


```

if (n % 2 == 1) {
    p = p + x;
    n = n - 1;
}

```

y

```

if (n % 2 == 1)
    p = p + x;
    n = n - 1;

```

En el primer caso el decremento de `n` forma parte del cuerpo de la selección condicional, mientras que en el segundo caso está fuera del `if`. En términos prácticos, en el segundo caso el decremento de `n` siempre se ejecuta, mientras que en el primero sólo se ejecuta si `n` es impar. Es importante no olvidar las llaves. También es importante no dejarse engañar por los espacios o la división en líneas; C++ los ignora completamente y sólo se rige por la estructura de agrupación que inducen las llaves. En todo caso, nunca debemos ser nuestros propios enemigos: conviene utilizar la división en líneas y la estructura visual para ayudarnos. Recomendamos los siguientes estilos. Para un condicional con una sola instrucción corta en el cuerpo:

```

if (<condición>) <Instrucción corta>

```

Cuando hay más de una instrucción es necesario poner llaves; la llave para abrir se debería poner en la misma línea que la condición y ser lo último; la llave para cerrar se debería poner sola en una línea, justo debajo de la “i” del `if`:

```

if (<condición>) {
    <Instrucción 1>
    <Instrucción 2>
}

```

Cuando hay una sola instrucción pero es larga, es mejor ponerla en la siguiente línea; para que el código sea lo más inteligible posible y, para que sea fácil aplicarle cambios, se deberían añadir unas llaves:

```

if (<condición>) {
    <Instrucción larga que requiere al menos una línea>
}

```

Una variante de esta instrucción sirve para elegir condicionalmente entre dos instrucciones:

```

if (<condición>) <Cuerpo a ejecutar si el predicado es cierto>
else <Cuerpo a ejecutar si el predicado es falso>

```

Igual que ocurría antes, los cuerpos de uno u otro caso deben ser una sola instrucción. Se deberían aplicar los mismos criterios para decidir cuándo poner llaves. La forma de combinar el `else` con la llave que cierra el bloque de instrucciones de la parte cierta puede ser ésta,

```

if (<condición>) {
    <Instrucción 1>
    <Instrucción 2>
}
else {
    <Instrucción 3>
    <Instrucción 4>
}

```

o, preferiblemente, así:

```
if (<condición>) {
    <Instrucción 1>
    <Instrucción 2>
} else {
    <Instrucción 3>
    <Instrucción 4>
}
```

En algunas ocasiones el código de una bifurcación es largo mientras que el de la otra es corto. Por ejemplo:

```
if (x % 2 == 1) {
    p = p + x;
    n = (n-1) / 2;
} else n = n-1;
```

En estos casos, es mejor rodear ambas ramas con llaves, verbigracia:

```
if (x % 2 == 1) {
    p = p + x;
    n = (n-1) / 2;
} else {
    n = n-1;
}
```

Algunas veces hay que distinguir condicionalmente entre tres o más casos. Lo propio es primero escindir un caso, luego otro y así hasta haberlos completado todos. Por ejemplo, el signo de un número se puede calcular así:

```
if (n < 0) signo = -1;
else if (n > 0) signo = 1;
else signo = 0;
```

En general, esta situación se expresa así:

```
if (<condición 1>) {
    <Instrucciones 1>
} else if (<condición 2>) {
    <Instrucciones 2>
} else if (<condición 3>) {
    <Instrucciones 3>
}
<Otros casos>
} else {
    <Instrucciones para el último caso>
}
```

La estructura anterior resuelve el problema con una búsqueda secuencial, porque recorre uno por uno los distintos casos hasta encontrar qué es lo que está ocurriendo. Hay veces que es lo único que se puede hacer; pero en ocasiones es mejor una búsqueda más estructurada que nos ahorre pasos. Un ejemplo interesante es el cálculo del número de días de un mes en un año no bisiesto. Se puede resolver secuencialmente:

```

if (mes == 1) dias = 31;
else if (mes == 2) dias = 28;
else if (mes == 3) dias = 31;
<Otros casos>

```

O más estructuradamente aprovechando que, hasta el séptimo mes, los meses con número par tienen 30 días (o 28 en el caso de febrero) y los de número impar tienen 31 días, y que, a partir del octavo mes, ocurre lo contrario:

```

if (mes <= 7) {
    if (mes % 2 == 1) dias = 31;
    else if (mes == 2) dias = 28;
    else dias = 30;
} else {
    if (mes % 2 == 1) dias = 30;
    else dias = 31;
}

```

Con frecuencia, una selección entre múltiples casos como la anterior se puede abreviar usando otra estructura, la instrucción `switch`, que estudiaremos en el apartado 2.5.

2.3 Iteración

C++ tiene varias instrucciones de repetición. Empezaremos con la más básica y potente. Es la instrucción `while`; tiene la forma siguiente:

```

while ((condición)) <Cuerpo del while>

```

Sobre la *condición* y el *cuerpo* hay que hacer las mismas observaciones que en el caso de la instrucción de selección condicional. La condición debe ser una expresión con tipo `bool`; siempre debe ir rodeada con unos paréntesis que no forman parte de la condición sino de la sintaxis del `while`. El cuerpo del `while` debe ser una sola instrucción; si queremos ejecutar repetitivamente más de una instrucción, habrá que convertirlas en un bloque rodeándolas con unas llaves.

Cuando la ejecución alcanza una instrucción de esta forma, evalúa la condición; si el resultado es `true`, se ejecuta el cuerpo y se vuelve a empezar; si el resultado es `false`, se termina la ejecución de esta instrucción. Este tipo de instrucciones que ejecutan repetidas veces una misma instrucción reciben el apelativo de *iterativas* o *bucles*; a cada ejecución del cuerpo de un bucle se le suele llamar *iteración* o *vuelta*. Como ejemplo:

```

while (n != 0) {
    if (n % 2 == 1) p = p + x;
    n = n / 2;
}

```

Cada vez que se ejecuta el cuerpo de este bucle, en cada vuelta, el valor de la variable `n` se reduce al menos a la mitad. Como es un dato entero, inevitablemente terminará valiendo 0, y entonces la siguiente comprobación de la condición del bucle resultará falsa, con lo que el bucle terminará.

Es fácil escribir un bucle que no termina nunca:

```

while (true) {
    cout << "No quiero terminar." << endl;
}

```

Sin pretenderlo, a menudo se escriben bucles que no terminan, y no es fácil descubrir el error.

Hay un detalle importante sobre la ejecución de un bucle `while`: la condición no se está comprobando continuamente, sólo se comprueba cuando se llega al bucle por primera vez o cuando se acaba de ejecutar el cuerpo y se ha de decidir si hay que volver a ejecutarlo. Por esto, el siguiente bucle nunca termina:

```
x = 1;
while (x > 0) {
    x = x - 1;
    if (x == 0) x = 1;
}
```

2.4 Otras instrucciones de iteración

2.4.1 Bucle for

El bucle de la sección anterior es tan simple y expresivo que suele resultar un desperdicio para algunas tareas. En esas ocasiones también se pierde tiempo de programación que se ahorraría usando un bucle más sencillo.

A menudo simplemente se quiere repetir una instrucción un número de veces conocido antes de entrar en el bucle. En estas ocasiones también es normal tener una variable que sirve para indicar la *vez* por la que vamos. Muchos lenguajes tienen un bucle especial para expresar esta idea; suele llamarse `for`. Existe un bucle `for` en C++; pero en vez de ser una restricción del bucle `while` resulta ser un bucle más complejo de usar. Afortunadamente, no es necesario pensar cada vez cómo funciona este bucle, sino que basta con ajustarse a unas pocas plantillas que resuelven los problemas más usuales. El bucle `for` tiene la siguiente estructura:

```
for ((Instrucción inicial); <condición>; <Incremento>) <Cuerpo>
```

La mejor forma de explicar su funcionamiento es escribiendo su equivalente con un bucle `while`:

```
{
    <Instrucción inicial>;
    while (<condición>) {
        <Cuerpo>;
        <Incremento>;
    }
}
```

Que toda esta equivalencia esté insertada en un bloque (las llaves más externas) es un detalle importante, porque la instrucción inicial puede contener declaraciones que no tienen efecto más allá del bucle.

Como se ha dicho, este bucle se suele usar para recorrer con una variable índice un rango de valores, a saber:

```
for (int i = <valor inicial>; i < <valor siguiente al final>; i++) {
    <Cuerpo, en donde se puede utilizar la variable i>
}
```

La variable índice no tiene por qué llamarse `i`. En el ejemplo anterior se recorre un rango que contiene el valor inicial pero no el final; basta convertir la desigualdad estricta (`<`) en una no estricta (`<=`) para que también se incluya el valor final.

En el capítulo 6, dedicado a la memoria dinámica, veremos otras formas de utilización de este bucle.

Hay que tener cuidado con la interacción de todas las partes de este bucle. Un ejemplo radical es el siguiente bucle que no termina nunca:

```

for (int i = 1; i <= INT_MAX; i++) {
    if (i % 5 == 0) cout << "Número múltiplo de cinco: " << i;
}

```

2.4.2 Bucle con una ejecución

Ocasionalmente, deseamos escribir un bucle cuyo cuerpo se ha de ejecutar al menos una vez. Esta contingencia se puede expresar con el bucle `do while`, que tiene la siguiente forma:

```

do {
    <Cuerpo>
} while (<condición>);

```

Los elementos reciben los mismos nombres que en el caso del bucle `while`. Es equivalente a ésta:

```

{
    <Cuerpo>
}
while (<condición>) {
    <Cuerpo>
}

```

Es decir, primero se ejecuta el cuerpo y luego se entra en un ciclo de comprobación de la condición y ejecución del cuerpo.

2.4.3 Interrupción de un bucle

En cambio, sí ocurre a menudo que ni el bucle `while` ni el `do while` son adecuados porque hay una parte del bucle que debe ejecutarse al menos una vez pero otra parte puede que no tenga que ejecutarse nunca. Ante esa situación, se suele recurrir a una estructura como la siguiente:

```

<Instrucciones que hay que ejecutar al menos una vez>
while (<condición>) {
    <Instrucciones que puede que no haya que ejecutar nunca>
    <Instrucciones que hay que ejecutar al menos una vez>
}

```

Si el conjunto de instrucciones que hay que ejecutar al menos una vez es sencillo, esta solución es adecuada. Pero si es complejo o grande, es una solución que complica el mantenimiento del código porque una mejora o la corrección de algún error necesita dos cambios. Entonces se suele recurrir a esta otra estructura,

```

bool continuar = true;
while (continuar) {
    <Instrucciones que hay que ejecutar al menos una vez>
    if (<condición>) {
        <Instrucciones que puede que no haya que ejecutar nunca>
    } else {
        continuar = false;
    }
}

```

que resulta particularmente embrollada y oculta la verdadera naturaleza de nuestro problema: tenemos un bucle con el punto de salida en su interior en vez de en un extremo. En C++ hay una instrucción que, al ejecutarse, interrumpe el bucle más inmediato a donde aparezca anidada; es la instrucción `break`. La solución más habitual de nuestro problema tiene la siguiente forma:

```

for (;;) {
    <Instrucciones que hay que ejecutar al menos una vez>
    if (!(condición)) break;
    <Instrucciones que puede que no haya que ejecutar nunca>
}

```

Obsérvese que esta construcción utiliza el bucle `for` de una manera un tanto extraña. Es una herencia de la tradición de C, aunque es igualmente válido y bastante usual utilizar un bucle `while`:

```

while (true) {
    <Instrucciones que hay que ejecutar al menos una vez>
    if (!(condición)) break;
    <Instrucciones que puede que no haya que ejecutar nunca>
}

```

2.5 Selección con tipos ordinales

Hay una instrucción especial para ejecutar código dependiendo del valor de una expresión que tenga un tipo ordinal. (Los tipos ordinales son los enteros y los valores de verdad.) La forma de esta instrucción es la siguiente:

```

switch (<expresión con tipo ordinal>) {
case <valor 1>:
    <Qué hacer para este valor>
    break;
case <valor 2>:
    <Qué hacer para este otro valor>
    break;
<Otros casos>
default:
    <Qué hacer cuando el valor no corresponde a ninguno de los anteriores>
    break;
}

```

La expresión que sigue a la palabra `switch` es la expresión selectora. Los distintos valores que siguen a la palabra `case` deben ser expresiones constantes en tiempo de compilación. Todos estos valores y la expresión selectora deben ser del mismo tipo.

El funcionamiento de esta instrucción es muy sencillo. Primero se evalúa la expresión selectora. Luego se busca si se ha especificado un caso para el valor que hemos obtenido. Si así ocurre, se ejecutarán las instrucciones de ese caso: lo que haya entre los correspondientes `case` y `break`. Si no se ha especificado un caso para ese valor, se ejecutarán las instrucciones para el caso *residual*: lo que haya entre `default` y el siguiente `break`.

El caso residual se puede omitir. Entonces, si el resultado de la expresión selectora no coincide con ningún caso, la instrucción `switch` no hará nada.

En C++ no es obligatorio terminar las instrucciones asociadas a cada caso con un `break`. Un caso que no termine con un `break` se prolonga a los siguientes hasta que aparezca el primer `break`. Es una capacidad de C++ muy discutible con una semántica muy extraña. En este libro no se utilizará; no es en absoluto una pérdida porque en la práctica hay muy pocas ocasiones en donde se pueda aprovechar.

La instrucción de selección `switch` no es fundamental porque siempre se puede conseguir su efecto recurriendo a una serie de selecciones condicionales secuenciadas. El ejemplo anterior quedaría así:

```
{
  int const v = <expresión con tipo ordinal>;
  if (v == <valor 1>) {
    <Qué hacer para este valor>
  } else if (v == <valor 2>) {
    <Qué hacer para este otro valor>
  }
  <Resto de casos (traducidos a ifs anidados)>
  else {
    <Qué hacer cuando el valor de v no responde a ninguno de los anteriores>
  }
}
```


cuesta 100 pesetas (0.6€).” Nos piden que hagamos un programa para facilitar el cálculo del coste de la destrucción. La entrada al programa estará organizada en líneas, cada una de éstas formada por un número de kilómetros cuadrados y una palabra que identifica el mecanismo de exterminio; la entrada acaba cuando en una línea aparece un 0; por ejemplo

```
123 nuclear
232 convencional
1200 químico
242 nuclear
0
```

El resultado del programa ha de ser el coste. Cualquier intento de evitar el desarrollo de este programa por motivos de conciencia será sancionado con un expediente interno y una posible tramitación de despido. Adapta tu programa para que sea independiente de la organización en líneas.

Bibliografía Gastos militares en el mundo en el año 2000: 756 000 000 000 dólares estadounidenses. Gasto del Estado español dentro del apartado de Investigación y Desarrollo: 508 120 000 000 pesetas (3 050 000 000€). El 41.2% de esta cantidad se ha derrochado en investigación armamentística. Lo que hace una cantidad 11 veces mayor que la dedicada a investigación sanitaria, 291 veces el presupuesto de investigación y evaluación educativa, 4 veces superior al programa de educación infantil y primaria, el doble del programa de becas y ayudas a estudiantes, 23 veces el de bibliotecas, 6 veces el de mejora del medio natural.



Podrás encontrar los datos suministrados en las publicaciones del Stockholm International Peace Research Institute (SIPRI, <http://editors.sipri.org>), y en la Cátedra sobre Paz y Derechos Humanos de la UNESCO (<http://www.pangea.org/unescopau/>).

2 4 Lectura de Rayuela



El capítulo 34 de *Rayuela* empieza así:

En setiembre del 80, pocos meses después, del
Y las cosas que lee, una novela, mal escrita,
fallecimiento de mi padre, resolví apartarme de los
para colmo una edición infecta, uno se pregunta
negocios, cediéndolos a otra casa extractora de Jerez
cómo puede interesarle algo así. Pensar que se ha
tan acreditada como la mía; realicé los créditos que
pasado horas enteras devorando esta sopa fría y de-
pude, arrendé los predios, traspasé las bodegas...
sabrída, tantas otras...

Para facilitar su lectura, se ha pensado en dividir el texto en dos, uno con las líneas impares y otro con las pares (aunque esta idea contradice evidentemente la intención de Cortázar).

En este ejercicio se pide desarrollar un programa que lleve a cabo esa tarea.

2 5 ¡Reflejos!



Se propone desarrollar un programa que nos ayude a medir la rapidez de reflejos. La idea es la siguiente: el sujeto de estudio ha de estar atento a la pantalla, en la que, de repente, aparecerá una línea de tamaño creciente; deberá pulsar una tecla tan pronto como pueda, y entonces se detendrá el crecimiento de la línea. Cuanto más corta quede, mayores serán los reflejos que nuestro sujeto habrá demostrado tener. (Véanse las pistas 2.5a y 2.5b.)

El *dictador perpetuus* Julio César utilizaba un código cuando quería mantener en secreto un mensaje. El cifrado consistía en sustituir la primera letra del alfabeto, *A*, por la cuarta, *D*, y así sucesivamente con las demás, es decir, la segunda, *B*, por la quinta, *E*, la tercera, *C*, por la sexta, *F*...

El alfabeto latino que utilizaba Julio César constaba de 21 letras, por tanto la sustitución de letras para cifrar o descifrar mensajes quedaba descrita en la tabla siguiente:

A	B	C	D	E	F	G	H	I	K	L	M	N	O	P	Q	R	S	T	V
D	E	F	G	H	I	K	L	M	N	O	P	Q	R	S	T	V	A	B	C

Para cifrar un mensaje, por ejemplo una frase que el emperador pronunció, “ALEA IACTA EST” (la suerte está echada), basta con buscar la letra escrita en la primera fila de la tabla anterior y escribir la letra que está en la fila de abajo. El procedimiento de descifrado es el inverso; dado un mensaje cifrado, cada letra del mensaje se busca en la fila de abajo y se escribe la letra de la fila de arriba.

A	L	E	A	I	A	C	T	A	E	S	T
D	O	H	D	M	D	F	A	D	H	A	B

Este tipo de cifrado es un *código de sustitución*, ya que cada letra es sustituida por otra. Más concretamente, podemos decir que es un *código de rotación*, ya que a dos letras del alfabeto original consecutivas les corresponden dos letras del alfabeto cifrado consecutivas (si consideramos una representación circular del alfabeto).

Cifrar y descifrar utilizando un código de rotación Escribe un programa que permita cifrar y descifrar utilizando un código de rotación. La clave del código podrá elegirse. (Véanse las pistas 2.6a y 2.6b.)



Figura 2.1: Disco para cifrar de Della Porta con dos alfabetos diferentes

Un poco de historia Podemos encontrar referencias al sistema de código de Julio César en el libro *Vidas de los doce cesares* del historiador romano Cayo Suetonio Tranquilo [Sue92]. Los códigos de rotación no

sólo se han utilizado en tiempos tan remotos; otras personas que los han estudiado son Leon Battista Alberti (1404–1472), arquitecto y filósofo del Renacimiento, y Giovanni Battista Della Porta (1535–1615), científico. En la figura 2.1 puedes ver una imagen del disco para cifrar de Della Porta que aparece en [dP63]. Aun siendo un código de rotación como los que hemos descrito, podemos apreciar cómo el alfabeto del mensaje original y el del mensaje cifrado son distintos; por ejemplo la letra “C” se cifra con un cuadrado negro.

Bibliografía Si te interesa estudiar los métodos criptográficos clásicos y modernos, puedes consultar el libro *Handbook of Applied Cryptography* en una biblioteca [MvV00], o en Internet, en la dirección <http://cacr.math.uwaterloo.ca/hac/>, de forma gratuita.

27 Juego de adivinación

Consideremos el siguiente juego entre los jugadores A (adivino) y P (pensador): P piensa un número comprendido entre 1 y N (digamos 1000, por ejemplo), y A trata de adivinarlo, mediante tanteos sucesivos, hasta dar con él. Por cada tanteo de A, P da una respuesta orientativa de entre las siguientes:

Fallaste. El número pensado es menor que el tuyo.
Fallaste. Mi número es mayor.
Acertaste al fin.

Naturalmente, caben diferentes estrategias para el adivino:

- Una posibilidad, si el adivino no tiene prisa en acabar, consiste en tantear números sucesivamente: primero el 1, después el 2, etc. hasta acertar.
- Otra estrategia, sin duda la más astuta, consiste en tantear el número central de los posibles de modo que, al fallar, se limiten las posibilidades a la mitad (por eso se llama *bipartición* a este modo de tantear).
- También es posible jugar caprichosamente, tanteando un número al azar entre los posibles. Al igual que la anterior, esta estrategia también reduce el intervalo de posibilidades sensiblemente.

Se plantea desarrollar tres programas independientes: uno deberá desempeñar el papel del pensador; otro el del adivino (usando la estrategia de bipartición), y un tercero deberá efectuar la simulación del juego, asumiendo ambos papeles (y donde el adivino efectúa los tanteos a capricho).

Para simular la elección de números al azar, puede consultarse el ejercicio 3.13.

28 Evaluación de polinomios 78

Deseamos desarrollar un programa que evalúe polinomios de una variable real, dados en dos líneas de la entrada estándar: en la primera, se dará el valor de la variable; en la segunda, la lista de coeficientes, no vacía. Se pide escribir dos versiones de este programa ateniéndose a lo que se describe seguidamente.

Órdenes crecientes En esta primera versión, se asume que los coeficientes del polinomio tienen pesos *crecientes*:

$$a_0 + a_1x^1 + \dots + a_{n-1}x^{n-1} + a_nx^n$$

(El grado n del polinomio no se conoce *a priori*.)

Órdenes decrecientes En esta segunda versión, se asume que los coeficientes del polinomio tienen pesos *decrecientes*:

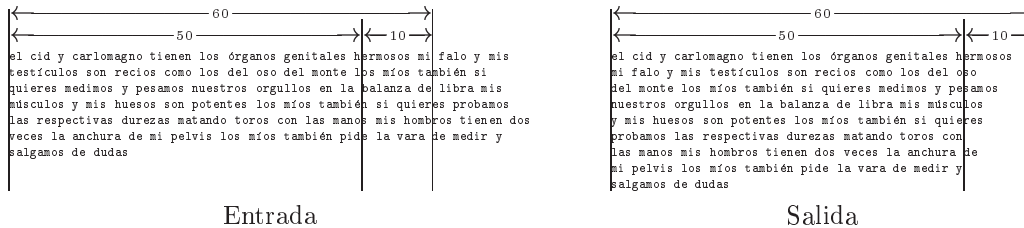
$$a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n$$

(Véase la pista 2.8a.)

2.9 Reorganiza las líneas de un texto

Por azar, hemos encontrado *Oficio de Tinieblas 5* de Camilo José Cela en *formato electrónico*. Pero, cuando nos disponemos a leerlo, encontramos que las líneas de los textos son más grandes que la capacidad horizontal de nuestra pantalla. Con premura barruntamos un método muy sencillo para reorganizar los textos, de forma que cada línea sea lo más grande posible pero que quepa en nuestra pantalla. Razonamos como sigue. Supongamos, tal vez arriesgadamente, que la palabra más larga tiene 15 letras. Como en nuestra pantalla caben líneas con 80 letras, podemos delimitar una región de 15 letras como intervalo de seguridad en el margen derecho. Una palabra que empiece antes de este intervalo puede meterse en él, pero nunca se saldrá por la parte derecha de la pantalla si se cumple nuestra arriesgada suposición. Pero una palabra que empiece dentro del intervalo de seguridad puede salirse de la pantalla; se comenzará una línea nueva siempre que se tenga esta situación.

Ejemplo Como muestra, el comportamiento (sobre la mónada 940) de un programa adaptado para producir líneas de anchura máxima 60 suponiendo que la palabra más larga tenga 10 letras.



2.10 Representación gráfica de funciones

Se trata de representar funciones $\mathbb{R} \rightarrow \mathbb{R}$ en la pantalla de la computadora, de forma aproximada. La función representada es fija para el programa; en nuestro ejemplo, se ha tomado $f(x) = \text{sen}(x)$, aunque puede cambiarse fácilmente. Los datos solicitados por el programa determinan el fragmento del plano XY que se desea representar:

$$[x_{\text{mín}}, x_{\text{máx}}] \times [y_{\text{mín}}, y_{\text{máx}}]$$

Por otra parte, como el tamaño de la pantalla es fijo, la representación se efectúa sobre una cuadrícula de tamaño fijo, $\text{núm}X \times \text{núm}Y$ puntos, que estará representado por sendas constantes del programa:

```
int const numeroX = 16; // Tamaño (abcisas) de la rejilla de dibujo
int const numeroY = 70; // Tamaño (ordenadas) de la rejilla de dibujo
```

Por comodidad, el eje de abcisas será vertical y avanzará descendentemente, y el de ordenadas será horizontal, y avanzará hacia la derecha de la pantalla, como en la figura 2.2, en que se ha elegido la zona $[0.5, 6.5] \times [-0.9, 0.9]$ por ser bastante ilustrativa.

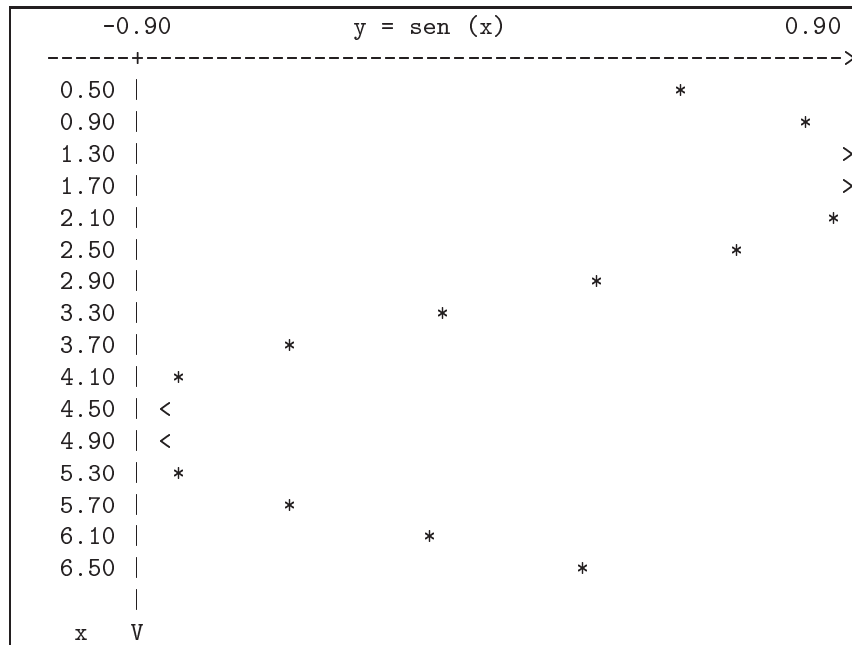


Figura 2.2: Gráfica aproximada de la función seno

Bibliografía En [POAR97] se da una solución en Pascal a este ejercicio.

2.11 Interés cambiante

Con el deseo de participar en su propia explotación y en los beneficios que produce, un programador con contrato basura decide guardar su ahorros en un fondo con referencia a bolsa e interés variable. Tras n años decide retirar el dinero allí depositado. A la hora de comprobar las cuentas del banco descubre que cada año ha recibido un interés distinto, lo que aumenta notablemente el número de cálculos a realizar. Lo solucionó escribiendo un pequeño programa; ¿cómo podría ser este programa?

2.12 De cómo quitar los comentarios de un programa

Para que nadie desconfíe de nuestras habilidades de programación al ver que añadimos abundantes comentarios a nuestros programas, es conveniente disponer de un metaprograma que elimine los comentarios de otro programa.

Recuerda que en nuestro lenguaje de programación favorito, C++, hay dos tipos de comentarios: los que empiezan con “//” y se extienden hasta el final de la línea y los que empiezan con “/*” y acaban con el primer “*/”. Hay que tener mucho cuidado con los caracteres literales, sueltos o agupados en forma de cadena.

2.13 Un dibujo de Escher

Observa el dibujo de M. C. Escher (1898–1972) que aparece en la figura 2.3 titulado *Wentelteeffe*. No pretendemos en este ejercicio imitar la maestría de Escher al dibujar sus *animalillos-cachivache*, como él los llama, pero sí el texto que sirve de marco para la acción de los mismos.

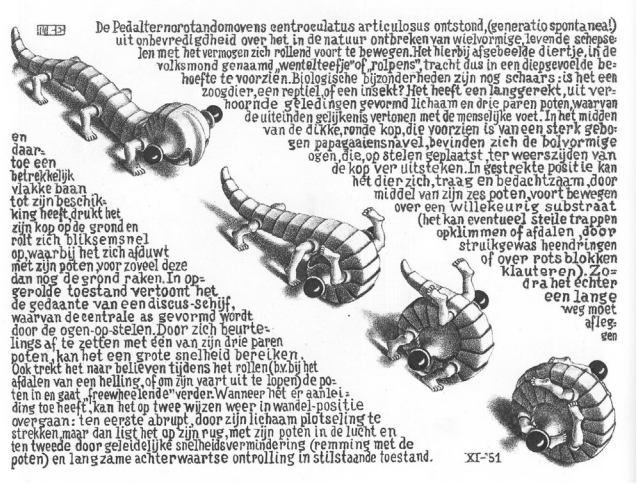


Figura 2.3: *Wentelteeftje* de M.C. Escher

Escribe un programa que lea los datos de un fichero de texto y los escriba en la pantalla siguiendo un patrón como el que muestra el dibujo de Escher. Ten en cuenta que hay varios parámetros que considerar a la hora de definir el patrón.

Ejemplo Aquí podemos ver una ejecución del programa que toma como entrada un fichero que contiene los 123 primeros dígitos del número π :

```

31415926535897932
384626433832795
0288419716939
93751058209
74          944592307
8164        0628620
899862      80348
25342117    067
9821480865  1
328230664709

```

Para este enunciado Consulta la pista 2.13a.

2.14 Espectro natural

El *espectro natural* de una circunferencia de radio r es el conjunto de puntos con coordenadas naturales (n, m) tales que $n^2 + m^2 = r^2$. El método obvio para calcular el espectro necesitaría dos bucles anidados en donde se irían explorando las abscisas y ordenadas de los puntos desde 0 a r . Afortunadamente existe un método más eficiente, que además aprovecha la simetría del problema: si (n, m) está en el espectro también está (m, n) . Definimos

$$B(x, y) = \{(n, m) \mid n^2 + m^2 = r^2 \wedge x \leq n \leq m \leq y\}$$

Obviamente el espectro es $B(0, r) \cup \{(m, n) \mid (n, m) \in B(0, r)\}$ que a su vez se puede calcular usando las siguientes reglas:

$$B(x, y) = \begin{cases} B(x+1, y), & \text{si } x^2 + y^2 < r^2 \\ \{(x, y)\} \cup B(x+1, y-1), & \text{si } x^2 + y^2 = r^2 \\ B(x, y-1), & \text{si } x^2 + y^2 > r^2 \end{cases}$$

Escribe un programa iterativo que calcule el espectro natural de una circunferencia de radio r utilizando exclusivamente las reglas dadas en el párrafo anterior. El programa pedirá el número entero r y a continuación pasará a escribir los puntos del espectro en la pantalla.

Indica el número de pasos que lleva calcular el espectro en relación al radio r .

215 Los cubos de Nicómaco



Considera la siguiente propiedad descubierta por Nicómaco de Gerasa:

*Sumando el primer impar, se obtiene el primer cubo;
sumando los dos siguientes impares, se obtiene el segundo cubo;
sumando los tres siguientes, se obtiene el tercer cubo, etc.*

Comprobémoslo:

$$\begin{aligned} 1^3 &= 1 &= 1 \\ 2^3 &= 3 + 5 &= 8 \\ 3^3 &= 7 + 9 + 11 &= 27 \\ 4^3 &= 13 + 15 + 17 + 19 &= 64 \end{aligned}$$

Desarrolla un programa que escriba los n primeros cubos utilizando esta propiedad. El valor de n puede ser un valor que se pide por el teclado o lo puedes declarar en tu programa como una constante.

Un poco de historia Nicómaco de Gerasa vivió en Palestina entre los siglos I y II de nuestra era. Escribió *Arithmetike eisagoge* (Introducción a la aritmética) que fue el primer tratado en el que la aritmética se consideraba de forma independiente de la geometría. Este libro se utilizó durante más de mil años como texto básico de aritmética, a pesar de que Nicómaco no demostraba sus teoremas, sino que únicamente los ilustraba con ejemplos numéricos.

216 Un bonito triángulo



Anidando bucles y con los dígitos $\{0, \dots, 9\}$ se pueden escribir triángulos tan interesantes como el siguiente:

```

      1
     232
    34543
   4567654
  567898765
 67890109876
7890123210987
890123454321098
90123456765432109
0123456789876543210
    
```

¿Te atreves?

2.17 Varianza



La varianza de n números x_1, \dots, x_n es

$$\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}$$

donde \bar{x} es la media.

Se necesita un programa que lea una secuencia de números y calcule su varianza. Los números se introducirán usando el teclado y se acabarán con un 0, que obviamente no será tenido en cuenta para el cálculo de la varianza. (Véase la pista 2.17a.)

2.18 ISBN de libros



Todo libro editado tiene un número identificativo que consta de diez cifras. A dicho número se le denomina ISBN (del inglés *International Standard Book Number*) y suele aparecer en las primeras páginas junto a otros detalles de la edición. El ISBN se divide en dos partes. La primera, formada por las nueve primeras cifras, identifica el idioma del libro, la editorial y el libro propiamente dicho. Estas primeras nueve cifras son siempre dígitos, es decir, valores entre 0 y 9. La segunda parte es el *dígito de control*, que en realidad puede ser un dígito o la letra X. Si llamamos x_i al dígito que aparece en la posición i -ésima, la décima cifra viene definida por los nueve anteriores según la ecuación siguiente:

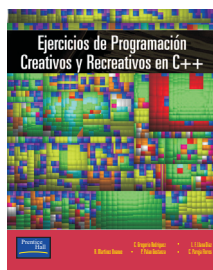
$$x_{10} = \left(\sum_{i=1}^9 i x_i \right) \bmod 11$$

Al dividir por 11 se obtiene un resto entre 0 y 10; si es 10, se pone como dígito de control la letra X.

Como el dígito de control verifica la ecuación anterior, se puede asegurar lo siguiente:

$$\left(\sum_{i=1}^{10} (11 - i) x_i \right) \bmod 11 = 0$$

Ejemplo Consideremos las siguientes fichas bibliográficas de libros. En ellas, el ISBN es un número formado por 10 dígitos; los nueve primeros son determinados por la editorial y el último de ellos es el dígito de control. A veces se utilizan guiones, que desempeñan un papel de meros separadores, por lo que podemos ignorarlos. Vamos a comprobar cómo los nueve primeros dígitos, de izquierda a derecha, determinan el valor del dígito de control.



Ejercicios de programación creativos y recreativos en C++
Varios autores
ISBN 84-205-3211-8
2002

$$\begin{aligned} (1 \cdot 8 + 2 \cdot 4 + 3 \cdot 2 + 4 \cdot 0 + 5 \cdot 5 + 6 \cdot 3 + 7 \cdot 2 + 8 \cdot 1 + 9 \cdot 1) \bmod 11 &= \\ (8 + 8 + 6 + 0 + 25 + 18 + 14 + 8 + 9) \bmod 11 &= \\ 96 \bmod 11 &= \\ 8 & \end{aligned}$$

Si obtenemos un número entre 0 y 9, como en el caso anterior, ése es el dígito de control. Pero si obtenemos 10, como ocurre en el caso siguiente, el dígito de control es la letra X.



El olvido está lleno de memoria
 Mario Benedetti
 168 p. Visor de Poesía
 ISBN 84-7522-332-X
 1995

$$\begin{array}{rcl}
 (1 \cdot 8 + 2 \cdot 4 + 3 \cdot 7 + 4 \cdot 5 + 5 \cdot 2 + 6 \cdot 2 + 7 \cdot 3 + 8 \cdot 3 + 9 \cdot 2) \bmod 11 & = & \\
 (8 + 8 + 21 + 20 + 10 + 12 + 21 + 24 + 18) \bmod 11 & = & \\
 142 \bmod 11 & = & \\
 10 & &
 \end{array}$$

Cálculo del dígito de control Escribe instrucciones que, a partir de las nueve cifras iniciales del ISBN de un libro, calculen el correspondiente dígito de control. (Véase la pista 2.18a.)

ISBN correcto Escribe instrucciones que, dado un número ISBN completo (es decir, de 10 cifras), indiquen si es correcto o no.

Bibliografía El dígito de control del ISBN, al igual que la letra del DNI (véase el ejercicio 1.3), están diseñados para asegurar la transmisión fiable de la información. En [Hil99] puedes aprender más sobre el tema. En el ejercicio 4.16 volveremos a ver, con mucho más detalle, cuestiones relacionadas con la teoría de códigos.

219 Los hexagramas del yin y el yang 78

En la filosofía tradicional china, la dualidad complementaria se expresa mediante los símbolos *yin* y *yang*. En el *I Ching* (El libro de los cambios), estos dos principios se representan con una línea formada por dos trazos (--) y por una línea continua (—) respectivamente.

Agrupándose de seis en seis, estos símbolos generan los *hexagramas* que ves en la figura 2.4. Se pide un programa que dibuje en la pantalla esos sesenta y cuatro hexagramas, sin necesidad de atenerse al caprichoso orden del rey Wen. (Véanse las pistas 2.19a y 2.19b.)

Bibliografía La idea de este enunciado está tomada de [Dew85b].

220 Suma marciana 86

Se ha encontrado en Marte la siguiente operación de sumar, resuelta en una roca:

$$\begin{array}{r}
 \clubsuit \quad \diamond \quad \spadesuit \\
 \quad \quad \clubsuit \quad \heartsuit \\
 \hline
 \diamond \quad \spadesuit \quad \clubsuit
 \end{array}$$

Se desea descifrar el significado (o sea, el valor) de esos símbolos, suponiendo que se ha empleado el sistema de numeración decimal.

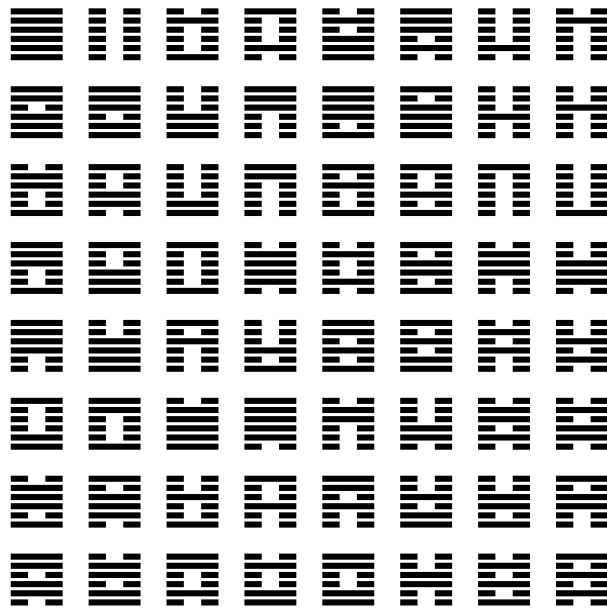


Figura 2.4: Ordenación de los 64 hexagramas según el rey Wen

221 El número y sus representaciones



Me dijo que hacia 1886 había discurrido un sistema original de numeración [...] Su primer estímulo, creo, fue el desagrado de que los treinta y tres orientales requirieran dos signos y tres palabras, en lugar de una sola palabra y un solo signo. Aplicó luego este disparatado principio a los otros números. En lugar de siete mil trece, decía (por ejemplo) *Máximo Pérez*; en lugar de siete mil catorce, *El Ferrocarril*; otros números eran *Luis Melián Lafinur*, *Olimar*, *azufre*, *los bastos*, *la ballena*, *el gas*, *la caldera*, *Napoleón*, *Agustín de Vedia*. En lugar de quinientos decía *nueve*. Cada palabra tenía un signo particular, una especie de marca; las últimas eran muy complicadas... Yo traté de explicarle que esa rapsodia de voces inconexas era precisamente lo contrario de un sistema de numeración. Le dije que decir 365 era decir tres centenas, seis decenas, cinco unidades; análisis que no existe en los “números” *El Negro Timoteo* o *manta de carne*. Funes no me entendió o no quiso entenderme.

Funes el Memorioso, J. L. Borges

Los números surgieron de la necesidad de expresar de forma abstracta una propiedad de un conjunto de elementos: su tamaño; esto es, la cantidad de elementos que posee.

Los sistemas de notación que la humanidad ha utilizado para expresar y manejar las cantidades son muy variados. En lengua castellana tenemos palabras distintas para distintas cantidades: uno, dos, tres, cien, dos mil ciento catorce... (véase el ejercicio 3.20) y lo mismo ocurre en casi todas las lenguas. El sistema de numeración moderno más extendido es el decimal, basado en los dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9, para expresar cualquier cantidad, p. ej. 100, 2 114, 67 431 936. Otros sistemas de numeración, como el que utilizan los ordenadores, se basan únicamente en dos dígitos 0 y 1 para representar cualquier cantidad, p. ej. 0, 1, 10, 11, 1 1001, 1 0010 0100.

Una misma cantidad puede tener diversas notaciones dependiendo del sistema de numeración adoptado. En este ejercicio repasamos algunos de los sistemas más importantes a lo largo de la historia.

Aditiva Las representaciones de números más básicas que se conocen se denominan *aditivas*. La definición de una cantidad se hace mediante la acumulación de signos que representan cantidades básicas. Muchas culturas, como la egipcia, la cretense y la sumeria, entre otras, han utilizado este sistema. Los aztecas, por ejemplo, consideraban los siguientes signos básicos:

Signo	•	□	∣	⊕
Valor	1	20	400	8 000

Cualquier otro número era representado a partir de la acumulación de estos signos. Las cifras se escribían de izquierda a derecha comenzando con los signos de mayor valor. Para expresar la cantidad 16 946 un azteca habría escrito:



Escribe un programa que lea un número azteca y que calcule la cantidad que representa.

Aditiva-sustractiva El sistema de numeración romano es básicamente un sistema aditivo: los números se forman por acumulación. Los signos para las cantidades básicas son los siguientes:

Signo	I	V	X	L	C	D	M
Valor	1	5	10	50	100	500	1 000

A diferencia de otros sistemas aditivos puros, el sistema de numeración romano es también *sustractivo*. Si a la izquierda de una determinada cifra se encuentra otra de inferior valor, esto significa sustracción: VI significa seis, pero IV significa cuatro ya que I, que es el símbolo de la unidad, se encuentra a la izquierda de V, que es el símbolo del número cinco. El número 3 489 expresado en el sistema de numeración romano es:

M M M C D L X X X I X

Escribe un programa que lea un número romano y halle la cantidad que representa.

Multiplicativa Otras representaciones de números, más complejas que la aditiva, son aquellas que se basan no sólo en la suma, sino también en la multiplicación. Entre las culturas que utilizaron sistemas *multiplicativos* están la asiria, la aramea y la etíope. Los arameos consideraban los siguientes símbolos:

Signo	∕	↷	⊃	↶	∕	⊃
Valor	1	10	20	100	1 000	10 000

Los números del 1 al 9 se representaban en forma aditiva a partir de la unidad. Para representar el 7 escribían:



Para números mayores utilizaban una representación multiplicativa: una cantidad de unidades seguida de uno de los símbolos para las potencias de 10 significaba el producto de dichas unidades por la potencia. Considerando que escribían de derecha a izquierda, la representación del número 500 era ésta:



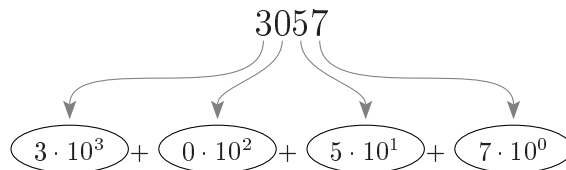
Leyendo de derecha a izquierda sería 5 (por) 100. Con el 20 hacían una excepción, ya que dicho número cuenta con un símbolo propio. El número 43 524 se expresaba así:



Escribe un programa que lea un número arameo e informe de la cantidad que representa.

Posicional Actualmente, los sistemas de numeración más empleados son los posicionales. Con estos sistemas, bastan muy pocos símbolos para poder representar cualquier número. La clave de estos sistemas es que el valor de un símbolo depende de la *posición* que ocupa en el número.

El valor de los símbolos, dependiendo de la posición que ocupen, viene determinado por la base en que se defina el sistema de numeración; habitualmente la base utilizada es 10. El significado de un número puede convertirse fácilmente a notación multiplicativa:



En general, si la base es B , el número cuyas cifras son $c_n c_{n-1} c_{n-2} \cdots c_2 c_1 c_0$ (siendo $0 \leq c_i \leq B - 1$) representa a la cantidad siguiente:

$$c_n \cdot B^n + c_{n-1} \cdot B^{n-1} + c_{n-2} \cdot B^{n-2} + \cdots + c_2 \cdot B^2 + c_1 \cdot B^1 + c_0 \cdot B^0$$

El número 100101, expresado en base dos, representa a la cantidad:

$$\begin{aligned} 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 &= \\ 2^5 + 2^2 + 2^0 &= \\ 32 + 4 + 1 &= \\ 37 & \end{aligned}$$

Observa cómo la noción abstracta de cantidad puede representarse de múltiples formas: treinta y siete en base diez se escribe 37, en base dos 100101 y en base cinco 122.

Escribe un programa que permita leer números en cualquier base e interpretar dicha cantidad. El programa debe recibir primero la base adoptada, luego tendrá que leer números e interpretarlos en la base fijada. El programa también tendrá que verificar que los números introducidos están expresados correctamente en dicha base.

Ejemplo Queremos que el programa pida la base adoptada; supongamos que indicamos la base 7. Entonces, el programa pedirá las cifras, en base 7, para informarnos del valor de dicho número. Si introducimos por ejemplo el 164, el programa debería realizar el cálculo $1 \times 7^2 + 6 \times 7^1 + 4 \times 7^0$ que, expresado en base 10, es 95. En la misma base, si introducimos el número 28, el programa debería informar de que el 8 no es un dígito válido en la base 7.

Para este enunciado Consulta la pista 2.21a.

Bibliografía Puedes hacer un recorrido fascinante a través de los distintos sistemas de numeración en [Gui75]. En [Ifr01] se describe con todo lujo de detalle y con miles de ilustraciones, no sólo la historia de las cifras escritas, sino en general la historia de los números. En este libro podrás descubrir la estrecha relación que existe entre los números y el pensamiento humano.

222 Códigos de Peter Elias

En la vida de su personaje Ezra Winthrop, Jorge Luis Borges pone un papel en el cuento “El Soborno,” una labor de profesor de universidad y una extraña vinculación con sus alumnos:

Me dijeron que en los exámenes prefería no formular una sola pregunta; invitaba al alumno a discurrir sobre tal o cual tema, dejando a su elección el punto preciso.

Peter Elias estuvo interesado en la representación consecutiva de múltiples números de un tamaño variable y desconocido; la utilización de base 2 era también otra de sus necesidades. Por ejemplo, la representación de los números 3 y 10, en base 2 y de forma continuada, genera 111010, donde los dos primeros dígitos son la transformación del 3 y los otros cuatro la transformación del 10; obviamente la recuperación de los datos originales es en todo punto imposible: el 1 y el 26, el 14 y el 2, son pares de números cuya representación consecutiva en base 2 también produce 111010; es más el 58 y la terna 3, 2 y 2 también se representan con el 111010. Peter solucionó esta ambigüedad calculando la secuencia de cada número por separado (digamos w), midiendo su longitud (digamos $k = |w|$) y generando w precedida por k ceros. Así el par 3 y 10 se codificaría

$$\begin{array}{ccccccc} & & 3 & & & & 10 \\ & & \underbrace{\quad} & & \underbrace{\quad} & & \\ \underbrace{00}_{3_l} & \underbrace{11}_{3_c} & & \underbrace{0000}_{10_l} & \underbrace{1010}_{10_c} & & \\ & & & & & & \end{array}$$

donde n_l indica los ceros que se han añadido para codificar la longitud de n mientras que n_c es la codificación en binario del número n .

Esta solución es muy poco económica, pero dándole una vuelta más a la idea se consigue la siguiente forma de codificar el número n ,

$$\underbrace{0 \dots 0}_k uv$$

donde v es la codificación en binario de n ($v = n_c$), u es la codificación en binario de la longitud de v ($u = |v|_c$) y k es la longitud de u ($|u|$). Es posible dar más vueltas a esta idea; incluso una cantidad infinita o indeterminada.

Bibliografía Peter Elias murió el 10 de diciembre de 2001, a los 78 años de edad, de la enfermedad de Creutzfeld-Jakob. Su investigación se centró en la teoría de la información y, en particular, en códigos de corrección de errores y en compresión de datos. Trabajó con Robert Fano (uno de los padres de la teoría de la información) en el MIT. En el artículo [Eli75] expuso, entre otras, las ideas que conforman este ejercicio. Se puede encontrar un análisis más tranquilo (y accesible porque está indexado en <http://citeseer.nj.nec.com>) en [Fen96].

223 Dibujos con asteriscos



Volcán Escribe un programa que dibuje en la pantalla la siguiente figura, compuesta por líneas de 2, 4, 8, 16, 32 y 64 asteriscos respectivamente:

```

          **
         *
        ***
       *****
      *
     *
    *****
   *****
  *****
 *****
*****

```

Mosaico Escribe un programa que dibuje en la pantalla la figura siguiente,

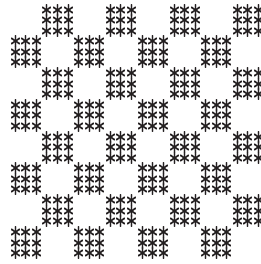
```

 * * * *
 * * * *
 * * * *
 * * * *
 * * * *
 * * * *

```

compuesta por una matriz de 8×8 caracteres, blancos o negros, como en un tablero de ajedrez.

Tablero Escribe un programa que dibuje el tablero siguiente,



como el anterior pero con escaques de lado L , dato del programa.

224 Aproximaciones al número π

Desde que el ser humano se ha preocupado por conocer el entorno y explicar el porqué de las cosas que lo rodean, ha habido personas que han intentado calcular la relación existente entre la longitud de la circunferencia y el radio (o diámetro) que la define.

Largo ha sido el periplo de los matemáticos en torno a este número. En este ejercicio te proponemos utilizar las siguientes fórmulas matemáticas para construir programas que permitan calcular aproximaciones al número π .

- François Viète (1540–1603) en 1593:

$$\frac{2}{\pi} = \sqrt{\frac{1}{2}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{2}}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{2}}}} \cdots$$

- John Wallis (1616–1703) en 1656:

$$\frac{4}{\pi} = \frac{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}$$

- William Brouncker (1620–1684), citado por Wallis en 1656:

$$\frac{4}{\pi} = 1 + \frac{1^2}{2 + \frac{3^2}{2 + \frac{5^2}{2 + \frac{7^2}{\dots}}}}$$

- Gottfried Wilhelm Leibniz (1646–1716) en 1673:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

- Leonhard Euler (1707–1783):

$$\frac{\pi}{4} = \sum_{n \geq 0} \frac{(-1)^n}{(2n+1)2^{2n+1}} + \sum_{n \geq 0} \frac{(-1)^n}{(2n+1)3^{2n+1}}$$

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots$$

$$\frac{\pi^4}{90} = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \dots$$

$$\frac{\pi^2}{8} = 1 + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \dots$$

$$\frac{\pi^3}{32} = 1 - \frac{1}{3^2} + \frac{1}{5^2} - \frac{1}{7^2} + \dots$$

- Borwein en 1987:

$$x_0 = \sqrt{2}$$

$$y_1 = 2^{\frac{1}{4}}$$

$$\pi_0 = 2 + \sqrt{2}$$

$$x_{n+1} = \frac{1}{2} \left(\sqrt{x_n} + \frac{1}{\sqrt{x_n}} \right)$$

$$y_{n+1} = \frac{y_n \sqrt{x_n} + \frac{1}{\sqrt{x_n}}}{y_n + 1}$$

$$\pi_n = \pi_{n-1} \frac{x_n + 1}{y_n + 1}$$

Tiene una convergencia muy rápida: $\pi_n - \pi < 10^{-2^{n+1}}$.

Bibliografía π es sin duda el más famoso de los números, y por eso la bibliografía sobre él es extensísima. Dos libros llenos de curiosidades sobre π son [AH01] y [Bec82]. En [Cha99] se dedica un capítulo a los diversos métodos usados a lo largo de la historia para calcular π . Tan distinguido número no podía faltar tampoco en Internet:

http://www.eecs.umich.edu/~grbarret/pi/pi_links.html

http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Pi_through_the_ages.html

<http://3.14159265358979323846264338327950288419716939937510582097.org>

En las dos primeras direcciones hay cientos de referencias diversas dedicadas a π ; en la tercera puedes encontrar muchas, pero que muchas, cifras decimales de π .

Un poco de historia El primero que utilizó el símbolo π fue William Jones (1675–1749) en 1706. A Euler le gustó este símbolo, lo adoptó y difundió su uso. La fórmula de Leibniz es una particularización de la serie que define el arcotangente de un ángulo; James Gregory (1638–1675) la había descrito con anterioridad, pero no hay ninguna información de que la usase para aproximar el número π .

He aquí una tabla con la cronología del número de cifras decimales de π calculadas:

Número de decimales	Año	Autores	Sistema informático
2 037	1949	G.W. Reitwiesner, ...	ENIAC
10 000	1958	F. Genuys	IBM 704
100 265	1961	D. Shanks y J. Wrench	IBM 7090
1 001 250	1974	J. Guilloud y M. Bouyer	CDC 7600
16 777 206	1983	Y. Kanada, S. Yoshino y Y. Tamura	HITAC M-280H
134 214 700	1987	Y. Kanada, Y. Tamura, Y. Kubo, ...	NEC SX-2
6 442 450 000	1995	D. Takahashi y Y. Kanada	HITAC S-3800/480 (2 procesadores)
51 539 600 000	1997	D. Takahashi y Y. Kanada	HITACHI SR2201 (1024 procesadores)
206 158 430 000	1999	D. Takahashi y Y. Kanada	HITACHI SR8000 (128 procesadores)

Haz un programa que libre a Juanito de la rutina de contar todos los cuadros encerrados por la curva para que pueda escaparse a tirar los rodapiés de los muebles de la cocina contra el quiosco de enfrente. El programa leerá de la entrada los movimientos abreviados con sus iniciales, sin espacios y acabados en el final de la entrada; escribirá el número de cuadrados contados. Puedes suponer que el interior del polígono queda a la izquierda cuando recorremos su contorno con el orden que se dibujó. (Véanse las pistas 2.26a y 2.26b.)

227 *n*-goros



¿Recuerdas los *n*-goros y los recorridos de los mismos que se plantearon en el ejercicio 1.19? Aquí tienes un tablero 4-goro relleno con el número de orden que seguimos para realizar un recorrido:

1	17	13	9	5
6	2	18	14	10
11	7	3	19	15
16	12	8	4	20

Recorrido Haz un programa que, dado un número *n*, escriba las coordenadas de las casillas del tablero *n*-goro según las encontramos con el recorrido anterior.

Ejemplo Para el tablero 4-goro el programa debería escribir esto:

```
1 1  2 2  3 3  4 4  1 5  2 1  3 2  4 3  1 4  2 5
3 1  4 2  1 3  2 4  3 5  4 1  1 2  2 3  3 4  4 5
```

¿Recorrido? Haz un programa que dada una secuencia de coordenadas diga si corresponden al recorrido de algún tablero *n*-goro. La secuencia de coordenadas terminará con un 0. Nuestro programa debe calcular el tamaño del tablero, es decir *n*, a partir de los valores de la secuencia. Además la secuencia puede ser un recorrido parcial, aunque siempre tendrá longitud par (excluyendo al 0 terminador).

Ejemplo La secuencia

```
1 1  2 2  3 3  1 4  2 1  3 2  1 3  2 4  3 1  1 2  2 3  3 4  0
```

es todo el recorrido de un 3-goro; con esta entrada el programa debería responder 3. La secuencia

```
1 1  2 2  1 3  0
```

es un recorrido parcial de un 2-goro; con esta entrada el programa debería responder 2. Pero la secuencia

```
1 1  2 2  3 2  0
```

no es recorrido, ni completo ni parcial, de ningún *n*-goro; con esta entrada el programa debería responder “La secuencia no es un recorrido goro.”

Tablero Haz un programa que dado un número *n* escriba por pantalla un tablero *n*-goro con sus casillas rellenas con el número correspondiente al momento en que se visitan.

Ejemplo Si el número dado es 4, el programa debería producir algo parecido al siguiente tablero:

```

1 17 13 9 5
6 2 18 14 10
11 7 3 19 15
16 12 8 4 20

```

Para este apartado consulta la pista 2.27a.

228 De notación polaca a código ensamblador



Las matemáticas y las computadoras hablan lenguajes distintos. Así, cierto modelo de computadora sólo calculará $(1 + A) \times (2 - B)$ cuando le digamos

```

mov ax, A
inc ax
mov bx, 2
sub bx, B
mul bx

```

El propósito de un compilador es mediar entre estas diferencias de pareceres. El propósito de este ejercicio es aprender algo, muy poco, sobre la construcción de compiladores. En parte porque simplificaremos los extremos en disputa: las expresiones se escribirán en notación postfija y nuestra computadora entenderá unas intrucciones muy simples.

En la notación *postfija* (también llamada *polaca*), las expresiones se escriben con los operandos primero y las operaciones después, en vez de entre ellos. Por ejemplo, en vez de $2 + 3$, se pone $2\ 3\ +$, y en vez de $(1 + A) * (2 - B)$, se pone $1\ A\ +\ 2\ B\ -\ *$. Y así no hacen falta los paréntesis. Las expresiones aritméticas que vamos a considerar en este ejercicio se ajustan a la siguiente descripción:

- Cada operando podrá ser un identificador, consistente en una letra mayúscula.
- Se admitirán las operaciones '+', '-', '*', '/' binarias, y el símbolo '@', que representa el cambio de signo.

Por ejemplo, se admitirán las expresiones siguientes:

$AB*@C+$	que se interpreta como	$-(A \times B) + C$
$ABC*+@$	que se interpreta como	$-(A + B \times C)$

Por otra parte, se considera una máquina de una dirección con un registro acumulador, cuyo lenguaje ensamblador es el siguiente:

Instrucción (y operando)	Efecto producido
LOAD A	Cargar el valor del operando A en el acumulador
STO n	Almacenar el valor del acumulador en la dirección n
ADD n	Sumar al acumulador el valor del registro n
SUB n	Restar al acumulador el valor del registro n
PROD n	Multiplicar el acumulador por el valor del registro n
DIV n	Dividir el acumulador por el valor del registro n
NEG	Cambiar el acumulador de signo

Por ejemplo, la expresión $-(A \times B) + C$ podría traducirse como sigue (escrito por columnas):

LOAD	A	LOAD	\$0	NEG		LOAD	\$0
STO	\$0	PROD	\$1	STO	\$0	ADD	\$1
LOAD	B	STO	\$0	LOAD	C		
STO	\$1	LOAD	\$0	STO	\$1		

(Consideramos que tenemos tantos registros como haga falta.)

Se pide desarrollar un programa que lea de la entrada estándar una expresión aritmética *postfija* y genere las instrucciones elementales correspondientes en código ensamblador, para una máquina de este tipo.

Un poco de historia La *notación polaca* que se trata en este ejercicio fue introducida por Jan Lukasiewicz (1878–1956). Entre otras áreas, Lukasiewicz trabajó en el campo de la lógica matemática, escribió distintos artículos sobre los principios de *no contradicción* y del *tercio excluso*, y desarrolló un cálculo proposicional trivalorado. Muchos de sus trabajos se recogen en sus *Elements of Mathematical Logic* y formaron la base del trabajo de Alfred Tarski (1902–1983).

2.29 Raíces y logaritmos



El teorema de Bolzano permite calcular el cero de una función $f : \mathbb{R} \rightarrow \mathbb{R}$ continua y monótona en un intervalo, de forma que el signo de la función en los extremos tenga signo contrario con un *error* > 0 que se desee. (Para más detalles véase el ejercicio 5.4.)

Además este método nos permite invertir funciones previamente definidas, siempre que verifiquen las condiciones exigidas: calcular $f^{-1}(y)$ es lo mismo que hallar el valor de x tal que $f(x) = y$, o sea, un cero de la función f' , definida como $f'(x) = f(x) - y$.

Raíz cuadrada Aplicar dicho mecanismo para implementar la función \sqrt{x} para $x \geq 1$. ¿Cómo se haría para $0 \leq x < 1$? Hacer un programa que calcule \sqrt{x} para $x \geq 0$.

Logaritmo También se puede aplicar para implementar la función $\log_b(x)$ en base b siendo $b > 1$ y $1 \leq x \leq b$. ¿Cómo se podría implementar la función $\log_b(x)$ para $x > 0$? (Véase la pista 2.29a.)

2.30 Parte entera de logaritmo



Supongamos que a y b son reales $a \geq 1$ y $b > 1$. Realiza un programa para calcular la parte entera del logaritmo en base b de a . El programa sólo podrá usar las operaciones básicas: sumas, restas, multiplicaciones y divisiones. (Véase la pista 2.30a.)

2.31 El principito

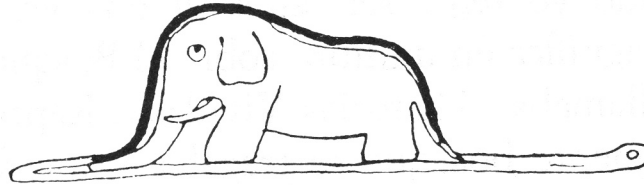


Mi dibujo número uno. Era así:



Mostré mi obra maestra a las personas mayores y les pregunté si mi dibujo les daba miedo. Me contestaron: “¿Por qué nos habría de asustar un sombrero?”

Pero mi dibujo no representaba un sombrero, sino una serpiente boa que digería un elefante. Dibujé entonces el interior de la serpiente boa, a fin de que las personas adultas pudiesen comprender, pues los adultos siempre necesitan explicaciones. Mi dibujo número dos era así:



Las personas mayores me aconsejaron abandonar los dibujos de serpientes boas abiertas o cerradas y que pusiera más interés en la geografía, la historia, el cálculo y la gramática. Y así fue como a la temprana edad de seis años, abandoné una magnífica carrera de pintor, desalentado por el fracaso de mis dibujos números uno y dos. Las personas mayores nunca comprenden por sí solas las cosas, y resulta muy fastidioso para los niños tener que darles continuamente explicaciones.

El principito, Antoine de Saint-Exupéry

Al principito le gustan mucho los números y prefiere las fracciones así:

$$\frac{5687171}{18686419}$$

ya que puede ver y disfrutar de muchas cifras. Sin embargo los mayores prefieren ver cosas más simples y explicadas:

$$\frac{5687171}{18686419} = \frac{7 \cdot 812453}{23 \cdot 812453} = \frac{7}{23}$$

(Véase la pista 2.31a.)

Bibliografía En el siglo III a. C., Euclides escribió los *Elementos* [Euc91, Euc94, Euc96], dividida en trece volúmenes, que ha sido la obra matemática por excelencia durante más de dos mil años. En su libro VII [Euc94], aparece el conocido algoritmo de Euclides para encontrar el máximo común divisor de dos números. En [Cha99] también se relata el texto original de Euclides explicando el algoritmo. Puede encontrarse la implementación de este algoritmo, y de otros similares, en infinidad de libros básicos de programación, como [BB00].

232 Suma que te suma

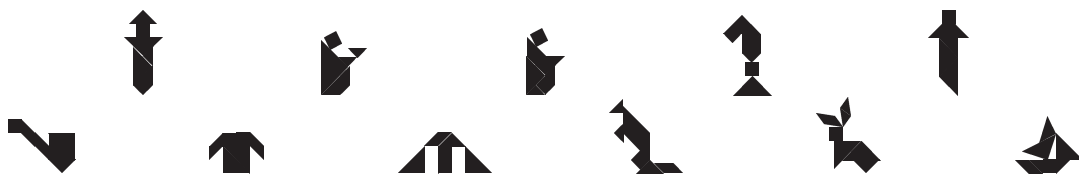


Sabiendo que $(i + 1)^2 = i^2 + 2 * i + 1$ y que $2(i + 1) + 1 = (2i + 1) + 2$, realiza los siguientes programas. En ellos, no se deberán usar en ningún caso multiplicaciones de números. Trata de conseguir la mayor eficiencia posible.

Elevar al cuadrado y al cubo Un programa que calcule el cuadrado de un número no negativo usando únicamente sumas. ¿Cómo se calcularía el cubo de un número?

Comprobación de cuadrados y cubos perfectos Un programa que, utilizando únicamente la suma como operación aritmética, indique si un número no negativo es un cuadrado perfecto. Análogamente si el número es un cubo perfecto.

Raíces cuadrada y cúbica: su parte entera Un programa que calcule la parte entera de la raíz cuadrada. Lo mismo para la raíz cúbica. (Véase la pista 2.32a.)



el código de Julio César sería 3, ya que cualquier letra es sustituida por la que está tres posiciones más allá en el alfabeto.

2.6b. Es muy importante para la corrección del programa fijar convenientemente el alfabeto sobre el que actúa el código. Al menos en un primer intento, no te compliques y elige un alfabeto sencillo.

2.8a. Tenemos que evaluar los datos que recibimos en una sola pasada, ya que sólo podemos leer una vez la entrada estándar. Por tanto, en principio no podemos usar la potencia x^n , puesto que no conocemos aún dicha n . La regla de Horner, debida al matemático inglés William George Horner (1786–1837), permite calcular un polinomio de grado n de forma acumulativa:

$$((\dots((a_0x + a_1)x + a_2)x + \dots)x + a_{n-1})x + a_n.$$

2.13a. Los diversos parámetros que pueden modificar el patrón de escritura son: el espacio en blanco inicial para la primera línea; la anchura total de las líneas; el número de líneas en blanco hasta que aparece la primera línea a la izquierda; y el número de caracteres en los que decrece o aumenta cada línea con respecto a la anterior.

2.17a. La fórmula de la varianza parece indicar que necesitamos hacer dos pasadas sobre los datos: una para calcular primero la media y otra para aplicar esa fórmula. Porque no es razonable pedir a quien use nuestro programa que escriba dos veces los mismo datos, parece que necesitamos almacenarlos internamente. Afortunadamente, un poco de esfuerzo mental y un poco de aritmética básica te permitirán reescribir la fórmula de la varianza, de forma que se pueda calcular de una sola pasada y puedas evitar todas las complejidades que produce el almacenamiento de unos datos arbitrariamente largos.

2.18a. Para poder realizar los cálculos que se indican en el enunciado del ejercicio necesitamos descomponer el número de 9 cifras que se recibe como dato en cada uno de los dígitos que lo componen. En el ejercicio 2.21 puedes encontrar más información.

2.19a. No te obsesiones con la estética. Tres guiones (---) es una buena aproximación al *yang*; al quitar el intermedio (- -) obtenemos el *yin*.

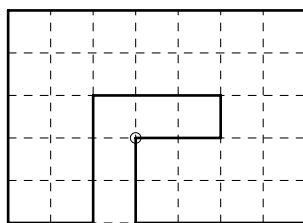
2.19b. La dicotomía entre el *yin* y el *yang* se resuelve con un *bit*. Una vez elegida una correspondencia entre estos principios humanos y los principios informáticos 0 y 1, un hexagrama se reduce a la representación binaria de un número de 6 bits, es decir, un número entre 0 y $2^6 - 1$ (= 63).

2.21a. En los apartados anteriores se pide que el programa lea cantidades que utilizan símbolos que no pueden escribirse con los teclados habituales. Adopta la conversión de símbolos que consideres más adecuada. Por ejemplo podemos proponerte la siguiente:

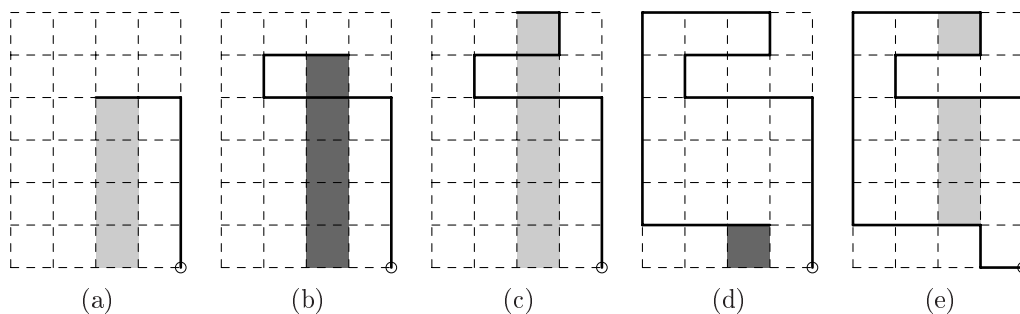
símbolos aztecas				símbolos arameos					
•	□	l	⊕	/	↷	3	↶	∕	↵
.	p	f	@	l	n	h	?	N	*

Con respecto a los sistemas posicionales, si se considera una base $n \leq 10$, lo habitual es utilizar los dígitos entre 0 y $n - 1$ para expresar los números en dicha base. Si utilizamos una base $n > 10$, suelen añadirse las letras necesarias del alfabeto tomadas en orden. Por ejemplo, para base 13, se consideran números que contiene los trece *dígitos* siguientes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C.

2.26a. No se debe suponer que el polígono queda por encima del punto de inicio, pero esta suposición puede facilitar la búsqueda de una primera solución. Ten cuidado de no trivializar la solución; se pueden producir polígonos arbitrariamente complejos; basta con tener la paciencia de enumerar BBDDDDSSSSIIIIIIIBBBBBDDSSSDDBII para conseguir éste:



2.26b. Fíjate en la siguiente figura:



2.27a. No te dejes llevar por la expresión que calculaste para el ejercicio 1.19. Hay una relación muy simple entre un número y el que está a su derecha o inmediatamente debajo. Descúbrela y escribe un programa que calcule el valor de una casilla a partir del valor de la casilla anterior.

2.29a. En ninguno de los casos es necesario suponer conocida la función inversa, es decir, la exponenciación en base b , sino que basta con usar la operación raíz cuadrada.

2.30a. Véase la pista 2.32a.

2.31a. Se trata de simplificar al máximo una fracción $\frac{n}{d}$. Para ello, tenemos que calcular el máximo común divisor del numerador y del denominador, $\text{mcd}(n, d)$, que es el mayor número por el que podemos dividir tanto n como d .

2.32a. La parte entera de un real r es el único entero n que verifica lo siguiente:

$$n \leq r < n + 1.$$

Aplicándolo a este caso tendremos esta equivalencia:

$$\text{raíz} \leq \sqrt{r} < \text{raíz} + 1 \quad \Leftrightarrow \quad \text{raíz}^2 \leq r < (\text{raíz} + 1)^2.$$

La solución a este ejercicio es tan trivial como parece: basta con recorrer el archivo del texto original e ir distribuyendo las líneas, de una en una,

```
una impar y otra par;
una impar y otra par;
...
```

y así mientras haya líneas en el archivo original; esto es:

```
while ((hay líneas en el archivo fuente)) {
    <Leer una línea y pasarla al archivo de las "impares">
    <Leer una línea y pasarla al archivo de las "pares">
}
```

El único detalle que empaña tal simplicidad es la posibilidad de que el número de líneas sea impar, con lo que cada línea par que se vaya a leer debe estar precedida por la correspondiente comprobación:

```
while ((hay líneas en el archivo fuente)) {
    <Leer una línea "impar", y pasarla al archivo de las "impares">
    if ((hay líneas en el archivo fuente)) {
        <Leer una línea "par", y pasarla al archivo de las "pares">
    }
}
```

El bucle anterior se concreta en nuestro lenguaje de programación como sigue:

```
#include <string>
#include <fstream>
int main() {
    ifstream fuente("rayuela.txt");
    ofstream pares("impares.txt");
    ofstream impares("pares.txt");
    while (!fuente.eof()) {
        string linea;
        getline(fuente, linea); pares << linea << endl;
        if (! fuente.eof()) {
            getline(fuente, linea); impares << linea << endl;
        }
    }
}
```

El archivo `rayuela.txt` se supone creado y situado en el directorio de trabajo.

Como podemos ver se ha trazado una cabecera, con los límites de la representación de las ordenadas (en la figura -0.90 y 0.90), el nombre de la función representada ($y = \text{sen } (x)$ en el ejemplo) y una línea horizontal de separación. Debajo, para cada línea, se ha escrito el valor de la abscisa ($0.50, 0.90, \dots, 6.50$) correspondiente, una línea vertical para representar un fragmento de eje, y un asterisco para representar la posición de la función. Si la función se sale fuera de la zona de representación, se ha escrito un símbolo " $<$ " o " $>$ ", según caiga por la izquierda o por la derecha. Así pues, el programa consta de cuatro pasos consecutivos:

⟨Pedir los datos x_{\min} , x_{\max} , y_{\min} , y_{\max} ⟩
 ⟨Trazar la cabecera de la gráfica⟩
 ⟨Trazar las líneas sucesivas⟩
 ⟨Trazar el pie de la gráfica⟩

La lectura de los datos es trivial, así como el trazado del pie de la gráfica. La cabecera debe reflejar el intervalo de ordenadas elegido, y escribir un eje del tamaño $\text{núm}Y$, centrando la función. Sólo el trazado de cada línea requiere alguna atención, y se puede desglosar así:

⟨Hallar la abcisa x_i ⟩
 ⟨Hallar la posición (en la pantalla) de la ordenada $f(x_i)$ ⟩
 ⟨Escribir la línea, comprobando si cae fuera de la zona⟩

Los ajustes entre las coordenadas reales y las de la pantalla se detallan a continuación:

- La abcisa x_i se halla fácilmente:

$$x_i = x_{\min} + i \frac{x_{\max} - x_{\min}}{\text{núm}X}, \text{ para } i \in \{0, \dots, \text{núm}X\}.$$

- Para cada ordenada $y_i = f(x_i)$, su posición será un entero de $\text{pos}Y_i \in \{0, \dots, \text{núm}Y\}$ cuando $y_i \in [y_{\min}, y_{\max}]$, lo que se consigue fácilmente haciendo que $\text{pos}Y_i$ sea el entero más próximo a

$$\text{núm}Y \frac{y_i - y_{\min}}{y_{\max} - y_{\min}}.$$

- Por último, un valor de $\text{pos}Y_i$ negativo o nulo indica que la función se sale por la izquierda del fragmento del plano representado, mientras que un valor mayor que $\text{núm}Y$ significa que se sale por la derecha, lo que se indica en la pantalla mediante el símbolo “<” o “>” al principio o al final de la línea, respectivamente.

Juntando todo esto, tenemos el siguiente programa completo:

```

#include <iostream>
#include <iomanip>
#include <math.h>
int main() {
    int const numeroX = 16; // tamaño (abcisas) de la rejilla de dibujo
    int const numeroY = 70; // tamaño (ordenadas) de la rejilla de dibujo
    // Lectura de los datos (tamaño de la zona de interés):
    float xMin, xMax, // abcisas que limitan la región que se va a dibujar
          yMin, yMax; // ordenadas que limitan la región que se va a dibujar
    cout << "xMín, xMáx: "; cin >> xMin >> xMax;
    cout << "yMín, yMáx: "; cin >> yMin >> yMax;
    // Cabecera: el eje superior (de ordenadas) de referencia:
    cout << setw(9) << setprecision(2) << yMin
         << "          y = sen(x)          ";
    cout << setw(32) << setprecision(2) << yMax << endl;
    cout << "-----+-----"
         << "----->" << endl;
    // Las líneas de la gráfica, con cada uno de los puntos (abcisa, ordenada):
    float const delta = (xMax-xMin)/numeroX; // distancia (fija) entre abcisas

```

```

for (int i = 0; i <= numeroX; i++) {
    float xi = xMin + i * delta; // valor de la abcisa que se va a pintar
    cout << setw(5) << setprecision(2)
         << xi << " | ";
    int yi = (int)rint(((sin(xi) - yMin) / (yMax - yMin)) * numeroY);
    if (yi < 1) {
        cout << "<" << endl;
    } else if (yi > numeroY) {
        cout << setw(numeroY) << ">" << endl;
    } else {
        cout << setw(yi) << "*" << endl;
    }
}
// El pie de la gráfica:
cout << " |" << endl
     << " x  V" << endl;
}

```

215 Los cubos de Nicómaco



Utilizaremos la variable `impar` para que vaya tomando los valores de los números impares. Su valor inicial será `-1` ya que así, al incrementarla sucesivamente en 2 unidades, se irán generando los valores impares. El valor de n lo pediremos por teclado almacenándolo en la variable `n`:

```

int impar = -1;
int n;
cin >> n;

```

Debemos calcular los n primeros cubos:

```

for (int i = 1; i <= n; i++) {
    <Cálculo del cubo i-ésimo a partir de i números impares>
}

```

Sabemos que el primer cubo se calcula sumando el primer impar; el cálculo del cubo i -ésimo, cuando $i > 1$, se realiza sumando los i siguientes impares; necesitaremos, por lo tanto, además de ir generando números impares consecutivos (`impar = impar + 2`), una variable que vaya acumulando su suma (`suma = suma + impar`) y un bucle que controle que estas operaciones se realizan el número adecuado de veces. Añadiendo las instrucciones de salida para que el resultado tenga un formato similar al del enunciado tenemos:

```

impar = impar + 2;
int suma = impar;
cout << i << "^3 = " << impar;
for (int j = 2; j <= i; j++) {
    impar = impar + 2;
    cout << " + " << impar;
    suma = suma + impar;
}
cout << " = " << suma << endl ;

```

El triángulo consta de 10 líneas y cada línea está formada por lo siguiente:

- Un número variable de espacios que comenzando en valor 9 va decreciendo una unidad en cada línea hasta llegar a la última con valor 0.
- Una secuencia de dígitos en orden consecutivo creciente, a excepción de la primera línea formada por un único dígito.
- Una secuencia de dígitos en orden consecutivo decreciente, salvo la primera línea.

La construcción del triángulo se puede plantear así:

```
for (int i = 1; i <= 10; i++) {  
    <Escribir la secuencia de caracteres espacio>  
    <Escribir la secuencia creciente>  
    <Escribir la secuencia decreciente>  
    cout << endl;  
}
```

El refinamiento de la solución anterior puede conducirnos al siguiente programa:

```
char const espacio = ' '  
for (int i = 1; i <= 10; i++) {  
    // Escribir la secuencia de caracteres espacio:  
    for (int j = 1; j <= 10 - i; j++) {  
        cout << espacio;  
    }  
    // Escribir la secuencia creciente:  
    int n = i;  
    for (int j = 1; j <= i; j++) {  
        if (n == 10) n = 0;  
        cout << n;  
        n++;  
    }  
    // Escribir la secuencia decreciente:  
    n = n - 2;  
    for (int j = 1; j <= i-1; j++) {  
        if (n == -1) n = 9;  
        cout << n;  
        n--;  
    }  
    cout << endl;  
}
```

La ecuación usual de la varianza es particularmente desconsiderada en lo que respecta a su implementación. Con esa ecuación difícilmente se calcula la varianza en una sola pasada, que sería lo deseable. Pero si se desarrolla queda:

$$\frac{1}{n-1} \sum_{i=1}^n x_i^2 - \frac{1}{n(n-1)} \left(\sum_{i=1}^n x_i \right)^2.$$

Esta fórmula se implementa fácilmente en una sola pasada:

```
#include <iostream.h>
int main() {
    double accMedia = 0, accMedia2 = 0;
    double x;
    int n = 0;
    cin >> x;
    while (x != 0) {
        accMedia = accMedia + x;
        accMedia2 = accMedia2 + x*x;
        n++;
        cin >> x;
    }
    double varianza = accMedia2 / (n-1) - (accMedia*accMedia) / (n*(n-1));
    cout << "\nVarianza = " << varianza << endl;
}
```

20 Suma marciana



Una posibilidad consiste en producir cada una de las combinaciones posibles,

♣	◇	♠	♥
0	0	0	0
0	0	0	1
...
0	0	0	9
0	0	1	0
...
0	9	9	9
1	0	0	0
...

y examinar cuáles de ellas verifican esa cuenta. Podemos describir ese proceso así:

```
for ((cada valor posible del trébol)) {
    for ((cada valor posible del diamante)) {
        for ((cada valor posible de la pica)) {
            for ((cada valor posible del corazón)) {
                if ((la combinación verifica la cuenta escrita en la roca)) {
                    <Escribir esa combinación>
                }
            }
        }
    }
}
```

Para concretar un poco más el problema, tendremos en cuenta lo siguiente:

- Por estar en el sistema de numeración decimal, los posibles valores de cada uno de los símbolos que intervienen en la cuenta son los valores del cero al nueve.

- Como se ha empleado el sistema de numeración decimal, la grafía ♣♦♠ representa la cantidad $100♣ + 10♦ + ♠$.

Entonces, el algoritmo descrito se traduce a C++ fácilmente así:

```
for (int trebol = 0; trebol < 10; trebol++) {
    for (int diamante = 0; diamante < 10; diamante++) {
        for (int pica = 0; pica < 10; pica++) {
            for (int corazon = 0; corazon < 10; corazon++) {
                int const sumando1 = 100*trebol + 10*diamante + pica;
                int const sumando2 = 10*trebol + corazon;
                int const suma = 100*diamante + 10*pica + trebol;
                if ((sumando1 + sumando2) == suma) {
                    // Generar la solución:
                    cout << "    " << trebol << diamante << pica << endl;
                    cout << " + " << ' ' << trebol << corazon << endl;
                    cout << " ----" << endl;
                    cout << "    " << diamante << pica << trebol << endl;
                    cout << endl;
                }
            }
        }
    }
}
```

Ahora, pueden hacerse dos observaciones que nos permiten limitar un poco los tanteos:

- Los símbolos ♣ y ♦ no pueden ser nulos, ya que están al principio de los números.
- Los cuatro símbolos empleados por los habitantes de Marte pueden suponerse distintos.

Intercalando estas restricciones en el programa anterior, se tiene el siguiente fragmento de programa:

```
for (int trebol = 1; trebol < 10; trebol++) {
    for (int diamante = 1; diamante < 10; diamante++) {
        if (trebol != diamante) {
            for (int pica = 0; pica < 10; pica++) {
                if (trebol != pica && diamante != pica) {
                    for (int corazon = 0; corazon < 10; corazon++) {
                        int const sumando1 = 100*trebol + 10*diamante + pica;
                        int const sumando2 = 10*trebol + corazon;
                        int const suma = 100*diamante + 10*pica + trebol;
                        if (trebol != corazon && diamante != corazon
                            && pica != corazon && (sumando1 + sumando2) == suma) {
                            <Generar esta solución>
                        }
                    }
                }
            }
        }
    }
}
```

Volcán Para trazar el dibujo descrito, usaremos un bucle que escribe una línea en cada vuelta,

```
for (int i = 1; i <= numFilas; i++) {
    <Dibujar la fila i>
}
```

donde el trazado de la fila i -ésima consiste en

```
<Poner los blancos de la izquierda>
<Poner los asteriscos>
<Salto de línea>
```

El fragmento de *<Poner los asteriscos>* es fácil, porque en cada fila hay el doble que en la anterior, de forma que, partiendo de `numAst = 1` antes de la primera fila, basta con duplicar el número de ellos cada vez y ponerlos, uno a uno,

```
int main() {
    int numFilas;
    cout << "Dame el número de filas: " << flush;
    cin >> numFilas;
    int const centro = 4 + (1 << (numFilas-1));
    int numAst = 1; // mitad de los asteriscos de cada fila
    for (int i = 1; i <= numFilas; i++) { // Dibujar la fila i:
        // Poner los blancos de la izquierda:
        repetirCaracter(' ', centro - numAst);
        // Poner los asteriscos:
        numAst = numAst * 2;
        repetirCaracter('*', numAst);
        // Salto de línea:
        cout << endl;
    }
}
```

donde `repetirCaracter` se puede encapsular como un subprograma trivial:

```
void repetirCaracter(char const c, int const n) {
    for (int j = 1; j <= n; j++) {
        cout << c;
    }
}
```

Mosaico La cosa consiste en escribir ocho filas:

```
for (int i = 1; i <= tamanyo; i++) {
    <Dibujar la fila i>
}
```

En cada línea i , los caracteres j (de 1 a `tamanyo`) son blancos o negros dependiendo de la paridad de $i+j$. Cada línea acaba con un salto a la siguiente:

```
for (int j = 1; j <= tamanyo; j++) {
    if ((i+j) % 2 == 0) cout << " ";
}
```

```

    else cout << "*";
}
cout << endl;

```

El programa completo queda como sigue:

```

#include <iostream>
int main() {
    int const tamaño = 8;
    for (int i = 1; i <= tamaño; i++) {
        // Dibujar la fila i:
        for (int j = 1; j <= tamaño; j++) {
            if ((i+j) % 2 == 0) cout << " ";
            else cout << "*";
        }
        cout << endl;
    }
}

```

Tablero Considerando el programa anterior como punto de partida, el cambio que se propone tiene dos efectos: uno consiste en repetir cada fila el número de veces que indique el ancho de los escaques, de forma que ahora, cada hilera de escaques se compone de ancho líneas de asteriscos, logrando así que los escaques tengan la altura deseada; el segundo efecto consiste en que, en cada fila, los blancos y los asteriscos se pintan de ancho en ancho, en vez de uno a uno, y así se logra el efecto ensanchador. En resumen, el programa completo es el siguiente:

```

#include <iostream>
void repetirCaracter(char const c, int const n) {
    for (int i = 1; i <= n; i++) {
        cout << c;
    }
}
int main() {
    int const tamaño = 8;
    int ancho;
    cout << "Ancho del escaque: ";
    cin >> ancho;
    for (int i = 1; i <= tamaño; i++) {
        for (int ii = 1; ii <= ancho; ii++) {
            for (int j = 1; j <= tamaño; j++) {
                if ((i+j) % 2 == 0) {
                    repetirCaracter(' ', ancho);
                } else {
                    repetirCaracter('*', ancho);
                }
            }
        }
        cout << endl;
    }
}
}

```

Para resolver este ejercicio supondremos que el polígono está situado por encima del punto de origen y que su interior queda a la izquierda cuando andamos por su perímetro. Después observaremos que estas dos restricciones se pueden solventar de forma sencilla; en realidad la primera ni siquiera es necesaria. En primer lugar es fácil llevar la cuenta de dónde nos encontramos dentro del área de dibujo; para ello introduciremos dos variables `posX` y `posY`.

Iremos calculando la superficie según nos vayamos moviendo en la cuadrícula. Según se puede observar en la figura de la pista 2.26b, cada vez que nos movemos a la derecha, debemos quitar de la superficie los cuadrados que tengamos por debajo de la línea (están a la derecha de la misma), y los debemos añadir cuando vayamos hacia la izquierda (los que están a la izquierda de la línea). Obsérvese que el trazo del polígono pasa un número par de veces por todas las columnas (puede ser cero), la mitad hacia la derecha y la otra mitad hacia la izquierda compensándose así todas las sumas y restas, para dar finalmente el número de cuadrados contenidos dentro del polígono para cada columna.

En (a) sumamos los 4 cuadrados que hay por debajo de la línea; en (b) restamos los 5 cuadrados por debajo de la línea y llevamos acumulado -1 ; en (c) sumamos 6 y llevamos acumulado 5; y por último en (d) restamos 1 y nos queda finalmente (en esa columna) 4.

Por tanto deberemos hacer un bucle recorriendo el contorno de la figura según la entrada de datos; en cada vuelta del mismo vamos actualizando la posición, si el movimiento es hacia la derecha restamos a la superficie el contenido de la variable `posY` y si es hacia la izquierda lo sumamos.

```
string recorrido;
cout << "Dame el recorrido: ";
cin >> recorrido;
char const baja = 'B';
char const sube = 'S';
char const izquierda = 'I';
char const derecha = 'D';
int posX = 0, posY = 0;
int superficie = 0;
for (unsigned int i = 0; i < recorrido.size(); i++) {
    switch(recorrido[i]) {
        case baja:
            posY--;
            break;
        case sube:
            posY++;
            break;
        case izquierda:
            posX--;
            superficie = superficie + posY;
            break;
        case derecha:
            posX++;
            superficie = superficie - posY;
            break;
    }
}
cout << endl << endl << "Superficie = " << superficie << endl;
```


Lectura de una variable A medida que estas variables se leen (con LOAD), se almacenan (con STO) en registros sucesivos: (LOAD A, STO \$0, LOAD B, STO \$1, ..., LOAD H, STO \$i):

```
cima = cima + 1;
cout << "LOAD " << c << endl;
cout << "STO $" << cima << endl;
```

Lectura y tratamiento de una operación binaria Produce que dicha operación se aplique a los contenidos de los registros penúltimo y último (cima - 1 y cima), y el resultado se guarde en el penúltimo. El último registro queda libre. Por ejemplo, la suma se trata así:

```
cout << "LOAD $" << cima-1 << endl;
cout << "ADD $" << cima << endl;
cima = cima -1;
cout << "STO $" << cima << endl;
```

Lectura y tratamiento de un cambio de signo Esta operación se aplica al contenido del último registro (cima), y el resultado se guarda en el mismo:

```
cout << "LOAD $" << cima << endl;
cout << "NEG" << endl;
cout << "STO $" << cima << endl;
```

Punto final Este carácter determina el fin de la lectura y de la generación de código ensamblador.

Así se completa el programa pedido.

229 Raíces y logaritmos



Raíz cuadrada Tal y como se ha dicho en el enunciado, calcular la raíz cuadrada de un real x equivale a buscar la solución y de la ecuación $y^2 - x = 0$. Como $x \geq 1$, tenemos que $1^2 - x \leq 0$ y $x^2 - x \geq 0$; por tanto podemos establecer como límites iniciales de búsqueda $li = 1$ y $ls = x$, y luego sustituir, repetidamente, los límites izquierdo o derecho por el punto medio según convenga, cuidando de que el cero quede dentro de estos límites:

```
li = 1; ls = x;
while (ls-li >= error) {
    medio = (li+ls)/2;
    valor = medio*medio - x;
    if (valor <= 0) li = medio;
    else ls = medio;
}
raiz = ls;
```

Obsérvese que, en cada vuelta del bucle anterior, se cumple que $li^2 - x \leq 0 \leq ls^2 - x$; o sea, que el cero se mantiene entre li y ls .

El problema al considerar $0 \leq x < 1$ es que $x^2 - x < 0$ y $1^2 - x > 0$, con lo que las asignaciones iniciales de li y ls , en este caso, deben ser $li = x$ y $ls = 1$.

Para que el programa sea válido para todo $x \geq 0$, sólo será necesario que estas asignaciones iniciales se efectúen según el caso en que estemos:

```

if (x < 1) {
    li = x;
    ls = 1;
} else {
    li = 1;
    ls = x;
}

```

Logaritmo La inversa de la función logarítmica es la función exponencial; por tanto tendremos que calcular los ceros de la función $b^y - x$. Sabemos que $1 \leq x \leq b$ por lo que $b^0 - x \leq 0$ y $b^1 - x \geq 0$.

```

li = 0; ls = 1;
while (ls-li >= error) {
    medio = (li+ls)/2;
    valor = <bmedio> - x;
    if (valor <= 0) li = medio;
    else ls = medio;
}
logaritmo = ls;

```

Obsérvese que, en cada vuelta del bucle anterior, el cero se mantiene entre li y ls .

Pero, en la solución de este ejercicio, se ha prohibido usar la función exponencial. Y para solventar ese inconveniente, hemos de darnos cuenta de lo siguiente:

$$b^{\text{medio}} = b^{(li+ls)/2} = \sqrt{b^{li+ls}} = \sqrt{b^{li}b^{ls}}$$

Así pues, introducimos dos variables nuevas bLi y bLs , cuyo valor en cada vuelta del bucle será b^{li} y b^{ls} respectivamente. Puesto que los valores iniciales de li y ls son 0 y 1, los valores iniciales de bLi y bLs serán 1 y b . Tampoco podemos olvidar que, al cambiar el valor de las variables li y ls , se ha de cambiar también el de bLi y bLs :

```

li = 0; ls = 1;
bLi = 1; bLs = b;
while (ls-li >= error) {
    medio = (li+ls)/2;
    bMedio = sqrt(bLi*bLs);
    valor = bMedio - x;
    if (valor <= 0) {
        li = medio;
        bLi = bMedio;
    } else {
        ls = medio;
        bLs = bMedio;
    }
}
logaritmo = ls;

```

Supongamos ahora que $x > b$; realizaremos entonces un bucle dividiendo x por b , de forma que en cada vuelta se verifique $b^{\text{exponente}} \cdot \text{resto} = x$. Obsérvese que, en cada momento, $\log_b(x) = \text{exponente} + \log_b(\text{resto})$. Si iteramos el bucle hasta que $\text{resto} \leq b$, podremos aplicar el algoritmo de arriba a la variable resto , y puesto que $b > 1$ llegará un momento que el algoritmo pare.

Si $0 < x < 1$, el razonamiento es análogo, pero cambiando la división por el producto.

Obsérvese, por último, que no es necesaria una instrucción condicional para tratar ambos casos: es suficiente con un bucle a continuación del otro; si se entra en el primero no se entrará en el segundo y viceversa. En resumen, tenemos lo siguiente:

```

exponente = 0; resto = x;
while (resto > b) {
    exponente++;
    resto = resto/b;
}
while (resto < 1) {
    exponente--;
    resto = resto*b;
}
<logaritmo de resto>;
logaritmo = logaritmo + exponente;

```

232 Suma que te suma



Elevar al cuadrado y al cubo Deseamos calcular el cuadrado de $n \geq 0$ sin usar multiplicaciones, sólo sumando. Nos planteamos proceder de forma incremental, calculando i^2 para $i = 0, 1, 2 \dots$ hasta llegar a n . Una primera versión del programa sería ésta:

```

i = 0; iCuadrado = 0;
while (i < n) {
    iCuadrado = <(i+1) al cuadrado>;
    i++;
}

```

El avance del bucle requiere calcular $(i + 1)^2$ sin efectuar multiplicaciones. Para ello podemos utilizar el valor i^2 que en la vuelta anterior se almacenó en la variable `iCuadrado`. Como $(i + 1)^2 = i^2 + 2i + 1$, la expresión para calcular $(i + 1)^2$ será `iCuadrado + 2*i + 1`, quedando el programa así:

```

i = 0; iCuadrado = 0;
while (i < n) {
    iCuadrado = iCuadrado + 2*i + 1;
    i++;
}

```

Aún tenemos que eliminar la multiplicación $2*i$ en cada vuelta. Podríamos utilizar que $2i = i + i$ y bastaría con sustituir el producto $2*i$ por la suma $i+i$. Pero lo podemos hacer también de forma incremental si introducimos una variable nueva `dosIMasUno` cuyo valor en todo momento sea el de $2i + 1$. Como en cada vuelta la variable `i` aumenta el valor en una unidad, el valor de `dosIMasUno` se actualiza aumentándolo en dos unidades. El valor inicial de `dosIMasUno` ha de ser 1, ya que `i` vale inicialmente cero:

```

i = 0; iCuadrado = 0; dosIMasUno = 1;
while (i < n) {
    iCuadrado = iCuadrado + dosIMasUno;
    dosIMasUno = <2(i+1)+1>;
    i++;
}

```

Para acabar, basta advertir que $2(i + 1) + 1 = 2i + 1 + 2$ con lo que finalmente el programa queda así:

```
i = 0; iCuadrado = 0; dosIMasUno = 1;
while (i < n) {
    iCuadrado = iCuadrado + dosIMasUno;
    dosIMasUno = dosIMasUno + 2;
    i++;
}
```

Calcular el cubo de un número con sumas es análogo al cálculo del cuadrado que acabamos de ver. Consideraremos una variable `iCubo` que en cada momento valga i^3 . En cada vuelta del bucle tendremos que actualizar el valor de `iCubo`. Desarrollando, tenemos lo siguiente:

$$(i + 1)^3 = i^3 + 3i^2 + 3i + 1$$

Necesitamos una variable `tresICuadrado` de forma que en cada vuelta del bucle su valor sea $3i^2 + 3i + 1$. Habrá que actualizar dicha variable en cada vuelta del bucle. Como antes, si desarrollamos

$$3(i + 1)^2 + 3(i + 1) + 1 = 3i^2 + 3i + 1 + 6i + 6$$

llegamos a la asignación:

$$\text{tresICuadrado} = \text{tresICuadrado} + 6 * i + 6$$

Ahora introducimos otra variable `seisIMasSeis` cuyo valor en cada vuelta será $6i + 6$. Para actualizar su valor es suficiente tener en cuenta lo siguiente:

$$6(i + 1) + 6 = 6i + 6 + 6$$

Y juntando todas las piezas, tenemos el programa siguiente:

```
i = 0; iCubo = 0; tresICuadrado = 1; seisIMasSeis = 6;
while (i < n) {
    iCubo = iCubo + tresICuadrado;
    tresICuadrado = tresICuadrado + seisIMasSeis;
    seisIMasSeis = seisIMasSeis + 6;
    i++;
}
```

Comprobación de cuadrados y cubos perfectos En primer lugar es necesario observar que cualquier número no negativo se encuentra entre dos cuadrados perfectos:

$$i^2 \leq n < (i + 1)^2$$

El programa entonces se reducirá a encontrar el primer i tal que $(i + 1)^2 > n$.

Construiremos un bucle como en el apartado anterior: una variable contador i , se incrementará hasta encontrar el primer valor tal que $(i + 1)^2 > n$; calcularemos i^2 de forma incremental guardando su valor en `iCuadrado`. El número será un cuadrado perfecto si $n = iCuadrado$. Teniendo en cuenta que $(i + 1)^2 = iCuadrado + dosIMasUno$, el programa queda así:

```

i = 0; iCuadrado = 0; dosIMasUno = 1;
while ((iCuadrado + dosIMasUno) <= n) {
    iCuadrado = iCuadrado + dosIMasUno;
    dosIMasUno = dosIMasUno + 2;
    i++;
}
esCuadrado = (n == iCuadrado);

```

Si además introducimos una variable `iMasUnoCuadrado` que en todo momento valga $(i + 1)^2$, el programa queda así:

```

i = 0; iCuadrado = 0; dosIMasUno = 1; iMasUnoCuadrado = 1;
while (iMasUnoCuadrado <= n) {
    iCuadrado = iMasUnoCuadrado;
    dosIMasUno = dosIMasUno + 2;
    iMasUnoCuadrado = iCuadrado + dosIMasUno;
    i++;
}
esCuadrado = (n == iCuadrado);

```

Para averiguar si un número es un cubo perfecto se procedería de forma similar.

Raíces cuadrada y cúbica: su parte entera En primer lugar es necesario precisar qué entendemos por parte entera de un real: el mayor entero menor o igual que el real dado. Así pues, la parte entera de la raíz cuadrada de un entero no negativo n es el único entero $raiz$ tal que

$$raiz \leq \sqrt{n} < raiz + 1$$

Si elevamos todo al cuadrado, y sabiendo que todos son números no negativos, tenemos lo siguiente:

$$raiz^2 \leq n < (raiz + 1)^2$$

Así pues, el programa será el mismo que el del apartado anterior y la solución será el último valor de i :

```

i = 0; iCuadrado = 0; dosIMasUno = 1; iMasUnoCuadrado = 1;
while (iMasUnoCuadrado <= n) {
    iCuadrado = iMasUnoCuadrado;
    dosIMasUno = dosIMasUno + 2;
    iMasUnoCuadrado = iCuadrado + dosIMasUno;
    i++;
}
raiz = i;

```

La raíz cúbica se calcularía de forma similar.

