11 Computadoras & Computación

Las computadoras hogareñas están siendo convocadas para realizar muchas funciones nuevas, incluyendo la realización de la tarea escolar previamente comida por el perro.
Doug Larson

La Ciencia de la Computación se trata tanto de computadoras como la astronomía de telescopios.
Edsger W. Dijkstra

¿Cuál es el centro de las ciencias de la computación? Mi respuesta es simple –es el arte de programar una computadora.. *Es el arte de diseñar métodos eficientes y elegantes para lograr que una computadora resuelva problemas, teóricos o prácticos, pequeños o grandes, simples o complejos. Es el arte de traducir este diseño en un programa de computación efectivo y preciso.*
C.A.R. Hoare

Imagen: La computadora portátil XO
La foto es cortesía del proyecto OLPC (www.olpc.org)

Hoy hay más computadoras que gente en la mayoría de los campus de las universidades en los Estados Unidos. La computadora portátil (laptop) que se muestra en la página anterior es la laptop XO desarrollada por el Proyecto Una Laptop Por Niño (OLPC). Es una computadora de bajo costo diseñada para niños. El proyecto OLPC aspira a llegar a 2 billones de niños en todo el mundo. Buscan hacer llegar las computadoras XO a sus manos para ayudar en la educación. Su misión es lograr un cambio radical en el sistema de educación global a través de las computadoras. Por supuesto que un proyecto con una misión tan grandilocuente no puede aparecer sin controversias. Ha tenido sus escollos desde sus inicios. Las cuestiones tecnológicas fueron el menor de sus problemas. Han tenido que convencer a gobiernos (es decir, políticos) para que apoyen su misión. Varios países han acordado su apoyo y luego se han echado atrás por todo tipo de razones. Las laptops no estaban bajo venta libre dentro de los Estados Unidos, lo cual ha llevado a otras controversias socio-políticas. De todas maneras, en el primer año de su lanzamiento, el proyecto apunta a tener más de 1 millón de XOs en manos de niños de muchos países en desarrollo. Algo que sucede con la tecnología que siempre ha sido verdad es que una buena idea es inefectiva. Otros jugadores han emergido que están desarrollando computadoras a bajísimo costo para niños. Pronto otros productos competitivos estarán disponibles para todos. Las preguntas que aparecen son: ¿Qué tipo de cambio traerá aparejado esto en el mundo? ¿Cómo serán usadas estas computadoras para enseñar a los niños en las escuelas primarias y secundarias? Etc. También se estarán preguntando si el robot Scribbler puede ser controlado por la XO. Sí, puede.

Han estado escribiendo programas para controlar su robot a través de una computadora. En el camino han visto muchas formas diferentes de controlar al robot y también de organizar sus programas Python. Básicamente, han estado comprometidos con la actividad generalmente entendida como *programación de computadoras* o resolución de problemas basada en computadoras. Deliberadamente hemos evitado usar esos términos porque la percepción general de los mismos convoca imágenes de personajes con protectores de bolsillos y ecuaciones matemáticas complejas que parecen ser evitables o fuera del alcance para la mayoría de las personas. Ojalá que para este momento hayan descubierto que la solución de problemas a través de una computadora puede ser bastante excitante y atrapante. El robot puede haber sido el verdadero motivo que los introdujo en esta actividad y eso fue pensado así. Les estamos confesando que usamos a los robots para atraer un poco su atención hacia la computación. Deben admitir, si están leyendo esto, ¡que el artilugio funcionó! Pero recuerden que el motivo por el cual están sosteniendo un robot propio en sus manos se debe también a las computadoras. El mismo robot es una computadora. Haya sido o no un artilugio, han asimilado muchas ideas clave de computación y de ciencia de la computación. En este capítulo haremos más explícitas estas ideas y también les daremos un sabor de lo que realmente se trata la *ciencia de la computación*. Como sostiene Dijkstra, *la ciencia de la computación es tanto sobre computadoras como la astronomía sobre telescopios*.

Las computadoras son tontas

Supongamos que están alojando a una estudiante internacional de intercambio en su casa. Al poco tiempo de su llegada, le dan a conocer las virtudes de los sándwiches de PB&J (manteca de maní y jalea- quizás equiparable a un pan con manteca y dulce de leche en Argentina). Luego de escucharlos con atención, su boca se empieza a hacer agua y muy educadamente les piden si podrían compartir la receta con ella. Ustedes la escriben en un pedazo de papel y se la dan.

Realizar la siguiente actividad: Adelante, escriban la receta para hacer un sándwich de PB&J.

De verdad, traten de escribirla, ¡insistimos!

OK, ahora tienen una receta para hacer sándwiches de PB&J.

Realizar la siguiente actividad: Usen la receta de arriba para hacerse un sándwich de PB&J. Intenten seguir las instrucciones lo más *literalmente* que puedan. Si lograron realizarse con éxito el sándwich, ¡felicitaciones! ¡Disfrútenlo! ¿Creen que su amiga podrá seguir la receta y también disfrutar de los sándwiches de PB&J?

Deben admitir que escribir una receta para sándwiches de PB&J al principio parecía trivial, pero al sentarse a escribirlo no les queda otra opción que cuestionarse varias cosas que se han dado por supuestas: ¿Sabe ella qué es la manteca de maní? ¿Debo recomendar una marca específica? ¿Mermelada Ditto? A todo esto, se acordaron de mencionar que se trata de *jalea de uva*? ¿El tipo de pan a usar? ¿Será pre-cortado? Si no, ¿necesitan un cuchillo para cortar la rodaja de pan? ¿Especificaron cuán gruesas deberían ser las rodajas de pan? ¿Debería usar el mismo cuchillo para untar la manteca de maní y la jalea? Etc. La cuestión es que, con tanto nivel de detalle, uno puede seguir y seguir... ¿Su amiga sabe cómo usar un cuchillo para untar manteca o jalea en una rodaja de pan? De repente, una tarea aparentemente trivial se transforma en un ejercicio desalentador. En realidad, al escribir su receta, dan varias cosas por hecho: que ella sabe lo que es un sándwich, que requiere rodajas de pan, untar cosas en rodajas, y juntarlas. ¡Listo, tienen un sándwich!

Piensen en la cantidad de recetas que han sido publicadas en libros de cocina de todo el mundo. La mayoría de los buenos autores de libros de cocina empiezan con un plato, lo escriben en una receta, lo prueban varias veces y lo refinan de diferentes maneras. Previo a la publicación, la receta es probada por otros. Después de todo, la receta es para que otros la sigan y recreen un plato. La receta es revisada y ajustada basándose en el feedback (la devolución) de los probadores (o testeadores) de la receta. Lo que los autores de libros de cocina asumen es que tendrán las suficientes competencias para seguir la receta y recrear el plato para su satisfacción. Quizás nunca resulte el mismo plato que preparó el autor. Pero les dará una base sobre la cual improvisar. ¿No sería lindo que hubiera una forma exacta de seguir una receta para que terminaran con exactamente el mismo plato que el del autor del libro de cocina cada vez que prepararan ese plato? ¿Sería bueno? Eso depende de sus gustos y preferencias. Sólo por diversión, aquí hay una receta de unos ricos Brochetas de pollo al Azafrán.

Brochetas de pollo al azafrán

Ingredientes

1lb de pechuga de pollo deshuesada, cortada en cubos de 1-2 pulgadas.
1 media cebolla en rodajas
1 cucharadita de hebras de azafrán
1 lima
1 cucharada de aceite de oliva
Sal y pimienta negra a gusto

Preparación

1. Mezclar el pollo y las cebollas en un bowl no-reactivo.
2. Apretar con los dedos y agregar las hebras de azafrán
3. Agreguen el jugo de lima, el aceite de oliva, la sal y la pimienta.
4. Dejar reposar en la heladera por lo menos por 30 minutos (o toda la noche).
5. Precalentar una parrilla o un horno a unos 200 grados.
6. Armar los pinchos, descartando las rodajas de cebolla. O poner todo en una bandeja de hornear forrada si se usa el horno.
7. Cocinar a la parrilla o al horno por 12-15 min. hasta que estén listas.

En las recetas de cocina como la de arriba, pueden dar por hecho varias cosas: serán usadas por personas (como ustedes y yo); serán capaces de seguirlas; quizás hasta improvisen. Por ejemplo, en la receta de arriba, no especificamos que se necesitará un cuchillo para cortar el pollo, las cebollas o la lima; o que se necesitará una parrilla o un horno; etc. La mayoría de las recetas asume que serán capaces de *interpretar* y seguir la receta así como está escrita.

Los programas de computadora son también como recetas, hasta cierto punto. Piensen en el programa que escribieron para coreografiar una danza de robot, por ejemplo. Hemos reproducido la versión del Capítulo 3 aquí:

```
# File: dance.py
# Purpose: A simple dance routine
# First import myro and connect to the robot

from myro import *
initialize("com5")

# Define the new functions...

def yoyo(speed, waitTime):
      forward(speed, waitTime)
      backward(speed, waitTime)
      stop()
```

```
def wiggle(speed, waitTime):
	motors(-speed, speed)
	wait(waitTime)
	motors(speed, -speed)
	wait(waitTime)
	stop()

# The main dance program
def main():
	print "Running the dance routine..."
	yoyo(0.5, 0.5)
	wiggle(0.5, 0.5)
	yoyo(1, 1)
	wiggle(1, 1)
	print "...Done"

main()
```

En muchos sentidos, el programa de arriba es como una receta:

Hacer una danza de robot

Ingredientes

1 función yoyo para que el robot vaya hacia adelante y hacia atrás a una velocidad dada
1 función wiggle que le permite al robot menearse a una velocidad determinada

Preparación

1. yoyo a velocidad 0.5, esperar 0.5
2. menearse a velocidad 0.5, esperar 0.5
3. yoyo a velocidad 1, esperar 1
4. menearse a velocidad 1, esperar 1

Luego, de manera similar podrían especificar los pasos involucrados en hacer los movimientos de `yoyo` y `wiggle` en forma de receta. Este puede parecer un ejemplo trivial, pero toca dos puntos importantes: un programa de computadora es como una receta en la cual se detalla el listado de ingredientes y un método o pasos para lograr las tareas dadas; y, como una receta, sus ingredientes y los pasos requieren un cuidadoso planeamiento y reflexión previos. Es de destacar que los programas de computadora son diferentes de las recetas en un aspecto: ¡están diseñados para ser seguidos por una computadora!

Una computadora es un dispositivo tonto diseñado para seguir instrucciones/recetas. Nos ahorraremos los detalles de cómo una computadora hace lo que hace para un curso posterior. Pero es casi un conocimiento popular que todo adentro de la computadora se representa con 0s y 1s. Empezando de 0s y 1s uno puede diseñar esquemas de codificación para representar números, letras del alfabeto, documentos, imágenes, películas, música, etc. y cualquier otra entidad

abstracta que deseen manipular utilizando una computadora. Un programa de computadora es, en última instancia, también representado como una secuencia de 0s y 1s y es con esta forma que a la mayoría de las computadoras les gusta seguir recetas. No importa cuán limitante o degenerado suene esto, pero es la clave del poder de las computadoras. Especialmente al entender que esta simplificación es la que le permite a la computadora manipular cientos de millones de fragmentos de información a cada segundo. El precio que debemos pagar por este poder es que debemos especificar nuestras recetas en forma de programas de computación de una manera más bien formal y precisa. Tanto que no hay lugar para la improvisación: ninguna vaguedad al estilo ¨pizca de sal¨, como en las recetas de cocina, es aceptable. Aquí es donde entran en escena los *lenguajes de programación*. Los científicos de la computación especifican sus recetas computacionales usando *lenguajes de programación*. Ustedes han estado usando el lenguaje de programación Python para escribir sus programas para el robot. Otros ejemplos de lenguajes de programación son Java, C++ (pron.: ce más más; en inglés ¨si plus plus¨), C# (pron.: ce sharp, en inglés ¨si sharp¨), etc. ¡Existen más de 2000 lenguajes de programación!

Realizar la siguiente actividad: ¿Pueden averiguar cuántos lenguajes de programación hay? ¿Cuáles son los diez lenguajes de programación más usados?

Previo a la existencia de lenguajes de programación, las computadoras se programaban usando secuencias de 0s y 1s. ¡No es necesario aclarar que muchas personas se volvieron locas con esto! Los lenguajes de programación, como Python, permiten una forma más amigable para que los programadores escriban sus programas. Los lenguajes de programación proveen un acceso fácil a codificaciones que representan el tipo de cosas con las que nosotros, los humanos, nos relacionamos. Por ejemplo, el enunciado Python:

```
sentidoDeLaVida = 42
```

es un comando para que la computadora asocie el valor 42 con el nombre `sentidoDeLaVida`. De esta manera, podemos pedirle a la computadora que verifique que es efectivamente 42:

```
if sentidoDeLaVida == 42:
    speak("Eureka!")
else:
    speak("Qué hacemos ahora?")
```

Una vez más, sería bueno recordarles que la elección del nombre sentidoDeLaVida, no significa realmente que estamos hablando acerca de “*el sentido de la vida”*. Bien podríamos haberlo llamado `timbuktoo`, como en:

```
timbuktoo = 42
```

Como pueden ver, ¡las computadoras son realmente tontas! Realmente depende de nosotros, los programadores, el asegurarnos de usar los nombres consistentemente y elegirlos, en primer lugar, cuidadosamente. Pero, al crear un lenguaje como Python, hemos creado una notación formal que al traducirse a 0s y 1s, cada enunciado querrá decir solamente una cosa, sin otras interpretaciones. Esto lo hace diferente a una receta de cocina.

El robot sale a comprar huevos frescos

Las recetas, sin embargo, forman una buena base conceptual para comenzar a pensar en un programa para resolver un problema. Digamos que tienen en mente hacer su *Strudel de manzana* favorito. Saben que necesitarán manzanas. Quizás es la temporada de manzanas lo que motivó la idea. También necesitarán masa. Pero antes que nada, necesitarán esa receta que les dio la abuela.

Cada vez que se nos pide que resolvamos un problema usando una computadora, empezamos armando un plan borrador para resolverlo. Es decir, trazamos una estrategia. Esto luego se refina en pasos específicos, quizás se identifican y nombran algunas variables, etc. Una vez que están convencidos de tener una forma de resolver el problema, lo que tienen es un *algoritmo*.

La idea de un algoritmo es fundamental para la ciencia de la computación, así que dedicaremos un tiempo aquí a desarrollar esta noción. Quizás la mejor manera de relacionarnos con esta idea es con un ejemplo. Supongamos que un robot entra en un almacén a comprar una docena de huevos frescos. Supongamos que es capaz de hacerlo, ¿cómo se asegurará de que ha seleccionado los huevos más frescos disponibles?

Su robot personal no es capaz de este tipo de tarea, pero imaginen que sí. Mejor aún, dejemos la mecánica de lado, veamos cómo *ustedes* irían a comprar los huevos más frescos. Bueno, de algún modo necesitarían saber qué fecha es la de hoy. Digamos que es 15 de septiembre de 2007 (¿por qué esta fecha? ¡Ya lo sabrán!). También saben que las cajas de huevos generalmente tienen la fecha de vencimiento. De hecho, el USDA (Departamento de Agricultura de los Estados Unidos) ofrece voluntariamente, sin costo, programas de certificación para granjas de huevos. Un granjero de huevos puede participar voluntariamente en el programa de certificación de huevos de USDA, en el cual el USDA realiza inspecciones regulares y brinda ayuda para categorizar los huevos por tamaños. Por ejemplo, los huevos generalmente se clasifican en Grado AA, Grado A o Grado B. La mayoría de los almacenes tienen huevos Grado A. También vienen en varios tamaños: Extra Grande, Grande, Pequeño etc. Pero más interesante aún es que el sistema de etiquetado de las cajas tiene una información muy útil codificada en el mismo.

Etiquetas en cajas de huevos

Cada caja de cartón certificada tiene por lo menos tres fragmentos de información (ver la imagen a la derecha): un ¨consumir antes de¨ o ¨fecha de vencimiento¨, un código que identifica la granja de la cual vienen los

huevos y una fecha en que los huevos fueron envasados en la caja. La mayoría de la gente compra los huevos mirando la fecha de ¨consumir antes de¨ o ¨fecha de vencimiento¨. Sin embargo, la información acerca de si son frescos está codificada en la fecha de envasado. Para agregar a la confusión, la fecha está codificada como el día del año. Por ejemplo, miren la caja de arriba mostrada en la página anterior. Su fecha de ¨consumir antes de¨ (¨sell by¨) es 4 de octubre. "P1107" es el código de granja. Este cartón fue envasado en el 248avo día del año. Además, el USDE requiere que todos los huevos certificados se envasen dentro de los 7 días de haber sido puestos. Por lo tanto, los huevos de la caja superior se pusieron en algún momento entre el día 241 y el 248 de 2007. ¿Qué fechas corresponden a esos días?

A continuación, observen la caja de abajo. Esos huevos tienen un ¨consumir antes de¨ más tardío (18 de octubre) pero una fecha de envasado anterior: 233. Es decir, esos huevos fueron puestos en algún momento entre el día 226 y el 233.

¿Cuáles son los huevos más frescos?

Aunque la fecha de ¨consumir antes de¨ en la segunda caja es de dos semanas más tarde, la primera caja contiene los huevos más frescos. De hecho, los huevos en la caja de arriba fueron puestos por lo menos dos semanas después.

La fecha de envasado está codificada en un número de 3 dígitos. Por lo tanto, los huevos envasados el 1 de enero serán etiquetados: 001; los huevos envasados el 31 de diciembre, 2007 serán etiquetados: 365.

Realizar la siguiente actividad: Vayan al sitio Web del USDA (www.usda.gov) y vean si pueden averiguar de qué granja vinieron las dos cajas de huevos.

Para un robot, el problema de comprar los huevos más frescos consiste en averiguar, dada la fecha de envase, ¿cuál fue la fecha en que se envasaron los huevos?

Ajusten sus cinturones, estamos por embarcarnos en un viaje computacional único...

Diseñar un algoritmo

Hasta aquí, hemos acotado el problema a las siguientes especificaciones:

```
Entrada
    Codificación de tres dígitos de la fecha de envasado

Salida
     Fecha en que fueron envasados los huevos.
```

Por ejemplo, si la fecha de envasado fue codificada como 248, ¿cuál será la fecha?

Bueno, eso depende. Podría ser 4 ó 5 de septiembre, dependiendo de si fue un año bisiesto o no. Entonces, resulta que el problema de arriba también requiere que

sepamos de qué año se trata. Resolver uno o dos problemas como ejemplo es siempre una buena idea porque ayuda a identificar información que falta que puede ser crítica para resolver el problema. Dado que también necesitamos conocer el año, podemos pedirle al usuario que lo ingrese a la vez que ingresa el código de 3 dígitos. La especificación del problema entonces se transforma en:

Entrada
    Codificación de tres dígitos de fecha
    Año actual

Salida
    Fecha en que se envasaron los huevos

Ejemplo:
    Entrada: 248, 2007
    Salida: Los huevos se envasaron el 5 de septiembre de 2007

¿Alguna idea sobre cómo resolverían el problema? Siempre ayuda que intenten hacerlo ustedes, con lápiz y papel. Tomen el ejemplo de arriba y vean cómo llegarían a la fecha de salida. Mientras lo están resolviendo, traten de escribir el proceso de resolución. Su algoritmo o receta será muy similar.

La etimología de algoritmo

Se cree que la palabra algoritmo, un anagrama de logaritmo, deriva de Al-Khowarizmi, un matemático que vivió de 780-850 DC. Su nombre completo era Abu Ja'far Muḥammad ibn Mūsā al-Khwārizmī, (Mohammad, padre de Jafar, hijo de Moses, un Khwarizmian). Gran parte del conocimiento matemático de la Europa medieval deriva de traducciones latinas de sus trabajos.

En 1983, la Unión Soviética imprimió la estampilla que se muestra arriba en homenaje a su 1200avo aniversario.

Supongamos que estamos intentando codificar el input 248, 2007. Si lo hicieran a mano, usando una lapicera y un papel, el proceso sería algo así:

La fecha no es en enero porque tiene 31 días y 248 es mucho mayor que 31.
Restemos 31 de 248: 248 - 31 = 217

217 también es mayor que 28, la cantidad de días de febrero, 2007.
Entonces, restemos 28 de 217: 217 - 28 = 189

189 es mayor que 31, la cantidad de días de marzo.
Restemos 31 de 189: 189 - 31 = 158

158 es mayor que 30, la cantidad de días de abril.
Entonces: 158 - 30 = 128

128 es mayor que 31, la cantidad de días de mayo.
Por lo tanto: 128 - 31 = 97

97 es mayor que 30, la cantidad de días de junio.
97 - 30 = 67

67 es mayor que 31, la cantidad de días de julio.
67 - 31 = 36

```
36 es mayor que la cantidad de días de agosto (31).
36 - 31 = 5

5 es menor que la cantidad de días de septiembre.
Por lo tanto debe ser el 5to día de septiembre.

La respuesta es: 248avo día de 2007 es 5 de septiembre de 2007.
```

Eso era evidentemente demasiado repetitivo y tedioso. Pero ahí es donde entran las computadoras. Observen el proceso de arriba y vean si hay un patrón para los pasos realizados. A veces, ayuda probar con otro ejemplo.

Realizar la siguiente actividad: Supongamos que el día y año de entrada es: 56, 2007. ¿Cuál es la fecha?

Cuando vean los cómputos de ejemplo que han realizado, encontrarán varios patrones. Identificarlos es la clave para diseñar un algoritmo. A veces, para que esto sea más sencillo, ayuda identificar o nombrar los fragmentos de información clave que se están manipulando. Generalmente esto comienza con las entradas y salidas identificados en la especificación del problema. Por ejemplo, en este problema, las entradas son: día del año, año actual. Comiencen por asignarle estos valores a nombres de variables específicos. Es decir, asignemos el nombre `dia` para representar el día del año (248 en este ejemplo), y `anio`[1] como el nombre para registrar el año actual (2007). ¡Noten que no elegimos nombrar a ninguna de estas variables `timbuktu` o `sentidoDeLaVida`!

También noten que repetidamente deben restar la cantidad de días del mes, empezando por enero. Asignemos la variable llamada `mes` para registrar el mes que se esté considerando.

A continuación, pueden reemplazar los nombres `dia` y `anio` en el cómputo de ejemplo:

```
Entrada:
  dia = 248
  anio = 2007

# Empezar considerando enero
mes = 1

La fecha no está mes = 1 porque tiene 31 días y 248 es mucho mayor que 31.
dia = dia - 31

# siguiente mes
mes = 2

dia (= 217) es también mayor que 28, el # de días en mes = 2
dia = dia - 28
```

1 Nota del Traductor: La letra ñ no forma parte de los caracteres permitidos para nombrar variables o funciones. Sucede lo mismo con los caracteres acentuados.

```
# siguiente mes
mes = 3
dia (= 189) es mayor que 31, el # de días en mes = 3.
dia = dia - 31

# siguiente mes
mes = 4
dia (= 158) es mayor que 30, el # de días en mes = 4.
dia = dia - 30

# siguiente mes
mes = 5
dia (= 128) es mayor que 31, el # de días en mes = 5.
dia = dia - 31

# siguiente mes
mes = 6
dia (= 97) es mayor que 30, el # de días en mes = 6.
dia = dia = 30

# siguiente mes
mes = 7
dia (= 67) es mayor que 31, el # de días en mes = 7.
dia = dia - 31

# siguiente mes
mes = 8
dia (= 36) es mayor que el # de días en mes = 8.
dia = dia - 31

# siguiente mes
mes = 9
dia (= 5) es menor que el # de días en mes = 9.
Por lo tanto debe ser el 5to día de septiembre.

La respuesta es: 5/9/2007
```

Observen ahora cuán repetitivo es el proceso. La repetición puede ser expresada más concisamente como se muestra abajo:

```
Entrada:
  dia
  anio

# empezar con mes = 1, para enero
mes = 1
repetir
  si día es menor que número de días en mes
    dia = dia – numero de días en mes
    # siguiente mes
    mes = mes + 1
  si no
    terminado

Salida: día/mes/año
```

Ahora empieza a parecer una receta o un algoritmo. Pruébenlo con las entradas del ejemplo de arriba y asegúrense de obtener los resultados correctos. Además, asegúrense de que este algoritmo funcione en los casos límite: 001, 365.

Treinta días tiene septiembre

Podemos afinar aún más el algoritmo de arriba: algo que dejamos sin especificar arriba es el cómputo de la cantidad de días en un mes. Esta información debe ser explicitada para que la computadora sea capaz de seguir la receta. Entonces, ¿cómo computamos la cantidad de días en un mes? La respuesta parece ser sencilla. Muchos recordarán el siguiente poema:

Treinta días tiene septiembre
Abril, junio y noviembre
El resto treinta y uno
Excepto febrero el mocho
Que tiene veintiocho
(Y veintinueve en cada año bisiesto)

Desde la perspectiva del diseño, podemos asumir que tenemos un ingrediente, una función en este caso, llamada `diasEnMes` que, dado un mes y un año, computará y devolverá la cantidad de días en el mes. Es decir, que podemos afinar nuestro algoritmo de la siguiente manera:

```
Ingredientes:
  1 función diasEnMes(m, a): devuelve la cantidad de días  en el mes m y en año a.

Entrada:
  dia
  anio

# empezar con mes = 1, para enero
mes = 1
repetir
  si el día es menor que la cantidad de días en el mes
        dia = dia - diasEnMes(mes, anio)
        # siguiente mes
        mes = mes + 1
  si no
    terminado

Salida: día/mes/año
```

Ahora, tenemos que resolver el problema secundario:

```
Entrada
   mes, M
   anio, Y
Salida
   cantidad de días en mes, M en año, Y
```

A primera vista parece fácil, el poema de arriba especifica que abril, junio, septiembre y noviembre tienen 30 días, y el resto, con excepción de febrero, tienen 31. Febrero tiene 28 ó 29, dependiendo de si cae en un año bisiesto o no. Por lo tanto, fácilmente podemos elaborar una receta o un algoritmo para esto de la siguiente manera:

```
Entrada:
  m, a

si m es abril (4), junio(6), septiembre(9), o noviembre (11)
  dias = 30
si no, si m es febrero
       si a es año bisiesto
           dias = 29
       si no
           dias = 28
si no
  (m es enero, marzo, mayo, julio, agosto, octubre, diciembre)
   dias = 31

Salida:
  dias
```

Aún hay un detalle más: ¿cómo sabemos si es un año bisiesto?

Primero, intenten responder a la pregunta, *¿Qué es un año bisiesto?*

Nuevamente, podemos afinar el algoritmo de arriba suponiendo que tenemos otro ingrediente, una función `esBisiesto`, que determina si un año dado es bisiesto o no. Entonces podemos escribir el algoritmo de arriba como:

```
Ingredientes:
  1 función esBisiesto(a)
    devuelve True si y es año bisiesto, de lo contrario, false


Entrada:
  m, a

si m es abril (4), junio(6), septiembre(9), o noviembre (11)
  dias = 30
si no, si m es febrero
       si esBisiesto(a)
           dias = 29
       si no
           dias = 28
si no
  (m es enero, marzo, mayo, julio, agosto, octubre, diciembre)
   dias = 31

Salida:
  dias
```

A la mayoría nos han enseñado que un año bisiesto es un año divisible por 4. Es decir, el año 2007 no es bisiesto, dado que 2007 no es divisible por 4, pero 2008 sí lo es, porque es divisible por 4.

Realizar la siguiente actividad: ¿Cómo determinan si algo es divisible por 4? Prueben la solución en los años 1996, 2000, 1900, 2006.

Años bisiestos: Bula Papal

Para diseñar una receta o un algoritmo que determine si el número correspondiente a un año es un año bisiesto o no es directo si aceptan la definición de la última sección. Por lo tanto, podemos escribir:

```
Entrada
   a, un año

Salida
   True (verdadero) si a es año bisiesto, de lo contrario, false (falso)
```

Método

```
 si a es divisible por 4
    es un año bisiesto, o True
  si no
    no es un año bisiesto, o False
```

Sin embargo, ésta no es la historia completa. El calendario occidental que seguimos se llama *Calendario Gregoriano*, y fue adoptado en 1582 por una Bula Papal dictada por el Papa Gregorio XIII. El Calendario Gregoriano define un año bisiesto agregando un día extra cada cuatro años. Sin embargo, se le aplica una corrección cada 100 años, lo cual complica un poco más la situación: Los centenarios no son años bisiestos a menos que sean divisibles por 400. Es decir que los años 1700, 1800, 1900, 2100 no son años bisiestos aunque sean divisibles por 4. Pero los años 1600, 2000, 2400 son años bisiestos. Para más información sobre esto, ver los ejercicios al final del capítulo. Nuestro algoritmo para determinar si un año es bisiesto se puede afinar tal como se muestra abajo:

```
Entrada
  a, un año

  si a es divisible por 400
     es un año bisiesto, o True
  si no, si a es divisible por 100
     no es un año bisiesto, o Falso
  si no, si a es divisible por 4
     es un año bisiesto, o True
  si no
     no es un año bisiesto, o Falso
```

Finalmente, hemos logrado diseñar todos los algoritmos o recetas que se requieren para resolver el problema. Quizás habrán notado que usamos algunas construcciones familiares para elaborar nuestras recetas o algoritmos. Luego, echemos una mirada rápida a las construcciones que se usan para expresar algoritmos.

Componentes esenciales de un algoritmo

Los científicos de la computación expresan soluciones a problemas en términos de algoritmos, que básicamente son recetas más detalladas. Los algoritmos pueden ser usados para expresar cualquier solución y sin embargo se componen de elementos muy básicos.

1. Los algoritmos son recetas paso-por-paso que identifican claramente las entradas y salidas.
2. Los algoritmos nombran las entidades que son manipuladas o usadas: variables, funciones, etc.
3. Los pasos en el algoritmo se siguen en el orden en que están escritos (de arriba a abajo).
4. Algunos pasos pueden especificar decisiones (si-entonces) sobre la elección de algunos pasos.
5. Algunos pasos pueden especificar repeticiones (loops) de pasos.
6. Todos los de arriba pueden ser combinados de cualquier manera.

Los científicos de la computación proclaman que las soluciones/algoritmos a *cualquier* problema pueden ser expresadas usando las construcciones de arriba. ¡No necesitan más! Esta es una poderosa idea y es lo que hace tan versátiles a las computadoras. Desde una perspectiva más amplia, si esto es verdad, entonces pueden ser usados como herramientas para pensar cualquier problema en el universo. Volveremos sobre esto más adelante en el capítulo.

Lenguajes de programación

También, como han visto anteriormente al escribir programas Python, los lenguajes de programación (Python, por ejemplo) proveen vías formales para especificar los componentes esenciales de los algoritmos. Por ejemplo, el lenguaje Python provee una manera para asociar valores con variables que ustedes nombren, provee una forma secuencial de codificar los pasos, provee los enunciados condicionales si-entonces (if-then) y también provee las construcciones while y for para expresar repeticiones. Python también provee modos de definir funciones además de maneras de organizar grupos de funciones relacionadas en librerías o módulos que pueden importar y usar según las necesidades. A modo de ejemplo, abajo les brindamos el programa Python que codifica el algoritmo `esBisiesto` mostrado arriba:

```
def esBisiesto(a):
  '''Retorna true si a es bisiesto y false en caso contrario.'''

  if a %400 == 0:
    return True
  elif a % 100 == 0:
    return False
  elif a % 4 == 0:
    return True
  else:
    return False
```

El mismo algoritmo, cuando es expresado en C++ (o Java) se verá así:

```
bool esBisiesto(int a) {
  //  Retorna true si a es bisiesto y false en caso contrario
  if (a % 400 == 0)
    return true
  else if (a % 100 == 0)
    return false
  else if (a % 4 == 0)
    return true
  else
    return false
}
```

Como pueden ver, hay variaciones sintácticas definidas entre lenguajes de programación. Pero por lo menos en los ejemplos de arriba, la codificación del mismo algoritmo se ve parecida. Sólo para dar otro sabor, aquí está la misma función expresada en el lenguaje de programación CommonLisp.

```
(defun esBisiesto (a)
  (cond
    ((zerop (mod a 400)) t)
    ((zerop (mod a 100)) nil)
    ((zerop (mod a 4)) t)
    (t nil)))
```

Nuevamente, esto puede parecer raro, pero aún está expresando el mismo algoritmo.

Lo más interesante de todo es que, dado un algoritmo, puede haber muchas formas de codificarlo, aún en el mismo lenguaje de programación. Por ejemplo, aquí hay otra manera de escribir la función `esBisiesto` en Python:

```
def esBisiesto(a):
  '''Retorna true si a es bisiesto y false en caso contrario.'''

  if ((a % 4 == 0) and (a % 100 != 0)) or (a % 400 == 0):
```

```
    return True
  else:
    return False
```

Nuevamente, éste es exactamente el mismo algoritmo. Sin embargo, combina todos, los prueba en una sola condición: y es divisible por 4 o por 400 pero no por 400. La misma condición puede ser usada para escribir una versión aún más sucinta:

```
def esBisiesto(a):
        '''Retorna true si a es bisiesto y false en caso contrario.'''

         return ((a % 4 == 0) and (a % 100 != 0)) or (a % 400== 0)
```

Es decir, devolver cual sea el resultado (`True`/`False`) de la prueba para que `a` sea un año bisiesto. En algún sentido, expresar algoritmos en un programa se parece a expresar un pensamiento o un conjunto de ideas en un párrafo o una narración. Puede haber muchas formas de codificar un algoritmo en un lenguaje de programación. Algunos parecen más naturales, y algunos más poéticos, o ambas cosas, y, como en la escritura, algunos pueden ser muy enrevesados. Como en la buena escritura, la habilidad para una buena programación viene de la práctica y, aún más importante, se aprende leyendo programas bien escritos.

De los algoritmos a un programa en funcionamiento

Para poder resolver el problema de los huevos frescos, se deben codificar todos los algoritmos en funciones Python y luego unirlos como un programa que funciona. Abajo presentamos una versión:

```
# File: huevosFrescos.py

def esBisiesto(a):
        '''Retorna true si a es bisiesto y false en caso contrario.'''
         return ((a % 4 == 0) and (a % 100 != 0)) or (a % 400== 0)


def diasEnMes(m,a):
  '''Retorna la cantidad de días en el mes m, m (1-12)
    ien el año a.'''

  if (m == 4) or (m == 6) or (m == 9) or (m == 11):
    return 30
  elif m == 2:
    if esBisiesto(a):
      return 29
    else:
      return 28
  else:
    return 31

def main():
```

```
  '''Dado un dia del año (ejemplo. 248, 2007),
     lo convierte a la fecha correspodiente (ejemplo. 5/9/2007)'''

  #Entrada: dia, año
  dia, anio = input("Ingrese dia , año: ")

  # Comenzamos con mes = 1, para enero
  mes = 1

  while dia > diasEnMes(mes, anio):
     dia = dia - diasEnMes(mes, anio)

     # Siguiente mes
     mes = mes + 1

  # Listo, Salida: dia/mes/año
  print "La fecha es:  %1d/%1d/%4d" % (dia, mes, anio)

main()
```

Si guardan este programa en un archivo, `huevosFrescos.py`, podrán hacerlo correr y probarlo con varias fechas. Pruébenlo. Aquí hay algunas salidas de ejemplo:

```
Ingrese dia , año: 248, 2007
La fecha es: 5/9/2007

>>> main()
Ingrese dia , año: 12, 2007
La fecha es:  12/1/2007

>>> main()
Ingrese dia , año:248, 2008
La fecha es:  4/9/2008

>>> main()
Ingrese dia , año: 365, 2007
La fecha es:  31/12/2007

>>> main()
Ingrese dia , año: 31, 2007
La fecha es:  31/1/2007
```

Todo parece andar bien. Observen cómo probamos o ¨testeamos¨ el programa para distintas entradas de valores para confirmar que nuestro programa está produciendo resultados correctos. Es muy importante probar el programa para un variado conjunto de entradas, asegurándose de incluir todas las condiciones límites: primer y último día del año, mes, etc. Probar programas es un arte en sí mismo y varios libros se han escrito sobre el tema. Uno debe asegurarse de que todos las posibles entradas están probadas para asegurarse de que el comportamiento del programa es aceptable y correcto. Ustedes hicieron esto con los programas de robot haciéndolos correr repetidamente y observando el comportamiento del robot. Lo mismo se aplica a la computación.

Testeo y chequeo de errores

¿Qué pasa si el programa de arriba recibe entradas que están fuera de su alcance? ¿Qué pasa si el usuario ingresa valores hacia atrás (por ej. 2007, 248 en lugar de 248, 2007)? ¿Qué pasa si el usuario ingresa su nombre en lugar de la fecha? (por ejemplo. Paris, Hilton)? Este es el momento de probar todo esto. Prueben el programa y observen su comportamiento con algunos de estas entradas.

Asegurarse de que un programa provea resultados aceptables para todos las entradas es crítico en la mayoría de las aplicaciones. Aunque no hay modo de evitar lo que sucede cuando un usuario ingresa su nombre en lugar de un día y un año, ustedes deberían ser capaces de proteger a sus programas de tales situaciones. Por ejemplo:

```
>>> main()
Ingrese dia , año:  400, 2007
Que corresponde a la fecha, 4/14/2007
```

¡Obviamente, no tenemos un mes 14!

Lo que viene al rescate aquí es darse cuenta de que su programa y la computadora sólo llevarán a cabo lo que ustedes han expresado en el programa. Es decir, pueden incluir facilidades de *chequeo de errores* en su programa para dar cuenta de tales condiciones. En este caso, cualquier valor de entrada de un día que esté fuera del rango 1..365 (o 1..366 para años bisiestos) no será aceptable. Además, también pueden asegurarse de que el programa sólo acepte años mayores que 1582 para el segundo valor de entrada. Aquí está el programa modificado (sólo mostraremos la función `main`):

```
def main():
  '''Dado un dia del año (ejemplo. 248, 2007),
     lo convierte a la fecha correspodiente (ejemplo. 5/9/2007)'''

  #Entrada: dia, año
   dia, anio = input("Ingrese dia , año: ")

   # Validamos los valores ingresados
   if anio <= 1582:
      print "Lo siento. Debés  ingresar un año válido (uno después de 1582). Por favor,
intentalo de nuevo”
      return

   if dia < 1:
      print "Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo de
nuevo”
      return

   if esBisiesto(anio):
      if dia > 366:
```

```
            print "Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo
de nuevo”
          return

  elif dia > 365:
            print "Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo
de nuevo”
       return

  # las entradas son seguras.... seguimos....
  # Comenzamos con mes = 1, para enero
  mes = 1

  while dia > diasEnMes(mes, anio):
    dia = dia - diasEnMes(mes, anio)

    # Siguiente mes
    mes = mes + 1

  # Listo, Salida: dia/mes/año
  print "La fecha es:  %1d/%1d/%4d" % (dia, mes, anio)


main()
```

Aquí están los resultados de algunas de las pruebas sobre el programa de arriba.

```
Ingrese dia , año: 248, 2007
La fecha es: 5/9/2007

>>> main()
Ingrese dia , año: 0, 2007
Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo de nuevo

>>> main()
Ingrese dia , año: 366, 2007
Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo de nuevo

>>> main()
Ingrese dia , año: 400, 2007
Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo de nuevo

>>> main()
Ingrese dia , año: 248, 1492
Lo siento. Debés  ingresar un año válido (uno después de 1582). Por favor, intentalo de
nuevo

>>> main()
Ingrese dia , año: 366, 2008
La fecha es:31/12/2008
```

Empezando de la descripción de un problema, es un viaje largo y cuidadosamente planeado que involucra el desarrollo del algoritmo, la codificación del algoritmo en un programa, y finalmente el testeo y la mejora del programa. Al final son recompensados no sólo con un programa útil, también han afilado sus habilidades para resolver problemas generales. Programar los fuerza a anticipar situaciones

inesperadas y dar cuenta de ellas antes de encontrárselas, que en sí mismo puede ser una maravillosa lección de vida.

Módulos para organizar componentes

Frecuentemente, durante el diseño de programas, terminan diseñando componentes o funciones que pueden ser usadas en muchas otras situaciones. Por ejemplo, en el problema de arriba, escribimos las funciones `esBisiesto` y `diasEnMes` para asistir en la resolución del problema. Sin duda estarán de acuerdo en que hay muchas situaciones en las cuales estas dos funciones pueden resultar útiles (ver ejercicios abajo). Python provee la facilidad *module* (módulo) para ayudarlos a organizar funciones útiles relacionadas entre sí, en un solo archivo que pueden entonces usar una y otra vez cuando lo necesiten. Por ejemplo, pueden tomar las definiciones de las dos funciones y ponerlas separadas en un archivo llamado `calendario.py`. Entonces, pueden *importar* estas funciones cuando las necesiten. Han usado el enunciado `import` de Python para importar funcionalidades de varios módulos distintos: `myro`, `random`, etc. Bueno, ahora saben crear los propios. Una vez que hayan creado el archivo `calendario.py`, pueden importarlo al programa `huevosFrescos.py` como se muestra abajo:

```
from calendario import *

def main():
  '''Dado un dia del año (ejemplo. 248, 2007),
     lo convierte a la fecha correspodiente (ejemplo. 5/9/2007)'''

  #Entrada: dia, año
   dia, anio = input("Ingrese dia , año: ")


   # Validamos los valores ingresados
   if anio <= 1582:
      print "Lo siento. Debés  ingresar un año válido (uno después de 1582). Por favor,
intentalo de nuevo”
      return

   if dia < 1:
      print "Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo de
nuevo”
      return

   if esBisiesto(anio):
      if dia > 366:
            print "Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo
de nuevo”
         return

   elif dia > 365:
            print "Lo siento. Debés  ingresar un día válido (1..365/366). Por favor, intentalo
de nuevo”
      return

   # las entradas son seguras.... seguimos....
   # Comenzamos con mes = 1, para enero
```

```
  mes = 1

  while dia > diasEnMes(mes, anio):
    dia = dia - diasEnMes(mes, anio)

    # Siguiente mes
    mes = mes + 1

  # Listo, Salida: dia/mes/año
  print "La fecha es:  %1d/%1d/%4d" % (dia, mes, anio)


main()
```

También para este momento se les puede haber ocurrido que para cualquier problema dado hay muchas diferentes soluciones o algoritmos. Ante la presencia de varios algoritmos alternativos, ¿cómo deciden cuál elegir? Los científicos de la computación han convertido en su tarea principal el desarrollar, analizar y clasificar algoritmos para ayudar a tomar estas decisiones. La decisión puede estar basada en facilidad de programación, eficiencia, o el número de recursos que lleva para un algoritmo dado. Esto también llevó a los científicos de la computación a crear una clasificación de problemas: de fácil a difícil, pero en un sentido formal. Algunas de los interrogantes abiertos más difíciles en el terreno de los problemas y la computación entran en este dominio de investigación. No entraremos en detalles aquí, pero estas cuestiones incluso han aparecido en varios programas populares de TV (ver foto arriba). Intentaremos darles un sabor acerca de esto a continuación.

Homero contempla ¿P = NP?

La segunda ecuación a la derecha es la ecuación de Euler.

La Complejidad del Tiempo & Espacio

Empecemos con otro problema: Deben viajar del Estado de Washington a Washington DC en los Estados Unidos de América. Para hacer las cosas más interesantes, agreguemos la restricción de que pueden viajar solamente a través de los estados cuyos nombres empiecen con las letras de la palabra ¨mujer¨ (se trabajará con la palabra en ingles: ¨woman¨). Es decir, está bien ir de Washington a Oregon dado que ambos ¨W¨ y ¨O¨ están en la palabra ¨woman¨, pero no está bien ir de Washington a California. ¿Es posible esto? Si lo es, ¿cuántos posibles caminos hay? ¿Cuál de ellos atraviesa la menor/mayor cantidad de estados? Etc.

Si están pensando en cómo resolverlo, dependerán de su conocimiento geográfico sobre los Estados Unidos. Como alternativa, pueden googlear un mapa con los

estados y entonces llegar a la solución. Pero al hacerlo, han tropezado con los dos ingredientes claves de la computación: datos y algoritmo.

Los datos les dan una representación de la información relevante, que en este caso es una forma de conocer cuáles estados son lindantes entre sí. El algoritmo les da una manera de hallar la solución: empezar en Washington, mirar a sus vecinos. Elegir un vecino cuyo nombre satisfaga la restricción, luego mirar el vecino de ese estado, y así sucesivamente. Además, un algoritmo los obliga a hacer determinadas elecciones: si hay más de un vecino elegible, ¿cuál eligen? ¿Qué pasa si terminan en un callejón sin salida?, ¿cómo vuelven a las elecciones que dejaron atrás para explorar pasos alternativos? Etc. Dependiendo de estas decisiones, es probable que terminen con varios algoritmos distintos: uno puede sugerir explorar todas las alternativas simultáneamente; o una que los obliga a elegir, sólo para volver a las otras elecciones en el caso de fracaso. Cada uno de estos impactarán luego en la cantidad de datos que necesitarán guardar para mantener un registro de su progreso.

Desarrollar programas de computadora para resolver cualquier problema requiere que uno diseñe representaciones de datos y elija entre un conjunto de algoritmos alternativos. Los científicos de la computación caracterizan las elecciones de representaciones de datos y algoritmos abstractamente en términos de recursos de *espacio* y *tiempo* de la computadora, necesarios para implementar esas elecciones. Las soluciones varían en términos de la cantidad de espacio (o memoria de la computadora) y tiempo (segundos, minutos, horas, días, años) requeridos en una computadora. Aquí hay algunos ejemplos:

1. Computar el producto de dos números requiere tiempo constante: para una computadora no hay diferencia entre multiplicar 5 por 2 o 5.564.198 por 9.342.100. Estos se llaman *algoritmos de tiempo constante*.

2. Encontrar un número en una lista de *N* números desordenados lleva el tiempo proporcional a ***N***, especialmente si el número que están buscando no está allí. Estos se llaman *algoritmos de tiempo lineal.*

3. Encontrar un número en una lista de N números ordenados (digamos, en orden ascendente) requiere como máximo el $\mathbf{log_2\ N}$ de tiempo. ¿Cómo? Tales algoritmos se llaman algoritmos de tiempo logarítmico.

4. Transformar una imagen de cámara de *NxN* píxeles manipulando todos sus píxeles lleva un tiempo proporcional a $\mathbf{N^2}$. Estos son *algoritmos cuadráticos.*

5. Encontrar un camino de un estado a otro, en un mapa de N estados, dadas ciertas restricciones, pueden llevar un tiempo proporcional a $\mathbf{b^d}$ donde *b* es la cantidad promedio de vecinos de cada estado y *d* es la cantidad de estados que conforman la solución. Generalmente, muchos problemas entran en la categoría donde N^k es el tamaño del problema. Estos se llaman *algoritmos de tiempo polinómico*.

6. También hay varios problemas irresolubles en el mundo.

En el Capítulo 9, al hacer transformaciones de imágenes, nos restringimos a imágenes de tamaños relativamente pequeños. Habrán notado que cuanto más grande es la imagen, más tiempo tarda en transformarse. Esto tiene que ver con la velocidad de la computadora que están usando, así como con la velocidad de implementación del lenguaje de programación. Por ejemplo, una computadora que corre a una velocidad de 4GHz es capaz de hacer aproximadamente 1 billón de operaciones aritméticas básicas en un segundo (o operaciones/segundo). Un programa escrito en el lenguaje de programación C les podría llegar a dar una velocidad de ½ billón de operaciones por segundo en la misma computadora. Esto se debe a las operaciones extra requeridas para implementar el lenguaje C y las tareas adicionales que el sistema operativo está llevando a cabo detrás. Los programas Python corren aproximadamente 20 veces más lento que los programas C. Es decir, un programa Python corriendo en la misma computadora les podría llegar a dar 25 millones de operaciones por segundo como mucho. Una transformación típica, digamos computar el negativo de una imagen de `WxH` píxeles requeriría el siguiente loop:

```
for x in range(W):

   for y in range(H):

      pixel = getPixel(myPic, x, y)
      r, g, b = getRGB(pixel)
      setRGB(pixel, (255-r,255-g,255-b))
```

Si la imagen es de 1000x1000 píxeles (por ej. W=1000 y H=1000), cada uno de las sentencias en el loop se ejecuta 1 millón de veces. Cada uno de esas sentencias a su vez requiere un promedio de 8-16 operaciones de bajo nivel de computación: los cálculos actuales, más los cálculos para ubicar los píxeles en la memoria, etc. Por lo tanto, la transformación de arriba requeriría arriba de 24-48 millones de operaciones. Como pueden imaginar, llevará unos segundos completar esa tarea.

En la industria de la computación, las velocidades de cómputo se clasifican en base a cálculos de referencia oficiales que calculan la velocidad en términos de la cantidad de operaciones de punto flotante por segundo, o *flops*. Una típica laptop en estos días es capaz de llegar a velocidades de entre ½ a 1 Gflops (Giga flops). La computadora más rápida del mudo puede llegar a velocidades de cómputo de 500 Tflops (Terra flops). Es decir, es alrededor de un millón de veces más rápida (y cuesta muchos millones construirla también). Sin embargo, si se detienen a pensar sobre el programa de juego de ajedrez que mencionamos en el Capítulo 10 que requeriría aproximadamente 10^{65} operaciones antes de realizar una sola jugada, ¡incluso a la computadora más rápida le llevará gazillion de años terminar ese cómputo! Tal problema sería considerado *incomputable*.

Los científicos de la computación han desarrollado un vocabulario elaborado para discutir problemas y clasificarlos como *solubles o*, *computables o incomputables,* basados en si hay modelos conocidos para resolver un problema dado o si los modelos son solucionables, computables, etc. También hay jerarquías de solución de problemas, de simple a complejo; tiempo constante a tiempo polinómico y más

largos; y clases de equivalencias, que implican que el mismo algoritmo puede resolver todos los problemas en una clase, etc. ¡Es realmente sorprendente conceptuar un algoritmo que sea capaz de resolver problemas no relacionados! Por ejemplo, los algoritmos que optimizan rutas de envíos también pueden ser usados para determinar las estructuras de plegamiento de proteínas de las moléculas de ADN. Esto es lo que hace a la ciencia de la computación intelectualmente interesante y emocionante.

Resumen

En este capítulo hemos unido varias ideas fundamentales sobre la computación y la ciencia de la computación. Nuestro viaje se inició en el Capítulo 1 jugando con los robots personales y en el proceso, hemos adquirido un bagaje de conceptos fundamentales de computación. Como pueden ver, la computación es una disciplina de estudio profundamente intelectual que tiene implicancias en todos los aspectos de nuestras vidas. Iniciamos este capítulo señalando que ahora hay más computadoras en un típico campus universitario en Estados Unidos que cantidad de gente. Seguramente no faltará mucho tiempo para que haya más computadoras que personas en todo el planeta. Sin embargo, la idea de *algoritmo* que es esencial en computación, apenas es comprendida por la mayoría de los usuarios a pesar de que está constituida por elementos simples e intuitivos. Muchos científicos de la computación creen que aún estamos sentados en la madrugada de la *era del algoritmo* y de que hay beneficios sociales e intelectuales mucho mayores por ser realizados. Especialmente si más gente toma conciencia de estas ideas.

Revisión Myro

No se introdujeron nuevas características Myro en este capítulo.

Revisión Python

La única característica Python introducida en este capítulo fue la creación de módulos. Cada programa que crean puede ser usado como un módulo de librería desde el cual pueden importar facilidades muy útiles.

Ejercicios

1. Para computar la cantidad de días en un mes, hemos usado lo siguiente:

```
def diasEnMes(m, a):
  '''Retorna la cantidad de dias en el mes m (1-12) en el año, a.'''

  if (m == 4) or (m == 6) or (m == 9) or (m == 11):
    return 30
  elif m == 2:
    if esBisiesto(a):
      return 29
    else:
      return 28
  else:
```

```
    return 31
```

Pueden simplificar más la escritura de la condición en el enunciado-if usando listas:

```
if m in [4, 6, 9, 11]:
   return 30
…
```

Reescriban la función usando la condición de arriba.

2. Definan una función llamada `valido(dia, mes, anio)` para que devuelva true (verdadero) si el `dia`, `mes` y `anio` conforman una fecha válida como se define en este capítulo. Usen la función para reescribir el programa como se muestra a continuación:

```
def main():
  '''Dado un dia del año (ejemplo. 248, 2007),
     lo convierte a la fecha correspodiente (ejemplo. 5/9/2007)'''

  #Entrada: dia, año
   dia, anio = input("Ingrese dia , año: ")


   # Validamos los valores ingresados
   if not valido(dia, mes, anio):
         print "Por favor,ingresa una fecha válida”
         return

   # las entradas son seguras.... seguimos....
   # Comenzamos con mes = 1, para enero
   mes = 1

   while dia > diasEnMes(mes, anio):
     dia = dia - diasEnMes(mes, anio)

     # Siguiente mes
     mes = mes + 1

   # Listo, Salida: dia/mes/año
   print "La fecha es:  %1d/%1d/%4d" % (dia, mes, anio)


main()
```

Reescribir el programa como se muestra arriba para usar la función nueva. Además de desarrollar un algoritmo correcto también es importante escribir programas de manera que sean más leíbles por ustedes.

3. Averigüen cuán rápida es su computadora observando el reloj de velocidad de la CPU. Basados en esto, estimen cuántas operaciones aritméticas será capaz de

realizar en 1 segundo. Escriban un programa para ver cuántas operaciones realmente realiza en un segundo.

4. Realicen una búsqueda Web para “Chazelle age of algorithm” (Chazelle era del algoritmo). Serán recompensados con un link a un ensayo escrito por el Prof. Chazelle. Lean el ensayo y escriban un breve comentario sobre el mismo.

5. ¿Cuál es la computadora más rápida en uso actualmente? ¿Pueden averiguarlo? ¿Cuán rápida es comparada con la computadora que ustedes usan? ¿100 veces? ¿1000 veces? ¿100.000 veces?

6. ¿Qué es/fue el problema “Y2K”?