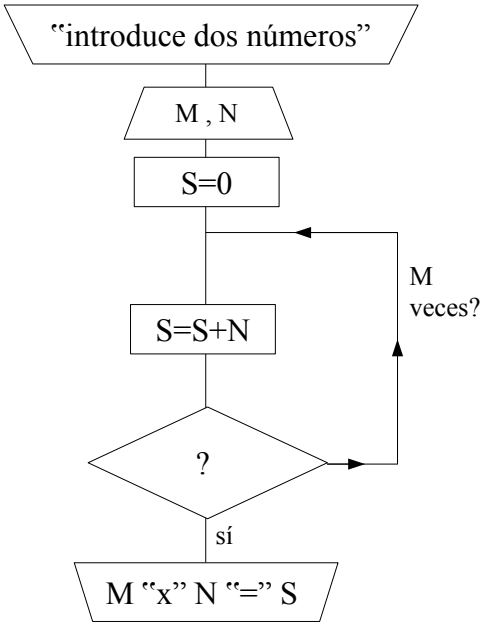
Es interesante observar que, debido al bucle, volvería hacia atrás 999 veces.

Ejemplo 3: Multiplicar dos números naturales

La idea para desarrollar el algoritmo es la siguiente: $2 \, x \, 3 = 2+2+2 = 3+3$

o, más en general: $M \times N = \underbrace{M+M+M+...+M}_{N \text{ veces}} = \underbrace{N+N+N+...+N}_{M \text{ veces}}$

Borrador

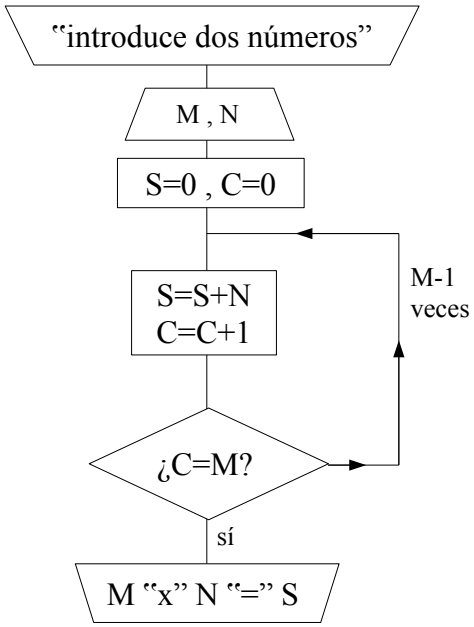


Comentarios

Para construir la solución, utilizamos la variable S. La solución se construye sumando M veces N.
De alguna forma tenemos que controlar cuándo tenemos que salir del bucle.

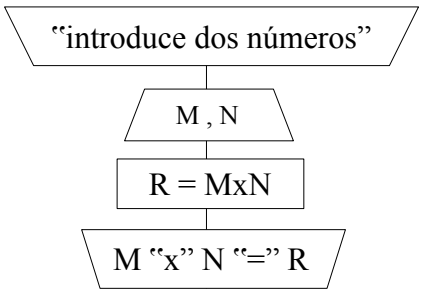
La condición NO puede ser que S=MxN, ya que eso es, precisamente, lo que estamos calculando y, por tanto, no conocemos el valor de la multiplicación.

solución:



La solución es aquí usar un contador, como el usado en el bucle de retardo (en la informática los procedimientos suelen ser muy usados en múltiples contextos). En esta ocasión el contador se utilizará para contar cuántas veces estamos sumando N.

El algoritmo correspondería a un lenguaje de bajo nivel. En un lenguaje de alto nivel (que contempla la multiplicación) el organigrama sería el siguiente:

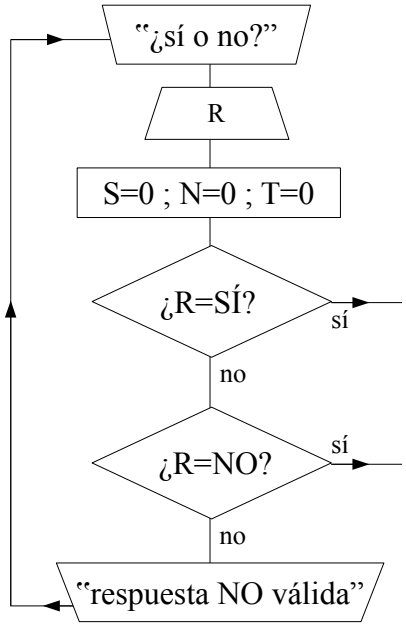


En este caso, cuando hubiera que ejecutar la operación R=MxN, ésta se traduciría en el código comprendido entre los dos trapecios en la solución de la izquierda.

Ejemplo 4: Hacer una encuesta (como las que aparecen en la web)

La idea es que el ordenador plantea una pregunta, el usuario contesta, y para cada respuesta se actualiza en la pantalla la estadística correspondiente a todas las preguntas contestadas.

Borrador



Comentarios

Utilizaremos la variable R para guardar la respuesta dada. R es una variable alfanumérica (a diferencia de las variables vistas hasta ahora, que son numéricas). Una variable alfanumérica puede tener valores tanto numéricos como alfabéticos.

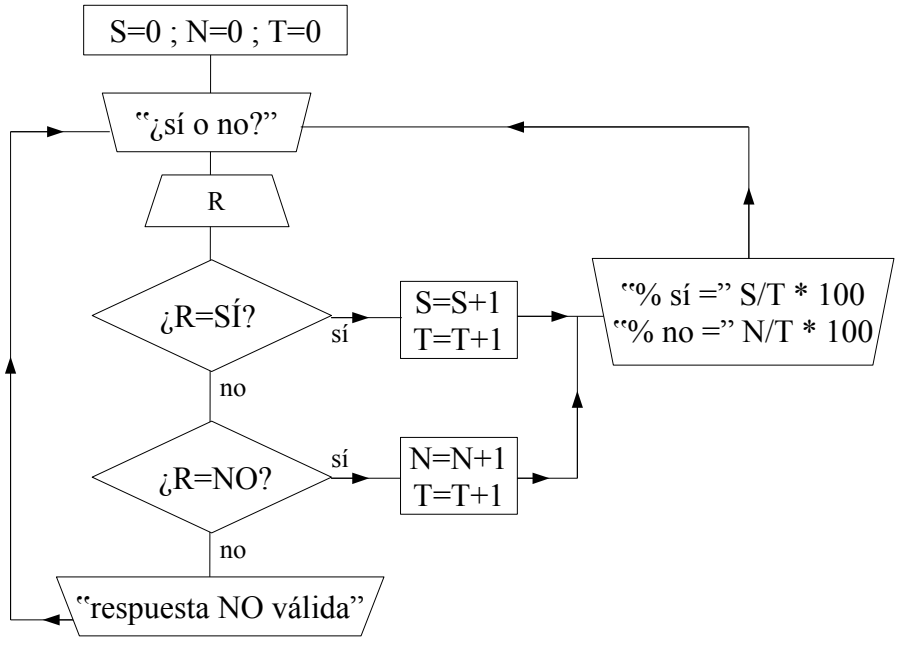
Utilizaremos, para contar las respuestas, las variables S (para los SÍ), N (para los NO) y T (para el total de respuestas).

¿Es la ubicación de la definición de los valores iniciales de S, N y T la adecuada?

El ordenador tiene que analizar cada una de las respuestas. Cuando detecte que la respuesta es SÍ tendrá que hacer algo. Hará algo análogo cuando detecte que la respuesta es NO.

Se puede diseñar el algoritmo para que se acepten otro tipo de respuestas o no...

solución



Comentarios

T no es necesaria: (T=S+N)

Los caminos se unen para ahorrar código. Igualmente se podría poner T=T+1 en el camino común.

Aquí se a elegido una forma simple de poner los resultados en pantalla. Si se hubiera querido presentar los resultados de forma gráfica se recurriría a un programa que construya esas gráficas a partir de los valores de las variables.

Ejemplo 5: Identificar números primos.

¿Qué es un número primo?

Da algunos ejemplo de números primos ¿Qué procedimiento has seguido para calcularlos?

La idea para ver si un número, N, es primo es hacer las divisiones: $\frac{N}{2}, \frac{N}{3}, \frac{N}{4}, \ldots, \frac{N}{N-1}$

si en este proceso llegamos a una división sin decimales, el número no es primo. Si llegamos a la última división, y todas tienen decimales, N es primo.

Es bastante fácil encontrar una fórmula matemática que genere todos los números pares o impares: si $n \in \mathbb{N}$; $2n$ es un número par y $2n+1$ es un número impar. Sin embargo, todavía no se ha descubierto una fórmula igual que genere todos los números primos (y quizá no exista). Lo que está claro es que, debido a la importancia de los números primos, si alguien encontrase esa fórmula se haría famoso y millonario.

Precisamente, debido a esa importancia de los números primos actualmente hay en el mundo miles de computadoras buscando números primos, ejecutando un algoritmo similar al que vamos a desarrollar aquí.

Lectura relacionada:

Extracto de la entrevista a Marcus Du Satoy, 20080122 Público, Autor: Vicente F. de Bobadilla [Marcus Du Satoy es un matemático catedrático de Oxford, su libro *La música de los números primos*, fue un best seller en todos los países donde se publicó.]

P. Y dentro de las matemáticas está su pasión por los números primos. ¿de dónde le viene?

R. Probablemente, porque son muy sencillos y, al mismo tiempo, constituyen el mayor misterio de las matemáticas. Los niños en el colegio aprenden que un número primo es aquel que sólo es divisible por uno o por sí mismo. Pero aún así, creo que lo fascinante de ellos es que llegan al núcleo de que es ser de verdad un matemático, que es la búsqueda de patrones, [...]. Tenemos estos números: dos, tres, cinco, siete, once, trece... y cada uno de ellos, como los asesinatos, entiendes por qué es un número primo, pero ¿cómo averiguas dónde va a estar el siguiente? ¿Dónde está el siguiente número primo, dónde va a producirse el siguiente asesinato?

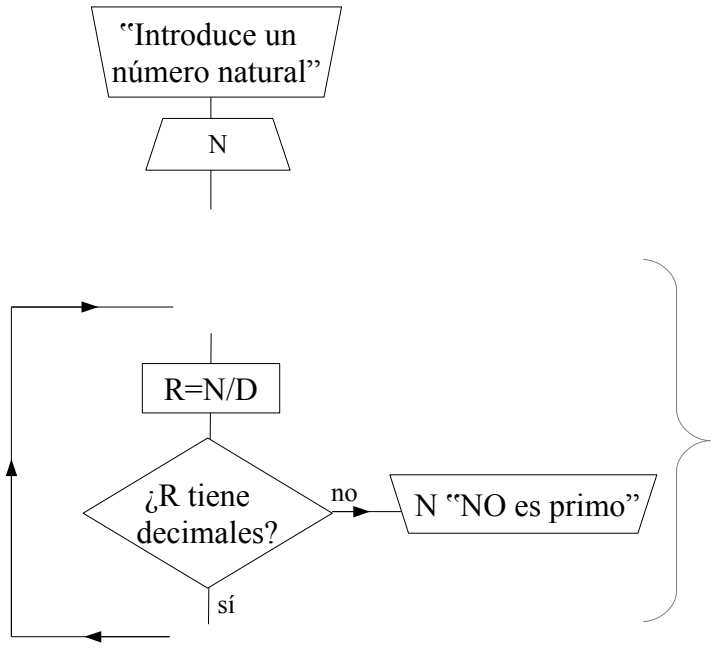
Para mí, eso es lo que resulta tan intrigante. Sabemos que hay infinitos números primos, el mayor que conocemos hasta ahora tiene 9,8 millones de dígitos... es un número enorme, pero no sabemos cuál va a ser el siguiente que aparezca. Y ese es uno de los campos más intrigantes de todas las matemáticas. Y estos números son fundamentales porque, como he dicho antes, son ladrillos, son el oxígeno y el hidrógeno de mi mundo. Hay muchos problemas matemáticos que se reducen a observar o comprender aspectos de los números primos. Y hay cosas que no comprendemos, están bloqueando nuestro progreso.

P. ¿El uso de ordenadores les ayuda a buscar el siguiente número primo?

R. Sí, desde luego. No podríamos haber encontrado esos números enormes sin la ayuda del ordenador. Pero no estamos hablando de grandes supercomputadoras, sino de un ordenador de mesa normal y corriente. Es un ejemplo excelente de un problema que se ataca con la ayuda de redes entrelazadas. De hecho, cualquier lector de su periódico puede apuntarse a este proyecto, e incluso podría ganar algo de dinero; la persona que

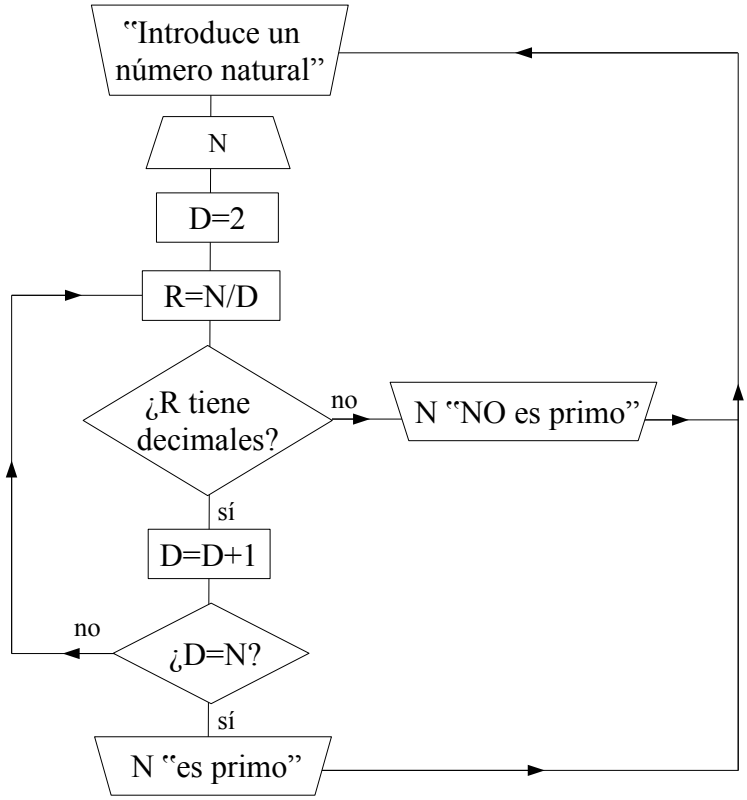
encuentre un número primo que rompa el récord de los diez millones de dígitos, se lleva un premio de 100.000 dólares, que es lo que ha ofrecido una organización americana. La website es www.mersenne.org. Pero creo que descubrir grandes números primos no es tan importante como averiguar se relacionan los números primos. Encontrar un número primo grande es como cantar en una nota muy alta; está bien, pero es necesario entender la música.

Borrador Comentarios



Tenemos que controlar el incremento de D y el fin del bucle, momento en el que podremos concluir que N es primo

Solución Comentarios



El valor inicial de D depende de dónde se ponga D=D+1. Igualmente, la condición para salir del bucle varía según la ubicación de D=D+1

Pegas:
¿Qué le ocurre al ordenador si N=1?
¿Y si N=2?
¿Y si N no es natural?
¿Es necesario dividir hasta N-1?

Comentamos todo esto a continuación.

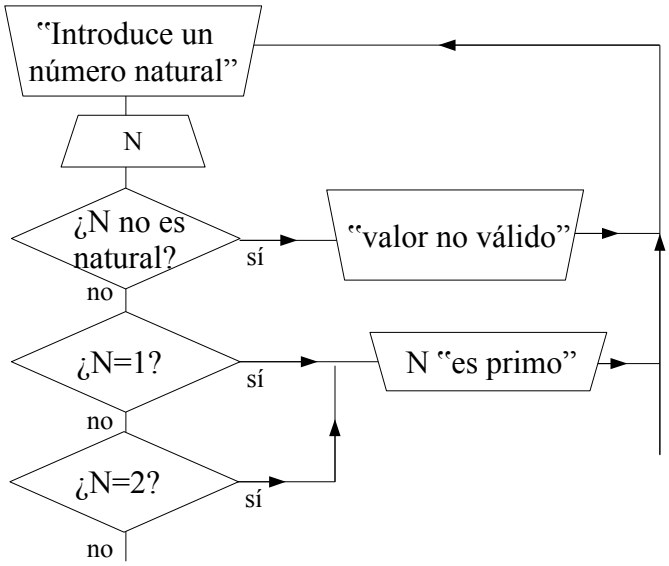
Depurando el algoritmo

Si N=1 el ordenador se queda colgado. Esta es una situación más típica que el primer motivo de cuelgue que hemos visto. Ocurre por que se diseña un algoritmo pensando en una situación genérica, y no caemos en la cuenta de que para unos valores concretos el algoritmo general, pese a estar bien planteado, falla.

Si N=2 el algoritmo ofrece un resultado erróneo.

Hemos de rechazar valores de N que no sean naturales.

Para solucionar todos estos problemas, una vez introducido el valor de N se analizará y se tomarán las siguientes decisiones en función del valores de N:



Esto se puede hacer, precisamente, porque los valores en los que el algoritmo falla son excepciones. Para los infinitos casos restantes en los que el algoritmo funciona se utilizará éste.

El concepto de depurar un programa contempla quitar todos aquellos fallos que tenga.

Aquí viene el algoritmo principal

Optimizando el algoritmo

Es evidente que no hace falta llegar hasta la última división: una vez que hemos llegado a la división N/(N/2), y todas las divisiones tienen decimales, ya no hace falta seguir, pues las que quedan también los tendrán. El número N es primo.

Igualmente podemos ahorrarnos divisiones utilizando propiedades matemáticas: si un número termina en 0 o 5 No es primo (ya que es divisible entre 5), si la suma de los dígitos de un número es múltiplo de 3 no es primo (ya que es divisible entre 3).

Igualmente podemos utilizar el siguiente resultado:

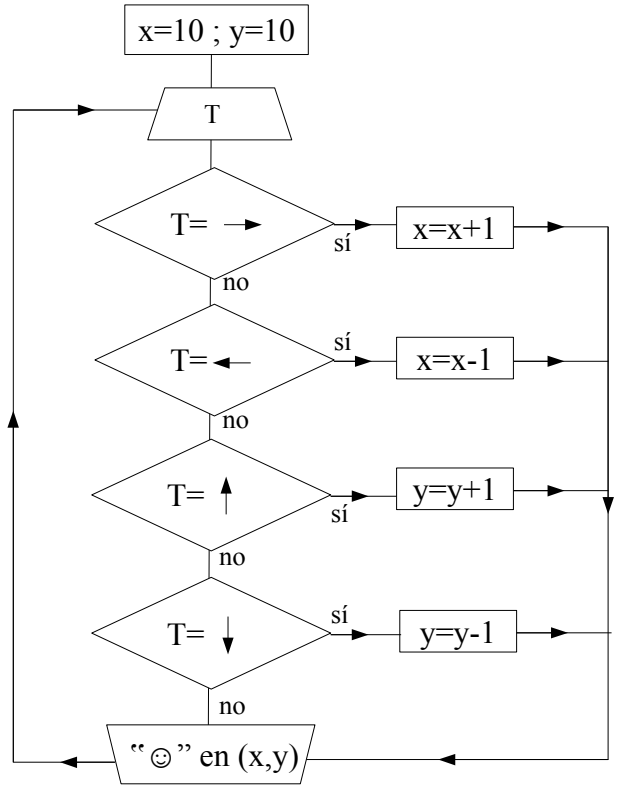
Si N NO es divisible entre 2 → NO es divisible entre 4, 6, 8, 10, 12...

El concepto de optimizar un programa contempla mejorarlo para que obtenga el mismo resultado en menos tiempo.

Ejemplo 6: Controlar el movimiento de un muñeco con el teclado

Queremos mover un muñeco por la pantalla utilizando los cursores.

Borrador



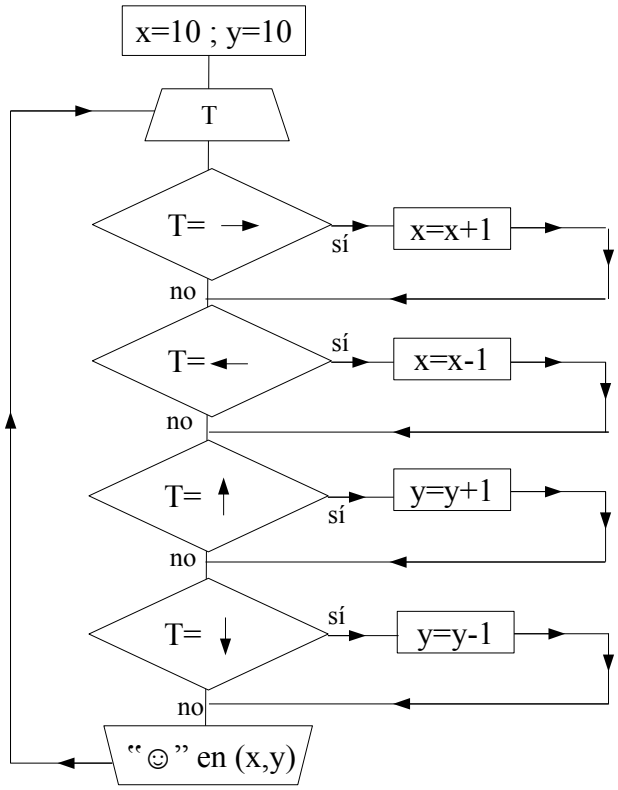
Comentarios

Tal y como se ha escrito el algoritmo, si se pulsa simultáneamente la tecla hacia la derecha y hacia arriba, el muñeco solo se irá hacia la derecha, cuando lo deseable es que fuera en diagonal.

Igualmente, si se pulsa simultáneamente hacia la derecha y hacia la izquierda, el muñeco se irá hacia la derecha, cuando lo lógico es que se quedara en el mismo sitio.

¿Cómo se puede hacer que el muñeco vaya en diagonal en el primer caso, y se quede quieto en el segundo?

Borrador 2

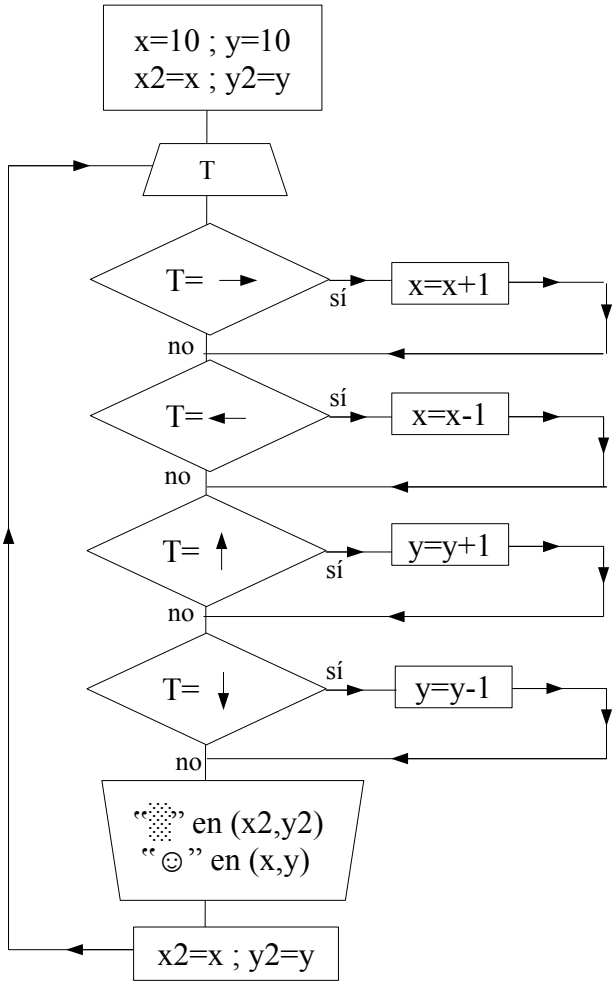


Comentarios

Ahora el movimiento del muñeco es el deseable. Pero queda un problema: el muñeco va dejando su traza conforme se mueve. Esto, a veces, es un efecto deseable, pero a nosotros, en nuestro algoritmo no nos interesa.

¿Cómo se puede hacer que no quede rastro de la traza del muñeco al moverse?

solución



Comentarios

(x2, y2) son las coordenadas de un cuadradito de la dimensiones de nuestro personaje y con el mismo color del fondo. Definir sus valores iniciales en función de los de x e y es lo más sensato, para evitar trabajo innecesario y errores futuros.

Si el fondo de la imagen no fuera uniforme, en vez de un cuadrado de ese color se copiaría el fondo antes de dibujar al muñeco sobre él.

¿Cómo hacemos para que el cuadradito vaya siguiendo al muñeco? Observar lo sencillo que resulta la forma de lograrlo. También es interesante observar la forma tan matemática como se trabaja.

Nota: aunque el muñeco no se mueva, no se espera que notemos el borrado y la nueva aparición del muñeco en el mismo sitio, por lo rápido que sucede. Es interesante observar que las luces de bajo consumo se apagan y se encienden 100 veces cada segundo.

Recursos

Noticias relacionadas con la programación:

Según estadísticas laborales del Gobierno de Estados Unidos, en 2020 se necesitarán 1,4 millones de ingenieros informáticos y en el país solo habrá 400.000 personas formadas para ello. [http://smoda.elpais.com/articulos/donde-estan-las-mujeres-programadoras/6134]

Si sabes programar tienes trabajo
http://www.expansion.com/multimedia/videos.html?media=TF6QIQQg7MC&cid=SIN8901
¿Cuánto puede llegar a ganar un desarrollador? ¿Qué conocimientos previos hay que tener para aprender a programar? ¿Qué tecnologías tienen mejores salidas profesionales? Éstas y otras preguntas las responde Gonzalo Manrique, cofundador de Ironhack [...]. ¿No hace falta ser ingeniero, ni tener un máster o un doctorado, para aprender a programar. De media se tarda unos dos años en adquirir la técnica, pero es factible hacerlo con un curso intensivo de dos meses, asegura.

"Existe una brecha entre las personas que programan muy bien y las que diseñan muy bien; muchas veces el buen programador es un mal diseñador y viceversa" [Rubén González Crespo, director de la Escuela de Ingeniería de UNIR e investigador de UNIR Research.
http://www.madrimasd.org/informacionidi/noticias/noticia.asp?id=62777&origen=notiweb&dia_suplemento=jueves]

Un diseñador web hoy en día debe saber programar. […] El perfil profesional que emerge con fuerza es híbrido: una mezcla de artista visual y tecnólogo, o como define certeramente el término anglosajón, arquitecto de la información. [extraído de la noticia *El diseño es el código*
http://elpais.com/diario/2008/07/03/ciberpais/1215050546_850215.html]

“La programación es un superpoder que te permite llegar donde quieras” [...] En un futuro cercano la programación será un diferenciador social tremendo en España, como sucede hoy con el inglés. [Extraído de la noticia El inglés del futuro será la programación
http://economia.elpais.com/economia/2015/12/17/actualidad/1450382673_653460.html]

[...]el futuro está en los ordenadores", alerta en un vídeo Bill Gates[...] "Aprender a programar no significa querer conocer todo de la ciencia de la computación o ser un maestro".
[...]Hay que usar el ordenador como herramienta para resolver problemas. Lo que los anglosajones llaman el computational thinking.
[...]La palabra 'programación' asusta, pero los monitores no se cansan de repetir que resulta tan fácil de controlar como la lectura o la escritura.
[...]"Cada vez más profesiones obligan a tener conocimientos de programación: la biogenética, las artes gráficas, el mundo empresarial… Así que la Informática se debería incluir en secundaria, en especial para los de ciencia y arte
[...]nadie pone en duda estas palabras del pensador chino Confucio (551-478 a. C.): "Me lo contaron y lo olvidé; lo vi y lo entendí; lo hice y lo aprendí".
[extraído de la noticia *Aprender a programar como se aprende a leer*
http://sociedad.elpais.com/sociedad/2013/03/07/actualidad/1362689630_904553.html]

webs y materiales en internet para programar

SCRATCH http://scratch.mit.edu/hoc2014/ Create stories, games, and animations. Share with others around the world. Programación en bloques de código (sistema de programación intuitivo, que imita a piezas de construción, creado por el MIT. Apto para ser usado por niños desde los 8 años e incluso desde los 6, pero en este caso se requiere la atención constante de un adulto.)

HORA DEL CÓDIGO http://hourofcode.com Tutoriales de una hora de duración disponibles en más de 30 idiomas. No se necesita experiencia. A partir de 4 años. Al igual que Scratch, utiliza bloques de código.

De esta web:
http://studio.code.org/ Cursos de 20 horas sobre fundamentos de la Ciencia Computacional (todas las edades)

Try an Hour of Code with Khan Academy https://www.khanacademy.org/hourofcode
Tutoriales de una hora de JAVASCRIPT, HTML/CSS y SQL

Go beyond the hour (Khan Academy)
Intro to JS: Drawing & Animation
https://www.khanacademy.org/computing/computer-programming/programming
Intro to HTML/CSS: Making webpages
https://www.khanacademy.org/computing/computer-programming/html-css

Codeacademy http://www.codecademy.com/ Learn to code interactively. (Java, HTML & CSS, JavaScript, PHP, Python, Ruby...)

MIT App Inventor http://appinventor.mit.edu/explore/hour-of-code.html
Creación de apps. Al igual que Scratch, utiliza bloques de código.

Fabric (de Twitter) https://get.fabric.io/ "una plataforma para que los desarrolladores puedan crear grandes applicaciones"

Programamos videojuegos y apps: http://programamos.es/ Al igual que Scratch, utiliza bloques de código. Busca fomentar la programación en estudiantes. Contiene recursos para profesores

Android Training https://developer.android.com/training/index.html "sets of lessons within classes that describe how to accomplish a specific task with code samples you can re-use in your app "

PseInt http://pseint.sourceforge.net/ "PseInt es una herramienta para asistir a un estudiante en sus primeros pasos en programación. Mediante un simple e intuitivo pseudolenguaje en español (complementado con un editor de diagramas de flujo), le permite centrar su atención en los conceptos fundamentales de la algoritmia computacional, minimizando las dificultades propias de un lenguaje".

Gdevelop http://compilgames.net/main-es.html Programa para crear juegos multiplataforma. No se requieren conocimientos previos de programación

Alice http://www.alice.org/index.php
"Alice is an innovative 3D programming environment that makes it easy to create an animation for telling a story, playing an interactive game, or a video to share on the web. Alice is a freely available teaching tool designed to be a student's first exposure to object-oriented programming."

Descubre la programación http://descubre.inf.um.es/ Para iniciarse en Java:
"En esta web podrás aprender programación paso a paso junto a tus amigos, y casi sin darte cuenta... ¿Empezamos?" (Web de la Facultad de Informática de la Universidad de Murcia)

Python http://www.educ.ar/dinamico/UnidadHtml__get__07d722ce-4b48-11e1-80db-ed15e3c494af/index.html Para iniciarse en Python
"Python es un excelente programa para comenzar a dar los primeros pasos en programación.

Trabajando colaborativamente con sus alumnos podrán ingresar al mundo de la programación orientada a objetos, programación funcional y programación estructurada. El software es libre y de código abierto. Esta cualidad es una gran oportunidad para que los alumnos más avanzados desarrollen procesos mentales."
Viene con un Mini tutorial de Python: http://crysol.org/node/43

Learn Code The Hard Way http://learncodethehardway.org/ Para aprender Python, Ruby, C, Regex y SQL
"By having students get code working first, and explaining it second, you cut down on much of the difficulty of explaining programming concepts to the uninitiated".
"*Learn Python The Hard Way* is the most successful beginner programming book on the market".
La web tiene materiales gratuitos y de pago.

Code School https://www.codeschool.com/ Learn by Doing. Ofrece diversos cursos de programación, tanto gratuitos como de pago.

Coursera https://www.coursera.org/ Los mejores cursos del mundo (también de programación). Coursera es una plataforma de cursos MOOC impulsada por prestigiosas universidades como Stanford o Princeton.

MIT: OpenCourseWare http://ocw.mit.edu/index.htm
Course Finder/ Topic/ Engineering/ Computer Science
Introductory Programming Courses http://ocw.mit.edu/courses/intro-programming/

edX https://www.edx.org/course/subject/computer-science Curso de Python y otros lenguajes.

BASIC-256 (http://www.basic256.org/)

An easy to use BASIC language and IDE for education.

BASIC256 Related Pages:
http://www.basicbook.org
Introduction to programming using the BASIC256 language. Originally written for middle-school aged students but suitable for anybody interested in learning to program.
[libro de Basic-256 en pdf gratuito: http://www.basicbook.org/files/syw2l2p_b256_freeEbookEdition.pdf]
http://basic256.blogspot.com/
Copy paste & run: Lots of sample code including: art, math, geometry, physics, and games.

Why BASIC? (http://www.basic256.org/whybasic)

When anyone asks if their children should learn BASIC, there is always a vocal group that says "NO! It teaches all the wrong things! Learn a modern language like (insert favorite language here)." But this is nonsense.
Just as spoken languages are tools we use to communicate ideas to other people, computer languages communicate ideas to a computer. What's really important is knowing what you want the computer to do, what idea to comunicate. The language you use to express that is largely irrelevant, as long as it works for your purposes. For the purposes of writing simple programs and games, BASIC works just fine, and has for years.
But as for the argument that learning BASIC will hinder further progress in learning about computers, why would that be true? It's not true for spoken languages. Nobody ever said, "I wish I never learned English, because I had to un-learn so many bad habits to learn Japanese." People who learn many spoken languages say it gets easier the more you learn, because you can relate them to each other and recognize patterns. It's the same for computer language, probably even more so, since computer languages are much more similar to each other than spoken languages are.
The problem is, "What programming language should my child learn?" is not the right question

to be asking. The ultimate goal is not to teach students "how to program." **The goal should be to teach them how computers work**. Once you understand how computers work, you get the "how to program" part almost for free. At that point, it's just a matter of syntax.
So how do you make someone understand these ideas that make computers work? Well, it's an interesting question. I guess it would depend on your audience, how much time you had, and probably a thousand other factors. Currently **there are about three distinct methods for teaching computer science**.
The first the traditional approach, which teaches students how to program in the latest industry standard language. My college used this approach. Prior to when I enrolled, students learned C. I learned C++, and now they teach in Java. I struggled, like everyone else, with classes and pointers. I don't know what the current students struggle with, but I'm sure it's not pointers, because they don't exist in Java. I'm not sure when the current students learn pointers, or if they're expected to learn that later on their own, like we were expected to learn how linkers worked, what the compiler was actually doing when we hit the green button, etc. Maybe they never learn what pointers are.
This brings up a dilemna. Either pointers are obsolete, or they're being glossed over in the current computer science curricula. If they're obsolete, my college did a disservice to me by teaching me a faddish programming construct like pointers in CompSci 101. But if they're still valuable, then the current students are getting the short end of the stick by not learning them early.
The traditional approach will always have this drawback. The higher the level at which they teach, the more stuff they have to leave out, by necessity. Now, a good program will cover a range of topics, including how the hardware works. But if the range continues to grow, something's going to get left out.
The second approach is the MIT approach. **MIT uses an excellent book** which teaches the mathematical model of how computer languages and algorithms are structured. It's very theoretical and complete. Any student taking that course will gain a complete understanding of how computer programming works, and be able to implement the entire process from the source code to machine code generation. Interesting to note is that the computer language they use to teach it (scheme) has its roots in Lisp (which is over 40 years old) and will probably never change. This is an indication that what they're teaching is universally valuable. It's not just this year's industry concensus.
So why not just use the MIT approach? Well, consider MIT's audience. I'd wager that nobody in the MIT computer science curriculum is taking CompSci 101 and writing their first computer program. These people have probably lived and breathed computers all of their lives, and want to learn something that's difficult to learn on their own. They want deep understanding, and probably already have all the motivation they need.
Now consider the audience we'd like to teach, young children. The approach MIT takes is going to fail with them. It's abstract, mathematically challenging, and you don't get to draw pictures on the screen.
There's a third method, and while no college uses it, I'm fairly certain that my generation would find it familiar. It's the bottom-up approach. In the bottom-up approach, you'd start with a machine that has a small subset of functionality. I had a Commodore 64, which had a BASIC on it that was just a step above assembly language. You would write your first program and run it:
Ian Rules!
And then you'd add more. You use a GOTO statement to print that over and over. You'd see exactly how the computer jumped around in memory doing things in the order you specified.
What's the point? Well, **one of the most necessary things when teaching Computer Science is to de-mystify the machinery**. High level languages like Java or Python don't do this. What's needed is a language that's directly analagous to how computers actually work, like assembly language.
Once you learn assembly language, nothing else is ever too difficult. Is that because assembly

language is so much harder than everything else? Well, not really. Compared to Java and its libraries, assembly language is tiny -- learning it all is simple. It's because **when you understand assembly, you understand computers**. You know that "functions" are nothing more than a jump with some stack manipulation to make it look like you're not using GOTO. You know that "objects" are nothing more than a collection of function pointers and data with compiler-enforced rules about how it all can be accessed.

So should children learn assembly language first? Certainly not. It's way too difficult to do fun things that a child would want to do, like write games, in assembly. They'd get bored and go do something else. Ninety percent of teaching children is keeping them interested long enough for them to absorb information. But if assembly language were easier and more fun, wouldn't it be an ideal language to teach children?

The truth is, the BASIC most of us grew up with was hopelessly low-level. At the same time, it was simple enough for a 7 year old child to understand. Why is that? Well, it's because computers are inherintly stupid, simple machines. There's no magic in there, despite what the Intel ads say. Making the transition from Commodore BASIC to Commodore assembly language was simple, because Commodore BASIC was so limited, just like assembly. They even had the same control constructs, like GOTO.

That's where BASIC comes in. It's as close to the computer hardware as you can get and still do fun things. It's the highest level low-level language there is.

So don't listen to the traditional approach folks. **To teach someone how to program, he first has to understand how computers work**. BASIC will teach him that easily, bottom-up, and he'll stay interested. Then, when he's got that down, he'll easily be able to learn more the MIT way.

Paradigmas de programación

Programación imperativa (procedimental)
https://es.wikipedia.org/wiki/Programaci%C3%B3n_imperativa

Programación funcional
https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional

Programación orientada a objetos
https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

Origen y relaciones entre los distintos lenguajes de programación

[La imagen siguiente ha sido tomada de:
http://www.georgehernandez.com/h/xComputers/Programming/Media/tongues-cleaner.png]

Mother Tongues

Tracing the roots of computer languages through the ages

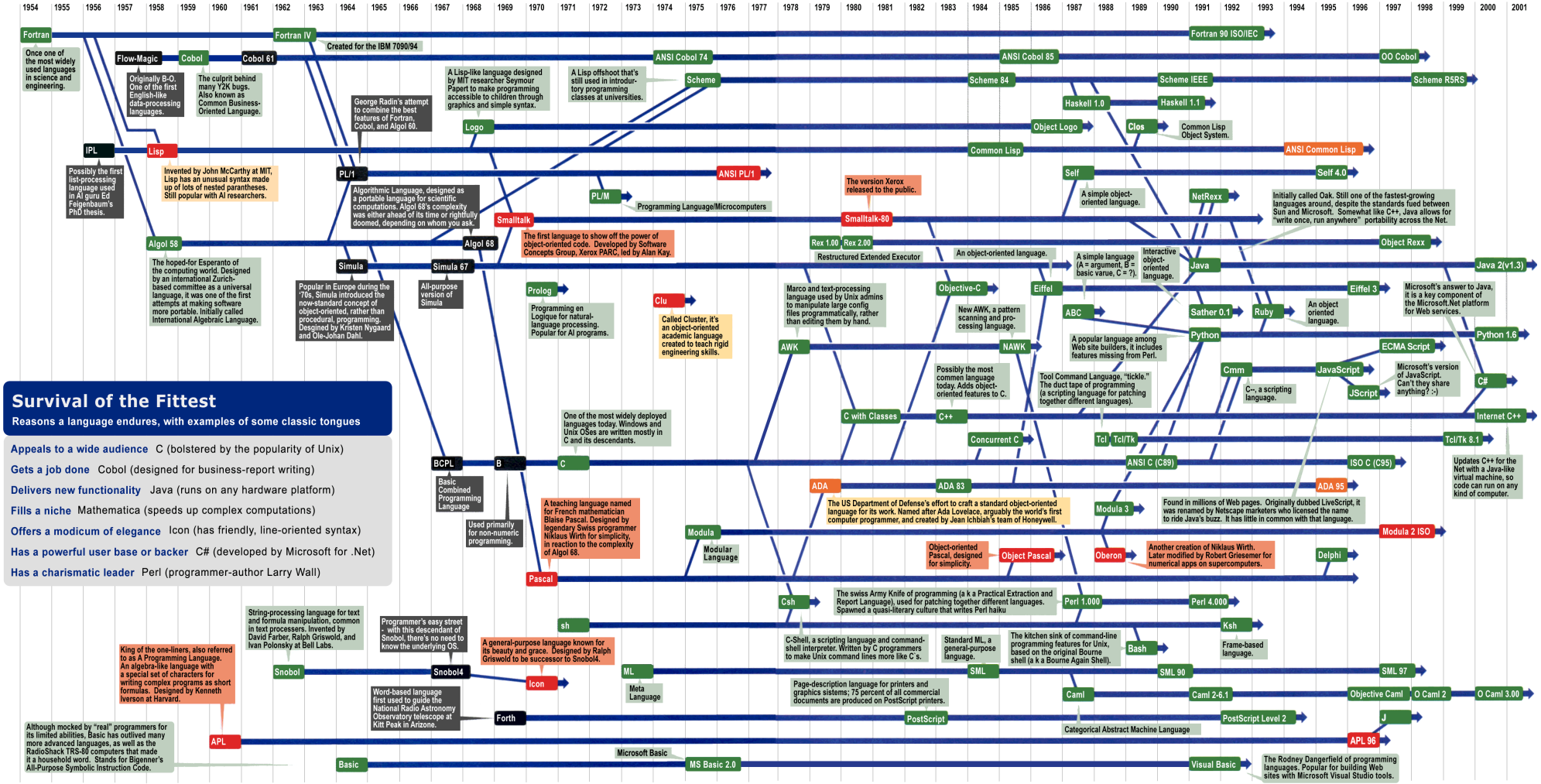
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Musuem in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. - Michael Mendeno

Key

- 1954 **Year Introduced**
- **Active:** thousands of users
- **Protected:** taught at universities; compilers available
- **Endangered:** usage dropping off
- **Extinct:** no known active users or up-to-date compilers
- **Lineage continues**



Survival of the Fittest

Reasons a language endures, with examples of some classic tongues

- **Appeals to a wide audience** C (bolstered by the popularity of Unix)
- **Gets a job done** Cobol (designed for business-report writing)
- **Delivers new functionality** Java (runs on any hardware platform)
- **Fills a niche** Mathematica (speeds up complex computations)
- **Offers a modicum of elegance** Icon (has friendly, line-oriented syntax)
- **Has a powerful user base or backer** C# (developed by Microsoft for .Net)
- **Has a charismatic leader** Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University