*A nuestras personas queridas*

... pleasure has probably been the main goal all along. But I hesitate to admit it, because computer scientists want to maintain their image as hard-working individuals who deserve high salaries. Sooner or later society will realise that certain kinds of hard work are in fact admirable even though they are more fun than just about anything else.

*Donald Edwin Knuth*

En aquel Imperio, el Arte de la Cartografía logró tal Perfección que el mapa de una sola Provincia ocupaba toda una Ciudad, y el mapa del Imperio toda una Provincia. Con el tiempo, esos Mapas Desmesurados no satisfacieron y los Colegios de Cartógrafos levantaron un Mapa del Imperio, que tenía el tamaño del Imperio y coincidía puntualmente con él. Menos Adictas al Estudio de la Cartografía, las Generaciones Siguientes entendieron que ese dilatado Mapa era Inútil y no sin Impiedad lo entregaron a las Inclemencias del Sol y de los Inviernos. En los desiertos del Oeste perduran despedazadas Ruinas del Mapa, habitadas por Animales y por Mendigos; en todo el País no hay otra reliquia de las Disciplinas Geográficas.

*Jorge Luis Borges*
*Museo*

Programming is a form of art. Not only is elegance and beauty possible in programming computers, these are at the core of a good programmer's value system. Computers present to mankind the first opportunity to do what religions all over the world have failed to present: the ability to receive unambiguous answers to incantations and prayers —computers are the man-made gods who *listen*. I am, of course, talking metaphorically about instructing computers to create what we want to exist.

Like other information should be available to those who want to learn and understand, program source code is the only means for programmers to learn the art from their predecessors. It would be unthinkable for playwrights not to allow other playwrights to read their plays, only be present at theater performances where they would be barred even from taking notes. Likewise, any good author is well read, as every child who learns to write will read hundreds of times more than it writes. Programmers, however, are expected to invent the alphabet and learn to write long novels all on their own. Programming cannot grow and learn unless the next generation of programmers have access to the knowledge and information gathered by other programmers before them.

*Erik Naggum*
*Ideas and Principles. Programming*
`http://www.naggum.no/erik/programming.html`
6 de Febrero de 1996

***Qué es*** Estas páginas recogen una colección de problemas con los que practicar mientras se aprende a programar. Aproximadamente la mitad de ellas se dedican a solucionar parte de los problemas propuestos. Hemos intentado que los ejercicios sean atractivos y variados; las soluciones que damos son las que nos han parecido más adecuadas. Este libro pretende ser un símil de esas clases de retórica que muchos ya no hemos tenido: problemas interesantes sobre los que laborar y buenas soluciones de las que aprender.

Hemos evitado un libro técnico y distante. Lejos de ser autocontenida y cerrada, esta obra trata de ser amena, plural y abierta, pretende motivar, inspirar y alentar a sus lectores para que profundicen en temas variados que pueden contribuir a su formación técnica, pero también, a su formación personal.

En los ejercicios, se pueden encontrar citas de obras literarias que nos han conmovido, referencias bibliográficas a temas que nos interesan y enlaces a páginas de Internet con las que hemos disfrutado. Todo ello con el objetivo de mostrar puertas abiertas y posibles caminos a explorar: los juegos, las matemáticas, la literatura, los fractales, la música, la criptografía, la filosofía, la historia...

***Pasado*** Esta obra no fue el objetivo inicial por el que empezamos a coleccionar los ejercicios que aquí se pueden encontrar. Nuestra intención prístina fue (y sigue siendo) preparar un sitio en Internet (`http://aljibe.sip.ucm.es`) en donde albergar todo tipo de material educativo referente a la programación. Pronto nos especializamos en la recolección de ejercicios; estas páginas son una selección de lo más interesante de todo ese material.

***El medio*** Un libro de programación, al menos uno tan práctico como éste, necesita un lenguaje formal con el que programar. La práctica docente y la empresarial han introducido multitud de lenguajes que resuelven con muy diverso grado de acierto el objetivo de ser un formalismo con el que expresar algoritmos. Porque hay más de un buen lenguaje, la discusión sobre cuál sería el mejor para este libro se prolongó durante algunas semanas. El elegido fue un subconjunto de C++, que en nuestra intimidad gustamos llamar C+-. Este nombre, que tiene algo de juego, trata de expresar que el resultado es C con algunas cosas de C++ pero sin otras del propio C. Para quienes conozcan estos lenguajes bastará decir que C+- es C++ sin orientación a objetos y evitando el paso de punteros a funciones cuando el objetivo es un paso por referencia.

Tal vez algún día vea la luz una reedición de este libro con alguno de esos otros buenos lenguajes que tuvimos que desechar: Pascal, Ada, Java o pseudocódigo. Mientras, quienes estén aprendiendo a programar con otros lenguajes, pueden sacar buen provecho de estas páginas porque la mayor parte de los ejercicios no dependen, o lo hacen escasamente, de C++. Además, en `http://aljibe.sip.ucm.es` hay variantes de muchos de estos ejercicios para otros lenguajes, con solución incluida.

***Qué no es*** Por el contrario, estas páginas no son muchas otras cosas. No son el lugar donde aprender la sintaxis, ni la semántica, ni las particularidades de ningún lenguaje de programación. Con ellas solas no se puede aprender a programar; se necesita otro libro, unas clases donde se cuenten con rigor y profusión los conceptos teóricos y la metodología de la programación.

***Uso*** La división por capítulos sigue la organización tradicional de un curso de introducción a la programación; pasa por los mismos temas y sigue el mismo orden. Cada capítulo tiene cuatro partes, dedicadas, en este orden, a resumir los conceptos de C++ que se tratan en el capítulo, plantear enunciados, recolectar pistas y solucionar parte de los problemas. Enunciados, pistas y soluciones comparten numeración. Como cabría esperar, los números de los enunciados se asignan consecutivamente. Pistas y soluciones heredan el orden y número de su enunciado, pero, como no todos los ejercicios tienen pistas ni están solucionados, aparecen huecos en estas otras partes. Para distinguir entre las distintas pistas de un ejercicio, acompaña al número una letra minúscula.


El tangram es un juego de mesa originario de China. Consta de siete piezas con formas geométricas muy básicas; las puedes ver en el comienzo de este párrafo: dos triángulos rectángulos grandes, dos pequeños, uno mediano, un cuadrado y un paralelogramo. Los triángulos pequeños son los ladrillos básicos; juntando dos se obtiene el cuadrado, el paralelogramo o el triángulo mediano, dependiendo de cómo se haga, y juntando cuatro se obtienen los triángulos grandes.


Consiste el juego en la recreación de una figura haciendo uso de las siete piezas. Al principio del párrafo anterior se muestra la disposición que genera un cuadrado; en éste, la que genera un cisne; en el siguiente ya no se revela el secreto de cómo construir una liebre, aunque debería ser evidente.


El esfuerzo intelectual del tangram depende de qué figura se tenga que conseguir; el cuadrado es una de las más difíciles. El esfuerzo también depende de si nos ofrecen alguna pieza ya colocada: así, los problemas de completar las figuras  ,  ,  y  para obtener un cuadrado son progresivamente más complejos. Cada ejercicio se adorna con uno de estos iconos para indicar su dificultad. Las imágenes ,  ,  y  marcan dónde hay que colocar alguna de las piezas que faltan; las usaremos para señalar que un ejercicio tiene pistas. Finalmente, con la figura  indicaremos que el ejercicio está resuelto. Los iconos para pistas y soluciones siempre irán acompañados del número de la página a la que recurrir.

***Código*** Forzados por la exposición, muchas soluciones presentan un código fragmentado en piezas. Siempre que su reorganización y unión en forma de programa completo sea obvia, evitaremos el gasto de ofrecer este resultado en papel.

Muchas veces, en puntos intermedios de una explicación, ofreceremos código que carece de algún detalle interno. En su lugar aparecerá una frase explicativa; para resaltar que es inadmisible como C++ correcto, se presentará así:

⟨*Una frase explicativa del código ausente*⟩

***Prolongación*** En `http://aljibe.sip.ucm.es` puedes encontrar, además de muchos otros problemas, soluciones en otros lenguajes de programación y diversos recursos que no tiene sentido poner en papel, como códigos completos, datos de pruebas, etc.

Pero la iniciativa de recopilar problemas de programación y material relacionado no es nueva ni exclusivamente nuestra. Consulta los ejercicios de la Olimpiada Informática Española (`http://www.aula-ee.com/oie/probs.htm`) o los del concurso internacional de la ACM de problemas de programación (*The ACM International Collegiate Programming Contest*, `http://icpc.baylor.edu/icpc/`). Estos últimos se han recopilado en `http://acm.uva.es/problemset/` y complementado con una herramienta que comprueba soluciones.

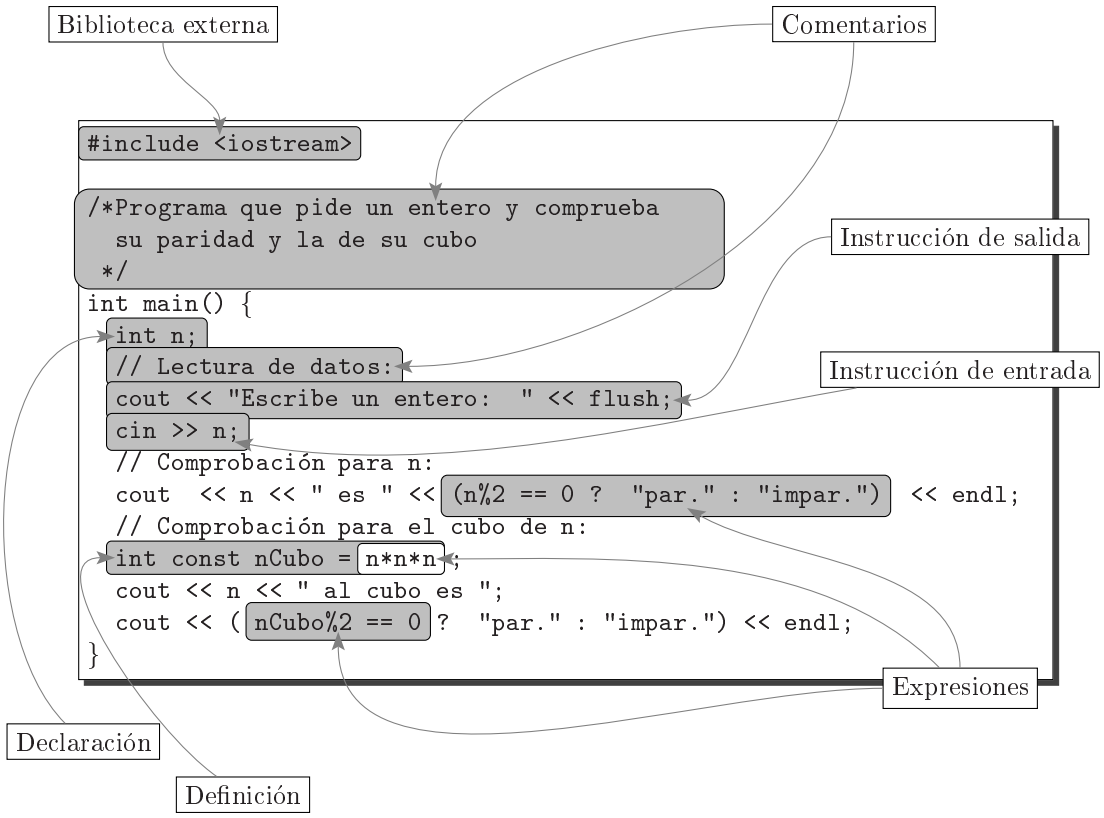
Finalmente, no dudes en consultar esa enciclopedia, inmensa y cambiante, que Internet y el *software libre* redactan día a día para quienes disfrutan de la práctica de la programación: `http://www.gnu.org`, `http://www.opensource.org`, `http://www.sourceforge.org`, etc.

***Agradecimientos*** Como puede verse, en la portada del libro figuran cinco autores; y si bien esto es verdad, no es toda la verdad: porque no podemos ignorar la valiosa contribución de Euclides, la del inventor del ajedrez, la de Julio Cortázar, el rey Wen, la tortuga de Lou Shu, la de los niños de Agra y Allahabad, la de Guido d'Arezzo, Martin Gardner, Borges, la de los campesinos rusos, los antiguos egipcios, y muchos otros cuyos nombres no podemos relacionar en su totalidad por obvias razones de espacio. Entre esos personajes ilustres, es imposible omitir a Óscar Martín Sánchez, por sus revisiones, tan profundas como acertadas, a Carmen Antón Luaces y Carmen Muñoz Serrano, por su ayuda con la bibliografía, y a David Gregorio Rodríguez, por su ayuda en la edición de las fotografías. También ha jugado un papel importante esa chispa que habita, agazapada, en el cerebro de muchos de nuestros alumnos, y que de vez en cuando salta alegremente de una neurona a otra, con regocijo de todos.

Y a ti precisamente, que estás leyendo estas líneas en este momento, y tal vez te dispones a recorrer las páginas siguientes, te dedicamos expresamente el libro, y te pedimos indulgencia con los defectos que, sin duda, tiene. De ellos, sí somos responsables los autores de la portada y nadie más.

# 1 Capítulo UNO

## Introducción a la programación



# En este capítulo

## 1.1 Contexto

El convenio dicta que los programas en C se guardan en ficheros con extensión '`.c`'. No hay un acuerdo tan rígido para los programas en C++. Distintos equipos introdujeron extensiones diferentes casi simultáneamente. Es costumbre que un programa en C++ se guarde en un fichero con extensión '`.cpp`', pero también se utilizan las extensiones '`.cxx`' o '`.cc`' o '`.C`'.

Los desarrolladores de C buscaban un lenguaje mínimo pero muy expresivo, y lo encontraron. Parte de la minimalidad se consigue relegando a bibliotecas externas lo que otros lenguajes dejan en su núcleo. Por esto, y porque ha sido y sigue siendo un lenguaje muy utilizado, C tiene una biblioteca estándar de funciones enorme. Con C++ esta biblioteca ha crecido más aún. En estos resúmenes describimos tanto el lenguaje como la parte de la biblioteca más básica e interesante o que resulta necesaria para resolver los problemas que planteamos.

ANSI C y C++ exigen que todo se haya *declarado* antes de ser nombrado. Este requisito tiene una clara implicación cuando queremos utilizar funciones que están en una biblioteca: tenemos que declarar en nuestros programas todas aquellas funciones que vayamos a usar. Para que no haga falta incorporar manualmente declaraciones en los programas que escribimos, casi todas las bibliotecas proporcionan unos ficheros llamados *header files* (*ficheros de cabeceras*). En C, es norma que los ficheros de cabeceras tengan la extensión '`.h`'; en C++, se utilizan las extensiones '`.hpp`', '`.hxx`' o '`.hh`'; algunas bibliotecas para C++ no ponen extensión a sus ficheros de cabeceras. Para que el compilador de C o C++ consulte un cierto fichero de cabeceras `biblioteca.h`, hay que añadir una línea semejante a ésta:

```
#include <biblioteca.h>
```

## 1.2 Comentarios

Está bastante asentada la idea falaz de que los programas son para que los lean las máquinas. Generalmente, quienes pasan más tiempo leyendo programas son los humanos que los crean, los modifican, extienden, etc. Un *comentario* es un texto libre, en el idioma que se quiera, con anotaciones importantes sobre partes de un programa que ayudan a su comprensión. Para que C++ lo ignore hay que precederlo con `/*` y terminarlo con `*/`. El símbolo `//` también inicia un comentario que C++ ignorará; acaba con el renglón:

```
/* Un comentario de párrafo. Lo normal es que se ex-
   tienda a lo largo de varias líneas. La marca de cierre
   se suele colocar bajo la de apertura.
 */
// Un comentario que acaba al final de esta línea.
```

## 1.3 Tipos básicos

El material de construcción más elemental en un lenguaje de programación son sus tipos básicos predefinidos. Llamaremos tipos básicos a los enteros, caracteres, reales y booleanos que se describen a continuación. También en esta sección, debido a su utilidad y sencillez, describiremos las cadenas de caracteres aunque no son un tipo básico.

### 1.3.1 Enteros

C++ tiene cuatro tipos distintos con los que representar valores numéricos enteros. Esta abundancia se debe, por una parte, a que los caracteres se representan con un tipo numérico (un pequeño desliz conceptual) y, por otra, a la necesidad práctica de controlar el gasto de memoria cuando se conoce el rango de valores. Por orden creciente en cuanto al subconjunto de los enteros que son capaces de almacenar, los nombres de estos tipos son `char`, `short`, `int` y `long`. Lo corriente es utilizar 8, 16, 32 y 64 bits, respectivamente, para guardar un valor de estos tipos.

C++ distingue entre enteros con signo (positivos y negativos) y enteros sin signo (naturales). Esto, de nuevo, es un detalle que sirve esencialmente para controlar más minuciosamente el consumo de memoria. Hay dos prefijos, `signed` y `unsigned`, que se pueden anteponer a cualquiera de los cuatro tipos numéricos para especificar si queremos una variante con o sin signo. Para los tipos `short`, `int` y `long`, si se omite este prefijo, se supone que el entero tiene signo. En cambio, cuando no se prefija el tipo `char`, cada compilador puede asumir lo que le venga en gana.

C++ supone que la máquina subyacente es binaria, como ocurre siempre hoy en día. (Consúltese el ejercicio 2.21 si no concibes números que no estén escritos con los diez dígitos usuales.) Los enteros con signo se representan en *complemento a 2*.

| Tipo | Bits | Mínimo | Máximo |
|---|---|---|---|
| `signed char` | 8 | (−128) `SCHAR_MIN` | `SCHAR_MAX` (127) |
| `unsigned char` | 8 | (0) | `UCHAR_MAX` (255U) |
| `short` | 16 | (−32768) `SHRT_MIN` | `SHRT_MAX` (32767) |
| `unsigned short` | 16 | (0) | `USHRT_MAX` (65536U) |
| `int` | 32 | (−2147483648) `INT_MIN` | `INT_MAX` (2147483647) |
| `unsigned int` | 32 | (0) | `UINT_MAX` (4294967295U) |
| `long` | 64 | (−9223372036854775808L) `LONG_MIN` | `LONG_MAX` (9223372036854775807L) |
| `unsigned long` | 64 | (0) | `ULONG_MAX` (18446744073709551615UL) |

Tabla 1.1: Los rangos de los distintos tipos enteros

Los valores enteros explícitos (literales enteros) se pueden escribir en tres bases distintas: 8, 10 y 16. Cualquier número *normal* que no empiece por 0 será un literal entero en base 10. Si un entero empieza con un 0, es un literal en base 8: entre sus dígitos no podrá haber ochos ni nueves. Los literales en base 16 empiezan con `0x`; las letras de la "A" a la "F" (o de la "a" a la "f," si se prefiere en minúsculas) sirven para denotar a los dígitos del 10 al 15. Por ejemplo, el número 16 se puede escribir como `020`, `16` o `0x10`; estos literales son respectivamente números en base 8, 10 y 16. Porque internamente los números se representan en base 2, las notaciones en base 8 o base 16 son muy cómodas para ciertas tareas.

Un literal entero tiene el rango de un `int`. Si se desea expresar un literal sin signo, hay que añadir un sufijo "U" inmediatamente al final del número. Si se desea expresar un literal con rango de `long`, es necesario añadir el sufijo "L" inmediatamente al final del número.

En la tabla 1.1 pueden consultarse los rangos de todos estos tipos numéricos y el nombre de las constantes que delimitan sus rangos. Estas constantes están definidas en el fichero `limits.h` y por tanto hay que añadir

```
#include <limits.h>
```

al principio de todo programa que quiera utilizarlas. (El ejercicio 5.19 te guiará en la construcción de unos enteros sin límites.)

### 1.3.2 Caracteres

Algunas veces el tipo `char` se utiliza como un entero que ocupa muy poco espacio. Pero su cometido principal, de ahí su nombre, es contener caracteres. La correspondecia entre un número y un símbolo

visible viene dada por una *tabla de caracteres;* la más frecuente hoy en día es la tabla ASCII (*an Standard Code for Information Interchange* que viene a significar código *americano normado* para el intercambio de información). Esta tabla dicta, por ejemplo, que el número 82 es la R. Por conveniencia y para no tener que atarnos a una cierta tabla, existe una forma para hacer referencia a un carácter: rodearlo con comillas simples (`'`). De esta forma, el literal `'R'` *es* el carácter `R`; o de otra forma, el valor entero de la letra `R` en la tabla de caracteres que use nuestro sistema.

Dentro de las comillas, el símbolo "\" no se representa a sí mismo, sino que sirve para poder escribir caracteres conflictivos según rige la siguiente tabla:

| | |
|---|---|
| `\\` | El carácter "\" |
| `\'` | La comilla simple |
| `\"` | La comilla doble |
| `\n` | Retorno de carro (fin de línea) |
| `\r` | Avance de línea |
| `\t` | Tabulador |
| `\b` | Carácter de borrado |

### 1.3.3 Reales

De nuevo, para poder jugar compensando el gasto de memoria con rango y precisión en los cálculos, hay dos tipos para representar números reales: `float` y `double`. El tipo `float` utiliza 4 octetos mientras que `double` utiliza 8. Quien esté interesado en *Lo que todo informático debería saber sobre la aritmética de coma flotante*, puede hallarlo en [Gol91].

Los literales reales utilizan la notación científica usual en los lenguajes de programación. La parte entera se separa de la parte fraccionaria con un punto ("`.`"). Opcionalmente, si se añade una "`e`" (o "`E`") seguida de un entero $n$, el literal estará multiplicado por $10^n$. Por ejemplo, la velocidad de la luz en km/s es `3E5`, y en km/h es `1.08E9`.

Un literal real tiene precisión de `double`. Para conseguir una precisión de `float` hay que añadir inmediatamente al final del número el sufijo "`f`" (o "`F`").

Las limitaciones de estos tipos a la hora de representar reales son bastante notables. No podría ser de otra forma, porque números reales hay demasiados y casi todos (por ejemplo, $\pi$; consulta el ejercicio 2.24) requieren una cantidad infinita de memoria para representar todos sus decimales. Para poder compaginar una buena precisión (muchos decimales), con un rango de valores amplio (un máximo y un mínimo muy grandes) y un gasto muy reducido de memoria, se hace variar el detalle junto con la escala, de forma que según nos alejamos del cero se conserva menos detalle de los números. Si dibujamos en la recta real los números que se pueden guardar en un `float`, veremos que la densidad de puntos decrece según nos alejamos del cero. Esta organización, aunque tiene algunos efectos negativos en la práctica, generalmente funciona muy bien porque la distancia relativa con respecto al tamaño siempre es la misma.

Todo este comportamiento se consigue con una técnica muy simple que se conoce con el nombre de *coma flotante*. Los números se guardan moviendo la coma para que todo lo que quede a la izquierda sean ceros pero no haya un cero inmediatamente a la derecha; por supuesto, se anota además el movimiento total de la coma. Así, el número 10.89 se guardaría como el par $(0.1089, 2)$ y el número $-0.000783$ como $(-0.783, -3)$. Las dos partes de este par se conocen como *mantisa* y *exponente* respectivamente. En ciertas circunstancias extremas, es admisible que la mantisa tenga uno o varios ceros a la derecha de la coma; se dice entonces que está *desnormalizada*. Para recuperar el número del par $(m, e)$ hay que calcular $m \times 10^e$. La capacidad del exponente delimita el rango total de reales que se abarcan; si $\bar{e}$ es el máximo exponente, $0.999\cdots \times 10^{\bar{e}}$ será el máximo número real que podremos guardar. El número de dígitos que tiene la mantisa marca la precisión. Porque la coma se ha desplazado, el detalle al que se refiere un dígito de la mantisa cambia dependiendo del exponente. Por ejemplo, el tercer dígito de la mantisa son las milésimas si el exponente es 0, las micras si es $-3$, o las unidades si es 3.

La mantisa y el exponente se guardan en base 2. Rangos y precisiones de `float` y `double` pueden verse en la tabla 1.2.

| | `float` | `double` |
|---|---|---|
| Mantisa (en bits) | 24 | 53 |
| Mantisa (en decimales) | 7.22 | 15.95 |
| Exponente (bits) | 8 | 11 |
| Exponente máximo | 127 | 1023 |
| Exponente mínimo | $-126$ | $-1022$ |
| Magnitud máxima | $3.4028 \times 10^{38}$ | $1.7976 \times 10^{308}$ |
| Magnitud mínima | $1.1754 \times 10^{-38}$ | $2.2250 \times 10^{-308}$ |
| Mínimo (desnormalizado) | $1.4012 \times 10^{-45}$ | $4.9406 \times 10^{-324}$ |

Tabla 1.2: Rangos y precisiones para los números en coma flotante

### 1.3.4 Valores de verdad

El concepto de verdad, lo que se conoce como el álgebra de Boole, se representa con el tipo `bool`. Los literales `true` y `false` son sus únicos posibles valores.

### 1.3.5 Cadenas de caracteres

Para muchos conceptos C++ tiene dos alternativas: una, más básica, que hereda de C y otra, más abstracta, que generalmente está implementada en la biblioteca estándar. Las cadenas de caracteres están en esta situación.

Una cadena de caracteres es una ristra de caracteres arbitrariamente larga. Sus elementos individuales son de tipo `char`. Sirven para contener mensajes, manipular texto, etc. Son fundamentales para la interacción entre humanos y máquinas.

En C, se representan con el tipo `char*`. En el capítulo 6 se verá que el "`*`" quiere decir que tenemos una secuencia (en nuestro caso, de caracteres) de longitud indeterminada. Los literales de tipo cadena de caracteres se escriben encerrándolos entre comillas dobles (`"`); por ejemplo: `"Esto es un mensaje para escribir"`. Dentro de estos literales, el símbolo "`\`" juega el mismo papel que en los caracteres: `"Esto es una \"palabra\" entrecomillada"`.

El manejo de cadenas de caracteres en C es bastante precario. No por falta de funciones predefinidas, sino porque es fácil cometer errores con efectos fatales para un programa. (La mayor parte de las debilidades que permiten a los *crackers* tomar el control de sistemas ajenos, tienen su origen en el uso incorrecto de estas funciones.)

La biblioteca estándar de C++ cuenta con una alternativa llamada `string` (en el fichero de cabecera `string`). Se puede usar un `char*` (en particular sus literales) en cualquier sitio donde sea necesario un `string`; en el apartado 1.9 se verá cómo conseguir usar un `string` donde se espera un `char*`.

### 1.3.6 Compatibilidad de tipos

Un tipo $A$ es compatible con un tipo $B$ cuando se puede usar un valor de tipo $A$ en lugar de un valor de tipo $B$. La relación de compatibilidad no tiene por qué ser simétrica: $A$ puede ser compatible con $B$ y $B$ no serlo con $A$. Como cabe esperar, todos los tipos enteros son compatibles con los reales, porque los enteros son un subconjunto de los reales. En cambio, los tipos reales no son compatibles con ningún tipo entero. Lo más curioso es que todos los tipos enteros son mutuamente compatibles; generalmente, los compiladores avisan cuando intentamos usar un valor demasiado grande para el tipo entero en cuestión.

## 1.4 Identificadores

Como casi todos los lenguajes de programación, C++ tiene identificadores, que son nombres con los que podemos hacer referencia a ciertos objetos. Cuando escribimos `int` estamos haciendo uso de un identificador para referirnos a un tipo de enteros; cuando escribimos `true` nos estamos refiriendo al valor cierto del tipo `bool`. Todos los identificadores que hemos visto hasta ahora están predefinidos en C++.

Los identificadores en C++ son una secuencia de letras, dígitos y subrayados; por ejemplo, `permutaFila`, `permutafila` y `permuta_fila` son identificadores correctos. C++ distingue entre letras minúsculas y mayúsculas; por eso, los tres identificadores que acabamos de escribir son distintos.

Para ayudar en la lectura de los identificadores compuestos por más de una palabra, se separan las distintas palabras con un subrayado o bien se pone en mayúscula la primera letra de cada palabra. En este libro adoptaremos esta segunda técnica. Igualmente, escribiremos en mayúscula la primera letra de los identificadores de tipos, mientras que los identificadores para el resto de conceptos tendrán su primera letra minúscula.

Con preferencia, los identificadores se escriben en minúscula (las mayúsculas son un grito, algo que clama violentamente). Tradicionalmente los identificadores de constantes en C se escriben totalmente en mayúsculas y se utiliza un subrayado para separar las palabras que los componen; en este libro no lo haremos así porque no creemos que haya una frontera tan nítida entre variables y constantes como para que haya que recalcarla tan ostensiblemente.

## 1.5 Declaración de variables

Para declarar la variable `var` del tipo `Tipo` hay que escribir

```
Tipo var;
```

En C++, al contrario que en muchos otros lenguajes (incluso C), una declaración de variable es una *instrucción* y puede aparecer entremezclada con otras instrucciones. La variable que se declara existe inmediatamente después de la declaración hasta el momento en que se acabe el *bloque* (concepto que veremos más adelante) donde se ha definido.

Una variable recién declarada tiene un valor indefinido. Usar una variable que todavía no ha recibido un valor es un error muy difícil de detectar y que se comete con bastante frecuencia cuando se programa. Para evitar este tipo de errores, C++ permite que demos un valor a una variable cuando se declara, de la siguiente forma:

```
Tipo var = valor;
```

Llamaremos *definición de variable* a una declaración con valor inicial.

Que las declaraciones sean instrucciones y que se pueda dar un valor cuando declaramos una variable son dos decisiones que se complementan. La idea es desplazar la declaración de una variable al punto más cercano donde se utiliza, para que así sea factible definirla con un valor inicial adecuado.

Se pueden declarar varias variables (si se desea con sus valores) simultáneamente, separándolas con comas; a saber:

```
Tipo var1, var2 = valor2, var3;
```

No es una práctica recomendable si se tienen que declarar muchas variables o si hay que darles un valor inicial. Economizar el nombre de un tipo no es ni mucho menos un requisito para escribir buenos programas. Lo anterior es equivalente a

```
Tipo var1;
Tipo var2 = valor2;
Tipo var3;
```

que resulta mucho más claro.

## 16 Definición de constantes

El concepto de constante en C++ es más amplio que en otros lenguajes. En general, una constante viene a ser una variable inmutable que toma un valor en el momento de su declaración. Pero es más natural entender una constante como un mecanismo de asociación inverso al que tiene lugar con las variables: asignar un valor a una variable consiste en dar a un nombre dicho valor; en cambio, definir una constante consiste en darle a un valor un nombre nuevo.

Por lo demás, la declaración de una constante es prácticamente igual que la de una variable con valor inicial; en un `const` que sigue al tipo estriba toda la diferencia.

El ejemplo típico es la asignación del nombre `pi` a la famosa constante $\pi$:

```
double const pi = 3.14159265358979238;
```

En C++ el valor de una constante puede resultar de un cálculo:

```
double const numeroE = exp(1);
```

Incluso puede ser el resultado de un cálculo que dependa de una variable; entonces el valor se calculará durante la ejecución y no en la compilación. Por eso, reciben el nombre de *constantes en tiempo de ejecución* y no son lícitas en ciertas construcciones del lenguaje que requieren el manejo de una *constante en tiempo de compilación*.

## 17 Expresiones

Los valores de los distintos tipos se manipulan con expresiones para construir otros valores. Se denomina *evaluar* al proceso que realiza el cálculo de una expresión hasta obtener un valor final.

Las expresiones son el núcleo de la capacidad de cómputo; pero en los lenguajes imperativos como C++ no se pueden complicar excesivamente y enseguida hay que echar mano de las instrucciones (que veremos en el apartado siguiente). Hasta tal punto que una expresión aislada no es capaz de realizar ningún cometido; siempre hay que usar el valor que devuelve en el contexto de una instrucción. Con lo que hemos visto hasta ahora, una expresión podría servir para definir una constante o una variable, pero lo típico es utilizarla para cambiar el valor de una variable mediante las asignaciones que veremos en el apartado 1.8.1.

Las expresiones se forman por aglutinación de expresiones básicas usando el pegamento de las *llamadas a funciones*. Las expresiones básicas son los identificadores de constantes, de variable o los literales, que realmente no calculan nada porque hacen referencia inmediata al valor que son o contienen.

Una función es un mecanismo de cálculo que a partir de unos valores genera otro nuevo. Se accede a una función a través de un identificador. Los valores que espera una función se llaman argumentos o parámetros. Se utiliza (*llama a*) una función escribiendo su identificador seguido por los parámetros separados por comas y rodeados por paréntesis; por ejemplo, para llamar a la función `f` con los argumentos `10` y `s`, hay que escribir `f(10,s)`.

Cada función fija el número de parámetros con los que sabe operar y sus tipos. Por ejemplo, si `f` espera un entero y una cadena, no se puede llamar con tres argumentos (`f(10,"mensaje",3.15)`), ni con valores que no sean compatibles con los tipos que espera (`f(3.14,"mensaje")` o `f(2,'a')`); todos estos ejemplos generarán errores del compilador de C++.

La llamada a una función es una expresión; expresiones son los argumentos que hay que pasar a una función. Así pues, se puede anidar la llamada a una función y pasar su resultado a otra función: `h(g(f(10,"mensaje")), g('a'))`.

Por comodidad y por similitud con la notación matemática usual, algunas funciones toman la forma de operadores. Esto es tan natural que cualquier explicación resultará ridícula. Los operadores son ciertos símbolos que tienen una forma de uso especial. La mayor parte de ellos se escriben en medio de los dos argumentos que reciben; normalmente, cuando sólo reciben uno, se escriben precediéndolo. Por ejemplo,

| Operador | Precedencia | Concepto | Ejemplo |
|---|---|---|---|
| $e_1$`[`$e_2$`]` | 13 | Indexación | `arr[i+1], str[2*i]` |
| $e_1$`->`$e_2$ | 13 | Un campo a través de un puntero | `aux->siguiente` |
| $e_1$`.`$e_2$ | 13 | Un campo de un registro | `fecha.mes` |
| `!`$e_1$ | 12 | Negación | `!(0 <= x && x < y)` |
| `~`$e_1$ | 12 | Complemento bit a bit | `~0xFF` |
| `-`$e_1$ | 12 | Opuesto | `-(x+y)` |
| `*`$e_1$ | 12 | Indirección de un puntero | `*(aux->persona)` |
| $e_1$`*`$e_2$ | 11 | Producto | `(x+y)*(x-y)` |
| $e_1$`/`$e_2$ | 11 | División | `(x+(y/2))/y` |
| $e_1$`%`$e_2$ | 11 | Resto | `x % 2` |
| $e_1$`+`$e_2$ | 10 | Suma | `x*x + y*y` |
| $e_1$`-`$e_2$ | 10 | Resta | `x*x - 2*x*y + y*y` |
| $e_1$`<<`$e_2$ | 9 | Desplazamiento a la izquierda | `x << 1` |
| $e_1$`>>`$e_2$ | 9 | Desplazamiento a la derecha | `x >> 8` |
| $e_1$`<`$e_2$ | 8 | Menor estricto | `x*x + y*y < z*z` |
| $e_1$`<=`$e_2$ | 8 | Menor o igual | `x >= 0` |
| $e_1$`>`$e_2$ | 8 | Mayor estricto | `x > x*x` |
| $e_1$`>=`$e_2$ | 8 | Mayor o igual | `x >= y + z` |
| $e_1$`==`$e_2$ | 7 | Igualdad | `x*x == y*y + z*z` |
| $e_1$`!=`$e_2$ | 7 | Distinto | `x != y + z` |
| $e_1$`&`$e_2$ | 6 | Conjunción bit a bit | `x & 0x7F` |
| $e_1$`^`$e_2$ | 5 | Disyunción exclusiva bit a bit | `x ^ 0x80` |
| $e_1$`\|`$e_2$ | 4 | Diyunción bit a bit | `x \| 0x80` |
| $e_1$`&&`$e_2$ | 3 | Conjunción | `x < 0 && y < 0` |
| $e_1$`\|\|`$e_2$ | 2 | Disyunción | `x < 0 \|\| y < 0` |
| $e_1$`?`$e_2$`:`$e_3$ | 1 | Selección condicional | `x < y ?  x :  y` |

Tabla 1.3: Operadores y su precedencia

la suma de dos expresiones `e1` y `e2` se escribe `e1+e2`, pero el opuesto de la variable `x` es `-x`. Se suele llamar *operandos* a los argumentos de un operador. En la tabla 1.3 se pueden ver todos los operadores de C++.

Cuando en una expresión aparece más de un operador, es necesario saber su orden de ejecución. Los paréntesis sirven para indicar que las expresiones más internas se evalúan antes que las más externas; así, la expresión `2*(10+x)` calcula primero la suma `10+x` y luego multiplica por 2 el resultado. En todo caso, si se omiten los paréntesis, C++ adoptará un orden que viene dado por la *precedencia* de los operadores en juego; a falta de paréntesis, se evalúan primero los operadores con mayor precedencia; a igual precedencia, se evalúa primero el que está más a la izquierda.

C++ admite que varias funciones distintas usen el mismo identificador (que puede tener forma de operador) siempre y cuando el número o tipo de los parámetros sea distinto de una a otra. Es lo que se conoce como *sobrecarga*. Casi todos los operadores están sobrecargados para que actúen de forma coherente sobre los tipos primitivos. En el capítulo 3 veremos cómo añadir nuestras propias definiciones de funciones, con o sin sobrecarga.

A continuación, enumeraremos algunos operadores y funciones predefinidos o que están en la biblioteca estándar de C++. Por economía de espacio, casi todos los presentaremos en una fila de una tabla. Las posibilidades de uso de una función u operador vienen dadas por el número de argumentos, sus tipos

y el tipo del valor que devuelve; para referirnos al tipo de una expresión $e$ escribiremos $\tau(e)$. Muchas operaciones existen para toda una familia de tipos; para evitar enumeraciones inútiles, diremos que necesita un *entero* cuando se pueda usar con `char`, `short`, `int` o `long`, que necesita un *real* cuando se pueda usar con `float` o `double`, y que necesita un *número* cuando se pueda usar con *enteros* o *reales*. Finalmente, con $\tau(e_1) \uparrow \tau(e_2)$ indicaremos que el tipo del resultado es el *mayor* entre los tipos de $e_1$ y $e_2$, según el siguiente orden, de menor a mayor, para los tipos númericos: `char`, `short`, `int`, `long`, `float` y `double`.

### 1.7.1 Operaciones aritméticas

Los operadores usuales en aritmética se utilizan en C++ para manejar números y algo más, como puede comprobarse en la tabla 1.4.

| $e$ | $\tau(e_1)$ | $\tau(e_2)$ | $\tau(e)$ | **Descripción** | **Ejemplo** |
|---|---|---|---|---|---|
| $e_1 + e_2$ | numérico | numérico | $\tau(e_1) \uparrow \tau(e_2)$ | Suma | `17+14.6` $\leadsto$ `31.6` |
| $e_1 + e_2$ | `string` | `string` | `string` | Concatenación | `"ab"+"c"` $\leadsto$ `"abc"` |
| $-e_1$ | numérico | | $\tau(e_1)$ | Opuesto | `-(7 + 7.6)` $\leadsto$ `-14.6` |
| $e_1 - e_2$ | numérico | numérico | $\tau(e_1) \uparrow \tau(e_2)$ | Resta | `17-14.6` $\leadsto$ `2.4` |
| $e_1$`*`$e_2$ | numérico | numérico | $\tau(e_1) \uparrow \tau(e_2)$ | Producto | `17*14.6` $\leadsto$ `248.2` |
| $e_1/e_2$ | real | real | $\tau(e_1) \uparrow \tau(e_2)$ | División con decimales | `11.5/2.5` $\leadsto$ `4.6` |
| $e_1/e_2$ | real | entero | $\tau(e_1)$ | División con decimales | `11.5/2` $\leadsto$ `5.75` |
| $e_1/e_2$ | entero | real | $\tau(e_2)$ | División con decimales | `11/2.5` $\leadsto$ `4.4` |
| $e_1/e_2$ | entero | entero | $\tau(e_1) \uparrow \tau(e_2)$ | División entera | `11/3` $\leadsto$ `3` |
| $e_1$`%`$e_2$ | entero | entero | $\tau(e_1) \uparrow \tau(e_2)$ | Resto de la división | `11%3` $\leadsto$ `2` |

Tabla 1.4: Operaciones aritméticas

### 1.7.2 Operaciones lógicas y relacionales

Se llama, respectivamente, *predicados* y *condiciones* a las funciones y expresiones que devuelven un valor de tipo `bool`. Son la base para decidir si hacer o evitar cómputos dependiendo de una circunstancia.

Las condiciones más simples son las constantes y variables de tipo `bool`; el siguiente nivel de complejidad se alcanza con los operadores relacionales, que aparecen en la tabla 1.5; finalmente se pueden combinar con las conectivas lógicas usuales, que en C++ adoptan la forma que se muestra en la tabla 1.6.

### 1.7.3 Manipulación de bits

C++ es un lenguaje atípico porque permite trabajar en diversos niveles de abstracción. En particular, en la parte menos abstracta, es posible tratar los enteros como lo que realmente son en la máquina: números binarios. Todos los operadores de la tabla 1.7 son muy fáciles de entender si concebimos los enteros como una ristra de ceros y unos, sin adjudicarles ninguna correspondencia numérica.

Algunos tienen un claro sentido cuando interpretamos su efecto numéricamente. En particular, `x << n` es $\mathtt{x} \times 2^{\mathtt{n}}$ (salvo desbordamiento) y `x >> n` es $\lfloor \mathtt{x}/2^{\mathtt{n}} \rfloor$.

### 1.7.4 Expresión de selección condicional

Algunas veces se quiere elegir entre un valor u otro dependiendo de una condición. Con la expresión de selección se consigue este cometido. Tiene la forma siguiente: $c?e_1 : e_2$; Su valor es el de $e_1$ si $c$ es `true` o el de $e_2$ si $c$ es `false`. Por ejemplo, la expresión `m < n ? m : n` da el mínimo de los números `m` y `n`.

### 1.7.5 Conversión de tipos

La conversión entre tipos numéricos (incluyendo los tipos enumerados que veremos en el apartado 4.4) se resuelve con *casting*: para convertir la expresión numérica $e$ al tipo `tipo` hay que escribir `(tipo)`$(e)$.

| $e$ | $\tau(e_1)$ | $\tau(e_2)$ | **Descripción** | **Ejemplo** |
|---|---|---|---|---|
| $e_1$`==`$e_2$ | básico | básico | Igualdad | `5 == 6.8` $\rightsquigarrow$ `false` |
| $e_1$`==`$e_2$ | `string` | `string` | Igualdad lexicográfica | `"Hola" == "hola"` $\rightsquigarrow$ `false` |
| $e_1$`!=`$e_2$ | básico | básico | Desigualdad | `5 != 6.8` $\rightsquigarrow$ `true` |
| $e_1$`!=`$e_2$ | `string` | `string` | Desigualdad lexicográfica | `"Hola" != "hola"` $\rightsquigarrow$ `true` |
| $e_1$`<`$e_2$ | básico | básico | Menor estricto | `10 < 10.3` $\rightsquigarrow$ `true` |
| $e_1$`<`$e_2$ | `string` | `string` | Menor lexicográfico | `"hola" < "hola"` $\rightsquigarrow$ `false` |
| $e_1$`<=`$e_2$ | básico | básico | Menor o igual | `'d' <= 'a'` $\rightsquigarrow$ `false` |
| $e_1$`<=`$e_2$ | `string` | `string` | Menor o igual lexicográfico | `"ho" <= "hola"` $\rightsquigarrow$ `true` |
| $e_1$`>`$e_2$ | básico | básico | Mayor estricto | `'a' > 10` $\rightsquigarrow$ `true` |
| $e_1$`>`$e_2$ | `string` | `string` | Mayor lexicográfico | `"hola" > "ho"` $\rightsquigarrow$ `true` |
| $e_1$`>=`$e_2$ | básico | básico | Mayor o igual | `false >= true` $\rightsquigarrow$ `false` |
| $e_1$`>=`$e_2$ | `string` | `string` | Mayor o igual lexicográfico | `"hola" >= "hola"` $\rightsquigarrow$ `true` |

Tabla 1.5: Operaciones relacionales

| $e$ | **Descripción** | **Ejemplo** |
|---|---|---|
| `!`$c_1$ | Negación | `!(3 <= 3.5)` $\rightsquigarrow$ `false` |
| $c_1$`||`$c_2$ | Disyunción | `false || (3 <= 3.5)` $\rightsquigarrow$ `true` |
| $c_1$`&&`$c_2$ | Conjunción | `(3 <= 3.5) && false` $\rightsquigarrow$ `false` |

Tabla 1.6: Operaciones lógicas

La conversión más importante para este libro es la que pasa de un real a un entero. C++ la ejecuta eliminando la parte decimal; por tanto, `(int)2.7` e `(int)2.1` valen 2 mientras que `(int)-2.7` e `(int)-2.1` valen `-2`.

### 1.7 6 Funciones de la biblioteca estándar

En la biblioteca estándar hay multitud de funciones útiles. Las tablas 1.8 y 1.9 son un pequeño resumen; en la documentación de tu compilador de C++ encontrarás una referencia más completa.

## 1 8 Instrucciones básicas

Las instrucciones son el mecanismo para coordinar la evolución de un cálculo. Al igual que las expresiones, se pueden anidar para construir elementos cada vez más complejos. En esta sección veremos las instrucciones básicas; son la asignación y la llamada a acciones o procedimientos. El resto se resumirán en el capítulo 2.

### 1.8 1 Asignación

La instrucción más básica de un lenguaje imperativo como C++ es la asignación, que da un valor nuevo a una variable. Esta instrucción tiene la forma siguiente:

`var =` ⟨*expresión*⟩`;`

Su modo de operación es harto sencillo: se evalúa la expresión hasta tener un valor que se guardará en la variable. El tipo de la expresión debe ser compatible con el tipo de la variable.

Una asignación no tiene nada que ver con una igualdad matemática. La asignación es completamente asimétrica: la expresión va siempre a la derecha y la variable a la izquierda. Más adelante veremos otros elementos que pueden ir a la izquierda de una asignación, aunque siempre serán conceptualmente similares a una variable.

| $e$ | $\tau(e_1)$ | $\tau(e_2)$ | $\tau(e)$ | **Descripción** | **Ejemplo** |
|---|---|---|---|---|---|
| `~`$e_1$ | entero | | $\tau(e_1)$ | Complemento bit a bit | `~0xF5300CAF` $\leadsto$ `0x0ACFF350` |
| $e_1$`\|`$e_2$ | entero | entero | $\tau(e_1) \uparrow \tau(e_2)$ | Disyunción bit a bit | `0x1245 \| 0x9ADE` $\leadsto$ `0x9ADF` |
| $e_1$`&`$e_2$ | entero | entero | $\tau(e_1) \uparrow \tau(e_2)$ | Conjunción bit a bit | `0x1245 & 0x9ADE` $\leadsto$ `0x1245` |
| $e_1$`^`$e_2$ | entero | entero | $\tau(e_1) \uparrow \tau(e_2)$ | O exclusivo bit a bit | `0x1245 ^ 0x9ADE` $\leadsto$ `0x889B` |
| $e_1$`>>`$e_2$ | entero | entero | $\tau(e_1)$ | Desplazamiento derecha | `0x12459ADE >> 3` $\leadsto$ `0x0248B35B` |
| $e_1$`<<`$e_2$ | entero | entero | $\tau(e_1)$ | Desplazamiento izquierda | `0x12459ADE << 3` $\leadsto$ `0x922CD6F0` |

Tabla 1.7: Operaciones de manipulación de bits

| $e$ | $\tau(e_1)$ | $\tau(e_2)$ | $\tau(e)$ | **Descripción** |
|---|---|---|---|---|
| `cos(`$e_1$`)` | `double` | | `double` | Coseno de radianes |
| `sin(`$e_1$`)` | `double` | | `double` | Seno de radianes |
| `tan(`$e_1$`)` | `double` | | `double` | Tangente de radianes |
| `acos(`$e_1$`)` | `double` | | `double` | Arcocoseno en radianes |
| `asin(`$e_1$`)` | `double` | | `double` | Arcoseno en radianes |
| `atan(`$e_1$`)` | `double` | | `double` | Arcotangente en radianes |
| `atan2(`$e_1, e_2$`)` | `double` | `double` | `double` | Arcotangente de $e_1/e_2$ |
| `exp(`$E_1$`)` | `double` | | `double` | $e^{E_1}$ |
| `log(`$e_1$`)` | `double` | | `double` | Logaritmo neperiano |
| `log10(`$e_1$`)` | `double` | | `double` | Logaritmo en base 10 |
| `pow(`$e_1, e_2$`)` | `double` | `double` | `double` | Potencia, $e_1^{e_2}$ |
| `sqrt(`$e_1$`)` | `double` | | `double` | Raíz cuadrada |
| `fabs(`$e_1$`)` | `double` | | `double` | Valor absoluto |
| `floor(`$e_1$`)` | `double` | | `double` | El entero por debajo |
| `ceil(`$e_1$`)` | `double` | | `double` | El entero por encima |
| `rint(`$e_1$`)` | `double` | | `double` | Redondeo al entero más cercano |

Tabla 1.8: Funciones interesantes en `math.h`

### 1.8.2 Incrementos y decrementos

Suele ser muy frecuente aumentar en 1 el valor de una variable. Aunque se puede escribir esto,

```
var = var + 1;
```

no se suele hacer porque C++ tiene una instrucción de incremento para esta tarea:

```
var++;
```

Igualmente, para la tarea

```
var = var - 1;
```

existe la instrucción de decremento siguiente:

```
var--;
```

### 1.8.3 Asignación con operación

El esquema

`x = x +` ⟨*cierto cálculo*⟩`;`

es tan habitual que C++ tiene una forma especial de asignación que lo abrevia:

| $e$ | $\tau(e_1)$ | $\tau(e)$ | **Descripción** |
|---|---|---|---|
| `atoi(`$e_1$`)` | `char*` | `int` | El entero que está en la cadena |
| `atol(`$e_1$`)` | `char*` | `long int` | El entero que está en la cadena |
| `atof(`$e_1$`)` | `char*` | `double` | El real que está en la cadena |

Tabla 1.9: Funciones interesantes en `stdlib.h`

`x +=` ⟨*cierto cálculo*⟩`;`

Estas asignaciones combinadas existen para los operadores `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^` y `|`, y adoptan la forma esperable de `+=`, `-=`, `*=`, `/`, `%=`, `<<=`, `>>=`, `&=`, `^=` y `|=`, respectivamente.

### 1.8.4 Llamada a una acción y la biblioteca estándar

Una acción es a las instrucciones lo que una función a las expresiones. Las acciones también se llaman procedimientos. Funciones y acciones reciben el nombre conjunto de subprogramas.

Las acciones no generan un valor; desempeñan su cometido produciendo algún efecto, ya sea visible para quien use el programa, ya sea para condicionar su evolución futura. Las acciones pueden recibir parámetros, aunque tal como veremos en el capítulo 3 admiten variantes inadecuadas en las funciones. Dichos parámetros son obviamente expresiones; es otra demostración de que la expresiones siempre recaen en una instrucción. La llamada a un procedimiento se efectúa con su nombre seguido de los parámetros entre paréntesis.

Los números seudoaleatorios son la versión informática del ruido blanco: una secuencia de números imprevisibles. La función `rand`, sin argumentos, devuelve el siguiente número de esta secuencia cada vez que se ejecuta. Pero los algoritmos usuales necesitan una *semilla* inicial que condicionará la secuencia de números aleatorios que vamos a obtener. Es el trabajo típico de una acción: realizar un cambio en el programa que condicionará parte de lo que ocurra a continuación. Efectivamente, hay un procedimiento llamado `srand`, que espera la semilla. Establecemos una nueva semilla ejecutando:

`srand(`⟨*cálculo para generar una nueva semilla*⟩`);`

## 1.9 Cadenas de caracteres y programación orientada a objetos

Las cadenas de caracteres en la versión `string` sólo tienen un inconveniente: están basadas en la parte de orientación a objetos de C++, que no consideraremos en este libro. Tendremos más veces este mismo problema; por ejemplo, con la entrada y salida del apartado 1.10. Afortunadamente, con unas recetas simples podremos salvar estos escollos.

Un cimiento de la programación orientada a objetos es la vinculación de un tipo con sus operaciones. Para recalcar estas situaciones, hay una nomenclatura paralela a la que aquí se ha visto: se llama clases a los tipos, objetos a los valores, funciones miembro a las funciones, acciones miembro a las acciones y, en general, métodos a los subprogramas. El tipo `string` es realmente una clase y muchas de sus operaciones son métodos. Por ejemplo, la función `substr` que extrae una porción (subcadena) de una cadena es realmente una función miembro.

Los métodos se utilizan de una forma ligeramente distinta a los subprogramas. Si `substr` fuera una función normal, el primer argumento sería la cadena objetivo de la que queremos extraer una porción; con el resto de parámetros especificaríamos la localización de la porción; por ejemplo:

```
substr(cadenaObjetivo, inicio, longitud);
```

Como es un método, la cadena objetivo precede a la función que parecerá tener un argumento menos; la llamada toma la forma `cadenaObjetivo.substr(inicio, longitud)`. Obsérvese el punto que precede

| Método | Tipo | Descripción |
|---|---|---|
| `length()` | F | Número de caracteres |
| `size()` | F | `length()` |
| `insert`$(p, w)$ | A | Inserta la cadena $w$ en la posición $p$ |
| `substr`$(p, n)$ | F | Devuelve un `string` con $n$ caracteres a partir de $p$ |
| `replace`$(p, n, w)$ | A | Cambia, a partir de $p$, $n$ caracteres por los de $w$ |
| `erase`$(p, n)$ | A | Elimina $n$ caracteres a partir de $p$ |
| `find`$(w, p)$ | F | Índice de la primera aparición de $w$, empezando en $p$ |
| `rfind`$(w, p)$ | F | Similar a `find` pero hacia atrás |

Tabla 1.10: Funciones (F) y acciones (A) miembros para las cadenas de caracteres

al nombre de la función. Este cambio sintáctico no es un capricho de la programación orientada a objetos, sino el resultado de un cambio de responsabilidades; ya no estamos ejecutando una función, sino que le estamos pidiendo a un objeto que ejecute un método con ciertos argumentos. En el ejemplo anterior, el objeto `cadenaObjetivo` recibe la petición de extraer una subcadena que empieza en `inicio` y abarca los siguientes `longitud` caracteres.

Al igual que las llamadas a acciones son instrucciones, las llamadas a acciones miembro son instrucciones; análogamente, las llamadas a funciones miembros son expresiones y devuelven un valor que hay que usar en algún contexto:

```
int const posicionPunto = nombreFichero.rfind(".", nombreFichero.size()-1);
string const extension =
  posicionPunto == string::npos // si no hay punto
    ?  ""
    :  nombreFichero.substr(posicionPunto+1, nombreFichero.size());
```

Los caracteres de un `string` se numeran consecutivamente desde 0. Para acceder al carácter en la posición `e` de la cadena `s`, se ha de escribir la expresión `s[e]`.

Concatenar dos cadenas es construir una con los caracteres de la primera seguidos por los de la segunda; en la tabla 1.4 está recogido que el operador `+` se encarga de esta tarea. También existe la asignación combinada para la concatenación, de forma que `s1 += s2` es equivalente a `s1 = s1+s2`; además, en este caso, no supone un ahorro de escritura sino también una mejora sustancial del rendimiento porque no se tiene que crear una cadena intermedia con el resultado de la concatenación.

En la tabla 1.10 se pueden encontrar algunos de los métodos más útiles para manipular `string`s. Las funciones miembro `find` y `rfind` devuelven el índice donde han encontrado $w$; si no lo encuentran, devuelven la constante `string::npos`.

Ya se ha dicho que, siempre que se espere una cadena en forma `string`, se puede dar una cadena en forma `char*` (en particular, los literales). Pero, cuando se necesita un `char*` (porque haya que llamar a alguna función de la biblioteca estándar) y tenemos una cadena `s` de tipo `string`, hay que escribir `s.c_str()`.

## 1.10 Rudimentos de entrada y salida

En C++ hay dos formas para leer y escribir datos: con la vieja biblioteca de C (para la que hay que incorporar el fichero de cabeceras `stdio.h`) y con la nueva biblioteca de C++ (que requiere incorporar el fichero `iostream`). Usaremos la segunda. Tiene sus ventajas y sus inconvenientes: es mucho más expresiva, más sencilla y tiene más funcionalidad; en cambio, está implementada usando la parte orientada a objetos C++ que no consideraremos en este libro.

Todo programa en C o C++ tiene tres *canales* (*streams*) de comunicación preparados. Por uno de ellos se puede leer y por los otros dos se puede escribir. Estos canales son la forma más básica en que un programa se puede comunicar con su entorno. La manifestación externa de estos canales depende del sistema operativo y de cómo ejecutemos nuestro programa. Pero generalmente el canal de entrada se conecta al teclado (de forma que el programa leerá por este canal lo que escribamos) y ambos canales de salida se conectan con la pantalla o con una ventana (de forma que allí veremos lo que el programa escriba en estos canales).

Cuando se incluye el fichero `iostream`, el canal de entrada se llama `cin`, y los canales de salida, `cout` y `cerr`. El programa debería escribir todos los resultados de una operación normal por el canal `cout` y todos los mensajes de error por el canal `cerr`.

Todo lo que se escribe o se lee por estos canales está en un formato legible para un humano. Porque cumplen esta propiedad se dice que son canales de *texto*. Hay canales *binarios* en donde los datos viajan en el formato interno de la máquina; se evita así tener que hacer conversiones a base 10 cuando se escriben números, etc. No los trataremos en este libro porque, aunque son más eficientes, no es tarea grata ni fácil leerlos directamente.

### 1.10.1 Canales de salida

Ambos canales de salida se utilizan de igual forma, así que todo lo que expliquemos para uno se puede aplicar al otro. Más aún, también se puede aplicar a la escritura en ficheros, como veremos en el apartado 1.10.3. Se escribe en un canal de salida utilizando el operador "`<<`". Por ejemplo, la instrucción

```
cout << "Hola a todos";
```

escribe "Hola a todos" en el canal de salida. Tal vez resulte desconcertante que estemos usando el operador `<<` que sirve para desplazar a la izquierda un número. ¿Qué hace este operador escribiendo en un canal? La respuesta es muy sencilla: en C++ se puede añadir nuevo comportamiento a los operadores (y en general a cualquier función). No hay restricción a esta posibilidad, salvo que el compilador pueda distinguir qué comportamiento debe utilizar en cada ocasión. El compilador se apoya en el tipo de los operandos: si un operador no se podía aplicar a ciertos tipos, es lícito definir un nuevo comportamiento para dicho operador con esos tipos. En su estado inicial, C++ no tiene ninguna definición para el operador `<<` cuando el operando izquierdo es un canal de salida y, por tanto, el compilador deja que se redefina con el comportamiento que nos venga en gana. Y así se ha hecho en la biblioteca `iostream`, de forma que escribe el argumento derecho en el canal que se da como argumento izquierdo. De hecho, hay una definición del operador `<<` con cada tipo básico como operando derecho. Es más, suele ser útil añadir nuevas definiciones de este operador para los tipos nuevos que se definan. (En el capítulo 3 veremos cómo añadir definiciones a un operador, y en el capítulo 4, cómo definir nuevos tipos.)

En definitiva, con el operador `<<`, que en el contexto de los canales de salida recibe el nombre de *insertador* (*inserter*), se puede escribir cualquier tipo predefinido. Por ejemplo,

```
int iteraciones = 0;
double pi;
⟨Cálculo de una aproximación a π, con un cierto número de iteraciones⟩
cout << "La aproximación a Pi tras ";
cout << iteraciones;
cout << " iteraciones es ";
cout << pi;
cout << "\n";
```

El insertador tiene otra peculiaridad: se puede *secuenciar*. El ejemplo anterior se puede escribir como

```
cout << "La aproximación a Pi tras " << iteraciones
     << " iteraciones es " << pi << "\n";
```

Los detalles de la presentación se gestionan con *manipuladores* (*manipulators*). El más usado es el que sirve a la vez para escribir un final de línea y para forzar a que todo lo que hayamos escrito hasta ahora se vea al otro extremo del canal: es el manipulador `endl`. Los manipuladores se *envían* a un canal usando el insertador:

```
cout << "La aproximación a Pi tras " << iteraciones
     << " iteraciones es " << pi << endl;
```

El carácter de final de línea "`\n`" y el manipulador `endl` (*end line*, fin de línea) no son exactamente iguales porque el segundo además obliga a mandar al otro extremo del canal todo lo que se ha escrito. Las diferencias son perceptibles en programas que interactúen con una persona o si hay algún problema y nuestro programa se detiene bruscamente. El comportamiento impaciente de `endl` se tiene por separado en el manipulador `flush`; en otros términos, `cout << endl` es equivalente a `cout << "\n" << flush`. Hay manipuladores para gestionar muchos detalles; cabe destacar los siguientes:

`dec`, `hex`, `oct` Sirven para controlar la base en la que se escriben los números enteros. Cambian el canal de forma que todos los números que sigan, y hasta que se cambie la base, se escriben en base 10, 16 u 8, respectivamente. Inicialmente se escribe en base 10.

`setw` Sirve para especificar la anchura mínima que ocupará el siguiente dato. Sólo afecta a un dato y vuelve a tomar el valor 0. Recibe un argumento con la anchura.

`setprecision` Establece el número de decimales que se mostrarán en los siguientes números en coma flotante que escribamos. Por defecto se muestran 6 decimales. Necesita un argumento con el número de decimales.

Para poder usar los manipuladores hay que incluir el fichero `iomanip`. Veamos un ejemplo de uso:

```
cout << setw(13) << 32 << endl;
cout << setw(13) << hex << 32 << endl;
cout << setw(13) << 32 << endl;
cout << setw(13) << 0.1234567890123456789 << endl;
cout << setw(13) << setprecision(10) << 0.1234567890123456789 << endl;
cout << 0.1234567890123456789 << endl;
cout << setw(13) << setprecision(10) << 0.1234567890123456789 << endl;
cout << setw(13) << setprecision(15) << 0.1234567890123456789 << endl;
```

Y su resultado:

```
           32
           20
           20
     0.123457
  0.123456789
0.123456789
  0.123456789
0.123456789012346
```

Obsérvese cómo `setw` es el único manipulador que pierde su efecto tras la siguiente escritura.

### 1.10.2 Canales de entrada

Como un canal de entrada cumple un papel simétrico a uno de salida, la forma de leer será con el operador "`>>`", que en este contexto recibe el nombre de *extractor*. Al igual que el insertador, existe una definición

para cada tipo predefinido y puede recibir nuevas definiciones para nuestros propios tipos (consulta los capítulos 3 y 4). Los extractores también se pueden concatenar. En definitiva, la siguiente instrucción

```
cin >> i >> d >> s;
```

leerá un `int`, un `double` y un `string`, suponiendo que `i`, `d` y `s` tienen esos tipos, respectivamente.

Pero al contrario que con los canales de salida, donde todo se resuelve con paciencia y esfuerzo, los canales de entrada exigen esencialmente mucho cuidado. Primero, porque se pueden acabar. Si se intenta extraer un elemento de un canal que se ha acabado no recibiremos ninguna notificación especial; simplemente, la variable destino no se modificará. Lo normal es preguntar al canal de entrada, con el método `eof()`, si queda algo por leer. EOF son las iniciales de *End Of File* (final de fichero); la expresión `cin.eof()` devuelve cierto precisamente si no queda ningún carácter por leer del canal de entrada.

El siguente problema es la (falta de) concordancia entre lo que pedimos leer y lo que viene a continuación por el canal de entrada. ¿Qué ocurre si intentamos leer un entero pero lo que viene a continuación son letras? De nuevo, la variable destino no se modificará, pero en el canal de entrada se anotará que algo ha fallado. Esta contingencia se puede comprobar con el método `fail()`; la expresión `cin.fail()` devuelve `true` precisamente si se ha fallado en una lectura. Para borrar la marca de fallo hay que ejecutar la instrucción `cin.clear()`. Porque en la práctica es lo más cómodo, alcanzar el final de fichero se considera un fallo; por tanto, si `cin.eof()` vale `true`, `cin.fail()` también lo vale.

El último problema es cuánto se lee. Los extractores para los tipos desprecian todos los espacios que preceden al objeto que vamos a leer. Para el tipo `char` se lee el primer carácter que no es un espacio. Para el resto de los tipos, se lee mientras que esté bien formado y no sea un espacio, de forma que, al terminar, el canal está en disposición de ofrecer el primer espacio que sigue al objeto.

Este comportamiento facilita la lectura consecutiva de elementos separados por espacios. Pero algunas veces se quiere tener un control más detallado. El manipulador `ws` (*whitespace*) quita del canal de entrada todos los espacios que vengan a continuación. El procedimiento `getline` lee la siguiente línea de un canal de entrada; de esta forma, la ejecución de `getline(cin, s)` pone en la cadena `s` todos los caracteres hasta el siguiente final de línea ("`\n`"); el propio final de línea se quita del canal, pero no se pone en la cadena.

Finalmente, los canales de entrada tienen dos métodos que no desprecian ningún carácter, con independencia de su uso. Los dos métodos se llaman `get` y se diferencian sólo en un detalle sutil. El primero es una acción con un argumento que debe ser una variable de tipo `char`; la ejecución de `cin.get(c)` deja en `c` el siguiente carácter. El segundo es una función sin argumentos que devuelve el valor del siguiente carácter; la instrucción `c = cin.get();` es equivalente a la anterior. Hay una diferencia más: el tipo de `cin.get()` es `int`, lo que capacita a esta función miembro a devolver un $-1$ para indicar que se ha alcanzado el final de fichero.

### 1.10.3 Ficheros o archivos

En C++ los ficheros o archivos se tratan como los canales que acabamos de ver en los dos apartados anteriores. Por supuesto, un fichero se puede abrir para leer o para escribir, y entonces se tratará como `cin` o como `cout`.

Un programa que utilice ficheros debe incluir la cabecera `fstream` (que automáticamente incluye `iostream`). Si queremos leer de un fichero, tenemos que declarar una variable de tipo `ifstream` (en inglés, *Input File Stream*); si queremos escribir, de tipo `ofstream` (*Output File Stream*). Una vez declarada, hay que vincularla con un fichero; lo hace el método `open`, que espera el nombre del fichero como argumento (una cadena en la forma `char*`). Cuando se termina de operar con un fichero, hay que llamar al método `close()`, para que pueda devolver al sistema operativo los recursos que consume por estar abierto. Como ejemplo, el siguiente código lee la primera palabra del fichero `texto`:

```
ifstream fichero;
fichero.open("texto");
string palabra;
fichero >> palabra;
fichero.close();
```

Parte de lo anterior se puede abreviar. La vinculación con un nombre de fichero se puede hacer en el momento de declarar la variable con la sintaxis:

```
ifstream fichero("texto");
```

Además, cuando se cierra el bloque donde se ha declarado una variable de tipo fichero, C++ llama implícitamente al método `close()`.

Lo más interesante, y es una magia fruto de la programación orientada a objetos, es que el tratamiento de ficheros y los canales `cin`, `cout` y `cerr` se pueden uniformar hasta el punto de que el mismo código funcione con unos u otros. Los tipos `istream` y `ostream` abstraen el concepto de canal de entrada y de salida, respectivamente, y olvidan los detalles de la fuente concreta de datos. Con ellos se puede escribir el mismo código para manipular `cin` que un `ifstream`. Pero hasta el capítulo 3 no se los podrá aprovechar correctamente; para entonces el siguiente código resultará perfectamente legible. Es una función que nos dice si ya no quedan, en la línea actual, caracteres que no sean espacios. Elimina todos los espacios que vengan a continuación, salvo el final de línea (si lo encuentra).

```
bool eoln(istream & in) {
  if (in.eof()) return true;
  if (in.fail()) return false;
  int c = in.get();
  while (c != -1 && c != '\n' && isspace(c)) c = in.get();
  if (c != -1) in.putback((char)c);
  return c == -1 || c == '\n';
}
```

## 1.11 Punto de arranque de un programa

Todo programa en C++ debe contener la definición de una función llamada `main`, que es por donde empieza a ejecutar. Como receta hasta el capítulo 3 en donde veremos cómo definir funciones, puedes reutilizar el siguiente esquema, añadiendo los fragmentos que desees probar entre las llaves:

```
int main() {
  cout << "Dame un número:  " << flush;
  int n;
  cint >> n;
  cout << "En base 8 :  " << oct << n << endl
       << "En base 10:  " << dec << n << endl
       << "En base 16:  " << hex << n << endl;
}
```

En este caso, se muestra en base 8, 10 y 16 el entero leído en `n`.

## 11 Espacios de color

33

Me contaron en el cole que se podían sacar todos los colores mezclando cantidades adecuadas de luz roja, verde y azul. Al volver a casa me acerqué a la tele con una lupa y vi un montón de lucecitas rojas, verdes y azules encendiéndose y apagándose y nada más. Como en la tele salía todo lo que podía imaginar, me lo creí. (Y ahora leo en este ejercicio que es mentira, que los humanos podemos ver colores que nunca podrá mostrar una tele con sus lucecitas rojas, verdes y azules.) Luego fui a la habitación de mi abuelo, pero al aplicar la lupa a su vieja tele en blanco y negro sólo vi lucecitas grises. Y pensé, "¡qué lista es la tele de mi abuelo! sabe mezclar rojo, verde y azul en un gris y se siguen entendiendo las imágenes". (Pero ahora leo en este ejercicio que la tele en blanco y negro no era tan lista porque por el aire no viajan los rojos ($r$), verdes ($v$) y azules ($a$) de cada punto, sino tres cantidades muy raras que se calculan con

$$\begin{aligned} Y &= 0.299r + 0.587v + 0.114a \\ I &= 0.596r - 0.275v - 0.321a \\ Q &= 0.212r - 0.528v + 0.311a \end{aligned}$$

Y resulta que la cantidad $Y$ es la intensidad de gris que mostraba la tele de mi abuelo. Ahora que sé invertir matrices, me doy cuenta de que la tele lista era la de color que tenía que hacer

$$\begin{aligned} r &= 1Y + 0.955I + 0.618Q \\ v &= 1Y - 0.271I - 0.645Q \\ a &= 1Y - 1.11I + 1.7Q \end{aligned}$$

para cada punto.) Luego vi que por la computadora de mi mamá pasaba otra vez el pez azul y llegué a tiempo para investigar su cola con mi lupa y de nuevo sólo el montón de lucecitas rojas, verdes y azules. (Pero ahora leo que seguramente la imagen del pez estaba guardada en un fichero JPEG, en donde tampoco hay rastro de rojo, ni verde, ni azul, sino unas cantidades muy raras que se calculan con

$$\begin{aligned} Y &= 0.299r + 0.587v + 0.114a \\ C_b &= -0.1687r - 0.3313v - 0.5a \\ C_r &= 0.5r - 0.4187v - 0.0813a \end{aligned}$$

Otra vez la $Y$ de la tele de mi abuelo. Sigo leyendo y descubro que RGB —las siglas de rojo, verde y azul—, YIQ e $\mathrm{YC}_b\mathrm{C}_r$ son espacios de color y que hay tareas para las que unos son más adecuados que otros.) Fui a por mis lápices de colores pero no encontré ese azul y mezclando siempre era peor, hasta que me manché la falda. (Ahora podría buscarlo con mi propio programa. Empezaré por escribir programas que me pasen de un espacio de color a otro.)

***Un poco de historia*** La cantidad $Y$ define la luminosidad con la que los humanos percibimos un color. Su fórmula indica que el verde y el rojo influyen mucho más que el azul en la percepción de luminosidad; es decir, que los humanos somos particularmente insensibles a la luz azul. Uno de esos hechos básicos que, gracias a la orgullosa incompetencia de gobiernos sucesivos, ha entrado a formar parte de la inmensa miseria de lo desconocido para casi todo el mundo.

Ciertamente, las emisoras de televisión en blanco y negro producían una señal con el sonido y otra con $Y$, que era lo que se grababa. Para cuando llegó la televisión en color, se resolvió el gran dilema de la doble emisión con el espacio de color $YIQ$ (al menos en los EE.UU.). Lo que parecería un apaño chapucero resultó tener sus propias ventajas. Los humanos somos mucho más sensibles al cambio de

intensidad luminosa (la señal $Y$) que al cambio de tinte y saturación (los otros elementos del color que viajan mezclados en $I$ y $Q$). Así, se pudo emitir con más imprecisión las señales $I$ y $Q$ que la señal $Y$, con el consiguiente ahorro en anchura de banda electromagnética.

La codificación JPEG aprovecha también esa deficiencia en nuestra percepción para comprimir más drásticamente las señales $C_b$ y $C_r$.

Los humanos no se desenvuelven bien en ninguno de estos espacios de color y, ni siquiera cuando se ayudan de un programa interactivo, encuentran fácilmente un color. El espacio HSV se ideó para remediar este problema. Desafortunadamente para el cuerpo de este ejercicio, su relación con RGB viene dada por una transformación espacial que exige bastante más que una simple expresión.

***Bibliografía*** En cualquier texto sobre informática gráfica encontrarás descripción y propiedades de estos espacios de color y otros muchos. Una referencia obligada es [FvDFH97]. Pero entender el color y sus implicaciones prácticas no es algo trivial; hay monografías técnicas para expertos; afortunadamente, hay también presentaciones asequibles, como [Bal02], y obras para el deleite visual, como [Pas01].

## 12 Ser o no ser... triángulo

Dadas tres cantidades reales positivas, se quieren dilucidar las siguientes situaciones:

***¿Es un triángulo?*** Si los valores de dichas cantidades pueden corresponder a las longitudes de los lados de un triángulo (quizá la figura 1.1 pueda ayudarte).

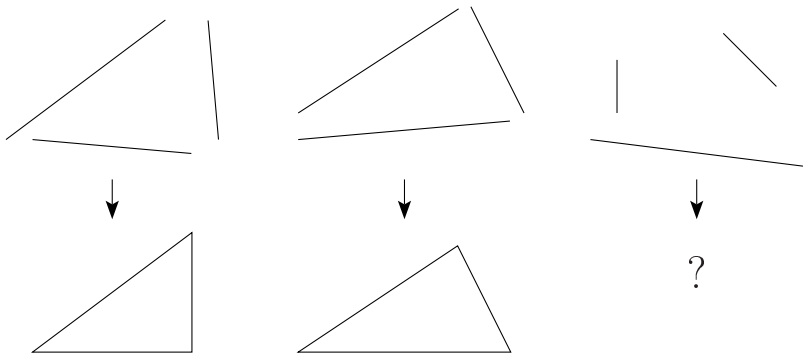



Figura 1.1: Construcción de triángulos a partir de sus lados

***¿Es escaleno?*** En el caso de que las medidas puedan corresponder a las longitudes de los lados de un triángulo, si dicho triángulo es escaleno.

***¿Es equilátero?*** En el caso de que las medidas puedan corresponder a las longitudes de los lados de un triángulo, si dicho triángulo es equilátero.

***¿Es isósceles?*** En el caso de que las medidas puedan corresponder a las longitudes de los lados de un triángulo, si dicho triángulo es isósceles.

## 13 Cálculo y verificación de la letra del D.N.I.

El documento nacional de identidad (DNI) en España, consta de un número de 8 cifras y de una letra. La letra del DNI se obtiene a partir de los números como describen los pasos siguientes:

1. Calcula el resto de dividir el número del DNI entre 23.

2. El número obtenido está entre 0 y 22. Selecciona la letra asociada a dicho número en la siguiente tabla:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | R | W | A | G | M | Y | F | P | D | X | B |

| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|
| N | J | Z | S | Q | V | H | L | C | K | E |

***Ejemplo*** Aquí puedes ver el DNI de un amigo nuestro:


El número del DNI es 31415927 y el resto de dividir entre 23 es 20: $31415927 = 1365909 \times 23 + 20$; por tanto la letra que le corresponde según la tabla es "C".

***Cálculo de la letra del DNI*** Escribe una expresión que permita calcular la letra correspondiente a un determinado número de DNI. (Véase la pista 1.3a.)

***DNI correcto*** Escribe un expresión que a partir de un DNI completo, letra y número, indique si dicho DNI es correcto o no.

***Un poco de historia*** No son pocos los avatares que han engrendrado a Tux. En 1984, Richard Stallman inició el proyecto GNU (`http://www.gnu.org`) para construir un *sistema operativo libre*. Por *sistema operativo* se debe entender bastante más de lo que los fabricantes de software comercial suelen vender; incluye editores, compiladores, sistema de ventanas, juegos..., y en general todo lo que se necesita día a día para utilizar una computadora. Es *libre* porque se puede usar, modificar y redistribuir (con o sin modificacion) libremente.

En 1991, Linus Torvalds inició Linux: un proyecto para construir un núcleo de sistema operativo que fuera un clon de Unix (`http://www.linux.org`, `http://www.kernel.org`). En breve, y como reconocimiento a las herramientas del proyecto GNU que estaba usando, Linux pasó a ser software libre. La combinación GNU/Linux constituye el sistema operativo más robusto y eficiente, y con mayor presencia en internet.

Finalmente, Larry Ewing dibujó a Tux, que se ha convertido en el logotipo no oficial de Linux (`http://www.isc.tamu.edu/~lewing/linux`).

En casi todas las direcciones anteriores encontrarás mucha información sobre software libre, pero `www.linuxdoc.org` está especialmente dedicada a la documentación de todo lo relativo a Linux.

## 14 Cuadrados perfectos

34

Los cuadrados perfectos son los números 1, 4, 9, 16, ..., esto es, los cuadrados de los números naturales: $1^2$, $2^2$, $3^2$, $4^2$, ...

***Cuadrado perfecto*** Encuentra una expresión, o una secuencia sencilla de instrucciones, adecuada para averiguar si un número natural $m$ es un cuadrado perfecto, o sea, si es de la forma

$$m = n^2$$

para algún natural $n$.

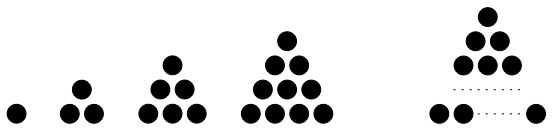
***Cuadrado perfecto previo*** Encuentra una expresión que, para un número $m$ entero, averigüe el número $n$ mayor posible pero que no supere a $m$, (o sea, $n \leq m$) y sea cuadrado perfecto.

***Bibliografía*** El libro *El diablo de los números, un libro para todos aquéllos que temen a las matemáticas* [Enz01] es ameno y contiene bellas ilustraciones. En él podrás encontrar curiosidades, definiciones e historias acerca de varias sucesiones de números, entre ellas los cuadrados perfectos, los números triangulares (ejercicio 1.5), la sucesión de Fibonacci...

## 15 Números triangulares

34

Considera la secuencia de los números triangulares: 1, 3, 6, 10..., cuyo nombre refleja su ley de formación:



$$1 \quad 3 \quad 6 \quad 10 \quad \cdots \quad \sum_{i=1}^{n} i \quad \cdots$$

***Identificación de números triangulares*** Escribe una expresión que indique si un número natural $t$ es triangular, esto es, si es de la forma:

$$t = \sum_{i=1}^{n} i$$

para algún natural $n$.

***Número triangular previo*** Encuentra una expresión que, para un número $n$ entero, averigüe el mayor número $m$ triangular que no supere a $n$.

## 16 Rectángulos

31

Un rectángulo se puede representar en un plano a partir de cuatro puntos. Como puedes observar en la figura 1.2, en este ejercicio vamos a trabajar con rectángulos cuyos lados son paralelos a los ejes de coordenadas.

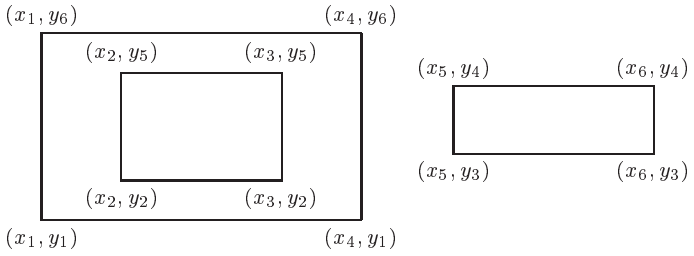



Figura 1.2: Representación de tres rectángulos

***Rectángulo*** Escribe una expresión que determine si dados cuatro puntos del plano, éstos pueden representar los vértices de un rectángulo. (Véanse las pistas 1.6a y 1.6b.)

***Inclusión de rectángulos*** Escribe una expresión que, dados los vértices de dos rectángulos, determine si el segundo está dentro del primero.

## 17 Movimientos en un tablero de ajedrez

31 36

Dados un tablero de ajedrez y las posiciones de dos casillas, queremos averiguar si se puede ir de una a otra con un movimiento de las siguientes figuras: caballo, alfil, torre, reina y rey. Para ello, vamos a suponer que la posición de una casilla viene dada por dos valores (`fila`, `columna`) en un tablero de $8 \times 8$. Se trata, por tanto, de escribir sendas expresiones que permitan determinar si el movimiento entre dos casillas es posible para cada una de las piezas. En la figura 1.3 se describen los movimientos correctos. El carácter "•" indica las casillas a las que se puede desplazar el rey desde su posición. El carácter "⋆" hace lo propio con el caballo. Las flechas indican la dirección correcta de desplazamiento (horizontal, vertical, diagonal) de la reina, el alfil y la torre dada su posición en el tablero. (Véase la pista 1.7a.)

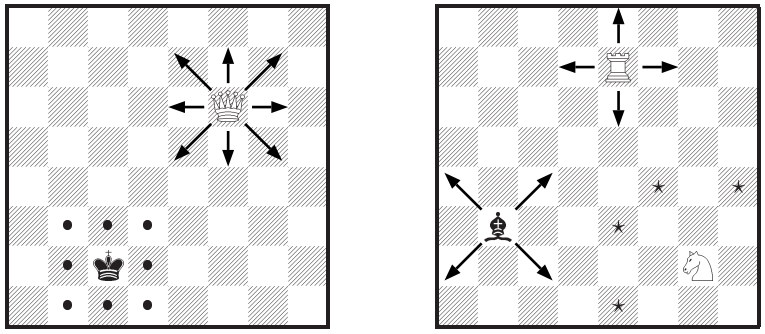


Figura 1.3: Descripción de los movimientos correctos de las piezas

***Un poco de historia*** El ajedrez es sin duda uno de los juegos de mesa más antiguos y famosos. Originario de la India, siglo IV a.C., se extendió a China y Persia adquiriendo con el tiempo diversas peculiaridades. En España lo introdujeron los árabes entre los siglos VIII y X. En esa misma época también se conocía en Escandinavia, probablemente llevado por los mongoles. Hasta el siglo XIX el ajedrez fue un juego principalmente aristocrático. Es a partir de su popularización y de la organización de torneos cuando el nivel del juego adquirió una gran sofisticación. La programación siempre ha estado ligada a los juegos, y en particular al del ajedrez debido a su complejidad. Existe una asociación de ajedrez computacional, *International Computer Chess Association*. Si te interesan las relaciones entre el ajedrez y la programación puedes encontrar mucha información en la página de Internet de dicha asociación: `http://www.dcs.qmul.ac.uk/~icca/`. Si quieres unas nociones introductorias al respecto, puedes consultar los capítulos relacionados con árboles de búsqueda en juegos con adversario en los libros de inteligencia artificial; un estudio más exhaustivo lo puedes encontrar en [Hei00].

## 18 División entera con redondeo

36

La división entera normal trunca el resultado, de forma que tanto 6/5 como 8/5 producen 1. En ciertas ocasiones es más interesante una división entera que redondee el resultado al entero más cercano, de forma que 8/5 valga 2. Escribe una expresión que calcule la división entera con redondeo de los enteros `a` y `b`.

## 19 División entera con préstamo

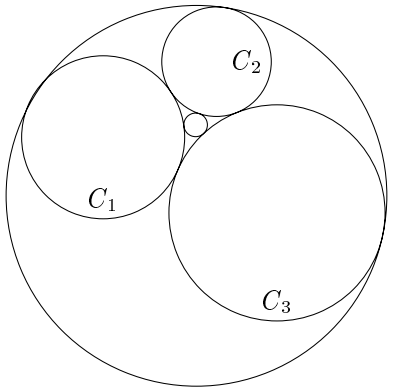
La división entera normal trunca el resultado, de forma que tanto 5/5 como 6/5 como 8/5 producen 1. En ciertas ocasiones es más interesante un división entera que, si no es exacta, devuelva el entero por encima. Con esta definición, 5/5 valdría 1, mientras que 6/5 o 8/5 valdrían 2. Porque todo se termina pagando, llamaremos a esta forma de división, división con préstamo. Escribe una expresión que calcule la división entera con préstamo entre `a` y `b`.

## 110 Cálculo de la división entera y resto entre reales

La división entera también se puede definir entre números reales. Si $a$ y $b$ son dos números reales positivos, la división entera de $a$ entre $b$ es el número entero $n$ que verifica que $bn \leq a < bn + b$. El resto de esta división es $a - bn$. Escribe expresiones para calcular la división entera entre números reales y su resto.

## 111 El beso exacto

Como puedes ver en el dibujo, dados tres círculos, $C_1$, $C_2$ y $C_3$, tangentes entre sí dos a dos, existen exactamente dos círculos que son tangentes a los tres anteriores:




Hay una fórmula que permite calcular el radio de estos círculos tangentes: si tenemos cuatro círculos $C_1$, $C_2$, $C_3$ y $C_4$, tangentes entre sí, con radios $r_1$, $r_2$, $r_3$ y $r_4$, respectivamente, entonces se verifica la fórmula

$$2(s_1^2 + s_2^2 + s_3^2 + s_4^2) = (s_1 + s_2 + s_3 + s_4)^2$$

donde $s_i = 1/r_i$. No hay magia alguna; de esta fórmula resultan dos soluciones porque es una ecuación de segundo grado. La solución positiva da el radio del círculo menor, y del valor absoluto de la negativa se obtiene el radio del mayor.

Utiliza esta propiedad para escribir un programa que, dados los radios de tres círculos tangentes entre sí, calcule el radio de los dos círculos que son tangentes a los tres anteriores.

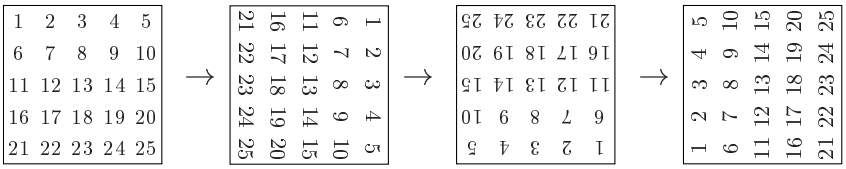
***Un poco de historia*** La fórmula que hemos utilizado ha sido descubierta y redescubierta varias veces a lo largo de la historia. Se cree que los griegos ya la conocían. René Descartes (1596–1650) la demostró. Actualmente se conoce como la fórmula de Soddy, en honor al químico Frederick Soddy (1877–1956) que la volvió a demostrar en 1936. Una curiosidad: Soddy obtuvo el premio Nobel de química en 1921 y fué quien acuñó el término *isótopo*.

Soddy *demostró* el teorema con un poema titulado *The Kiss Precise*. Te mostramos la segunda estrofa del poema, en la columna de la izquierda, y, a la derecha, una traducción que aparece en [Gar94].

| | |
|---|---|
| Four circles to the kissing come. | Cuatro círculos llegaron a besarse, |
| The smaller are the benter. | es el menor el más curvado. |
| The bend is just the inverse of | La curvatura no es sino la inversa |
| The distance form the center. | de la distancia desde el centro. |
| Though their intrigue left Euclid dumb | Aunque este enigma a Euclides asombrara |
| There's now no need for rule of thumb. | las reglas empíricas no son necesarias. |
| Since zero bend's a dead straight line | Como la recta tiene curvatura nula |
| And concave bends have minus sign, | y las curvas cóncavas tienen signo menos, |
| The sum of the squares of all four bends | la suma de los cuadrados de las cuatro curvaturas |
| Is half the square of their sum. | es igual a la mitad del cuadrado de su suma. |

## 1.12 ¿Pueden las matrices dar volteretas?

Pues parece que sí:



***Voltereta*** Suponiendo que la matriz es cuadrada y su tamaño es $N$, se pide una función que transforme un número en el que lo desplazará cuando la matriz dé una voltereta. Por ejemplo, para $N = 12$, la secuencia de números que ocupan la casilla $(3, 2)$ es la siguiente:

$$12 \to 8 \to 14 \to 18$$

Por tanto, la función que buscamos deberá convertir el 12 en 8, el 8 en 14, etc.

***Vuelta completa*** Comprueba que, aplicando la función cuatro veces sobre cualquier elemento de la matriz, dicho elemento no varía.

## 1.13 Todas la funciones booleanas de dos argumentos

31

Una función booleana de dos argumentos (por ejemplo la conjunción lógica, `&&`) transforma un par de valores booleanos en un valor booleano. Comprueba que hay 16 funciones distintas de este tipo. Escribe en C++ la expresión que define a cada una de ellas. (Véase la pista 1.13a.)

## 1.14 Fichas de dominó

31 38

Las fichas del dominó se pueden enumerar de forma ordenada como se muestra en la figura 1.4.

### 1.14.1 Número de una ficha

Escribe una expresión tal que dadas dos variables `menor` y `mayor`, conteniendo los valores menor y mayor de una ficha de dominó respectivamente, calcule el número asociado según la figura anterior.

***Ejemplo*** Dada una ficha con números 1 y 5, la expresión debe calcular el número de orden 11. Dada una ficha con números 4 y 6, la expresión debe calcular el número de orden 24.

### 1.14.2 Ficha de un número

Escribe ahora las expresiones inversas a ésta, es decir, las que obtienen el valor menor y el valor mayor de una ficha a partir del número de ficha $n$.

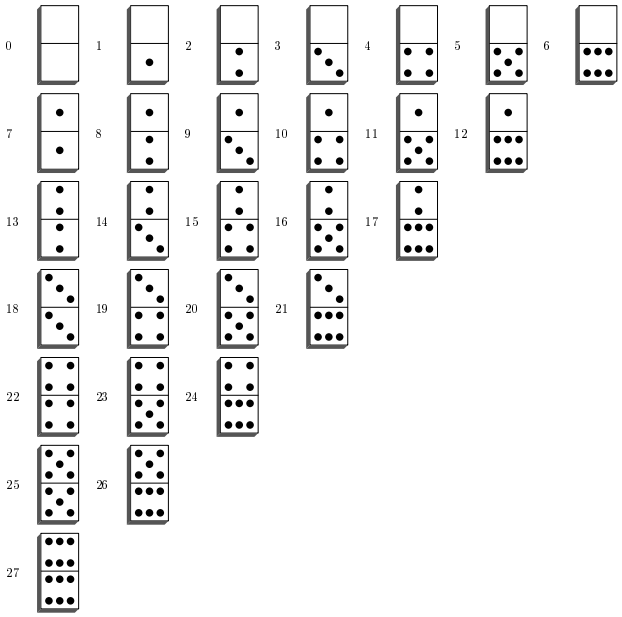


Figura 1.4: Numeración de las fichas de un juego de dominó

***Ejemplo*** Dado el número 7, la expresión debe calcular los números de la la ficha correspondiente que son el 1 y el 1. Dado el número 19 debe calcular el 3 y el 4. (Véase la pista 1.14a.)

## 1.15 Expresiones relacionadas con una formación rectangular de números

39

Considera la siguiente formación de números

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{array}$$

Para formaciones de este estilo con $N$ columnas, escribe expresiones para los siguientes cálculos y situaciones:

***Cálculo del número*** El número que aparece en la fila $i$ y columna $j$ ($i \geq 1$ y $1 \leq j \leq N$).

***Fila de un número*** La fila que ocupa un número $p \geq 1$.

***Columna de un número*** La columna que ocupa un número $p \geq 1$.

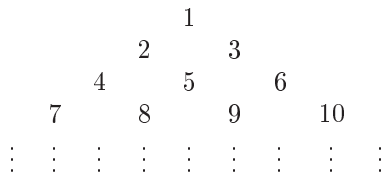
***¿Están en la misma fila?*** Cuándo dos números $p$ y $q$ ($p, q \geq 1$) están en la misma fila.

***¿Están en la misma columna?*** Cuándo dos números $p$ y $q$ ($p, q \geq 1$) están en la misma columna.

***¿Están en la misma diagonal?*** Cuándo dos números $p$ y $q$ ($p, q \geq 1$) están en la misma diagonal.

## 1.16 Expresiones relacionadas con una formación triangular de números

Considera la siguiente formación de números:



***Número en posición*** Calcula el número que aparece en la posición $j$ de la fila $i$ ($i \geq 1$ y ¿qué condición debe cumplir $j$?).

***Fila de un número*** Calcula la fila que ocupa un número $p$.

***¿Están en la misma fila?*** Indica si dos números $p$ y $q$ están en la misma fila.

***¿Están en la misma diagonal?*** Indica si dos números $p$ y $q$ están en la misma diagonal.

## 1.17 Reflexión de los bits de una palabra de anchura fija

La *reflexión de los bits* de un entero es otro entero que tiene los bits reflejados en un espejo: el bit menos significativo del primero pasa a ser el bit más significativo del segundo... y el bit más significativo del primero pasa a ser el menos significativo del segundo. Formalmente, la reflexión del entero $\sum_{i=0}^{k-1} a_i 2^i$ es el entero $\sum_{i=0}^{k-1} a_i 2^{k-i-1}$. Gráficamente, la reflexión del entero

| $a_{k-1}$ | $a_{k-2}$ | $\cdots$ | $a_1$ | $a_0$ |
|---|---|---|---|---|

es este otro:

| $a_0$ | $a_1$ | $\cdots$ | $a_{k-2}$ | $a_{k-1}$ |
|---|---|---|---|---|

Supongamos que en `bits` tenemos un entero sin signo de 32 bits. Escribe una secuencia de asignaciones que deje en esa misma variable su reflexión. Intenta que esa secuencia sea lo más corta posible. (Véase la pista 1.17a.)

***Bibliografía*** Este problema está extraído del libro *Exploiting 64-Bit Parallelism* [Gut00], donde se describen otras operaciones sobre bits. Como indica su título, el código que allí se describe está pensado para una arquitectura con enteros de 64 bits.

## 1.18 Organización de torneo

Un grupo de personas está organizando un torneo de ajedrez por eliminatorias. En cada ronda los participantes se enfrentan uno contra uno, y el vencedor de cada enfrentamiento pasa a la siguiente ronda hasta que sólo queda un ganador. Al organizar las rondas surge una curiosa situación: dependiendo del número de jugadores, en algunas rondas habrá un jugador que no tendrá rival y pasará directamente a la ronda siguiente.

Escribe un programa que, a partir del número de jugadores de un torneo por eliminatorias, nos informe de las rondas en las que algún jugador quedará liberado de jugar.

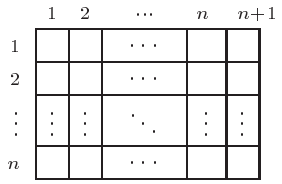
***Ejemplo*** Supongamos que queremos organizar un torneo con diez jugadores. En la primera ronda podremos enfrentar a todos ellos pasando cinco jugadores a la siguiente. En la segunda ronda uno de los jugadores no tiene contrincante y pasa automáticamente a la tercera; de los otros cuatro, pasan dos y por tanto tenemos tres jugadores para la tercera ronda. Nuevamente uno de ellos pasa directamente a la final. La tabla siguiente resume esta información:

| Ronda del torneo | Número de jugadores | Jugador libre |
|---|---|---|
| 1 | 10 | No |
| 2 | 5 | Sí |
| 3 | 3 | Sí |
| 4 | 2 | No |

***Para este enunciado*** Consulta la pista 1.18a.

## 19 $n$-goros

Un tablero $n$-goro es un tablero con $n \times (n+1)$ casillas. Nos referiremos a las casillas de este tablero mediante las coordenadas $(i, j)$, en donde $i$ es la fila (contadas desde 1 y de arriba a abajo) y $j$ es la columna (contadas desde 1 y de izquierda a derecha). Como ejemplo:



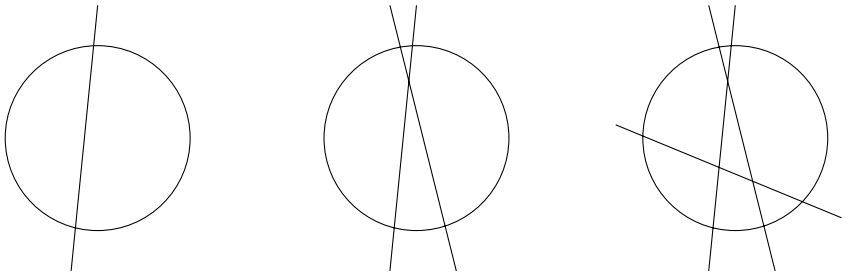

Una propiedad interesante es que se pueden visitar todas sus casillas haciendo el siguiente recorrido por diagonales. Empezamos por la casilla $(1, 1)$ y recorremos la diagonal principal hacia la derecha y hacia abajo hasta llegar a la casilla $(n, n)$. La siguiente casilla a visitar sería la $(n+1, n+1)$ que no existe porque nos saldríamos del tablero por abajo. En estos casos siempre se pasa a la casilla equivalente en la primera fila, es decir, la $(1, n+1)$. Ahora seguimos moviendonos hacia la derecha y hacia abajo. Pero la siguiente casilla sería la $(2, n+2)$ que no existe porque nos hemos salido por la derecha. En estos casos se continúa por la casilla equivalente de la primera columna, es decir, la $(2, 1)$. De nuevo nos movemos hacia la derecha y hacia abajo, hasta alcanzar la casilla $(n, n-1)$. La siguiente casilla sería la $(n+1, n)$, pero como nos saldríamos por abajo pasamos a la casilla equivalente de la primera fila $(1, n)$. Si se continúa con este proceso se termina visitando todas las casillas del tablero goro. Como ejemplo este tablero de $4 \times 5$ casillas, rellenas con el número que indica cuando se visitó:

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

Escribe una expresión para calcular el valor que este recorrido va a dejar en una casilla arbitraria $(i, j)$.

## 1.20 Cortes en una tarta

Observa los siguientes dibujos:



Piensa un poco y escribe una expresión que nos diga el *máximo* número de porciones en que podemos dividir una tarta al realizar $n$ cortes. (Véase la pista 1.20a.)

## 1.21 Número de bits a 1

Aunque pueda parecer un problema inútil, no pocas veces es necesario contar rápidamente cuántos bits de la representación binaria de un entero están a 1 (y por ende, cuántos están a 0).

Supongamos que en `bits` tenemos un entero sin signo de 32 bits. Escribe una secuencia de asignaciones que deje en esa misma variable la cantidad de bits que tenía a 1. Busca una solución lo más eficiente posible.

***Bibliografía*** Para saber más acerca de operaciones sobre bits consulta la bibliografía del ejercicio 1.17.

**1.3a.** Utiliza la cadena `"TRWAGMYFPDXBNJZSQVHLCKE"` de forma adecuada.

**1.6a.** La distancia entre los puntos $(x_1, y_1)$ y $(x_2, y_2)$ vale $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

**1.6b.** Recuerda que en un rectángulo los lados deben ser de igual longitud dos a dos.

**1.7a.** Para decidir si podemos pasar de una casilla del tablero, $(f_1, c_1)$, a otra casilla, $(f_2, c_2)$, con una determinada pieza, lo más importante es encontrar la relación que han de verificar las componentes de las casillas origen y destino entre sí.

Por ejemplo, desde la casilla $(f_1, c_1)$ con una torre, podemos ir a cualquier casilla que esté en la fila $f_1$ o en la columna $c_1$.

**1.13a.** Una manera muy conveniente de representar las funciones booleanas es mediante *tablas de verdad*. En la primera columna aparecen los posibles valores del primero de los argumentos. En la primera fila aparecen los posibles valores del segundo de los argumentos. El punto donde se unen una fila y una columna indica el valor de la función que se está definiendo si se aplica a argumentos que tienen como valor el primero de la fila y de la columna, respectivamente.

La siguiente figura muestra la tabla de verdad para el operador lógico de conjunción:

| conjunción | verdadero | falso |
|---|---|---|
| verdadero | verdadero | falso |
| falso | falso | falso |

Esta tabla aglutina todos los resultados posibles para la función booleana de conjunción cuando se le aplica a dos valores booleanos. Por ejemplo, la conjunción de verdadero y falso da como resultado falso. Para describir fácilmente todos los operadores booleanos utiliza la propiedad de que cada operador tiene una tabla de verdad diferente.

**1.14a.** Este problema es parecido al de encontrar el mayor cuadrado perfecto posible (véase el ejercicio 1.4) sin superar un entero dado: ahora, a partir de una ficha $t$, deseamos encontrar el número de la primera ficha de su fila; esto es, la cantidad $\frac{15m-m^2}{2}$ entera, mayor posible, que no supere a $t$.

**1.17a.** La idea clave es conseguir el intercambio entre dos mitades; gráficamente, todo consiste en saber pasar de

| $a_{k-1}$ | $a_{k-2}$ | $\cdots$ | $a_{k/2+1}$ | $a_{k/2}$ | $a_{k/2-1}$ | $a_{k/2-2}$ | $\cdots$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|

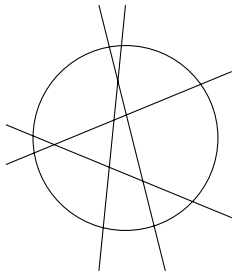
a

| $a_{k/2-1}$ | $a_{k/2-2}$ | $\cdots$ | $a_1$ | $a_0$ | $a_{k-1}$ | $a_{k-2}$ | $\cdots$ | $a_{k/2+1}$ | $a_{k/2}$ |
|---|---|---|---|---|---|---|---|---|---|

**1.18a.** Si tenemos $n$ jugadores y consideramos el menor $k$ tal que $2^k \geq n$, el número total de rondas en las que algún jugador no juega coincide con el número de unos que aparecen en la expresión binaria del número $2^k - n$. Pero además, el orden en el que aparecen los unos y ceros en dicha expresión binaria de $2^k - n$ nos informa de las rondas en las que un jugador queda libre y en las que todos los jugadores están enfrentados.

Por ejemplo, para un torneo de diez jugadores, $k = 4$ es el menor $k$ tal que $2^k \geq n$. La expresión binaria de $2^4 - 10$ es 110. Si la leemos de derecha a izquierda, lo que podemos interpretar es que en la primera ronda todos los jugadores tienen rival, por el contrario, en la segunda y en la tercera ronda sí que habrá un jugador que pasará directamente a la ronda siguiente.

**1.20a.** Esta figura tiene el máximo número posible de porciones para cuatro cortes.



## 11 Espacios de color

19

Todos estas transformaciones dan lugar a programas con una estructura similar:

```
int main() {
  ⟨Lectura del modelo de color original⟩
  ⟨Cálculo de las componentes del modelo de color destino⟩
  ⟨Escritura de las componentes del modelo de color destino⟩
}
```

Supongamos que el modelo original es RGB y el destino es YIQ. La lectura

```
double r, v, a;
cout << "Dame rojo, verde y azul:  " << flush;
cin >> r >> v >> a;
```

y la escritura

```
cout << "YIQ = " << y << " " << i << " " << q << endl;
```

son obvias. Para calcular las componentes simplemente hay que dejarse llevar por las expresiones matemáticas:

```
double const y = 0.299 * r + 0.587 * v + 0.114 * a;
double const i = 0.596 * r - 0.275 * v - 0.321 * a;
double const q = 0.212 * r - 0.528 * v + 0.311 * a;
```

## 12 Ser o no ser... triángulo

20

Supongamos que ya se han declarado las tres variables reales que representan las longitudes de los lados, tal vez así:

```
double a, b, c;
```

Una vez leídos sus valores de la entrada estándar,

```
cin >> a >> b >> c;
```

podemos resolver los apartados.

***¿Es un triángulo?*** Estas medidas pueden representar las longitudes de los lados de un triángulo si la suma de dos de ellas es mayor que la otra. El resultado de la expresión que determina este aspecto lo almacenamos en la constante `esTriangulo` porque lo vamos a utilizar en el resto de los apartados:

```
bool const esTriangulo = (a + b > c) && (a + c > b) && (c + b > a);
```

***¿Es escaleno?*** Pueden corresponder a las longitudes de los lados de un triángulo escaleno precisamente si los tres lados son distintos entre sí:

```
bool const esEscaleno = esTriangulo && (a != b) && (b != c) && (a != c);
```

***¿Es equilátero?*** Pueden corresponder a las longitudes de los lados de un triángulo equilátero si los tres lados son iguales:

```
bool const esEquilatero = esTriangulo && (a == b) && (b == c);
```

***¿Es isósceles?*** Pueden corresponder a las longitudes de los lados de un triángulo isósceles si al menos dos lados son iguales:

```
bool const esIsosceles = esTriangulo && (((a == b) && (a != c)) ||
                                         ((b == c) && (b != a)) ||
                                         ((a == c) && (a != b)));
```

O por eliminación:

```
bool const esIsosceles = esTriangulo && !esEscaleno && !esEquilatero;
```

## 13 Cálculo y verificación de la letra del D.N.I.

20 31

***Cálculo de la letra del DNI*** Para el cálculo de la letra del DNI seguimos los pasos que se indican en el enunciado del ejercicio. El primer paso en el cálculo es trivial ya que basta utilizar el operador módulo (%); el segundo no es mucho más complicado si se precomputa en forma de tabla la correspondencia entre números y letras. En este caso, basta la siguiente definición

```
string const tablaDNI = "TRWAGMYFPDXBNJZSQVHLCKE";
```

para tener un punto al que recurrir a la hora de hacer el paso entre resto y letra:

```
cout << numeroDNI << '-' << tablaDNI[numeroDNI % 23] << endl;
```

***DNI correcto*** Una expresión para saber si es correcto un DNI completo (letra y número) es la siguiente:

```
tablaDNI[numeroDNI % 23] == letraDNI
```

## 14 Cuadrados perfectos

21

***Cuadrado perfecto*** Empezamos por calcular la raíz de $m$, redondeada al entero más próximo $n$, mediante la siguiente instrucción:

```
const int n = (int)(rint(sqrt(m)));
```

Se ha de suponer el requisito previo de que $m$ es positivo: el cálculo de la raíz cuadrada exige esta condición.

Ahora, si $m$ es un cuadrado perfecto, este redondeo no tendrá mayor efecto que convertir en entera una cantidad real, sin alterar su valor; por tanto, se tendrá $n^2 = m$. De lo contrario, el redondeo cambiará el valor de la raíz, de forma que al elevar $n^2$ no se llegará a $m$, sino a otro número distinto. Por tanto, después de la instrucción anterior, la expresión lógica `n*n == m` indica si es $m$ un cuadrado perfecto o no.

## 15 Números triangulares

22

***Identificación de números triangulares*** Con la definición de número triangular,

$$t = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

se obtiene una ecuación de segundo grado en $n$,

$$n^2 + n - 2t = 0$$

de donde, despreciando la solución negativa, tenemos:

$$n = \frac{-1 + \sqrt{1 + 8t}}{2}.$$

Luego, si $t$ es triangular, $n$ debe ser natural y, por tanto, deben darse dos circunstancias:

- En primer lugar, $\sqrt{1 + 8t}$ tiene que ser entero; es decir, $1 + 8t$ tiene que ser un cuadrado perfecto.
- Además, para que el cociente de la división entre 2 sea entero, $-1 + \sqrt{1 + 8t}$ debe ser par y por tanto $\sqrt{1 + 8t}$ impar.

  Pero esta segunda condición queda garantizada por la anterior: al ser $t$ entero, $1 + 8t$ es impar y, si su raíz es exacta (propiedad anterior), también será impar.

Así pues, basta con exigir que $1 + 8t$ sea un cuadrado perfecto, para lo cual podemos usar la expresión del ejercicio 1.4. En definitiva, después de las instrucciones siguientes,

```
int const m = 1 + 8*t;
int const n = (int)(rint(sqrt(m)));
```

la expresión `n*n == m` indica si $t$ es un número triangular.

Es necesario exigir el requisito de que sea $t \geq 0$, previamente a las dos instrucciones anteriores; de lo contrario, la cantidad $1 + 8t$ sería negativa y el cálculo de su raíz daría lugar a un error.

***Número triangular previo*** Buscar el mayor número triangular $m \leq n$ equivale a encontrar el mayor $k \geq 0$ tal que

$$m = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} \leq n.$$

Desarrollando la última desigualdad obtenemos

$$k^2 + k - 2n \leq 0.$$

Resolviendo la ecuación $x^2 + x - 2n = 0$, al igual que en el apartado anterior, obtenemos dos soluciones, una positiva y otra negativa; la negativa no nos sirve y nos quedamos con la positiva:

$$x_0 = \frac{-1 + \sqrt{1 + 8n}}{2}.$$

Puesto que la función $x^2 + x - 2n$ es creciente para $x > 0$, el valor de $k$ buscado es $\lfloor x_0 \rfloor$. Todo esto en C++ se puede expresar así:

```
int const k = (int)((-1 + sqrt(1 + 8*n)) / 2);
```

Y el valor buscado es:

```
int const m = (k * (k+1)) / 2;
```

## 17 Movimientos en un tablero de ajedrez

Suponemos dos casillas distintas del tablero de ajedrez representadas por `(f1, c1)` y `(f2, c2)`.

***Movimiento de la torre*** El desplazamiento de una torre entre dos casillas será posible si

```
(f1 == f2) || (c1 == c2)
```

es decir, si están en la misma horizontal (fila) o en la misma vertical (columna).

***Movimiento del alfil*** En el caso de un alfil será necesario que ambas casillas se encuentren en la misma diagonal. Dos casillas están en la misma diagonal de izquierda a derecha si `c1 - f1 == c2 - f2` y en la misma diagonal de derecha a izquierda si `c1 + f1 == c2 + f2`; la expresión buscada será la disyunción de las dos anteriores que se puede abreviar en la siguiente:

```
abs(f1 - f2) == abs(c1 - c2)
```

***Movimiento del caballo*** El desplazamiento de un caballo supone un salto de dos casillas en horizontal y una en vertical, o bien un salto de dos casillas en vertical y una en horizontal. Utilizamos los cuadrados del número de casillas recorridas tanto en horizontal como en vertical para poder despreocuparnos del sentido del desplazamiento; y llegamos a la siguiente expresión:

```
(f1 - f2)*(f1 - f2) + (c1 - c2)*(c1 - c2) == 5
```

También se puede comprobar de esta otra forma:

```
abs((f1 - f2) * (c1 - c2)) == 2
```

¿Podrías explicar por qué?

***Movimiento de la reina*** Los movimientos de la reina son los de la torre más los del alfil:

```
(f1 == f2) || (c1 == c2) || (abs(f1 - f2) == abs(c1-c2))
```

***Movimiento del rey*** El rey puede desplazarse a una casilla contigua a la que ocupa con un movimiento horizontal, vertical o diagonal. Por lo tanto exige que:

```
(abs(f1 - f2) <= 1) && (abs(c1 - c2) <= 1)
```

## 18 División entera con redondeo

La división entera en C++, como en todo lenguaje de programación, trunca hacia abajo. Eso significa que, si `a` y `b` son enteros positivos, la operación `a/b` es el entero $q$ tal que $q\mathtt{b} \leq \mathtt{a} < (q+1)\mathtt{b}$. Por otra parte, la división con redondeo también se puede expresar con una fórmula similar a la anterior, ya que es el entero $r$ tal que $(r-1/2)\mathtt{b} \leq \mathtt{a} < (r+1/2)\mathtt{b}$. Estas desigualdades son equivalentes a $r\mathtt{b} - \mathtt{b}/2 \leq \mathtt{a} < r\mathtt{b} + \mathtt{b}/2$. Sumando $\mathtt{b}/2$ a todos los términos de esta ecuación queda $r\mathtt{b} \leq \mathtt{a}+\mathtt{b}/2 < (r+1)\mathtt{b}$, es decir, una inecuación con la misma estructura que la que caracterizaba a la división entera. Pero antes de traducir estas desigualdades a código hay que meditar sobre la división $\mathtt{b}/2$ que es sólo una forma abreviada de escribir $(1/2) \times \mathtt{b}$. Obviamente, la división en $1/2$ es real y, por ende, la división $\mathtt{b}/2$ es real. Afortunadamente, si $r$ es un real y $n$ un entero tales que $n \leq r$, también tenemos que $n \leq \lfloor r \rfloor$. Por otra parte, también es obvio que si $r < n$ entonces $\lfloor r \rfloor < n$. Usando estas propiedades, es seguro que $r\mathtt{b} \leq \mathtt{a} + \lfloor \mathtt{b}/2 \rfloor < (r+1)\mathtt{b}$. Estas ecuaciones ya sólo involucran enteros y se traducen en el siguiente código para calcular la división entera con redondeo entre `a` y `b`:

```
(a + (b/2)) / b
```

## 19 División entera con préstamo

24

La división entera en C++, como en todo lenguaje de programación trunca hacia abajo. Eso significa que si `a` y `b` son enteros positivos, la operación `a/b` es el entero $q$ tal que $q\texttt{b} \leq \texttt{a} < (q+1)\texttt{b}$. La división con préstamo se puede expresar con una fórmula similar a la anterior, verbigracia: es el entero $p$ tal que $(p-1)\texttt{b} < \texttt{a} \leq p\texttt{b}$. En esta ecuación, las desigualdades no ocurren de la misma forma que en la de la división entera. Pero todos los números son enteros y las desigualdades entre enteros cumplen propiedades curiosas. Por ejemplo, si $m$ y $n$ son enteros, afirmar que $m < n$ es igual que afirmar que $m+1 \leq n$; igualmente $m \leq n$ es equivalente a $m < n+1$. Con estas observaciones, reescribimos la ecuación de la división con préstamo de la siguiente forma: $(p-1)\texttt{b}+1 \leq \texttt{a} < p\texttt{b}+1$. Sumando $\texttt{b}-1$ a todos los términos de esta ecuación queda $p\texttt{b} < \texttt{a}+(\texttt{b}-1) \leq (p+1)\texttt{b}$. Con lo que ya tenemos la forma de calcular la división con préstamo entre `a` y `b` con una división entera:

```
(a + (b-1)) / b
```

## 111 El beso exacto

24

Habrá que empezar echando unas pocas cuentas. Como indica el enunciado, supondremos dados los círculos $C_1$, $C_2$ y $C_3$. Conoceremos sus radios y habrá que averiguar el radio de un cuarto círculo tangente. Por tanto, hay que despejar $s_4$ en la fórmula de Soddy. Para simplificar el álgebra, definimos $a = s_1 + s_2 + s_3$ y $b = s_1^2 + s_2^2 + s_3^2$. La fórmula de Soddy queda

$$2(b + s_4^2) = (a + s_4)^2.$$

Desarrollando obtenemos la siguiente ecuación de segundo grado:

$$s_4^2 - 2as_4 + (2b - a^2) = 0.$$

Aunque andábamos buscando el radio de un círculo tangente, esta ecuación nos da dos soluciones y, sin más trabajo, ya tenemos los radios de los dos círculos tangentes:

$$s_4 = \frac{2a \pm \sqrt{4a^2 - 4(2b - a^2)}}{2} = a \pm \sqrt{2a^2 - 2b}.$$

Generalmente, hay que tener cuidado a la hora de resolver ecuaciones de segundo grado en un programa porque el discriminante puede ser negativo y, por ende, tener soluciones complejas. En este caso, como $r_i$ es siempre mayor que cero, se tiene que $2a^2 - 2b > 0$. Podemos calcular sin temor, tal y como hacemos en el siguiente código:

```
double const s1 = 1/r1;
double const s2 = 1/r2;
double const s3 = 1/r3;
double const a = s1 + s2 + s3;
double const b = s1*s1 + s2*s2 + s3*s3;
double const raiz = sqrt(2*a*a - 2*b);
cout << "Los radios de los círculos tangentes son "
     << 1/(a+raiz)
     << " y "
     << fabs(1/(a-raiz))
     << endl;
```

### 1.14.1 Número de una ficha

Como puede observarse en la figura 1.4, el valor menor de una ficha determina la fila, mientras que la diferencia entre ambos indica la columna. Así pues, habrá que comenzar determinando el número que corresponde a la primera ficha de cada fila. Éste aparece en la siguiente tabla:

| valor menor | primer valor de la fila |
|---|---|
| 0 | 0 |
| 1 | 7 |
| 2 | 7+6 |
| 3 | 7+6+5 |
| 4 | 7+6+5+4 |
| 5 | 7+6+5+4+3 |
| 6 | 7+6+5+4+3+2 |

Observamos que el número asociado a la primera ficha de cada fila equivale a la suma de una sucesión aritmética. En particular, el primer valor de la fila $m$ es $\sum_{i=7-m+1}^{7} i$, que se puede calcular fácilmente:

$$\frac{(7-m+1\ +\ 7)m}{2} = \frac{m(15-m)}{2} = \frac{15m-m^2}{2}.$$

Y sólo queda por determinar el desplazamiento dentro de la fila, que será función de los dos valores de la ficha. La relación que guardan los dos valores de la ficha con el desplazamiento dentro de la fila es la siguiente,

$$desplazamiento = mayor - menor.$$

Por lo tanto, la expresión pedida será la suma de las dos expresiones obtenidas,

$$\frac{15menor - menor^2}{2} + mayor - menor = \frac{13menor - menor^2}{2} + mayor \tag{1}$$

lo que se puede escribir sencillamente:

```
int const ficha = (13*menor - menor*menor) / 2 + mayor;
```

### 1.14.2 Ficha de un número

Calculemos ahora las expresiones para realizar las operaciones inversas. Dado un número de ficha $n$ hemos de calcular el valor menor y el mayor. En primer lugar calculemos el menor; debemos encontrar el mayor $m$ que verifique esta igualdad,

$$\frac{15m-m^2}{2} \leq n$$

lo que implica solucionar la ecuación siguiente en $\mathbb{R}$,

$$m^2 - 15m - 2n = 0$$

y quedarnos con la parte entera de dicha solución. Al ser una ecuación de segundo grado tiene dos soluciones:

$$\frac{15+\sqrt{15^2-8n}}{2} \quad \text{y} \quad \frac{15-\sqrt{15^2-8n}}{2}.$$

Por un lado sabemos que $n < 27$, con lo que

$$15^2 - 8n > 0$$

de forma que ambas soluciones son reales. Pero, por otra parte, la primera solución es mayor que 7, que no es un valor aceptable para nuestro problema, donde sólo son válidos los enteros menores o iguales que 6. Por consiguiente, el valor buscado será la parte entera de la segunda solución, que en C++ se puede escribir así:

```
int const menor = (int)((15 - sqrt(15*15 - 8*ficha))/2);
```

Una vez calculado el valor menor, el mayor es más sencillo, puesto que simplemente tenemos que calcular la inversa de la fórmula (1) del apartado 1.14.1, que en C++ se puede escribir así:

```
int const mayor = ficha - (13*menor - menor*menor)/2;
```

## 1.15 Expresiones relacionadas con una formación rectangular de números 26

***Cálculo del número*** El primer número de la fila $i$ es $N(i-1)+1$, el segundo será $N(i-1)+2$, y el $j$-ésimo será $N(i-1)+j$. El número $p$ buscado se calculará de la siguiente forma:

```
int const p = j + N*(i-1);
```

***Fila de un número*** Como ya hemos visto, los elementos en la fila $i$ son de la forma $N(i-1)+j$, por lo que para calcular la fila será necesario calcular el cociente de la división entre $N(i-1)+j$ y $N$. Pero como $1 \leq j \leq N$, esta división daría un resultado incorrecto cuando $j = N$, restaremos 1 antes de hacer la división; para calcular la fila será necesario por tanto sumar 1:

```
int const filaP = 1 + (p-1)/N;
```

***Columna de un número*** Puesto que los elementos en la columna $j$ son de la forma $N*(i-1)+j$, para calcular la columna será necesarior recurrir a la operación de resto. Si calculásemos directamente el módulo $N$ a ese número, nos daría como resultado una columna entre 0 y $N-1$, donde el valor 0 se correspondería con la columna $N$. Para evitar ese problema, podemos restar uno antes de hacer la operación de módulo y, después, volver a sumar 1, obteniendo así directamente una columna entre 1 y $N$.

```
int const columnaP = 1 + (p-1)%N;
```

***¿Están en la misma fila?*** Teniendo en cuenta el cálculo de la fila, la solución es

```
bool const mismaFila = filaP == filaQ;
```

siendo `filaP` el calculado antes, y `filaQ` se hallaría análogamente.

***¿Están en la misma columna?*** Teniendo en cuenta el cálculo de la columna de un número, la expresión buscada es

```
bool const mismaColumna = columnaP == columnaQ;
```

siendo `columnaP` el calculado antes, y `columnaQ` se hallaría análogamente.

***¿Están en la misma diagonal?*** Procederemos por partes. En primer lugar, dos números estarán en la misma diagonal de izquierda a derecha si la diferencia entre la fila y la columna coincide tanto en $p$ como en $q$; es decir,

```
bool const mismaDiagonalIzDer = (filaP-columnaP) == (filaQ-columnaQ);
```

siendo `columnaP`, `columnaQ`, `filaP` y `filaQ` las calculadas en los apartados anteriores. Análogamente, dos números estarán en la misma diagonal de derecha a izquierda si coincide la suma de la fila y la columna tanto en $p$ como en $q$:

```
bool const mismaDiagonalDerIz = (filaP+columnaP) == (filaQ+columnaQ);
```

Por último, los números están en la misma diagonal si están en una de las dos diagonales anteriores:

```
bool const mismaDiagonal = mismaDiagonalIzDer || mismaDiagonalDerIz;
```

## 116 Expresiones relacionadas con una formación triangular de números

27

***Número en posición*** En primer lugar, observemos que en la fila $n$ existen exactamente $n$ números por lo que $j$ deberá cumplir esta acotación:

$$1 \leq j \leq i$$

El último número de la fila $n$ es el $n$-ésimo número triangular, a saber $\frac{n(n+1)}{2}$ (véase el ejercicio 1.5), por lo que el número buscado será $j + \frac{(i-1)i}{2}$. Además, todas las divisiones que aparecen tienen resultado entero. Por tanto el número $p$ se calcula así:

```
int const p = j + ((i-1)*i)/2;
```

***Fila de un número*** Puesto que el último número de la fila $n$ es el $n$-ésimo número triangular y todo número natural está entre dos números triangulares, será necesario calcular el único número $n$ de forma que

$$\frac{(n-1)n}{2} < p \leq \frac{n(n+1)}{2}$$

de lo que obtenemos lo siguiente:

$$\frac{(n-1)^2}{2} < p < \frac{(n+1)^2}{2}, \quad (n-1)^2 < 2p < (n+1)^2, \quad (n-1) \leq \lfloor\sqrt{2p}\rfloor < n+1.$$

Llamando $k = \lfloor\sqrt{2p}\rfloor$ tenemos que $n = k+1$ o $n = k$. ¿Cómo resolvemos esa disyunción? Basta con fijarnos en lo siguiente:

$$\begin{array}{ll} \frac{k(k-1)}{2} < p \leq \frac{k(k+1)}{2} & \text{si } n = k \\ \frac{k(k+1)}{2} < p \leq \frac{(k+1)(k+1)}{2} & \text{si } n = k+1 \end{array}$$

Entonces podemos asegurar que $n = k+1$ si $k(k+1) < 2p$ y $n = k$ en caso contrario. Por tanto la solución será ésta:

```
int const k = (int)(sqrt(2*p));
int const fila = k*(k+1) < 2*p ?  k+1 :  k;
```

***¿Están en la misma fila?*** Para ver si los elementos $p$ y $q$ están en la misma fila, habrá que tomar la expresión del apartado anterior para cada número y comprobar su igualdad:

```
int const filaP = ⟨fila de p⟩;
int const filaQ = ⟨fila de q⟩;
bool const mismaFila = (filaP == filaQ);
```

***¿Están en la misma diagonal?*** Procederemos por partes: primero escribiremos expresiones para ver si dos números están en la misma diagonal de izquierda a derecha, despúes haremos lo mismo para la otra diagonal y, por último, daremos solución al apartado combinando las dos soluciones anteriores.

Para ver si los dos números están en la misma diagonal de izquierda a derecha conviene distribuir los elementos de otra forma:

$$\begin{array}{cccc} 1 & & & \\ 2 & 3 & & \\ 4 & 5 & 6 & \\ 7 & 8 & 9 & 10 \end{array}$$

Las diagonales de izquierda a derecha según la distribución original siguen siendo diagonales según la distribución actual; salvo que ahora se observa que dos elementos están la misma diagonal si la diferencia entre filas coincide con la diferencia entre columnas (en la nueva disposición). Si `filaP` es la fila del número $p$, el número triangular anterior a $p$ será $\frac{\texttt{filaP}(\texttt{filaP}-1)}{2}$ y por tanto la columna de $p$ será $p - \frac{\texttt{filaP}(\texttt{filaP}-1)}{2}$. Para ver si dos números $p$ y $q$ están en la misma diagonal de izquierda a derecha bastará con la siguiente secuencia de asignaciones,

```
int const columnaP = p - (filaP*(filaP-1))/2;
int const columnaQ = q - (filaQ*(filaQ-1))/2;
bool const mismaDiagonalIzDer = ((columnaP-filaP) == (columnaQ-filaQ));
```

donde las cantidades `filaP` y `filaQ` son las calculadas en el apartado anterior. Para determinar si dos elementos están en la otra diagonal observamos que, si distribuimos los elementos como antes, las diagonales de derecha a izquierda se transforman en una columna, y basta entonces con comprobar las columnas:

```
bool const mismaDiagonalDerIz = (columnaP == columnaQ);
```

Por último, para averiguar si dos números están en la misma diagonal, es suficiente con calcular la disyunción de las dos expresiones anteriores:

```
bool const mismaDiagonal = mismaDiagonalDerIz || mismaDiagonalIzDer;
```

## 17 Reflexión de los bits de una palabra de anchura fija

27 31

Medítese un momento sobre la siguiente asignación:

```
bits = ((bits & 0x0000FFFFU) << 16)
     | ((bits & 0xFFFF0000U) >> 16);
```

Es una especie de inversión grosera porque ha intercambiado los dos octetos superiores con los dos inferiores, pero ha olvidado hacer la oportuna y necesaria reflexión dentro de esos dos pares de octetos. No importa, porque con el mismo truco podemos conseguir otra reflexión grosera que intercambie, simultáneamente, el octeto 0 con el 1 y el 2 con el 3:

```
bits = ((bits & 0x00FF00FFU) << 8)
     | ((bits & 0xFF00FF00U) >> 8);
```

Ahora estamos bastante más cerca porque cada bit ya está en el octeto en que debe, aunque falta reflejar todos y cada uno de los octetos:

```
bits = ((bits & 0x0F0F0F0FU) << 4)
     | ((bits & 0xF0F0F0F0U) >> 4);
```

Ahora falta invertir cada grupo de 4 bits. Siguiendo con el truco, en un par de asignaciones más hemos invertido `bits`:

```
bits = ((bits & 0x33333333U) << 2)
     | ((bits & 0xCCCCCCCCU) >> 2);
bits = ((bits & 0x55555555U) << 1)
     | ((bits & 0xAAAAAAAAU) >> 1);
```

## 21 Número de bits a 1

29

Lo primero que hace falta es entender nuestro entero como si fuera la concatenación de varios números. De esta forma podemos dividir el problema en cuestión en dos partes: primero saber cuántos bits a 1 tiene cada parte y luego sumar esas cantidades.

Si consideramos que nuestro entero de 32 bits está formado por la concatenación de 32 enteros de un solo bit, la primera parte de nuestra descomposición se resuelve trivialmente. Parecerá que todo este argumento nos ha vuelto a dejar en el mismo estado; pero no es así, simplemente porque ahora ya no tenemos que contar bits, sino sumar 32 números. Es un cambio de enfoque que resulta crítico para poder resolver este problema eficientemente.

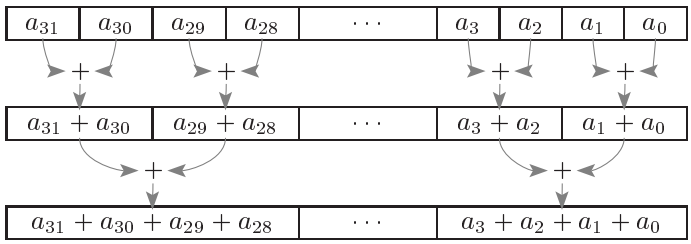



Figura 1.5: Forma de sumar los bits de un entero

Supongamos que nuestro entero original es $\sum_{i=0}^{31} a_i 2^i$. Tenemos que calcular $\sum_{i=0}^{31} a_i$. Procederemos a sumar estos números en un orden muy particular que puede verse ilustrado en la figura 1.5. Primero sumaremos pares consecutivos $a_{2i+1} + a_{2i}$. De esta forma habremos reducido nuestro problema a la mitad. Lo sorprendente es que estas 16 sumas se resuelven trivialmente con el siguiente código:

```
bits = (bits & 0x55555555U) +
      ((bits & 0xaaaaaaaaU) >> 1);
```

El entero resultante se ha de interpretar como la concatenación de 16 enteros de 2 bits. Cada uno de esos enteros es la suma de un par de viejos enteros de 1 bit y, por tanto, es el número de bits que tenían ese par de viejos enteros de 1 bit.

Aplicando la misma idea, continuamos nuestro proceso de suma juntando cada par consecutivo de estos nuevos enteros de 2 bits:

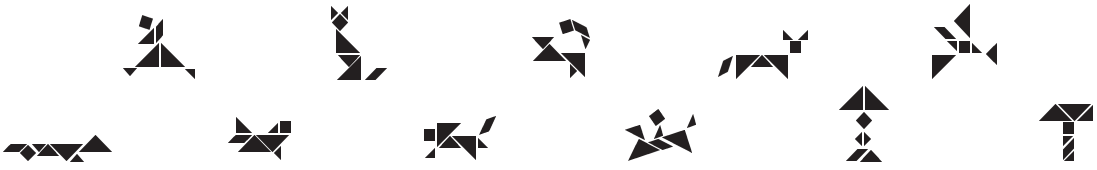
```
bits = (bits & 0x33333333U) +
      ((bits & 0xccccccccU) >> 2);
```

Un par de pasos más en donde seguimos sumando enteros consecutivos:

```
bits = (bits & 0x0f0f0f0fU) +
      ((bits & 0xf0f0f0f0U) >> 4);
bits = (bits & 0x00ff00ffU) +
      ((bits & 0xff00ff00U) >> 8);
```

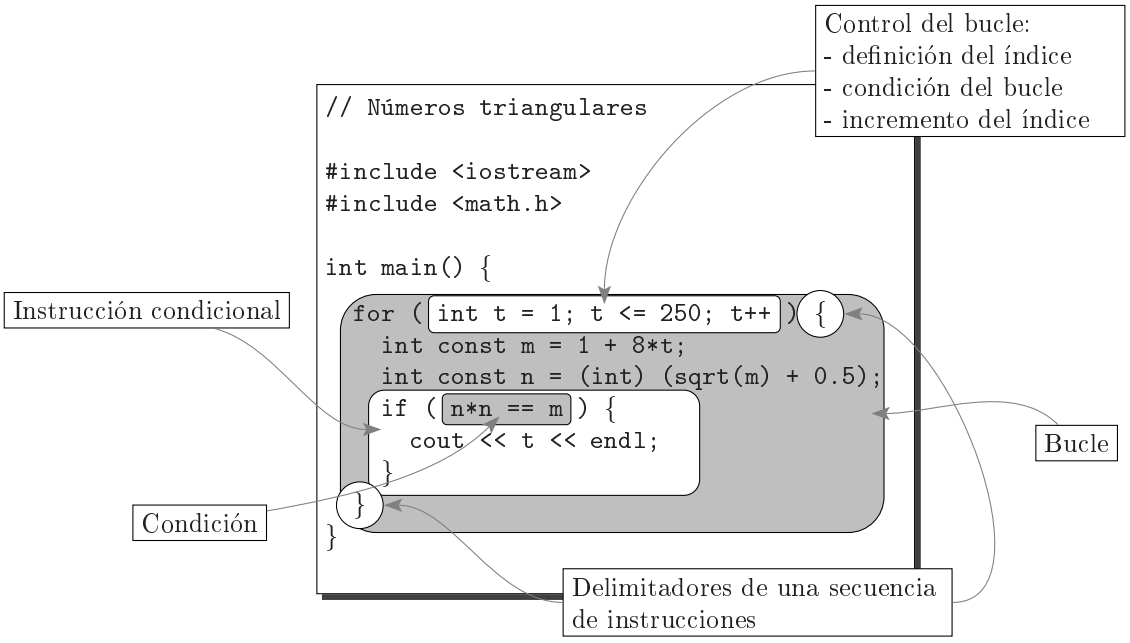
En este punto, `bits` se debe interpretar como la concatenación de dos enteros de 16 bits, cada uno con la cantidad de bits de cada mitad del entero original. Un paso más y tendremos resuelto el problema:

```
bits = (bits & 0x0000ffffU) +
      ((bits & 0xffff0000U) >> 16);
```



# 2 Capítulo DOS

## Instrucciones estructuradas



## En este capítulo

## 2.1 Composición secuencial

Para indicar que queremos ejecutar varias instrucciones seguidas, basta con escribirlas en ese orden. La secuencia

```
int x = 0;
x = x+1;
x++;
```

realiza por orden las siguientes instrucciones: define la variable entera `x` con valor inicial cero, calcula el sucesor del valor en `x` y lo guarda en esta misma variable, y vuelve a incrementar en una unidad la variable `x`, esta vez con el operador `++`, de incremento.

Una secuencia de instrucciones no es una instrucción. Esta afirmación puede parecer trivial, pero en breve se verá por qué es importante poder tratar una secuencia de instrucciones como si fuera una sola instrucción. En todo caso, para lograrlo basta con rodear con unas llaves toda la secuencia. El ejemplo anterior se convierte en una sola instrucción compuesta, así:

```
{
  int x = 0;
  x = x+1;
  x++;
}
```

Las llaves, además de agrupar, introducen un *bloque*. Las declaraciones sólo sobreviven hasta que se alcanza el final del bloque donde se han declarado. Si intentáramos compilar esto,

```
{
  int x = 0;
  x = x+1;
}
x++;
```

obtendríamos un error en la última línea: el compilador se quejaría de que en ese punto el identificador `x` no tiene una declaración asociada.

## 2.2 Selección condicional

Se puede condicionar la ejecución u omisión de una instrucción con la instrucción de selección condicional `if`. Su forma es la siguiente:

`if (`⟨*condición*⟩`)` ⟨*Cuerpo del* `if`⟩

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 `if`. El *cuerpo* del `if` debe ser una sola instrucción; para ejecutar condicionalmente más de una instrucción, hay que convertirlas en un bloque rodeándolas con unas llaves.

Cuando la ejecución alcanza el condicional, primero se evalúa la condición; si el resultado es `true`, se ejecuta el cuerpo, pero se obvia si es `false`. Por ejemplo,

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

incrementa `p` en `x` si `n` es un número impar. Hay una diferencia muy importante entre

```
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`:

```
{
  ⟨Instruccion 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 menundo 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⟩
  }
}
```

## 21 Signo del producto

Dados dos números enteros $n$ y $m$, indaga una forma de calcular el signo de su producto sin llegar a calcular dicho producto.

## 22 Ponle tú el título

Contempla detenidamente el siguiente bloque de programa:

```
int n, m;
do {
  cin >> n >> m;
} while ((n < 0) || (m < 0));
int x = 0;
while (m != 0) {
  if (m % 2 != 0) x = x + n;
  n = n + n;
  m = m / 2;
}
cout << "El resultado es " << x << endl;
```

¿Qué hace?

***Un poco de historia*** Este método de multiplicación, que ha perdurado en Europa Oriental hasta hace poco tiempo, es una variante de un algoritmo ya conocido por los antiguos egipcios. Este último sólo necesitaba sumar y multiplicar por 2 como operaciones básicas.

Veamos con un ejemplo cómo se las apañaban los egipcios para multiplicar. Supongamos que se desea calcular $22 \times 47$: calculamos la multiplicación de 47 por todas las potencias de 2 inferiores, o iguales, a 22. Para esto sólo hace falta saber multiplicar por dos:

$$\begin{array}{rcrcr} 1 & \times & 47 & = & 47 \\ 2 & \times & 47 & = & 94 \\ 4 & \times & 47 & = & 188 \\ 8 & \times & 47 & = & 376 \\ 16 & \times & 47 & = & 752 \end{array}$$

Ahora, tenemos que sumar algunos de los términos anteriores (en este caso, 16, 4 y 2) para obtener el resultado:

$$22 \times 47 = (16 + 4 + 2) \times 47 = 16 \times 47 + 4 \times 47 + 2 \times 47 = 752 + 188 + 94 = 1034.$$

## 23 El escaso coste de la destrucción

Citamos de forma textual pero indirecta al periódico *Norte de Castilla* en un número de marzo de 1998: "Los militares han hecho cálculos de cuánto cuesta destruir un kilómetro cuadrado de territorio enemigo: unas 300 000 pesetas (1 800€) con explosivos convencionales; si se usa un ingenio atómico 100 000 pesetas (600€). Los gases son aún más baratos. Sembrar un kilómetro cuadrado con una toxina mortal apenas

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 fallecimiento de mi padre, resolví apartarme de los negocios, cediéndolos a otra casa extractora de Jerez para colmo una edición infecta, uno se pregunta cómo puede interesarle algo así. Pensar que se ha pasado horas enteras devorando esta sopa fría y tan acreditada como la mía; realicé los créditos que pude, arrendé los predios, traspasé las bodegas... Y las cosas que lee, una novela, mal escrita, sabrida, 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 césares* 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 Cryptograpy* 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*:

$$a_0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1} + a_n x^n$$

(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*:

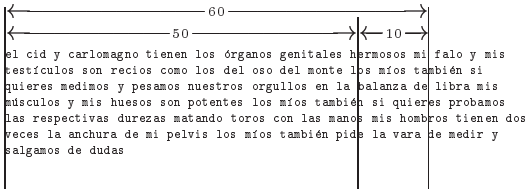
$$a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$$

(Véase la pista 2.8a.)

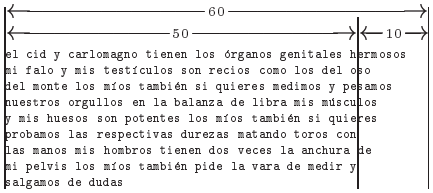
## 2 9 Reorganiza las líneas de un texto

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.




Entrada




Salida

## 2 10 Representación gráfica de funciones

82

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

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

Por otra parte, como el tamaño de la pantalla es fijo, la representación se efectúa sobre una cuadrícula de tamaño fijo, $númX \times númY$ 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 abscisas 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.

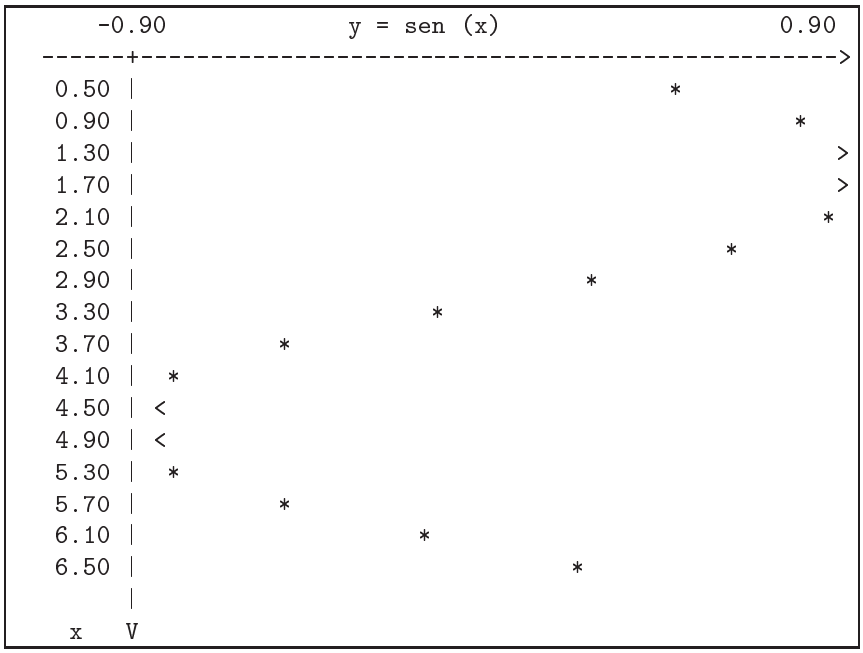



Figura 2.2: Gráfica aproximada de la función seno

***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

78

Observa el dibujo de M. C. Escher (1898–1972) que aparece en la figura 2.3 titulado *Wentelteefje*. 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.

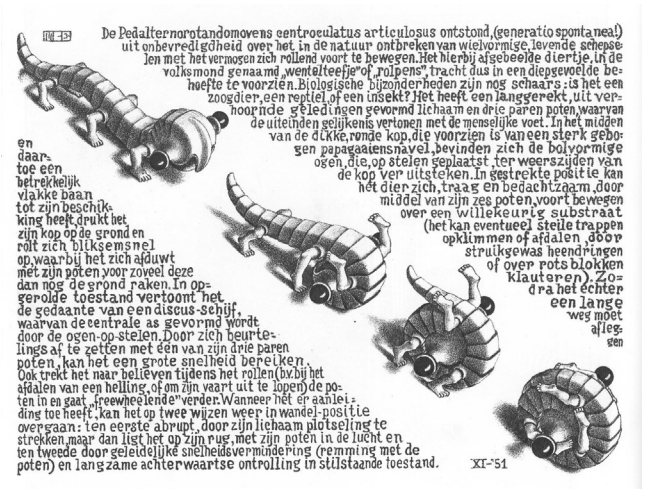



Figura 2.3: *Wentelteefje* de M.C. Escher

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 $\pi$:

```
                  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 abcisas 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 $r$.

## 2.15 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:

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

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.

## 2.16 Un bonito triángulo

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

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

¿Te atreves?

## 217 Varianza

78 85

La varianza de $n$ números $x_1, \ldots, x_n$ 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.)

## 218 ISBN de libros

78

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:

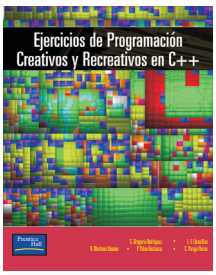
$$x_{10} = \left(\sum_{i=1}^{9} i x_i\right) \bmod 11$$

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:

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

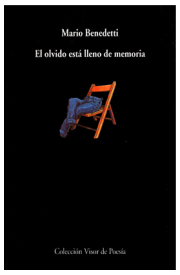
***Ejemplo*** Consideremos las siguientes fichas bibliográficas de libros. En ellas, el ISBN es un número formado por 10 dígitos; los nueve primeros son determinados por la editorial y el último de ellos es el dígito de control. A veces se utilizan guiones, que desempeñan un papel de meros separadores, 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 programacion 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

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

***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.

## 2.19 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.

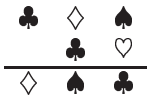
Agrupándose de seis en seis, estos símbolos generan los *hexagramas* que ves en la figura 2.4. Se pide un programa que dibuje en la pantalla esos sesenta y cuatro hexagramas, sin necesidad de atenerse al caprichoso orden del rey Wen. (Véanse las pistas 2.19a y 2.19b.)

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

## 2.20 Suma marciana

86

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



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

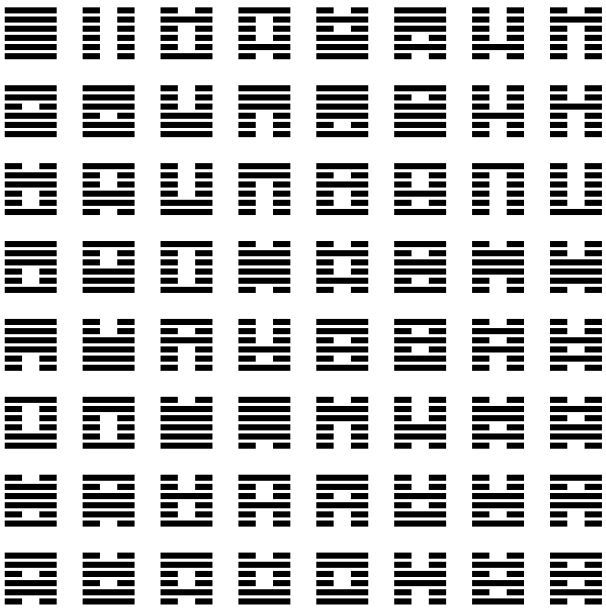


Figura 2.4: Ordenación de los 64 hexagramas según el rey Wen

## 2.21 El número y sus representaciones

78

> 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 aquéllas 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:

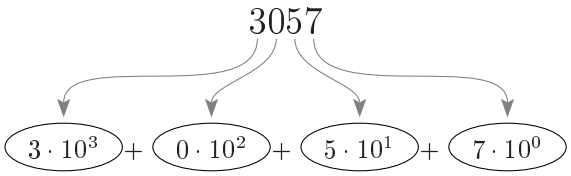
Para números mayores utilizaban una representación multiplicativa: una cantidad de unidades seguida de uno de los símbolos para las potencias de 10 significaba el producto de dichas unidades por la potencia. Considerando que escribían de derecha a izquierda, la representación del número 500 era ésta:

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:

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

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.

## 2.22 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

$$\overbrace{\underbrace{00}_{3_l}\ \underbrace{11}_{3_c}}^{3}\ \overbrace{\underbrace{0000}_{10_l}\underbrace{1010}_{10_c}}^{10}$$

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 $n$,

$$\underbrace{0\cdots0}_{k}uv$$

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].

## 2 23 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:

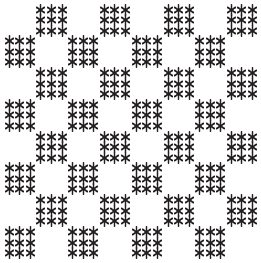
```
                                ** 
                              **** 
                            ******** 
                          **************** 
                  ********************************
**************************************************************** 
```

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

```
 * * * *
* * * *
 * * * *
* * * *
 * * * *
* * * *
 * * * *
* * * *
```

compuesta por una matriz de $8 \times 8$ 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.

## 2 24 Aproximaciones al número $\pi$

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 $\pi$.

- 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:

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

- 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):

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

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots \qquad \frac{\pi^4}{90} = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \cdots$$

$$\frac{\pi^2}{8} = 1 + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \cdots \qquad \frac{\pi^3}{32} = 1 - \frac{1}{3^2} + \frac{1}{5^2} - \frac{1}{7^2} + \cdots$$

- Borwein en 1987:

$$x_0 = \sqrt{2} \qquad y_1 = 2^{\frac{1}{4}} \qquad \pi_0 = 2 + \sqrt{2}$$

$$x_{n+1} = \frac{1}{2}\left(\sqrt{x_n} + \frac{1}{\sqrt{x_n}}\right) \qquad y_{n+1} = \frac{y_n\sqrt{x_n} + \frac{1}{\sqrt{x_n}}}{y_n + 1} \qquad \pi_n = \pi_{n-1}\frac{x_n + 1}{y_n + 1}$$

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

***Bibliografía*** $\pi$ es sin duda el más famoso de los números, y por eso la bibliografía sobre él es extensísima. Dos libros llenos de curiosidades sobre $\pi$ son [AH01] y [Bec82]. En [Cha99] se dedica un capítulo a los diversos métodos usados a lo largo de la historia para calcular $\pi$. 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 $\pi$; en la tercera puedes encontrar muchas, pero que muchas, cifras decimales de $\pi$.

***Un poco de historia*** El primero que utilizó el símbolo $\pi$ 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 $\pi$.

He aquí una tabla con la cronología del número de cifras decimales de $\pi$ 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) |

## 25 Formas de determinar la pertenencia de un punto a un polígono

Sean $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ los sucesivos vértices de un polígono con $n$ lados; hay una arista que une cada par de vértices consecutivos y una última entre $(x_{n-1}, y_{n-1})$ y $(x_0, y_0)$ que cierra el polígono. Nos dan además otro punto $(a, b)$ y nos preguntan si $(a, b)$ cae en el interior del polígono.

***Paridad del número de cortes de una semirrecta*** Un método tradicional para resolver este problema consiste en trazar una semirrecta que empiece en $(a, b)$ con cualquier dirección y contar el número (llamémosle $k$) de intersecciones entre la semirrecta y los lados del polígono; siempre ocurre que $k$ es impar si $(a, b)$ está en el interior del polígono y que es par cuando el punto está fuera del polígono. Como la paridad de $k$ es independiente de la semirrecta elegida, conviene trabajar siempre con semirrectas verticales u horizontales.

***Ángulo de rotación*** Un método tradicional para resolver este problema consiste en medir el ángulo que forman los segmentos $(a, b)$–$(x_i, y_i)$ y $(a, b)$–$(x_{i+1}, y_{i+1})$ y que llamaremos $\alpha_i$; también medimos el ángulo que forman $(a, b)$–$(x_{n-1}, y_{n-1})$ y $(a, b)$–$(x_0, y_0)$ y que llamaremos $\alpha_{n-1}$; una vez calculados todos estos ángulos se suman para obtener $\alpha = \sum_{i=0}^{n-1} \alpha_i$. Siempre ocurre que $\alpha$ es 0 si $(a, b)$ está fuera del polígono y que $\alpha$ es un múltiplo de $2\pi$ si el punto está en el interior del polígono. Obsérvese que el ángulo que forman los segmentos $(a, b)$–$(x_{i+1}, y_{i+1})$ y $(a, b)$–$(x_i, y_i)$ es justamente $-\alpha_i$; como queremos estos valores para sumarlos, es muy importante ser consistente en el método de cálculo y que los ángulos siempre reflejen el movimiento de un vértice al siguiente.

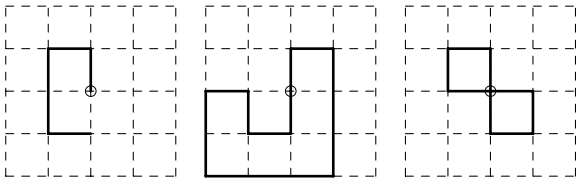
***Diferencias entre los métodos*** ¿Los métodos anteriores definen de igual forma cuándo un punto está dentro de un polígono o, por el contrario, hay algunos polígonos para los que un método dice que cierto punto está en su interior mientras que otro método dice que está fuera?

***Bibliografía*** Éste es un problema muy básico en geometría computacional. El compendio clásico en este área es [PS91]. Pero tanto [O'R01] como [dvOS00] son fuentes más amenas con un planteamiento eminentemente práctico.

## 26 Área encerrada por unos movimientos simples

78 90

En espera del temible Juanito y de su inquieta caja de rotuladores, que ya ha coloreado todo el pasillo y al pobre Toby, has conseguido un cuaderno de papel cuadriculado y barruntado un juego que lo mantendrá entretenido toda la tarde. Juanito tendrá que poner la punta de su rotulador rojo en la esquina de algún cuadro; a la orden de derecha (respectivamente baja, izquierda y sube) tendrá que trazar el segmento que une el punto actual con el que queda a su derecha (respectivamente abajo, izquierda y arriba); el rotulador quedará colocado en este último punto esperando la siguiente instrucción. Por ejemplo y abreviando las órdenes con su inicial, las secuencias de instrucciones `SIBBD`, `BISIBBDDDSSSIB` y `SIBDDBIS` producirán los siguientes tres dibujos, donde se ha marcado con un pequeño círculo el punto de inicio:



(a) (b) (c)

Sólo daremos secuencias de órdenes que, al ser dibujadas en el papel cuadriculado, produzcan un polígono cerrado sin intersecciones, es decir, con el aspecto del ejemplo (b) pero no del (a) ni del (c). Finalmente, para olvidarnos un buen rato de Juanito, le mandaremos contar y colorear en verde todos los cuadrados que quedan dentro del polígono.

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.)

## 2.27 $n$-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.

## 2.28 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 intoducida 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 $f : \mathbb{R} \to \mathbb{R}$ 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 $x \geq 0$.

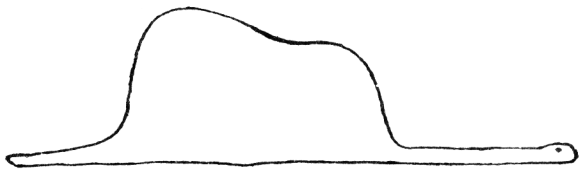
***Logaritmo*** También se puede aplicar para implementar la función $\log_b(x)$ en base $b$ siendo $b > 1$ y $1 \leq x \leq b$. ¿Cómo se podría implementar la función $\log_b(x)$ para $x > 0$? (Véase la pista 2.29a.)

## 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.)

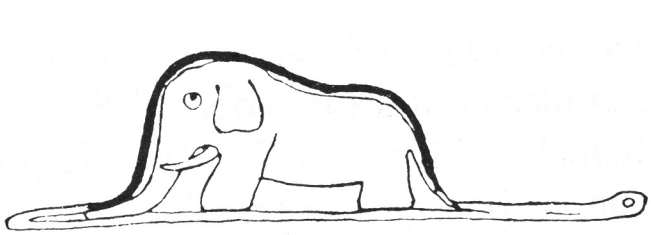
## 2 31 El principito

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].

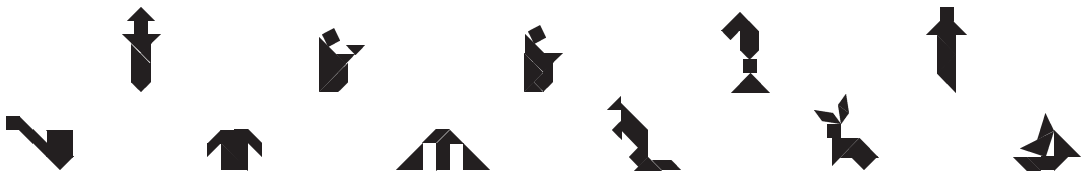
## 2 32 Suma que te suma

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 únicamenta 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.)



**2.5a.** Para llevar a cabo este programa se propone el siguiente esquema:

```
cout << "Estate atento" << endl;
⟨Esperar un instante aleatorio⟩
while (⟨no se haya pulsado ninguna tecla⟩) {
  cout << '*' << flush;
  ⟨Esperar un lapso de tiempo⟩
}
```

Si no sabes a dónde recurrir para calcular un número aleatorio, o para detener la ejecución del programa durante un cierto tiempo, consulta la pista 2.5b.

**2.5b.** Para detallar este ejercicio necesitamos recurrir a la biblioteca de subprogramas de nuestro lenguaje en busca de dos métodos: uno para detener la ejecución un periodo de tiempo y otro para consultar si se ha pulsado una tecla. La generación de números aleatorios está explicada en el ejercicio 3.13.

**Espera** El procedimiento

```
void usleep(unsigned long microsegundos);
```

que está en el fichero de cabecera `unistd.h`, detiene la ejecución del programa durante los microsegundos especificados.

**Pulsación** En Unix (el contexto en el que casi siempre vive C), el teclado es un *dispositivo* de entrada. Hay varias funciones que nos permiten saber en qué dispositivos hay datos listos para leer. Si tenemos datos en el dispositivo del teclado será porque se ha pulsado una tecla. La función más simple para este propósito es ésta,

```
int poll(pollfd ds[], unsigned int nds, int espera);
```

que aparece en el fichero de cabecera `sys/poll.h`. El array `ds` contiene los dispositivos que deseamos vigilar; `nds` es la longitud del array anterior; y `espera` son los milisegundos que queremos esperar a que ocurra algo en los dispositivos de `ds`. Si `espera` es negativo, se espera indefinidamente; si es 0, se mira el estado actual de los dispositivos y termina. El resultado de esta función es el número de dispositivos en los que ocurrió algo. El tipo `pollfd` es un registro con dos campos interesantes: `fd` es el número del dispositivo que queremos consultar (0 para el teclado) y `events` que, para indicar que estamos interesados en leer, debe valer `POLLIN`. El siguiente código espera a que pulsemos una tecla:

```
pollfd pulsacion[1];
pulsacion[0].fd = 0;
pulsacion[0].events = POLLIN;
poll(pulsacion, 1, -1);
```

**2.6a.** Tanto el cifrado como el descifrado tienen que basarse en una *clave*, que será capaz de describir el código de rotación que estamos utilizando. Una forma sencilla de representar dicha clave es indicando cuál es el cifrado de la primera letra del alfabeto. Por ejemplo, en el caso del código de Julio César, la clave sería la letra $D$ que es la letra en la que ciframos la $A$; con este dato podemos conocer el cifrado de cualquier otra letra. Otra forma sencilla para representar la clave es con un número entero que indique cuánto hay que avanzar (o retroceder) por el alfabeto para cifrar una letra. En este caso, la clave para

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 Willian George Horner (1786–1837), permite calcular un polinomio de grado $n$ de forma acumulativa:

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

**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*.

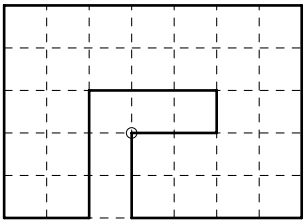
**2.19b.** La dicotomía entre el *yin* y el *yang* se resuelve con un *bit*. Una vez elegida una correspondencia entre estos principios humanos y los principios informáticos 0 y 1, un hexagrama se reduce a la representación binaria de un número de 6 bits, es decir, un número entre 0 y $2^6 - 1$ (= 63).

**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:

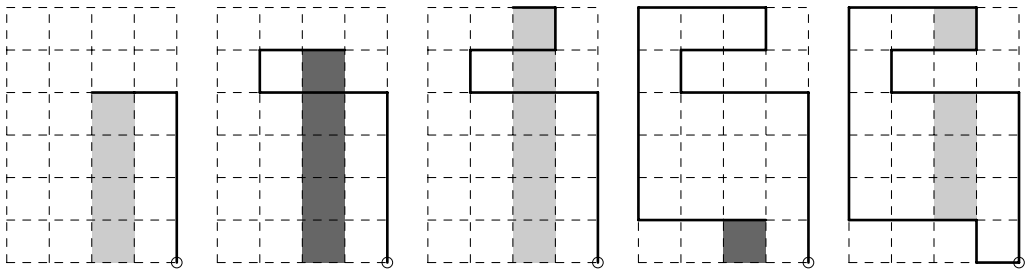
| símbolos aztecas | | | | símbolos arameos | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| • |  |  |  |  |  |  |  |  |  |
| . | p | f | @ | 1 | n | h | ? | N | * |

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 `BBDDDDSSSSSIIIIIIIBBBBBDDSSSDDDBII` 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, $\mathrm{mcd}(n, d)$, 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:

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

## 21 Signo del producto

55

Encontrar una solución suficientemente buena o elegante es cosa de paciencia. Lo interesante es buscar *la mejor*, por supuesto desde una perspectiva estética. Daremos el signo como uno de los enteros $-1$, 0 o 1, según el producto sea negativo, cero o positivo, respectivamente. Ésta es nuestra mejor solución:

```
int signo;
if ((n == 0) || (m == 0)) signo = 0;
else if ((n < 0) == (m < 0)) signo = 1;
else signo = -1;
```

La primera condición captura cuándo el resultado es 0: precisamente si uno de los dos factores es 0. La segunda condición recoge el concepto de *tener el mismo signo*. La expresión `n < 0` pregunta si `n` es negativo. Su resultado es `true` si `n` es negativo y `false` si es positivo. Por tanto, los resultados de dos expresiones de esta forma, una sobre la variable `n` y otra sobre `m`, son iguales precisamente si el signo de ambas variables es el mismo.

## 22 Ponle tú el título

55

El siguiente bucle,

```
do {
  cin >> n >> m;
} while ((n < 0) || (m < 0));
```

realiza la lectura de dos valores numéricos hasta que ambos sean enteros no negativos. A continuación, el siguiente bucle,

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

se ejecuta mientras el contenido de `m` sea distinto de cero. En este caso, si `m` es impar acumula en la variable `x` el valor de `n`, tras lo que, independientemente de que `m` sea o no impar, duplica el valor de `n` y reduce a la mitad entera el valor de `m`. Cuando se produce la salida del bucle, en la variable `x` queda almacenado el producto de `m` y `n`. Veamos su funcionamiento con un ejemplo. Supongamos que, inicialmente, el valor de `m` es 5 y el de `n` 3. Los valores que irán tomando `m`, `n` y `x` tras cada ejecución del cuerpo del bucle `while` quedan reflejados en la siguiente tabla:

| m | n | x |
|---|---|---|
| 5 | 3 | 0 |
| 2 | 6 | 3 |
| 1 | 12 | **15** |
| 0 | 24 | |

Existen diferentes algoritmos para multiplicar dos números enteros sin utilizar el operador producto "`*`"; el que implementa este ejercicio es uno de ellos y recibe el nombre de *multiplicación a la rusa*. En `http://mathforum.org/dr.math/faq/faq.peasant.html` puedes encontrar más información.

## 2 4 Lectura de Rayuela 56

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.

## 2 10 Representación gráfica de funciones 59

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 = 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`,...,`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* $x_{mín}$, $x_{máx}$, $y_{mín}$, $y_{máx}$⟩
⟨*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 $númY$, 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:

$$x_i = x_{mín} + i\frac{x_{máx} - x_{mín}}{númX}, \text{ para } i \in \{0, \ldots, númX\}.$$

- Para cada ordenada $y_i = f(x_i)$, su posición será un entero de $posY_i \in \{0, \ldots, númY\}$ cuando $y_i \in [y_{mín}, y_{máx}]$, lo que se consigue fácilmente haciendo que $posY_i$ sea el entero más próximo a

$$númY\frac{y_i - y_{mín}}{y_{máx} - y_{mín}}.$$

- Por último, un valor de $posY_i$ negativo o nulo indica que la función se sale por la izquierda del fragmento del plano representado, mientras que un valor mayor que $númY$ 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;
}
```

## 2.15 Los cubos de Nicómaco

62

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 ;
```

## 216 Un bonito triángulo

62

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;
}
```

## 217 Varianza

63 78

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:

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

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;
}
```

## 2.20 Suma marciana

64

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 $100\clubsuit + 10\diamondsuit + \spadesuit$.

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 tamanyo = 8;
  for (int i = 1; i <= tamanyo; i++) {
    // Dibujar la fila i:
    for (int j = 1; j <= tamanyo; 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 tamanyo = 8;
  int ancho;
  cout << "Ancho del escaque:  ";
  cin  >> ancho;
  for (int i = 1; i <= tamanyo; i++) {
    for (int ii = 1; ii <= ancho; ii++) {
      for (int j = 1; j <= tamanyo; 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.

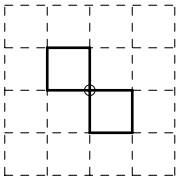
Por tanto deberemos hacer un bucle recorriendo el contorno de la figura según la entrada de datos; en cada vuelta del mismo vamos actualizando la posición, si el movimiento es hacia la derecha restamos a la superficie el contenido de la variable `posY` y si es hacia la izquierda lo sumamos.

```
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;
```

Podemos comprobar fácilmente que en esta solución no importa si el polígono está o no por encima del punto de origen. Si estamos por debajo del punto de origen y vamos hacia la derecha deberemos añadir los cuadrados hasta el punto de origen. Pero como `posY` tendrá un valor negativo, para hacer esa suma sigue siendo válido restar el valor de esa variable. Y lo mismo ocurre si vamos hacia la izquierda.

Faltaría por último averiguar qué ocurre si, en lugar de suponer que tenemos el polígono a la izquierda, lo tenemos a la derecha. Simplemente basta con observar que, donde antes sumábamos, ahora restamos y viceversa, por lo que nos quedaría la superficie negativa. En definitiva, para que funcione en ambos casos es suficiente con devolver el valor absoluto de la superficie calculada.

Además esto implica que la superficie no se calculará de forma adecuada si la línea poligonal se cruza con ella misma, puesto que después del cruce el polígono pasa de estar a un lado a estar al otro por lo que las superficies se restarían. Así nuestro cálculo de la superficie para la siguiente figura es cero.



## 2 28 De notación polaca a código ensamblador

73

Numerando los registros desde el cero, llamamos `cima` al último ocupado. Inicialmente, todos están libres:

```
int cima = -1;
```

Examinando ahora la notación postfija, vemos que cada elemento (variables u operaciones) viene dado por un carácter, lo que simplifica su lectura y tratamiento:

```
int main() {
  int cima = -1;
  char c;
  cout << "Expresión postfija (acabada en punto):  ";
  while (cin >> c) {
    if ('A' <= c && c <= 'Z') {
      ⟨Lectura de una variable⟩
    } else if (c == '+') {
      ⟨Lectura y tratamiento de una suma⟩
    } else if (c == '-') {
      ⟨Lectura y tratamiento de una resta⟩
    } else if (c == '*') {
      ⟨Lectura y tratamiento de una multiplicación⟩
    } else if (c == '/') {
      ⟨Lectura y tratamiento de una división⟩
    } else if (c == '@') {
      ⟨Lectura y tratamiento de un cambio de signo⟩
    } else if (c == '.')  {
      ⟨Punto final⟩
    }
  }
}
```

Y las situaciones posibles para cada carácter leído son las siguientes:

**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.

## 2.29 Raíces y logaritmos

74 79

***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 $y^2 - \mathtt{x} = 0$. Como $\mathtt{x} \geq 1$, tenemos que $1^2 - \mathtt{x} \leq 0$ y $\mathtt{x}^2 - \mathtt{x} \geq 0$; 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 $\mathtt{li}^2 - \mathtt{x} \leq 0 \leq \mathtt{ls}^2 - \mathtt{x}$; o sea, que el cero se mantiene entre `li` y `ls`.

El problema al considerar $0 \leq \mathtt{x} < 1$ es que $\mathtt{x}^2 - \mathtt{x} < 0$ y $1^2 - \mathtt{x} > 0$, 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 $\mathtt{x} \geq 0$, 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 $\mathtt{b}^y - \mathtt{x}$. Sabemos que $1 \leq \mathtt{x} \leq \mathtt{b}$ por lo que $\mathtt{b}^0 - \mathtt{x} \leq 0$ y $\mathtt{b}^1 - \mathtt{x} \geq 0$.

```
li = 0; ls = 1;
while (ls-li >= error) {
  medio = (li+ls)/2;
  valor = ⟨b^medio⟩ - 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:

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

Así pues, introducimos dos variables nuevas `bLi` y `bLs`, cuyo valor en cada vuelta del bucle será $\mathtt{b}^{\mathtt{li}}$ y $\mathtt{b}^{\mathtt{ls}}$ 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 $\mathtt{x} > \mathtt{b}$; realizaremos entonces un bucle dividiendo `x` por `b`, de forma que en cada vuelta se verifique $\mathtt{b}^{\mathtt{exponente}} \cdot \mathtt{resto} = \mathtt{x}$. Obsérvese que, en cada momento, $\log_{\mathtt{b}}(\mathtt{x}) = \mathtt{exponente} + \log_{\mathtt{b}}(\mathtt{resto})$. Si iteramos el bucle hasta que $\mathtt{resto} \leq \mathtt{b}$, podremos aplicar el algoritmo de arriba a la variable `resto`, y puesto que $\mathtt{b} > 1$ llegará un momento que el algoritmo pare.

Si $0 < \mathtt{x} < 1$, 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;
```

## 2 32 Suma que te suma

75 79

***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 $\texttt{i}^2$ para $\texttt{i} = 0, 1, 2 \ldots$ 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 $(\texttt{i}+1)^2$ sin efectuar multiplicaciones. Para ello podemos utilizar el valor $\texttt{i}^2$ que en la vuelta anterior se almacenó en la variable `iCuadrado`. Como $(\texttt{i}+1)^2 = \texttt{i}^2 + 2\texttt{i} + 1$, la expresión para calcular $(\texttt{i}+1)^2$ 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 $2\texttt{i} = \texttt{i} + \texttt{i}$ 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 $2\texttt{i}+1$. 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 $2(\mathtt{i}+1)+1 = 2\mathtt{i}+1+2$ 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 $\mathtt{i}^3$. En cada vuelta del bucle tendremos que actualizar el valor de `iCubo`. Desarrollando, tenemos lo siguiente:

$$(\mathtt{i}+1)^3 = \mathtt{i}^3 + 3\mathtt{i}^2 + 3\mathtt{i} + 1$$

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

$$3(\mathtt{i}+1)^2 + 3(\mathtt{i}+1) + 1 = 3\mathtt{i}^2 + 3\mathtt{i} + 1 + 6\mathtt{i} + 6$$

llegamos a la asignación:

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

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

$$6(\mathtt{i}+1) + 6 = 6\mathtt{i} + 6 + 6$$

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:

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

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

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

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

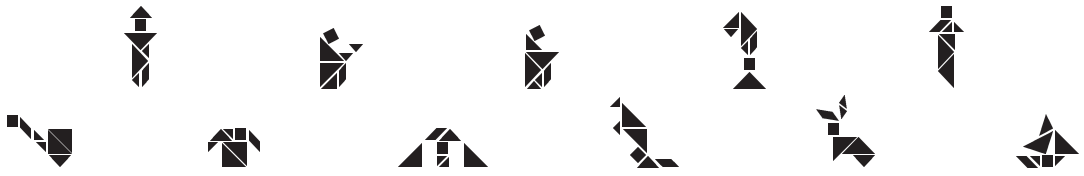
Si elevamos todo al cuadrado, y sabiendo que todos son números no negativos, tenemos lo siguiente:

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

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

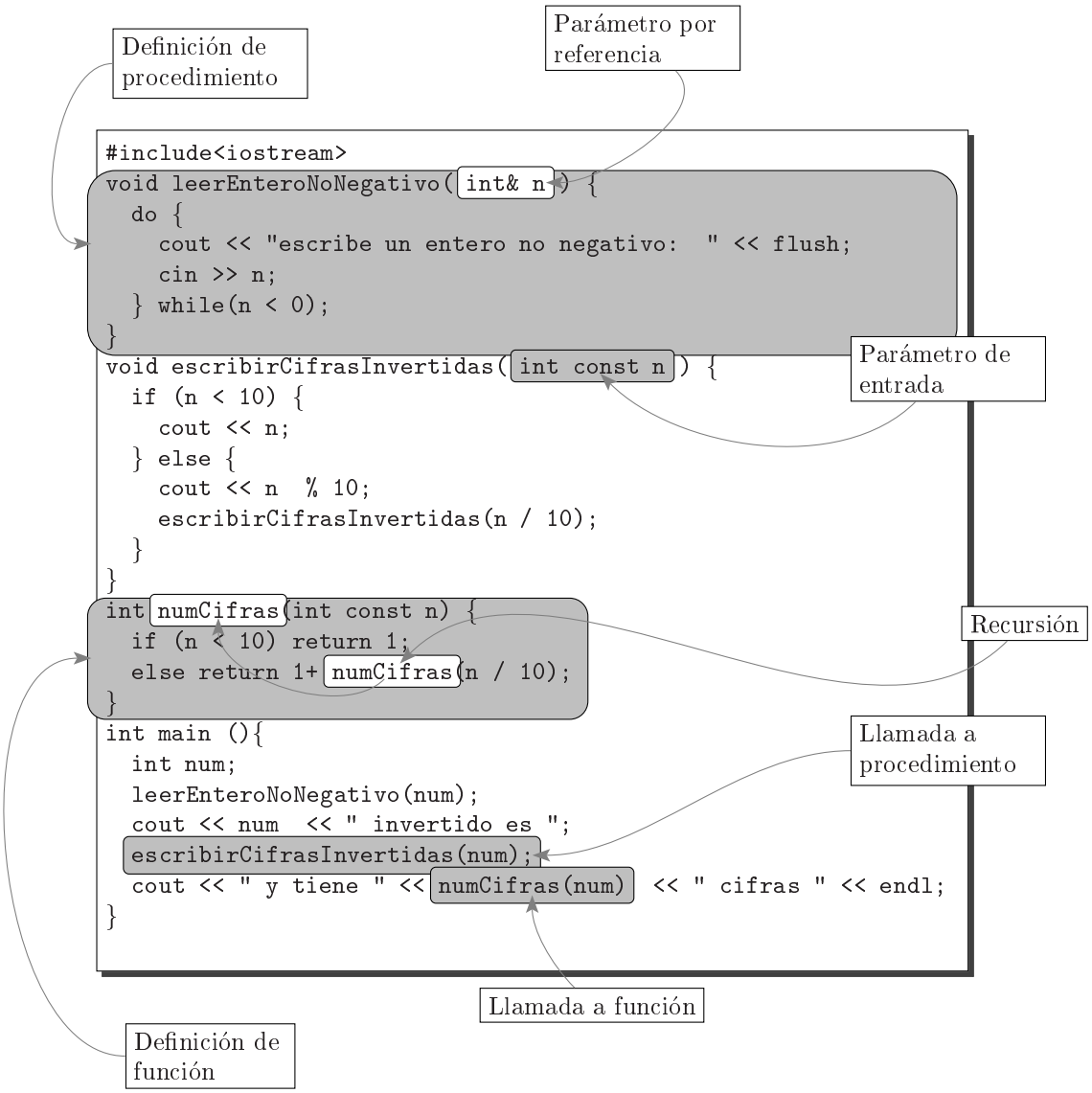
```
i = 0; iCuadrado = 0; dosIMasUno = 1; iMasUnoCuadrado = 1;
while (iMasUnoCuadrado <= n) {
  iCuadrado = iMasUnoCuadrado;
  dosIMasUno = dosIMasUno + 2;
  iMasUnoCuadrado = iCuadrado + dosIMasUno;
  i++;
}
raiz = i;
```

La raíz cúbica se calcularía de forma similar.



# 3 Capítulo TRES

## Subprogramas



# En este capítulo

Un *subprograma* es un fragmento de código que resuelve una tarea claramente demarcada. Se llama *funciones* a los subprogramas que desempeñan el papel de una expresión: su resultado es un valor y no tienen ningún efecto ni sobre variables globales ni sobre el entorno. Se llama *acciones* o *procedimientos* a los subprogramas que desempeñan el papel de una instrucción: leen, escriben o modifican variables.

En C++, funciones y acciones se definen prácticamente de la misma forma, con unas diferencias sintácticas muy pequeñas. Esto no tiene nada de extraño. Es más desconcertante la escasa distinción semántica entre funciones y acciones. Por ejemplo, se puede usar una función como si fuera una acción y C++ ignorará su valor resultante sin la más mínima queja. No obstante, la distinción entre funciones y acciones no sólo es interesante desde el punto de vista expositivo o metodológico, sino también desde un punto de vista práctico, porque representan dos conceptos distintos que, también en su uso, están claramente diferenciados. Recomendamos escribir funciones o acciones tal y como se explica en los apartados 3.1 y 3.2.

C++ admite recursión, técnica que supondremos conocida y que usaremos sin más en algunos de los ejemplos de este resumen.

## 3.1 Funciones

Una función es una entidad que recibe datos mediante unos *parámetros de entrada* y devuelve como resultado un valor de un cierto tipo. Una función se define así:

⟨*Tipo resultado*⟩ `nombreDeLaFuncion(`⟨*parámetros de entrada*⟩`) {`
  ⟨*Cuerpo de la función*⟩
`}`

Un *parámetro de entrada* tiene la misma forma que la declaración de una constante; recibe el apelativo de *entrada* precisamente porque son, a todos los efectos, constantes de las que podremos coger su valor pero que no se pueden modificar. Los parámetros de una función deben ser siempre de entrada y se expresan separados por comas.

El cuerpo de la función puede ser arbitrariamente complejo, con asignaciones, bucles, llamadas a funciones o acciones, o lo que sea menester. En el cuerpo se pueden usar los parámetros como si fueran constantes. Justo antes de que acabe la ejecución de este cuerpo, es necesario indicar el resultado que se quiere devolver con la instrucción `return`. La forma de uso de esta instrucción es

`return` ⟨*expresión que calcula el resultado por devolver*⟩`;`

Como ejemplo, he aquí una función que calcula $x^n$:

```
double potencia(double const x, int const n) {
  double pot = 1;
  for (int i = 0; i < n; i++) pot = pot * x;
  return pot;
}
```

Las funciones que definimos se utilizan exactamente igual que las funciones de la librería estándar. No hay diferencia porque aquéllas suelen estar definidas también en C++.

En el cuerpo de la función puede haber más de una instrucción `return`; por ejemplo, si es necesario regir el cálculo por medio de un condicional:

```
int signo(int const v) {
  if (v < 0) return -1;
  else if (v > 0) return 1;
  else return 0;
}
```

La instrucción `return` termina inmediatamente la ejecución de la función, que tendrá como resultado el valor de la expresión que la acompaña. Por tanto, no tiene sentido que haya código después de un `return`; los compiladores de C++ razonables suelen informar de este desliz indicando que hay *código muerto*, que nunca se alcanzará. Esta terminación inmediata es útil en diversas circunstancias y permite escribir un código muy legible si se usa adecuadamente. Dedicaremos el resto de este apartado a explorar situaciones que aprovechan correctamente la terminación inmediata.

Supongámonos enfrentados a un problema que tiene casos particulares triviales, irrelevantes o imposibles de tratar. Por ejemplo, la parte entera de la raíz cuadrada de un número. Lo usual es desembarazarse de esos casos devolviendo un valor preacordado:

```
int raizCuadrada(int const n) {
  if (n < 0) return -1;
  int raiz = 0;
  while ((raiz+1) * (raiz+1) <= n) raiz++;
  return raiz;
}
```

Supongámonos enfrentados a un problema de búsqueda; por ejemplo, la determinación del menor factor primo de un número. Es necesario un bucle que recorra el espacio de alternativas. Podremos tener éxito en esta búsqueda o podremos fracasar. En caso de éxito, estaremos en medio de un bucle cuando hayamos encontrado nuestro objetivo; la terminación inmediata nos permite acabar la búsqueda y devolver lo encontrado. Si fracasamos, llegaremos al final del bucle y allí tendremos que devolver algún valor razonable:

```
int menorFactor(int const n) {
  int factor = 2;
  int const limite = raizCuadrada(n);
  while (factor <= limite) {
    if (n % factor == 0) return factor;
    factor++;
  }
  return n;
}
```

Veamos finalmente un caso que mezcla la terminación inmediata y la recursión. Supongámonos enfrentados esta vez a un problema que tiene dos casos, pero que uno se puede expresar en función del otro. Por ejemplo, la potencia, que hemos implementado antes de forma incompleta porque no hemos tenido en cuenta que el exponente puede ser negativo. Afortunadamente, $x^{-n} = 1/(x^n)$. La terminación inmediata nos permite reconvertir el problema, al principio de la función:

```
double potencia(double const x, int const n) {
  if (n < 0) return 1 / potencia(x, -n);
  double pot = 1;
  for (int i = 0; i < n; i++) pot = pot * x;
  return pot;
}
```

## 3.2 Acciones o procedimientos

Las acciones se diferencian de las funciones en que no devuelven un resultado. Sin embargo, les está permitido hacer muchas más cosas. Por ejemplo, pueden leer o escribir. Además, una acción puede tener parámetros de salida, por los que puede devolver resultados, o de entrada y salida, para modificar valores. Una acción se define así:

```
void nombreDeLaAccion(⟨parámetros⟩) {
  ⟨Cuerpo de la acción⟩
}
```

El tipo `void` es un tipo especial que no tiene valores y se utiliza para indicar la carencia de algo: en este caso, de un valor de retorno.

Los parámetros de una acción pueden ser de dos formas: primero, los parámetros *de entrada* que hemos visto para las funciones; segundo, los parámetros *por referencia*, o de (entrada y) salida, por los que la acción puede (recibir y) devolver datos; su sintaxis es ésta:

```
Tipo& nombreDelParametro
```

Cuando se llama a una acción, en el lugar de un parámetro de entrada se puede escribir una expresión cualquiera, mientras que en el lugar de un parámetro por referencia es necesario poner una variable. Dentro de la acción, el parámetro por referencia se convierte en otro nombre para la variable con la que se hizo la llamada; cuando accedemos al valor del parámetro, estamos consultando el valor de la variable; cuando modificamos el parámetro, estamos asignando a la variable. Como ejemplo, esta acción incrementa una variable en una cierta cantidad:

```
void incrementar(int& var, int const delta) {
  var = var + delta;
}
```

Dentro de una acción también se puede utilizar la instrucción `return`. Pero, como no hay que devolver ningún valor, no lleva expresión asociada. Su papel se restringe a la terminación inmediata. No suele tener tanto sentido en acciones como en funciones y, por eso, se utiliza mucho menos. Las situaciones que explicamos para funciones se pueden extrapolar a acciones.

## 3.3 Elementos avanzados

### 3.3.1 Referencias constantes

Un parámetro de entrada implica una copia del valor; por eso, también se conocen como parámetros por valor. Un parámetro por referencia implica el paso de un puntero (que se verán en el capítulo 6). Para los tipos predefinidos que hemos visto hasta ahora, pasar un dato por valor o por referencia tiene un coste similar, porque un puntero es aproximadamente del tamaño de un entero.

En el capítulo 4 veremos cómo se pueden definir nuevos tipos por aglomeración de los primitivos. Cuando estas aglomeraciones crecen, el paso por valor es bastante más caro que el paso por referencia. Sería interesante tener parámetros de entrada por referencia en vez de por valor. Un parámetro de este estilo se declara en C++ juntando en el orden adecuado los calificadores `&` y `const`, verbigracia:

```
Tipo const& parametroDeEntradaPorReferencia
```

Que un parámetro de entrada sea `const` o `const&` resulta indistinguible por el resto del programa; se puede cambiar de uno a otro por meras consideraciones de eficiencia, sin preocuparse porque el programa deje de compilar o cambie su funcionamiento.

### 3.3.2 Sobrecarga

C++ no sólo distingue entre subprogramas por su nombre sino también por el tipo de sus parámetros. Podemos definir varios subprogramas que se llamen igual siempre y cuando sus parámetros sean distintos, ya sea en número o en tipo. C++ distinguirá entre ellos y sabrá cuál utilizar en una cierta llamada analizando el tipo de los valores que pasamos a los parámetros. Es lo que se conoce como *sobrecarga* de identificadores. (Véase el apartado 5.1 para un ejemplo interesante de sobrecarga.)

### 3.3.3 Definición de operadores

En C++ los operadores son meros nombres para subprogramas y aceptan nuevas definiciones. Al dar una nueva definición a un operador (todos los operadores tienen ya uno o varios papeles dentro del lenguaje, así que una nueva definición siempre supone una sobrecarga) hay que prefijar su nombre con la palabra `operator`. (Véase el apartado 4.3 para un ejemplo interesante de definición de operadores.)

## 31 Incierta igualdad de números reales

131

Como quizá la representación de los números reales resulte algo imprecisa, la igualdad de dos reales posiblemente esté sujeta a algún error. Por eso, nos tememos que la operación == predefinida en C++ no será satisfactoria en general. A lo mejor deberíamos escribir una función que considere iguales dos números reales si son lo suficientemente próximos.

## 32 Triángulos rectángulos enteros

Buscamos triángulos rectángulos cuyos lados $a$, $b$ y $c$ sean enteros tales que $a^2 + b^2 = c^2$. Los llamaremos TRE para abreviar, y los escribiremos mediante la terna $\langle a, b, c\rangle$, asumiendo que $a \leq b \leq c$.

Para construirlos, tendremos en cuenta que, si $a$ es impar ($a = 2k + 1$), forma un TRE con los lados $b = 2k(k + 1)$ y $c = b + 1$ y que, si $a$ es par ($a = 2k$), los lados $b = k^2 - 1$ y $c = k^2 + 1$ completan un TRE. (Las fórmulas anteriores se deben a Pitágoras y Platón respectivamente.)

Se pide un programa que genere los TRE correspondientes a los primeros $n$ enteros.

***Un poco de historia*** Pitágoras de Samos (569–475 a.C.) y Platón (427–347 a.C.) son considerados filósofos y matemáticos. En ambas disciplinas destacaron y fundaron escuelas. En general, el pensamiento griego disfrutó de una visión del mundo más amplia que la que actualmente tenemos. En particular, no sufrió la división entre ciencias y letras que predomina en nuestra cultura.

*La tentación pitagórica* [Góm99] es un libro interesante que repasa la historia conjunta de las matemáticas y la filosofía desde Grecia hasta nuestros días.

## 33 Ponle tú el título

131

Contempla detenidamente el siguiente programa:

```
int siguiente(int const nn, int const tope) {
  if (nn == tope) {
    return 0;
  } else {
    return nn+1;
  }
}
int main() {
  int const n = 5;
  int num = 0;
  for (int i = 0; i <= n; i++) {
    for (int k = 0; k <= n; k++) {
      cout << num << ' ';
      if (k != n) num = siguiente(num, n);
    }
    cout << endl;
  }
}
```

¿Qué escribe en la pantalla?

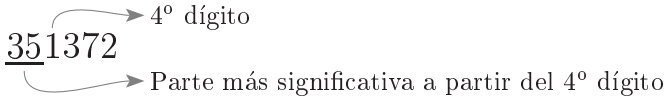
## 3.4 Descomposición de un número

129

En este ejercicio proponemos la definición de funciones sencillas que permitan descomponer un número de múltiples formas.

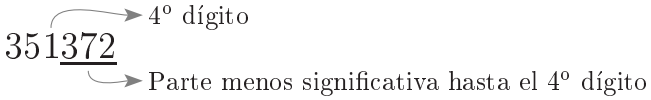
***Parte más significativa*** Escribe una función que devuelva la parte más significativa desde el $n$-ésimo dígito de un número.

***Ejemplo***



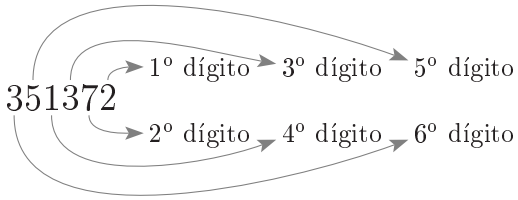

***Parte menos significativa*** Escribe una función que devuelva la parte menos significativa hasta el $n$-ésimo dígito de un número.

***Ejemplo***




***Dígito $n$-ésimo*** Escribe una función que nos devuelva el dígito $n$-ésimo de un número.

***Ejemplo***




***Cualquier base*** Escribe funciones que permitan realizar los apartados anteriores sea cual sea la base con la que estemos trabajando.

***Para este enunciado*** Consulta la pista 3.4a.

## 3.5 Palíndromos

129

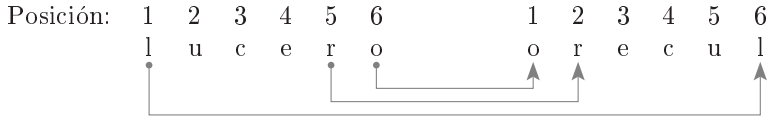
Un palíndromo es una palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda; por ejemplo, la palabra "anilina", o las frases "anula la luz azul a la luna" y "sé verla al revés", despreciando las diferencias entre mayúsculas y minúsculas, los espacios en blanco entre palabras y las tildes.

De igual forma, decimos que la expresión de un número en una determinada base (véase el ejercicio 2.21) es un palíndromo, o capicúa, si dicho número se lee igual de izquierda a derecha que de derecha a izquierda, por ejemplo 81318 en base 10, o el número nueve expresado en base 2, 1001. Obviamente, suponemos que los números siempre están expresados por su representación mínima, es decir, sin ceros redundantes. Por ejemplo, 100 en base 10 no es un palíndromo, ya que no se lee igual de derecha a izquierda de izquierda a derecha; no consideramos expresiones de 100 como 00100, que en principio también son el mismo número, ya que entonces podríamos decir que sí que es un palíndromo.

### 3.5.1 Reverso de caracteres

Escribe una función que tenga como parámetro de entrada una cadena de caracteres y devuelva su reverso.

***Ejemplo*** El reverso de una cadena de caracteres es otra cadena pero cuyas letras están en orden inverso de aparición.




O sea, el reverso de la palabra *lucero* tiene por primera letra la *o*, que es la última de *lucero*, por segunda letra a *r* que es la penúltima, y así sucesivamente.

### 3.5.2 Comprobación de palíndromo

Escribe una función que tenga como parámetro de entrada una cadena de caracteres y nos indique si es un palíndromo o no. Observa que, si se trata de una frase, los espacios que separan las palabras no se tienen en cuenta.

### 3.5.3 Reverso de un número

Escribe una función que tenga como parámetro un número entero y devuelva el reverso de dicho número. (Véase la pista 3.5a.)

### 3.5.4 Número palíndromo

Escribe una función que tenga como parámetro de entrada un número y nos indique si éste es palíndromo o no.

## 3.6 Cuentaletras

131

Un *cuentaletras* es un programa que recibe como entrada una secuencia de palabras y va contando y escribiendo el número de letras de cada una de ellas. En este caso, queremos que el cuentaletras haga su trabajo con el contenido de un fichero de texto.

Supongamos que en un fichero de texto tenemos almacenado el siguiente poema de Manuel Golmayo:

*Soy y seré a todos definible.*
*Mi nombre tengo que daros;*
*cociente diametral siempre inmedible*
*soy de los redondos aros*

el cuentaletras devolvería como resultado: 31415926535897932384.
¿No te resulta familiar esta secuencia de dígitos? Efectivamente, son los 20 primeros dígitos del número $\pi$. ¿Qué te parece la siguiente reflexión en la lengua shona de Zimbabue?:

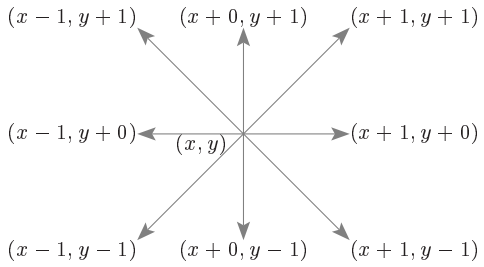
*Iye "P" naye "I" ndivo vadikanwi.*
*"Pi" achava mwana.*

Su autor es Martin Mugochi de la Universidad de Zimbabue y su traducción más o menos es: *"P" e "I" son amantes. "Pi", su hijo, será un cerebrito.* Ante esta frase el cuentaletras debería responder: 314159265. ¡Qué casualidad!

Si quieres encontrar más $\pi$nemotécnicos y $\pi$emas puedes consultar la página: `http://www.cilea.it/~bottoni/www-cilea/F90/piph.htm`.

Proponemos la realización de programas que permitan estudiar los movimientos de una partícula dentro de una determinada superficie geométrica.

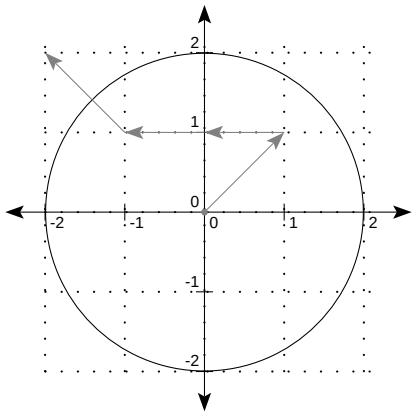
***Movimientos dentro de un círculo*** Supongamos que tenemos una partícula que se mueve de forma aleatoria en el plano en las coordenadas enteras. La partícula puede pasar con la misma probabilidad a cualquiera de las posiciones en las que una o ambas de sus coordenadas actuales varíe de forma entera en $-1$ o 1, como muestra el siguiente gráfico:




Queremos realizar un programa que, dado un círculo y una partícula en el interior de dicho círculo, calcule el número de movimientos que realiza la partícula antes de salirse del círculo.

***Ejemplo*** Consideremos un círculo de radio 2 centrado en el origen. Queremos ver cuántos movimientos realiza una partícula, que inicialmente se halla situada en el origen y que se mueve de forma aleatoria, antes de salirse del círculo.

En el siguiente gráfico podemos ver una secuencia aleatoria de movimientos de la partícula:




La partícula parte del origen de coordenadas (0,0), y realiza el movimiento aleatorio, que consiste en elegir dos números entre $-1$ y 1 que determinarán el movimiento de cada coordenada, en este caso [1,1]. La partícula se mueve y por tanto llega a la posición $(0+1, 0+1) = (1,1)$. El segundo movimiento de la partícula viene dado por la elección de los números [$-1$,0] y por tanto la posición final después de este segundo movimiento es $(1-1, 1-0) = (0,1)$. La partícula realiza un tercer movimiento [$-1$,0] que la lleva a la posición $(0-1, 1-0) = (-1,1)$. Finalmente, el cuarto movimiento [$-1$, 1] lleva a la partícula fuera del círculo, y ahí termina el experimento. Se han necesitado tres movimientos. (Véase la pista 3.7a.)

***Media de movimientos*** Considera el apartado anterior y modifica ligeramente la solución para que podamos hacer, no un único experimento, sino la cantidad que deseemos, y así con ellos calcular la media de movimientos necesarios para salirse de un círculo con un determinado radio.

***Otras figuras geométricas de dos dimensiones*** Escribe un programa general para movimientos en el plano que permita utilizar otras figuras geométricas para delimitar las superficies dentro de las cuales se mueven las partículas. Por ejemplo cuadrados, rectángulos, triángulos, etc.

***Movimiento dentro de un cubo*** Escribe un programa que, dado un cubo y una partícula en su interior, calcule el número de movimientos que la partícula realiza antes de salirse del cubo.

***Otras figuras geométricas de tres dimensiones*** Escribe un programa general para movimientos en el espacio de tres dimensiones que permita utilizar diversas figuras geométricas para delimitar las regiones. Por ejemplo esferas, cilindros, poliedros, etc.

***Para este enunciado*** Consulta la pista 3.7b.

***Bibliografía*** El movimiento aleatorio de partículas se conoce como *movimiento browniano* y tiene interés en distintas áreas, desde la física a los modelos estadísticos de inventarios. Con la ayuda de [Tho00], por ejemplo, puedes ampliar tus conocimientos sobre los modelos estadísticos subyacentes.

***Un poco de historia*** En 1637 René Descartes (1596–1650) publicó *La Géométrie*, donde presenta la geometría analítica, de importancia fundamental para el desarrollo posterior del cálculo por parte de Isaac Newton (1643–1727) y Gottfried Wilhelm Leibniz (1646–1716).

Descartes, matemático, filósofo y científico, escribió en 1637 *El discurso del método* (cuyo subtítulo era, *para bien dirigir la razón y buscar la verdad en las ciencias*), trabajo que marcó de forma decisiva la transición de la ciencia y la filosofía medievales a la edad moderna.

## 38 Generación de primos

Considera las siguientes propiedades relacionadas con la primalidad de un número:

- Un entero superior a 2 sólo puede ser primo si es impar.
- Un entero $n$ superior a 3 sólo puede ser primo si verifica la propiedad $n^2 \bmod 24 = 1$.
- Un entero positivo $n$ es primo si y sólo si no tiene divisores entre 2 y $\lfloor\sqrt{n}\rfloor$.

y, basándote en ellas, escribe las siguientes funciones:

***Filtro: "mod24-1"*** Una función que indique si un entero verifica la propiedad siguiente:

$$n^2 \bmod 24 = 1$$

***Filtro: divisores*** Una función que indique si un número es primo, buscando si tiene algún divisor entre 2 y $\lfloor\sqrt{n}\rfloor$.

***Filtro: primos*** Una función que indique si un número es primo o no, descartando primero los pares, comprobando luego la propiedad "mod24-1," y finalmente buscando sus posibles divisores.

***Bibliografía*** La literatura sobre los números primos es muy abundante. La referencia [Pom83] es un interesante punto de partida para ampliar información sobre este tema.

## 39 Conjetura de Goldbach

En una carta escrita a Leonhard Euler en 1742, Christian Goldbach (1690–1764) afirmó (sin demostrarlo) que todo número par es la suma de dos números primos. Para poner a prueba esta conjetura (hasta cierto punto, claro está), basta con avanzar a través de los primeros $n$ números pares hasta encontrar uno que no verifica esa propiedad, o hasta llegar al último, ratificándose la conjetura hasta ese punto,

```
int k = 0; bool seCumpleHastaK = true;
while (seCumpleHastaK && k <= n) {
  k = ⟨el siguiente par⟩;
  ⟨Tantear la descomposición de k⟩
  if (⟨falla el intento⟩) {
    ⟨seCumpleHastaK se anota como falso⟩
  }
}
```

donde cada tanteo se puede expresar mediante un subprograma que responda a la siguiente llamada,

```
descomponer(numPar, conseguido, sumando1, sumando2);
```

esto es, un subprograma que, dado un entero `numPar` (supuestamente positivo y par), busca una descomposición del mismo en dos sumandos `sumando1` y `sumando2` primos, indicando además si lo ha `conseguido` o no. Esa descomposición de `numPar` se busca probando pares de sumandos,

$$(1, \texttt{numPar} - 1), (2, \texttt{numPar} - 2), \ldots$$

hasta que ambos sean primos o bien el primero supere al segundo, para no repetir los tanteos finales,

$$\ldots, (\texttt{numPar} - 2, 2), (\texttt{numPar} - 1, 1)$$

que son iguales a los iniciales.

***Descomposición*** Escribe en C++ el subprograma anterior, suponiendo que existe una función que verifique la primalidad de un número. (De hecho, se ha desarrollado en el ejercicio 3.8.)

***Conjetura de Goldbach*** Finalmente, desarrolla el programa correspondiente al algoritmo completo, siguiendo los pasos descritos.

***Bibliografía*** Esta conjetura aparece recogida en las *Meditaciones algebraicas* de Edward Waring (1734–1793) junto con otros resultados interesantes. Te proponemos desarrollar un programa que permita comprobar los siguientes enunciados (pruébalos con los primeros millares de los números naturales):

- Conjetura: todo entero impar es un número primo o la suma de tres números primos.
- Teorema (Leonhard Euler, 1707–1783): todo entero positivo es la suma de, a lo más, cuatro cuadrados.
- Teorema (John Wilson, 1741–1793): para todo primo $p$, el número $(p-1)! + 1$ es múltiplo de $p$.

En la novela [Dox00] se relata la historia de un matemático que únicamente vivió para intentar demostrar la conjetura de Goldbach.

## 3.10 El factorial en la sociedad del futuro

Ahora hay mucha policía; nos aseguran que es necesaria. Mi abuela me ha contado que antes sólo existían dos o tres policías, distintas pero iguales; ahora hay una sola, pero con tantas personas que necesitan vigilarse mutuamente. Por eso está organizada jerárquicamente, y los de un nivel vigilan a los del siguiente hasta que se llega a la ciudadanía. Nadie conoce el primer nivel; no se sabe cuántas personas lo forman; dicen que se llama CIA y que vigila al siguiente nivel. El segundo nivel se llama BICIA; tiene una sola persona (dicen que es un rey) que vigila a las dos personas del siguiente nivel. El tercer nivel se llama TRICIA; cada uno de sus dos miembros vigila a tres personas en el siguiente nivel. Así se sigue, por varios niveles más de los que poca gente conoce el nombre; siempre se cumple que una persona del nivel $n$ vigila

a $n+1$ personas del siguiente nivel, y que a cada persona del nivel $n+1$ sólo la vigila una del nivel $n$; se deduce que en cada nivel hay $(n-1)!$ personas. Al último nivel lo llamamos POLICIA porque no sabemos exactamente a cuántas personas tocan. Las personas que quedamos somo ciudadanos, aunque se está extendiendo el término CIADAANO. Mi abuela dice que antes se podía trabar conversación con cualquier persona desconocida; que antes los policías tenía un carnet y un traje especial. Ahora no es así; sólo hay una documentación, no hay una ropa especial. En busca de la prima de productividad, la policía para intempestivamente a la gente por la calle. En un principio hubo muchos altercados y tiroteos entre la misma policía; luego se impuso la costumbre de gritar el nivel para que los demás supiéramos por quién tomar partido; pero ahora, para hacerlo todo más sutil, se grita el número de la documentación. Afortunadamente la documentación está numerada consecutivamente por niveles; es fácil hacerse una idea de quién manda más, pero como los niveles son tan grandes, es difícil saber si dos números son del mismo nivel o los separan muy pocos niveles. El otro día me pararon el agente número 47335 y la agente número 80157; me trataron de muy malos modos; no sabía que sólo estaban un nivel por encima de mí y que les podía haber interpuesto una denuncia. Ahora voy a escribir un programa en mi microcomputadora portátil para auxiliarme la próxima vez; le daré dos números de documentación y me dirá cuántos niveles los separan.

## 311 Sucesión bicicleta

134

Consideremos las sucesiones cuyos términos están definidos del siguiente modo:

$$b_1, b_2 \in \mathbb{R}^+$$

$$b_n = \frac{b_{n-1}+1}{b_{n-2}}, \quad \forall n \geq 3$$

Al igual que ocurre con la sucesión de Fibonacci se observa que, para avanzar a través de sus términos, basta con hacer *rodar* dos de ellos consecutivos. Por ejemplo, si tomamos los dos primeros términos 2 y 3, se obtienen los siguientes, $2, 3, 2, 1, 1, 2, 3, \ldots$ que, sólo por casualidad, son todos números enteros. Por otra parte, es fácil verificar que, para dos términos iniciales cualesquiera de $\mathbb{R}^+$, la sucesión generada mediante la relación recurrente anterior es cíclica,

$$\forall b_1, b_2 \in \mathbb{R}^+, \exists p \in \mathbb{N} \text{ tal que } \forall n \geq 1, b_{n+p} = b_n$$

por lo que muy bien podría ser llamada *bicicleta*.

***Periodo*** Escribe una función que calcule el período $p$ de una sucesión dada por sus dos primeros elementos.

***Serie*** Escribe ahora un programa que halle la suma de los $n$ primeros términos de dicha sucesión.

## 312 Cotejo de $n$-gramas

Si quisiéramos determinar si dos palabras son iguales, podríamos plantearnos comparar uno a uno sus caracteres y, si todos coinciden, concluir que las palabras son *iguales*. Sin embargo, este mecanismo sólo nos indica si dos palabras o frases son iguales o no pero, en el caso de que no lo sean, no nos dice en qué medida son diferentes.

Una primera idea para determinar el grado de similitud de dos palabras puede ser contar el número de caracteres que tienen en común. Por ejemplo, las palabras *listo* y *cínico* comparten 2 caracteres. No obstante, esta idea no nos aporta información sobre si la distribución de esos 2 caracteres comunes es similar en las dos palabras. Para obtener este tipo de información podemos recurrir a compararlas cotejando *n-gramas*. Veamos de qué se trata.

Un $n$-grama es una secuencia de $n$ caracteres. Si nos referimos a una secuencia de dos caracteres podemos hablar de bigrama; si la secuencia es de tres caracteres, trigrama. Comprobemos con el ejemplo anterior cuál es el resultado si comparamos las palabras mediante bigramas. Compararíamos los bigramas de *listo* (*li*, *is*, *st*, *to*) con los de *cínico* (*ci*, *in*, *ni*, *ic*, *co*) para concluir que no tienen ningún bigrama en común; luego los 2 caracteres comunes no dan lugar a ninguna secuencia de dos caracteres consecutivos en común.

Otro ejemplo puede ilustrar aún más la diferencia entre ambas estrategias. Las palabras *loco* y *comida* comparten también 2 caracteres y, además, 1 bigrama. Es decir, no sólo tienen caracteres en común sino que éstos tienen un cierto grado de similitud en su distribución en ambas palabras.

***N*-gramas** Escribe un programa que dadas dos palabras las coteje utilizando $n$-gramas e indique, además, el número de caracteres que tienen en común. El valor de $n$ lo propondrá el usuario teniendo en cuenta que no debería ser mayor que el número de caracteres de la palabra más corta.

***Coeficiente de Dice*** Vamos a añadir una medida de similitud: el coeficiente de Dice. Este coeficiente se utiliza a menudo para determinar la similitud entre elementos con partes constituyentes. Se calcula así,

$$CD(p_1, p_2) = \frac{2 \times \text{comun}(p_1, p_2)}{\text{elementos}(p_1) + \text{elementos}(p_2)} \in [0, 1]$$

siendo $\text{comun}(p_1, p_2)$ el número de partes constituyentes ($n$-gramas o caracteres según la aproximación que utilicemos) que $p_1$ y $p_2$ tienen en común, y $\text{elementos}(p_1), \text{elementos}(p_2)$ el número de partes constituyentes de $p_1$ y $p_2$ respectivamente.

La fórmula anterior proporciona un valor real en el rango $[0, 1]$. Si las dos palabras no tienen ningún $n$-grama en común obtendrán un coeficiente de Dice igual a 0, mientras que dos palabras idénticas tendrán un coeficiente de Dice igual a 1.

Amplía el programa desarrollado en el primer apartado para que incluya el cálculo del coeficiente de similitud en las dos estrategias que hemos considerado: caracteres y $n$-gramas.

***Bibliografía*** El coeficiente de similitud de Dice [Dic45] fue ideado para medir el grado de asociación entre especies; sin embargo se utiliza con profusión en algunas tareas de procesamiento de lenguaje natural, como determinar el grado de asociación entre términos, oraciones o incluso documentos. También se utiliza para medir la similitud entre dos mapas de bits de igual longitud, comparar grafos conceptuales, moléculas, etc.

## 3.13 Simulación de variables aleatorias

> Como todos los hombres en Babilonia, he sido procónsul; como todos, esclavo; también he conocido la omnipotencia, el oprobio, las cárceles. Miren: a mi mano derecha le falta el índice. [...] He conocido lo que ignoran los griegos: la incertidumbre. [...] Debo esta variedad casi atroz a una institución que otras repúblicas ignoran o que obra en ellas de un modo imperfecto: la lotería.
>
> *La lotería en Babilonia*, Jorge Luis Borges

### 3.13.1 Introducción

Casi todos los lenguajes de programación proporcionan recursos para simular variables aleatorias. Normalmente, dichos recursos consisten en funciones para generar variables aleatorias uniformes, ya que con ellas se puede simular cualquier otra.

Aquí se resumen los recursos mínimos que proporciona C++, y se explica cómo manejar variables uniformes para construir otras. Justamente eso es lo que se propone al final: usar los recursos dados para implementar distintas variables aleatorias.

### 3.13.2 Los recursos en C++

En C++, existen distintos juegos de recursos para generar números aleatorios: unos son estándar y otros no; unos tienen más limitaciones que otros; algunos tienen un uso más cómodo que otros. Sin mayores discusiones, consideraremos la colección de funciones que tiene `stdlib.h` como fichero de cabeceras.

La función `int rand()` genera un número seudoaleatorio entre `0` y `RAND_MAX`. Por ejemplo, la expresión `rand() % N` da un número natural aleatorio, entre `0` y `N-1` (siempre que $N > 0$). La expresión `(double)rand()/RAND_MAX` da un número aleatorio real, del intervalo $[0, 1]$. (Nótese que la conversión a real de alguno de los miembros de la segunda expresión es necesaria para que la división se realice en $\mathbb{R}$.)

Aunque desde un punto de vista estrictamente teórico la primera expresión no se distribuye uniformemente, desde un punto de vista práctico, cuando $N$ es mucho menor que `RAND_MAX`, se obtienen unos resultados de uniformidad aceptables.

En principio, la secuencia de números que proporciona la función `rand()` es siempre la misma; para que cambie de una ejecución a otra, se ha de dar un valor inicial distinto. Este valor se conoce como *semilla*; por ejemplo, con la instrucción siguiente:

```
srand(time(NULL));
```

El empleo del reloj del sistema para proporcionar una semilla es habitual. Para poder usarlo es necesario cargar el fichero de cabeceras `time.h`.

### 3.13.3 Uso de estos recursos a ojo

Jugando con la función dada en el apartado anterior, pueden obtenerse expresiones aleatorias variadísimas. Por ejemplo, con expresiones como las de la figura 3.1, se pueden simular la aleatoriedad de un dado justo (o sea, equiprobable) de seis caras; un dado de caras entre $a$ y $b$; un número real extraído uniformemente del intervalo $[a, b]$; una moneda justa, donde cara y cruz salen con igual probabilidad; una moneda trucada, donde la cara sale con una probabilidad distinta que la cruz, etc.

| | |
|---|---|
| `rand()%2 + 1` | `(double)rand()/RAND_MAX < p` |
| `rand()%6 + 1` | `(3*rand()) % 5 + 10` |
| `rand()%(b-a+1) + a` | `(b - a) * ((double)rand()/RAND_MAX) + a` |

Figura 3.1: Expresiones aleatorias

### 3.13.4 El método de la transformada inversa para variables aleatorias continuas

Pero generar una variable aleatoria arbitraria requiere cierta dosis de perspicacia. En este apartado se introduce el método de la *transformada inversa*. En primer lugar se presenta este método para variables aleatorias continuas.

Necesitamos el siguiente teorema, cuya demostración es trivial:

Si $F : \mathbb{R} \to [0, 1]$ es una función de distribución cualquiera, se tiene la siguiente equivalencia:

$$Y \sim U[0, 1] \Leftrightarrow F^{-1}(Y) \sim F$$

Es decir, si la variable aleatoria $Y$ sigue una distribución uniforme en $[0, 1]$, la variable aleatoria $F^{-1}(Y)$ sigue una función de distribución $F$, y viceversa.

El método es entonces una mera aplicación de dicho teorema: para simular una variable aleatoria $X$ de función de distribución $F$, podemos generar primero una $Y \sim U[0, 1]$, y con ella calcular $X = F^{-1}(Y)$. El teorema anterior nos garantiza que $X \sim F$. La figura 3.2 resume la situación:

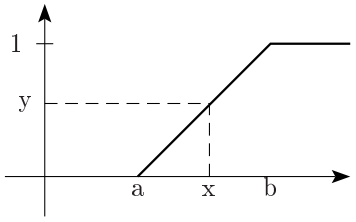



Figura 3.2: Función de distribución de una variable aleatoria continua

Supongamos que deseamos generar una *variable aleatoria uniforme* en el intervalo $[a, b]$. Como su función de distribución es,

$$F(x) = \begin{cases} 0 & \text{si } x \leq a \\ \frac{x-a}{b-a} & \text{si } a \leq x \leq b \\ 1 & \text{si } b \leq x \end{cases}$$

para $y \in [0, 1]$ se tiene $F^{-1}(y) = a + y * (b - a)$. Por lo tanto, si se genera $y \sim U[0, 1]$, basta con hallar $x = a + y * (b - a)$:

```
double y = (double)rand()/RAND_MAX;
double x = a + y*(b-a);
```

y se tiene $x \sim U[a, b]$.

### 3.13.5 El caso discreto

El método anterior se adapta a variables discretas fácilmente: sea la función de probabilidad $Prob(x = i) = p_i$, para $1 \leq i \leq n$; su función de distribución es $P(k) = \sum_{i=1}^{k} p_i$ para $1 \leq k \leq n$, y expresa la probabilidad de que $x \leq i$. Entonces, el método consiste en generar la variable aleatoria $y \sim U[0, 1]$, y hallar el mínimo $k$ tal que $P(k) \geq y$. Supongamos que se desea simular un dado que dé los números entre 1

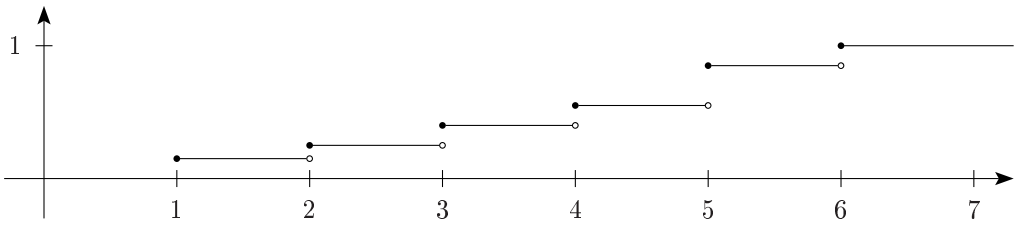



Figura 3.3: Función de distribución de una variable aleatoria discreta

y 6, y que la probabilidad $p_i$ de obtener cada número sea $0.15, 0.1, 0.15, 0.15, 0.3$ y $0.15$ respectivamente. Para ello, se hallan las cantidades $P(i)$ por acumulación: $0.15, 0.25, 0.40, 0.55, 0.85$ y $1.00$. (Véase la figura 3.3.) Luego se genera $y \sim U[0, 1]$,

```
double y = (double)rand()/RAND_MAX;
```

y, finalmente, basta con hallar el $\min\{k$ tal que $P(k) \geq y\}$. Si las cantidades $P(i)$ se almacenaron en un array, esta búsqueda se puede realizar por inspección de la tabla, que tiene el contenido de la columna $P(i)$:

| $i$ | $p_i$ | $P(i)$ |
|---|---|---|
| 1 | 0.15 | 0.15 |
| 2 | 0.1 | 0.25 |
| 3 | 0.15 | 0.40 |
| 4 | 0.15 | 0.55 |
| 5 | 0.3 | 0.85 |
| 6 | 0.15 | 1.00 |

Si, por ejemplo, la instrucción `double y = (double)rand()/RAND_MAX;` genera el valor 0.75, hay que buscar el menor número $k$ que verifique que $P(k) \geq 0.75$. En este caso la cara $k$ del dado es el cinco.

### 3.13.6 Expresiones anteriores

Calcula el conjunto de valores que puede producir cada una de las expresiones de la figura 3.1.

### 3.13.7 Uniforme continua

Deduce en C++ una función que genere una variable aleatoria uniforme real, del intervalo $[a, b]$.

### 3.13.8 Uniforme entera

Deduce en C++ una función que genere una variable aleatoria uniforme entera, del conjunto de números $\{a, \ldots, b\}$.

### 3.13.9 Variable aleatoria continua

Escribe una función que genere una variable aleatoria continua de función de distribución $F$, definida así:

$$F(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ \sqrt{x} & \text{si } 0 \leq x \leq 1 \\ 1 & \text{si } x > 1 \end{cases}$$

### 3.13.10 Variable aleatoria exponencial continua

Escribe una función que genere una variable aleatoria continua de función de densidad exponencial de parámetro $\lambda$; esto es, $f(x) = \lambda e^{-\lambda x}$. Recuerda que la función de distribución $F$ de una función de densidad $f$ viene dada por $F(x) = \int_{-\infty}^{x} f(t)dt$.

### 3.13.11 Dado infinito

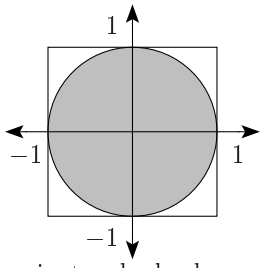
Escribe una función para simular un dado con infinitas caras, de forma que saque un 1 con probabilidad 1/2; un 2 con probabilidad 1/4; un 3 con probabilidad 1/8; etc.

### 3.13.12 Bibliografía

El método de la transformada inversa se estudia en multitud de textos, como [Yak77, Mor84, PV87] entre otros, junto con otros métodos generales, como el del rechazo y el de la composición.

## 3.14 Aproximación hacia $\pi$ con dardos

Considera el siguiente experimento: se dispone de una diana de radio unidad centrada en el origen de coordenadas y del cuadrado en el cual se inscribe dicha diana:




Si se efectúa un buen número de lanzamientos de dardos, uniformemente distribuidos en el cuadrado circunscrito $[-1,1]\times[-1,1]$, el número de los que caerán en la diana será aproximadamente proporcional a su superficie:

$$\frac{\text{número de disparos dentro}}{\text{número de disparos total}} \simeq \frac{\text{superficie de la diana}}{\text{superficie del cuadrado}}$$

y, como sabemos que

$$\frac{\text{superficie de la diana}}{\text{superficie del cuadrado}} = \frac{\pi}{4}$$

se puede estimar que

$$\pi \simeq 4\frac{\text{número de disparos dentro}}{\text{número de disparos total}}$$

aproximación que será, probablemente, tanto más precisa cuanto mayor sea el número de lanzamientos.

Se pide un programa que efectúe un buen número de lanzamientos, que tantee los aciertos y deduzca de ahí una aproximación de $\pi$.

***Bibliografía*** La idea de este enunciado y otras sobre simulación de variables aleatorias pueden ampliarse en [Ben84, Dew85a], entre otras muchas referencias.

## 3.15 Conjetura para la formación de palíndromos

129

Consideremos el siguiente procedimiento:

> Dado un número, lo sumamos a su reverso. Si esta suma es un palíndromo, entonces paramos; y si no, repetimos el proceso con el número obtenido de dicha suma, hasta dar con un palíndromo.

Una curiosa conjetura de teoría de números afirma que, partiendo de cualquier número expresado en base 10, el procedimiento anterior para, y por tanto nos lleva a un palíndromo.

***Ejemplo*** Aquí vemos un ejemplo de cómo funciona la conjetura. Supongamos que partimos del número 59; lo sumamos a su reverso y obtenemos 154. Repetimos la operación con 154: la suma con su reverso es 605. Por último, al sumar 605 a su reverso obtenemos un palíndromo:

$$59 \;\rightarrow\; \begin{array}{r} 59 \\ +\;\;95 \\ \hline 154 \end{array} \;\rightarrow\; \begin{array}{r} 154 \\ +\;451 \\ \hline 605 \end{array} \;\rightarrow\; \begin{array}{r} 605 \\ +\;506 \\ \hline 1111 \end{array} \;\rightarrow\; 1111$$

***Conjetura*** Escribe un programa que lleve a cabo el procedimiento descrito por la conjetura para encontrar un palíndromo a partir de un número. (Véase la pista 3.15a.)

***Terminación*** Una conjetura es una hipótesis no demostrada ni refutada. La conjetura que estamos considerando describe un método que, en caso de terminar, conduce a un palíndromo a partir de un número. Aunque el método en general termina, con algunos números no se sabe qué ocurre: por ejemplo, con el 196 se han realizado centenares de iteraciones pero no se ha conseguido llegar a un palíndromo.

Escribe un programa que lleve a cabo el procedimiento descrito por la conjetura para encontrar un palíndromo a partir de un número. El número de iteraciones tiene que limitarse para así evitar los desbordamientos y asegurar que el programa termina.

***Bibliografía*** Desde que hay personas que piensan y demuestran ha habido conjeturas. Las conjeturas son misteriosas y siempre han generado leyendas y mitos. Muchas conjeturas en matemáticas son más populares que los teoremas más importantes. En el ejercicio 3.9 encontrarás una de las conjeturas más conocidas. La conjetura de la formación de palíndromos aparece en [Gar95], un libro ligero, ameno y divertido del maestro de la divulgación matemática Martin Gardner (1914–).

## 3.16 Juegos perdedores ganan 129

Queremos realizar un programa que permita simular juegos de azar y así observar su evolución a largo plazo. Para ello nos vamos a ir a nuestro casino particular en el que se encuentran los siguientes juegos:

***Casi iguales pero no tanto*** El juego de *casi iguales pero no tanto* tiene las siguientes reglas: se lanza una moneda en la que sale cara con una probabilidad del 49.5% y cruz con una del 50.5%; el jugador gana un punto si sale cara y pierde un punto si sale cruz.

Escribe un programa que simule el juego a largo plazo, y que muestre los datos para que se aprecie la evolución de las ganancias o pérdidas del jugador.

***Por tres es al revés*** El juego de *por tres es al revés* se juega con dos monedas $M_1$ y $M_2$. Al lanzar la primera de ellas, $M_1$, sale cara con un 9.5% de probabilidad y cruz con un 90.5%; al lanzar la segunda, $M_2$, sale cara con una probabilidad del 75.5% y cruz con el 24.5%. El jugador gana un punto si sale cara y pierde un punto si sale cruz. Si el capital con el que cuenta el jugador es múltiplo de tres, se lanza la moneda $M_1$, y si no se lanza la moneda $M_2$. El jugador puede comenzar con un capital arbitrario, en puntos.

Escribe un programa que simule el juego a largo plazo y que muestre los datos para que se aprecie la evolución de las ganancias o pérdidas del jugador.

***Mezclando juegos*** Si has realizado los ejercicios anteriores, te habrás dado cuenta de que nuestro casino es un negocio rentable. En los juegos que hemos propuesto el jugador lleva la peor parte y, a largo plazo, siempre acaba perdiendo.

Supongamos que abreviamos por C al juego *casi iguales pero no tanto* y por T al juego *por tres es al revés*. Se abre una nueva mesa de juego en nuestro casino que consiste en alternar jugadas a los dos juegos anteriores de la siguiente forma CCTTCCTTCCTT..., es decir dos jugadas al juego C, dos jugadas al juego T... ¿Apostarías tus puntos a este nuevo juego?

Escribe un programa que simule el juego a largo plazo y que muestre los datos para que se aprecie la evolución de las ganancias o pérdidas del jugador. El jugador puede comenzar con un capital arbitrario, en puntos. ¿Te sorprende el resultado?

***Para este enunciado*** Consulta la pista 3.16a.

***Bibliografía*** El resultado de mezclar juegos perdedores para encontrar un juego ganador es muy reciente y se conoce como *paradoja de Parrondo*. El propio autor, Juan Parrondo (1964–), lo explica muy claramente en el artículo [Par01]. En realidad, no se trata de una verdadera *paradoja* matemática, sino de un resultado sorprendente. El libro *Fotografiando las matemáticas* [Mar00] ofrece cuidadas fotografías que nos muestra cómo las matemáticas aparecen en muy diversos ámbitos de la vida. Uno de los cincuenta artículos que aglutina esta obra está dedicado al trabajo de Parrondo.

## 3.17 La tabla de Galton

Se dispone de una tabla ligeramente inclinada y cubierta con pivotes, representados mediante "•" en la figura 3.4. Los pivotes se hallan situados en forma triangular como puede verse en la figura, con tan perfecta regularidad que la caída de las bolas (representadas por los símbolos "∘") puede producirse en todos ellos a la izquierda o a la derecha, equiprobablemente. Y las bolas se van recogiendo al final en celdas, formando las columnas de un auténtico histograma.

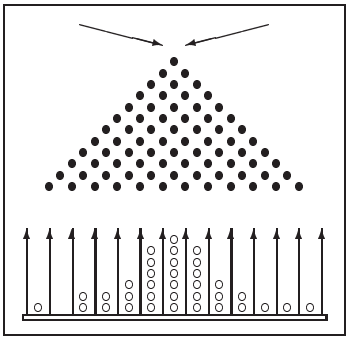


Figura 3.4: La tabla de Galton

Se pide simular el experimento con un número de bolas suficientemente grande (10 000 por ejemplo) y dibujar el histograma que resulte. ¿A que se parece a una campana de Gauss?

***Bibliografía*** Para saber más acerca de simulación de variables aleatorias puedes consultar las referencias bibliográficas que aparecen en el ejercicio 3.14.

***Un poco de historia*** El enunciado del problema toma su nombre de Francis Galton (1822–1911), explorador y antropólogo, sobrino de Charles Darwin (1809–1882). Aunque no fue un gran matemático contribuyó al desarrollo de la estadística. Por el contrario Johann Carl Friedrich Gauss (1777–1855) es uno de los más grandes matemáticos de la historia con aportaciones a multitud de áreas como la teoría de números, análisis matemático, teoría de probabilidades, geometría, fisica, astronomía y geodesia. Se cuenta que siendo niño le mandaron como castigo sumar todos los números entre 1 y 100. Gauss realizó los deberes al momento, al darse cuenta de que la suma total era igual a multiplicar 50 por 101.

## 3.18 El retrato robot

Debido a las dificultades de la policía para identificar a los vagos y maleantes, hace tiempo se decidió sustituir a los dibujantes tradicionales por un programa que efectúa el retrato robot a partir de una descripción del delincuente.

***En fichero*** Un primer prototipo del programa requería descripciones a muy bajo nivel, donde cada línea del dibujo se consigna indicando cuántas veces aparece cada carácter, como se puede apreciar a continuación:

```
1   9 W                                       WWWWWWWWW
1   1 | 2   1 o 1   1 o 2   1 |               |  o o  |
1 @ 4   1 U 4   1 @                         @    U    @
1   1 | 2   3 = 2  1 |                       |  ===  |
2   1 \ 5 _ 1 /                               \_____/
```

Escribe un programa que lea una serie de líneas de un archivo y confeccione el dibujo correspondiente.

***Rasgos independientes*** En una segunda versión, se buscaba facilitar la descripción, ofreciendo el programa diferentes peinados, ojos, orejas y expresión para que el testigo ocular los escogiese, fijando los rasgos del sujeto. Por ejemplo, los peinados disponibles son los siguientes:

```
Pelo denso    WWWWWWWWW
Pelo escaso   |||||||||
Rapado        |"""""""""|
A raya        \\\//////
```

y residen en un archivo que tiene la siguiente disposición:

```
1  9 W
1  9 |
1  1 | 7 " 1 |
1  3 \ 6 /
```

Confecciona un programa que presente las distintas opciones para cada rasgo, para que el usuario elija una, y dibuje el individuo seleccionado.

```
Estilos de pelo disponibles:
    1.- Pelo denso   WWWWWWWWW
    2.- Pelo escaso  |||||||||
    3.- Rapado       |"""""""""|
    4.- A raya       \\\//////
Escoja opción: _
```

***Rasgos en caras*** Con todo, la identificación de rasgos separados no siempre resultaba fácil. Por eso se confeccionó la versión definitiva, que presentaba cuatro individuos, como los siguientes,

```
 WWWWWWWWW     \\\//////     |"""""""""|     |||||||||
 |  0 0  |     |-(. .)-|     |-(o o)-|     |  \ /  |
@    J    @   {    "    }   [    j    ]   <    -    >
 |  ===  |     |   -   |     |  ___  |     |  ---  |
  \_____/       \_____/       \_____/       \_____/

 -- n. 1 --    -- n. 2 --    -- n. 3 --    -- n. 4 --
```

para que el testigo describiese al sospechoso combinando los ojos de un modelo, las orejas de otro y así sucesivamente. Desarrolla esta versión final, donde ahora el menú consiste en mostrar los retratos de referencia y en pedir las correspondientes elecciones:

```
Escoja los rasgos:
   Pelo:            _
   Ojos:            _
   Orejas/nariz:    _
   Boca:            _
```

***Bibliografía*** La idea de este enunciado está tomada de [AR95].

## 3.19 Número de ceros en que termina un factorial

135

Para conocer el número de ceros finales del factorial de un número, $M! = 1 \times 2 \times 3 \times \cdots \times M$, sin calcular dicho factorial, puede emplearse el siguiente método:

- Por cada factor $i \in \{1, \ldots, M\}$ que sea múltiplo de 5, pero no de $5^2$, resulta un cero.
- Por cada factor $i \in \{1, \ldots, M\}$ que sea múltiplo de $5^2$, pero no de $5^3$, resultan dos ceros.
- Y en general, por cada factor $i \in \{1, \ldots, M\}$ que sea múltiplo de $5^i$, pero no de $5^{i+1}$, resultan $i$ ceros.

***Grado de multiplicidad*** Escribe una función (iterativa) que, dados los enteros $n \geq 1$ y $d \geq 2$, halla el grado de multiplicidad de $d$ en $n$; esto es, el número $i \geq 0$ tal que $d^i$ es divisor de $n$, pero $d^{i+1}$ no lo es. Por ejemplo, `grado`$(2, 24) = 3$, ya que $2^3$ es divisor de 24, y en cambio $2^4$ no lo es.

***Ídem, recursiva***

***Multiplicidad de 5*** Usando una cualquiera de las definiciones anteriores, escribe una función que, dado $n \geq 1$, halla el grado de multiplicidad de $n$ respecto de 5. Por ejemplo, `grado5`$(250) = 3$, ya que $5^3$ es divisor de 250, y en cambio $5^4$ no lo es.

***Los ceros de un factorial, al fin*** Usando el apartado anterior, escribe una función que, conocido el entero $n \geq 1$, halla el número de ceros finales de su factorial.

## 3.20 Representación de un número con palabras

Se pide escribir un procedimiento que, dado un número natural, escriba por pantalla la *lectura* de ese número en castellano y en femenino. Por ejemplo, al llamar al procedimiento con el número `1204` obtendremos `mil doscientas cuatro`. Como muestra, los siguientes números: cero, una, dos, tres, cuatro, cinco, seis, siete, ocho, nueve, diez, once, doce, trece, catorce, quince, dieciséis, diecisiete, dieciocho, diecinueve, veinte, veintiuna, veintidós, veintitrés, veinticuatro, veinticinco, veintiséis, veintisiete, veintiocho, veintinueve, treinta, treinta y una, treinta y dos, treinta y tres, treinta y cuatro, treinta y cinco, treinta y seis, treinta y siete, treinta y ocho, treinta y nueve, cuarenta y siete, trescientas sesenta y cuatro, seiscientas ochenta y una, novecientas noventa y ocho, mil trescientas quince, mil seiscientas treinta y dos, mil novecientas cuarenta y nueve, dos mil doscientas sesenta y seis, dos mil quinientas ochenta y tres, dos mil novecientas, tres mil doscientas diecisiete, tres mil quinientas treinta y cuatro, tres mil ochocientas cincuenta y una, cuatro mil ciento sesenta y ocho, cuatro mil cuatrocientas ochenta y cinco, cuatro mil ochocientas dos, cinco mil ciento diecinueve, cinco mil cuatrocientas treinta y seis, cinco mil setecientas cincuenta y tres, seis mil setenta, seis mil trescientas ochenta y siete, seis mil setecientas cuatro, siete mil veintiuna, siete mil trescientas treinta y ocho, siete mil seiscientas cincuenta y cinco, siete mil novecientas setenta y dos, ocho mil doscientas ochenta y nueve, ocho mil seiscientas seis, ocho mil novecientas veintitrés, nueve mil doscientas cuarenta, nueve mil quinientas cincuenta y siete, nueve mil ochocientas setenta y cuatro, once mil cuatrocientas tres, ciento cuatro mil quinientas setenta y ocho, ciento noventa y siete mil setecientas cincuenta y tres, doscientas noventa mil novecientas veintiocho, trescientas ochenta y cuatro mil ciento tres, cuatrocientas setenta y siete mil doscientas setenta y ocho, quinientas setenta mil cuatrocientas cincuenta y tres, seiscientas sesenta y tres mil seiscientas veintiocho, setecientas cincuenta y seis mil ochocientas tres, ochocientas cuarenta y nueve mil novecientas setenta y ocho, novecientas cuarenta y tres mil ciento cincuenta y tres, un millón treinta y seis mil trescientas veintiocho, un millón ciento veintinueve mil quinientas tres, un millón doscientas veintidós mil seiscientas setenta y ocho, un millón trescientas quince mil ochocientas cincuenta y tres, un millón cuatrocientas

nueve mil veintiocho, un millón quinientas dos mil doscientas tres, un millón quinientas noventa y cinco mil trescientas setenta y ocho, un millón seiscientas ochenta y ocho mil quinientas cincuenta y tres, un millón setecientas ochenta y una mil setecientas veintiocho, un millón ochocientas setenta y cuatro mil novecientas tres, un millón novecientas sesenta y ocho mil setenta y ocho, dos millones sesenta y una mil doscientas cincuenta y tres, dos millones ciento cincuenta y cuatro mil cuatrocientas veintiocho, dos millones doscientas cuarenta y siete mil seiscientas tres, dos millones trescientas cuarenta mil setecientas setenta y ocho, dos millones cuatrocientas treinta y tres mil novecientas cincuenta y tres, dos millones quinientas veintisiete mil ciento veintiocho, dos millones seiscientas veinte mil trescientas tres, dos millones setecientas trece mil cuatrocientas setenta y ocho, dos millones ochocientas seis mil seiscientas cincuenta y tres, dos millones ochocientas noventa y nueve mil ochocientas veintiocho, dos millones novecientas noventa y tres mil tres, cien millones trescientas cuarenta y cinco mil seiscientas cinco, y trescientos veinticinco millones novecientas cuarenta y tres mil quinientas ochenta y seis... *y entró en el pueblo a pie, contando sus pasos, para que cada uno tuviera un nombre, y para no olvidarlos nunca más.*

*Seda*, Alessandro Baricco

## 3.21 ¿Cuál es el mejor orden para recibir los datos de un polinomio?

129 136

Junto con un amigo, que no ha leído este libro pero que sabe mucho Fortran, asistes a una entrevista con un pequeño empresario que necesita un programa para evaluar polinomios. Al empresario le gustaría un programa que, para evaluar el polinomio

$$a_n x^n + \cdots + a_1 x + a_0$$

en un cierto valor de $x$, leyera todos los datos en una línea con el orden $a_n$, ..., $a_1$, $a_0$, $x$. Intentáis convencerlo para que se conforme con un programa que lea primero el valor de la variable $x$ y luego los coeficientes $a_i$, ya sea así,

$$x, a_0, a_1, \ldots, a_n$$

o así:

$$x, a_n, a_{n-1}, \ldots, a_1.$$

Al terminar la entrevista, tu amigo afirma que, si el empresario se empeña en exigir su orden, con el poco C++ que sabes, tendrás que abandonar el proyecto y las copiosas ganancias que te habría producido.

Pero tú, rebelde por naturaleza y no menos práctico, te resistes a perder los beneficios antedichos, y decides demostrar que tu amigo estaba equivocado. Para ello, haz un procedimiento que evalúe un polinomio leyendo de la entrada los coeficientes y el valor de la variable así,

$$a_0, a_1, \ldots, a_n, x$$

y otro que los lea así:

$$a_n, a_{n-1}, \ldots, a_1, x.$$

El procedimiento no escribirá *nada*: devolverá el resultado de la evaluación en un parámetro. (Véase la pista 3.21a.)

## 3.22 Calificación de oído

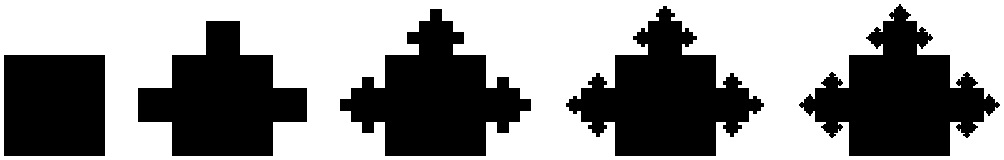
Un profesor, ya curtido por los años, se ha ido convenciendo por su experiencia de que el tamaño de las orejas está directamente relacionado con la inteligencia, y ésta con la nota que se obtiene en los exámenes. No entraré en los pormenores de su cuidadoso razonamiento, pero el caso es que ha llegado a la conclusión (inconfesable) de que le basta con medir las orejas de sus alumnos para aprobar a los que estén por encima de la media y suspender al resto. Así de rápido, y sin las engorrosas e innecesarias correcciones de exámenes.

Para llevar a cabo su ingenioso plan, ha escrito un programa que lee de la entrada estándar una lista formada por pares (nombre, tamaño) y escribe a continuación la lista de nombres con su calificación: apto o no apto. Naturalmente, la lista de datos sólo se ha de introducir una vez.

Y ahora sólo una cosa le resta por hacer: obtener los datos sin levantar sospechas...

## 3.23 Los cuadrados abominables y cáncer

El *cuadrado abominable* se define por niveles como se ve en las siguientes figuras:



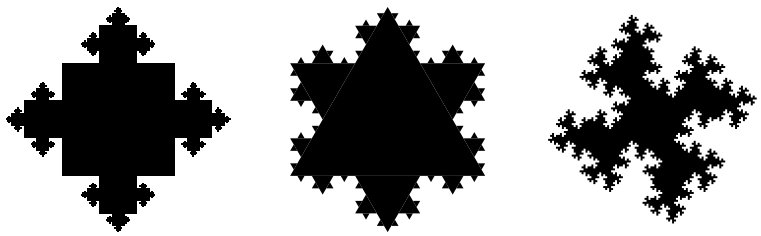
nivel 0 nivel 1 nivel 2 nivel 3 nivel 4

Es decir, en el nivel 0 tenemos un cuadrado de lado $\ell$; en el nivel 1 pegamos un cuadrado de lado $\ell/3$ en tres lados del cuadrado anterior; en el nivel 2 pegamos un cuadrado de lado $\ell/9$ en tres lados (los que quedan libres) de los cuadrados añadidos en el nivel anterior; etc. Tal vez sea más útil darse cuenta de que para hacer el nivel $n+1$ hay que pegar tres cuadrados abominables de nivel $n$.

***Pertenencia*** Escribe una función que, dado un nivel $n$ y un punto del plano $(x, y)$, nos diga si dicho punto cae o no dentro del cuadrado abominable de nivel $n$. Se supondrá que el cuadrado abominable de nivel 0 tiene su extremo inferior izquierdo en el $(0, 0)$ y su lado mide 1.

***En el límite*** Definimos el *cuadrado cáncer* como la unión de *todos* los niveles del cuadrado abominable. Escribe otra función que dado un punto $(x, y)$ diga si dicho punto cae dentro del cuadrado cáncer. Haz las mismas suposiciones que para el caso anterior.

***Otros entes abominables***



***Bibliografía*** Este insípido cuadrado cáncer es simplemente un fractal. Porque es *autosimilar*: si se mira de cerca cualquiera de las protuberancias, se observa un objeto similar al original. Tiene propiedades matemáticas harto curiosas; por ejemplo, su perímetro es infinito.

Benoît Mandelbrot (nacido el 20 de noviembre de 1924 en Varsovia, Polonia) acuñó el término fractal y se encargó de difundir la importancia y utilidad de las teorías matemáticas que circundan este concepto. Su libro más proselitista es [Man97].

Pero no es nada arriesgado afirmar que la aceptación de los fractales se debe más a su belleza visual que a su belleza elucubrativa. Las imágenes del libro [PR86] son una demostración prepotente de este hecho; a su lado, el cuadrado cáncer es un simple borrón de tinta.

## 3.24 Codificaciones de plantas con cadenas

138

Lindomayo ha descubierto que la estructura de las plantas planas está escrita con el abecedario de las letras `T`, `I`, `D`, `B`, `C`, `H` y `A`. Una palabra escrita con estas letras no describe a una especie, sino a un ejemplar. Para aprender a leer el lenguaje de Lindomayo basta con entender una correspondencia muy simple, a saber: `T` indica una pequeña prolongación, por un *Tramo*, de la rama en la que estamos; `B` describe un *Brote* que acaba la rama y `C` una rama que acaba bruscamente en un *Corte*; empieza una subrama a la *Izquierda* o la *Derecha* cuando leemos una `I` o una `D`; una `H` o una `A` representan una *HojA* que cuelga a la izquierda o a la derecha. Por ejemplo, en `TTHTTATTHTTATTHTTATTB` leemos la larga rama de una cangorza recién comprada, adornada con hojas a la izquierda y a la derecha alternativamente; pero, en cuanto pasan unos días en nuestro higiénico urbanismo, pierde ansia, se le alargan los tallos y la describimos con `TTTTHTTTTATTTTB`. Un níspero nos servirá para ilustrar el mecanismo de las subramas: `TTTTTT`$_{6}$`I`$_{13}$`TTHTATI`$_{20}$`ITHTATB`$_{27}$`I`$_{29}$`DTHTATB`$_{36}$`TB`$_{43}$`DTTHTATI`$_{50}$`ITHTATB`$_{52}$`DTHTATB`$_{54}$`TB`$_{61}$`TTI`$_{68}$`ITHTATB`$_{70}$`DTHTATB`$_{t}$`TB`$_{t}$. El tronco del níspero está pelado (o podado) en sus 6 primeros tramos. Se bifurca por primera vez tras la etiqueta 6; desde aquí hasta 29, leemos la descripción de una rama que brota a la izquierda; de 29 hasta 52 tenemos otra rama, idéntica a la anterior, salvo en que brotó hacia la derecha. Cada una de estas dos ramas consta de otras dos subramas, descritas con las letras en los intervalos $[13, 20]$, $[20, 27]$, $[36, 43]$ y $[43, 50]$. La lectura de estas palabras obliga a detener temporalmente la construcción mental de una rama cuando encontramos una bifurcación por culpa de `I` o `D`; pero en cuanto la subrama termina, la siguientes letras contribuyen a nuestra imagen de la rama original. Así, las letras en los intervalos $[52, 54]$ y $[68, 70]$ forman parte de la descripción del tronco del árbol.

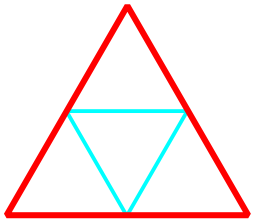
El *nivel* de una rama indica cuántas bifurcaciones hubo que tomar para llegar a ella. En el níspero, el tronco está en el nivel 0, la rama $[6, 29]$ está en el nivel 1, y la rama $[13, 20]$ está en el nivel 2. La *longitud* de una rama es el número de tramos que contiene.

En las dos horas que le quedan antes de que vuelva su marido, Lindomayo predente construir varios programas para insuflar vida a sus plantas. El programa `crece` añade $n$ tramos delante de cada brote de una planta. El programa `nivel` calculará el nivel máximo de las ramas de un árbol (posiblemente también devuelva una de las subramas en ese nivel). El programa `es` concebirá un níspero saludable con subramas de hasta nivel $n$. El programa `complu` podará con un método que genera buenos beneficios de los árboles públicos: cada rama se cortará a la mitad de su longitud. El programa `beneficio` calculará el número de tramos de un árbol que se podan con el método del programa anterior.

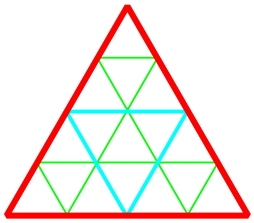
***Un poco de historia*** Aristid Lindenmayer (1925–1989) presentó en 1968 una formalización matemática del crecimiento de las plantas que luego se ha dado en llamar sistemas de Lindenmayer o simplemente sistemas-L (*Lindenmayer Systems* o *L-systems* en inglés, para que lances una búsqueda en tu indexador favorito). El tema de este ejercicio se basa en este formalismo, convenientemente desforestado. El área de los lenguajes formales engloba a los sistemas-L y a otros formalismos como las gramáticas EBNF con las que usualmente se define la sintaxis de los lenguajes de programación. Por otro lado, los sistemas de Lindenmayer también tienen conexión con los fractales.

### 3.25 1 Pasatiempo

¿Eres capaz de dibujar la siguiente figura sin levantar el lápiz del papel y sin pasar dos veces por el mismo segmento de línea?

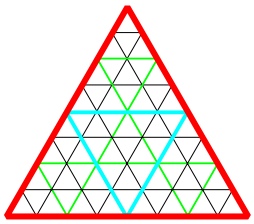


No es muy difícil. ¿Te atreves con este otro?



### 3.25 2 Desafío

Si has solucionado los pasatiempos anteriores, habrás notado que hay una manera *recurrente* de realizar el dibujo. Utilizando esta recurrencia podríamos hacer dibujos de triángulos *anidados* con tantos niveles de anidación como deseasemos. Por ejemplo el siguiente dibujo tiene 3 niveles:



Te proponemos realizar un programa que dibuje una figura como las anteriores con el nivel de anidación en los triángulos que deseemos. (Véase la pista 3.25a.)

### 3.25 3 Un poco de ayuda

Para realizar este ejercicio contaremos con cierta ayuda a la hora de dibujar. Supondremos que tenemos un *trazador* al que daremos las órdenes para que dibuje lo que necesitemos.

El trazador dispone de una pluma multicolor que permite el dibujo de líneas de cierto tamaño en una dirección a partir del punto en el que está situado. El punto de origen del trazado de las líneas, la dirección del trazo, la longitud y color de las líneas se pueden establecer con los procedimientos y tipos que se detallan a continuación.

- Procedimiento

```
void construirTrazador(double const tamX, double const tamY, string const titulo);
```

que prepara un área de trazado de dimensión `tamX` × `tamY`. El título de la ventana se indica en el parámetro `titulo`. La pluma del trazador se establece en el origen de coordenadas (esquina inferior izquierda) y el ángulo de dibujo es cero.

- Procedimiento

  ```
  void ponerCoordenadas(double const x, double const y);
  ```

  que establece el punto de dibujo del trazador en las coordenadas (`x`, `y`).

- Procedimiento

  ```
  void girar(double const alpha);
  ```

  que añade el ángulo de dibujo indicado en la variable `alpha` al ángulo actual de dibujo. El ángulo viene dado en radianes.

- Procedimiento

  ```
  void dibujarLinea(double const longitud);
  ```

  que dibuja una línea de la `longitud` indicada en el ángulo actual del dibujo, desde el punto en que está la pluma. El punto de dibujo de la pluma se establece al final de la línea dibujada.

- Tipo enumerado

  ```
  typedef enum Colores {
     blanco, amarillo, naranja, rosa, rojo, marron, verde, morado, azul, negro
  } Colores;
  ```

- Procedimiento

  ```
  void ponerColor(Colores const c);
  ```

  que establece el `color` de dibujo al indicado en la variable `c`.

- Procedimiento

  ```
  void esperar();
  ```

  que espera hasta que se cierre la ventana de dibujo.

Todos estos procedimientos están en un paquete que puedes encontrar en `http://aljibe.sip.ucm.es/recursos/trazador/`.

## 3.26 Cálculo puntual de la matriz de mediotono de Judice-Jarvis-Ninke

130 144

La *matriz de mediotono de Limb* de nivel 0, que denotaremos con $L_0$, es una matriz entera de tamaño $1 \times 1$ cuyo único elemento vale 0. La matriz de mediotono de Limb de nivel $n > 0$, $L_n$, es una matriz entera de tamaño $2^n \times 2^n$ que viene definida por

$$L_n = \begin{bmatrix} 4L_{n-1} & 4L_{n-1} + 3U_{n-1} \\ 4L_{n-1} + 2U_{n-1} & 4L_{n-1} + 1U_{n-1} \end{bmatrix}$$

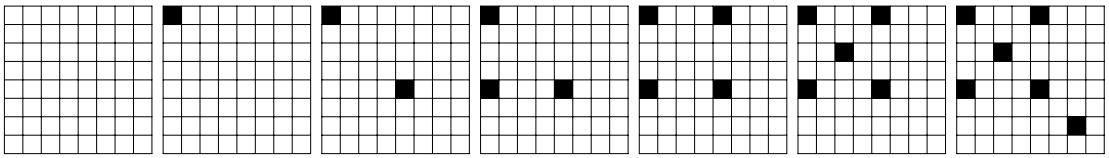
donde $U_n$ es una matriz $2^n \times 2^n$ con todos sus elementos a 1.

Haz primero una función recursiva que, dada una fila $i$, una columna $j$ y un nivel $n$, calcule el elemento de la posición $(i, j)$ de la matriz de Limb de nivel $n$. Implementa luego una variante iterativa. (Véase la pista 3.26a.)

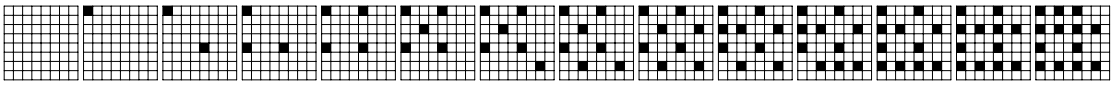
***Un poco de historia*** Estas matrices pretenden solventar el problema de plasmar una imagen con múltiples tonos de gris en un dispositivo que sólo admite blanco o negro. Para articular esta exposición utilizaremos $L_3$, que es una matriz $8 \times 8$:

$$L_3 = \begin{bmatrix} 0 & 48 & 12 & 60 & 3 & 51 & 15 & 63 \\ 32 & 16 & 44 & 28 & 35 & 19 & 47 & 31 \\ 8 & 56 & 4 & 52 & 11 & 59 & 7 & 55 \\ 40 & 24 & 36 & 20 & 43 & 27 & 39 & 23 \\ 2 & 50 & 14 & 62 & 1 & 49 & 13 & 61 \\ 34 & 18 & 46 & 30 & 33 & 17 & 45 & 29 \\ 10 & 58 & 6 & 54 & 9 & 57 & 5 & 53 \\ 42 & 26 & 38 & 22 & 41 & 25 & 37 & 21 \end{bmatrix}$$

Con un poco de paciencia se verá que en esta matriz aparecen todos los enteros entre el 0 y el 63. Y con un poco de ingenio se podrá demostrar que una matriz $L_n$ tiene todos los enteros entre el 0 y el $2^{2n} - 1$. Tanto esta propiedad, como que haya que recorrer la matriz a saltos para seguirlos, forma parte de lo que querían conseguir sus autores. Se pretende que la matriz $L_3$ coordine la generación de las siguientes 64 figuras (una imagen vale más que mil palabras):



¡No cabe! Un poco más pequeño:



¡Vaya! Lo volvemos a intentar:

El resultado semejará una secuencia de tonos de gris progresivamente más oscuros. Tal vez, a esta escala, los puntos individuales de cada figura sean todavía ostensibles; pero con una reducción adicional, el efecto es mucho mejor:

***Bibliografía*** El proceso de construcción de esta matriz se explica en [JJN74], fuente que es prácticamente imposible consultar. Pero, como casi todo lo relativo a informática gráfica, se puede encontrar una breve descripción en el fabuloso compendio [FvDFH97]. El libro [Rim93] se dedica exclusivamente al problema de plasmar, en un dispositivo monocromático, imágenes con múltiples tonos o colores; el tiempo lo ha castigado con creces por la arrogancia de querer ser muy práctico y moderno en su tiempo.

## 3.27 Dibujo de árboles mediante fractales

130 146

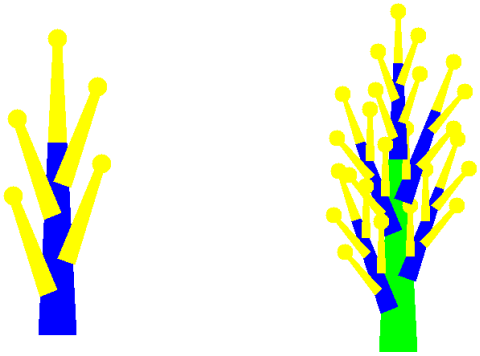
Muchas estructuras de la naturaleza se pueden dibujar en una computadora mediante el uso de fractales, no sólo *cánceres* como en el ejercicio 3.23. En este ejercicio usaremos un fractal sencillo que sirve para dibujar árboles similares a los siguientes:

***Cómo dibujar un árbol*** El dibujo básico del fractal será un tronco con una hoja:

Y se procederá de forma recursiva con un algoritmo similar al siguiente

⟨*Dibujar el tronco*⟩
`if (`⟨*nivel básico*⟩`) {`
  ⟨*Dibujar la hoja*⟩
`} else {`
  ⟨*Dibujar cinco subárboles de tamaño la mitad*⟩
`}`

para obtener árboles cómo los siguientes, dependiendo del nivel de profundidad que elijamos:



Obviamente, para obtener resultados que se parezcan a árboles, habrá que escoger los colores de forma apropiada. Si quieres obtener mayor realismo en el dibujo de árboles tendrás que pensar un poco y hacer algunas pruebas. (Véase la pista 3.27a.)

***Un poco de ayuda*** Para simplificar las tareas de dibujo, supondremos que disponemos del trazador que definimos en el ejercicio 3.25 teniendo en cuenta que se han añadido las siguientes funcionalidades:

- Procedimiento

  `void ponerColor(double const rojo, double const verde, double const azul);`

  que establece el color de dibujo en el espacio RGB (*red*, *green*, *blue*; rojo, verde, azul en inglés), siendo $0 \leq$ `rojo`, `verde`, `azul` $\leq 1$. (Véase el ejercicio 1.1.)

- Procedimiento

  `void dibujarDisco(double const centroX, double const centroY, double const radio);`

  que dibuja un disco en la posición (`centroX`, `centroY`) con el radio indicado, obviamente `radio` $> 0$.

- Procedimiento

  `void dibujarPoligono(double const x[], double const y[], int const numPuntos);`

  que dibuja un polígono cuyos vértices son (`x[i]`,`y[i]`) con $0 \leq$ `i` $<$ `numPuntos`.

## 3.28 Dragones y teselas

148

¿Te gusta el dibujo que aparece en la figura 3.5? Es un fractal, llamado la *curva del dragón* debido a

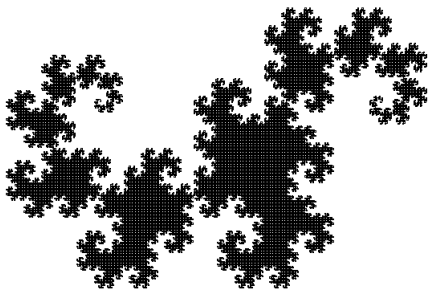


Figura 3.5: Curva del dragón

que, echándole bastante imaginación, se asemeja a un dragón.

En este ejercicio tendrás que desarrollar un programa que dibuje *dragones*. No te preocupes, es más fácil de lo que crees.

### 3.28.1 Técnica de construcción

Existen diversas formas de construir este fractal, pero la que nos va a servir de más ayuda fue la que describió el físico Bruce A. Banks:

> *Empezamos con un gran ángulo recto, y en cada paso sustituimos cada segmento por un nuevo ángulo recto de dimensiones menores tal y como se ilustra en la figura 3.6.*

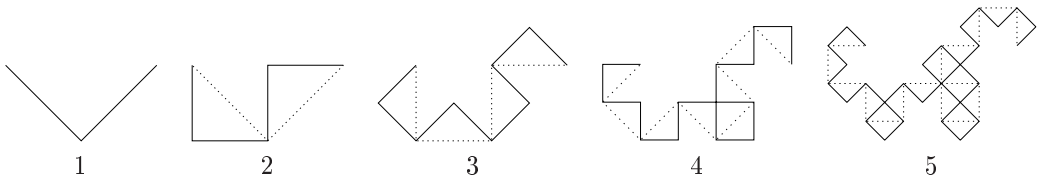



Figura 3.6: Pasos en la generación de una curva dragón

El número de veces que realicemos esa transformación de un lado por un par de lados nos indicará el grado de la curva; en la figura 3.6 tenemos las curvas desde el grado uno al grado cinco.

### 3.28.2 Un poco de ayuda

Para ayudarnos a dibujar, contamos con el trazador que se define en el ejercicio 3.25.

### 3.28.3 Un poco de historia

Al fractal que presenta el ejercicio se le suele llamar el dragón de Harter-Heighway, en honor a John Heighway y William Harter, que junto con su colega Bruce A. Banks, fueron los primeros en estudiarlo. En [Gar86] puedes encontrar más detalles.

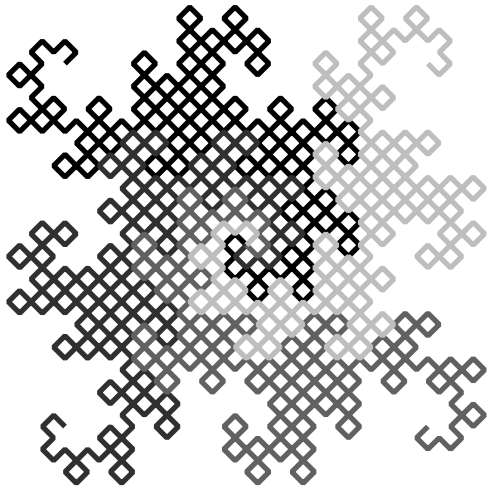


Figura 3.7: Cuatro dragones acoplados entre sí

El dragón tiene varias propiedades interesantes: se puede dibujar de un solo trazo sin que en ningún momento se cruce la línea ya dibujada; dos fractales coinciden exactamente en el borde de forma que se pueden acoplar unos con otros y se puede llegar a recubrir todo el plano con fractales de este tipo. Esta última característica le sirvió al afamado informático Donald E. Knuth (1938– ) para alicatar un muro de su casa.

**3.4a.** Comprender la representación de un número en el sistema de numeración posicional es esencial para solucionar el problema. El haber trabajado el ejercicio 2.21 puede ayudarte a ello.

**3.5a.** Haber trabajado con el ejercicio 3.4 puede ser de gran ayuda para saber cómo *obtener* los dígitos que componen un número.

**3.7a.** Puede ser muy útil definir una función aleatoria que reciba dos enteros como parámetros de entrada, y devuelva un número entero aleatorio entre dichos parámetros. Con el ejercicio 3.13 puedes aprender a definir variables aleatorias.

**3.7b.** Aquí recogemos algunas expresiones que permiten determinar si un punto $(x, y)$, en dos dimensiones, o $(x, y, z)$, en tres, es interior a las figuras geométricas que se describen.

- Círculo con centro en el punto $(a, b)$ y radio $r$:
$$(x-a)^2 + (y-b)^2 \leq r^2$$
- Elipse con centro en el punto $(a, b)$ y radio de la abcisa $r_1$ y radio de la ordenada $r_2$:
$$\frac{(x-a)^2}{{r_1}^2} + \frac{(y-b)^2}{{r_2}^2} \leq 1$$
- Esfera con centro en el punto $(a, b, c)$ y radio $r$:
$$(x-a)^2 + (y-b)^2 + (z-c)^2 \leq r^2$$
- Cilindro con base en el círculo de centro $(a, b, c)$, radio $r$ y altura $h$:
$$(x-a)^2 + (y-b)^2 = r^2$$
$$c \leq z \leq c+h$$

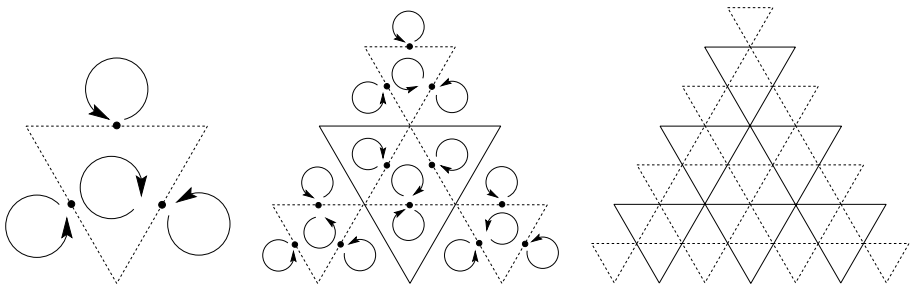
**3.15a.** En el ejercicio 3.5 tienes funciones que te serán útiles.

**3.16a.** El ejercicio 3.13 te será útil para definir las funciones aleatorias que necesitas utilizar en la simulación de los juegos que propone el ejercicio.

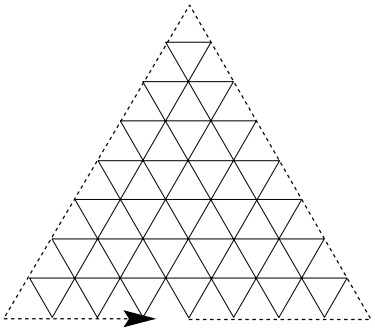
**3.21a.** C++ tiene recursión.

**3.25a.** El nivel más básico consiste en dibujar un triángulo, que contiene otros tres triángulos, de lado la mitad del exterior; este proceso podemos repetirlo tantas veces como deseemos dentro de cada uno de los triángulos resultantes del nivel anterior.

Lo mejor es escribir un subprograma recursivo que dibuje estos triángulos anidados. En realidad el programa recursivo deberá dibujar solamente el interior de los triángulos:



Después, se dibujará el borde del triángulo principal,



con lo que habremos terminado el dibujo que nos proponíamos.

**3.26a.** Si no tienes suficiente práctica con el álgebra matricial, la fórmula que define $L_n$ te puede resultar extraña. Cuando se construye una matriz juntando otras, se entiende que las internas pierden el caparazón que sujeta sus elementos y éstos pasan a formar parte de la matriz externa. Por eso,

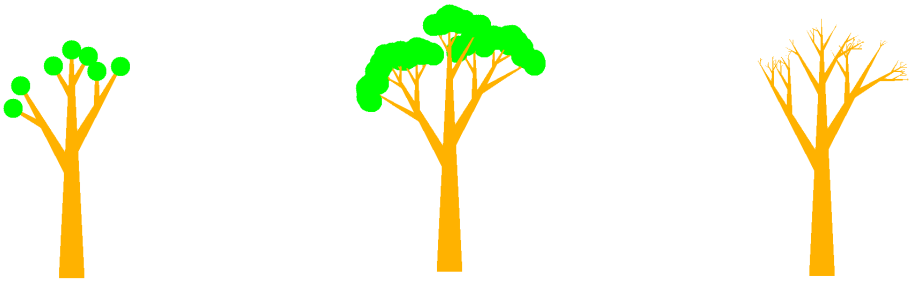
$$L_1 = \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}$$

y

$$L_2 = \begin{bmatrix} 0 & 12 & 3 & 15 \\ 8 & 4 & 11 & 7 \\ 2 & 14 & 1 & 13 \\ 10 & 6 & 9 & 5 \end{bmatrix}.$$

**3.27a.** Para obtener un mayor realismo en el dibujo conviene tener en cuenta lo siguiente:

- Para obtener un árbol con suficiente detalle de ramificación hay que iterar el proceso unos ocho niveles.
- Si se llega hasta niveles altos, las hojas pueden ser demasiado pequeñas para ser apreciadas, tendríamos en este caso un árbol típicamente invernal, sin hojas. Si queremos ver un árbol con las hojas verdes propias de la primavera, la solución consiste en establecer un radio mínimo para los discos que acaban siendo las hojas. Dependiendo de ese radio mínimo el árbol aparecerá más o menos verde.
- Siguiendo el proceso tal y como se ha dicho obtenemos un árbol demasiado *geométrico*; para obtener más realismo se puede introducir algo de aleatoriedad: los subárboles más pequeños se pintarán dependiendo de un cierto número escogido de forma aleatoria.
- Por último se pueden intentar diversas configuraciones para obtener distintos tipos de árboles.



## 31 Incierta igualdad de números reales

103

Una primera idea consiste en cambiar la comparación $x = y$ por la siguiente,

$$|x - y| < \varepsilon$$

siendo $\varepsilon$ el margen de tolerancia entre dos reales *supuestamente iguales:*

```
bool proximos(double const x, double const y) {
  float const eps = 1E-6;
  return fabs(x-y) < eps;
}
```

Pero esta solución presenta el inconveniente de conceder el mismo margen de tolerancia a cualquier par de reales, independientemente de su orden de magnitud. Por ejemplo, un margen $\varepsilon = 10^{-3}$ resulta excesivo para comparar cifras de órdenes pequeños (tales como medidas de átomos), y a la vez ese mismo margen es en cambio estrechísimo para números de órdenes grandes (como son las distancias astronómicas).

Una solución consiste en sustituir la expresión $|x - y| < \varepsilon$ por la siguiente,

$$\frac{|x - y|}{x^2 + y^2} < \varepsilon$$

que relativiza la distancia según su orden de magnitud.

Ahora surge la posible división por cero, que se presenta en el caso de ser $x$ e $y$ nulos o de valor muy pequeño. Por ello, en vez de la expresión anterior, optamos por la siguiente:

$$|x - y| < \varepsilon(x^2 + y^2).$$

En resumen, basta con usar la función anterior, cambiando su última línea por ésta:

```
return fabs(x-y) < eps * (x*x + y*y);
```

## 33 Ponle tú el título

103

El resultado es la siguiente matriz:

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 5 | 0 | 1 | 2 | 3 | 4 |
| 4 | 5 | 0 | 1 | 2 | 3 |
| 3 | 4 | 5 | 0 | 1 | 2 |
| 2 | 3 | 4 | 5 | 0 | 1 |
| 1 | 2 | 3 | 4 | 5 | 0 |

## 36 Cuentaletras

105

Podría parecer que una solución a este problema requiere un par de bucles anidados: el interno para contar las letras de cada palabra; el externo para recorrer cada palabra de un texto. Pero esta primera intuición lleva a una solución muy compleja. Para que se entienda por qué, nos dejaremos llevar por un momento al fracaso para, seguidamente, mostrar una solución más limpia.

### 3.6.1 Bucles anidados

Supongamos que tenemos una función predicado llamada `esLetra` que determina si un carácter es una letra. El siguiente bucle cuenta las letras consecutivas que encuentra en el fichero `in` y, por ende, si se ejecuta al principio de una palabra, cuenta las letras que tiene.

```
int letras = 0;
char siguiente;
in.get(siguiente);
while (esLetra(siguiente)) {
  letras++;
  in.get(siguiente);
}
```

Pero este bucle ignora un detalle fundamental siempre que se trabaje con ficheros: antes de leer, se ha de comprobar que no hemos llegado al fin.

Centremos nuestra atención en el bucle. Parecerá que el problema se soluciona así:

```
while (esLetra(siguiente) && !in.eof()) {
  letras++;
  in.get(siguiente);
}
```

Ahora el problema está en el otro extremo porque comprobamos demasiado pronto si hay algo que leer. De esta forma, si el fichero acaba con una letra, no la tendremos en cuenta y, por tanto, atribuiremos un carácter menos a la última palabra.

El momento oportuno para comprobar si podemos leer un nuevo carácter es justamente antes de leerlo:

```
while (esLetra(siguiente)) {
  letras++;
  if (!in.eof()) in.get(siguiente);
  else ⟨¿Qué hacer en este caso?⟩;
}
```

Pero ¿qué podemos poner en `siguiente` si no queda nada por leer del fichero? Se suele recurrir a un valor que termine el bucle y no tenga ningún efecto sobre la cuenta; por ejemplo, un espacio.

Ahora que hemos dado con el bucle interno que cuenta las letras de una palabra, sólo queda rodearlo de uno que lo repite hasta que se acabe el fichero. Por supuesto, entre palabra y palabra, hay espacios que es necesario quitar con otro pequeño bucle.

```
while (!in.eof()) {
  char siguiente;
  in.get(siguiente);
  while (!esLetra(siguiente) && !in.eof()) in.get(siguiente);
  int letras = 0;
  while (esLetra(siguiente)) {
    letras++;
    if (!in.eof()) in.get(siguiente);
    else siguiente = ' ';
  }
  if (letras > 0) out << letras;
}
```

### 3.6.2 Un solo bucle

Hay algunas otras formas de organizar los bucles doblemente anidados anteriores; pero no conocemos ninguna que sea suficientemente elegante. Busquemos una vía para escapar de este fracaso.

Lo que intuitivamente puede parecer un par de bucles anidados, muchas veces se puede aplanar a uno solo si recurrimos al concepto de *estado*. En nuestro problema, cuando estamos recorriendo un fichero, tenemos dos posibles estados: o bien estamos dentro de una palabra, o bien estamos fuera. Estaremos *dentro de una palabra* cuando se haya leído alguna de sus letras, pero todavía no se haya leído un carácter que indique que se ha acabado. Estaremos fuera en cualquier otro caso.

Hay que analizar cuatro casos, resultado de combinar dos posibilidades sobre el carácter (si es o no letra) con dos estados (estamos dentro o fuera de una palabra). El siguiente código es un boceto de lo que hay que hacer en cada caso. Al igual que en la solución anterior, usamos la variable entera `letras` para contar las letras de una palabra.

```
if (esLetra(siguiente)) {
  if (⟨estamos dentro de una palabra⟩) {
    letras++;
  } else {
    letras = 1;
    ⟨Se anota que estamos dentro de una palabra⟩
  }
} else if (⟨estamos dentro de una palabra⟩) {
  out << letras;
  ⟨Se anota que estamos fuera de una palabra⟩
}
```

El estado se suele representar con diferentes técnicas. Algunas son explícitas; por ejemplo, cuando se utiliza una variable que toma un valor distinto por cada posible estado. Aquí utilizaremos un técnica implícita; aunque en un principio, el propósito de la variable `letras` era contar el número de letras de cada palabra, también servirá para indicar el estado, porque ocurre que estamos en una palabra precisamente si `letras > 0`.

```
int letras = 0;
while (!in.eof()) {
  char siguiente;
  in.get(siguiente);
  if (esLetra(siguiente)) {
    letras++;
  } else if (letras > 0) {
    out << letras;
    letras = 0;
  }
}
if (letras > 0) out << letras;
```

### 3.6.3 Ser letra

Para implementar el predicado `esLetra(siguiente)` podemos aprovechar la numeración correlativa de los caracteres alfabéticos en el código ASCII,

```
bool esLetra(char const caracter) {
  return ('a' <= caracter) && (caracter <= 'z')
         || ('A' <= caracter) && (caracter <= 'Z');
}
```

o, directamente, podemos utilizar la función `isalpha()`, que se encuentra en `ctype.h`:

```
bool esLetra(char const caracter) {
  return isalpha(caracter);
}
```

Aunque esta solución ignora la letra `ñ` y las tildes, no será difícil modificarla para que se tengan en cuenta.

## 3.11 Sucesión bicicleta

109

***Periodo*** La idea es sencilla: considerando los términos consecutivos $c_1$ y $c_2$ (con valores iniciales $b_1$ y $b_2$), basta con hacerlos avanzar hasta que se tenga, nuevamente, $c_1 = b_1$ y $c_2 = b_2$, y el período $p$ será el número de avances efectuados:

```
int periodo(double const primero, double const segundo) {
  double const b1 = primero, b2 = segundo;
  double c1 = b1, c2 = b2;
  int numPasos = 0;
  do {
    double c = (c2+1)/c1; c1 = c2; c2 = c;
    numPasos++;
  } while (b1 != c1 || b2 != c2);
  return numPasos;
}
```

Teniendo en cuenta que el bucle con que generamos los términos $b_i$ de la sucesión está controlado por la expresión siguiente,

$$(c_1 = b_1) \wedge (c_2 = b_2)$$

que valora la igualdad de números reales, es preciso evitar el peligro de caer en un bucle infinito debido a las deficiencias de precisión en la representación de los números reales. Por ello, usamos la función `proximos`, que se explica en el ejercicio 3.1. Y así sustituimos la condición `b1 != c1 || b2 != c2`, de salida del bucle por esta otra:

```
!proximos(b1, c1) || !proximos (b2, c2)
```

***Serie*** Ahora, el modo trivial sería avanzar a través de la sucesión, pasando por todos los términos que se desea sumar, y añadiéndolos uno a uno. Pero al ser cíclica la serie, es más eficiente proceder como sigue:

1. Se calcula primero la suma `sumCiclo` de los términos de un ciclo:

   $$\texttt{sumaCiclo} = \sum_{i=1}^{p} b_i$$

   Este cálculo se puede llevar a cabo a la vez que se halla $p$, como se ha hecho en el apartado anterior.

2. Se averigua cuántos ciclos completos (`numCiclos`) hay en $n$ términos y cuántos términos sueltos (`numSueltos`) restan hasta el $n$-ésimo. Se multiplican la suma anterior y `numCiclos`.

3. Se suman los términos sueltos ignorados en los `numCiclos` completos.

Para solucionar el primer punto adaptamos la función del apartado anterior para que calcule, además del período, la suma de los términos de un ciclo. Puesto que se devuelven dos valores, la función se convierte en el procedimiento `recorrerCiclo`. Lo único que hay que cambiar con respecto a la función anterior es la cabecera,

```
void recorrerCiclo(double const primero, double const segundo,
                   double& sumaCiclo, int& periodo)
```

y además, en cada vuelta del bucle será necesario acumular el valor actual calculado:

```
sumaCiclo = sumaCiclo + b;
```

El valor inicial de `sumaCiclo` es 0.

Así la suma de los $N$ primeros términos de la sucesión se calcula con la siguiente función:

```
double sumaN(double const primero, double const segundo, int const hasta) {
  double sumaCiclo;
  int peri;
  recorrerCiclo(primero, segundo, sumaCiclo, peri);
  int numCiclos = hasta / peri;
  int numSueltos = hasta % peri;
  sumaCiclo = sumaCiclo * numCiclos;
  double c1 = primero, c2 = segundo;
  for (int i = 0; i < numSueltos; i++) {
    sumaCiclo = sumaCiclo + c1;
    double b = (c2+1) / c1; c1 = c2; c2 = b;
  }
  return sumaCiclo;
}
```

## 3 19 Número de ceros en que termina un factorial

118

***Grado de multiplicidad*** El grado de multiplicidad $g$ de $d$ en $n$ es la máxima potencia del factor $d$ que es divisor de $n$; o sea: $n = k \times d^g$, donde $k$ no es divisible por $d$. El grado de multiplicidad se puede calcular fácilmente, efectuando las divisiones enteras correspondientes y llevando a la vez la cuenta del número de ellas efectuadas:

$$24 \xrightarrow{/2} 12 \xrightarrow{/2} 6 \xrightarrow{/2} 3$$

Es decir:

```
int grado(int const d, int const n) {
  int auxN = n;
  int grado = 0;
  while (auxN % d == 0) {
    auxN = auxN / d;
    grado = grado++;
  }
  return grado;
}
```

Se ha de exigir que $n \geq 1$ y que $d \geq 2$ para que esté definido el grado de multiplicidad.

***Ídem, recursiva*** El caso base es, trivialmente, el de multiplicidad cero, cuando el primer entero no es múltiplo del segundo,

$$\texttt{grado(7, 24)} \rightsquigarrow 0$$

y el caso recurrente se halla a partir del grado de multiplicidad del primero, dividido exactamente por el segundo:

$$\texttt{grado(2, 24)} \rightsquigarrow 1 + \texttt{grado(2, 12)}$$

Es decir:

```
int gradoRec(int const d, int const n) {
  if ((n % d) != 0) {
    return 0;
  } else {  // n es múltiplo de d
    return 1 + gradoRec(d, n/d);
  }
}
```

***Multiplicidad de 5*** Este apartado no es más que una aplicación directa de la función `grado`, definida en el caso anterior:

```
int grado5(int const n) {
  return grado(5, n);
}
```

***Los ceros de un factorial, al fin*** Cada cero en un producto requiere la existencia de un 2 y un 5 entre los factores de dicho producto. Como en $1 \times 2 \times 3 \times 4 \ldots$ siempre hay más doses que cincos, el número de cincos que haya entre los factores determina el número de ceros que habrá en el producto. Así pues, en $n!$ habrá tantos ceros como cincos haya en las descomposiciones de los factores $1, 2, 3, \ldots, n$.

```
int ceros(int const n) {
  int tot = 0;
  for (int i = 1; i <= n; i++) {
    tot = tot + grado5(i);
  }
  return tot;
}
```

## 3.21 ¿Cuál es el mejor orden para recibir los datos de un polinomio?

119 129

La principal característica de este problema es que el número de coeficientes del polinomio es un dato implícito; en ningún momento el usuario del programa dará el valor $n$; es responsabilidad de nuestro código calcularlo a partir del número de datos, o mejor, ser capaz de operar sin llegar a conocerlo jamás.

Vamos a darle un nombre de letra griega a cada orden propuesto: el del empresario será $\delta$, el nuestro $\alpha$, el intermedio será $\beta$, mientras que $\gamma$ será un cuarto orden $a_0, \ldots, a_n, b$.

Tanto el orden $\alpha$ como el $\beta$ son muy sencillos. En ambos casos vamos a guardar en la variable `b` el punto de evaluación; por la variable `a` irán pasando los sucesivos coeficientes, y por `r` irán pasando los resultados parciales de la evaluación. Para el orden $\alpha$ necesitamos las sucesivas potencias de `b`. Se pueden conseguir de dos formas: calculándolas completamente en cada momento, o de forma incremental en una variable auxiliar que aquí llamaremos `p`:

```
void eval(int& r) {
  r = 0;
  int p = 1;
  int b;
  cin >> b;
  while (!eoln(cin)) {
    int a;  cin >> a;
    r = r + a*p;
    p = p * b;
  }
}
```

El orden $\beta$ se resuelve fácilmente con la pista 2.8a. Aunque el código es más sutil, también es más breve. Las ideas fundamentales son las mismas que para convertir una secuencia de dígitos en el número que denotan:

```
void eval(int& r) {
  r = 0;
  int b;  cin >> b;
  while (!eoln(cin)) {
    int a;  cin >> a;
    r = r*b + a;
  }
}
```

Los dos órdenes que quedan por resolver son más complicados, pero se resuelven fácilmente haciendo uso adecuado de la recursión. Empezemos con $\gamma$. Cuando nuestro procedimiento se enfrente a la secuencia $a_0, \ldots, a_n, b$, aparentemente no puede hacer nada útil: para poder evaluar el polinomio tenemos que multiplicar a cada $a_i$ por una potencia adecuada de $b$; pero no conoceremos $b$ si no hemos leído (y perdido) antes todos los $a_i$. Hace falta un cambio de enfoque. Pensemos recursivamente. Si leemos $a_0$ resultará que la secuencia se ha convertido en $a_1, \ldots, a_n, b$; es un elemento más corta y, por tanto, tiene sentido intentar resolver este subproblema de forma recursiva. Supongamos que lo hemos hecho así y llamemos $r_1$ al resultado. Para construir la solución del problema original basta con calcular $a_0 + r_1 b$. De esto se deduce que nuestro procedimiento recursivo, además de devolver el valor de la evaluación del subproblema, también tendrá que devolver el valor de $b$. El caso base de esta recursión es trivial, y se explica mejor con el propio código:

```
void eval(int& r, int& b) {
  int a;  cin >> a;
  if (eoln(cin)) {
    r = 0;
    b = a;
  } else {
    eval(r, b);
    r = a + r*b;
  }
}
```

El orden $\delta$ es ligeramente más complejo. La secuencia original es $a_n, \ldots, a_0, b$; el subproblema que resuelve la llamada recursiva es $a_{n-1}, \ldots, a_0, b$. Supongamos, como antes, que devolvemos tanto la evaluación del subproblema (en $r$) como el punto de evaluación $b$. Para reconstruir el valor del problema

original necesitamos calcular $a_n b^n + r$. Es decir, es necesario devolver un dato más, que puede ser bien $n$ o bien $b^n$. En el procedimiento que sigue, hemos optado por devolver $b^n$ en el parámetro `p`:

```
void eval(int& r, int& p, int& b) {
  int a;  cin >> a;
  if (eoln(cin)) {
    r = 0;
    p = 1;
    b = a;
  } else {
    eval(r, p, b);
    r = a*p + r;
    p = p * b;
  }
}
```

## 3.24 Codificaciones de plantas con cadenas

121

Antes de empezar con la solución de cada uno de los programas conviene definir una constante por cada letra del abecedario de las plantas:

```
char const brote = 'B';
char const tramo = 'T';
char const hojaIzq = 'H';
char const hojaDer = 'A';
char const subramaIzq = 'I';
char const subramaDer = 'D';
char const corte = 'C';
```

Empezamos por realizar una función `crece`, que recibe como argumento una planta (dada simplemente con un `string`) y el número de tramos que queremos añadir a cada brote, y devuelve como resultado la planta resultante.

Esta función es bastante sencilla: basta con recorrer la planta copiándola en una nueva; cuando encontramos un brote, añadimos antes del mismo tantos tramos como indique el parámetro correspondiente:

```
string crece(string const original, int const n) {
  string nuevo = "";
  for (int i = 0; i < original.size(); i++) {
    if (original[i] == brote) {
      for (int j = 0; j < n; j++) nuevo = nuevo+tramo;
    }
    nuevo = nuevo+original[i];
  }
  return nuevo;
}
```

El resto de los programas se pueden realizar de forma sencilla utilizando la recursión. Para el programa `es`, realizaremos una función que crea el níspero con el nivel deseado. Vamos por partes:

- El caso base de esta recursión se tiene cuando `nivel == 0`, en cuyo caso realizaremos un níspero con un solo tronco y varias hojas.

- En el caso recursivo, se realizan varios tramos que dependerán del nivel en el que estemos dibujando la rama y varias subramas. Su distribución concreta se puede cambiar a gusto de Lindomayo.

Es decir:

```
string es(int const nivel) {
  string resultado="";
  if (nivel == 0) {
    resultado = resultado + tramo + hojaIzq +
                tramo + hojaDer + tramo + brote;
  } else {
    ⟨Añadir un trozo de tronco⟩
    resultado = resultado + subramaIzq + es(nivel-1);
    ⟨Añadir un trozo de tronco⟩
    resultado = resultado + subramaDer + es(nivel-1);
    resultado = resultado + es(nivel-1);
  }
  return resultado;
}
```

Para ⟨*Añadir un trozo de tronco*⟩ realizamos un procedimiento `anyadeTramo` que añade tramos según el nivel en que nos encontremos:

```
void anyadeTramo(string& arbol, int const nivel) {
  for (int i = 0; i < nivel; i++) arbol = arbol + tramo;
}
```

Los programas recursivos que faltan se complican algo más, porque en ellos no es posible distinguir *a priori* el caso base de los casos recurrentes.

Empecemos con el programa `nivel`. En líneas generales, este programa recorrerá el árbol, y cuando se encuentre una bifurcación realizará una llamada recursiva para calcular el nivel de esa rama: si la rama tiene un nivel mayor que el encontrado hasta el momento, debemos acordarnos de ella, y como parte del nivel siguiente al actual, será necesario sumar uno al nivel obtenido. Para recorrer el árbol necesitamos una variable auxiliar `posicion` para saber en qué posición del `string` nos encontramos.

Puesto que el análisis de la rama cambia esa posición, el parámetro deberá ser de entrada y salida: como parámetro de entrada, indica la posición inicial donde nos encontramos; y como parámetro salida comunica la última posición del subárbol analizado. Debido al uso de un parámetro de entrada y salida, parece conveniente realizar un procedimiento en lugar de una función; además, también deseamos saber cuál es la rama de mayor nivel, lo que requiere un segundo parámetro de salida y, por tanto, se refuerza la necesidad de que el subprograma sea un procedimiento. El perfil de este procedimiento es el siguiente

```
void calculaNivel(string const arbol, int& posicion, int& nivel);
```

y la llamada principal será así:

```
string arbol = ⟨contenido del arbol⟩;
int nivel = 0;
int posicion = 0;
calculaNivel(arbol, posicion, nivel);
```

Y vamos con el contenido de dicho procedimiento. En primer lugar necesitamos una variable auxiliar para almacenar el nivel máximo de las ramas; como el nivel mínimo que tendremos es 0, éste será su valor inicial. Seguidamente, realizaremos un bucle para recorrer toda la subrama; supondremos que toda

subrama acaba bien en un corte bien en un brote, e iremos analizando lo que nos encontramos:

```
int nivelMax = 0;
while (arbol[posicion] != brote && arbol[posicion] != corte) {
  ⟨Análisis de la posicion actual⟩
  posicion++;
}
nivel = nivelMax;
```

En cada posición deberemos analizar si tenemos una bifurcación o no; en caso afirmativo realizamos una llamada recursiva a la subrama. Por hipótesis de inducción nos devolverá la última posición de la subrama en la variable `posicion` y el nivel de la subrama en una variable auxiliar que definiremos a tal efecto; puesto que hemos calculado el nivel de una subrama deberemos aumentar en uno el nivel calculado:

```
if (arbol[posicion] == subramaIzq || arbol[posicion] == subramaDer) {
  posicion++;
  int nivelAux = 0;
  calculaNivel(arbol, posicion, nivelAux);
  nivelAux++;
  if (nivelAux>nivelMax) nivelMax = nivelAux;
}
```

Si pretendemos calcular la rama de mayor nivel, el procedimiento deberá tener un parámetro de salida adicional que sea la rama de mayor nivel. Es necesario llevar la cuenta de cuál es la rama de mayor nivel calculado hasta ese momento y de la rama leída hasta el momento. Para ello llevamos dos variables cuyo valor inicial será la cadena vacía. El procedimiento queda como sigue:

```
void calculaNivel(string const arbol, int& posicion, string& rama, int& nivel) {
  string tronco = "", ramaMax = "";
  int nivelMax = 0;
  while (arbol[posicion] != brote && arbol[posicion] != corte) {
    ⟨Análisis de la posicion actual⟩
    posicion++;
  }
  nivel = nivelMax;
  ⟨Terminar la rama de nivel máximo⟩
}
```

Para terminar la rama de nivel máximo debemos indicar cuál ha sido la rama; para ésta tenemos que distinguir dos casos: que la rama no tenga subramas (`nivel == 0`) o que sí las tenga (`nivel > 0`). En el primero deberemos devolver el tronco calculado hasta el momento concatenado con la finalización (brote o corte), y en el segundo la rama de máximo nivel calculado:

```
if (nivel == 0) rama = tronco + arbol[posicion];
else rama = ramaMax;
```

En cuanto al análisis es necesario complicar un poco el caso de la bifurcación para poder calcular de forma adecuada la rama, y en el caso de que el carácter actual sea un tramo o una hoja añadirlo a la variable `tronco`:

```
if (arbol[posicion] == subramaIzq || arbol[posicion] == subramaDer) {
  char desviacion = arbol[posicion];
  posicion++;
```

```
    string ramaAux = "";
    int nivelAux = 0;
    calculaNivel(arbol, posicion, ramaAux, nivelAux);
    nivelAux++;
    if (nivelAux > nivelMax) {
      nivelMax = nivelAux;
      ramaMax = tronco + desviacion + ramaAux;
    }
  } else if (arbol[posicion] == tramo || arbol[posicion] == hojaDer ||
             arbol[posicion] == hojaIzq) {
    tronco = tronco + arbol[posicion];
  }
```

Pasemos ahora a solucionar el programa `complu`; la idea será similar a la anterior: realizaremos un programa recursivo que irá recorriendo el árbol; llevaremos un parámetro de entrada y salida que nos indicará la posición por la que vamos recorriendo el árbol y un parámetro de salida que será el resultado de hacer la poda al árbol original. He aquí el perfil de este subprograma,

```
void complu(string rama, int& posicion, string& ramaPodada);
```

así como la llamada inicial al mismo:

```
string arbol = ⟨contenido del árbol⟩;
int posicion = 0;
string arbolPodado = "";
complu(arbol, posicion, arbolPodado);
```

El algoritmo, recursivo, será parecido al del cálculo del nivel pero, como veremos, es necesario añadir ciertos detalles. Iremos construyendo la rama podada según vayamos recorriendo el árbol, iremos copiando los tramos y las hojas y podando las subramas según vayan apareciendo hasta que lleguemos al final o a la mitad de la rama. Por último habrá que finalizar la rama de forma correcta, e indicar la última posición que ocupa la rama.

```
ramaPodada = "";
while (rama[posicion] != brote && rama[posicion] != corte
       && ⟨longitud recorrida⟩ < ⟨longitud de total de la rama⟩/2) {
  ramaPodada = ramaPodada + rama[posicion];
  ⟨Análisis del la posicion actual⟩
  posicion++;
}
⟨Terminar la rama⟩
⟨Colocar posicion al final de la rama⟩
```

Para acabar la rama podada es necesario tener en cuenta si la rama actual tiene longitud cero y acaba ya en un brote, y entonces seguiremos respetando que la rama podada acabe en un brote. Podemos comprobar que la rama tiene longitud cero simplemente examinando si `rama[posicion]` es igual o no a `brote`, porque si la longitud es mayor que cero el elemento en `rama[posicion]` será un tramo:

```
if (rama[posicion] != brote) {
  ramaPodada = ramaPodada + corte;
} else {
  ramaPodada = ramaPodada + brote;
}
```

No debemos, por último, olvidar que hemos de colocar la variable `posicion` al final de la rama:

`posicion` = ⟨*última* `posicion` *de la rama*⟩;

En lo anterior, podemos ver que necesitamos una serie de variables auxiliares: para saber la longitud total de la rama introducimos la variable `longitudTotal`; para saber la posición final de la rama usaremos la variable `posicionFinal`; y para saber la longitud recorrida de la rama, la variable `longitudActual`.

El valor inicial de la variable `longitudActual` será cero, y habrá que incrementarla cada vez que nos encontremos un tramo. Para calcular la longitud total de la rama y su posición final realizaremos un programa recursivo, con el siguiente perfil:

```
void longitud(string const& rama, int& posicion, int& longTotal);
```

El parámetro `posicion` será de entrada y salida: de entrada porque indica la posición de inicio de la rama, y de salida porque nos devolverá la última posición de la rama; el parámetro `longTotal` será de salida. Así antes del bucle deberemos incluir las siguientes líneas:

```
int posicionFinal = posicion;
int longitudTotal = 0;
int longitudActual=0;
longitud(rama, posicionFinal, longitudTotal);
```

El procedimiento `longitud` es similar tanto al del cálculo del nivel como al de la poda, por lo que se deja al lector su desarrollo. Con todo esto, el ⟨*Análisis del la* `posicion` *actual*⟩ deberá realizar las llamadas recursivas si la `posicion` actual es una bifuración o aumentar `longitudActual` si se trata de un tramo:

```
if (rama[posicion] == subramaIzq || rama[posicion] == subramaDer) {
  posicion++;
  string subRamaPodada = "";
  complu(rama, posicion, subRamaPodada);
  ramaPodada = ramaPodada + subRamaPodada;
} else if (rama[posicion] == tramo) {
  longitudActual++;
}
```

El programa `beneficio` tiene una versión trivial que consiste en calcular el valor de un árbol, podar el árbol, calcular el valor del árbol podado y restar las dos cantidades obtenidas.

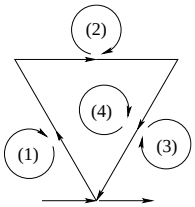
```
int valorEjemplar(string const ejemplar) {
  int resultado = 0;
  for (unsigned int i = 0; i < ejemplar.size(); i++) {
    if (ejemplar[i] == tramo) resultado++;
  }
  return resultado;
}
int beneficio(string const arbol) {
  string arbolPodado;
  int posicion = 0;
  complu(arbol, posicion, arbolPodado);
  return valorEjemplar(arbol) - valorEjemplar(arbolPodado);
}
```

También se puede realizar un algoritmo recursivo similar a los anteriores que calcule el beneficio de podar un árbol sin necesidad de calcular el árbol podado.

Para entender bien el programa es necesario tener en cuenta que en cada momento la plumilla está en una posición y preparada para dibujar en un determinado ángulo.

Ahora, nos centraremos en el dibujo del interior del triángulo, que detallamos así:

> Se dibuja el interior del triángulo que estamos considerando,
> en la posición donde está actualmente la plumilla,
> a la izquierda según la dirección en la que está actualmente la plumilla, y
> al finalizar, el ángulo de la plumilla debe ser el ángulo inicial.




En primer lugar se dibuja una línea desde el lugar *donde estamos* hasta el punto medio del primer lado del triángulo interior, hacemos la primera llamada recursiva (1) y acabamos de dibujar el primer lado. Vamos con el segundo lado: dibujamos la primera mitad, hacemos la segunda llamada recursiva (2) y acabamos de dibujar el segundo lado. Dibujamos la primera mitad del tercer lado, hacemos la tercera llamada recursiva (3), y... en este punto podemos caer en la tentación de completar la mitad que nos falta del tercer lado.

Pero ¡ay! en la descripción que hemos hecho, hemos dibujado los triángulos externos, ¡y nos hemos olvidado del triángulo interior! Este triángulo se puede dibujar en cualquier momento, justo antes o después de dibujar cualquier triángulo externo. Pero ¿cómo? Los triángulos se dibujan siempre en la parte izquierda de la dirección actual, de forma que, para dibujarlo en la parte interna, habrá que girar la plumilla $\pi$ radianes (180 grados) para que ésta apunte en sentido contrario. Y después de haber hecho la llamada recursiva deberemos volver a la dirección original volviendo a girar $\pi$ radianes.

Para poder distinguir de forma adecuada cada uno de los niveles de dibujo, cada nivel se dibuja en un color diferente. Así el programa recursivo queda:

```
void dibujarRecursivo(double const lado, int const nivel) {
  if (nivel > 0) {
    ponerColor(nivel);
    entrar();
    dibujarLinea(lado/2);
    dibujarRecursivo(lado/2, nivel-1);
    dibujarEsquina(lado);
    dibujarRecursivo(lado/2, nivel-1);
    dibujarEsquina(lado);
    dibujarRecursivo(lado/2, nivel-1);
    girar(M_PI);
    dibujarRecursivo(lado/2, nivel-1);
    girar(-M_PI);
    dibujarLinea(lado/2);
    salir();
    ponerColor(nivel+1);
  }
}
```

Hemos hecho uso de los siguientes pocedimientos auxiliares:

```
void entrar() {
  girar(2*M_PI/3);
}
```

```
void salir() {
  girar(2*M_PI/3);
}
void dibujarEsquina(double const lado) {
  dibujarLinea(lado/2);
  girar(-2*M_PI/3);
  dibujarLinea(lado/2);
}
void ponerColor(int const nivel) {
  Colores const colores[] = {rojo, amarillo, azul, verde};
  ponerColor(colores[nivel%4]);
}
```

Y ya sólo falta el programa que dibuja el exterior del triángulo:

```
void dibujarTriangulos(double const tam, int const nivel) {
  double tamanyo = tam;
  ponerCoordenadas(tamanyo*.1, tamanyo*.2);
  tamanyo = tamanyo*0.8;
  ponerColor(nivel+1);
  dibujarLinea(tamanyo/2);
  dibujarRecursivo(tamanyo, nivel);
  ponerColor(nivel+1);
  dibujarLinea(tamanyo/2);
  girar(2*PI/3);
  dibujarLinea(tamanyo);
  girar(2*PI/3);
  ponerColor(nivel+1);
  dibujarLinea(tamanyo);
}
```

## 3.26 Cálculo puntual de la matriz de mediotono de Judice-Jarvis-Ninke

123 130

Para encontrar el elemento de la posición $(i, j)$ hay que vagar por la matriz, empezando en el nivel más externo y pasando a la submatriz adecuada del nivel anterior. Veamos cómo sería el recorrido para buscar el elemento $(4, 3)$ en una matriz de $8 \times 8$, que llamaremos $L_3$. Como $(4, 3)$ cae en la submatriz de $4 \times 4$ de la esquina superior izquierda ($L_2$) pasamos a buscarlo ahí. En este nuevo nivel, $(4, 3)$ cae en la submatriz de $2 \times 2$ de la esquina inferior derecha ($L_1$). A continuación deberíamos buscar la posición $(2, 1)$ en $L_1$. Observa que esta posición es la misma que la $(4, 3)$ de $L_2$, y que además cae en la esquina inferior izquierda de $L_1$ ($L_0$). La posición que debemos buscar en $L_0$ es la $(1, 1)$: sólo disminuye el índice de fila.

Una vez que sabemos cómo localizar una cierta posición, calcular el valor que ha de contener es relativamente sencillo. Cada una de las dos posibles implementaciones refleja una forma de realizar este cálculo. Con recursión, los cálculos se realizan cuando se hace el camino de vuelta; mientras que pasar a una de las submatrices se convierte en una llamada recursiva (la mejor forma de calcular $2^m$ en C++ es con `1 << m`):

```
int limb(int const i, int const j, int const n) {
  if (n == 0) {
    return 0;
  } else {
```

```
    int const m = 1 << (n-1);  // 2^(n-1)
    if (i <= m) {
      if (j <= m) return 4*limb(i, j, n-1);
      else return 4*limb(i, j - m, n-1) + 2;
    } else {
      if (j <= m) return 4*limb(i - m, j, n-1) + 3;
      else return 4*limb(i - m, j - m, n-1) + 1;
    }
  }
}
```

En la versión iterativa hay que hacer los cálculos según vamos pasando a las matrices más internas. No hay camino de vuelta; cuando localicemos la posición tendremos almacenado su valor en la variable `r`. Cada paso de nuestro camino exige aumentar el valor de `r` con 0, 1, 2 o 3, dependiendo de a qué submatriz pasemos. Como hay que tener presente el producto por 4, resulta que los incrementos anteriores sólo sirven para el primer paso, mientras que en el segundo hay que multiplicarlos por 4, en el tercero por 16 y así sucesivamente. La potencia de 4 con la que estemos trabajando se tendrá en la variable `b` y se incrementará en cada paso de nuestro camino:

```
int limb(int const i, int const j, int const n) {
  int pi = i, pj = j;
  int m = 1 << n;  // 2^n
  int r = 0, b = 1;
  while (m > 1) {
    m >>= 1;
    if (pi > m) {
      pi -= m;
      if (pj > m) {
        pj -= m;
        r += b;
      } else {
        r += 3*b;
      }
    } else if (pj > m) {
      pj -= m;
      r += 2*b;
    }
    b *= 4;
  }
  return r;
}
```

Las dos soluciones anteriores recorrían la matriz pasando a submatrices cada vez más pequeñas. También se puede recorrer en el sentido opuesto, empezando por una matriz $L_0$ y pasando a matrices cada vez más grandes. Si suponemos que los índices $i$ y $j$ son naturales entre 0 y $2^n - 1$, su representación en base 2 indica la posición relativa de un nivel con respecto al siguiente. En el ejemplo que pusimos al principio, las representaciones en base 2 de $4 - 1$ y $3 - 1$ son 11 y 10 respectivamente. La relación entre $L_0$ y $L_1$ viene definida por los bits menos significativos: $L_0$ está en la parte inferior de $L_1$ por el 1 de 11, y en la parte izquierda por el 0 de 10. Igualmente, los dos siguientes bits indican la relación entre $L_1$ y $L_2$: como ambos son 1, $L_1$ está en la esquina inferior derecha de $L_2$. Finalmente, la relación entre $L_2$

y $L_3$ viene dada por los dos siguientes bits que no aparecen porque son ceros a la izquierda y, por tanto, $L_2$ está en la esquina superior izquierda de $L_3$. El programa que sigue utiliza esta búsqueda. El acceso a los bits se hace explorando el bit menos significativo (con el resto de la división entre 2) y descartándolo a continuación (con una división entera entre 2):

```
int limb(int const i, int const j, int const n) {
  int pi = i-1, pj = j-1, k = n;
  int r = 0;
  while (k > 0) {
    if (pi % 2 == 1) {
      if (pj % 2 == 1) r = 4*r + 1;
      else r = 4*r + 3;
    } else {
      if (pj % 2 == 1) r = 4*r + 2;
      else r = 4*r;
    }
    k--;
    pi / 2;
    pj / 2;
  }
  return r;
}
```

## 3.27 Dibujo de árboles mediante fractales

124 130

Como se ha dicho en el enunciado, este programa es sencillo, al menos algorítmicamente:

```
void dibujarArbol(double const tamanyo, Posicion const& posicion,
                  double const angulo, int const nivel) {
  dibujarTronco(tamanyo, posicion, angulo);
  if (nivel > 1) {
    Posicion posAux;
    posAux = coordenada1(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo-20, nivel-1);
    posAux = coordenada2(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo-20, nivel-1);
    posAux = coordenada3(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo, nivel-1);
    posAux = coordenada4(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo+20, nivel-1);
    posAux = coordenada5(tamanyo, posicion, angulo);
    dibujarArbol(tamanyo/2, posAux, angulo+20, nivel-1);
  }
  else dibujarHoja(tamanyo, posicion, angulo);
}
```

Podemos observar que estamos usando un tipo nuevo llamado `Posicion`. Si bien no se verá hasta el capítulo 4, su uso es muy sencillo y simplifica la solución. Se podría haber hecho la solución sustituyendo las variables de tipo `Posicion` por dos variables de tipo `double`.

Pasemos a los detalles pendientes. En la cabecera vemos que hace falta indicar dónde empezamos a dibujar y el ángulo en que se trazará el árbol. Esto se ha adelantado por simplicidad, pero su necesidad va a ponerse de manifiesto en seguida.

Empecemos por ver cómo dibujar el tronco. Aparentemente, el tronco es un triángulo isósceles; pero la mitad superior del tronco *es* un subárbol, por lo que el tronco propiamente dicho será la *mitad inferior* de ese triángulo, esto es, un trapecio de forma que la base superior mide la mitad que la inferior, según el teorema de Tales (Tales de Mileto, 624–547 a.C.), y está centrada con respecto a ésta. El tamaño de la base es un tanto arbitrario; obviamente debe ser relativamente pequeña con respecto a la altura del árbol. Aquí hemos tomado 1/10 de la altura total del árbol. Entonces, si `tam == tamanyo/2`, las coordenadas del trapecio serán $(0, 0)$, $(0.05 \times \texttt{tam}, \texttt{tam})$, $(0.15 \times \texttt{tam}, \texttt{tam})$ y $(0.2 \times \texttt{tam}, 0)$. Pero hay que tener en cuenta que estas coordenadas son relativas a la base del dibujo (`posicion`) y están rotadas un `angulo`. Para calcular ese cambio de coordenadas se usa el procedimiento `cambioCoordenadas`.

```
void dibujarTronco(double const tamanyo, Posicion const& posicion,
                   double const angulo) {
  ponerColor(1.0, 0.7, 0.0);
  Posicion pos[4];
  double tam = tamanyo/2;       // altura
  pos[0].x = posicion.x; pos[0].y = posicion.y;
  pos[1] = cambiarCoordenadas(posicion, angulo, 0.05*tam, tam);
  pos[2] = cambiarCoordenadas(posicion, angulo, 0.15*tam, tam);
  pos[3] = cambiarCoordenadas(posicion, angulo, 0.2*tam, 0);
  double cx[4], cy[4];
  for (int i = 0; i < 4; i++) {
    cx[i] = pos[i].x; cy[i] = pos[i].y;
  }
  dibujarPoligono(cx, cy, 4);
}
```

Vamos con el trazado de las hojas. El `tamanyo` de la hoja lo establecemos como 1/10 del tronco, pero cuando estamos en un nivel bastante avanzado en la recursión ese `tamanyo` resulta demasiado pequeño y no se aprecia, por lo que establecemos un `tamanyo` mínimo. La posición debe ser encima del tronco, sobre su eje:

```
void dibujarHoja(double const tamanyo, Posicion const& posicion,
                 double const angulo) {
  ponerColor(verde);
  double tam = tamanyo/2; // tamaño del tronco
  double radio = tam*0.1; // un 1/10 del tamaño del tronco
  radio = radio < 10 ?  10 :  radio; // el tamaño mínimo es 10
  Posicion aux = cambiarCoordenadas(posicion, angulo, tam*0.1, tam*1.08);
  dibujarDisco(aux.x, aux.y, radio);
}
```

Los procedimientos para calcular las coordenadas dependen del sitio exacto donde deseemos colocar los subárboles, del `tamanyo` del árbol y del `angulo` actual de dibujo. Por ejemplo, la base del primer subárbol está elevada 1/5 del tamaño del árbol con respecto a su base. Si tenemos en cuenta que el lado superior del trapecio que forma el tronco es la mitad que el inferior y éste es, a su vez, 1/10 de la altura del árbol, la coordenada $x$ de la base debe estar desplazada $0.05 \times$ `tamanyo` (otra vez Tales). Así, el procedimiento para la primera coordenada será:

```
Posicion coordenada1(double const tamanyo, Posicion const& pos,
                     double const angulo) {
  return cambiarCoordenadas(pos, angulo, tamanyo*0.05, tamanyo*0.2);
}
```

Los demás serán similares. Poco hay que decir sobre el procedimiento de cambio de coordenadas, que se resuelve con trigonometría elemental. Simplemente hay que observar que hemos tratado los ángulos en grados, mientras que las funciones trigonométricas predefinidas trabajan con radianes, por lo que se requiere la oportuna conversión:

```
Posicion cambiarCoordenadas(Posicion const& pos, double const angulo,
                            double const x, double const y) {
  Posicion aux;
  double anguloRadianes = PI*angulo/180;
  aux.x = pos.x + x*cos(anguloRadianes) - y*sin(anguloRadianes);
  aux.y = pos.y + y*cos(anguloRadianes) + x*sin(anguloRadianes);
  return aux;
}
```

Aún podemos mejorar el trazado de árboles descrito introduciendo cierta aleatoriedad en el dibujo de los subárboles. Para ello, podemos añadir un parámetro `corte` que indica la probabilidad con la que queremos dibujar los subárboles. Hemos de tener en cuenta que el tercer árbol se debe dibujar en cualquier caso. Para decidir si dibujamos cada subárbol, podemos usar una distribución de Bernoulli con el parámetro deseado (véase el ejercicio 3.13). Además, cada vez que hacemos una llamada recursiva, la probabilidad debe disminuir multiplicando por `indiceDisminucionProbabilidad` que está definida como una constante entre 0 y 1. Es necesario hacer pocas modificaciones al programa anterior: en primer lugar, añadir el parámetro `corte`, (que recoge la probabilidad con que se dibujarán los subárboles) en la cabecera de la función:

```
void dibujarArbol(double const tamanyo, Posicion const& posicion,
                  double const angulo, int const nivel, double const corte);
```

Y cada llamada recursiva (salvo la tercera) ha de protegerse dentro de un condicional como éste:

```
if (uniforme(0, 1) <= corte) {
  dibujarArbol(tamanyo/2, posAux, angulo-20, nivel-1,
               corte*indiceDisminucionProbabilidad);
}
```

## 3.28 Dragones y teselas

126

Observando la figura 3.6, podemos comprobar que los siguientes pasos generan una curva dragón:

1. Girar $\pi/4$ radianes (45 grados) a la derecha.
2. Generar un dragón a la derecha (según la línea previa de puntos).
3. Girar $\pi/2$ radianes (90 grados) a la izquierda.
4. Generar un dragón a la izquierda.

El trazador que nos dan para la realización de este ejercicio tiene, en cada momento, una posición de dibujo y una orientación. Si empezamos a dibujar en una posición y una orientación, debemos acabar

dibujando en una posición incrementada en el tamaño del dragón según la dirección original y en esa misma orientación; por tanto antes de acabar será necesario recuperar la orientación:

5. Girar $\pi/4$ radianes (45 grados) a la izquierda.

Pero lo descrito expresa el *caso inductivo*: todavía no hemos dibujado nada. Los trazos del dibujo se harán en el caso base, que consistirá únicamente en una línea (dragón de grado 0) de la longitud requerida. Los giros comentados siempre son relativos a la orientación en la que deseamos hacer el dibujo. En lo descrito estamos suponiendo que queremos dibujar un dragón "a la derecha" (tal y como se ve en la figura 3.6). Pero si queremos dibujar un dragón "a la izquierda", habrá que cambiar de mano. Para representar la orientación definimos sendas constantes:

```
int const derecha = 1;
int const izquierda = -1;
double const reduccion = sqrt(2)*0.5; // proporción del lado de dos dragones consecutivos
```

Así, el procedimiento recursivo para dibujar el dragón queda como sigue:

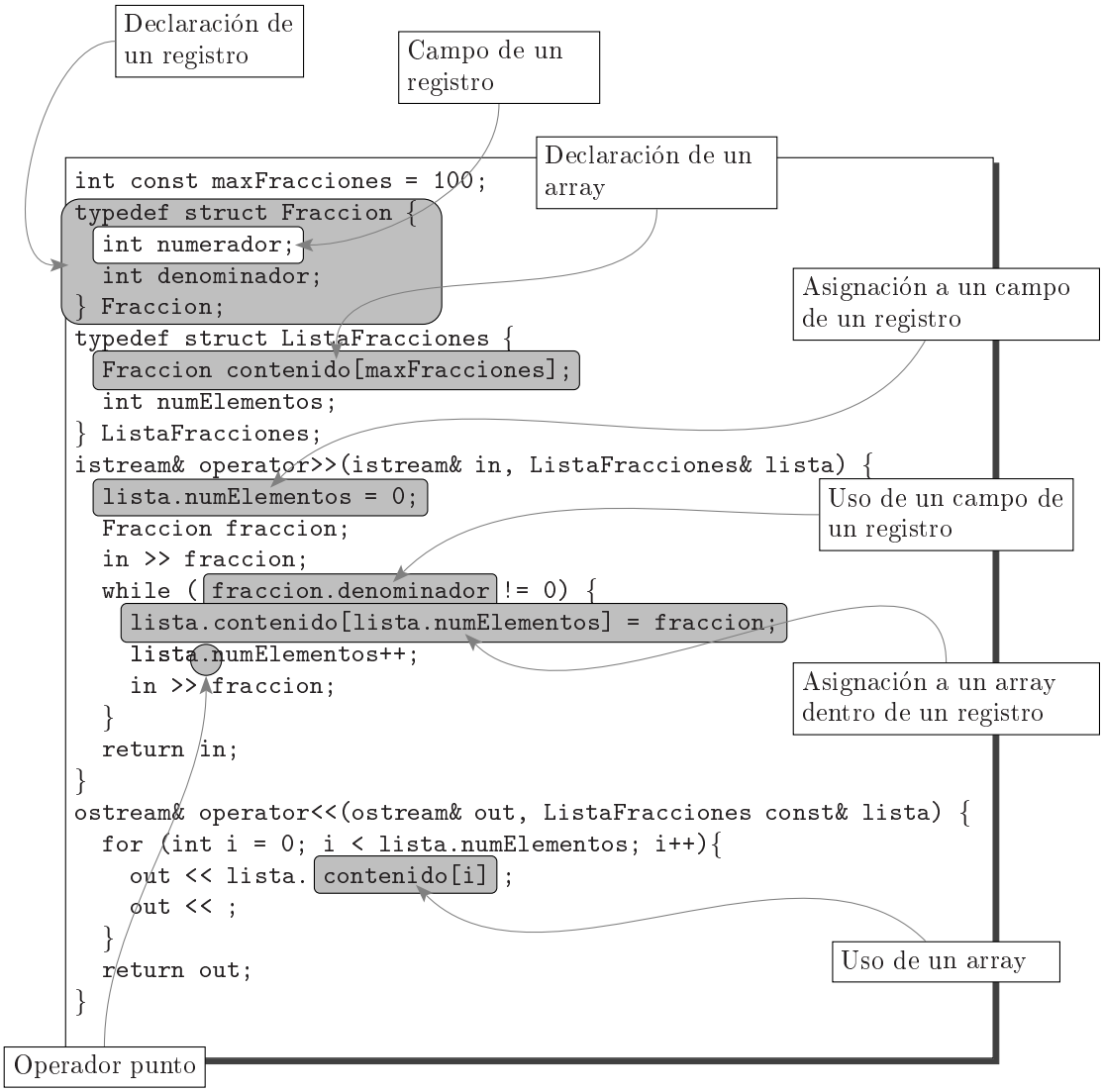
```
void dibujaDragon(double const tamanyo, int const orientacion, int const nivel) {
  if (nivel > 0) {
    girar(orientacion * (-PI/4));
    dibujaDragon(tamanyo*reduccion, derecha, nivel-1);
    girar(orientacion * (PI/2));
    dibujaDragon(tamanyo*reduccion, izquierda, nivel-1);
    girar(orientacion * (-PI/4));
  } else {
    dibujarLinea(tamanyo);
  }
}
```

Vamos a hacer un procedimiento que dibuje cuatro dragones acoplados tal y como aparecen en la figura 3.7:

```
void dibujaDragones(double const tamanyo, int const nivel) {
  double angulo = PI/4;
  Colores const color[] = {rojo, azul, verde, amarillo};
  for (int i = 0; i < 4; i++) {
    ponerCoordenadas(tamanyo*0.5, tamanyo*0.5);
    ponerAngulo(angulo);
    ponerColor(color[i]);
    dibujaDragon(tamanyo*0.4, derecha, nivel);
    angulo += PI/2;
  }
}
```

# 4 Capítulo CUATRO

# Definición de tipos



# 4444444444444444444444444444444444444444 En este capítulo

Todo programa no trivial exige la definición de nuevos tipos que servirán para aglutinar y almacenar los datos de trabajo. Estos tipos, según se añadan nuevas operaciones en forma de subprogramas que los manejen, irán constituyendo el centro de piezas que tomarán identidad por sí mismas y que podrán ser de utilidad en otros programas.

En este capítulo practicamos con los aspectos más básicos de la definición de tipos en C++. Esta visión crecerá y se completará en los dos siguientes capítulos.

## 4.1 Arrays

Un *array* es una ristra de elementos del mismo tipo. Mediante la declaración

```
Tipo arr[N];
```

se introduce el array `arr`, formado por `N` elementos de `Tipo`. La expresión `N` ha de ser constante en el momento de la compilación. Los elementos de este array tienen índices del 0 al $N-1$, ambos incluidos. Se accede a los elementos con el operador de indexación: `arr[indice]`. Una indexación puede aparecer en la parte izquierda de una asignación.

El bucle `for` es la forma más fácil de recorrer un array. Supongamos que tenemos el array

```
double pesos[N];
```

que va a contener pesos. Para definir este array de forma que todos los pesos sean cero, escribimos

```
for (int i = 0; i < N; i++) pesos[i] = 0;
```

Sumamos todos los pesos así:

```
pesoTotal = 0;
for (int i = 0; i < N; i++) pesoTotal = pesoTotal + pesos[i];
```

No sólo se pueden declarar variables de tipo array, sino también definirlas; los distintos valores del array se separan por comas y se rodean con unas llaves:

```
double ejeY[3] = {0, 1.0, 0};
```

Se puede omitir el tamaño, y el compilador contará los elementos dados:

```
double ejeZ[] = {0, 0, 1.0};
```

Se pueden definir arrays de más de una dimensión; basta añadir, entre corchetes, tantas longitudes como dimensiones queramos. Por ejemplo, una matriz de `N` × `M` reales se define así:

```
double mat[N][M];
```

Para acceder a la posición (`i`, `j`) hay que escribir `mat[i][j]`. Es correcto escribir `mat[i]`: estaremos accediendo a la fila `i` de la matriz, que es un array unidimensional de longitud `M`.

Los parámetros de tipo array en un subprograma tienen ciertas prescripciones curiosas. La primera y más importante es que siempre se pasan por referencia; no hay cabida ni para `const` ni para `&`. La segunda es que también se devuelven por referencia cuando son el resultado de una función; por tanto, una función nunca debe devolver un array que sea una variable local. La tercera es que la longitud del primer índice se puede omitir; esto permite definir subprogramas independientes del tamaño de los arrays. Pero C++ no pasa al subprograma información relativa al tamaño real del array; es responsabilidad nuestra añadir un parámetro adicional con esa información si queremos escribir subprogramas realmente independientes del tamaño. Por ejemplo, la función que suma todos los elementos de un array se puede

escribir como sigue:

```
double suma(double arr[], int const longitud) {
  double laSuma = 0;
  for (int i = 0; i < longitud; i++) laSuma = laSuma + arr[i];
  return laSuma;
}
```

## 4.2 De cómo nombrar tipos

Es interesante poder dar nombre a un nuevo tipo. Se consigue con la palabra reservada `typedef`, que tiene una sintaxis cuanto menos curiosa: debe preceder a una declaración correcta de variable. Pero no definirá una nueva variable, sino un nuevo tipo. Así, para definir un `Vector` como un array de tamaño `N` debemos escribir:

```
typedef double Vector[N];
```

Sin el `typedef` estaríamos declarando la variable `Vector`; pero con el `typedef` estamos definiendo el tipo `Vector` como equivalente a un array con `N` `double`s.

Como colofón de este apartado y del anterior, y a modo de ejemplo ilustrativo, implementaremos algunas operaciones tradicionales del álgebra matricial. Tendremos vectores de dimensión `N` y matrices $\texttt{N} \times \texttt{N}$:

```
typedef double Vector[N];
typedef Vector Matriz[N];
```

El producto escalar de dos vectores se puede escribir así:

```
double productoEscalar(Vector v1, Vector v2) {
  double pe = 0;
  for (int i = 0; i < N; i++) pe = pe + v1[i]*v2[i];
  return pe;
}
```

La operación para sumar dos vectores no puede ser una función porque (1) la suma habrá que guardarla en un array local, (2) los arrays se devuelven por referencia y (3) nunca se debe devolver una variable local por referencia. Por tanto, no queda más remedio que escribir esta operación como una acción con tres argumentos: los dos primeros serán los vectores que queremos sumar, y el tercero el vector donde vamos a dejar el resultado:

```
void suma(Vector v1, Vector v2, Vector v3) {
  for (int i = 0; i < N; i++) v3[i] = v1[i] + v2[i];
}
```

Los tres parámetros, por ser arrays, se pasan por referencia. Que los dos primeros argumentos son de entrada y el tercero de salida es algo que no se puede plasmar directamente en el código y habría que hacerlo en forma de comentario al programa.

Terminamos con el producto de una matriz por un vector. Igual que la operación anterior, ésta también será una acción que tendrá como tercer parámetro un vector donde dejaremos el resultado:

```
void producto(Matriz mat, Vector vec, Vector result) {
  for (int i = 0; i < N; i++) result[i] = productoEscalar(mat[i], vec);
}
```

## 4.3 Registros

Con los arrays podemos juntar sólo elementos del mismo tipo. Con los registros es posible juntar diferentes tipos para formar otro. Cada uno de los elementos componentes se llama *campo* y ha de tener un nombre. La sintaxis para declarar una variable que sea un registro es muy simple: cada campo se declara a su vez como si fuera una variable, todos ellos se rodean con unas llaves y se preceden con la palabra clave `struct`; el nombre de la variable termina esta construcción. Por ejemplo, con el siguiente registro podríamos representar un número complejo:

```
struct {
  double parteReal;
  double parteImaginaria;
} complejo;
```

Y con este otro podríamos guardar información de una persona:

```
struct {
  char nombre[100];
  int edad;
  int dni;
  double altura;
} persona;
```

Casi nunca se utilizan los registros para definir una variable directamente, porque generalmente se necesita más de una variable de ese mismo tipo registro y repetir la definición es muy costoso. Lo normal es definir un tipo con `typedef`:

```
typedef struct Complejo {
  double parteReal;
  double parteImaginaria;
} Complejo;
```

Que se repita el nombre del tipo también detrás de `struct` es una costumbre que viene de C, que es necesaria en ciertas ocasiones y que en este libro siempre respetaremos.

A un campo de una variable de tipo registro se accede poniendo un punto entre la variable y el campo:

```
Complejo c;
c.parteReal = 1;
c.parteImaginaria = 2;
cout << "Parte real:  " << c.parteReal
     << ", parte imaginaria:  " << c.parteImaginaria
     << endl;
```

Al igual que con los arrays, una variable de tipo registro no sólo se puede declarar, sino que también se puede definir. Los valores de los campos se especifican entre llaves, separados por comas y respetando el orden de aparición dentro del registro:

```
Complejo c_i = {0.0, 1.0};
```

Los registros no tienen ninguno de los comportamientos extraños de los arrays cuando son parámetros de subprogramas. Se pasan por valor o por referencia dependiendo de si aparece el calificador `const` o el calificador `&`. Esto es así incluso si el registro contiene un array; de hecho, una buena forma de hacer que los arrays se *comporten* es encerrarlos dentro de un registro:

```
int const N = 3;
typedef struct Vector {
  double datos[N];
} Vector;
typedef struct Matriz {
  Vector datos[N];
} Matriz;
```

Las operaciones ahora se pueden escribir como funciones porque los registros se devuelven por valor:

```
double productoEscalar(Vector const& vec1, Vector const& vec2) {
  double pe = 0;
  for (int i = 0; i < N; i++) pe = pe + vec1.datos[i]*vec2.datos[i];
  return pe;
}
Vector suma(Vector const& vec1, Vector const& vec2) {
  Vector result;
  for (int i = 0; i < N; i++) result.datos[i] = vec1.datos[i] + vec2.datos[i];
  return result;
}
Vector producto(Matriz const& mat, Vector const& vec) {
  Vector result;
  for (int i = 0; i < N; i++) {
    result.datos[i] = productoEscalar(mat.datos[i], vec);
  }
  return result;
}
```

Terminaremos este apartado retomando el ejemplo de los complejos. La práctica matemática utiliza los mismos operadores para manipular números complejos que para números reales o enteros. Por tanto, es razonable sobrecargar los operadores aritméticos para que se puedan usar con números complejos. Por ejemplo, la suma quedaría así:

```
Complejo operator+(Complejo const& c1, Complejo const& c2) {
  Complejo result = {c1.parteReal + c2.parteReal,
                     c1.parteImaginaria + c2.parteImaginaria};
  return result;
}
```

Seguir con esta tarea hasta completar la definición de todas las operaciones sobre complejos es una actividad interesante que dejamos como ejercicio. Pero para que la biblioteca de funciones resultante sea realmente útil, también habrá que añadir operaciones para manipulación cruzada; por ejemplo, para sumar un real y un complejo:

```
Complejo operator+(double const r, Complejo const& c) {
  Complejo result;
  result.parteReal = r + c.parteReal;
  result.parteImaginaria = c.parteImaginaria;
  return result;
}
```

Si además contamos con una sobrecarga cruzada del producto y la definición del número $i$ ($= \sqrt{-1}$),

```
Complejo const I = {0, 1};
```

podremos escribir cosas tan elegantes como `2 + 3*I` para representar al número complejo $2 + 3i$. Esta capacidad para hacer que un tipo nuevo se comporte casi como si estuviera predefinido en el lenguaje es una de las características distintivas de C++.

## 4.4 Enumeraciones

Las enumeraciones son la vía para definir constantes que queremos utilizar en contextos restringidos. Cuando se define una enumeración, se crea a la vez un tipo y los únicos valores que contiene. Por ejemplo, si queremos marcar que las tres posibles `Direcciones` de movimiento en un espacio tridimensional son `ejeX`, `ejeY` y `ejeZ`, damos la siguiente definición:

```
typedef enum Direcciones {ejeX, ejeY, ejeZ} Direcciones;
```

La sintaxis es muy similar a la de los registros: las enumeraciones se marcan con `enum` y lo que aparece rodeado entre llaves son sus valores. A los tipos definidos como una enumeración se los califica de *enumerados*.

Un tipo enumerado es un tipo entero; sus valores son constantes enteras. En muchas circunstancias podemos ignorar la asignación de valores que realiza C++. Pero en otras, cuando se ven involucrados arrays o bucles, es importante saber que C++ respeta el orden que hemos dado y asigna consecutivamente números a partir del 0.

```
typedef double Caja[ejeZ+1][2];
double volumen(Caja caja) {
  double elVolumen = 1;
  for (int dir = ejeX; dir <= ejeZ; dir++) {
    elVolumen *= caja[dir][1] - caja[dir][0];
  }
  return elVolumen;
}
```

Pero si tenemos alguna preferencia distinta, la podemos imponer:

```
typedef enum Nota {
  ut = 1, re = 3, mi = 5, fa = 6, sol = 8, la = 10, si = 12, silencio = 0
} Nota;
```

Los tipos enumerados no son compatibles entre sí. Son compatibles con los tipos enteros, pero para el paso inverso hay que hacer una conversión anteponiendo el nombre del tipo:

```
Nota n = (Nota)(mi+1);
```

No hay ninguna operación predefinida para los tipos enumerados; siempre que se hace aritmética con ellos es porque se están tratando como algún tipo entero. Por eso, en la función `volumen` se declara como `int` la variable con la que se recorren las direcciones.

## 4.1 Ponle tú el título

193

Indica cuál es la salida de la siguiente función:

```
int const tamanyo = 3;
typedef int Tabla[tamanyo][tamanyo];
bool x(Tabla tabla)
{
  int i = 0;
  bool resp = true;
  while(i < tamanyo && resp) {
    int j = 0;
    while (j < tamanyo-1 && resp) {
      resp = tabla[i][j] < tabla[i][j+1];
      j++;
    }
    i++;
  }
  return resp;
}
```

## 4.2 Yahoos

193

Los Yahoos cuentan con los dedos: *uno*, *dos*, *tres* y *cuatro*. Conocen el *cero*, pero no conciben cantidades negativas: a la izquierda del *cero* sitúan acertadamente lo *desconocido*. Conscientes de su ignorancia, sospechan la existencia de una cantidad mayor que *cuatro*: *muchos*.

### 4.2.1 Tipo de datos

Define un tipo enumerado para expresar *cantidades*, que comprenda toda la gama conocida por los Yahoos.

### 4.2.2 Escritura de cantidades

Da un subprograma de escritura para tales cantidades.

### 4.2.3 Operación siguiente

Para contar, ya hace siglos que los Yahoos conocen la operación *siguiente*, que convierte una cantidad en otra, así:

- El siguiente de lo *desconocido* continúa siendo *desconocido*
- Para las cantidades *cero*... *cuatro*, el siguiente es *uno*... *muchos* respectivamente.
- Por último, el siguiente de *muchos* es también *muchos*.

Define una función que responda a esa descripción.

### 4.2.4 Operación suma

Los Yahoos emplean una función *suma*, que actúa así:

- La suma de *desconocido* con cualquier cantidad, a la derecha o a la izquierda, resulta nuevamente *desconocido*.
- $suma(cero, x) \rightsquigarrow x$, para cualquier $x$ no *desconocido*.
- $suma(uno, x) \rightsquigarrow siguiente(x)$, para cualquier $x$ no *desconocido*.
- $suma(dos, x) \rightsquigarrow siguiente(siguiente(x))$, para cualquier $x$ no *desconocido*.

  ...
- $suma(muchos, x) \rightsquigarrow muchos$, para cualquier $x$ no *desconocido*.

Define una función *suma* que trabaje del modo descrito.

### 4.2.5 Las tablas

Los Yahoos han elaborado tablas de sumar para ayudarse en sus transacciones comerciales con otros pueblos. Dichas tablas tienen la siguiente disposición:

| | desconocido | cero | uno | dos | tres | cuatro | muchos |
|---|---|---|---|---|---|---|---|
| desconocido | | | | | | | |
| cero | | | | | | | |
| uno | | | | | | | |
| dos | | | | | | | |
| tres | | | | | | | |
| cuatro | | | | | | | |
| muchos | | | | | | | |

donde las casillas centrales están destinadas a contener *cantidades*. Define un tipo de datos adecuado para representar esas tablas, así como una variable de este tipo para la tabla concreta de la suma.

### 4.2.6 Tabla de sumar

Define un subprograma para rellenar la tabla, apoyándose en la función para sumar descrita anteriormente.

### 4.2.7 Tabla de multiplicar

Suponemos que la tabla de multiplicar está almacenada en un archivo de texto, donde las cantidades están representadas por los caracteres '?', '0', '1', '2', '3', '4', 'M' para mayor comodidad, y se organizan por filas.

¿Cómo se carga una tabla de multiplicar, con los datos del archivo indicado?

### 4.2.8 Uso de los recursos definidos

Con las definiciones anteriores, escribe la(s) instrucción(es) que usarían los Yahoos para rellenar las tablas $s$ y $p$ y realizar lo siguiente,

$$a \leftarrow\ b + c * d$$

con, por ejemplo, $b = 1, c = 3$ y $d = 1$.

#### 4.2.9 Suma, recursivamente

Define una versión recursiva de la función para sumar antes descrita.

#### 4.2.10 Cálculo sin tablas

Escribe un fragmento de programa que realice la operación del apartado 4.2.8 sin hacer uso de las tablas.

#### 4.2.11 Bibliografía

El primer cronista, que sepamos, de la vida y el modo de pensar de los Yahoos fue Gulliver en su *Viaje al país de los Houyhnhnms*, tal como nos lo cuenta Jonathan Swift. Pero su aritmética, tal como se describe en este ejercicio, procede de *El informe de Brodie*, de J. L. Borges.

## 4.3 Regla de Ruffini

195

La regla de Paolo Ruffini (1765–1822) sirve para realizar de forma sencilla la división de un polinomio (de grado mayor que uno) entre un binomio de la forma $x - a$, ambos con coeficientes enteros.

***Ejemplo*** Aquí tienes la división de $x^3 + 2x^2 - x - 2$ entre $x - 3$:

| | 1 | 2 | −1 | −2 |
|---|---|---|---|---|
| 3 | | 3 | 15 | 42 |
| | 1 | 5 | 14 | 40 |

El cociente de la división es $x^2 + 5x + 14$ y el resto es 40; además, al tomar como divisor $x - 3$, 40 es también la evaluación del polinomio en el punto $x = 3$.

***División de un polinomio por un binomio*** Realiza un programa que aplique la regla de Ruffini para dividir un polinomio $p(x)$ de coeficientes enteros por un binomio de la forma $x - a$, siendo $a$ un entero.

***División repetida de un polinomio por un binomio*** Realiza un programa que calcule cuántas veces es divisible un polinomio de coeficientes enteros por el binomio $x - a$, donde $a$ es un entero.

***Cálculo de todas las raíces enteras*** Realiza un programa que calcule *todas* las raíces enteras de un polinomio, sabiendo que cualquier raíz entera debe ser un divisor del término independiente.

## 4.4 Un pequeño sistema formal

189

Un *sistema formal* define cómo generar cadenas de símbolos. Para ello, se tienen unas cadenas iniciales que llamaremos *axiomas* y unas *reglas* que dicen cómo construir cadenas nuevas. A los axiomas y a las cadenas generadas por las reglas los llamaremos *teoremas*.

Consideremos el sistema formal **mg** que únicamente tiene una familia de axiomas y una regla para definir teoremas. Los axiomas del sistema **mg** son las cadenas de la forma $x$`m-g`$x$`-`, siempre que $x$ sea una cadena que esté compuesta únicamente por guiones; el símbolo $x$ debe representar la misma cadena en las dos ocasiones en que aparece. La única regla del sistema **mg** dice que, si $x$, $y$ y $z$ son cadenas formadas únicamente por guiones y $x$`m`$y$`g`$z$ es un teorema, entonces la cadena $x$`m`$y$`-g`$z$`-` es también un teorema.

***Ejemplo*** Las cadenas `-m-g--` y `--m-g---` son axiomas del sistema **mg**; en la primera $x$ es `-` y en la segunda $x$ es `--`. Las cadenas `-----m-g------` y `--------m-g---------` también son axiomas.

Para encontrar teoremas nuevos tenemos que partir de otros ya conocidos. Los axiomas son teoremas; por tanto `---m-g----` es un teorema, siendo $x$ la cadena `---`, $y$ la cadena `-` y $z$ la cadena `----`. Según la regla del sistema **mg**, la cadena $x$`m`$y$`-g`$z$`-` es también un teorema; es decir, la cadena `---m--g-----` es un teorema. Sobre este último teorema podríamos aplicar nuevamente la regla para formar otro teorema diferente.

Como puedes comprobar, la cantidad de axiomas y teoremas que podemos definir utilizando el sistema **mg** es infinita.

***¿Es teorema?*** Te proponemos que realices un programa que te ayude a identificar las cadenas del sistema **mg**. El programa tiene que recibir una cadena de símbolos y debe verificar si pertenece o no al sistema **mg** mediante la manipulación de sus elementos. (Véase la pista 4.4a.)

***Alternativas*** ¿Se te ocurren otras formas de comprobar cuándo una cadena pertenece al sistema **mg**? Tus posibles alternativas, ¿se fijan en la *forma* de las cadenas, o en el *contenido* de las cadenas? Implementa estas otras posibilidades para comprobar si una cadena es teorema o no en el sistema **mg**.

***Bibliografía*** El ejercicio está inspirado en el libro de Douglas R. Hofstadter (1945–) *Gödel, Escher, Bach, an Eternal Golden Braid* [Hof99], un libro diferente, lleno de poesía, de vida, de matemáticas, de imágenes, de música y, como no, de sistemas formales. Esta obra ha recibido, entre otros, el premio Pulitzer. La página personal de Douglas R. Hofstadter es `http://www.psych.indiana.edu/people/homepages/hofstadter.html`.

Debido a la complejidad de la obra, la traducción al castellano ha sido más un proceso de reescritura que de traducción. Fíjate, por ejemplo, en el título: *Gödel, Escher, Bach, un Eterno y Grácil Bucle* [Hof01].

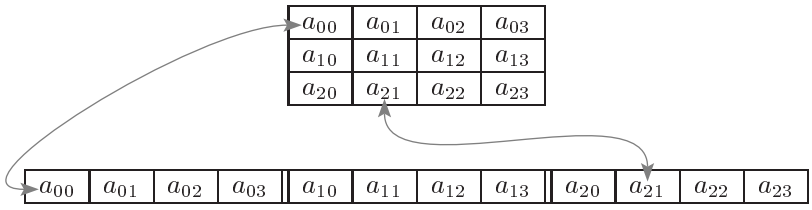
## 4.5 Tratamiento de matrices como vectores

189

Muchos lenguajes de programación permiten definir matrices que admiten el acceso a sus componentes utilizando las coordenadas de la fila y la columna. Por ejemplo en C++, escribimos `int m[3][4]` para definir la matriz `m`, que tiene 3 filas y 4 columnas.

Sin embargo, estrictamente hablando, tal capacidad del lenguaje puede considerarse redundante. La definición y el uso de las matrices pueden simularse mediante arrays lineales.

***Ejemplo*** Consideremos la siguiente matriz, con 12 componentes organizadas en 3 filas de a 4. Los mismos 12 elementos también podríamos organizarlos de forma consecutiva:




Acceder a la componente $a_{00}$ en la matriz es ir, en la fila 0, a la columna 0; en el vector es, simplemente, ir a la posición 0. Acceder a la componente $a_{21}$ de la matriz es ir, en la fila 2, a la columna 1; en el vector es ir directamente a la posición 9. Etcétera.

***Matrices bidimensionales*** Escribe los subprogramas necesarios para poder operar con matrices, pero en última instancia sólo se han de utilizar arrays. (Véase la pista 4.5a.)

***Generalización*** El ejercicio puede resolverse en general para simular matrices de dimensión arbitraria.

***Matrices simétricas*** Esta misma idea se puede usar para la implementación de matrices simétricas y matrices triangulares, de forma que no se desperdicie memoria.

## 46 Centro de un vector

197

Consideremos un vector $V$ con índices entre 0 y $n$ y que contiene valores numéricos. Definimos el *centro*, $c$, del vector $V$ como el índice entre 0 y $n$ que verifica la siguiente propiedad:

$$\sum_{i=0}^{c-1}(c-i)\cdot V[i] = \sum_{j=c+1}^{n}(j-c)\cdot V[j]$$

Esta propiedad no siempre se verifica; en ese caso, decimos que el vector no tiene centro.

Escribe un subprograma que dado un vector nos devuelva dos valores: el primero indica si existe el centro del vector, y el segundo, el índice en el que se encuentra en caso de existir.

***Ejemplo*** Consideremos el siguiente vector, con índices entre 0 y 4, que contiene, en este caso, números naturales:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 6 | 2 | 3 | 0 | 1 |

El centro de este vector es el índice 1. En efecto, si aplicamos la definición con $c = 1$ y con $n = 4$, obtenemos lo siguiente:

$$\sum_{i=0}^{c-1}(c-i)\cdot V[i] = 1\cdot 6 = \sum_{j=c+1}^{n}(j-c)\cdot V[j] = 1\cdot 3 + 2\cdot 0 + 3\cdot 1$$

Por el contrario, el siguiente vector no tiene centro,

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 2 | 1 | 1 |

porque con ninguno de los índices se cumple la igualdad.

***Bibliografía*** El centro de un vector puede interpretarse de forma física: si consideramos a los valores que contiene el vector como pesos, el centro del vector, si existe, es el *centro de gravedad*, es decir, un punto de equilibrio. Libros básicos para disfrutar aprendiendo física son [AF00] y [Fey98].

## 47 Un solitario

Tenemos una baraja española de 48 cartas, del 1 (*as*) al 12 (*rey*), y un ratito para jugar un solitario. El programa que te proponemos desarrollar simulará un solitario consistente básicamente en reconstruir la baraja de forma ordenada (desde el *as* hasta el *rey* de cada palo), sacando las cartas del mazo de dos en dos. Veamos el funcionamiento del juego:

- Inicialmente se dispone de toda la baraja, y los cuatro montones que hay que reconstruir están vacíos. (Se entiende que a cada palo le corresponde un montón.) Un montón comienza a construirse con su *as* correspondiente, tras él se podrá añadir el dos, tras éste el tres, y se continuará hasta completar cada montón con su *rey* respectivo, o hasta que no se pueda continuar con el solitario.
- Para añadir cartas a sus correspondientes montones, se toman del mazo dos de ellas y se descubren de forma que una quede encima de la otra. Se comprueba si la primera de ellas se puede colocar en el montón correspondiente a su palo. (Sólo se podrá colocar si es la siguiente a la carta mayor que haya en su montón o, en el caso de que su montón esté vacío, si es un *as*.) En caso afirmativo, la carta se coloca en su montón... y se procede de la misma manera con la siguiente carta descubierta, hasta llegar a una que no se pueda colocar en su montón. Ten en cuenta que al ir descubriendo las cartas

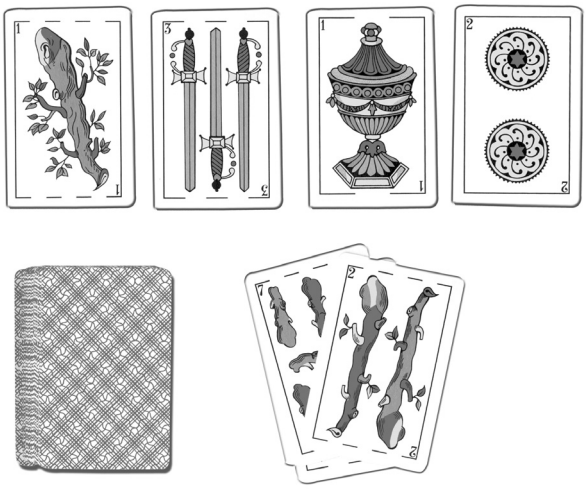


Figura 4.1: Instantánea de una partida de solitario

del mazo se irá conformando el mazo de cartas descubiertas. Cada una de ellas se podrá intentar colocar en el montón que le corresponda cuando quede la primera del mazo de cartas descubiertas.

En la figura 4.1 se puede observar una instantánea del solitario en la que se han sacado dos cartas del mazo, el dos y el siete de bastos. El dos de bastos, al ser la primera carta del mazo de cartas descubiertas y al ser la siguiente a la que aparece en el montón del palo bastos, podría colocarse en su montón. Una vez colocado el dos de bastos se comprobaría que el siete de bastos no se puede colocar.

- Se toman otras dos cartas y se procede de la misma manera hasta agotar el mazo inicial.
- Una vez agotado el mazo, las cartas que han sido descubiertas y no han podido ser colocadas constituyen el nuevo mazo. Se continúa con el procedimiento de descubrir cartas de dos en dos e intentar colocar las cartas descubiertas en los montones correspondientes hasta finalizar el solitario.
- El solitario acaba cuando se han completado los cuatro montones (¡enhorabuena!) o bien cuando, dado un determinado mazo, tras haber acabado de descubrir todas sus cartas, se comprueba que no se ha podido colocar en su sitio ninguna de ellas (¡otra vez será!).

## 4.8 Descomposición en sumas 189

Había un 8 que quería ser un ocho.

—No te podrás dividir siendo un ocho —le dijo su padre—. No te empeñes, es mejor ser un 8 que un ocho.
—No lo entiendo —respondía el 8.
—Por ejemplo —le decía su padre con paciencia—, 4 es la mitad de 8, pero *cua* no es la mitad de ocho.
—¿Cómo que no?
—Como que no. *Cua* es la mitad de *cua*ndo, de *cua*rzo, de *cua*dra, pero no la mitad de ocho.

El pequeño 8 se quedó pensativo. Llevaba razón su padre. Los números tenían ventajas sobre las letras. [...]

Finalmente, el 8 le dijo a su padre:
—Quiero ser lo que soy, porque siendo lo que soy puedo ser otras cosas.
—¿Qué cosas?
—Un 8 puede ser un grupo de ocho unos: 1–1–1–1–1–1–1–1.
—O un grupo de cuatro doses: 2–2–2–2.
—O un grupo de dos cuatros: 4–4.
—O un 5 y un 3 —dijo el padre.
Y se pasaron la noche haciendo cuentas.

*Numeros pares, impares e idiotas*, Juan José Millas y Antonio Fraguas "Forges"

Escribe un programa que, dado un número natural, escriba todas sus posibles descomposiciones en sumas de números naturales.

***Ejemplo*** Todas las posibles sumas, salvo el orden de los sumandos, de números naturales que dan como resultado 6 son:

$$\begin{array}{llllll} 1+1+1+1+1+1 & 2+1+1+1+1 & 3+1+1+1 & 4+1+1 & 5+1 & 6 \\ & 2+2+1+1 & 3+2+1 & 4+2 & & \\ & 2+2+2 & 3+3 & & & \end{array}$$

(Véase la pista 4.8a.)

***Bibliografía*** La descomposición de un número en sumas está dentro de la teoría de particiones y fue estudiada, entre otros, por Srinivasa Aiyangar Ramanujan (1887–1920).

## 49 El juego del master mind

En el juego del *master mind* intervienen dos jugadores. El primero de ellos elige en secreto un número formado por cuatro dígitos distintos. El otro trata de adivinarlo mediante tanteos sucesivos. Cada tanteo consiste también en un número de cuatro cifras, que el primer jugador examina, diciendo cuántos dígitos coinciden y están en su posición correcta (*aciertos*), y cuántos se han acertado pero no están en su posición (*semiaciertos*).

***Permutación*** Escribe un subprograma que produzca, aleatoriamente, un número formado por cuatro elementos del conjunto $\{1, 2, \ldots, 9\}$.

***Posición*** Define una función que informe de la posición que tiene un dígito en una secuencia, o cero si no está presente.

***Aciertos*** Escribe un subprograma que, dadas dos secuencias de dígitos, halle el número de aciertos y semiaciertos de la segunda (el tanteo) con respecto a la primera (la que se trata de adivinar).

***Juego*** Desarrolla un programa que integre los apartados anteriores y represente al primer jugador. Tendremos así un programa contra el que podremos jugar. Eso sí, la tarea más difícil, la de adivinar la combinación, la realizaremos nosotros.

***Bibliografía*** En [Knu77], Donald E. Knuth estudió una estrategia para adivinar la secuencia escondida en este juego. In 1994, Kenji Koyama y Tony W. Lai demostraron que dicha estrategia tiene el mejor comportamiento esperado posible [KL93]. Tampoco en Internet podía faltar información sobre este juego y sus muchas variantes; tómese este punto de partida como ejemplo: `http://www.tnelson.demon.co.uk/mastermind/`.

En breve, Disney y los demás laboratorios de la felicidad habrán acabado con nuestra imaginación. Para entonces, miraremos los fotogramas de la figura 4.2 y no apreciaremos los brillos de los dos robots ni oiremos el sonido tecnológico de una compuerta que se cierra; veremos simplemente un montón de rayas inconexas.

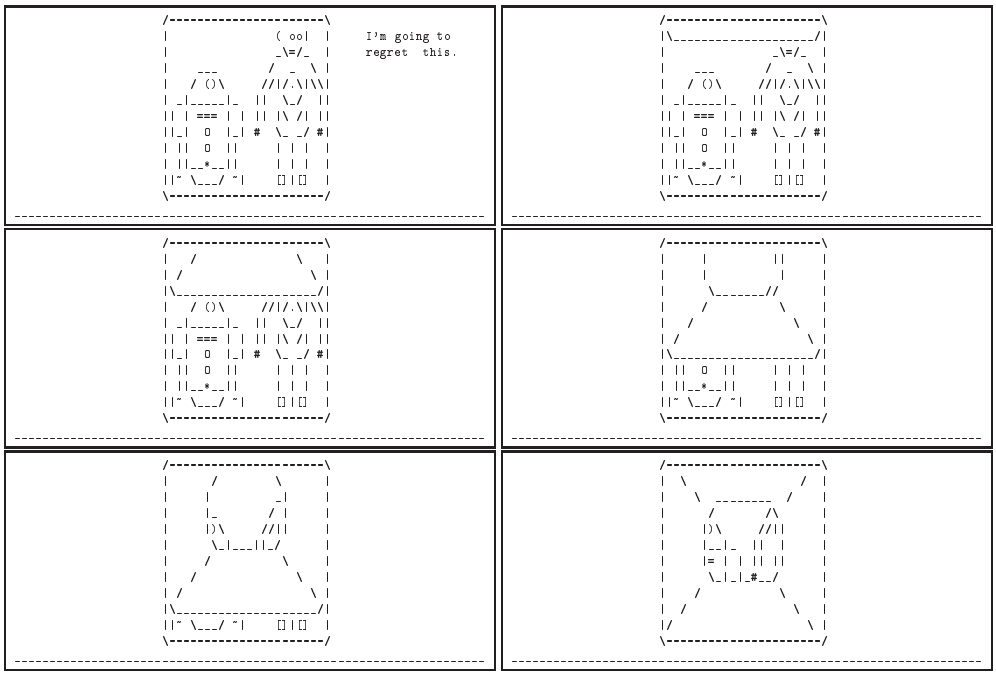



Figura 4.2: Fotogramas de una película *Star Wars I* en ASCII

Pero todavía no está todo perdido, porque Simon Jansen ha creado AnimASCIIón (`http://www.asciimation.co.nz`), el avanzado estándar informático que salvará la imaginación de pequeños y mayores. Ya no tendremos que subyugar nuestro intelecto a las ñoñerías multimillonarias de Disney, porque con una mínima cantidad de recursos podremos crear nuestras propias películas. Basta un editor de texto y ajustarnos a las reglas que explicaremos en breve. Y para verlas..., bueno, para verlas..., seguro que no tardas mucho en hacer un programa que muestre por pantalla una AnimASCIIón al ritmo adecuado.

Una película en el formato AnimASCIIón es un simple fichero de texto. Cada grupo de 14 líneas define un fotograma; la primera línea de un grupo contiene un número que indica cuántas veces se debe repetir este fotograma; las otras 13 conforman la imagen. Las líneas de un fotograma tienen como mucho 67 caracteres; por tanto, las imágenes ocupan $13 \times 67$ caracteres.

Un visor correcto de AnimASCIIones debe mostrar 15 fotogramas por segundo; o lo que es lo mismo, cada imagen debe permanecer durante $1/15$ de segundo. Por supuesto, hay que tener en cuenta el número de veces que se debe repetir un fotograma; si este número es $k$, será mejor conservarlo durante $k/15$ de segundo que borrarlo y volverlo a mostrar $k$ veces. (Véase la pista 4.10a.)

***Un poco de historia*** Lo que aquí se cuenta es una historia basada en hechos reales. Los nombres de los personajes (Simon Jansen) y los lugares (`http://www.asciimation.co.nz`) no se han cambiado. Simplemente, se ha traducido Asciimation por Animasciión. En el sitio de Asciimation en Internet podrás

leer por qué Simon se enfrascó en la tarea de crear, con sólo la ayuda de un editor de texto, una versión en caracteres ASCII de *Star Wars I (La guerra de las galaxias)*, y cómo lleva a cabo su minucioso proceso de edición.

## 4.11 Criterios de divisibilidad



Desde la antigüedad se conocen reglas que permiten saber si un número es divisible por otro simplemente con la *inspección* de las cifras que componen dicho número. Quizás la más famosa es la *regla del 9*, que dice que un número es múltiplo de 9 si y sólo si la suma de sus dígitos es también múltiplo de 9. De esta forma podemos saber con facilidad y rapidez si un número grande es múltiplo de 9; por ejemplo, 8294527125 sí lo es, ya que la suma de sus cifras es $8+2+9+4+5+2+7+1+2+5=45$ y $4+5=9$, que obviamente es múltiplo de 9.

Blaise Pascal (1623–1662) se interesó por estos temas y encontró una generalización a la regla del 9, que sólo requiere sumar las cifras que componen el número; en el caso general, para comprobar si un número $n$ es múltiplo de $k$, no basta con sumar las cifras que componen el número $n$, sino que antes hemos de multiplicar dichas cifras por coeficientes apropiados, que previamente se han calculado según el número $k$. Por ejemplo, los coeficientes para comprobar si cierto número es múltiplo de 7 son los siguientes:

| 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|

(Esta *tabla de módulos* se calcula de forma sencilla; posteriormente se describirá cómo.)

Ahora, si queremos saber si un número (pongamos 21756) es múltiplo de 7, basta con que hagamos la siguiente operación:

| 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|
| | × | × | × | × | × |
| | 2 | 1 | 7 | 5 | 6 |
| | ‖ | ‖ | ‖ | ‖ | ‖ |
| | 8 | 6 | 14 | 15 | 6 |

El número 21756 es múltiplo de 7 si y sólo si el número $8+6+14+15+6=49$ lo es. Averigüémoslo: para saber si 49 es múltiplo de 7 repetimos el proceso,

| 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|
| | | | | × | × |
| | | | | 4 | 9 |
| | | | | ‖ | ‖ |
| | | | | 12 | 9 |

y resulta que ahora necesitamos saber si 21 lo es: damos un último paso y obtenemos,

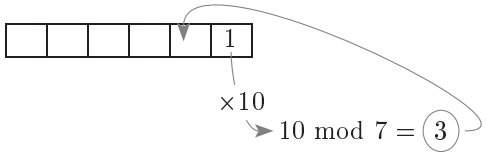
| 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|
| | | | | × | × |
| | | | | 2 | 1 |
| | | | | ‖ | ‖ |
| | | | | 6 | 1 |

que suma 7, que es múltiplo de 7 obviamente. Por tanto, 21756 es múltiplo de 7. Si, por el contrario, al final del proceso hubiésemos obtenido un número inferior a 10 diferente de 7, sabríamos que el número inicial no era múltiplo de 7.

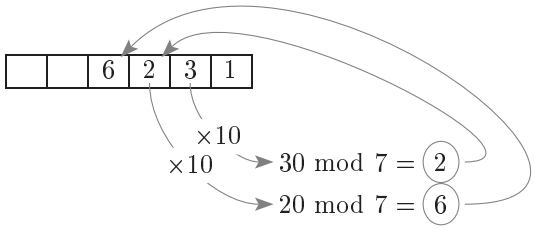
Así pues, teniendo la tabla de módulos del 7 es fácil comprobar si un número es o no múltiplo de 7. Veamos ahora cómo calcularlas para los números mayores o iguales a 2.

Sea cual sea la tabla que calculemos ponemos en el extremo derecho un 1. El mecanismo general consiste en tomar la última entrada multiplicarla por diez y calcular el módulo de la división por el número del que deseamos calcular la tabla; este resultado pasa a ser una nueva entrada.

Supongamos que queremos calcular la tabla de módulos del número 7 que hemos utilizado antes. Comenzamos poniendo 1 en el extremo derecho. Para calcular el siguiente número multiplicamos el 1 por 10 y ponemos el resto del producto dividido entre 7 en la segunda posición de la tabla:




Repetimos el mismo proceso comenzando con el último número de la tabla, es decir, lo multiplicamos por diez y calculamos el resto de dividir por 7; este número ocupa la siguiente casilla:




Repitiendo el proceso obtenemos la tabla siguiente:

| 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|

Se termina de calcular la tabla de módulos cuando se obtiene el primer factor repetido. En el caso de la tabla del 7, si realizamos el cálculo de la siguiente entrada para el 5, obtenemos que $50 \bmod 7 = 1$ que es la primera entrada de la tabla. No es necesario continuar ya que el proceso entraría en un ciclo y se repetiría la secuencia de números que ya se tiene.

***Tablas de módulos*** Escribe un programa que, dado un número entre 2 y 9, calcule la tabla de módulos de dicho número. Sólo tienen que calcularse las entradas de la tabla necesarias.

***Divisibilidad*** Escribe un programa que, dado un número $n$ presumiblemente grande, y un número $m$ entre 2 y 9, calcule si dicho $n$ es múltiplo de $m$. (Véase la pista 4.11a.)

***Otras bases*** El enunciado describe el método para números expresados en base 10, pero también funciona para otras bases. Estudia esta posibilidad.

***Bibliografía*** El libro en el que Pascal publicó estos resultados es [Pas65]. El libro [Cha99] es mucho más fácil de conseguir y en él se puede encontrar el extracto de la obra anterior de Pascal en la que se describe el mecanismo que presenta el ejercicio.

***Un poco de historia*** Etienne Pascal, padre de Blaise Pascal, fue matemático. Etienne mantuvo a Blaise apartado de los libros de matemáticas, ya que prefería que su hijo se dedicase a otras cosas. A pesar de eso, el talento de Blaise para la geometría era tal que hizo recapacitar a su padre y, cuando sólo tenía doce años de edad, lo inició en las matemáticas con los *Elementos* de Euclides [Euc91, Euc94, Euc96]. A los 17 años de edad Blaise escribió y publicó *Essay pour le coniques*.

El cálculo algorítmico siempre interesó a Blaise Pascal. A los 18 años diseñó y construyó una de las primeras calculadoras mecánicas, *La Pascalina*, que sumaba y restaba.

## 4 12 Otra variación sobre los números primos 190

En el ejercicio 3.8 se vio que, para comprobar si un número entero positivo $n$ es primo, basta con examinar si tiene algún divisor entre 2 y $\lfloor\sqrt{n}\rfloor$. En realidad, basta con probar con los números entre 2 y $\lfloor\sqrt{n}\rfloor$ que sean primos. Ahora bien, averiguar si un posible divisor $d$ es primo es más costoso que ver directamente si divide a $n$ o no. En cambio, lo que sí puede interesar es tener archivados los primeros primos, y repasar esa lista en busca de divisores, en vez de avanzar sobre los naturales.

Y de eso trata este ejercicio.

***Generación de una tabla de primos*** En este apartado se propone escribir un programa que genere, por cualquier método, un archivo de enteros `primos.dat`, con los primos entre 2 y 100. Como se trata de generar sólo unos pocos y más bien pequeños, se puede emplear cualquier método sencillo. (Véase, por ejemplo, el ejercicio 3.8).

***Consulta de la tabla*** La tabla generada en el apartado anterior tiene un primer uso: comprobar si es primo un número entre 2 y 100 consiste ahora en ver si está en ella.

Desarrolla una función `esPrimoPeq` que efectúe esa comprobación con números entre 2 y 100. (Véase la pista 4.12a.)

***Generación de primos*** La tabla anterior es útil también para verificar si un número $n$ entre 101 y 10000 es primo, ya que contiene todos los posibles divisores (¡primos!) entre 2 y $\lfloor\sqrt{n}\rfloor$.

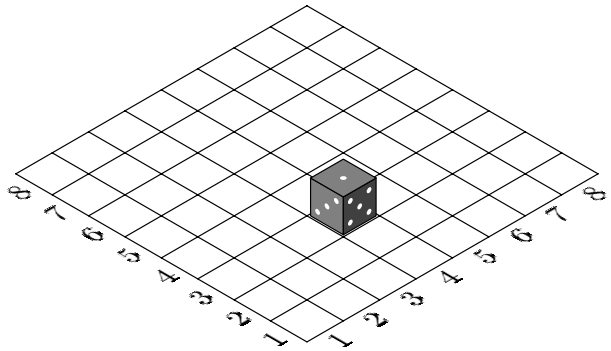
Desarrolla una función `esPrimoGr` que efectúe esa comprobación con números entre 101 y 10000.

Como se puede imaginar, generando esos primos en una segunda tabla, también sería posible comprobar la primalidad de números mucho mayores...

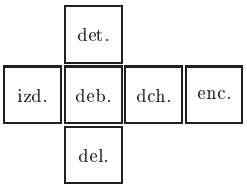
***Bibliografía*** La referencia [Pom83] es un buen punto de partida para ampliar información sobre este tema. Ten cuidado: el asunto es fascinante y la bibliografía interminable.

## 4 13 Un par de juegos con dados rodantes

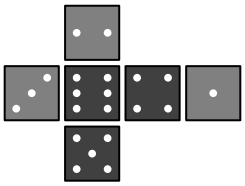
Te proponemos desarrollar programas para jugar con dados que se mueven, rodando, en un tablero cuadriculado, como el de ajedrez, aunque de tamaño arbitrario:




Las piezas del juego son dados de seis caras (delante, detrás, encima, debajo, izquierda, derecha), que se pueden mostrar desarrollando el cubo para distinguir cada una de ellas, así:




Para indicar la *orientación* del dado en un momento de la partida, basta con decir los valores que tiene cada una de dichas caras; por ejemplo, en el dado del tablero anterior, son 2, 5, 1, 6, 3, 4 respectivamente:



Además, su *posición* en un tablero de $N \times N$ queda registrada en un par de números del conjunto $\{1, \ldots, N\}$. Por ejemplo, el dado de la figura de arriba está en la posición $(4, 3)$.

***El estado de la partida*** Define los tipos de datos adecuados para reflejar el estado de un dado (orientación y posición) en cierto momento de la partida. Escribe también subprogramas para leer de la entrada estándar la posición y orientación inicial de un dado.

***Las operaciones básicas*** En los juegos que se describirán posteriormente, va a ser posible efectuar dos tipos de movimientos con un dado:

- **Ruedo,** indicando una dirección, para que el dado *ruede* siguiendo la misma tantos pasos como indique su cara superior. Este movimiento cambia la posición y la orientación del dado.
- **Giro,** indicando una dirección y un número, para que el dado cambie su disposición, girando apropiadamente sobre sí mismo pero sin moverse de la casilla que ocupa.

Se pide ahora definir un tipo de datos para representar un movimiento, indicando su tipo (ruedo o giro), su dirección y, en su caso, su magnitud. Será también útil un subprograma para leer de la entrada estándar un movimiento de un jugador.

Como ejemplo, en la figura 4.3 se detalla el cambio de disposición de un dado que rueda exactamente una casilla en cada uno de los cuatro sentidos posibles.

***Los movimientos*** Desarrolla subprogramas apropiados para los siguientes movimientos:

- Efectuar un ruedo sobre un dado.
- Efectuar un giro sobre un dado.

***Juegos*** Desarrolla los siguientes juegos, utilizando para ello los subprogramas anteriores y, tal vez, otros necesarios:

- Un solitario, en que un solo dado efectúa movimientos sucesivos hasta alcanzar una de las esquinas del tablero.
- Un juego bipersonal, con dos dados, en que los jugadores tratan de *comerse* mutuamente, o sea, lograr un movimiento que alcance la posición del enemigo.

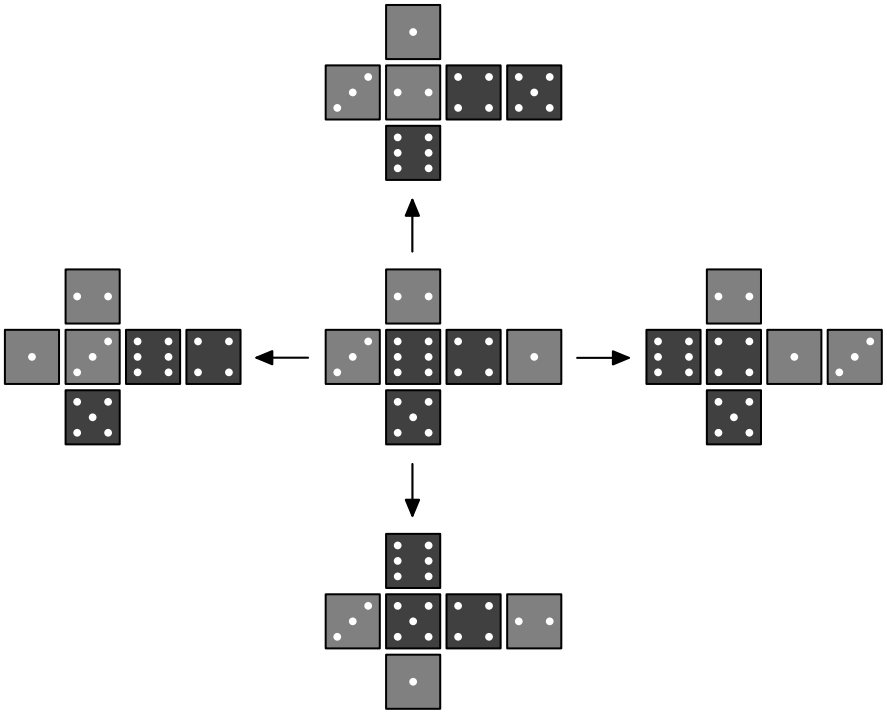


Figura 4.3: Cambio de disposición de un dado que rueda

El atractivo de estos juegos se puede aumentar incluyendo una salida clara, que enseñe el estado del tablero en cada momento del juego. Por otra parte, ambos juegos se pueden dejar en manos del azar, generando los movimientos de los jugadores aleatoriamente.

## 4.14 El juego de sumar quince

En el juego de *sumar quince* participan dos jugadores sobre el siguiente tablero:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

Los jugadores juegan por turno y en cada turno el jugador selecciona un número que esté libre. Gana el primero que consiga sumar quince, utilizando *tres* de los números que tiene marcados.

***Ejemplo*** Veamos una partida. Tenemos dos jugadores, A y B. En cada turno el jugador que le toca marca uno de los números con su nombre. Comienza el jugador A y decide marcar el 5; el jugador B, en su turno, marca el 3:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| | | B | | A | | | | |

En la siguiente ronda el jugador A decide marcar el 2 y el B elige el 8 para evitar que A consiga 15 en la siguiente ronda:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| | A | B | | A | | | B | |

Entonces A marca el número 4 y B se da cuenta de que está a punto de perder la partida. En efecto, tras marcar A el número 4,

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
|  | A | B | A | A |  |  | B |  |

el jugador A puede hacer quince sumando $4+5+6$ y también $4+9+2$. Entonces, el jugador B, en su turno, sólo podrá marcar un número (el 9 o el 6) y no podrá evitar que A gane.

***Juego de sumar quince*** Realiza un programa que permita a dos jugadores humanos jugar al sumar quince. El programa debe asignar el turno a cada jugador y comprobar, en cada jugada, si el jugador ha conseguido sumar quince utilizando tres de los números que tiene marcados.

***Bibliografía*** La idea para este ejercicio está sacada del libro [Gar95]. Aunque no parezca claro a primera vista, existe una estrategia para poder jugar con más posibilidades de ganar. El siguiente dibujo es un cuadrado mágico de dimensión 3 (véase el ejercicio 6.9).

| 4 | 9 | 2 |
|---|---|---|
| 3 | 5 | 7 |
| 8 | 1 | 6 |

En dicho cuadrado las filas, las columnas y las diagonales suman 15. ¿Sabes jugar a las tres en raya?

## 4.15 Movimiento planetario

190 203

En este ejercicio simularemos el movimiento de planetas que se atraen mutuamente siguiendo las leyes gravitatorias clásicas de Isaac Newton (1642–1727). Según estas leyes la fuerza con que se atraen dos cuerpos es directamente proporcional al producto de sus masas e inversamente proporcional al cuadrado de su distancia,

$$F = G\frac{m_1 \cdot m_2}{d^2}$$

donde $G$ es la constante gravitatoria universal, $m_1$ y $m_2$ las masas respectivas y $d$ es la distancia entre los objetos. En lo que concierne a este ejercicio no será demasiado importante el valor de $G$ ni las medidas en las que se expresan los datos; se ajustarán simplemente para obtener un resultado estético adecuado en la simulación.

Haremos una simulación en el plano, por lo que un planeta tendrá, en cada momento, una posición $(p_x, p_y)$ y una velocidad $(v_x, v_y)$. Supondremos que en cada intervalo de tiempo suficientemente pequeño, que denominaremos $\Delta t$, la fuerza que actúa sobre cada planeta será constante. Esa fuerza se calcula según la fórmula anterior, y como estamos en el plano tendrá también dos componentes $(f_x, f_y)$. Según las leyes de Newton tenemos que

$$F = m \cdot a$$

por lo que la aceleración a la que está sometido cada cuerpo será de la forma $(a_x, a_y) = (f_x/m, f_y/m)$. Aplicamos entonces las fórmulas

$$E = E_0 + V_0 \cdot \Delta t + a \cdot \Delta t^2 \quad \text{y} \quad V = V_0 + a \cdot \Delta t$$

para actualizar la posición y velocidad de cada objeto.

- Da una definición de tipos adecuada para poder representar la configuración en cada instante de un planeta.
- Da una definición adecuada para las listas de planetas.

- Realiza el programa propiamente dicho de simulación. Habrá que definir dos constantes cuyo valor dependerá esencialmente de la configuración de la máquina donde se vaya a ejecutar. En primer lugar hay que expresar en C++ la *constante gravitatoria*. Si es demasiado grande, los planetas se acelerarán demasiado rápido y los perderemos de vista en seguida; si es demasiado pequeña, los planetas se moverán sin interactuar unos con otros. Habrá que definir una segunda constante para representar $\Delta t$, cuanto más pequeña sea esta constante más precisos serán nuestros cómputos pero también más lentos, por tanto habrá que buscar un equilibrio entre ambos conceptos en principio opuestos. (Véase la pista 4.15a.)

## 4.16 Códigos para la corrección de errores


¿Has pensado alguna vez en cómo es posible que tu lector de discos compactos reproduzca el sonido con tanta exactitud a pesar del polvo, las huellas y los rayajos que habitualmente hay en un disco compacto? El secreto radica en que los discos compactos están grabados utilizando un *código corrector de errores*, que el lector de discos utiliza para superar, en la medida de lo posible, los fallos de lectura.

Los códigos correctores de errores se utilizan cuando se transmite información a través de un canal de comunicación *ruidoso*, es decir, un canal que puede producir errores, como son los soportes de almacenamiento magnético, en transmisión de datos por satélite, en transacciones bancarias... Añadiendo un poco de redundancia a la información transmitida, se puede detectar y, a menudo, corregir un error producido en la transmisión.

Para entender el funcionamiento básico de los códigos de corrección de errores, pongamos un ejemplo real: con gran esfuerzo se ha conseguido situar un robot sobre la superficie de Marte (ver figura 4.4). Dicho robot puede moverse obedeciendo a cuatro órdenes diferentes de movimiento, una por cada dirección: norte, sur, este y oeste. Supongamos que tenemos el siguiente código para enviar las órdenes al robot:

| Orden de movimiento | Código |
|---|---|
| norte | 0 0 |
| este | 0 1 |
| oeste | 1 0 |
| sur | 1 1 |

Si se produce una interferencia al transmitir las órdenes de desplazamiento al robot, dicha interferencia puede modificar el código enviado. Si el código consta de dos bits, cada uno de ellos puede sufrir un cambio durante la transmisión. Pongamos por caso que queremos enviar la orden de ir hacia el norte, 00, y se produce sólo un error; el mensaje recibido puede ser 01 o 10. Los errores en el código que estamos considerando son muy graves, ya que un error de transmisión implica que el robot se mueva en una dirección no deseada con los problemas que esto puede acarrear.

Una forma de solucionar este problema es enviar órdenes que *no se parezcan* tanto entre sí, como ocurría con las anteriores. Supongamos que tenemos el siguiente código:

| Orden de movimiento | Código |
|---|---|
| norte | 0 0 0 |
| este | 0 1 1 |
| oeste | 1 1 0 |
| sur | 1 0 1 |

Enviar estos mensajes es algo más costoso que en el caso anterior ya que cada orden consta de tres señales, en lugar de dos. Pero este código tiene una gran ventaja con respecto al anterior: si se produce

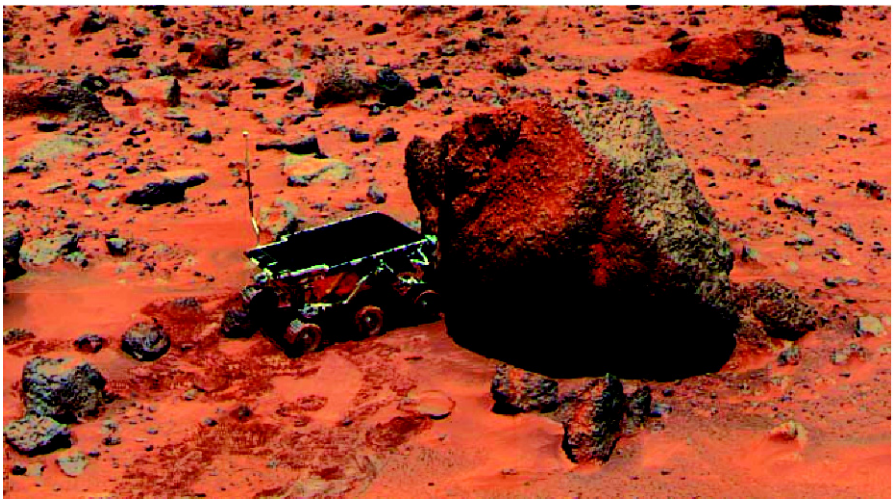


Figura 4.4: Robot enviado por la NASA a Marte

un único error, el mensaje recibido no corresponde a ninguno de los mensajes posibles, y por tanto, puede *detectarse* que la orden que ha llegado es errónea.

En efecto, supongamos que enviamos la orden de ir hacia el este, 011. Si se produce un único error en la transmisión, sólo uno de los bits del mensaje cambia y podemos recibir el mensaje 111, 001 o 010, según sea el bit alterado el primero, el segundo o el tercero respectivamente. En cualquiera de los casos, el mensaje recibido no coincide con ninguna de las órdenes y por tanto el robot puede saber que la orden que le ha llegado es defectuosa y no realizar ningún movimiento.

Este tipo de códigos que *detectan* errores son útiles en multitud de ocasiones, pero no en el ejemplo que estamos considerando. Si el robot detecta una orden defectuosa debería informar de ello y esperar a recibir una nueva orden de movimiento. Teniendo en cuenta lo que tardan los mensajes en recorrer la distancia de la Tierra a Marte (¡12 minutos, sólo de ida, a la velocidad de la luz, en su acercamiento máximo!), parece inadecuado depender de una comunicación bidireccional.

Pero si los mensajes enviados se *diferencian mucho* entre sí, quizás podamos *restaurarlos* aunque se haya producido algún error. Consideremos el siguiente código:

| Orden de movimiento | Código |
|---|---|
| norte | 0 0 0 0 0 |
| este | 0 1 1 0 1 |
| oeste | 1 0 1 1 0 |
| sur | 1 1 0 1 1 |

Si suponemos que se produce como mucho un error en la transmisión de cada mensaje, entonces el mensaje recibido sigue estando *próximo* al mensaje enviado y por tanto podemos reconocer el mensaje original. Supongamos que el mensaje enviado al robot es moverse hacia el norte, 00000. Si suponemos que se produce un único error, entonces el robot puede recibir alguno de los mensajes 10000, 01000, 00100, 00010 o 00001. Cada uno de estos mensajes está más *próximo* al original que a cualquier otro de los códigos y por tanto el robot puede entender que la orden enviada fue ir al norte.

Por supuesto hay que definir formalmente qué se entiende por *próximo*: en teoría de códigos se suele utilizar la *distancia de Hamming*...

### 4.16.1 Distancia de Hamming

Dados dos vectores, $v_1$ y $v_2$, se define la *distancia de Hamming*, $d(v_1, v_2)$, como el número de posiciones en que difieren los vectores $v_1$ y $v_2$. Por ejemplo, si $v_1 = 00001$ y $v_2 = 10100$, $d(v_1, v_2) = 3$ ya que existen tres posiciones (la primera, la tercera y la quinta) en las que los vectores tienen diferente valor.

Escribe una función que, dadas dos palabras de igual longitud pertenecientes a un código, indique cuál es la distancia de Hamming entre ellas.

### 4.16.2 Distancia mínima de un código

Si $C = \{p_1, \ldots, p_n\}$ es un código con $n$ palabras, se define la *mínima distancia del código* $C$ así:

$$d(C) = \min\{d(p_i, p_j) \mid p_i, p_j \in C, i \neq j\}$$

Esta distancia mínima informa de los errores que podremos detectar y corregir con el código que estamos considerando. Si $d(C) \geq 2k + 1$, entonces podemos detectar si se han producido hasta $2k$ errores; y podremos corregir hasta $k$ errores.

Escribe un subprograma que, dado un código (es decir, las palabras que lo componen), calcule la mínima distancia de dicho código e informe de la posibilidad de detección y corrección de errores de dicho código.

### 4.16.3 Simulación de ruido

Para simular un canal de transmisión de datos ruidoso se han de tener en cuenta ciertas hipótesis:

- Todos los bits de un mensaje tienen la misma probabilidad de ser recibidos con error.
- La probabilidad $p$ con la que cada bit transmitido se recibe con error ha de ser menor que $\frac{1}{2}$.

Escribe un subprograma que, fijada una cierta probabilidad de fallo, $p$, y teniendo en cuenta las condiciones anteriores, simule la transmisión de datos a través de un canal que puede producir fallos. (Véase la pista 4.16a.)

### 4.16.4 Detección de errores

Escribe un subprograma que, dado un código $C$, simule la transmisión de datos a través de un canal ruidoso e informe de la detección de errores.

### 4.16.5 Corrección de errores

Escribe un subprograma que, dado un código $C$, simule la transmisión de datos a través de un canal ruidoso y, si el código lo permite, corrija los posibles errores de transmisión que se produzcan.

### 4.16.6 Simulación global

Escribe un programa que permita hacer una simulación total del uso de un código corrector de errores. (Véase la pista 4.16b.)

### 4.16.7 Pruébalo primero

En general, los programas propuestos pueden hacerse de forma independiente al código que se utilice. A continuación te damos algunos códigos para utilizarlos en tu programa:

$$C_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad C_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix} \quad C_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$C_1$ y $C_2$ permiten codificar cuatro palabras. El primero sólo detecta un error y el segundo detecta dos errores y corrige uno. $C_3$ es un código que permite enviar ocho palabras diferentes y que utiliza para ello siete bits.

### 4.16.8 Bibliografía

En el ejercicio, por motivos de claridad y sencillez, hemos asumido ciertas simplificaciones. Por ejemplo hemos considerado únicamente códigos sobre el alfabeto binario 0 y 1, pero las mismas técnicas de codificación se pueden utilizar sobre otros alfabetos. El código ISBN de los libros utiliza los números del 0 al 9 y la letra X. (Véase el ejercicio 2.18.)

En Internet, hay un buen sitio dedicado al funcionamiento de los ordenadores personales que explica los mecanismos de codificación de datos utilizados en diversos componentes: discos compactos (`http://www.pcguide.com/ref/cd/mediaECC-c.html`), disco duro (`http://www.pcguide.com/ref/hdd/geom/error_ECC.htm`) y memoria (`http://www.pcguide.com/ref/ram/err.htm`).

Libros para conocer más sobre la teoría de la codificación son [Hil99] y [HLL+92].

## 4.17 Ajuste de imagen


Se pretende representar gráficamente la temperatura que hay en diversas partes de un objeto. Para ello, tras aplicar diversas fuentes de calor de distinta intensidad en algunas de las partes del objeto, se realiza la medición de la temperatura de cada una de ellas y su resultado se transcribe en una matriz. Cada elemento de la matriz almacenará la temperatura de una parte del objeto. Se ha valorado que la temperatura puede oscilar entre 30 y 99 grados centígrados (se redondea al entero más próximo). Una vez que se dispone de la matriz con la información de la temperatura, se desea que su contenido se represente gráficamente de forma que a cada intervalo de 10 grados se le asocie un nivel de intensidad de color gris distinto. Por ejemplo, a los valores de 30 a 39 grados les corresponderá el nivel de intensidad de gris más bajo, mientras que a los valores de 90 a 99 les corresponderá el gris más oscuro. Los que realizan el experimento creen necesario corregir posibles errores en la medición. Consideran que se ha producido un error en la medición si, dado un elemento de la matriz, su valor difiere en más de 10 unidades de los valores de cada uno de sus vecinos (ocho vecinos, cinco o tres según la posición del elemento en la matriz). La corrección consistirá en asignar a dicho elemento el valor resultante de calcular la media de los valores de sus vecinos redondeándola al entero más próximo. Ten en cuenta que el resultado de la corrección se debe registrar en otra matriz. En la figura 4.5 puedes ver un ejemplo de una matriz con temperaturas y la representación gráfica que le corresponde antes y después de la corrección.

***Generar temperaturas*** Escribe un subprograma que genere de forma aleatoria una matriz de $15 \times 15$ elementos con valores enteros en el conjunto $\{30, \ldots, 99\}$, equiprobablemente.

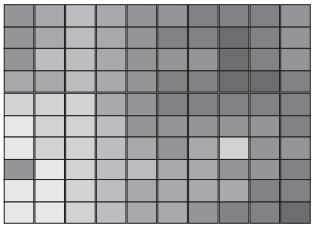
***Representación gráfica*** Escribe un subprograma que represente la matriz de temperaturas.

***Corrección de las temperaturas*** Escribe un subprograma que obtenga una nueva matriz como resultado de realizar la corrección de las temperaturas a la matriz obtenida de forma aleatoria.

***Idempotencia*** ¿Es *idempotente* este proceso de corrección? Una función $f$ es idempotente si $\forall x, f(x) = f(f(x))$, es decir, aplicarla una o más veces tiene el mismo efecto. En nuestro caso, que el proceso de corrección fuera idempotente querría decir que en su resultado no hay errores.

***Repetición de la corrección*** Si la respuesta a la pregunta del apartado anterior es negativa, escribe un procedimiento que aplique la corrección de errores repetidamente hasta que el resultado no tenga ningún error.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 72 | 63 | 57 | 68 | 73 | 78 | 80 | 85 | 81 | 76 |
| 75 | 65 | 56 | 69 | 75 | 85 | 89 | 90 | 86 | 75 |
| 70 | 59 | 55 | 64 | 73 | 79 | 70 | 94 | 85 | 78 |
| 64 | 63 | 50 | 65 | 76 | 85 | 88 | 94 | 90 | 87 |
| 45 | 53 | 49 | 63 | 75 | 83 | 84 | 89 | 88 | 86 |
| 39 | 49 | 49 | 59 | 70 | 76 | 74 | 78 | 74 | 80 |
| 37 | 43 | 48 | 59 | 69 | 71 | 69 | 40 | 70 | 75 |
| 70 | 42 | 45 | 55 | 58 | 60 | 68 | 75 | 78 | 80 |
| 35 | 39 | 43 | 52 | 62 | 64 | 65 | 67 | 86 | 85 |
| 32 | 36 | 40 | 50 | 60 | 69 | 74 | 85 | 88 | 90 |

≡ 

Tras la corrección:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 72 | 63 | 57 | 68 | 73 | 78 | 80 | 85 | 81 | 76 |
| 75 | 65 | 56 | 69 | 75 | 85 | 89 | 90 | 86 | 75 |
| 70 | 59 | 55 | 64 | 73 | 79 | 70 | 94 | 85 | 78 |
| 64 | 63 | 50 | 65 | 76 | 85 | 88 | 94 | 90 | 87 |
| 45 | 53 | 49 | 63 | 75 | 83 | 84 | 89 | 88 | 86 |
| 39 | 49 | 49 | 59 | 70 | 76 | 74 | 78 | 74 | 80 |
| 37 | 43 | 48 | 59 | 69 | 71 | 69 | 73 | 70 | 75 |
| 39 | 42 | 45 | 55 | 58 | 60 | 68 | 75 | 78 | 80 |
| 35 | 39 | 43 | 52 | 62 | 64 | 65 | 67 | 86 | 85 |
| 32 | 36 | 40 | 50 | 60 | 69 | 74 | 85 | 88 | 90 |

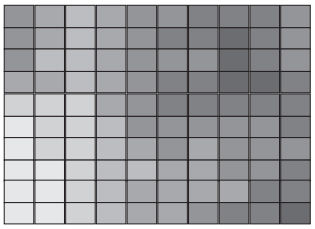
≡ 

Figura 4.5: Matriz de temperaturas con el resultado de su corrección y sus respectivas representaciones gráficas

## 4.18 Diferencias finitas

190

Un problema que se presenta a menudo en multitud de áreas (física, matemáticas, ciencias aplicadas, ingenierías, informática gráfica...) es encontrar la expresión general de una función de la que sólo se conoce el valor en unos pocos puntos.

Supongamos que se conocen los valores que toma una determinada función $f$ en algunos puntos, por ejemplo en 0, 1, 2, 3 y 4 (figura 4.6, gráfica izquierda) y se desea encontrar una expresión de la función para poder estimar el valor en otros puntos, por ejemplo en 3.25 o en $-2$ (figura 4.6, gráfica derecha).

Dados $n$ puntos en los que se conoce el valor de una función, el *método de las diferencias finitas* se utiliza para buscar un polinomio de grado menor que $n$ que pase por dichos puntos. Veámoslo. Sea

| $P_0^0$ | $P_1^0$ | $P_2^0$ | $\cdots$ | $P_{n-1}^0$ | $P_n^0$ |
|---|---|---|---|---|---|

la secuencia conocida de valores de $f(x)$. La primera secuencia de diferencias está formada por los valores $P_{i=1\ldots n}^1 = P_i^0 - P_{i-1}^0$, dando lugar a:

| $P_1^1 = P_1^0 - P_0^0$ | $P_2^1 = P_2^0 - P_1^0$ | $\cdots$ | $P_{n-1}^1 = P_{n-1}^0 - P_{n-2}^0$ | $P_n^1 = P_n^0 - P_{n-1}^0$ |
|---|---|---|---|---|

En general, calculamos las secuencias de diferencias con la fórmula $P_{i=k\ldots n}^k = P_i^{k-1} - P_{i-1}^{k-1}$, que da como resultado una tabla del tipo:

| $P_0^0$ | $P_1^0$ | $P_2^0$ | $\cdots$ | $P_{n-1}^0$ | $P_n^0$ |
|---|---|---|---|---|---|
| | $P_1^1 = P_1^0 - P_0^0$ | $P_2^1 = P_2^0 - P_1^0$ | $\cdots$ | $P_{n-1}^1 = P_{n-1}^0 - P_{n-2}^0$ | $P_n^1 = P_n^0 - P_{n-1}^0$ |
| | | $P_2^2 = P_2^1 - P_1^1$ | $\cdots$ | $P_{n-1}^2 = P_{n-1}^1 - P_{n-2}^1$ | $P_n^2 = P_n^1 - P_{n-1}^1$ |
| | | | $\vdots$ | $\vdots$ | $\vdots$ |
| | | | | $P_{n-1}^{n-1} = P_{n-1}^{n-2} - P_{n-2}^{n-2}$ | $P_n^{n-1} = P_n^{n-2} - P_{n-1}^{n-2}$ |
| | | | | | $P_n^n = P_n^{n-1} - P_{n-1}^{n-1}$ |

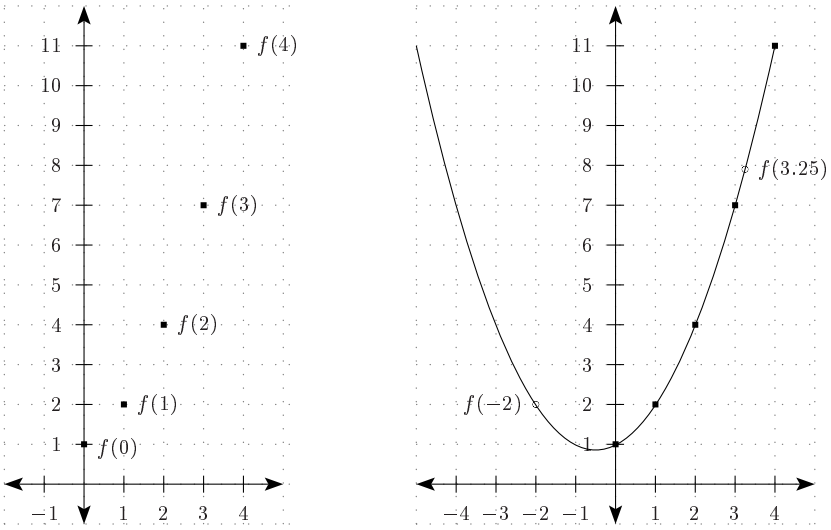



Figura 4.6: Puntos conocidos de una función (izquierda) y su aproximación (derecha)

Para el ejemplo que muestra la gráfica de la figura 4.6, teniendo en cuenta los valores conocidos de $f(x)$, la tabla de diferencias quedaría así:

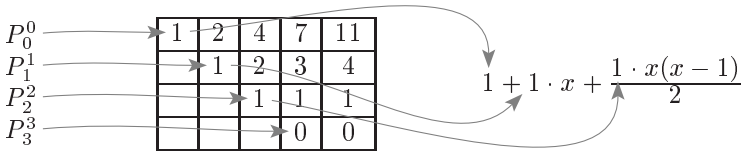
| | | | | |
|---|---|---|---|---|
| 1 | 2 | 4 | 7 | 11 |
| | 1 | 2 | 3 | 4 |
| | | 1 | 1 | 1 |
| | | | 0 | 0 |
| | | | | 0 |

Observa que, si una sucesión es constante, las restantes diferencias son cero.

A partir de la tabla de diferencias finitas se puede construir un polinomio que tiene el mismo valor que la función en los puntos conocidos. Si los puntos en que se ha calculado la función son los naturales 0, 1, 2, 3... se puede utilizar la fórmula de Isaac Newton (1643–1727):

$$P_0^0 + P_1^1 \cdot x + \frac{P_2^2 \cdot x(x-1)}{2} + \frac{P_3^3 \cdot x(x-1)(x-2)}{2 \cdot 3} + \frac{P_4^4 \cdot x(x-1)(x-2)(x-3)}{2 \cdot 3 \cdot 4} + \cdots$$

Continuando con nuestro ejemplo, si consideramos la tabla de diferencias para los valores 1, 2, 4, 7, 11, que hemos calculado anteriormente, los números $P_i^i$ son los primeros de cada fila. Todos los $P_i^i$ que son ceros hacen que el término correspondiente de la fórmula de Newton se anule y por tanto no necesitemos considerarlos. Sustituyendo en la fórmula de Newton los valores de la tabla, obtenemos el polinomio resultante:




***Construcción de tablas de diferencias*** Escribe un subprograma que, dados $n$ valores, calcule la tabla de diferencias correspondiente. No derroches espacio de memoria con variables que no utilizas. (Véase la pista 4.18a.)

***Construcción del polinomio de Newton*** Escribe un subprograma que, dados $n$ valores que supuestamente son los resultados de evaluar una función $f(x)$ en los $n$ primeros naturales, utilice la fórmula de Newton para construir el polinomio de aproximación.

***Ejemplo*** En [Gar94] aparece un bonito ejemplo al que aplicar el método de diferencias finitas: ¿Cuál es el número máximo de triángulos que pueden formarse con $n$ líneas rectas?

Para 0, 1 y 2 rectas la solución es muy fácil: no se puede formar ningún triángulo. Con 3 rectas obviamente se puede formar un triángulo. Para 4 rectas es fácil encontar 4 triángulos. Para 5 rectas la cosa se complica y puedes ver en la figura 4.7 una configuración de rectas que forman 10 triángulos. La

| aci | agd | ajf | bdj | bhe | bgf | cef | chg | dhi | eij |
|---|---|---|---|---|---|---|---|---|---|

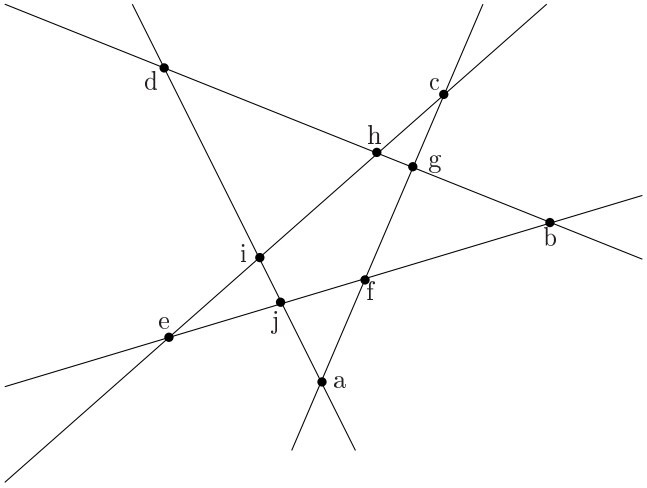



Figura 4.7: Diez triángulos, definidos por sus vértices, formados por la intersección de cinco rectas

solución para 6 parece difícil. Utilicemos el método que describe el ejercicio. La tabla de diferencias para dichos valores es la siguiente,

| 0 | 0 | 0 | 1 | 4 | 10 |
|---|---|---|---|---|---|
| | 0 | 0 | 1 | 3 | 6 |
| | | 0 | 1 | 2 | 3 |
| | | | 1 | 1 | 1 |

y por tanto, utilizando la fórmula de Newton, el polinomio que define dicha proporción es:

$$\frac{1 \cdot x(x-1)(x-2)}{6}$$

El método que hemos descrito en el ejercicio proporciona un polimonio de aproximación a una función de la que sólo se conoce el valor en unos pocos puntos. Esta aproximación puede coincidir exactamente con la función que se buscaba, pero en general, no tiene por qué ser así. Si la función es polinómica de grado $n$, la aproximación lograda por interpolación de $n+1$ puntos distintos aproxima exactamente la función.

***Bibliografía*** El origen del método de diferencias finitas se debe a Brook Taylor (1685–1731) [Tay15]. En [Cha99] se dedica un capítulo a la historia de los algoritmos de interpolación.

## 4.19 Búsqueda de la persona famosa en una reunión

190

Decimos que una persona que asiste a una reunión es *famosa* si es conocida por todos los asistentes a la reunión pero no conoce a nadie. Si en una reunión hay $k$ personas y sus nombres son $p_1, \ldots, p_k$, el reto que te planteamos es sencillo: encuentra a la persona que es famosa en la reunión, si es que existe. La única pregunta que puedes realizar, tantas veces como quieras a cada persona $p_i$ asistente a la reunión es: "¿conoces a $p_j$?" para cualquier $j$. ¿Te animas?

***Representación de los datos*** Piensa en cómo podríamos representar los datos del problema de una forma adecuada.

***Búsqueda de un famoso*** Escribe un programa que permita encontrar a una persona famosa en un grupo de personas, si es que existe.

***Rapidez en la búsqueda*** Existe una solución relativamente sencilla al apartado anterior, pero dicha solución hace muchas más preguntas a los asistentes de las que son necesarias y, por tanto, se emplea más tiempo del que realmente hace falta en encontar a la persona famosa. Busca una solución, si es que no la has dado ya, que saque el máximo partido a la información que se obtiene de la respuesta a cada pregunta y minimice, por consiguiente, el número de preguntas que se realizan. (Véase la pista 4.19a.)

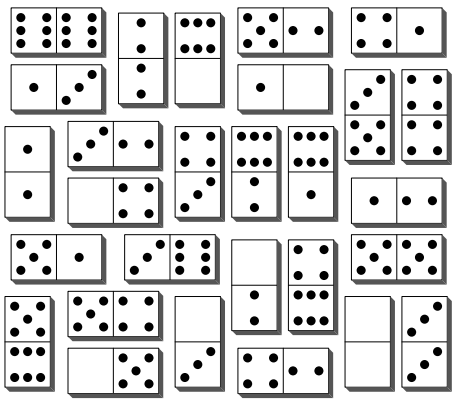
***Bibliografía*** Para encontrar una solución al ejercicio se necesita modelar una relación (la de conocimiento) entre personas. Se denomina *grafo* a un modelo para expresar relaciones. El estudio de grafos tiene aplicaciones en áreas muy diversas: estadística, telecomunicaciones, neurología, relaciones sociales...

Muchos resultados sobre grafos pueden expresarse de una forma sencilla utilizando la metáfora de las relaciones personales. En *Fotografiando las matemáticas* [Mar00] tenemos algunos ejemplos: "En una reunión de seis o más personas siempre hay tres que se conocen o se desconocen mutuamente". O este otro: "Si en una reunión cada par de personas tiene exactamente un amigo en común, hay alguien que es amigo de todos".

## 4.20 El efecto dominó

190

El juego de dominó corriente consta de 28 fichas distintas, cada una de las cuales tiene representados dos números entre el cero (representado por el blanco) y el seis, como puede verse en el ejercicio 1.14. La colección completa de fichas puede disponerse en un rectángulo de $7 \times 8$ números, donde cada ficha ocupa dos casillas contiguas, poniendo un número en cada una. Por ejemplo, así:



En este ejercicio se pide un programa que analice una matriz de $7 \times 8$ enteros como la siguiente,

```
6 6 2 6 5 2 4 1
1 3 2 0 1 0 3 4
1 3 2 4 6 6 5 4
1 0 4 3 2 1 1 2
5 1 3 6 0 4 5 5
5 5 4 0 2 6 0 3
6 0 5 3 4 2 0 3
```

e informe si se puede cubrir con un juego de dominó, como el de la figura anterior, indicando todas las formas de lograrlo, si hay alguna. Por ejemplo, con la entrada anterior, la salida debería ser la siguiente (en cuatro matrices consecutivas),

```
28 28 14  7 17 17 11 11
10 10 14  7  2  2 21 23
 8  4 16 25 25 13 21 23
 8  4 16 15 15 13  9  9
12 12 22 22  5  5 26 26
27  6 24  3  3 18  1 19
27  6 24 20 20 18  1 19
```

```
28 28 14  7 17 17 11 11
10 10 14  7  2  2 21 23
 8  4 16 25 25 13 21 23
 8  4 16 15 15 13  9  9
12 12 22 22  5  5 26 26
27 24 24  3  3 18  1 19
27  6  6 20 20 18  1 19
```

```
28 28 14  7 17 17 11 11
10 10 14  7  2  2 21 23
 8 15 15 20 18 13 21 23
 8  5  5 20 18 13  9  9
12 12 22 22  3 25 26 26
27  6 24  4  3 25  1 19
27  6 24  4 16 16  1 19
```

```
28 28 14  7 17 17 11 11
10 10 14  7  2  2 21 23
 8 15 15 20 18 13 21 23
 8  5  5 20 18 13  9  9
12 12 22 22  3 25 26 26
27 24 24  4  3 25  1 19
27  6  6  4 16 16  1 19
```

indicando en cada casilla el número de la pieza que la cubre. (Véase la pista 4.20a.)

***Bibliografía*** El enunciado de este ejercicio procede del *1991 ACM Scholastic Programming Contest Finals* (`http://icpc.baylor.edu/icpc/`).

## 4.21 Código de sustitución polialfabético


Un código de sustitución monoalfabético es aquél en el que se elige una permutación del alfabeto de letras en el que se escriben los mensajes; de esta forma, cada letra del mensaje original se cifra con una letra del alfabeto de cifrado. (Veáse el ejercicio 2.6.) Esta relación es biyectiva y por tanto dos apariciones de una misma letra en el mensaje cifrado corresponden al cifrado de una misma letra en el mensaje original. Esto puede utilizarse para descifrar el mensaje original: la frecuencia de aparición de las diferentes letras en un idioma es un dato conocido o calculable. Un código de sustitución *polialfabético* es aquél en el que se utiliza más de un alfabeto de cifrado y, por tanto, distintas apariciones de una misma letra en el mensaje original pueden terminar sustituidas por diferentes letras en el mensaje cifrado.

Puesto que ahora contamos con varios alfabetos de cifrado, necesitamos algún método para elegir uno u otro en cada paso de cifrado de un mensaje. Habitualmente este método se basa en una palabra o frase clave que, junto con los múltiples alfabetos de cifrado, forman la clave del código polialfabético.

Un buen ejemplo de código polialfabético es el del diplomático francés Blaise de Vignère (1523–1596). Este código tiene como alfabetos de cifrado aquéllos que están representados en lo que se conoce como el cuadrado de Vignère y que podemos ver en la figura 4.8. Veamos cuáles son los pasos que hay que seguir para cifrar un mensaje con un ejemplo; el mensaje por cifrar es éste:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z A B C D E F G H I J K L M N O P Q R S T U V W X Y

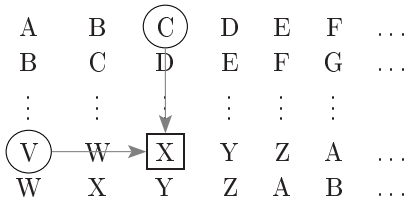
Figura 4.8: Cuadrado de Vignère

CODIGO POLIALFABETICO

Se elige una palabra clave, por ejemplo VIGNERE. Se escribe la palabra clave debajo del mensaje por cifrar, repitiéndola las veces necesarias:

C O D I G O  P O L I A L F A B E T I C O
V I G N E R  E V I G N E R E V I G N E R

Ya tenemos dos entradas para cada letra del mensaje original que podemos utilizar en el cuadrado de Vignère para encontrar la letra que cifrará a cada letra del mensaje original. Consideremos el primer par de letras C y V. Buscamos la $C$ de la primera fila y la $V$ de la primera columna y buscamos la intersección de la columna a la que pertenece la $C$ y de la fila a la que pertenece la $V$; en este caso es $X$. Dicha letra es el cifrado de $C$.




Si procedemos de igual forma con cada una de las letras del mensaje original, tenemos el texto cifrado siguiente:

C O D I G O  P O L I A L F A B E T I C O
V I G N E R  E V I G N E R E V I G N E R
X W J V K F  T J T O N P W E W M Z V G F

Conviene recordar cuál era el problema que intentaban resolver los códigos polialfabéticos: con un código monoalfabético diversas apariciones de la misma letra en el mensaje original se codifican con

la misma letra en el mensaje cifrado y viceversa. Ahora al utilizar un código polialfabético, podemos comprobar que esta relación no se cumple; por ejemplo, tanto la letra $P$ como la primera $L$ que aparece en el mensaje dan lugar a la misma letra $T$ en en el mensaje cifrado. También, las dos letras $O$ de la primera palabra, dan lugar a letras distintas, la primera a $W$ y la segunda a $F$.

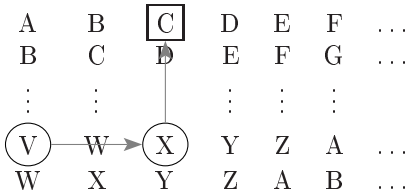
Para descifrar un mensaje cifrado con un código polialfabético, necesitamos saber la clave que se ha utilizado para cifrar. Obviamente el conjunto de alfabetos para codificar que se utiliza forma parte esencial de la clave. En el ejemplo anterior, dicho conjunto es el cuadrado de Vignère (figura 4.8). Pero como veremos más adelante, podríamos haber utilizado otros. Además del conjunto de alfabetos necesitamos saber la palabra clave que se ha utilizado para codificar un mensaje concreto. En nuestro ejemplo anterior la palabra fue VIGNERE.

El proceso de descifrado es inverso al de cifrado; es decir, una vez que conocemos el mensaje cifrado y la palabra clave, tenemos que ser capaces de recuperar el mensaje original. Veamos cuáles son estos pasos utilizando el mensaje que cifrabamos anteriormente utilizando el cuadrado de Vignère y la palabra clave VIGNERE.

Colocamos el mensaje cifrado y sobre él escribimos otra línea con la palabra clave repetida tantas veces como sea necesaria:

| V | I | G | N | E | R | | E | V | I | G | N | E | R | E | V | | I | G | N | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | W | J | V | K | F | | T | J | T | O | N | P | W | E | W | M | Z | V | G | F | |

Ahora tenemos que utilizar el cuadrado de Vignère de forma inversa a como lo hicimos a la hora de codificar. Centrémonos en la primera letra del mensaje, que es $X$; encima tiene la letra $V$ de la palabra clave; entonces localizamos la letra $V$ en la primera columna y buscamos en la fila a la que pertenece dicha $V$ hasta encontrar la letra $X$. Ahora, la primera letra de la columna en la que se encuentra la letra $X$ es la letra del mensaje original, que para este ejemplo es $C$:




Si repetimos el proceso para todas la letras del mensaje cifrado obtenemos:

| V | I | G | N | E | R | | E | V | I | G | N | E | R | E | V | | I | G | N | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | W | J | V | K | F | | T | J | T | O | N | P | W | E | W | M | Z | V | G | F | |
| C | O | D | I | G | O | | P | O | L | I | A | L | F | A | B | E | T | I | C | O | |

El código de Vignère es un ejemplo muy bueno de código polialfabético, pero no es el único. Con anterioridad a Vignère, un abad benedictino, Johannes Trithemius (1462–1516), ya lo había utilizado para cifrar mensajes. Sin embargo, dicho código era mucho más sencillo. La palabra clave que se utilizaba era siempre la misma: el propio alfabeto. Dicho de otra forma, para la primera letra utilizaba la primera fila, para la segunda letra la segunda fila y así sucesivamente.

Otro código polialfabético famoso es el del belga José de Bronckhors, conde de Gronsfeld. En este caso el conjunto de alfabetos que se utiliza es más sencillo, como puede verse en la figura 4.9.

En este caso sólo tenemos 10 filas y, por tanto, si quisiésemos hacer una codificación con palabra clave al estilo del código de Vignère, sólo podríamos utilizar palabras que contuviesen las 10 primeras letras del alfabeto. Pero Gronsfeld no utilizaba palabras clave, sino números clave y por eso le bastaba con las 10 primeras filas. Es decir, la clave elegida podría ser 31415926 por ejemplo (que es fácil de recordar, ya que son las ocho primeras cifras del número $\pi$). Si utilizásemos dicha clave, la primera letra se codificaría utilizando la fila que comienza con 3 en el cuadrado de Gronsfeld, la segunda letra utilizando la fila que comienza con 1 y así sucesivamente.

```
0: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
2: C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
3: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
4: E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
5: F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
6: G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
7: H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
8: I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
9: J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
```

Figura 4.9: Alfabetos de cifrado de Gronsfeld

#### 4.21.1 Tablas para códigos polialfabéticos

Escribe un programa que permita dibujar tablas para códigos polialfabéticos. Haz que dicho programa sea bastante flexible y que permita dibujar tanto tablas ya conocidas, como la de Vignère o la de Gronsfeld, como tablas nuevas determinadas por el usuario que sigan un esquema parecido. (Véase la pista 4.21a.)

#### 4.21.2 Funciones como tablas

Las tablas que dibujaste en el apartado anterior tienen cierta regularidad y eso hace posible que puedas realizar un programa para dibujarlas. Pero en realidad, no necesitamos ni siquiera dibujar las tablas. Escribe ahora una función que reciba los parámetros adecuados (tipo de cifrado, letra origen y letra de la palabra clave) y devuelva la codificación o la descodificación correspondiente.

#### 4.21.3 Tablas aleatorias para códigos polialfabéticos

Escribe un programa que permita crear una tabla de cifrado para un código polialfabético que sea aleatoria, es decir, sin ninguna regularidad.

#### 4.21.4 Almacenamiento en ficheros de tablas de alfabetos

Las tablas de códigos polialfabéticos con ciertas regularidades no necesitan ser almacenadas ya que se puede definir una *fórmula* que las defina. (Veáse el apartado 4.21.2.) En cambio, las tablas aleatorias sí necesitan ser almacenadas. Utiliza archivos para poder almacenar las tablas aleatorias de códigos polialfabéticos que has generado y así utilizar dichos códigos más fácilmente.

#### 4.21.5 Cifrado y descifrado utilizando un código de sustitución polialfabético

Escribe un programa que permita cifrar y descifrar utilizando un código polialfabético. Recuerda que el código consiste en una tabla en la que están los múltiples alfabetos de codificación que utilizaremos y una palabra clave. (Véase la pista 4.21b.)

## 4.22 Triángulo de Pascal




Consideremos el triángulo de Pascal,

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
 & & & & & \ldots & & & & &
\end{array}
$$

cuyos elementos pueden generarse bien directamente mediante números combinatorios, siendo $\binom{i}{j}$ el elemento j-ésimo de la fila i-ésima (para $0 \leq j \leq i$),

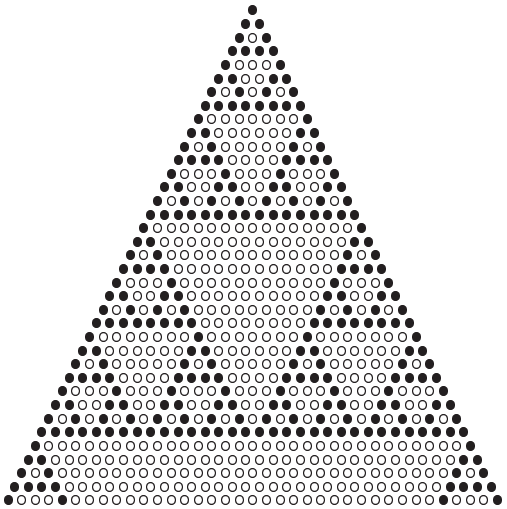
$$\begin{array}{ccccccccccc}
 & & & & \binom{0}{0} & & & & & & \\
 & & & \binom{1}{0} & & \binom{1}{1} & & & & & \\
 & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & \\
 & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & & \\
\binom{4}{0} & & \binom{4}{0} & & \binom{4}{0} & & \binom{4}{0} & & \binom{4}{0} & &
\end{array}$$

$$\cdots$$

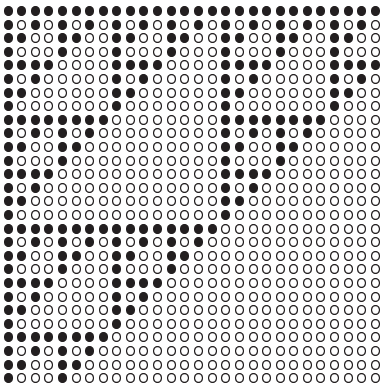
o bien sumando los dos situados sobre él:

$$\begin{array}{ccccccccccc}
 & & & & \cdots & & & & & & \\
 & & & & 1 \quad 8 \quad 28 \quad 56 \quad 70 \quad 56 \quad 28 \quad 8 \quad 1 & & & & \\
 & & & \swarrow\searrow & \swarrow\searrow & \swarrow\searrow & \swarrow\searrow & \swarrow\searrow & \swarrow\searrow & \swarrow\searrow & \swarrow\searrow & & \\
 & & 1 \quad 9 \quad 36 \quad 84 \quad 126 \quad 126 \quad 84 \quad 36 \quad 9 \quad 1 & & & & & & & & & & \\
 & & & & \cdots & & & & & & &
\end{array}$$

Sustituyendo cada número impar por un punto negro y cada número par por uno blanco, se obtiene el triángulo siguiente:



Se pide desarrollar un programa que reproduzca ese dibujo, aunque rotando la figura del siguiente modo, para aprovechar mejor la pantalla,



y sustituyendo cada "•" por un asterisco ('*'), y cada "◦" por un blanco ('␣') para usar los caracteres disponibles.

***Bibliografía*** En este ejercicio se han presentado variaciones sobre el tema de la construcción del triángulo, enfocando distintos problemas de eficiencia que se presentan a menudo. Sin embargo, esta sencilla construcción es abundante en propiedades: además de estar en él presentes los números naturales y, obviamente, los números combinatorios, no es difícil descubrir los triangulares (de diferentes órdenes), los de Fibonacci, etc. Consulta por ejemplo [Enz01]. Los artículos y libros sobre el tema son muy abundantes. Entre ellos, citamos el capítulo 15 de [Gar87], de donde proviene la idea de jugar con la paridad de los coeficientes binomiales. Este atrayente libro incluye, a su vez, una pequeña colección de reseñas bibliográficas sobre el mencionado triángulo.

***Un poco de historia*** El triángulo de Pascal debe su nombre a Blaise Pascal (1623–1662), que en 1653 publicó *Traité du triangle arithmétique*, obra en la que se incluye el estudio más importate sobre el tema. Sin embargo, el triángulo de Pascal era conocido por los matemáticos árabes. Al-Samawal (1130–1180) cita un trabajo de Al-Karaji (953–1029) en el que se daba la construcción de dicho triángulo.

## 4.23 El solitario de los quince


191

En 1878, Samuel Loyd (1841–1911) presentó un juego llamado simplemente *los quince*. Los quince es un solitario que se juega en un tablero de $4 \times 4$ en el que los números del 1 al 15 son colocados al azar en las casillas. Al haber dieciséis casillas, siempre una queda libre. Por ejemplo:

| | | | |
|---|---|---|---|
| 10 | 2 | 15 | 3 |
| 6 | 7 | 5 | 11 |
| 9 | 1 | 8 | 12 |
| 13 | 14 | 4 | |

El objetivo del juego es ordenar completamente los números del tablero. El único movimiento posible en el juego es deslizar, sin levantar, un número al único hueco libre que en cada momento tiene el tablero. Por ejemplo, partiendo de la posición anterior, el 12 puede pasar abajo, luego el 8 a la derecha, el 4 arriba... hasta que se llegue a la solución, es decir, a un tablero con los números ordenados:

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

Cuando Sam Loyd propuso el juego, dio una configuración inicial muy sencilla: todos los números estaban ya ordenados salvo el 1 y el 15 que tenían intercambiadas sus posiciones.

| | | | |
|---|---|---|---|
| 15 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 1 | |

Loyd ofreció una enorme recompensa (1 000 dólares de 1878) al primero que le presentase una solución al juego que proponía. El juego hizo furor rápidamente y se puso de moda en Estados Unidos y en Europa.

***¿Existe solución?*** No siempre hay solución, y para comprobarlo podemos hacer una prueba de paridad sencilla. (Véase la pista 4.23a.)

***Cálculo de permutación resoluble*** Escribe un programa que genere una permutación que pueda solucionarse para jugar a los quince.

***Juego de los 15*** Escribe un programa que permita a una persona jugar al juego de los quince.

***Bibliografía*** Un algoritmo sencillo para el cálculo de una permutación aleatoria que se adapta muy bien a este problema es el conocido como *shuffling* (barajar), y aparece en [Knu95] (sección 3.4.2, algoritmo P).

Si quieres saber algo más sobre el juego y, de paso, jugar un rato, prueba la siguiente dirección: `http://www.cut-the-knot.com/pythagoras/fifteen.html`.

***Un poco de historia*** Los libros de historia a veces nos cuentan el pasado a grandes rasgos y por tanto es difícil hacerse una idea de cómo fue la vida cotidiana en épocas pasadas.

Parece que las modas internacionales y la ludomanía son propias de nuestra sociedad actual y es difícil llegar a asimilar lo que supuso el juego en el siglo XIX, pero en [Per88] nos cuentan que en los Estados Unidos y en Europa, el juego llegó a ser una *plaga*. Se jugaba en todo momento y lugar, y los jefes estaban horrorizados ya que todos sus empleados estaban completamente absorbidos por el juego. En el propio *Reichstag* alemán, los diputados jugaban a los quince...

En el interesante libro [Eli93] del sociólogo alemán Norbert Elias (1897–1990) puedes encontrar un estudio sobre las costumbres europeas desde la Edad Media hasta nuestros días.

## 24 Segmentador de oraciones


Muchas aplicaciones que procesan textos requieren identificar las oraciones que contienen, es decir, distinguir dónde comienza y dónde termina cada oración. Las razones pueden ser diversas. Por ejemplo, el análisis sintáctico del contenido de un texto suele obligar a una identificación previa de las oraciones, antes de abordar el análisis de cada una de ellas. Lo mismo ocurre si se quiere obtener de forma automática un resumen de un documento, porque muchas de las técnicas que se emplean actualmente para esta labor se basan en la identificación de oraciones relevantes; antes de determinar la relevancia de una oración, habrá que delimitarla en el texto.

Un programa que identifica las oraciones en un texto y las *marca* recibe el nombre de *segmentador de oraciones*. En este ejercicio te proponemos desarrollar un segmentador de oraciones que utiliza *patrones tipográficos*.

Para marcar las oraciones vamos a utilizar las etiquetas `<s>` y `</s>`. Cuando el segmentador detecte el comienzo de una oración, añadirá al texto la etiqueta `<s>` y, cuando detecte su final, la etiqueta `</s>`. Veamos qué es lo que se espera del segmentador con un ejemplo. Frente a un documento con el siguiente contenido (extraído del relato "El jardín de senderos que se bifurcan" de *Ficciones* de Jorge Luis Borges),

```
Antes de exhumar esta carta, yo me había preguntado de qué
manera un libro puede ser infinito.   No conjeturé
otro procedimiento que el de un volumen cíclico, circular...
Recordé también esa noche que está en el centro de las
1.001 Noches, con riesgo de llegar otra vez a la noche en que
la refiere, y así hasta lo infinito.
```

el segmentador de oraciones debería crear uno nuevo como el que sigue:

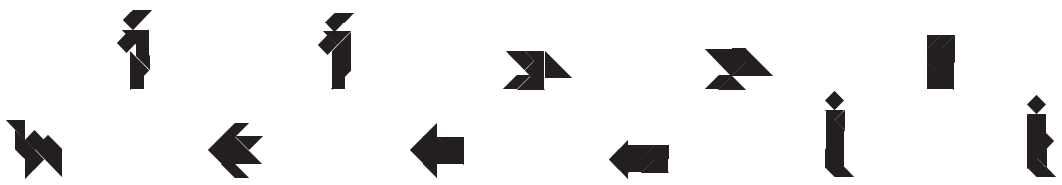
```
<s>Antes de exhumar esta carta, yo me había preguntado de qué
manera un libro puede ser infinito.</s>   <s>No conjeturé
otro procedimiento que el de un volumen cíclico, circular...</s>
<s>Recordé también esa noche que está en el centro de las
1.001 Noches, con riesgo de llegar otra vez a la noche en que
la refiere, y así hasta lo infinito.</s>
```

Del ejemplo precedente podemos extraer algunos de los patrones tipográficos más comunes, aunque tendremos que ampliar la lista:

- Un carácter "." seguido de uno o más espacios y de una letra mayúscula es un patrón tipográfico que refleja el final de una oración y el comienzo de otra (punto y seguido).
- Al patrón anterior le podemos añadir la posibilidad de que detrás del carácter punto y antes de la letra mayúscula pueda aparecer un carácter de fin de línea y, posiblemente, uno o más espacios o un carácter de tabulación (punto y aparte).
- Si detrás de unos puntos suspensivos hay al menos un espacio y detrás una letra mayúscula, podemos estar ante el indicio del paso de una oración a otra. También debemos contemplar la posibilidad de que un carácter de fin de línea aparezca entre los puntos suspensivos y la mayúscula.
- Los signos ortográficos "¿", "?" y "¡", "!" e incluso las comillas también se deberían tener en cuenta en el segmentado de oraciones, ¿no te parece?

Escribe un programa que segmente oraciones utilizando los patrones descritos anteriormente. Con ellos podemos identificar oraciones aunque, desde luego, su cobertura es limitada. Ampliando la lista de patrones podríamos mejorarla; aun así, la técnica de identificación de oraciones con patrones tipográficos tiene sus limitaciones. ¿Se te ocurre alguna?

***Bibliografía*** La esencia de este ejercicio es la búsqueda y tratamiento de ciertos conglomerados de letras en un texto. Hay multitud de problemas que encajan en este replanteamiento, ligeramente más abstracto. Tantos que, con el tiempo, se ha terminado consensuando un minilenguaje, llamado *expresiones regulares*, muy especializado y bastante efectivo para solucionarlos. Algunos lenguajes de programación, como Perl, incorporan expresiones regulares entre sus capacidades primitivas, y para muchos otros hay librerías. En [Fri98] podrás ilustrarte sobre todo lo concerniente a expresiones regulares.



**4.4a.** Si el programa recibe la cadena `--m--g---`, que llamaremos $l$, para saber si $l$ pertenece al sistema **mg**, tenemos que comprobar que $l$ es un axioma o que se ha construido aplicando la regla de formación de teoremas. Como $l$ no es un axioma, ya que no tiene la forma de los axiomas, para ver si es un teorema tenemos que comprobar que proviene de un teorema, es decir, tenemos que aplicar *a la inversa* la regla de formación:

```
--m--g---
    ↓
 --m-g--
    ↓
  --mg-
```

Entonces, para saber si $l$ es un teorema tenemos que comprobar que `--mg-` es un teorema. Esta última cadena no puede ser un axioma y tampoco puede provenir de ningún otro teorema, luego no es un teorema y por tanto no pertenece al sistema **mg**. Por consiguiente, la cadena original $l$ tampoco pertenece al sistema **mg**.

**4.5a.** Define un tipo matriz que en realidad será un array, y realiza los siguientes subprogramas:

```
⟨Tipo base⟩ valorEn(Matriz const& matriz, int const fila, int const columna);
```

para acceder a los elementos de la matriz y

```
void ponerEn(Matriz& matriz, int const fila, int const columna
             ⟨Tipo base⟩ const valor);
```

para poner un valor en la matriz.

**4.8a.** Recuerda que la recursión existe en el lenguaje de programación que estamos utilizando, C++.

**4.10a.** Para detener temporalmente la ejecución del programa entre fotogramas es necesario recurrir a algún servicio del sistema operativo subyacente. Suele haber dos posibilidades. La primera, utilizará alguna función que deje *dormido* a nuestro programa durante un periodo de tiempo. Es muy económica, porque, mientras, el sistema operativo se puede dedicar a otras tareas. La segunda, es una espera activa organizada alrededor de un bucle que consulta sistemáticamente la hora hasta que se sobrepasa el tiempo de espera necesario. Aun así, la sincronización perfecta será difícil. En el primer caso porque es posible que el sistema operativo no reanude la ejecución de nuestro programa hasta bien sobrepasada nuestra intención de espera. En el segundo, porque alguna vez nuestro programa tendrá que ceder el control de ejecución a otras tareas y puede que esto ocurra en el peor momento, cuando tengamos que mostrar el siguiente fotograma.

En C++, la función

```
void usleep(unsigned long microsegundos);
```

en el fichero cabecera `unistd.h`, detiene la ejecución del programa durante los microsegundos que reciba como argumento.

**4.11a.** Aunque el número tenga más cifras que elementos tiene la tabla de módulos, el algoritmo puede aplicarse igualmente. De hecho, hemos calculado una tabla de módulos mínima para no repetir información, pero puede aplicarse a cualquier número, por grande que sea. Si tenemos un número que tiene más cifras que entradas en la tabla, entonces tenemos que extender la tabla tantas veces como sea necesario.

Si consideramos la tabla de módulos del 7,

| 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|

y el número 2398765541172 que tiene 13 cifras, entonces necesitamos *pegar* tres tablas para poder abarcar todas sus cifras:

| 5 | 4 | 6 | 2 | 3 | 1 | 5 | 4 | 6 | 2 | 3 | 1 | 5 | 4 | 6 | 2 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 5 | 4 | 1 | 1 | 7 | 2 |

**4.12a.** Ten en cuenta que

- La lectura de ficheros es una operación muy costosa, no abuses de ella.
- La tabla de primos está (o puede estar) ordenada, por tanto, aprovecha esta información en las búsquedas.

**4.15a.** Sugerencia de presentación: genera una salida compatible con el ejercicio 4.10.

**4.16a.** Es aconsejable pensar que la transmisión de datos a través del canal ruidoso se hace enviando las palabras del código de una en una. Por ejemplo, si queremos mandar el mensaje "ir al norte, ir al este, ir al norte...", codificado con "000,011,000,...", podemos suponer que primero enviamos por el canal "000", luego "011", luego "000" y así sucesivamente. Cada uno de estos *bloques* enviados puede sufrir los errores de transmisión.

El ejercicio 3.13 te será de ayuda a la hora de construir funciones aleatorias con la distribución de probabilidad que desees.

**4.16b.** El programa tendrá que realizar las siguientes tareas:

- Dar a elegir el código con el que se trabajará informándonos de sus propiedades (distancia mínima) para detectar y corregir errores.
- Gestionar la lectura y codificación de los datos que se envían.
- Enviar los datos codificados, es decir, las palabras del código, a través de un canal ruidoso.
- Recibir los datos como salen del canal e intentar descodificar dichos datos, detectando y corrigiendo errores cuando sea posible.
- Comparar el mensaje original con el mensaje descodificado para comprobar la eficacia del código elegido con la probabilidad de fallo de transmisión considerada.

Muchas de estas tareas ya las has implementado en los apartados anteriores.

**4.18a.** Si te fijas un poco, no es necesario tener una *tabla* de dos dimensiones. Basta con tener un vector.

**4.19a.** Supón que preguntamos a la persona $p_i$ si conoce a $p_j$; si la respuesta es afirmativa, $p_i$ no puede ser una persona famosa ya que conoce a alguien; si, por el contrario, $p_i$ no conoce a $p_j$, $p_j$ no puede ser una persona famosa.

**4.20a.** Recurre una vez más a nuestra amiga *Recursión*.

**4.21a.** Estudia con detenimiento cuáles son los parámetros necesarios para poder escribir un cuadrado parecido al de Vignère o de Gronsfeld. Por ejemplo, el primer alfabeto de cifrado de la tabla, el número de alfabetos de que consta la tabla...

**4.21b.** Realiza los métodos de cifrado y descifrado de forma suficientemente abstracta para que puedas utilizar indistintamente las tablas aleatorias o las tablas regulares. Para las tablas aleatorias deben usarse los ficheros donde están almacenadas (4.21.4) y para las tablas regulares tienes que utilizar la función que define dicha tabla (4.21.2).

**4.23a.** Si escribimos los números que se encuentran en el tablero (incluyendo el hueco) como la permutación $a_1, a_2, \ldots, a_n$, cada movimiento del juego supone un intercambio de dos elementos de la permutación, es decir, pasar de la permutación $a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n$ a la permutación $a_1, \ldots, a_j, \ldots, a_i, \ldots, a_n$. Un intercambio de dos elementos de la permutación produce un cambio en la paridad de su índice (número de inversiones de sus elementos). Como debemos hacer un número par de movimientos para dejar el hueco en la esquina inferior derecha, la paridad del índice de la permutación inicial será igual que la paridad del índice de la permutación final.

## 41 Ponle tú el título

159

La función devuelve el valor booleano `true` si, en cada fila de la matriz, los elementos están ordenados en orden estrictamente creciente. Por el contrario, devolverá valor booleano `false` si hay al menos una fila en la que los elementos no están en orden estrictamente creciente.

## 42 Yahoos

159

### 4.2.1 Tipo de datos

Aunque las cantidades podrían definirse como números enteros, el papel de *desconocido* y de *muchos* no se corresponde directamente con ningún número. Además, se trata de un número de valores muy reducido. Por tanto, optamos por definir los valores del tipo mediante una enumeración:

```
typedef enum Cantidad {
  desconocido, cero, uno, dos, tres, cuatro, muchos
} Cantidad;
```

### 4.2.2 Escritura de cantidades

Puesto que las posibles cantidades están definidas por enumeración, han de considerarse caso por caso. Por sencillez, elegimos los caracteres que se indican seguidamente para cada una de las cantidades:

```
ostream& operator<<(ostream& out, Cantidad const cantidad) {
  switch (cantidad) {
    case desconocido:  out << '?'; break;
    case cero       :  out << '0'; break;
    case uno        :  out << '1'; break;
    case dos        :  out << '2'; break;
    case tres       :  out << '3'; break;
    case cuatro     :  out << '4'; break;
    case muchos     :  out << 'M'; break;
  }
  return out;
}
```

### 4.2.3 Operación siguiente

La función que se pide avanza siempre un paso salvo con los valores extremos, `desconocido` y `muchos`:

```
Cantidad siguiente(Cantidad const c) {
  if (c == desconocido) return desconocido;
  else if (c == muchos) return muchos;
  else return (Cantidad)(c+1);
}
```

### 4.2.4 Operación suma

La descripción de la operación suma que aparece en el enunciado detalla prácticamente el procedimiento de cálculo: primero consideramos el caso en que uno de los datos es desconocido; en caso contrario, incrementamos uno de ellos (`b`) tantas veces como diga el otro (`a`), o sea, le añadimos a `b` la cantidad `a`, de unidad en unidad.

Estas adiciones se hacen con la función `siguiente` definida antes, para evitar desbordamientos:

```
Cantidad operator+(Cantidad const a, Cantidad const b) {
   if (a == desconocido || b == desconocido) {
     return desconocido;
   } else {
     Cantidad auxB = b;
     for (int i=uno; i<=a; i++) {
       auxB = siguiente(auxB);
     }
     return auxB;
   }
}
```

### 4.2.5 Las tablas

Estas tablas tienen estructura de matriz bidimensional: los índices (columna de la izquierda y fila superior) son ambos del tipo `Cantidad`, y los valores de las casillas centrales también:

```
typedef Cantidad Tabla[muchos+1][muchos+1];
```

### 4.2.6 Tabla de sumar

El cometido de este subprograma es considerar cada una de las casillas de la matriz y, una a una, hallar su valor (que viene definido por la función `suma`, anteriormente descrita) y asignarlo en la celda correspondiente; recordemos que los arrays en C++ siempre se pasan a los subprogramas por referencia.

```
void rellenarTablaSuma(Tabla ts) {
  for (Cantidad i = desconocido; i <= muchos; i = (Cantidad)(i + 1)) {
    for (Cantidad j = desconocido; j <= muchos; j = (Cantidad)(j + 1)) {
      ts[i][j] = i+j;
    }
  }
}
```

### 4.2.7 Tabla de multiplicar

En este caso, hay que realizar un recorrido simultáneo del archivo (del que se van tomando los datos) y de la matriz:

```
ifstream archiMulti("producto.txt");
Tabla tablaMulti;
for (Cantidad i = desconocido; i <= muchos; i = (Cantidad)(i + 1)) {
  for (Cantidad j = desconocido; j <= muchos; j = (Cantidad)(j + 1)) {
    archiMulti >> tablaMulti[i][j];
  }
}
```

Hemos definido el extractor >> para que se puedan leer directamente de un fichero las cantidades de los Yahoos.

```
istream& operator>>(istream& in, Cantidad& cantidad) {
  char c;
  in >> c;
  switch (c) {
    case '?':  cantidad = desconocido ; break;
    case '0':  cantidad = cero        ; break;
    case '1':  cantidad = uno         ; break;
    case '2':  cantidad = dos         ; break;
    case '3':  cantidad = tres        ; break;
    case '4':  cantidad = cuatro      ; break;
    case 'M':  cantidad = muchos      ; break;
  }
  return in;
}
```

### 4.2.8 Uso de los recursos definidos

Una vez definidas y rellenas las tablas de sumar y multiplicar, no hay más que introducir las variables con los valores indicados y reescribir la expresión planteada, aunque cambiando las operaciones de suma y producto '+' y '∗' por consultas apropiadas a las tablas correspondientes:

```
Cantidad const b = uno;                                  // b = 1
Cantidad const c = tres;                                 // c = 3
Cantidad const d = uno;                                  // d = 1
Cantidad const a = tablaSuma[b][tablaMulti[c][d]]; // a = b+c*d
cout << "uno + tres * uno -> " << a << endl;
```

## 4.3 Regla de Ruffini

161

***División de un polinomio por un binomio*** Empezamos con una definición de polinomio sencilla, si bien no la mejor de todas: un polinomio será un array de coeficientes, más un entero que indica su grado:

```
int const maxGrado = ⟨un límite al grado de los polinomios⟩;
typedef struct Polinomio{
  int coeficiente[maxGrado];
  int grado;
} Polinomio;
```

La división entre un polinomio cualquiera (de grado mayor que 0) y un polinomio de grado uno da como cociente un polinomio de grado menor en una unidad, y como resto un entero (0 si la división es exacta).

Para aplicar la regla de Ruffini, el polinomio divisor es un binomio de la forma $(x-a)$. Hagámosla con el ejemplo que se muestra en el enunciado, prestando atención a los grados del cociente y del dividendo:

| | | | | | |
|---|---|---|---|---|---|
| grados del dividendo | | 3 | 2 | 1 | 0 |
| | | 1 | 2 | −1 | −2 |
| | 3 | | 3 | 15 | 42 |
| | | 1 | 5 | 14 | 40 |
| grados del cociente | | 2 | 1 | 0 | |

significando que el cociente es $1x^2 + 5x^1 + 14x^0$, y 40 el resto. El coeficiente de mayor grado del cociente es igual al coeficiente de mayor grado del dividendo. A partir de ahí, para calcular el coeficiente de grado $i$ hay que multiplicar el divisor por el de grado $i+1$ del cociente más el coeficiente de grado $i+1$ del dividendo. El resto es la suma del coeficiente de grado 0 con el producto del divisor por el coeficiente de grado 0 del cociente:

```
void divide(Polinomio const& polinomio, int const& divisor,
            Polinomio& cociente, int& resto) {
  cociente.grado = polinomio.grado-1;
  cociente.coeficiente[cociente.grado] = polinomio.coeficiente[polinomio.grado];
  for (int i = cociente.grado-1; i >= 0; i--) {
    cociente.coeficiente[i] =
      polinomio.coeficiente[i+1] + cociente.coeficiente[i+1]*divisor;
  }
  resto = polinomio.coeficiente[0] + cociente.coeficiente[0]*divisor;
}
```

***División repetida de un polinomio por un binomio*** Para realizar la división múltiple, se divide el polinomio por el divisor hasta dar con un resto distinto de cero. Siempre que el resultado sea 0 hay que actualizar el dividendo.

```
void divideMultiple(Polinomio const& polinomio, int const& divisor,
                    Polinomio& cociente, int& multiplicidad) {
  multiplicidad = 0;
  int resto = 0;
  cociente = polinomio;
  while (resto == 0 && cociente.grado > 0) {
    Polinomio dividendo;
    divide(cociente, divisor, dividendo, resto);
    if (resto == 0) {
      cociente = dividendo;
      multiplicidad++;
    }
  }
}
```

***Cálculo de todas las raíces enteras*** Por último, para calcular todas las raíces enteras habrá que considerar los divisores del término independiente; cada divisor debe ser considerado con su signo positivo y su signo negativo. Además, en cada vuelta del bucle, el polinomio dividendo será diferente. Cuando podemos hacer una división, y la hacemos, el término independiente del nuevo polinomio será un divisor del anterior, por lo que el bucle principal será un `while`. Una forma sencilla de tratar los signos positivo y negativo es mediante un bucle como el siguiente:

```
for (int signo = -1; signo <= 1; signo = signo+2) {...}
```

Así pues, el programa que calcula todas las raíces enteras de un polinomio consiste básicamente en recorrer, con `divisor`, los divisores del término independiente, considerando los de signo positivo y negativo:

```
int main() {
  Polinomio polinomio = leePolinomio();
  int divisor = 1;
  while (divisor <= abs(polinomio.coeficiente[0]) && polinomio.grado > 0) {
    if (polinomio.coeficiente[0] % divisor == 0) {
      for (int signo = -1; signo <= 1; signo = signo+2) {
        Polinomio cociente;
        int multiplicidad;
        divideMultiple(polinomio, divisor*signo, cociente, multiplicidad);
        if (multiplicidad > 0) {
          polinomio = cociente;
          cout << "Encontrada raíz  " << divisor*signo;
          cout << " con multiplicidad " << multiplicidad << endl;
        }
      }
    }
    divisor++;
  }
  cout << "Polinomio restante:  ";
  for (int i = polinomio.grado; i >= 0; i--) {
    cout << polinomio.coeficiente[i] << " ";
  }
  cout << endl;
}
```

## 4.6 Centro de un vector 163

### 4.6.1 Solución directa

La solución más inmediata consiste en ir tanteando todos los índices entre 0 y $n$ en busca de uno que cumpla la ecuación que lo instaura como centro.

```
void centroTrivial(double pesos[], int const n, bool& tiene, int& centro) {
  int i = 0;
  while (i <= n && palanca(pesos, 0, i-1, i) != palanca(pesos, i+1, n, i)) i++;
  tiene = i <= n;
  centro = i;
}
```

Los argumentos de este procedimiento son los que cabría esperar: recibe el array con los `pesos` de cada casilla y el índice (`n`) de la última casilla (`pesos` tiene `n` + 1 casillas); devuelve el índice del `centro` si acaso lo `tiene`. En breve haremos una solución más eficiente con estos mismos argumentos, pero primero vamos a terminar ésta.

La función `palanca` calcula las sumas de cada lado de la ecuación que define el centro. Si se observan esas sumas, se verá que difieren en un pequeño detalle que uniformamos recurriendo al valor absoluto:

$$\sum_{i=0}^{c-1} |c-i| \cdot V[i] = \sum_{c+1}^{n} |c-j| \cdot V[j]$$

Ahora, ambos lados tienen la misma forma y se ajustan a una función que recibe el array de `pesos`, los extremos del rango de índices a sumar ([`inicio`, `fin`]) y el candidato a `centro`:

```
double palanca(double pesos[], int const inicio, int const fin, int const centro) {
  double p = 0;
  for (int i = inicio; i <= fin; i++) {
    p += abs(centro - i) * pesos[i];
  }
  return p;
}
```

La solución que acabamos de dar es tan directa como ineficiente, porque se recorren repetidas veces las mismas posiciones del array. Para tener una medida cuantitativa con la que comparar otros algoritmos, vamos a contar el número de sumas (o restas) y productos que requiere esta solución. La función `palanca` tiene un bucle que da $\texttt{fin} - \texttt{inicio} + 1$ vueltas; en cada vuelta hace 2 sumas (una suma y una resta) y 1 producto. El procedimiento `centroTrivial` también tiene un bucle, que llama dos veces a `palanca` en cada vuelta; los índices de estas llamadas casi encajan y sólo dejan fuera una posición del array de pesos; por tanto, esas dos llamadas involucran $2n$ sumas y $n$ productos. ¿Cuántas vueltas da el bucle de `centroTrivial`? Depende de los datos; desde 1 hasta $n+1$. Si suponemos que cualquier número de vueltas es equiprobable, al promediar muchas ejecuciones parecerá que da $1 + n/2$ vueltas. Desde este punto de vista, `centroTrivial` ejecuta $2n + n^2$ sumas y $n + n^2/2$ productos. En breve veremos que esto es mucho.

### 4.6.2 Solución incremental

Para tantear si varias posiciones consecutivas son centro, no es necesario realizar todos los cálculos cada vez. Se puede reciclar gran parte de ellos de un intento para el siguiente. Fijemos nuestra atención en el lado izquierdo de la ecuación que define a $c$ como centro:

$$\sum_{i=0}^{c-1}(c-i)V_i.$$

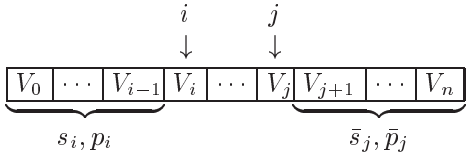
Si acaso $c$ fue un intento fallido de centro y queremos probar con $c+1$, habrá que calcular lo siguiente:

$$\sum_{i=0}^{(c+1)-1}((c+1)-i)V_i = \sum_{i=0}^{c}((c-i)V_i + V_i) = \sum_{i=0}^{c}(c-i)V_i + \sum_{i=0}^{c}V_i = \sum_{i=0}^{c-1}(c-i)V_i + \sum_{i=0}^{c}V_i$$

Si llamamos $p_c$ a $\sum_{i=0}^{c-1}(c-i)V_i$ y $s_c$ a $\sum_{i=0}^{c-1}V_i$, resulta que $p_{c+1} = p_c + s_{c+1}$. Obviamente, $s_{c+1} = s_c + V_c$. Se puede hacer un análisis análogo del lado derecho de la ecuación.

Cuando se resuelve un cómputo basándose en uno anterior, se dice que se ha calculado de forma incremental; es una técnica que se puede aprovechar en multitud de problemas. (Véase el ejercicio 4.18.)

La otra idea clave para esta solución es *acorralar el centro*. En vez de suponer que un cierto índice es el centro y dedicarnos a comprobarlo, vamos a intentar descubrir dónde está. Vamos a seguir la estrategia del siguiente diagrama:




Tenemos dos índices, $i$ y $j$, que delimitan la región de candidatos a centro. Todo lo que queda a la izquierda de $i$ está acumulado en $s_i$ y $p_i$ (siguiendo la notación que acabamos de introducir para el

cálculo incremental); igualmente, todo lo que queda al lado derecho está acumulado en $\bar{s}_j$ y $\bar{p}_j$ (que será la notación para los elementos equivalentes en el cálculo incremental por la derecha).

Con el propósito de acorrarlar el centro, hay que ir estrechando el margen entre $i$ y $j$. El criterio para mover uno u otro es trivial: intentar que se cumpla la ecuación del centro. Si $p_i < p_j$, habrá que mover $i$ un paso a la derecha con el objetivo de atrapar un nuevo elemento que permita compensar la diferencia; si $p_i > p_j$, será $j$ el que se desplace una posición a la izquierda; si $p_i = p_j$, pero todavía hay más de un candidato a centro, habrá que abandonar esta idílica situación moviendo cualquiera de los dos (o ambos). Por supuesto, los movimientos de los índices exigen cambios incrementales de $s_i$ y $p_i$ o $\bar{s}_j$ y $\bar{p}_j$.

¿Cómo comienza la estrategia acorraladora? Los índices $i$ y $j$ empezarán en 0 y $n$ respectivamente; como no hay rango de pesos que acumular, $s_0$, $p_0$, $\bar{s}_n$ y $\bar{p}_n$ serán 0.

Uniendo todo lo anterior, tenemos el cálculo incremental del centro:

```
void centroIncremental(double pesos[], int const n, bool& tiene, int& centro) {
  int i = 0;  double si = 0, pi = 0;
  int j = n;  double sj = 0, pj = 0;
  while (i < j) {
    if (pi < pj) {
      si += pesos[i];
      pi += si;
      i++;
    } else {
      sj += pesos[j];
      pj += sj;
      j--;
    }
  }
  tiene = i == j;
  centro = i;
}
```

Sólo queda contar sumas y productos para comprobar que hemos mejorado. El bucle del subprograma `centroIncremental` aumenta o bien `i` o bien `j` hasta que son iguales; como inicialmente difieren en `n`, estos índices serán iguales tras justamente $n$ vueltas. Cada vuelta elige una u otra rama del condicional; ambas tienen la misma estructura que se resume en 2 sumas. Por tanto, `centroIncremental` realiza $2n$ sumas y ningún producto.

La mejora es abismal. Para aprehenderla calcularemos tiempos de ejecución. Supongamos que tenemos que encontrar el centro de un vector con $10^5$ entradas en una máquina que es capaz de ejecutar un millón ($10^6$) de sumas o multiplicaciones por segundo. Con `centroIncremental` tardaremos 2/10 de segundo. Con `centroTrivial` tardaremos $(3 \cdot 10^5 + \frac{3}{2}(10^5)^2)/10^6$ segundos; si simplificamos, resulta algo más de 4 horas.

### 4.6.3 Solución matemática

Aparte de las soluciones puramente algorítmicas propuestas, se puede razonar de forma más abstracta sobre la definición de centro; afirma que el centro de un vector con índices entre 0 y $n$ es un índice $c$ que verifica la siguiente igualdad:

$$\sum_{i=0}^{c-1}(c-i)\cdot V_i = \sum_{j=c+1}^{n}(j-c)\cdot V_j.$$

Si se pasan los dos sumatorios al mismo miembro tenemos lo siguiente,

$$\sum_{i=0}^{c-1}(c-i)\cdot V_i - \sum_{j=c+1}^{n}(j-c)\cdot V_j = 0$$

o lo que es lo mismo:

$$\sum_{i=0}^{c-1}(c-i)\cdot V_i + \sum_{j=c+1}^{n}(c-j)\cdot V_j = 0.$$

Puesto que $c-c$ es igual a 0, el índice $c$ puede incluirse en cualquiera de los dos sumatorios; por ejemplo, en el primero:

$$\sum_{i=0}^{c}(c-i)\cdot V_i + \sum_{j=c+1}^{n}(c-j)\cdot V_j = 0.$$

Y ya se pueden juntar los sumatorios en uno, dando lugar a

$$\sum_{i=0}^{n}(c-i)\cdot V_i = 0.$$

Ahora, distribuimos y sumamos término a término,

$$\sum_{i=0}^{n} c\cdot V_i + \sum_{i=0}^{n} -i\cdot V_i = 0$$

sacamos $c$, por ser una constante,

$$c\cdot\sum_{i=0}^{n} V_i = \sum_{i=0}^{n} i\cdot V_i$$

y obtenemos la relación siguiente:

$$c = \frac{\sum_{i=0}^{n} i\cdot V_i}{\sum_{i=0}^{n} V_i}$$

¡Y ya está! Basándonos en esta relación, el cálculo del centro de un vector, si existe, es tarea fácil:

```
void centroFacil(double pesos[], int const n, bool& tiene, int& centro) {
  double s = 0;
  double ps = 0;
  for (int i = 0; i <= n; i++) {
    s += pesos[i];
    ps += i * pesos[i];
  }
  double const posible = (s != 0) ?  (ps / s) :  0;
  centro = (int) posible;
  tiene = posible == centro && centro != 0;
}
```

Es muy fácil contar las operaciones que hace `centroFacil`. Toda la tarea se concentra en un bucle que hace dos sumas y un producto en cada una de sus $n+1$ vueltas. En definitiva, requiere $2(n+1)$ sumas y $n+1$ productos. Los productos hacen que esta solución sea algo peor que `centroIncremental`, aunque no es un cambio radical; basta volver a echar las cuentas de nuestro caso hipotético para comprobar que

necesitaríamos 3/10 de segundo, un 50% más. Ahora bien, puede que los productos no sean inherentes a esta idea sino a la forma de abordar su concreción en forma de código. ¿Quién va a ser capaz de reestructurar ese bucle para que en cada vuelta haga sólo 2 sumas?

## 4.10 AnimASCIIón 166 189

La siguiente función es el núcleo de nuestro programa:

```
void anima(istream& fuente) {
  Imagen imagen;
  bool hubo;
  cargaImagen(fuente, imagen, hubo);
  while (hubo) {
    muestraImagenSuave(imagen);
    int const vecesAnterior = imagen.veces;
    cargaImagen(fuente, imagen, hubo);
    usleep(vecesAnterior*microsPorMarco);
  }
}
```

Se organiza alrededor del tipo `Imagen`.

Es bastante fácil representar un fotograma de AnimASCIIón en memoria; hay que guardar, por una parte, el número de veces que se repite el fotograma y, por otra, los caracteres de la imagen. Sabemos que una imagen tiene

```
int const altura = 13;
```

y

```
int const anchura = 67;
```

Podríamos representar una imagen con una matriz de caracteres de tamaño `altura` × `anchura`. Pero todo será más cómodo si recurrimos al tipo `string`; por de pronto, nos podemos olvidar de la `anchura`, porque este tipo admite una cantidad arbitraria de caracteres. Así, un fotograma será

```
typedef struct Imagen {
  int veces;
  string lineas[altura];
} Imagen;
```

Simplificando al máximo, la función que muestra un fotograma se reduce a

```
void muestraImagen(Imagen& imagen) {
  for (int i = 0; i < altura; i++) {
    cout << imagen.lineas[i] << '\n';
  }
  cout << flush;
}
```

Para que la proyección sea lo más suave posible es necesario que la imagen se escriba en un solo golpe. Como el manipulador `endl` no sólo termina una línea sino que fuerza a que se muestre todo lo escrito hasta el momento, hay que limitarse a `'\n'`. Y por eso hay que completar la escritura con un uso del manipulador `flush`.

Hay que entender que `cout` puede mostrar lo que se ha escrito antes de hacer un `endl` o un `flush`.

De hecho podría mostrar carácter a carácter. Si queremos garantizar la escritura en un solo golpe hay que construir una cadena de caracteres intermedia que contenga toda la imagen:

```
void muestraImagenSuave(Imagen& imagen) {
  string medio;
  for (int i = 0; i < altura; i++) {
    medio += imagen.lineas[i];
    medio += '\n';
  }
  cout << medio << flush;
}
```

Cargar un fotograma tiene sus puntos delicados. Primero, puede que no haya más fotogramas; el tercer parámetro de `cargaImagen` debe indicarnos si continúa la película. Segundo, hay que mezclar dos tipos de lectura: la del número de repeticiones y la de las líneas de la imagen. La instrucción

```
fuente >> imagen.veces;
```

lee el siguiente entero de `fuente`. Quitará todos los blancos que haya antes del número, pero no tocará nada de lo que siga. En cambio, la instrucción

```
getline(fuente, imagen.linea[i]);
```

leerá todos los caracteres hasta el siguiente final de línea incluido; este último será el único que no ponga en `imagen.linea[i]`. Esta diferencia de comportamiento nos obliga a consumir explícitamente todo lo que haya tras el número de repeticiones y hasta el primer final de línea incluido. Se resuelve esta tarea con el código

```
do {
  fuente.get(c);
} while (!fuente.fail() && c != '\n');
```

que podríamos poner en un subprograma pero que, por deferencia a la filosofía de la librería `iostream`, vamos a preferir ponerlo en un manipulador. Lo llamaremos `endls`:

```
istream& endls(istream& fuente) {
  char c;
  do {
    fuente.get(c);
  } while (!fuente.fail() && c != '\n');
  return fuente;
}

void cargaImagen(istream& fuente, Imagen& imagen, bool& hubo) {
  fuente >> imagen.veces >> endls;
  if (fuente.fail()) {
    hubo = false;
  } else {
    hubo = true;
    for (int i = 0; i < altura; i++) getline(fuente, imagen.lineas[i]);
  }
}
```

Tal y como hemos dicho en la pista 4.15a generaremos una salida compatible con el ejercicio 4.10 de forma que podamos visualizar fácilmente la simulación. Antes de comenzar con la simulación propiamente dicha debemos leer la configuración inicial: el tamaño de la ventana de visualización, el tiempo de simulación, el fichero donde guardar la simulación y la posición, masa y velocidad de cada planeta. En primer lugar establecemos el tamaño de la ventana de simulación. Asumimos que la esquina inferior se corresponde con la coordenada $(0, 0)$ por lo que es suficiente con saber la coordenada de la esquina superior derecha. Definimos el tipo `Coordenada` que nos servirá para representar una coordenada en el plano:

```
typedef struct Coordenada {
  double x, y;
} Coordenada;
```

Así los parámetros iniciales se pueden establecer a partir de la línea de comandos: los dos primeros establecen la esquina superior derecha de la ventana de animación; el siguiente es el nombre del fichero de salida; a continuación viene el tiempo total de animación, y después la configuración de cada uno de los planetas. Necesitamos un tipo `ListaPlanetas`, que será definido posteriormente, con el que manejar nuestra colección de planetas.

```
int main(int argc, char * args[]) {
  Coordenada esquinaSupDer = {atof(args[1]), atof(args[2])};
  ofstream salida(args[3]);
  int tiempoTotal = atoi(args[4]);
  ListaPlanetas listaPlanetas;
  ⟨Construye listaPlanetas a partir de args⟩
  ⟨Realizar animación de listaPlanetas, se graba en el fichero salida⟩
  salida.close();
}
```

Dejaremos el procedimiento para leer la configuración de planetas

```
void construyeListaPlanetas(ListaPlanetas& listaPlanetas, char * args[]);
```

para más adelante, puesto que para realizarlo conviene tener antes una definición de tipo más completa que vendrá determinada por el resto de los procedimientos. Este procedimiento construirá una configuración inicial de planetas a partir de los datos que se introducirán en la línea de comandos. Puesto que la configuración de los planetas empieza en la quinta posición de los parámetros de entrada, la llamada al mismo será:

```
construyeListaPlanetas(listaPlanetas, &args[5]);
```

A continuación definiremos el procedimiento para llevar a cabo la simulación. Los parámetros necesarios para ello son: la lista de planetas, que actuará como un parámetro de entrada y salida, y el tiempo total de simulación. Y para la visualización de la misma, será necesario proporcionar el tamaño de la ventana de visualización y el fichero de salida:

```
void animacion(ListaPlanetas& listaPlanetas, int const tiempoTotal,
               Coordenada const& esquinaSupDer, ostream& salida);
```

La llamada a este subprograma será de la forma siguiente:

```
animacion(listaPlanetas, tiempoTotal, esquinaSupDer, salida);
```

El cuerpo de este procedimiento será un bucle: tenemos en cuenta el tiempo transcurrido de simulación y saldremos del bucle cuando este tiempo supere el tiempo máximo de la simulación. En cada vuelta del bu-

cle incrementamos el tiempo transcurrido con $\Delta t$, que estará definida como una constante `unidadTiempo`, y cambiaremos la configuración de los planetas. El programa de animación necesita un fotograma cada $\frac{1}{15}$ segundos; por tanto cada intervalo de $\frac{1}{15}$ segundos sacaremos una foto de la configuración en el fichero de salida.

```
double tiempoTranscurrido = 0;
while (tiempoTranscurrido < tiempoTotal) {
  cambiaConfiguracion(listaPlanetas);
  tiempoTranscurrido = tiempoTranscurrido + unidadTiempo;
  if (⟨ha transcurrido el tiempo entre fotogramas⟩) {
    ⟨Sacar una foto del estado de los planetas⟩
  }
}
```

El intervalo de tiempo entre fotogramas está definido como una constante `intervaloEntreFotogramas`. Para saber si debemos sacar una foto o no, introducimos una variable en la que llevamos el tiempo que falta hasta la siguiente foto `faltaHastaFotograma`. Sacaremos la foto cada vez que esta variable tenga un valor menor o igual que cero; además, deberemos reestablecer el valor de esta variable para sacar el siguiente fotograma. En cada vuelta del bucle disminuimos este valor con `unidadTiempo`. Así, el cuerpo del bucle se completa de la siguiente forma:

```
faltaHastaFotograma = faltaHastaFotograma - unidadTiempo;
if (faltaHastaFotograma <= 0) {
  sacaFoto(listaPlanetas, esquinaSupDer, salida);
  faltaHastaFotograma = intervaloEntreFotogramas;
}
```

El valor de la variable `faltaHastaFotograma` se establece antes del bucle así:

```
double faltaHastaFotograma = intervaloEntreFotogramas;
```

En este procedimiento han aparecido los siguientes procedimientos

```
void cambiaConfiguracion(ListaPlanetas& listaPlanetas);
```

para cambiar la configuración de los planetas, y este otro:

```
void sacaFoto(ListaPlanetas& listaPlanetas, Coordenada const& esquinaSupDer,
              ostream& salida);
```

para sacar una foto de la configuración de planetas.

Ambos procedimentos deben recorrer la `listaPlanetas`: el primero calculando la nueva posición y velocidad de cada planeta, y el segundo para generar un fotograma con la posición de cada planeta.

Para poder realizarlos definiremos antes el tipo `ListaPlanetas`, que básicamente consistirá en un array de planetas más un entero que nos indica el tamaño del mismo:

```
int const maxPlanetas = ⟨un número razonable de planetas⟩;
typedef struct ListaPlanetas {
  Planeta planetas[maxPlanetas];
  int numPlanetas;
} ListaPlanetas;
```

La definición del tipo `Planeta` la dejaremos para más adelante, centrándonos ahora en el procedimiento `cambiaConfiguracion` que queda como sigue:

```
void cambiaConfiguracion(ListaPlanetas& listaPlanetas) {
  for (int i = 0; i < listaPlanetas.numPlanetas; i++)
    actualizaPlaneta(listaPlanetas, i);
}
```

El procedimiento `actualizaPlaneta` debe calcular la fuerza total que ejercen en un instante el resto de los planetas sobre el planeta considerado, para actualizar su posición y velocidad aplicando dicha fuerza:

```
void actualizaPlaneta(ListaPlanetas& listaPlanetas, int const planeta) {
  Fuerza fuerza = calculaFuerzaSobre(listaPlanetas, planeta);
  aplicaFuerza(listaPlanetas.planetas[planeta], fuerza);
}
```

Han aparecido la función `calculaFuerzaSobre`, el procedimiento `aplicaFuerza` y el tipo `Fuerza`. Puesto que estamos considerando un universo en el plano, la fuerza tendrá dos componentes, es decir, que la podemos definir a partir del tipo `Coordenada`

```
typedef Coordenada Fuerza;
```

La función `calculaFuerzaSobre` debe recorrer la lista de planetas totalizando la fuerza que ejercen los demás planetas sobre el indicado:

```
Fuerza calculaFuerzaSobre(ListaPlanetas& listaPlanetas, int const planeta) {
  Fuerza fuerzaTotal;
  fuerzaTotal.x = 0;
  fuerzaTotal.y = 0;
  for (int i = 0; i < listaPlanetas.numPlanetas; i++) {
    if (i != planeta) {
      Fuerza aux = esAtraidoPor(listaPlanetas, planeta, i);
      fuerzaTotal = fuerzaTotal + aux;
    }
  }
  return fuerzaTotal;
}
```

La función suma compone las fuerzas indicadas, y opera componente a componente:

```
Fuerza operator+(Fuerza const& f1, Fuerza const& f2) {
  Fuerza aux;
  aux.x = f1.x + f2.x;
  aux.y = f1.y + f2.y;
  return aux;
}
```

Y la función `esAtraidoPor` se basa en la fórmula de Newton para calcular la fuerza con la que el planeta `i` atrae al `planeta` dado. Para poder calcularla será necesario saber cómo están definidos los planetas. En primer lugar deberá tener una coordenada que defina su posición, velocidad y masa. La velocidad tiene sus dos componentes puesto que estamos considerando un universo plano. En definitiva, tenemos la siguiente definición de tipos:

```
typedef Coordenada Velocidad;
typedef struct Planeta {
  Coordenada posicion;
  Velocidad velocidad;
  double masa;
} Planeta;
```

Para completar la función `esAtraidoPor` es necesario darse cuenta de que la fórmula de Newton da el valor absoluto de la fuerza; nosotros debemos descomponer esa fuerza en sus dos componentes.

```
Fuerza esAtraidoPor(ListaPlanetas& listaPlanetas, int const elMio, int const elOtro)
{
  double difX = listaPlanetas.planetas[elOtro].posicion.x -
                listaPlanetas.planetas[elMio].posicion.x;
  double difY = listaPlanetas.planetas[elOtro].posicion.y -
                listaPlanetas.planetas[elMio].posicion.y;
  double distancia = sqrt(difX*difX + difY*difY);
  double f = constanteGravitacional*
    (listaPlanetas.planetas[elMio].masa * listaPlanetas.planetas[elOtro].masa)/
    (distancia*distancia);
  Fuerza fuerza;
  fuerza.x = f*difX / distancia;
  fuerza.y = f*difY / distancia;
  return fuerza;
}
```

Para completar el procedimiento `actualizaPlaneta` falta el procedimiento `aplicaFuerza` que aplica la fuerza a un planeta en un *intervalo pequeño de tiempo*:

```
void aplicaFuerza(Planeta& planeta, Fuerza const& fuerza) {
  // aceleración = fuerza / masa
  double ax = fuerza.x / planeta.masa;
  double ay = fuerza.y / planeta.masa;
  // E = E_0 + V · T + 1/2 · A · T^2
  planeta.posicion.x = planeta.posicion.x
                       + planeta.velocidad.x*unidadTiempo
                       + 0.5 * ax * unidadTiempo*unidadTiempo;
  planeta.posicion.y = planeta.posicion.y
                       + planeta.velocidad.y*unidadTiempo
                       + 0.5 * ay * unidadTiempo*unidadTiempo;
  // V = V_0 · T
  planeta.velocidad.x = planeta.velocidad.x + ax*unidadTiempo;
  planeta.velocidad.y = planeta.velocidad.y + ay*unidadTiempo;
}
```

Ahora nos centramos en el procedimiento para sacar un fotograma a partir de la configuración de los planetas. Para ello, usamos una matriz de caracteres del tamaño de los fotogramas, pintamos los planetas y después almacenamos la matriz en el fichero. El tamaño del fotograma lo tenemos definido en dos constantes `tamanyoPantallaX` y `tamanyoPantallaY`. Así definimos el tipo `Pantalla` como

```
int const tamanyoPantallaX = 67;
int const tamanyoPantallaY = 13;
```

```
typedef struct {
  char datos[tamanyoPantallaX][tamanyoPantallaY];
} Pantalla;
```

Entonces para sacar una instantánea a la configuración de los planetas tenemos el siguiente procedimiento:

```
void sacaFoto(ListaPlanetas& listaPlanetas, Coordenada const& esquinaSupDer,
              ostream& salida) {
  Pantalla pantalla;
  vaciaPantalla(pantalla);
  for (int i = 0; i < listaPlanetas.numPlanetas; i++) {
    ponEnPantallaPlaneta(listaPlanetas.planetas[i], esquinaSupDer, pantalla);
  }
  salida << pantalla;
}
```

El procedimiento `vaciaPantalla` llena la matriz de espacios:

```
void vaciaPantalla(Pantalla& pantalla) {
  for (int i = 0; i < tamanyoPantallaX; i++) {
    for (int j = 0; j < tamanyoPantallaY; j++) {
      pantalla.datos[i][j] = ' ';
    }
  }
}
```

Hemos de definir el operador `<<` para que *muestre* una foto de la pantalla; no debemos olvidar que, antes del fotograma, debemos decir cuántas veces se repite, en nuestro caso sólo una vez:

```
ostream& operator<<(ostream& out, Pantalla const& pantalla) {
  out << 1 << endl;
  for (int y = tamanyoPantallaY-1; y >= 0; y--) {
    for (int x = 0; x < tamanyoPantallaX; x++) {
      out << pantalla.datos[x][y];
    }
    out << endl;
  }
  return out;
}
```

El procedimiento `ponEnPantallaPlaneta` debe colocar en la matriz el planeta identificado con `'O'`; para ello debemos hacer una proyección de la ventana de visualización en la pantalla:

```
void ponEnPantallaPlaneta(Planeta const& planeta, Coordenada const& esquinaSupDer,
                          Pantalla& pantalla) {
  double factorX = (tamanyoPantallaX-1) / esquinaSupDer.x;
  double factorY = (tamanyoPantallaY-1) / esquinaSupDer.y;
  int posX = (int)(planeta.posicion.x * factorX);
  int posY = (int)(planeta.posicion.y * factorY);
  if (posX < tamanyoPantallaX && posY < tamanyoPantallaY &&
      posX >= 0 && posY >= 0) {
    pantalla.datos[posX][posY] = 'O';
  }
}
```

Para acabar, falta el procedimiento que calcula la configuración inicial de los planetas a partir de los datos que se dan en la línea de comandos. Si observamos el procedimiento `main` podemos ver que los primeros cuatro datos son, respectivamente, el ancho y el alto de la ventana que muestra la simulación, el nombre del archivo de salida y el tiempo (en segundos) de la simulación.

A continuación vamos a establecer los siguientes parámetros: en primer lugar el número de planetas de la simulación y, después, los datos de cada uno de los planetas, por este orden: posición (abcisa y ordenada), masa y velocidad (componentes horizontal y vertical). Como ejemplo, nuestro programa de planetas puede ser invocado así:

```
planetas 1340 260 planetas.txt 30 2 670 130 1200 0 0 670 230 30 400 0
```

En resumen, el procedimiento queda como sigue:

```
void construyeListaPlanetas(ListaPlanetas& listaPlanetas, char* args[]) {
  int numPlanetas = atoi(args[0]);
  listaPlanetas.numPlanetas = numPlanetas;
  int numParametros = 5;
  for (int i = 0; i < listaPlanetas.numPlanetas; i++) {
    listaPlanetas.planetas[i].posicion.x = atof(args[i*numParametros+1]);
    listaPlanetas.planetas[i].posicion.y = atof(args[i*numParametros+2]);
    listaPlanetas.planetas[i].masa = atof(args[i*numParametros+3]);
    listaPlanetas.planetas[i].velocidad.x = atof(args[i*numParametros+4]);
    listaPlanetas.planetas[i].velocidad.y = atof(args[i*numParametros+5]);
  }
}
```

Faltaría dar valor a las constantes utilizadas:

```
double const constanteGravitacional = 1E+4;
double const unidadTiempo = 1E-3;
double const intervaloEntreFotogramas = 1.0/15;
```

Muchos de estos valores dependen esencialmente de la máquina donde se pretende ejecutar la simulación. Cuanto menor sea `unidadTiempo`, más exactos serán los cálculos pero más lenta la simulación; la `constanteGravitacional` establece las unidades en las que se realiza la simulación (en ningún momento hemos hablado ni de segundos ni de metros).

## 4.17 Ajuste de imagen

176

Primero vamos a nombrar las constantes mencionadas en el enunciado para poder olvidar sus valores:

```
int const minGrados = 30;
int const maxGrados = 99;
int const limiteError = 10;
int const filas = 15;
int const columnas = 15;
```

En las discusiones posteriores, abreviaremos `filas` y `columnas` con $\bar{f}$ y $\bar{c}$ respectivamente. También vamos a darle nombre al otro concepto fundamental, la matriz de temperaturas:

```
typedef int Matriz[filas][columnas];
```

Los puntos críticos de este problema son (1) saber si una casilla es errónea y (2) calcular la media de sus vecinas para rectificarla. En el enunciado ya se advierte que los bordes de la matriz son delicados a la hora

de calcular estas propiedades. Tratar de forma separada los bordes provoca una explosión del tamaño del código y un plantel de casos escondidos. Por ejemplo, deberíamos escribir un código que soporte una redefinición arbitraria del tamaño de la matriz; ¿qué pasaría si una de las dimensiones fuera 1? Recorrer con los índices $(i, j)$ el entorno de la posición $(f, c)$ significa dejar que $i$ se mueva en $[f-1, f+1]$ y que $j$ lo haga en $[c-1, c+1]$. Para que los índices $(i, j)$ nunca accedan a posiciones fuera de la matriz, debería restringirse al rango $[0, \bar{f}-1] \times [0, \bar{c}-1]$. Para tener las dos propiedades, basta intersecar los rangos; el resultado es $[\max(0, f-1), \min(f+1, \bar{f}-1)] \times [\max(0, c-1), \min(c+1, \bar{c}-1)]$. Este rango es justamente el que explora la siguiente función, que nos informará de si la casilla (`f`, `c`) es errónea:

```
bool esError (Matriz fuente, int const f, int const c) {
  for (int i = max(0, f-1); i <= min(f+1, filas-1); i++) {
    for (int j = max(0, c-1); j <= min(c+1, columnas-1); j++) {
      if (abs(fuente[i][j] - fuente[f][c]) > limiteError) {
        return true;
      }
    }
  }
  return false;
}
```

Debemos recorrer el mismo rango para calcular la media del entorno de una casilla. Pero hay un detalle: la propia casilla no puede intervenir en los cálculos de dicha media porque el resultado se vería afectado. Por el contrario, en la función `esError` la casilla se contrastaba contra ella misma; pero allí es irrelevante porque no afecta al resultado. No obstante, cabe preguntarse si no hubiera sido mejor haber evitado la propia casilla en `esError` porque así habríamos ahorrado los cómputos

```
abs(fuente[i][j] - fuente[f][c]) > limiteError
```

Sin sacrificar la sencillez del doble bucle anidado, la única alternativa es proteger la condición interna con otra condición, a saber:

```
if (i != j) {
  if (abs(fuente[i][j] - fuente[f][c]) > limiteError) {
    return true;
  }
}
```

Pero esto significa una comprobación adicional en cada vuelta, es decir, bastante más trabajo que el que queremos ahorrar. Esto nos lleva a intentar resolver el cálculo de la media del entorno de una casilla sin evitar a la propia casilla. Nuestra mejor solución es la siguiente:

```
int mediaEntorno(Matriz fuente, int const f, int const c) {
  int vecinos = 0;
  int acumulado = 0;
  for (int i = max(0, f-1); i <= min(f+1, filas-1); i++) {
    for (int j = max(0, c-1); j <= min(c+1, columnas-1); j++) {
      vecinos++;
      acumulado = acumulado + fuente[i][j];
    }
  }
  vecinos--;
  acumulado = acumulado - fuente[f][c];
  return (acumulado + vecinos/2) / vecinos;
}
```

Lo aportado por la propia casilla se quita al salir del bucle. Esto desperdicia varias operaciones (poner para luego quitar) pero nos ahorra una comprobación en cada vuelta del bucle. Con estas dos operaciones básicas el resto del problema se reduce a iteraciones triviales sobre todas las entradas de una matriz. Por último puede llamar la atención la expresión `(acumulado + vecinos/2) / vecinos` para calcular la media; si es el caso es porque no te acuerdas del ejercicio 1.8.

Por ejemplo, la función que corrige una matriz `fuente` y el resultado lo deja en la matriz `destino`:

```
void corrigeMatriz(Matriz fuente, Matriz destino) {
  for (int i = 0; i < filas; i++) {
    for (int j = 0; j < columnas; j++) {
      if (esError(fuente, i, j)) {
        destino[i][j] = mediaEntorno(fuente, i, j);
      } else {
        destino[i][j] = fuente[i][j];
      }
    }
  }
}
```

La misma estructura tiene la función que comprueba si hay algún error en una matriz (es una disyunción de errores: la matriz es errónea en cuanto haya un error):

```
bool tieneErrores(Matriz fuente) {
  for (int i = 0; i < filas; i++) {
    for (int j = 0; j < columnas; j++) {
      if (esError(fuente, i,j)) return true;
    }
  }
  return false;
}
```

Si una matriz es suficientemente irregular, un paso de corrección no basta para eliminar sus errores. Iterar el proceso significa construir una secuencia de matrices $M_i$, donde $M_0$ es la matriz original (una aleatoria si nos regimos por el enunciado) y cada $M_n$ es la corrección de $M_{n-1}$. ¿Significa esto que necesitamos una secuencia de matrices arbitrariamente larga donde almacenar las matrices intermedias? La respuesta es claramente no, porque para calcular una nueva corrección basta con recordar la última matriz. Entonces, podríamos tener dos variables, digamos `matriz1` y `matriz2`. La llamada a la función `corrigeMatriz` dejaría la correción en la segunda variable; inmediatamente después copiaríamos ésta en la primera variable. Esta copia gasta tiempo y es tediosa de escribir en C++. Lo mejor es ir alternado el papel de estas dos variables: en la segunda llamada a `corrigeMatriz` la matriz de entrada sería `matriz2` y la de salida `matriz1`. Para no tener que repetir código lo mejor es juntar estas dos matrices en un array. En una variable `origen` tendremos el índice de la matriz más reciente. Este índice es 0 o 1. El índice de la posición donde dejar la siguiente matriz corregida es simplemente $1 -$ `origen`:

```
int main(void) {
  Matriz matriz[2];
  hazMatrizAleatoria(matriz[0]);
  cout << matriz[0] << endl;
  int origen = 0;
  while (tieneErrores(matriz[origen])) {
    corrigeMatriz(matriz[origen], matriz[1-origen]);
```

```
      cout << matriz[1-origen] << endl;
      origen = 1-origen;
    }
  }
```

Por completar, damos también las funciones para generar una matriz aleatoria y para escribir la matriz a un *stream*.

```
void hazMatrizAleatoria(Matriz matriz) {
  for (int i = 0; i < filas; i++) {
    for (int j = 0; j < columnas; j++) {
      matriz[i][j] = minGrados + rand() % (maxGrados - minGrados + 1);
    }
  }
}
ostream& operator<<(ostream& out, Matriz matriz) {
  for (int i = 0; i < filas; i++) {
    for (int j = 0; j < columnas; j++) {
      out << matriz[i][j] << ' ';
    }
    out << endl;
  }
  return out;
}
```

## 4.22 Triángulo de Pascal

184

Vamos a resolver este ejercicio de distintas maneras, empezando con una muy sencilla, hasta llegar a un programa eficiente.

### 4.22.1 Solución directa

Al rotar el triángulo de Pascal como pide el enunciado,

$$\begin{array}{ccccccc}
\binom{0}{0} & \binom{1}{1} & \binom{2}{2} & \binom{3}{3} & \cdots & \binom{j}{j} & \cdots \\
\binom{1}{0} & \binom{2}{1} & \binom{3}{2} & \binom{4}{3} & \cdots & \binom{1+j}{j} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\binom{i}{0} & \binom{i+1}{1} & \binom{i+2}{2} & \binom{i+3}{3} & \cdots & \binom{i+j}{j} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{array}$$

se observa que el elemento situado en la fila i-ésima y en la columna j-ésima es $\binom{i+j}{j}$, para $i, j \geq 0$. Como consecuencia, se tiene un primer algoritmo que consiste en escribir los elementos de la matriz correspondiente, de `NumF` filas y `NumC` columnas, que reflejan el tamaño de la pantalla,

```
int const NumF = 15;
int const NumC = 15;
int main() {
  for (int f = 0; f <= NumF; f++) {
    ⟨Escribir la fila i-ésima⟩
  }
}
```

donde ⟨*Escribir la fila i-ésima*⟩ consiste en lo siguiente:

```
for (int c = 0; c <= NumC; c++) {
  ⟨Escribir el ejemento i-j-ésimo⟩
}
⟨Pasar a la línea siguiente⟩
```

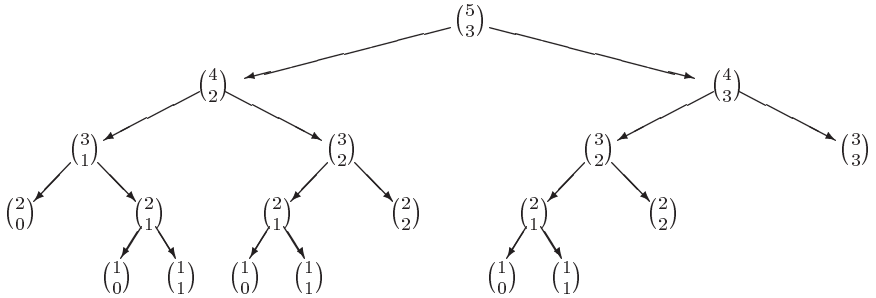
Ahora bien, ⟨*Escribir el ejemento i-j-ésimo*⟩ equivale a

```
cout << (nComb(f+c, c)%2 == 1 ?  '*' :  '  ');
```

donde `nComb(m, n)`$= \binom{m}{n} = \frac{m!}{n!(m-n)!}$ para $0 \leq n \leq m$, y puede hallarse iterativa o recursivamente.

### 4.22.2 Solución tabulando los números combinatorios

Puede objetarse un inconveniente al algoritmo propuesto: el cálculo independiente de cada elemento $\binom{m}{n}$ presenta redundancias, tanto si se calcula recursivamente,



como en su versión iterativa:

$$\ldots, \frac{(m-2)\ldots(m-n)}{(n-1)!}, \frac{(m-1)\ldots(m-n)}{n!}, \ldots$$

$$\ldots, \frac{(m-1)\ldots(m-n+1)}{(n-1)!}, \frac{m\ldots(m-n+1)}{n!}, \ldots$$

Una alternativa es tabular los números combinatorios $\binom{i+j}{j}$, para $0 \leq i \leq$ `numF`$, 0 \leq j \leq$ `numC`. Llamando *Comb* a la tabla mencionada, tenemos lo siguiente,

$$Comb[i+j, j] \in \mathbb{N}, \text{ para } (i,j) \in \{0,\ldots,\texttt{numF}\} \times \{0,\ldots,\texttt{numC}\}$$

que, en C++ se define así:

```
typedef int Tabla[NumF+1][NumC+1];
```

Ahora, la fila superior (la número cero) de esta tabla se rellena fácilmente así:

```
Tabla tabla;
for (int j = 0; j <= NumC; j++) {
  tabla[0][j] = 1;
}
```

y las siguientes se completan como sigue:

```
for (int i = 1; i <= NumF; i++) {
  tabla[i][0] = 1;
  for (int j = 1; j <= NumC; j++) {
```

```
      tabla[i][j] = tabla[i][j-1] + tabla[i-1][j];
    }
  }
```

Una vez construida la tabla, reproducir la figura resulta casi igual que en la versión anterior: basta con cambiar en ella cada llamada `nComb(m, n)`, que calcula un número combinatorio, por la consulta de la tabla para encontrar su valor: `tabla[m][n]`.

#### 4.22.3 Solución tabulada y usando la paridad de los números combinatorios

Un inconveniente que tiene ahora la solución anterior es el del desbordamiento de los números que surgen en las expresiones, debido al rápido crecimiento de los números combinatorios. Sin embargo, para resolver este problema no es necesario recurrir a dichos números: puesto que finalmente sólo estamos interesados en la paridad de los números $\binom{m}{n}$, basta con calcular sólo este dato y registrarlo en una tabla de paridades,

$$Paridad[i+j, j] \in \{0,1\} \text{ para } (i,j) \in \{0,\ldots,\texttt{numF}\} \times \{0,\ldots,\texttt{numC}\}$$

La tabla se puede definir como la anterior, con componentes enteras, aunque sólo se emplearán ceros y unos, para representar respectivamente a los valores par e impar. Y rellenar esta tabla se puede llevar a cabo sencillamente así:

```
Tabla tablaPar;
for (int j = 0; j <= NumC; j++) {
  tablaPar[0][j] = 1;
}
for (int i = 1; i <= NumF; i++) {
  tablaPar[i][0] = 1;
  for (int j = 1; j <= NumC; j++) {
    tablaPar[i][j] = sumaParidad(tablaPar[i][j-1], tablaPar[i-1][j]);
  }
}
```

donde la operación `sumaParidad` se describe así:

```
unsigned int sumaParidad(int const m, const int n) {
 return (m == 0 ?  n :  1-n);
}
```

Una vez rellena la tabla con ceros y unos, reproducir la figura es aún más sencillo que antes, ya que los elementos son ahora sólo ceros y unos:

```
for (int f = 0; f <= NumF; f++) {
  for (int c = 0; c <= NumC; c++) {
    cout << (tablaPar[f][c] == 1 ?  '*' :  '  ');
  }
  cout << endl;
}
```

#### 4.22.4 Solución avanzando por filas

Finalmente, observamos que, para construir cada fila de la tabla, basta con mantener la fila anterior, pudiendo evitarse así el gasto de memoria de la tabla completa, que se reduce al de una única fila de paridades:

$$Paridad[j] \in \{0,1\} \; para \; j \in \{0,\ldots,\texttt{numC}\}$$

esto es,

```
typedef int Vector[NumC+1];
```

La primera fila se rellena con una hilera inicial de unos,

```
Vector vector;
for (int j = 0; j <= NumC; j++) {
  vector[j] = 1;
  cout << '*';
}
cout << endl;
```

y se muestra en la pantalla al mismo tiempo la línea de asteriscos correspondiente. Y ahora, cada una de las líneas siguientes (números 1 a `numF`) se construye y representa así:

```
for (int f = 1; f <= NumF; f++) {
  cout << '*'; // El primer asterisco
  for (int c = 1; c <= NumC; c++) {
    vector[c] = sumaParidad(vector[c-1], vector[c]);
    cout << (vector[c] == 1 ?  '*' :  '  ');
  }
  cout << endl;
}
```

## 4.24 Segmentador de oraciones

187

La solución que damos aquí se limita a cubrir los patrones tipográficos que se enumeran en el enunciado, pero está organizada de forma que sea fácil añadir algunos casos más.

Para hacer la exposición lo menos confusa y palabrera posible empezamos introduciendo una forma de escribir patrones tipográficos. Afortunadamente, no hemos tenido que inventar esta notación, sino que nos hemos limitado a adaptar y reducir la notación usual para *expresiones regulares*.

La representación de un carácter en `este tipo de letra` hace referencia a ese carácter; por ejemplo, "`?`" es el signo de interrogación cerrada. Nos referiremos a los caracteres sin representación visible, como el espacio, con símbolos especiales; el signo "␣" es el espacio, "`\t`", el tabulador y "`\n`", el final de línea. Un rango de caracteres se representa dando sus extremos separados por un guión; así, podemos referirnos a todas las letras mayúsculas escribiendo simplemente "`A–Z`". Un conjunto de caracteres es una secuencia de caracteres o rangos encerrados entre corchetes; se necesita este concepto para poder decir, por ejemplo, que los caracteres que comienzan una frase son [`A–Z¡¿"`]. Finalmente, para referirnos a una secuencia de 0 o más caracteres extraídos de un conjunto $C$, escribiremos $C\star$; así, podremos escribir que, tras un punto, viene [␣`\t\n`]$\star$ para referirnos a una secuencia quizá vacía de espacios, tabuladores o finales de línea.

Con esta notación, podemos decir que ocurre un cambio de oración cuando en el texto aparece [`.!?`]['"]$\star$[␣`\t\n`][␣`\t\n`]$\star$[`A–Z¡¿'"`]. Este patrón tipográfico se puede leer grupo a grupo de la siguiente manera: primero debe aparecer uno de los caracteres [`.!?`], que levantará la alarma para indicar que una frase puede estar a punto de acabarse. Si luego aparecen unas comillas, la alarma seguirá en pie, porque es lícito transponer comillas y puntos finales. A continuación necesitamos un carácter no visible que actúe como separador, que se puede convertir en un ristra arbitrariamente larga. Para terminar, cualquiera de los caracteres [`A–Z¡¿'"`] confirmará que la oración anterior ha acabado porque son el comienzo de una nueva.

Definiremos una función predicado para cada uno de los conjuntos de caracteres anteriores:

```
bool esFinOracion(char const c) {
  return c == '.'  || c == '?'  || c == '!';
}
bool esProlongacionFinOracion(char const c) {
  return c == '"' || c == '\'';
}
bool esSeparacion(char const c) {
  return c == ' ' || c == '\t' || c == '\n';
}
bool esComienzoOracion(char const c) {
  return ('A' <= c && c <= 'Z') || c == '¡' || c == '¿' || c == '"' || c == '\'';
}
```

Para hilar las cuatro funciones anteriores hay que reflexionar sobre la situación en que estamos según nos vamos encontrando los caracteres que forman el patrón tipográfico de separación. En un principio estaremos `enOracion`. Será cuando encontremos [.!?] cuando pasaremos a estar `enFinOracion`. Mientras estamos `enFinOracion` podemos encontrar caracteres de tres conjuntos diferentes; si son del conjunto ['"] o [.!?], seguiremos potencialmente estando `enFinOracion`; si son del conjunto [␣\t\n], habremos avanzado un paso, porque estaremos `enSeparacion`; finalmente, cualquier otro carácter nos niega el derecho a acabar la frase, por lo que volveríamos a estar (seguiríamos) `enOracion`. Todos los [␣\t\n] que vengan a continuación nos dejan `enSeparacion`; cualquier otro carácter nos mete `enOracion`, ya sea un [A–Z¡¿'"], lo que nos permitiría cambiar de oración, o ya sea cualquier otro carácter, que nos impediría el cambio. En definitiva, durante del análisis de nuestro texto podemos estar en tres situaciones:

```
typedef enum Estado {
  enOracion, enFinOracion, enSeparacion
} Estado;
```

El párrafo anterior se plasma en forma de código como sigue:

```
void segmenta(istream& in, ostream& out) {
  Buffer buffer;
  iniciaBuffer(buffer);
  Estado esta = enSeparacion;
  bool huboAntes = false;
  for(;;) {
    char const c = in.get();
    if (in.eof()) break;
    switch (esta) {
    case enOracion:
      out << c;
      if (esFinOracion(c)) esta = enFinOracion;
      break;
    case enFinOracion:
      if (esSeparacion(c)) {
        meteBuffer(buffer, c);
        esta = enSeparacion;
      } else {
        out << c;
        if (!esProlongacionFinOracion(c) || !esFinOracion(c)) esta = enOracion;
      }
      break;
```

```
      case enSeparacion:
        if (esSeparacion(c)) {
          meteBuffer(buffer, c);
        } else {
          if (esComienzoOracion(c)) {
            if (huboAntes) out << "</s>";
            else huboAntes = true;
            vaciaBuffer(buffer, out);
            out << "<s>";
          } else {
            vaciaBuffer(buffer, out);
          }
          out << c;
          esta = enOracion;
        }
        break;
      }
    }
    if (huboAntes) out << "</s>";
    vaciaBuffer(buffer, out);
  }
```

Hay dos detalles de este código que no hemos explicado. El primero es simple: empezamos suponiendo que estamos en un separador, pero el comienzo de la primera oración, cuando no `huboAntes` ninguna otra, no puede generar una marca `</s>`.

El otro detalle es más delicado porque cuida que se conserve la separación original entre frases; exactamente los mismos caracteres que antes aparecían entre dos oraciones, ahora deben aparecer entre el `</s>` que cierra una y el `<s>` que abre la siguiente. Dedicamos un *buffer* para esa tarea; metemos en él todos los separadores que encontramos entre dos frases; hay que descargarlo en cuanto se acaben los separadores, hayamos o no encontrado una separación de frase.

Podríamos definir el buffer como un simple array de caracteres. Un tamaño fijo suficientemente grande nos bastará para casi cualquier texto. Por supuesto, si el tamaño es fijo, siempre nos pueden dar un texto con una separación lo suficientemente larga como para que no la podamos almacenar. En un intento de evitar esta limitación al máximo, vamos a definir un buffer especializado para nuestro problema. Por el papel que en la práctica desempeña cada uno de estos los tres caracteres [␣\t\n], una separación suele estar formada por unas pocas subsecuencias en donde se repite un solo carácter; por ejemplo, una separación podría estar formada por un final de línea, 3 tabuladores y 4 blancos, en ese orden. Para aprovechar estas repeticiones, en nuestro buffer podremos guardar caracteres junto con un número de repeticiones:

```
typedef struct CharRepetido {
  char caracter;
  int veces;
} CharRepetido;
```

```
int const longitudBuffer = 10;
typedef struct Buffer {
  int usado;
  CharRepetido caracteres[longitudBuffer];
} Buffer;
```

A la hora de añadir un carácter nuevo hay que vigilar si estamos repitiendo el último:

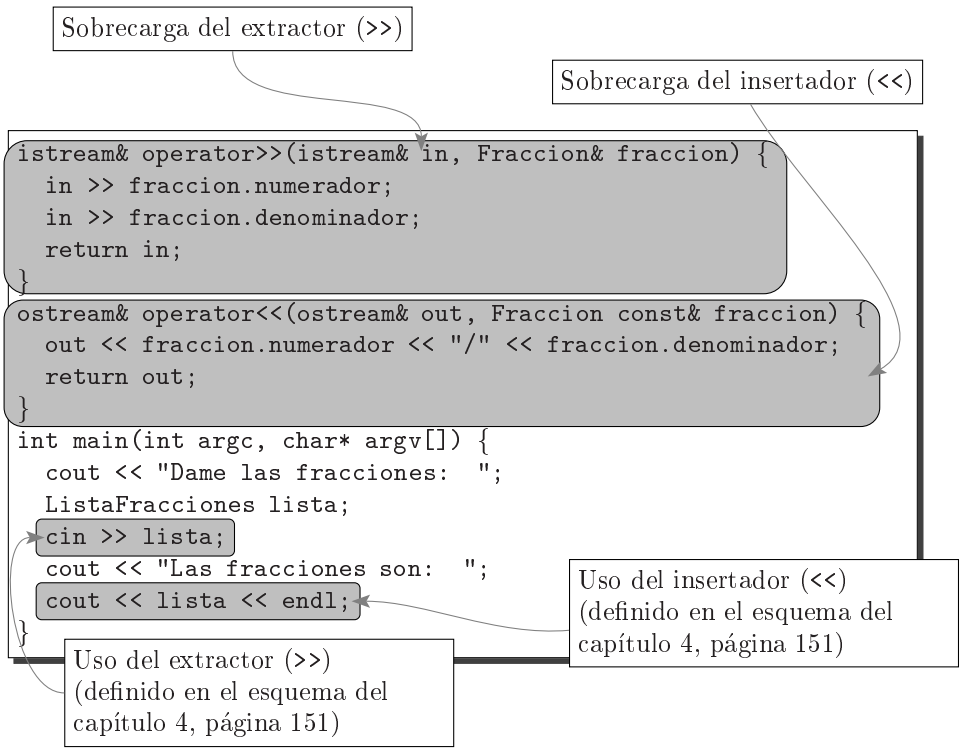
```
void meteBuffer(Buffer& buffer, char c) {
  if (buffer.usado > 0 && buffer.caracteres[buffer.usado-1].caracter == c) {
    buffer.caracteres[buffer.usado-1].veces++;
  } else if (buffer.usado < longitudBuffer) {
    buffer.caracteres[buffer.usado].caracter = c;
    buffer.caracteres[buffer.usado].veces = 1;
    buffer.usado++;
  } else {
    cerr << "El buffer es demasido pequeño para guardar el espaciado.";
    exit(1);
  }
}
```

Terminamos esta solución con el código que imprime los contenidos de un buffer y lo deja vacío para subsiguientes inserciones:

```
void vaciaBuffer(Buffer& buffer, ostream& out) {
  for (int i = 0; i < buffer.usado; i++) {
    for (int j = 0; j < buffer.caracteres[i].veces; j++) {
      out << buffer.caracteres[i].caracter;
    }
  }
  buffer.usado = 0;
}
```

# 5 Capítulo CINCO

## Miscelánea de tipos



Este capítulo abunda en la definición y manipulación de los tipos. Con la teoría que se resumió en el capítulo anterior, se pueden hacer muchos de los ejercicios propuestos en éste; simplemente se han traído aquí porque son más largos o complejos. Otros en cambio propician los conceptos nuevos que se exponen a continuación: registros con variantes y orden superior. No se suelen necesitar en programas sencillos o cortos; pero en los programas un poco más complejos siempre hay un hueco para ellos y, entonces, simplifican la tarea enormemente.

## 5.1 Las uniones son registros variantes

Las *uniones* deben usarse cuando queremos representar un domino que tiene objetos con distintas formas posibles. Una unión se declara de forma idéntica a una estructura, salvo que está etiquetada con la palabra reservada `union` en vez de `struct`.

En una unión sólo uno de los campos contiene un valor correcto. Los demás tienen valores potencialmente incorrectos, y no se deberían usar, ni mucho menos modificar porque entonces podríamos perder el valor del campo correcto. Internamente, todos los campos de una unión comparten el mismo espacio de memoria; por eso, la modificación de un campo altera los demás.

En rigor, no hay forma de saber qué campo de una unión es el que tiene un valor correcto. Hay que mantener esa información en otro lugar. Se suele utilizar la siguiente estructura, que llamaremos *registro con variantes*. Primero, se define un tipo enumeración con tantos valores como posibles alternativas. Luego, se define un tipo registro con dos campos: uno, que llamaremos *campo discriminante*, para la enumeración y otro para la unión.

Veamos un ejemplo. Supongamos que queremos manejar figuras geométricas planas sencillas: puntos, segmentos, triángulos y circunferencias. Primero definimos el tipo enumerado con las cuatro figuras:

```
typedef enum TiposFiguras {
  tipoPunto, tipoSegmento, tipoTriangulo, tipoCircunferencia
} TiposFiguras;
```

Luego definimos un tipo para representar cada una de las posibles figuras:

```
typedef struct Punto {
  double x;
  double y;
} Punto;
typedef struct Segmento {
  Punto p;
  Punto q;
} Segmento;
typedef struct Triangulo {
  Punto vertices[3];
} Triangulo;
typedef struct Circunferencia {
  Punto centro;
  double radio;
} Circunferencia;
```

Finalmente, utilizamos la unión de los elementos anteriores para construir el tipo `Figura`, que queda como sigue:

```
typedef struct Figura {
  TiposFiguras tipo;
  union {
    Punto punto;
    Segmento segmento;
    Triangulo triangulo;
    Circunferencia circunferencia;
  } figura;
} Figura;
```

No se llega a dar nombre de tipo a la unión porque no hace falta en ningún otro sitio para poder declarar el campo `figura` del registro `Figura`. Esto es normal en las uniones, justo al contrario que en las estructuras, porque casi siempre se utilizan dentro de un registro con variantes.

Se accede a los campos de una unión de la misma forma que a los de un registro: con un punto. Para dar un poco más de cuerpo a este ejemplo, implementaremos una función que rota una figura `angulo` radianes alrededor del origen. La función es trivial si contamos con operaciones que rotan cada una de las posibles figuras:

```
Figura rotada(Figura const & figura, double const angulo) {
  Figura figuraRotada;
  figuraRotada.tipo = figura.tipo;
  switch (figura.tipo) {
  case tipoPunto:
    figuraRotada.figura.punto = rotada(figura.figura.punto, angulo);
    break;
  case tipoSegmento:
    figuraRotada.figura.segmento = rotada(figura.figura.segmento, angulo);
    break;
  case tipoTriangulo:
    figuraRotada.figura.triangulo = rotada(figura.figura.triangulo, angulo);
    break;
  case tipoCircunferencia:
    figuraRotada.figura.circunferencia
      = rotada(figura.figura.circunferencia, angulo);
    break;
  }
  return figuraRotada;
}
```

Hay muchos elementos que se llaman figura: el parámetro, el tipo, el segundo campo del tipo; no hay ambigüedad posible aunque cabe un poco de confusión la primera vez que nos enfrentamos a un código similar. La estructura de esta función, organizada alrededor de una instrucción `switch`, es la vertiente en código del registro con variantes. Aprovechándonos de la sobrecarga, hemos llamado `rotada` tanto a esta función que trabaja sobre `Figuras` como a las que operan sobre tipos particulares de figuras.

Si seguimos delegando, el resto de funciones de rotación también se implementan fácilmente:

```
Segmento rotada(Segmento const & segmento, double const angulo) {
  Segmento segmentoRotado;
  segmentoRotado.p = rotada(segmento.p, angulo);
  segmentoRotado.q = rotada(segmento.q, angulo);
```

```
  return segmentoRotado;
}
Triangulo rotada(Triangulo const & triangulo, double const angulo) {
  Triangulo trianguloRotado;
  for (int i = 0; i < 3; i++) {
    trianguloRotado.vertices[i] = rotada(triangulo.vertices[i], angulo);
  }
  return trianguloRotado;
}
Circunferencia rotada(Circunferencia const& circunferencia, double const angulo) {
  Circunferencia circunferenciaRotada;
  circunferenciaRotada.centro = rotada(circunferencia.centro, angulo);
  circunferenciaRotada.radio  = circunferencia.radio;
  return circunferenciaRotada;
}
```

La rotación de un punto, la única operación que exige un poco de ingenio, se va a resolver en el ejercicio 5.17.

## 5.2 Orden superior

Cuando usamos un subprograma, los parámetros de entrada condicionan su comportamiento, al pedirle que trabaje con unos ciertos datos. Desde el otro lado, ante la tesitura de cómo definir un subprograma, los parámetros son la forma de generalizar su comportamiento, de ofrecer algo útil más allá de la situación particular que nos incumbe. Hasta ahora parece que el proceso de abstracción siempre se resuelve con datos, pero en ciertas ocasiones también involucra código. Ocurrirá cuando una actividad sea independiente de la forma en que se realice cierto cálculo.

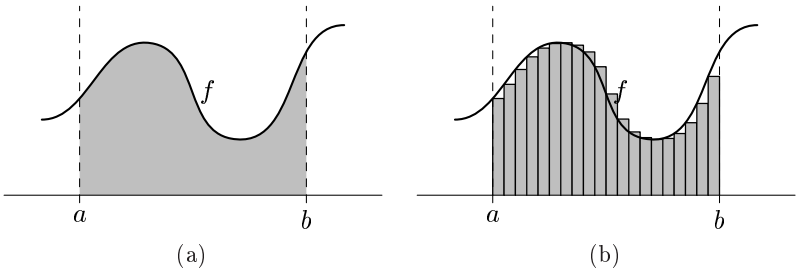



Figura 5.1: La integral definida (a) y su aproximación por cajas (b)

Un ejemplo es la integral definida de una función. En términos matemáticos, la integral definida de la función $f$ en el rango $(a, b)$ se denota con $\int_a^b f$. El símbolo $\int$ es el nombre de un *subprograma* que recibe tres parámetros (una función $f$, y un par de límites $a$ y $b$) y devuelve el valor del área encerrada bajo la curva de la función (figura 5.1a). Una correspondencia informática sencilla sería la construcción de una función, llamémosla `integral`, que aproxima la integral con el método que se insinúa en la figura 5.1b. Si queremos conseguir una buena abstracción, `integral` tendrá que ofrecer como parámetros de entrada tanto los rangos de integración como la propia función por integrar.

Tras esta discusión, está claro que necesitamos tratar a los subprogramas como valores en sí mismos, que se puedan almacenar en variables, pasar a otros subprogramas, etc. Pero ni parámetros ni variables se pueden declarar sin hacer referencia a un tipo; necesitamos pues una forma de definir el tipo de un

subprograma. No se puede meter a todos los subprogramas en el mismo saco de un solo tipo, porque claramente una acción no se puede usar en el lugar de una función, ni una función con tres argumentos en donde necesitamos una con dos, ni una acción que recibe un `string` cuando queremos hacer una llamada con un `int`, etc. De forma similar a los registros, lo que caracteriza a un subprograma es el número y tipo de los parámetros, y el tipo de lo que devuelve si acaso es una función. Todos los subprogramas que tengan la misma organización de parámetros estarán en el mismo tipo. Así, `cos` y `sin` tienen el mismo tipo porque reciben y devuelven un `double`; pero `atan2` tiene un tipo distinto porque, aunque devuelve un `double`, recibe dos `double`s.

Aunque hay tipos muy distintos para los subprogramas, todos se englobarán bajo un mismo apelativo de tipos *funcionales*.

Igual que con los registros, se puede definir directamente una variable o un parámetro con un tipo funcional. Pero lo más normal es dar previamente un nombre al tipo que nos interesa. Como siempre, se resuelve anteponiendo un mágico `typedef` a lo que, salvo un detalle, sería una cabecera correcta de función. Veamos el tipo de las funciones que reciben un `double` y devuelven un `double` (que llamaremos `RenR` porque en términos matemáticos se dice que van de $\mathbb{R}$ en $\mathbb{R}$):

```
typedef double (*RenR)(double);
```

El detalle es el "`*`" que acompaña al nombre de la función. Es necesario porque las variables y parámetros con un tipo funcional, no contienen directamente una función, sino que apuntan al comienzo de su código. Ahora ya podemos declarar o definir variables que *contengan* funciones,

```
RenR grafica = cos;
```

cambiarlas para que contengan otras funciones distintas,

```
grafica = sin;
```

o llamarlas para que calculen, con el código que contienen, sobre los parámetros que les demos:

```
cout << "El valor de la función en 0 es " << grafica(0);
```

Obsérvese que estamos abstrayendo el código concreto que se ejecuta. Pero no se olvida ningún detalle que pudiera permitir una actividad incorrecta. Por ejemplo, como sabemos que `grafica` sólo puede contener funciones de un argumento, la siguiente llamada es incorrecta y un compilador de C++ no la aceptará

```
cout << "El valor de la función en (0,0) es " << grafica(0,0);
```

Finalmente, estamos en disposición de resolver las integrales que nos han llevado hasta aquí.

```
double integral(RenR f, double const a, double const b) {
  int const muestreos = 100;
  double const base = (b-a) / muestreos;
  double integral = 0;
  for (int i = 0; i < muestreos; i++) integral += f(a + i*base) * base;
  return integral;
}
```

## 5.1 Lógica e incertidumbre

257

En buena lógica, las afirmaciones no sólo pueden ser ciertas o falsas, sino que también pueden resultar inciertas. En este ejercicio se propone definir un nuevo tipo de datos, `BoolPlus`, cuyos objetos puedan tomar tres valores (`falso`, `talVez` y `cierto`), así como operaciones para su manejo.

***Tipo de datos*** Define el tipo de datos descrito.

***Conjunción*** Se define en esta lógica la operación de conjunción, `&&`, con arreglo a la siguiente tabla de verdad:

| && | falso | talVez | cierto |
|---|---|---|---|
| falso | falso | falso | falso |
| talVez | falso | talVez | talVez |
| cierto | falso | talVez | cierto |

Define una función que responda al operador definido en esa tabla.

***Disyunción*** Análogamente, se define una operación `||`, que intenta capturar el significado de la disyunción y que se describe como sigue:

| \|\| | falso | talVez | cierto |
|---|---|---|---|
| falso | falso | talVez | cierto |
| talVez | talVez | talVez | cierto |
| cierto | cierto | cierto | cierto |

En este caso, se opta por representarla con una tabla de $3 \times 3$. Define un tipo de datos para dicha tabla y desarrolla un subprograma que la rellene adecuadamente.

***Negación*** Finalmente, se necesita una operación, `!`, correspondiente a la negación lógica, descrita así:

| a | !a |
|---|---|
| falso | cierto |
| talVez | talVez |
| cierto | falso |

En vez de una función para sustituir a este operador, se prefiere definir un subprograma que invierta el valor de una variable de tipo `BoolPlus`, según el operador `!` descrito. Desarrolla ese subprograma.

***Uso de las operaciones definidas*** Considerando que se han declarado las variables `a`, `b`, `c`, `d` de tipo `BoolPlus`, y suponiendo que tienen asignado un valor inicial, expresa una o varias instrucciones que produzcan un efecto equivalente a la asignación siguiente:

```
a = !((b && c) || d)
```

## 5.2 El juego del ahorcado

253 258

Hay varios juegos que básicamente consisten en que un jugador tiene que adivinar una palabra o una frase en un determinado número de intentos. El *ahorcado* es uno de ellos, aunque por su nombre parezca un juego más siniestro. El funcionamiento de un programa que implemente este juego (véase la figura 5.2) debe ser el siguiente:

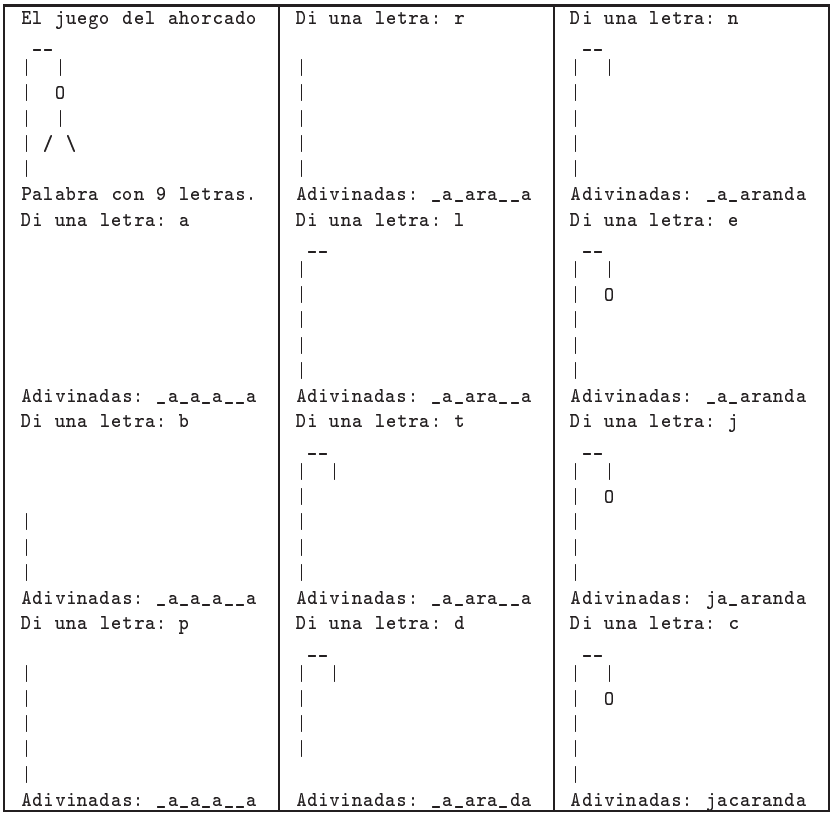



Figura 5.2: Una partida al juego del ahorcado

- La computadora *piensa* una palabra y muestra de forma gráfica el número de letras que tiene, por ejemplo un carácter "_" por cada una.
- El jugador propone una letra.
- Si la letra forma parte de la palabra, aquélla se hará visible pero, en caso contrario, se añadirá un trazo al dibujo del ahorcado.
- El jugador seguirá proponiendo letras y el programa actuará en consecuencia hasta que todos los caracteres de la palabra se hagan visibles (gana el jugador) o hasta que se complete el dibujo del ahorcado (pierde el jugador).

***Para este enunciado*** Consulta las pistas 5.2a y 5.2b.

## 53 Guerra de las galaxias

253 262

Partiendo de la Tierra, el comandante Solo debe recorrer los planetas del sistema solar en busca de la princesa Leia. Para ello, dispone de una relación de los planetas que le quedan por visitar. Posee asimismo una tabla que, dado un par de planetas, ofrece su distancia.

***Tipos de datos*** Define en C++ tipos apropiados para relacionar los planetas, el conjunto de los planetas pendientes de inspeccionar y la matriz de distancias. (Véase la pista 5.3a.)

***Carga de la matriz de distancias*** Se tiene un archivo en disco con las distancias entre los planetas, dispuestas en forma de matriz de números reales. Define un procedimiento que rellene la tabla correspondiente.

***¿Cuál es el planeta más próximo?*** Define una función que averigüe el planeta más cercano a uno dado, consultando la tabla de distancias.

***Ídem entre los no visitados*** Define una función que averigüe el planeta más cercano a uno dado, de entre los que quedan por visitar.

***Inspección de un planeta*** Aceptando que Leia se encuentra en uno cualquiera de los planetas no visitados con igual probabilidad, define una función aleatoria que resulte ser cierta con probabilidad $\frac{1}{resto}$, siendo *resto* el número de planetas sin rastrear.

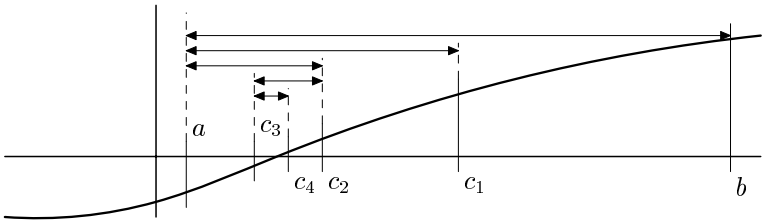
***¡A por ella!*** Escribe un programa que simule el viaje del comandante hasta hallar a la princesa, dando un informe apropiado de su recorrido.

## 5.4 Ceros de una función

El teorema de Bolzano dice así:

> Toda función $f : \mathbb{R} \to \mathbb{R}$, continua y monótona en un intervalo, y que en sus extremos toma valores de distinto signo, tiene en dicho intervalo al menos un valor $x$ que hace que $f(x) = 0$.

Una técnica empleada a menudo para encontrar dicha $x$ es el *método de la bisección*, que se explica seguidamente siguiendo esta figura:




A partir de un intervalo $[a, b]$ se calcula el punto medio $c_1$, y se comprueba si ese valor anula la función; si es así la búsqueda ha finalizado, en caso contrario, la función en ese punto tendrá el mismo signo que uno de los extremos del intervalo, y además estará más cercana al cero. El punto medio ($c_1$ en el ejemplo) sustituirá a ese extremo (el derecho) en el intervalo. Sucesivamente, se procede del mismo modo hasta encontrar un valor aceptable como cero de la función. En la gráfica, el intervalo evoluciona así:

$$[a, b] \to [a, c_1] \to [a, c_2] \to [c_3, c_2] \to [c_3, c_4] \to \ldots,$$

hasta hacerse tan pequeño como deseemos. Entonces, su punto medio será un cero de $f$ con un error menor que la mitad del ancho que tiene dicho intervalo.

Lo que se pide es escribir este método de la bisección como una función de orden superior, que recibe como parámetro la función (supuestamente en las condiciones del teorema) y los extremos del intervalo de partida, y halla un cero de dicha función en ese intervalo.

Como aplicación, utiliza el método para el caso particular de las funciones propuestas en el ejercicio 2.29.

***Un poco de historia*** El funcionamiento de este ejercicio se basa en el teorema de Bolzano. Bernhard Bolzano (1781–1848) fue un filósofo, teólogo y matemático checo.

## 5.5 Derivadas y desarrollos en serie

265

Supongamos definida una función $f : \mathbb{R} \to \mathbb{R}$, derivable en todo $\mathbb{R}$.

***Derivada*** Define una función que dé una aproximación de la derivada de $f$ en un cierto punto. Esto es, una función $d : (\mathbb{R} \to \mathbb{R}, \mathbb{R}) \to \mathbb{R}$ tal que $d(f, x) \simeq f'(x) = \frac{df}{dx}$.

***Derivada enésima*** Suponiendo que $f$ es infinitamente derivable (o sea, todas sus derivadas existen) en todo $\mathbb{R}$, define igualmente una función para la derivada enésima, denotada $f^{(n}(x)$, de una función $f$ en un punto $x$, así:

$$\begin{array}{ll} f^{(0}(x) & = f(x) \\ f^{(n}(x) & \simeq (f^{(n-1)})'(x), \text{ para } n \geq 1 \end{array}$$

***Desarrollo en serie de Taylor*** Escribe finalmente un programa que acumule los primeros $k$ términos del desarrollo de Brook Taylor (1685–1731) de $f$ alrededor de $a$,

$$f(x) \simeq f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2!} + \ldots + \frac{f^{(k}(a)(x-a)^k}{k!}$$

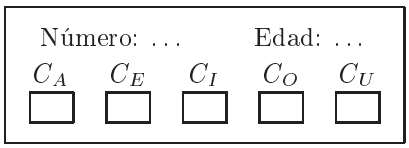
para $k \in \mathbb{N}, a \in \mathbb{R}$ dados, en un formato similar al anterior.

(En este apartado, la precisión se verá afectada fuertemente al avanzar el orden de la derivada.)

## 5.6 Un sistema electoral utópico

En el sistema electoral de Dusitania, los ciudadanos pueden distribuir su voto entre varios candidatos, reflejando así sus preferencias con gran precisión, tal como expresa el lema "Un hombre, cien votos", característico de aquel pequeño país. Además, pueden emitirse votos de castigo (negativos).

De ahí que, para votar, los dusitanos rellenen un *array*, repartiendo en sus componentes (correspondientes a los cinco candidatos de siempre: $C_A$, $C_E$, $C_I$, $C_O$, $C_U$) un máximo de cien puntos, ya sean positivos o negativos (cuenta su valor absoluto). El modelo de tarjeta que se entrega es el siguiente:




Lleva un número entero identificativo, propio de cada persona; oficialmente, este dato tiene por objeto invalidar todo intento de votar repetidamente, aunque también hay quien desconfía del anonimato. Para controlar el peligro de votar más de una vez, existe un censo con los números identificativos de los ciudadanos con capacidad de voto.

Por otra parte, si bien todos los votos cuentan, no lo hacen por igual, sino de un modo proporcional a la edad de sus votantes, por lo cual resulta necesario este dato a la hora de recontar un voto en el marcador.

El mecanismo electoral establece la lista de ciudadanos que han votado (inicialmente vacía), y pone además a cero una tabla de marcadores al objeto de contar los puntos. Entonces, el recuento de las votaciones consiste en lo siguiente,

```
while (⟨queden votos efectuados⟩) {
  ⟨Leer una ficha⟩
  if (⟨el número está en el censo⟩) {
    ⟨Tachar el número del censo⟩
    if (⟨el voto es válido⟩) {
      ⟨Agregar puntuación (puntos * edad) a cada candidato⟩
    }
  }
}
```

donde la validez de un voto consiste en que no se han usado más de los cien puntos permitidos.

Tras el recuento, se deberán ofrecer los resultados (en tanto por ciento) obtenidos por cada candidato.

***Tipos de datos*** Define tipos de datos apropiados para registrar cada voto, el censo y la tabla de los totales.

***Lectura de tarjetas*** Escribe un subprograma de lectura de una tarjeta de un archivo de texto, cuyos datos se suponen registrados en una misma línea.

***Validez de un voto*** Escribe una función lógica que indique si un voto es válido o no, en el sentido de hacer uso de, a lo más, cien puntos.

***Cómputo de un voto*** Escribe un subprograma útil para incluir en el marcador los puntos correspondientes a un voto, supuestamente válido.

***Escrutinio*** Escribe un programa que desarrolle el proceso del escrutinio, extrayendo los datos de un archivo externo que registra una tarjeta en cada línea, como ya se ha indicado. Durante el recuento, se informará del titular de cada tarjeta, de su voto, de su edad y de la legalidad y validez del voto.

## 5.7 Sobre el juego del ajedrez

El ajedrez se juega sobre un tablero cuadrado, de $8 \times 8$ casillas o escaques. Durante una partida, cada casilla puede estar vacía u ocupada por una pieza, como se muestra en la figura 5.3. Las piezas que

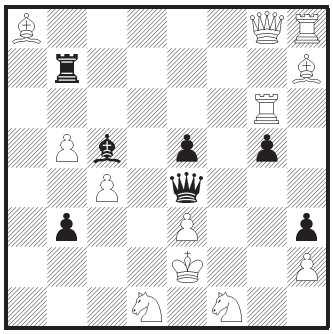


Figura 5.3: Tablero de ajedrez, durante una partida

intervienen son rey, reina, torre, caballo, alfil y peón, y pueden ser blancas o negras, según el jugador que las mueva. No tendremos en cuenta el color de los escaques.

***Tipos de datos*** Define tipos de datos para manejar los (colores de los) jugadores, las piezas, el contenido de las casillas y el tablero.

***Número de piezas*** Define un subprograma que averigüe qué jugador tiene más piezas en juego en el tablero, en un momento determinado. Por ejemplo, en el tablero de la figura 5.3 se trata de las blancas, ya que tienen doce piezas sobre el tablero, mientras que las negras sólo tienen siete.

***Conjunto de piezas*** Define un subprograma que, dado un tablero y un jugador, averigüe el conjunto de piezas que tiene ese jugador en el tablero, sin importar si están repetidas o no. En el tablero de la figura 5.3 por ejemplo, el conjunto de las piezas negras en juego es el siguiente:

{rey, torre, alfil, peón}

***Almacenamiento en el disco*** En cada momento, la situación del tablero se puede almacenar en un archivo de texto. Una buena forma es guardar el tablero y la posición de cada pieza en un archivo de $8 \times 8$ caracteres. Las casillas en blanco se consignan mediante caracteres en blanco, y las demás mediante los caracteres `R`, `Q`, `T`, `C` y `A` respectivamente, con mayúscula para las blancas y con minúscula para las negras. Por ejemplo, el tablero de la figura 5.3 se corresponde con un archivo como el siguiente (donde se han colocado puntos en vez de blancos para mostrar mejor cada carácter):

```
A.....RT
.t.....A
......T.
.Pa.p.p.
..P.r...
.p..P..p
....Q..P
...C.C..
```

Escribe un subprograma que, dado el nombre del archivo, cargue un tablero con la situación indicada por ese archivo.

## 58 Un traductor de Morse

266

Se desea confeccionar un programa traductor de textos entre el alfabeto sajón y el de *Morse*:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | .- | B | -... | C | -.-. | D | -.. |
| E | . | F | ..-. | G | --. | H | .... |
| I | .. | J | .--- | K | -.- | L | .-.. |
| M | -- | N | -. | O | --- | P | .--. |
| Q | --.- | R | .-. | S | ... | T | - |
| U | ..- | V | ...- | W | .-- | X | -..- |
| Y | -.-- | Z | --.. | | | | |

***Tabla: su tipo de datos*** Describe la estructura de una tabla que refleje la correspondencia entre cada letra mayúscula y su código Morse.

***Traducción*** Desarrolla un programa que lea unas líneas de la entrada estándar y traduzca a Morse las letras mayúsculas, ignorando los demás caracteres, y separando los códigos de las letras en Morse con un espacio.

***Ejemplo*** La traducción de la frase *ALFABETO MORSE* quedaría así:

.-␣.-..␣..-.␣.-␣-...␣.␣-␣---␣--␣---␣.-.␣...␣.

Hemos utilizado el signo ␣ para indicar los espacios en blanco que aparecen separando las letras.

***Traducción inversa*** Para la traducción inversa, se presenta el problema de que no es posible dar una tabla cuyos índices sean códigos en Morse. Se puede, sin embargo, buscar la letra correspondiente a un código recorriendo secuencialmente la tabla.

Siguiendo ese método, define una función que proporcione la letra mayúscula correspondiente a un código Morse dado según una tabla.

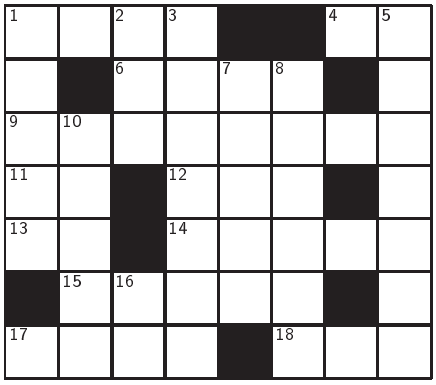
***Un poco de historia*** A Samuel F. B. Morse (1791–1872) lo que más le gustaba era pintar. Con el objetivo de conseguir independencia económica para poder dedicar todo su tiempo a la pintura, Samuel emprendió una serie de proyectos. En 1836 con la invención del telégrafo consiguió su objetivo.

El telégrafo utiliza la relación entre magnetismo y electricidad para producir y enviar diferentes señales. El código Morse es simplemente una codificación del alfabeto utilizando estas señales. El primer mensaje enviado por telégrafo de Washington a Baltimore fue *What hath God wrought?* (¿Qué ha hecho Dios?).

## 5.9 Crucigramas

253

Te proponemos desarrollar un programa para resolver crucigramas como el que se plantea en la figura 5.4. Nuestro crucigrama tiene forma rectangular de dimensiones fijas $M \times N$. El número de casillas negras



**Horizontales:**
1 Dios egipcio. 4 Nota musical. 6 Pelo blanco. 9 Color. 11 Dios egipcio. 12 Mamífero volador. 13 Preposición. 14 Triste color. 15 Abandonar. 17 Fruto de la zarza. 18 Época.

**Verticales:**
1 Anudaré. 2 Ave con la que se hace un delicioso paté. 3 Fruta y color. 5 Con enojo. 7 Como la nieve. 8 Contento. 10 Ordeno y ... 16 lacisum atoN.

Figura 5.4: Un crucigrama

es arbitrario, así como el de aquéllas que están numeradas para ofrecer las definiciones correspondientes, horizontales o verticales.

El programa que vamos a desarrollar es algo más sencillo: en vez de dar la descripción de las palabras, se ofrecen éstas directamente:

**Horizontales:**
1 Amón. 4 Re. 6 Cana. 9 Amarillo. 11 Ra. 12 Ave. 13 En. 14 Negro. 15 Dejar. 17 Mora. 18 Era.

**Verticales:**
1 Ataré. 2 Oca. 3 Naranja. 5 Enojosa. 7 Nívea. 8 Alegre. 10 Mando 16 Er.

Por tanto, la solución es casi trivial: partiendo de un crucigrama con las casillas negras ya marcadas y las demás en blanco, podemos resolverlo en dos pasos:

- Primero, podemos empezar por rellenar las palabras horizontales, pudiendo quedar algunas casillas en blanco. Opcionalmente, en este paso se puede comprobar que las palabras dadas en las pistas tienen la longitud adecuada al hueco libre en el crucigrama.

- Seguidamente, rellenamos las verticales. En este paso se puede incluir la misma comprobación (de longitud) que en el anterior. Además, opcionalmente se puede comprobar que las letras ya escritas coinciden con las nuevas, por si alguna de las pistas horizontales contradice alguna vertical.

Los dos pasos anteriores se pueden ver en la figura 5.5.

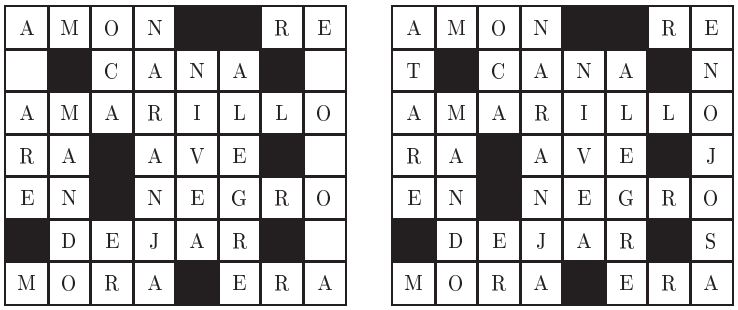



Figura 5.5: Resolución de un crucigrama

***Tipos de datos*** Para manejar el crucigrama, necesitamos definir una estructura de datos adecuada: una matriz, en la que ir escribiendo la solución.

Además, se va a necesitar una operación de carga de la misma, que la prepara para rellenar, poniendo en negro los cuadros negros y en blanco los blancos. Para esta operación, se supone que la relación de cuadros negros está almacenada en un archivo externo de texto, cuyas filas contienen las posiciones en que hay cuadros negros. (Véase la pista 5.9a.)

***Resolución*** Las pistas para la resolución han de cargarse de un archivo de disco, donde han sido previamente almacenadas. Convenimos en que se han guardado en sendos archivos con arreglo al siguiente formato:

```
1 1 'AMON'
2 3 'CANA'
3 1 'AMARILLO'
...
```

Se pide desarrollar un subprograma adecuado para rellenar el crucigrama siguiendo las pistas horizontales, y otro para las verticales. En ambos se puede incluir la opción de verificar cada palabra que se registra en la tabla.

Finalmente, se pide integrar los apartados anteriores en un programa que resuelve un crucigrama a partir de las palabras dadas como definiciones en los archivos de datos.

***Variante*** En vez de que las definiciones consistan directamente en las palabras que deben rellenarse, cada definición puede consistir en una palabra en un idioma (inglés, por ejemplo), y la respuesta sería la palabra en el otro (como el castellano).

En este caso, se debe incluir una estructura adecuada para manejar el diccionario, junto con las operaciones necesarias (carga de disco, consulta, etc.)

## 5.10 Parchís

253

¿Quién no ha jugado alguna vez al parchís? En este ejercicio llevamos este popular juego de mesa a la computadora.

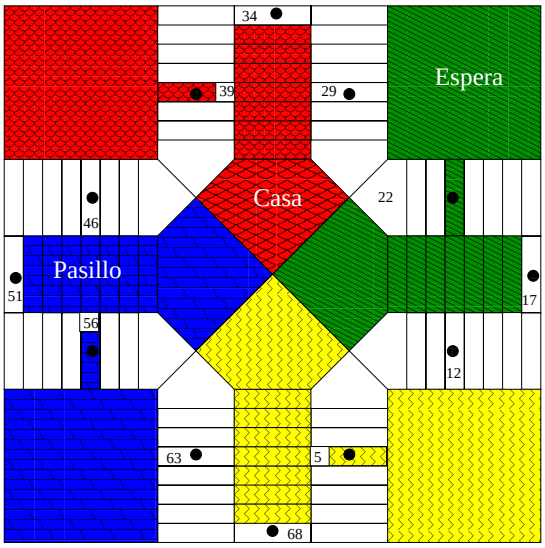



Figura 5.6: Un tablero de parchís

***Juego*** Realiza un programa que permita jugar al parchís. El parchís es un juego para cuatro jugadores. Para esas ocasiones en que falta alguien, el programa permitirá elegir el número de jugadores *humanos* y se hará cargo del resto. En el caso extremo, podremos hacer que la computadora controle todos los jugadores, y observaremos lo bien que se lo pasa ella sola. (Véase la pista 5.10a.)

***Un poco de historia*** El parchís que se conoce en Occidente es una simplificación, una versión para niños, de un juego indio llamado *parchisi*. Hay documentos que acreditan la existencia del parchisi desde el siglo IV de nuestra era. El tablero de juego indio consiste en un paño de tela de bellos colores en forma de cruz. En los palacios de Agra y de Allahabad existen grandes tableros con casillas de mármol rojo y blanco.

## 5.11 El juego de la vida

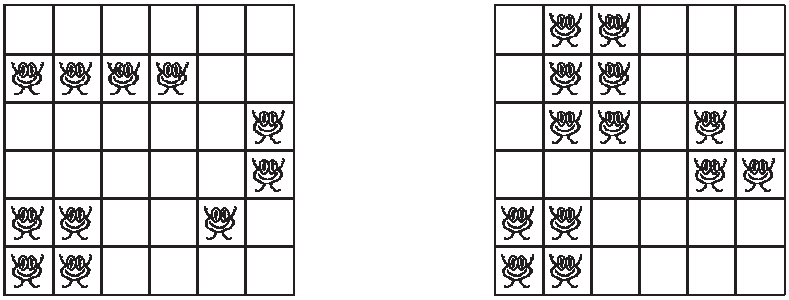
253

Vamos a desarrollar un programa de simulación, que juega a un bonito solitario inventado en 1970 por John H. Conway (1937–). Dicho juego simula la evolución de una colonia de bichos que vive en un mundo cuadriculado cuyas casillas, en un momento dado, pueden estar vacías o habitadas por un solo ser.
La evolución de nuestra población de un instante a otro está sujeta a las siguientes reglas:

**Nacimiento** En una casilla vacía que tenga exactamente tres casillas vecinas habitadas, nace un nuevo bicho. Consideramos vecinas a las ocho casillas circundantes a una dada.

**Supervivencia** Cada bicho que tenga dos o tres bichos vecinos sobrevive y pasa a la generación siguiente.

**Muerte** Cada bicho que sólo tenga un bicho vecino o no tenga ninguno, muere de soledad; cada bicho con cuatro bichos vecinos o más, muere por superpoblación.



(a) (b)

Figura 5.7: Una colonia de bichos (a) y su evolución en una generación (b)

Es importante observar que todos los nacimientos o muertes ocurren simultáneamente.

Por ejemplo, la colonia de bichos de la figura 5.7a se convierte, en el paso de una generación, en la colonia de la figura 5.7b.

Dependiendo de la configuración de partida, la población va experimentando cambios curiosos que, en ciertos aspectos, remedan a menudo el comportamiento de la vida real: a veces se extingue; otras crece sin cesar; otras adopta un comportamiento cíclico...

***Tipos de datos*** Define tipos de datos adecuados para representar y manejar los conceptos siguientes:

- El estado de una casilla en particular, que puede estar vacía u ocupada.
- La posición de una casilla.
- El mundo, de dimensiones fijas, con el estado de la colonia de bichos en una generación dada.
- El número de vecinos que rodean una casilla en cierto momento.

***Vecindario*** Escribe una función que averigüe el número de bichos vivos vecinos de una casilla dada en un mundo concreto. (Véase la pista 5.11a.)

***Un paso en la evolución*** Desarrolla un subprograma que, a partir del mundo en un instante dado, halle el estado del mundo en la siguiente generación. (Véase la pista 5.11b.)

***Fotografía del mundo*** Desarrolla un subprograma que muestre el mundo en la pantalla, usando para ello los caracteres disponibles.

***La creación, a la carta*** Para evitar introducir la generación inicial cada vez que se desea contemplar la evolución del juego, sería conveniente disponer de varias configuraciones iniciales grabadas en archivos de disco. Se pide desarrollar un subprograma de carga que cree un mundo con el estado descrito en un archivo. (Véase la pista 5.11c.)

***Evolución del mundo*** Con todas esas piezas, ya sólo falta el programa que nos ofrece la película de cómo evoluciona una población inicial dada.

***Otros mundos*** Las variantes y mejoras que el programa admite son múltiples. Seguidamente se incluyen algunas, aunque es posible imaginar otras muchas posibilidades:

- Se puede considerar que el mundo tiene estructura *toroidal,* esto es, que sus lados inferior y superior están conectados, siendo vecinas las casillas de la primera y última filas, y lo mismo con las de sus lados izquierdo y derecho.

- Es también posible modificar el programa para que avance solo, mostrando los distintos instantes a intervalos fijos de tiempo.
- También puede interesar permitir que el espectador acelere la evolución, indicando el número de pasos que se desea avanzar.

Se sugiere ahora modificar el programa del apartado anterior, tal como explican estas variantes.

***Bibliografía*** Este juego se hizo popular gracias a Martin Gardner, que divulgó por primera vez este juego en su columna "Mathematical Games", *Scientific American*, en octubre de 1970. En [Gar88], el autor recopila y amplía este artículo, junto con otros muchos, de matemática recreativa. [Wol96] ofrece un tratamiento más profundo de los autómatas celulares, a los que pertenece este juego.

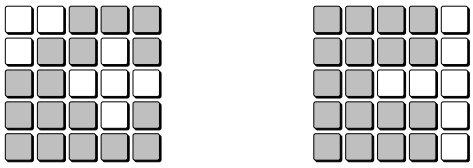
También en Internet abundan las páginas dedicadas a este juego, ofreciendo simulaciones *online*, el código fuente en distintos lenguajes, etc. Puedes consultar la dirección `http://directory.google.com/Top/Computers/Artificial_Life/Cellular_Automata/Conway's_Game_Of_Life/` o si lo prefieres esta otra: `http://dir.yahoo.com/Science/Artificial_Life/Cellular_Automata/Conway_s_Game_of_Life/`.

## 5.12 Lights Out 254

Se plantea desarrollar distintos tipos de datos y subprogramas útiles para manejar un rompecabezas conocido como *luces fuera*. Veamos primero los rudimentos de ese juego y luego pasaremos a los ejercicios que se piden.

**Reglamento del juego**

El juego se desarrolla en una matriz de $M \times N$ casillas. Durante el juego, cada una de esas casillas puede estar apagada o encendida. Cuando se pulsa una de ellas, cambia su estado (se enciende si estaba apagada, o se apaga si estaba encendida), y al mismo tiempo cambia el de sus cuatro casillas vecinas (encima, debajo, izquierda y derecha). El objetivo del juego es apagar todas las casillas. Por ejemplo, si se parte del tablero de la figura 5.8a y se pulsan



(a) (b)

Figura 5.8: Un tablero (a) y su evolución tras varias pulsaciones (b)

consecutivamente las casillas $(1,1)$, $(2,5)$ y $(4,5)$, se llega a la situación de la figura 5.8b. Obsérvese que la casilla $(3,5)$ ha quedado como al principio, debido a que ha cambiado su estado dos veces.

***Tipos de datos*** Define tipos de datos para indicar el estado de una casilla, así como el del tablero.

***Carga*** La posición de partida se halla almacenada en un archivo de disco. Define el tipo adecuado de ese archivo así como un subprograma adecuado para cargar el tablero con la información del disco.

***Generación al azar*** También puede interesar empezar un juego con un tablero generado al azar. Desarrolla un subprograma que genere aleatoriamente un tablero, donde cada casilla está encendida con una cierta probabilidad $p$, y vacía en caso contrario.

***Pulsación*** Ahora, una jugada consiste en pulsar una casilla, y su efecto es cambiar algunas luces del tablero como indican las reglas del juego. (Véase la pista 5.12a.)

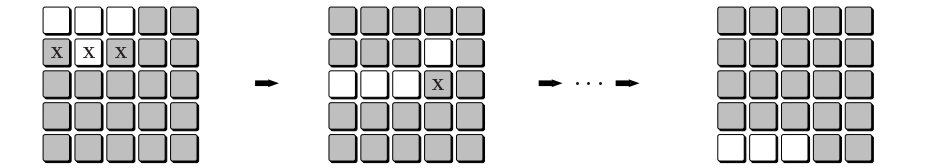
***¿Está ya resuelto?*** Define un subprograma que averigüe si un tablero está resuelto; es decir, si tiene todas sus luces apagadas. (Véase la pista 5.12b.)

***Pues a resolverlo a mano*** Desarrolla un programa que plantee un tablero inicial y efectúe repetidamente los movimientos que indique el usuario, hasta que el rompecabezas esté resuelto.
El tablero inicial puede ser cargado del disco o generado aleatoriamente, a gusto del usuario.

***Que lo resuelva la máquina tanteando*** Ahora trataremos de que el programa resuelva el rompecabezas él solo. Para ello, proponemos el siguiente método para tantear una solución:

- Empezamos pulsando, en la primera fila, unas casillas sí y otras no, aleatoriamente (cada una de ellas con probabilidad $\frac{1}{2}$).
- Luego pulsamos las casillas de la segunda fila necesarias para apagar las de la primera.
- Luego pulsamos las casillas de la tercera fila necesarias para apagar las de la segunda.
- ...
- Finalmente, pulsamos las casillas de la última fila necesarias para apagar la penúltima.

El método resuelve el rompecabezas si la última fila queda completamente apagada.
Un ejemplo del funcionamiento de este algoritmo se muestra a continuación, donde se han marcando con **X** las casillas que hay que pulsar para apagar las luces de la fila anterior.




Escribe un subprograma que simule este método.

***Resolución sistemática*** En el método anterior, las casillas pulsadas en la primera fila se han elegido al azar, pero siempre es posible tantear las $2^N$ combinaciones posibles de casillas pulsadas en la primera fila y, con cada una de ellas, tantear el procedimiento anterior. Por ejemplo, para un tablero de $M \times 5$, tenemos 32 posibles *pulsaciones* de la primera fila, que son todos los posibles subconjuntos de las luces $\{1, 2, 3, 4, 5\}$ de la primera fila:

| | | | |
|---|---|---|---|
| $\{\}$ | $\{1\}$ | $\{2\}$ | $\{1, 2\}$ |
| $\{3\}$ | $\{1, 3\}$ | $\{2, 3\}$ | $\{1, 2, 3\}$ |
| $\{4\}$ | $\{1, 4\}$ | $\{2, 4\}$ | $\{1, 2, 4\}$ |
| $\{3, 4\}$ | $\{1, 3, 4\}$ | $\{2, 3, 4\}$ | $\{1, 2, 3, 4\}$ |
| $\{5\}$ | $\{1, 5\}$ | $\{2, 5\}$ | $\{1, 2, 5\}$ |
| $\{3, 5\}$ | $\{1, 3, 5\}$ | $\{2, 3, 5\}$ | $\{1, 2, 3, 5\}$ |
| $\{4, 5\}$ | $\{1, 4, 5\}$ | $\{2, 4, 5\}$ | $\{1, 2, 4, 5\}$ |
| $\{3, 4, 5\}$ | $\{1, 3, 4, 5\}$ | $\{2, 3, 4, 5\}$ | $\{1, 2, 3, 4, 5\}$ |

Escribe un subprograma que genere cada una de las $2^N$ posibles combinaciones de teclas pulsadas en la primera fila y, con cada una de esas filas generadas, tantea si el método del apartado anterior resuelve el rompecabezas. Por supuesto, sería conveniente que la secuencia de tanteos terminara cuando uno de ellos tenga éxito.

***Bibliografía*** El rompecabezas *Lights Out* ha despertado pasiones. Si a ti también te atrae, puedes empezar a degustar las páginas de Ken Barr (`http://www.mit.edu/~kbarr/lo/`) y de Matthew Baker (`http://www.haar.clara.co.uk/Lights/`). En ellas puedes encontrar información diversa sobre este juego: su origen, programas que lo simulan, artículos sobre el mismo, etc. En [MP01] se analiza este rompecabezas desde dos puntos de vista, reflejados mutuamente: usando el sentido común y mediante las no menos eficaces matemáticas. Este artículo y unos pocos recursos útiles se pueden recoger en Internet: `http://dalila.sip.ucm.es/~cpareja/lo`.

## 5 13 Pequeña teoría de la música

En esta ocasión, vamos a aprender los rudimentos de música necesarios para interpretar partituras, o al menos para poder adiestrar a nuestra computadora y que sea ella quien nos deleite.

### 5.13 1 Las notas musicales y la escala diatónica

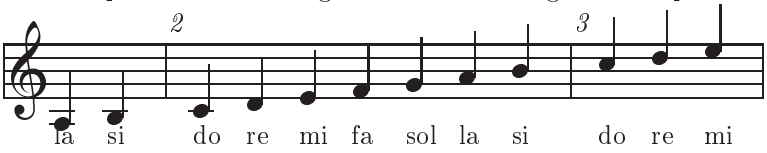
Empecemos por aprender las notas musicales:

```
typedef enum Nota {
  ut, re, mi, fa, sol, la, si, silencio
} Nota;
```

A la nota *do* la hemos llamado *ut* (que fue su primer nombre) para distinguirla de la palabra reservada `do`. También hemos añadido el silencio, como una nota musical sin sonido pero que tendrá una duración.

En el teclado del piano, estas notas se corresponden con las teclas blancas señaladas:


y en el pentagrama son las que se muestran seguidamente en el segundo compás:




Esta secuencia de notas se llama *escala diatónica* de *do* mayor, y en ella están ausentes las notas correspondientes a las teclas negras. De hecho, si numeramos las notas blancas según su posición en la escala, los huecos de las teclas negras son aún más notorios. Replanteamos la enumeración anterior de las notas de acuerdo con su posición:

```
typedef enum Nota {
  ut = 1, re = 3, mi = 5, fa = 6, sol = 8, la = 10, si = 12, silencio = 0
} Nota;
```

### 5.13 2 Los intervalos, las alteraciones y la escala cromática

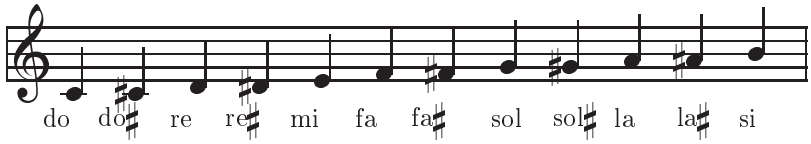
La *distancia* entre dos notas se llama *intervalo*. El intervalo entre dos teclas consecutivas se llama *semitono*. Pero obsérvese que el intervalo entre dos notas consecutivas (teclas blancas) es desigual:

- Hay un semitono entre *mi* y *fa*, y también entre *si* y el *do* de la siguiente escala.
- Hay un tono entre *do* y *re*, entre *re* y *mi*, entre *fa* y *sol*, entre *sol* y *la* y entre *la* y *si*.

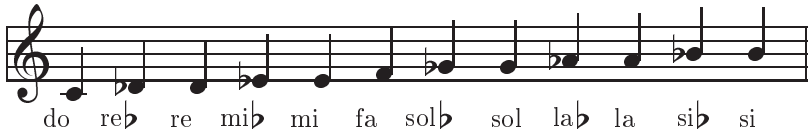
Cualquier nota puede alterarse medio tono descendentemente con un bemol (♭), dejarse como está (sonido natural) o alterarse ascendentemente con un sostenido (♯):

```
typedef enum Alteracion {
  bemol = -1, natural = 0, sostenido = +1
} Alteracion;
```

de forma que ya tenemos también las teclas negras: por ejemplo, la tecla negra que hay entre *do* y *re* es *do*♯ (leído *do sostenido*), o también *re*♭ (leído *re bemol*). También la nota *mi* natural suena igual que *fa*♭, y *mi*♯ suena igual que *fa* natural, etc. Con las teclas blancas y las negras se forma la escala cromática, que tiene doce teclas distintas:




Esta escala se puede escribir, equivalentemente, de la siguiente manera:




Con la numeración de las alteraciones, la posición de una tecla cualquiera se alcanza simplemente sumando los valores de su nota y su alteración.

Pero un teclado es bastante más amplio, e incluye unas siete escalas. En resumidas cuentas, una altura musical viene dada por el nombre de la nota, su posible alteración, y la escala en que está:

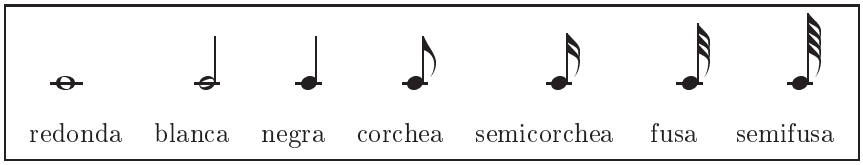
```
typedef struct Altura {
  Nota nota;
  Alteracion alteracion;
  int escala;
} Altura;
```

Así, una nota con su posible alteración y el número de la escala en que está, determina la posición de la tecla correspondiente:

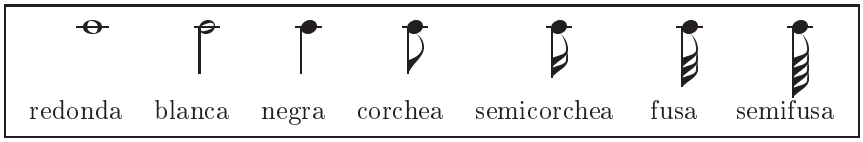
```
int numTeclaCromatica(Altura const altura) {
  return 12*(altura.escala-1) + altura.nota + altura.alteracion;
}
```

### 5.13 3 El tiempo y el ritmo: las figuras

Vamos ahora con las duraciones y el ritmo. Las figuras expresan el tiempo que suena una nota. De mayor a menor duración, las figuras son redonda, blanca, negra, corchea, semicorchea, fusa y semifusa. En notación musical, estas figuras se escriben así,




o también así, con la plica hacia abajo,




que se pueden codificar adecuadamente por enumeración:

```
typedef enum Figura {
  redonda, blanca, negra, corchea, semicorchea, fusa, semifusa
} Figura;
```

Cada una dura el doble de la siguiente: una redonda es igual a dos blancas, o cuatro negras, u ocho corcheas, o dieciséis semicorcheas, o treinta y dos fusas, o sesenta y cuatro semifusas. Por tanto, replanteamos la enumeración anterior del siguiente modo:

```
typedef enum Figura {
  redonda = 256, blanca = 128, negra = 64,
  corchea = 32, semicorchea = 16, fusa = 8, semifusa = 4
} Figura;
```

Tomando como unidad la semifusa, que es la figura más breve, cualquier duración se puede expresar con un número (entero) de semifusas:

```
int duracion; // número de semifusas
```

Pero en la lectura musical por humanos, esto sería tan incómodo como llevar todo el dinero en céntimos sea cual sea la cantidad de dinero que llevemos: es más cómodo usar las monedas (figuras) mayores, agrupándolas siempre que se pueda. En resumidas cuentas, se acostumbra a poner cualquier duración como una secuencia de figuras, y la duración se obtiene directamente sumando las duraciones de estas figuras, que son sencillamente sus valores asociados.

### 5.13.4 Pulsación = altura + duración

Ahora, hay que replantear las pulsaciones, que han de recoger la altura del sonido y su duración:

```
typedef struct Pulsacion {
  Altura altura;
  int    duracion;
} Pulsacion;
```

de forma que el tipo de datos `Altura` definido antes no va a ser necesario por estar contenido completamente en una `Pulsacion`, y la función `numTeclaCromatica` definida antes, se redefine trivialmente, cambiando sólo el tipo de su parámetro:

```
int pulsoAtecla(Pulsacion const pulso) {
  return 12*(pulso.altura.escala-1)
         + pulso.altura.nota + pulso.altura.alteracion;
}
```

### 5.13.5 Un poco de acústica

Hemos hablado de intervalos como *distancias* entre notas, pero eso no es del todo correcto: cada intervalo viene caracterizado en realidad por una *proporción* entre las frecuencias de las notas. El intervalo canónico es el de *octava* (de un *do* al *do* siguiente, por ejemplo, o de un *la* al *la* siguiente) y denota que la frecuencia de la nota más aguda es el *doble* que la de la nota grave.

Por consiguiente, aceptando que todos los intervalos son iguales, el intervalo de semitono (de *do* a $do\sharp$, o de *fa* a $sol\flat$) ha de ser $\sqrt[12]{2}$, y si de una nota a otra se va en $n$ pasos de semitono, las frecuencias guardan una proporción de $(\sqrt[12]{2})^n$. En resumidas cuentas, la frecuencia de la nota $n$-ésima es la frecuencia de la nota cero multiplidada por $2^{n/12}$.

```
float frecuencia(int const numTecla) {
  float const frecuenciaNotaCero = 442/pow(2, 46.0/12.0);
  return frecuenciaNotaCero * pow(2, numTecla/12.0);
}
```

La frecuencia de la nota cero se ha elegido así para ajustar los sonidos a la afinación oficial en nuestros días: el *la*-4 (que es la nota número 46 del teclado) se afina a $442Hz$; esto es, `frecuenciaNotaCero` ha de verificar lo siguiente:

$$\texttt{frecuenciaNotaCero} \cdot 2^{46/12} = 442$$

Como hemos visto, para hallar la frecuencia hemos partido de la posición correspondiente a una `Pulsacion`, que viene dada por la función `pulsoAtecla`.

### 5.13.6 *El* tempo

La velocidad de lectura de una pieza se indica al principio, mediante uno de los siguientes términos italianos: *largo*, *adagio*, *andante*, *allegro*, *presto*, *vivace*. Su significado suele describirse con alguna imprecisión: lento, despacio, tranquilo pero sin lentitud, deprisa, rápido y muy rápido. Nosotros caracterizamos esos *tempos* indicando el tiempo que va a durar la nota más breve, la semifusa: respectivamente 75, 40, 20, 10, 4 y 2 milisegundos. Los enumeramos y asociamos con su duración:

```
typedef enum Tempo {
  largo = 75, adagio = 40, andante = 20, allegro = 10, presto = 4, vivace = 2
} Tempo;
```

### 5.13.7 *El solfeo = la lectura musical*

El proceso de lectura musical se resume mediante el siguiente esquema:

```
⟨Elegir la partitura⟩
⟨Leer el tempo, y ajustar la semifusa⟩
// Lectura de las pulsaciones, así:
while (⟨no se termine la partitura⟩) {
  ⟨Leer la pulsación (altura y tiempo del sonido)⟩
  ⟨Pronunciar dicha pulsación, según la altura y duración leídas⟩
}
```

Refinando esta descripción tenemos el siguiente programa, útil para solfear:

```
// Elección de la partitura:
ifstream partitura("preludio.dat");
// Lectura del tempo:
string cadenaVelocidad;
partitura >> cadenaVelocidad;
```

```
Tempo const tiempoSemifusa = cadenaAtempo(cadenaVelocidad);
// Lectura de las pulsaciones:
string cadenaCars;
while (partitura >> cadenaCars) {
  // Traducir el string a notación musical:
  Pulsacion const pulso = caracteresApulsacion(cadenaCars);
  ⟨Recitar la Pulsacion pulso⟩
}
```

El seudocódigo ⟨*Recitar la* `Pulsacion pulso`⟩ consiste simplemente en escribir en el monitor la información de cada pulsación. Pero no estamos interesados en solfear, sino en cantar. En efecto, el solfeo no genera música, sino sólo una lectura monótona. Por eso, en vez de *recitar* las notas leídas, es más agradable *entonarlas*, al `tempo` adecuado según nuestra unidad (la semifusa):

```
entonarTecla(pulso, tiempoSemifusa);
```

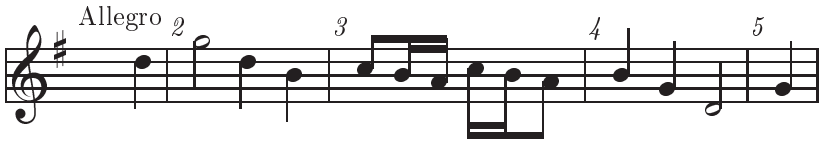
siendo `entonarTecla` el subprograma siguiente:

```
void entonarTecla(Pulsacion const pulso, int const tpoUnidad) {
  int const numTecla = pulsoAtecla(pulso);
  int const tiempo = tpoUnidad*pulso.duracion;
  double const frec = frecuencia(numTecla);
  sound((int)frec);
  delay(tiempo);
}
```

### 5.13.8 Codificación digital de la notación musical

El almacenamiento de las partituras en el disco para su posterior lectura por nuestro programa no puede codificarse usando la notación musical, sino que requiere una codificación *secuencial*, usando los recursos de la computadora.

Por ejemplo, el fragmento de partitura siguiente,




puede codificarse así en un archivo de disco:

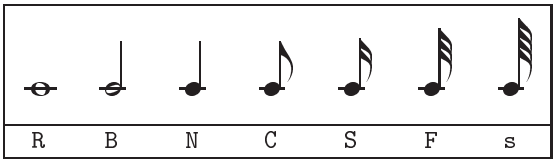
```
Allegro
R-6N S-6B R-6N s-5N D-6C s-5S L-5S
D-6S s-5S L-5C s-5N S-5N R-5B S-5N
```

donde cada palabra (cadena de letras consecutiva) representa una pulsación, así: la primera letra es la nota; el carácter siguiente su posible alteración,

| do | re | mi | fa | sol | la | si |
|---|---|---|---|---|---|---|
| D | R | M | F | S | L | s |

| ♭ | natural | ♯ |
|---|---|---|
| b | - | # |

luego viene un número que determina la escala, y finalmente las figuras (pueden ser varias letras):




Estos detalles pueden confinarse en funciones que traducen una cadena de caracteres (como "Allegro") en un tempo (como `Allegro`), y caracteres (como '`R`', '`6`' y '`N`'), desprovistos de significado musical, en una nota, una escala y una duración (como el *re* de la sexta escala, con una duración de una negra) respectivamente. Juntando estas conversiones, se tiene la función `caracteresApulsacion`, que ya hemos usado.

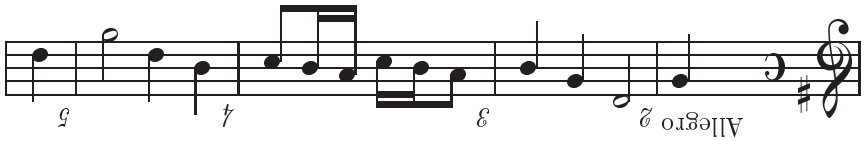
### 5.13.9 Y fin

Ahora, sólo queda presentar nuestro programa a los mejores certámenes internacionales... y lanzarlo al *estrellato*.

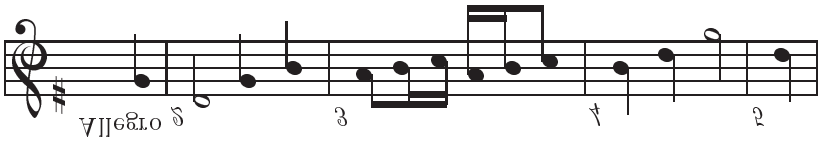
### 5.13.10 Reflexiones músicales

Cualquier estudiante de música ha invertido alguna vez una partitura, tal vez por error. Quién sabe si no le gustará incluso más así que en su forma original.

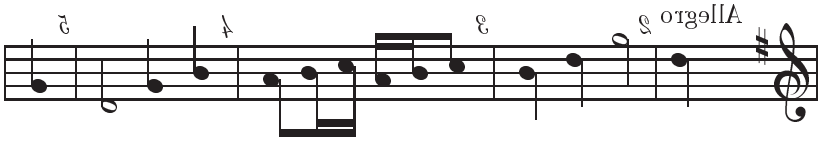
- Esto nos lleva a proponer un programa (y de hecho lo planteamos en este ejercicio) que lea partituras en forma invertida.
  Es curioso observar que la partitura anterior, queda igual cuando se invierte:




- Otros modos de leer una partitura consisten en usar un *espejo* colocado por encima del pentagrama. De esta forma las notas se leen en el mismo orden en que vienen dadas, pero ahora las agudas suenan más graves y las graves más agudas...




  Y si se coloca el espejo verticalmente, bien a la derecha o a la izquierda de la partitura, se obtiene otra bonita melodía:




  Por supuesto, lo que se propone ahora es alterar nuestro programa para que interprete la música expresada por estas otras variantes.

### 5.13.11 Bibliografía

Ya puedes suponer que en este ejercicio no está incluida toda la teoría de la música completa: si deseas ampliar tu formación musical a partir de este punto, puedes acudir a [dPC00], entre otras buenas referencias.

## 5.14 Números de Eudoxus

268

Los números de Eudoxus se definen como sigue:

$$\begin{array}{lll} x_r = y_r + y_{r-1} & \text{si} & r \geq 1 \\ y_r = x_{r-1} + y_{r-1} & \text{si} & r \geq 1 \\ x_0 = 1 & & \\ y_0 = 0 & & \end{array}$$

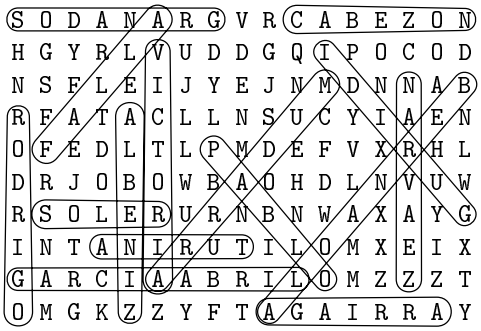
Se pide un programa eficiente que genere la secuencia de pares $(x_i, y_i)$ para $0 \leq i \leq n$.

***Un poco de historia*** Eudoxus (408–355 a.C.), como era habitual en la época griega, compaginó la filosofía, la astronomía y las matemáticas. Se dice que fue el primer griego en hacer un mapa de las estrellas. Sus trabajos matemáticos tuvieron una gran repercusión. El libro V de los *Elementos* de Euclides [Euc94] está basado en sus descubrimientos sobre las proporciones.

## 5.15 Sopas de letras

Uno de los pasatiempos más conocidos es la *sopa de letras*. Una sopa es un rectángulo de $N$ filas y $M$ columnas en el que hay letras, en principio, completamente desordenadas. El pasatiempo consiste en encontrar, entre dichas letras, palabras relacionadas con un tema. Las palabras pueden leerse en la sopa de izquierda a derecha o de derecha a izquierda, en dirección vertical, horizontal o diagonal.

***Ejemplo*** La siguiente sopa de letras contiene 18 nombres de músicos españoles:




***Tipos para la sopa de letras*** Realiza las definiciones de tipo necesarias para manejar una sopa de letras.

***Tipos para las palabras*** Cada palabra válida, oculta en una sopa de letras se puede indicar mediante la secuencia de sus letras, la posición en que empieza y su dirección de lectura. Define un tipo de datos apropiado para manejar esta información.

***Búsqueda de una palabra*** Realiza un subprograma que, dada una sopa de letras y una palabra, averigüe si dicha palabra está en la sopa, indicando además, en caso afirmativo, la posición en que empieza y su dirección de lectura.

***Uso de archivos*** El planteamiento de una sopa de letras se puede desglosar en tres partes:

- La matriz de letras.
- El rótulo que plantea el tema de la sopa de letras (por ejemplo, músicos españoles).
- La lista de palabras que forman la solución.

Define los tipos de datos y desarrolla los subprogramas necesarios para cargar una sopa de letras, previamente almacenada en el disco.

***El programa completo*** Desarrolla un programa que plantee sopas de letras y ayude a resolverlas.

## 5.16 Códigos lineales  254

En el ejercicio 4.16 se presenta una forma sencilla y general de utilizar códigos correctores de errores. Consideraremos una familia especial de estos códigos: los *códigos lineales*, que permiten que la codificación y la descodificación puedan realizarse eficazmente.

Los códigos lineales pueden representarse mediante una *matriz generadora* que tiene un aspecto como el siguiente,

$$G = [I_k|A] = \begin{bmatrix} 1 & \cdots & 0 & a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & a_{k1} & \cdots & a_{kn} \end{bmatrix}$$

donde $A$ es una matriz de dimensión $k \times n$ e $I_k$ es la matriz identidad de dimensión $k$, es decir, todas sus componentes son ceros salvo en la diagonal principal que tiene unos.

Las palabras que forman el código propiamente dicho se obtienen multiplicando todos los vectores posibles de dimensión $k$ por la matriz generadora.

Consideremos el siguiente ejemplo: sea $G_0$ una matriz generadora.

$$G_0 = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

$G_0$ es una matriz de $2 \times 5$ y, siguiendo el patrón, está formada por la matriz identidad de dimensión 2 y un matriz de dimensión $2 \times 3$. El número de posibles vectores de dimensión 2 es cuatro. Cada fila de la siguiente matriz es un vector distinto:

$$V = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Las palabras del código que genera la matriz $G_0$ se obtienen al multiplicar la matriz $V$ de los vectores, de dimensión $4 \times 2$, por la matriz $G_0$, de dimensión $2 \times 5$. El código obtenido, $C$, de dimensión $4 \times 5$, es un código de 4 palabras que utiliza 4 bits para describir cada palabra.

$$C = V \cdot G_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Ten en cuenta que estamos utilizando *aritmética modular*, en este caso de módulo 2, y por tanto $1+1 = 0$. Para diferenciar el significado de las matrices que utilizamos hemos adoptado la siguiente notación: rodearemos las matrices de códigos con paréntesis ( ) y las demás con corchetes [ ].

En los códigos lineales se aprecia muy bien cuál es el mensaje de origen y cuál es la redundancia de información introducida. En efecto, el código de un vector $v$ se obtiene al multiplicar $v$ por la matriz generadora $G$. Como toda matriz generadora es de la forma $G = [I_k|A]$, entonces el código tiene como primeras componentes al propio vector $v$ y el resto de los componentes son la redundancia de información que introducimos. Por ejemplo, con la matriz $G_0$ definida anteriormente y $v = [1, 1]$, se tiene:

$$v \cdot G_0 = [1,1] \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} = [\overbrace{1,1}, \underbrace{1,0,1}]$$

origen

redundancia

Una de las grandes ventajas que poseen los códigos lineales frente a los códigos generales es el cálculo de la distancia mínima. En el apartado 4.16.2 se encuentra la forma general de calcular la distancia mínima. Sin embargo, la distancia mínima de un código lineal es simplemente el menor de los pesos de los vectores no nulos del código. El *peso* de un vector es el número de componentes a uno que tiene. Por ejemplo, el peso de 0000 es cero y el peso de 010101 es tres. En el ejemplo que estamos considerando, el peso de los vectores no nulos se recoge en la siguiente tabla,

| Vectores no nulos de C | Peso |
|---|---|
| 0 1 0 1 1 | 3 |
| 1 0 1 1 0 | 3 |
| 1 1 1 0 1 | 4 |

y por tanto la distancia mímina de $C$ es tres: $d(C) = 3$.

### 5.16.1 Matrices generadoras

Escribe un programa que permita calcular un código $C$ a partir de una matriz generadora $G$ en forma estándar. Es decir, el programa debe identificar las dimensiones de la matriz $G$, comprobar que está en forma estándar, generar la matriz de vectores $V$ adecuada y llevar a cabo la multiplicación $V \cdot G$ para definir $C$. Además, calculará la distancia mímina de $C$ e informará de las propiedades de detección y corrección de errores que posee el código $C$.

***Ejemplo*** Sea $G$ una matriz generadora de dimensión $3 \times 7$:

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Los vectores binarios de dimensión tres son ocho ($2^3$). Cada uno de ellos es una fila de la matriz $V$:

$$V = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad C = V \cdot G = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Las palabras del código se obtienen al multiplicar la matriz $V$ de los vectores, de dimensión $8 \times 3$, por la matriz $G$ de dimensión $3 \times 7$ para obtener un código $C$ de dimensión $8 \times 7$, es decir, un código de ocho palabras que utiliza siete bits para describir cada palabra. La distancia mímina del código $C$ es 4: $d(C) = 4$.

### 5.16.2 Tabla para la descodificación

Para descodificar códigos lineales de una forma más eficaz a aquélla que se presenta en el ejercicio 4.16 vamos a construir lo que se denomina *tabla de descodificación*, que es una tabla en la que se agrupa (por columnas) a cada palabra $p$ del código con todas aquellas palabras que sin pertenecer al código están a distancia mínima de $p$. De esta forma, si se recibe una palabra $q$, posiblemente con errores, basta con buscar en la tabla $q$ y a partir de $q$ se encuentra $p$ que es la palabra original que supuestamente fue enviada.

Construir esta tabla para códigos lineales es muy fácil: en la primera fila de la tabla ponemos las palabras que forman el código. El código generado por la matriz $G_0$ es ésta:

| | | | |
|---|---|---|---|
| 00000 | 01011 | 10110 | 11101 |

El resto de las filas se definen como sigue:

- Se elige un vector $x$ cuyo peso sea menor al de los no nulos que ya están en la tabla.
- Para cada palabra $c$ del código se calcula $x + c$ y se coloca el resultado en la columna de $c$.

Y así hasta que todos los vectores de la dimensión que estamos considerando aparezcan en la tabla.

En nuestro ejemplo, podemos elegir como vector de mínimo peso a 10000 y realizamos las sumas, por ejemplo $10000 + 01011 = 11011$ y $1000 + 10110 = 00110$. La tabla de descodificación tras este primer paso queda así:

| | | | |
|---|---|---|---|
| 00000 | 01011 | 10110 | 11101 |
| 10000 | 11011 | 00110 | 01101 |

Cada una de las filas forma una *clase,* y a los vectores pertenecientes a la primera columna se les denomina *líderes de clase*. Los líderes de clase nos informan del error que se ha producido en los vectores de su clase, que son aquéllos que están en su fila.

Si completamos la ejecución del algoritmo obtenemos la tabla de descodificación completa para el código de la matriz generadora $G_0$:

| | | | |
|---|---|---|---|
| 00000 | 01011 | 10110 | 11101 |
| 10000 | 11011 | 00110 | 01101 |
| 01000 | 00011 | 11110 | 10101 |
| 00100 | 01111 | 10010 | 11001 |
| 00010 | 01001 | 10100 | 11111 |
| 00001 | 01010 | 10111 | 11100 |
| 11000 | 10011 | 01110 | 00101 |
| 01100 | 00111 | 11010 | 10001 |

Escribe un programa que, dada una matriz generadora $G$, calcule la tabla de descodificación correspondiente.

### 5.16.3 Descodificación

Finalmente veremos cómo se utiliza la tabla de descodificación para detectar y corregir los errores de transmisión.

Para descodificar una palabra recibida $q$, posiblemente con errores, buscamos $q$ en la tabla y una vez encontrada ya tenemos en su fila y su columna toda la información relativa. La palabra enviada $p$ es la primera palabra de la columna en la que se encuentre $q$ y el error cometido en la transmisión es el líder de clase de la palabra $q$, es decir, la primera palabra de la fila en la que se encuentra $q$. Además, la primera parte de la palabra $p$ (al quitar la redundancia) nos dice cuál es la palabra original.

Consideremos la tabla de descodificación anterior. Supongamos que hemos recibido la palabra 10010. Suponiendo que se ha producido un error, o ninguno, la palabra de origen es la primera de la columna en la que se encuentra, 10110, y el error cometido es el que indica el líder de clase, 00100, situado el primero de la fila que ocupa la palabra 10010.

| | | | |
|---|---|---|---|
| 00000 | 01011 | 10110 | 11101 |
| 10000 | 11011 | 00110 | 01101 |
| 01000 | 00011 | 11110 | 10101 |
| 00100 | 01111 | 10010 | 11001 |
| 00010 | 01001 | 10100 | 11111 |
| 00001 | 01010 | 10111 | 11100 |
| 11000 | 10011 | 01110 | 00101 |
| 01100 | 00111 | 11010 | 10001 |

Así, la palabra supuestamente enviada es 10110 y por tanto, al ser un código lineal, el mensaje original es 10.

Es importante darse cuenta de que el uso de la tabla de descodificar está sujeto a las propiedades de corrección del código que estemos utilizando. En efecto, si utilizamos un código con distancia mínima 2, sólo podemos utilizar dicho código para identificar errores, pero no para corregirlos. (Véase el apartado 4.16.2.)

Al construir la tabla de descodificación pueden aparecer líderes de clase que tienen un peso mayor al error que se puede corregir. Si esto ocurre, las palabras que se encuentran en dichas clases no pueden ser descodificadas *con certeza*, ya que se han producido más errores de los que el código puede corregir.

En la tabla de descodificación que hemos calculado para la matriz generadora $G_0$, podemos observar cómo las dos últimas filas tienen por líderes de clase a los vectores 11000 y 01100, con peso 2 y sin embargo la distancia mínima del código generado por $G_0$ es tres, $d(C) = 3 = 2 \cdot 1 + 1$, lo que implica que podemos detectar dos errores pero corregir sólo uno.

Si la palabra que se intenta descodificar es una de las pertenecientes a las clases cuyos líderes son 11000 y 01100, sabemos que se han producido dos errores, y la descodificación no puede ser exacta, ya que hay varias palabras del código que podrían, tras dos errores de transmisión, transformarse en la palabra que se recibió.

Escribe un programa que permita hacer una simulación total del uso de un código lineal corrector de errores. El programa tendrá que realizar las siguientes tareas:

- Dar a elegir el código lineal con el que se trabajará informándonos de sus propiedades (distancia mínima) para detectar y corregir errores.
- Gestionar la lectura y codificación de los datos que se envían.
- Enviar los datos codificados, es decir, las palabras del código, a través de un canal ruidoso.
- Recibir los datos como salen del canal e intentar descodificar dichos datos utilizando la tabla de descodificación para detectar y corrigir los errores posibles.
- Comparar el mensaje original con el mensaje descodificado para comprobar la eficacia del código elegido con la probabilidad de fallo considerada.

### 5.16 4 Pruébalo primero

A continuación se da una matriz generadora $G_0$ y la solución que ha de obtenerse al resolver los dos primeros apartados:

$$G_0 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad C = V \cdot G_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Tiene distancia mínima 2 y su tabla de descodificación completa es:

| Vectores no nulos de C | Peso |
|---|---|
| 1 0 1 1 | 3 |
| 0 1 0 1 | 2 |
| 1 1 1 0 | 3 |

| | | | |
|---|---|---|---|
| 0000 | 1011 | 0101 | 1110 |
| 1000 | 0011 | 1101 | 0110 |
| 0100 | 1111 | 0001 | 1010 |
| 0010 | 1001 | 0111 | 1100 |

***Para este enunciado*** Consulta la pista 5.16a.

### 5.16 5 Bibliografía

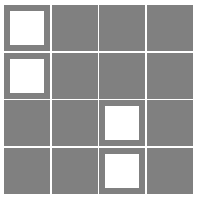
En los textos [Hil99] y [HLL+92] podemos encontrar las explicaciones y demostraciones de los resultados que utilizamos en este ejercicio.

## 5.17 Códigos de trasposición utilizando rejillas

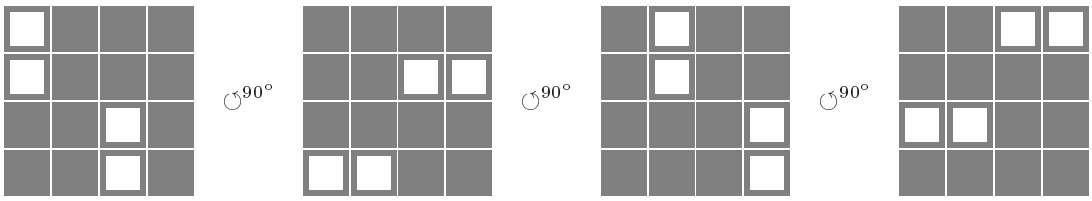
254 273

Los códigos de *sustitución* son aquéllos que para cifrar un mensaje sustituyen las letras originales que forman parte del mensaje por otras pertenecientes a uno o varios alfabetos de cifrado. Por el contrario, los códigos de *transposición* son aquéllos que reordenan las letras que forman parte del propio mensaje, para hacerlo ininteligible. Todas las letras de un mensaje cifrado con un código de transposición se encuentran en el mensaje original. Gran parte de los códigos de transposición se basan en el empleo de *rejillas* que al parecer fueron incorporadas a la criptografía por el científico italiano Girolamo Cardano (1501–1576).

Como su nombre indica, un código de transposición es un mecanismo de reorganización de las letras de un mensaje para convertirlo en un texto cifrado. Las rejillas son un soporte cuadrangular que tiene una serie de agujeros o huecos. En la figura siguiente podemos ver una rejilla de dimensión cuatro y que tiene cuatro huecos:



Los huecos de una rejilla deben estar dispuestos de forma que no coincida ninguno al sobreponer cuatro copias, la primera en la orientación original y las siguientes giradas un cuarto de vuelta con respecto a la anterior, siempre en el mismo sentido. El siguiente dibujo muestra cuatro copias de nuestro ejemplo de rejilla, giradas como acabamos de explicar:

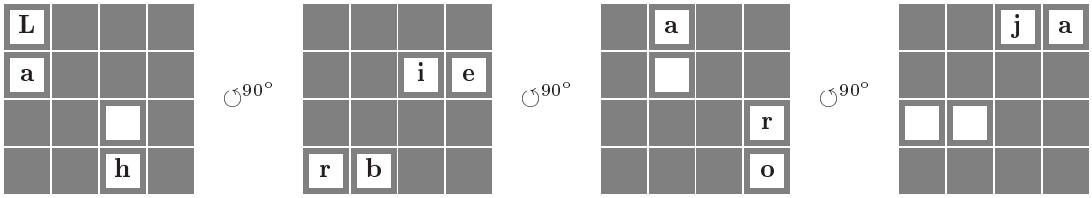


Si se sobreponen mentalmente se verá que no coincide ningún hueco. Más aún, y esta propiedad es igualmente importante, en cada posición de la rejilla cae algún hueco. La siguiente matriz permite constatar este hecho y en qué momento, a lo largo de los cuatro giros, alcanzamos a ver cada una de las posiciones de la rejilla; un 0 hace referencia a la rejilla inicial, un 1, a la rejilla girada un cuarto de vuelta, un 2, a la siguiente, etc.

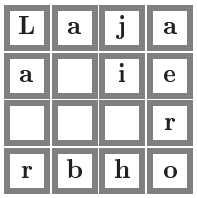
$$\begin{array}{|c|c|c|c|}\hline 0 & 2 & 3 & 3 \\ \hline 0 & 2 & 1 & 1 \\ \hline 3 & 3 & 0 & 2 \\ \hline 1 & 1 & 0 & 2 \\ \hline \end{array} \tag{5.1}$$

Una vez que tenemos una rejilla con las dos propiedades descritas, podemos utilizarla para definir un código de transposición. Supongamos que queremos mandarle a una amiga el título de uno de nuestros libros favoritos, por ejemplo *La hierba roja* de Boris Vian. Lo que haremos será escribir en los huecos que deja la rejilla en cada rotación, y luego dar el mensaje por filas; de esta forma las letras que formaban el mensaje original se han entremezclado. Los espacios en blanco forman parte de la frase y por tanto los respetaremos como al resto de los caracteres.

Primero colocamos la rejilla en su posición inicial y escribimos (en el orden habitual de izquierda a derecha y de arriba a abajo) las letras que forman parte del mensaje en los huecos que quedan libres. Cuando completamos dichos huecos damos un cuarto de vuelta a la rejilla y continuamos el proceso.




Tras realizar la escritura en los huecos, obtenemos la siguiente matriz:




Si mostramos la matriz por filas y representamos, por claridad, los espacios en blanco con ␣, obtenemos el siguiente mensaje:

`Lajaa␣ie␣␣␣rrbho`

El método que hemos descrito permite, por supuesto, cifrar mensajes más grandes porque, una vez que hemos acabado de completar una matriz, podemos volver a comenzar con otra y repetir el proceso.

La forma de descifrar un mensaje es seguir los pasos de cifrado de forma inversa. La clave en estos códigos está compuesta esencialmente por los huecos de la rejilla. Además, ciertos detalles, como que escribimos en el orden usual y que mostramos la codificación por filas, forman también parte de la clave.

Para descifrar un mensaje sabiendo que vamos a usar una determinada rejilla de dimensión cuatro, tenemos que colocar el mensaje original en una matriz de cuatro por cuatro y aplicar la rejilla anotando cuáles son las letras que quedan en los huecos; de esta forma, al acabar de completar el ciclo, tenemos reordenadas las letras que están escritas en la matriz.

***Cifrado y descifrado con rejillas*** Suponiendo que tenemos una rejilla que cumple las dos propiedades mencionadas, escribe un programa que cifre y descifre mensajes. (Véase la pista 5.17a.)

***Construcción de rejillas*** Escribe un programa que fabrique rejillas de cualquier dimensión. Estudia un poco (con lápiz y papel) la estructura de las rejillas para aprender a generar rejillas válidas. (Véase la pista 5.17b.)

***Bibliografía*** Los cifrarios de rejilla se remontan a los tiempos de Girolamo Cardano (1501–1576). El escritor Julio Verne (1828–1905) era muy aficionado a la criptografía y en varios de sus libros podrás encontrar referencias a cifrarios. En concreto, en *Mathias Sandorf* aparecen los cifrarios de rejillas.

## 5.18 Un laberinto

254

Sea un laberinto con un punto de entrada y otro de salida. El objetivo usual es alcanzar la salida en el menor tiempo. Supongamos que las siguientes reglas rigen en el laberinto:

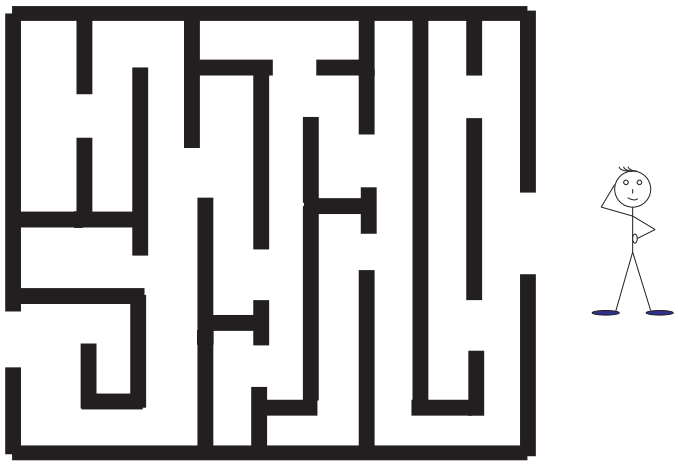


Figura 5.9: Un laberinto con una entrada y una salida

- Tanto el punto de entrada como la salida deben estar situados en uno de los bordes del laberinto.
- Alguien, en nuestro ejercicio un monigote, que entre en el laberinto sólo podrá desplazarse hacia arriba, abajo, derecha o izquierda. Además, deberá tener cuidado con los bordes del laberinto.
- Un movimiento supondrá el desplazamiento desde la posición en la que se encuentra el monigote a una de las cuatro posibles adyacentes.
- Un obstáculo es infranqueable para el monigote.

Escribe un programa que contemple los siguientes aspectos:

***Creación del laberinto*** Un subprograma se puede encargar de esta tarea solicitando al usuario las dimensiones del laberinto, el punto de entrada, de salida, el número de obstáculos y sus posiciones. (Véase la pista 5.18a.)

***Esquema del laberinto*** Representa el laberinto. (Véase la pista 5.18b.)

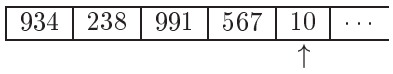
***Búsqueda de la salida*** Habrá que decir si existe una forma de conducir al monigote desde la casilla de entrada a la de salida. (Véanse las pistas 5.18c y 5.18d.)

***Camino solución*** La solución consistirá en la relación de las posiciones de las casillas por las que tiene que pasar el monigote para salir del laberinto.

## 5.19 Implementación de números grandes

255 279

Los enteros en cualquier lenguaje de programación están limitados en su tamaño. En este ejercicio se deberá definir un nuevo tipo de entero que no tenga dicha limitación. Para ello se elegirá una `base` *suficientemente grande*, y se representarán los números como listas de cifras en esa base. Si `base` = 1000 el número 10567991238934 (expresado en base 10) se puede representar así,




puesto que

$$10567991238934 = 10 \times 1000^4 + 567 \times 1000^3 + 991 \times 1000^2 + 238 \times 1000^1 + 934 \times 1000^0.$$

***Definición de tipo*** Diseña un tipo de enteros basándote en cifras entre 0 y `base`. No hay que olvidarse del signo. (Véase la pista 5.19a.)

***Operadores de comparación*** Realiza algoritmos que implementen las relaciones binarias de igualdad, mayor o igual y menor o igual; el resto de las relaciones se pueden expresar negando las anteriores.

***Suma y resta*** Diseña algoritmos para sumar y restar.

***Por 2, entre 2*** Diseña algoritmos para multiplicar y dividir por 2 y comprobar la paridad. (Véase la pista 5.19b.)

***Multiplicación y división*** Diseña algoritmos para multiplicar y dividir este tipo de números.

***Elección de la base*** Llega el momento de tomar la decisión sobre la base. Para ello, se ha de observar la implementación de las operaciones realizadas. En primer lugar piensa que el espacio que ocupa un `int` en C++ es fijo e independiente de su contenido; por tanto, cuanto más pequeña sea la base más memoria se desperdiciará. ¿Cuál es la base que propones? (Véase la pista 5.19c.)

***Bibliografía*** La necesidad de manejar números grandes no es un capricho, aunque tampoco está exenta de agradables sorpresas, dicho sea de paso. En [Gru84] se atiende a ambos aspectos: al lado práctico y al meramente placentero. Por cierto, la comprobación de la conjetura que se plantea en el ejercicio 3.15 requiere en seguida el uso de números de gran tamaño...

**5.2a.** Las palabras pueden estar almacenadas en un fichero de texto; por ejemplo, una palabra por línea. Así, obteniendo un número entero de forma aleatoria y asociándolo a un número de línea, se puede seleccionar la palabra que el jugador tendrá que adivinar.

**5.2b.** Te proponemos que representes cada letra de la palabra que hay que adivinar con la siguiente estructura de datos:

```
typedef struct Letra {
  char letra;
  bool visible;
} Letra;
```

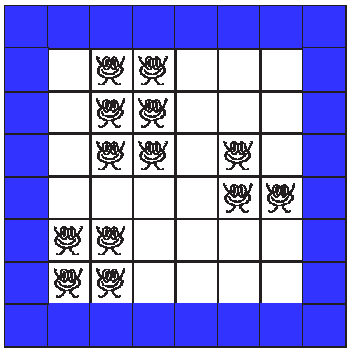
El campo `visible` servirá tanto para saber las letras que tienes que mostrar en cada jugada, como para determinar si el jugador las ha adivinado todas.

**5.3a.** Observa que, puesto que la matriz de distancias es simétrica, queda perfectamente definida con los elementos que están por debajo de la diagonal (o por encima), lo que se conoce como la matriz triangular inferior (o superior). Piensa en una estructura para almacenar únicamente estos valores representativos. El ejercicio 4.5 puede darte algunas ideas.

**5.9a.** Convendrá disponer de un subprograma para enseñar en la pantalla el crucigrama durante el proceso de su resolución, de una forma adecuada.

**5.10a.** Obviamente el programa se puede realizar hasta el último detalle, y si sabemos cómo hacer dibujos y sonidos puede quedar bastante bonito. Entre los muchos detalles que aparecen, hacer que la computadora juegue de forma perspicaz requiere que examinemos detenidamente cuáles son los movimientos más convenientes en cada caso. Intenta hacer un programa bien estructurado que permita ir añadiendo *inteligencia* a la forma de jugar del programa. Puedes comenzar con jugadas completamente aleatorias e ir añadiendo subprogramas que detecten si alguno de los movimientos permite entrar en casa o comer a alguien, etc.

**5.11a.** Debe tenerse cuidado con las casillas de los bordes. En este caso, la solución más sencilla consiste en tener un mundo *ampliado* en el que el borde siempre esté deshabitado. Por ejemplo, para el mundo de la figura 5.7b, el programa puede mantener la siguiente representación interna:



La representación propuesta permite que nos olvidemos de los límites del mundo, pero obliga a otros subprogramas a tener en cuenta dicha representación.

**5.11b.** Como los nacimientos o muertes de una generación a otra ocurren de forma simultánea, la evolución

no puede *calcularse* modificando un tablero casilla a casilla: en efecto, en caso de proceder de este modo, los nacimientos nuevos podrían contribuir a fomentar nuevamente la natalidad en su misma generación, o provocar asfixias prematuras; además, el orden en que se actualizaran las casillas repercutiría en el modo de evolucionar del mundo. Por consiguiente, hace falta un segundo tablero en el que calcular la población de la generación siguiente, conservando el anterior intacto, como referencia.

**5.11c.** Lo más sencillo es que dichos archivos sean de texto. Bastan dos caracteres para definir una generación inicial.

**5.12a.** Para simplificar al usuario la indicación de las casillas que desea pulsar, podemos suponer dichas casillas numeradas de 0 a $M \times N - 1$. Por ejemplo para un tablero $4 \times 5$ los números de casillas pueden ser:

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |

El ejercicio 4.5 resuelve problemas similares.

**5.12b.** El recorrido del tablero debe detenerse en cuanto se encuentre una luz encendida.

**5.16a.** Implementa de forma independiente todas las operaciones que necesitas realizar con vectores y matrices: multiplicación, transposición, generación de vectores de una determinada dimensión...

**5.17a.** La solución del problema se simplifica mucho si encontramos el formalismo adecuado para expresarlo. En este caso, ya que estamos interesados en rotaciones, debemos considerar a la rejilla desde un punto de vista geométrico. Una rejilla puede verse como una matriz de puntos en el plano. Con poco que sepas de geometría te será fácil expresar ciertas operaciones como *girar la rejilla*. Ten en cuenta que las coordenadas **no** coindicen con las posiciones que habitualmente se escriben en una matriz, pero esto no es ningún problema, ya que la forma en que escribimos una matriz es un puro convenio que podemos alterar para nuestro programa.

**5.17b.** Para ver cuáles son las posiciones que ocupa un punto en las sucesivas rotaciones consulta el ejercicio 1.12.

**5.18a.** Una representación posible de un laberinto es un *array* de dos dimensiones en el que cada elemento puede asociarse al punto de entrada, a un obstáculo, a una vía libre o a la salida del laberinto. La posición de estos elementos se dará mediante su fila y su columna. Habrá que controlar que las posiciones de las casillas de entrada y de salida del laberinto estén en uno de los bordes de la matriz.

**5.18b.** Para representar el laberinto de forma simple, lo más sencillo será asociar un carácter a la existencia de un obstáculo en una casilla, otro que represente la salida y un tercero para representar el monigote.

**5.18c.** Ten en cuenta que, al ir avanzando por un camino, puede darse el caso de que llegues a una casilla que no te permita continuar; en este caso, tendrás que deshacer el camino andado hasta que alcances una casilla en la que puedas escoger otra ruta. Este planteamiento de búsqueda de una solución es de naturaleza recursiva.

**5.18d.**

> —Sólo hay una manera —recitó, en efecto, Guillermo— de encontrar la salida de un laberinto. Al llegar a cada nudo nuevo, o sea hasta el momento no visitado, se harán tres signos en el

camino de llegada. Si se observan signos en alguno de los caminos del nudo, ello indicará que el mismo ya ha sido visitado, y entonces sólo se marcará un signo en el camino de llegada. Cuando todos los pasos de un nudo ya estén marcados, habrá que retroceder. Pero si todavía quedan uno o dos pasos sin marcar, se escogerá uno al azar, y lo marcará con dos signos. Cuando se escoja un paso marcado con un solo signo, se marcarán dos más, para que ya tenga tres. Si al llegar a un nudo sólo se encuentran pasos marcados con tres signos, o sea, si no quedan pasos que aún falte marcar, ello indicará que ya se han recorrido todas las partes del laberinto.

—¿Cómo lo sabéis? ¿Sois experto en laberintos?

—No, recito lo que dice un texto antiguo que leí en cierta ocasión.

—¿Y con esa regla se puede encontrar la salida?

—Que yo sepa, casi nunca. Pero igual probaremos.

*El nombre de la rosa*, Umberto Eco

**5.19a.** Para aprovechar la memoria es mejor usar la versión `unsigned` de alguno de los enteros predefinidos en C++. Hay que pensar que será necesario realizar operaciones de suma y resta de forma que el resultado no se salga del rango apropiado.

**5.19b.** Para encontrar algoritmos adecuados para dividir por 2, conviene que la base que estamos utilizando para representar los números sea par. ¿Por qué? Como aún no hemos tomado la decisión del valor o valores adecuados para la base, este punto tendrá que considerarse más adelante.

**5.19c.** ¿Qué problemas presenta la elección `base` = `UINT_MAX`?

## 51 Lógica e incertidumbre



***Tipo de datos*** Basta con enumerar los valores del tipo. Los ordenamos de menor a mayor grado de certidumbre, y los vinculamos con números para facilitar su comparación:

```
typedef enum BoolPlus {
  falso, talVez, cierto
} BoolPlus;
```

***Conjunción lógica*** Esta función podría definirse caso a caso, considerando todos los pares posibles de valores lógicos. Esta solución es sencilla pero algo tediosa. Con los valores lógicos de siempre, se observa que la conjunción lógica da el valor de menor certidumbre; y esto también ocurre con nuestra lógica trivalorada. En resumen, podemos dar la solución siguiente:

```
BoolPlus operator&&(BoolPlus const p, BoolPlus const q) {
  return (p <= q ?  p :  q);
}
```

***Disyunción*** Empecemos por ver el funcionamiento de la disyunción: de manera similar al caso anterior, la disyunción da el valor de mayor certidumbre, tanto en la lógica estándar como en ésta:

```
BoolPlus operator||(BoolPlus const p, BoolPlus const q) {
  return (p >= q ?  p :  q);
}
```

Pero ahora no se pide una función, sino una estructura de datos en que almacenar dicha correspondencia: la tabla tiene la misma estructura que las tablas de verdad de la lógica estándar: es de doble entrada (una por argumento). Se trata por tanto de una matriz de tres (valores `BoolPlus`) por tres (valores `BoolPlus`), y sus componentes son también valores `BoolPlus`:

```
typedef BoolPlus TablaBool[3][3];
```

Ahora, el modo de rellenar la tabla es sencillo, y lo podemos describir haciendo uso de la disyunción anterior:

```
void rellenarTablaOr(TablaBool tabla) {
  for (int i=falso;i<=cierto;i++)
    for (int j=falso;j<=cierto;j++)
      tabla[i][j] = (BoolPlus)i || (BoolPlus)j;
}
```

***Negación*** Para modificar el parámetro del subprograma, cambiando su valor, éste se ha de pasar por referencia:

```
void negarBool(BoolPlus& a) {
  a = !a;
}
```

Para efectuar la negación puede sobrecargarse, como en los apartados anteriores, dicho operador. La negación puede describirse brevemente caso a caso:

```
BoolPlus operator!(BoolPlus const a) {
  switch(a) {
    case cierto:  return falso;
    case talVez:  return talVez;
    case falso:  return cierto;
  }
}
```

***Uso de las operaciones definidas*** Pongamos la secuencia de instrucciones necesaria, incluyendo las asignaciones de los valores iniciales:

```
cout << "Cálculo de '!  ((b && c) || d)':" << endl;
BoolPlus b = cierto;
BoolPlus c = talVez;
BoolPlus d = falso;
BoolPlus a = (b && c) || d;
negarBool(a);
cout << "!((" << b << " && " << c << ") || " << d << ") -> "
     << a << endl;
```

## 5.2 El juego del ahorcado

225 253

Ahorcar no es siempre de buen gusto. Dependiendo de la pía tradición religioso-cultural que nos embriague, preferiremos empalar, crucificar, aplicar garrote, someter a tortura inquisidora, o mantener un buen retén de parados para controlar las demandas salariales.

### 5.2.1 Dibujo

Como no queremos perder clientela para nuestro juego, haremos configurable el modelo de muerte. En todo caso, iremos viéndola llegar en forma de dibujo bidemensional, con más y más detalles según cometemos errores. Por simplicidad, el dibujo se construirá con caracteres ASCII y tendrá un tamaño máximo prefijado:

```
int const maxAnchuraDibujo = 10;
int const maxAlturaDibujo = 10;
```

Regularemos la aparición de detalles asociando a cada carácter un entero que indica el momento en que empezará a mostrarse:

```
typedef struct Dibujo {
  int anchura;
  int altura;
  char dibujo[maxAlturaDibujo][maxAnchuraDibujo+1];
  int oportunidad[maxAlturaDibujo][maxAnchuraDibujo];
} Dibujo;
```

En cada momento sólo podremos ver los caracteres del `dibujo` cuya `oportunidad` asociada sea menor o igual que el número de veces que hayamos fallado. Un dibujo del ahorcado, que sólo necesita $5 \times 6$ caracteres, podría ser así:

```
Dibujo const ahorcado = {
  5, 6,
  {" __ ",
```

```
    "|  |",
    "|  O",
    "|  |",
    "| / \\",
    "|"},
   {{0,3,3,0},
    {2,0,0,4},
    {2,0,0,5},
    {1,0,0,6},
    {1,0,7,0,8},
    {1}}
};
```

Si se revisa al `ahorcado` con atención, se verán algunos detalles raros: hay entradas de `oportunidad` que valen 0 y que siempre coinciden con los blancos; no hay suficientes datos, y algunas filas no tienen cinco columnas. Estos dos últimos detalles van de la mano. C++ completa los arrays con ceros, tanto si son de enteros, como de `char`s o de reales. En la parte del `dibujo` resultarán muy molestos, porque quién sabe a qué carácter corresponde un 0. Pero en la parte de `oportunidad` nos resuelven todos los problemas si acordamos que un 0 aquí hace que se muestre un espacio en vez de su carácter asociado.

### 5.2 2 Palabra

Una sola palabra y evitaremos la muerte. Afortunadamente no es muy larga:

```
int const maxLetras = 20;
```

Y puede que sólo estén `usadas` algunas `letras`.

```
typedef struct Palabra {
  Letra letras[maxLetras];
  int usadas;
} Palabra;
```

No basta con utilizar `char`s para recordar las letras de nuestra salvación; el juez exige, adjunta a cada letra, una casillita donde añadir una marca a las que, por ahora, estén `acertadas`.

```
typedef struct Letra {
  char letra;
  bool acertada;
} Letra;
```

### 5.2 3 Trámites

El juicio comienza y se atiene al funcionamiento de siempre:

```
void jugar(string const& str, Dibujo const& muerte) {
  Palabra palabra;
  iniciaPalabra(palabra, str);
  int const oportunidades = maxOportunidades(muerte);
  cout << "El juego del ahorcado" << endl;
  dibuja(muerte, oportunidades);
  cout << "La palabra tiene " << str.length() << " letras." << endl;
  int errores = 0;
```

```
  while (errores < oportunidades && quedanLetras(palabra)) {
    char letra;
    cout << "Di una letra:  " << flush;
    cin >> letra;
    if (quedaLetra(palabra, letra)) marcaAcertada(palabra, letra);
    else  errores++;
    dibuja(muerte, errores);
    cout << "Adivinadas:  " << palabra << endl;
  }
  muestraResultado(palabra);
}
```

El juez destapa la palabra secreta, la mira y nos dice su longitud, pero nosotros no conocemos ninguna de sus letras:

```
void iniciaPalabra(Palabra& palabra, string const& con) {
  palabra.usadas = con.length();
  for (int i = 0; i < palabra.usadas; i++) {
    palabra.letras[i].acertada = false;
    palabra.letras[i].letra = con[i];
  }
}
```

Un funcionario vocea nuestras oportunidades:

```
int maxOportunidades(Dibujo const& dibujo) {
  int oportunidades = 0;
  for (int i = 0; i < dibujo.altura; i++) {
    for (int j = 0; j < dibujo.anchura; j++) {
      if (dibujo.oportunidad[i][j] > oportunidades) {
        oportunidades = dibujo.oportunidad[i][j];
      }
    }
  }
  return oportunidades;
}
```

El fiscal muestra la pena que nos merecemos:

```
void dibuja(Dibujo const& dibujo, int const errores) {
  for (int i = 0; i < dibujo.altura; i++) {
    for (int j = 0; j < dibujo.anchura; j++) {
      int const oportunidad = dibujo.oportunidad[i][j];
      if (oportunidad <= 0 || errores < oportunidad)  cout << ' ';
      else  cout << dibujo.dibujo[i][j];
    }
    cout << endl;
  }
}
```

El juez pide que declaremos la siguiente letra. Lo hacemos y la busca tranquilo:

```
bool quedaLetra(Palabra const& palabra, char const letra) {
  for (int i = 0; i < palabra.usadas; i++) {
    if (!palabra.letras[i].acertada && palabra.letras[i].letra == letra)
      return true;
  }
  return false;
}
```

No sabemos si hemos fallado; ojalá alguno de esos pequeños titubeos del lápiz sea una marca de acierto:

```
void marcaAcertada(Palabra& palabra, char const letra) {
  for (int i = 0; i < palabra.usadas; i++) {
    if (palabra.letras[i].letra == letra)
      palabra.letras[i].acertada = true;
  }
}
```

La angustia se decanta en alegría o decepción cuando un funcionario displicente nos muestra nuestra palabra. Nunca cometerá el desliz de enseñar una `letra` que no esté `acertada`:

```
ostream& operator<<(ostream& out, Palabra const& palabra) {
  for (int i = 0; i < palabra.usadas; i++) {
    if (palabra.letras[i].acertada) out << palabra.letras[i].letra;
    else out << '_';
  }
  return out;
}
```

Pasa el tiempo. Debemos estar a punto de terminar. A lo mejor ya no quedan más letras por acertar.

```
bool quedanLetras(Palabra const& palabra) {
  for (int i = 0; i < palabra.usadas; i++) {
    if (!palabra.letras[i].acertada) return true;
  }
  return false;
}
```

Por último, el juez lee nuestra sentencia.

```
void muestraResultado(Palabra const& palabra) {
  if (!quedanLetras(palabra)) {
    cout << "Te salvaste." << endl;
  } else {
    cout << "Lo siento, has sido condenado." << endl
         << "La palabra buscada era:  ";
    for (int i = 0; i < palabra.usadas; i++) {
      cout << palabra.letras[i].letra;
    }
    cout << endl;
  }
}
```

### 5.2.4 Fuente

La fábrica de las palabras tiene un fichero muy grande, con `muchas` palabras, tal vez una en cada línea pero, en todo caso, separadas por un blanco. Hay una manivela en forma de función que a cada golpe de vuelta saca aleatoriamente una:

```
string random(string const& fnombre, int const muchas) {
  int n = rand() % muchas;
  ifstream fichero(fnombre.c_str());
  string palabra;
  for (int i = 0; i < n; i++) fichero >> palabra;
  return palabra;
}
```

## 5.3 Guerra de las galaxias

226 253

***Tipos de datos*** Los planetas se pueden definir de forma natural por simple enumeración.

```
typedef enum Planeta {
  mercurio, venus, tierra, marte, jupiter, saturno, urano, neptuno, pluton,
  noPlaneta
} Planeta;
int const numPlanetas = noPlaneta;
```

Añadimos ya la función `siguientePlaneta`, trivial, que será útil para recorrer los planetas sistemáticamente, y el operador `<<` para mostrar datos por un fichero.

```
Planeta siguientePlaneta(Planeta p) {
  return (Planeta)(p+1);
}
ostream& operator<<(ostream& out, Planeta p) {
  string nombrePlaneta[] = {"Mercurio", "Venus", "Tierra", "Marte",
                            "Júpiter", "Saturno", "Urano", "Neptuno", "Plutón"};
  out << setw(8) << nombrePlaneta[p];
  return out;
}
```

Un conjunto de planetas es equivalente a un array de booleanos que indican, respectivamente, la presencia o ausencia del correspondiente planeta en dicho conjunto. El conjunto vacío se crea poniendo todos los valores a `false` en dicho array:

```
typedef struct ConjuntoPlanetas {
  bool datos[numPlanetas];
} ConjuntoPlanetas;
void iniciaConjuntoPlanetas(ConjuntoPlanetas& conjunto) {
  for (int i = mercurio; i <= pluton; i++) {
    conjunto.datos[i] = false;
  }
}
```

En cuanto a las distancias entre planetas, la solución trivial de usar una matriz de `numPlanetas` × `numPlanetas` no es satisfactoria ya que, al ser simétrica, presenta redundancias en un 50% y valores triviales (ceros) en la diagonal principal. Por tanto, es suficiente con registrar la matriz triangular

superior (véase ejercicio 4.5), lo que hacemos en un array con $\big(\texttt{numPlanetas} \times (\texttt{numPlanetas} - 1)\big)/2$ posiciones, que podrían considerarse inicialmente nulas:

```
typedef struct Distancias {
  double datos[(numPlanetas*(numPlanetas-1)) / 2];
} Distancias;
void iniciaMatrizDistancias(Distancias& distancias) {
  for (int i = 0; i< (numPlanetas*(numPlanetas-1)) / 2; i++)
    distancias.datos[i]=0;
}
```

Para acceder a las posiciones $(i, j)$ de la matriz usaremos los subprogramas

```
double valorEn(Distancias const& matriz, int const fila, int const columna);
```

y

```
void ponerEn(Distancias& matriz, int const fila, int const columna,
             double const valor);
```

tal y como se indica en el ejercicio 4.5.

***Carga de la matriz de distancias*** La lectura de la matriz de distancias toma los valores uno a uno de un archivo. Para ello, efectuamos un bucle doble que recorre la matriz bidimensional:

```
istream& operator>>(istream& in, Distancias& distancias) {
  iniciaMatrizDistancias(distancias);
  for (int i = mercurio; i <= pluton-1; i++) {
    for (int j = i+1; j <= pluton; j++) {
      int distij;
      in >> distij;
      ponerEn(distancias, i, j, distij);
    }
  }
  return in;
}
```

La escritura de dicha matriz sigue idéntico esquema de recorrido:

```
ostream& operator<<(ostream& out, Distancias const& matriz) {
  for (Planeta i = mercurio; i <= pluton; i = siguientePlaneta(i)) {
    for (Planeta j = mercurio; j <= pluton; j = siguientePlaneta(j)) {
      out << "|";
      out << setw(8) << valorEn(matriz,j,i);
    }
    out << "|" << endl;
  }
  return out;
}
```

***¿Cuál es el planeta más próximo?*** Para saber cuál es el planeta más cercano, basta con averiguar cuál es el de menor distancia... exceptuando el propio planeta, claro está:

```
Planeta masProximo(Planeta const planeta, Distancias const& distancias) {
  Planeta planetaMasProximo = mercurio;
  if (planeta == mercurio) planetaMasProximo = venus;
  for (Planeta i = planeta; i <= pluton; i = siguientePlaneta(i)) {
    if (planeta != i) {
      if (valorEn(distancias, planeta, i) <
          valorEn(distancias, planeta, planetaMasProximo)) {
        planetaMasProximo = i;
      }
    }
  }
  return planetaMasProximo;
}
```

De ahí la distinción que hacemos inicialmente con Mercurio y la salvedad `(planeta != i)` en cada vuelta del bucle.

***Ídem entre los no visitados*** Ahora, nos interesamos por el planeta a menor distancia, pero de entre los no visitados. (Suponemos que, en efecto, hay al menos un planeta sin visitar.) Para ello, empezamos por buscar el primer planeta sin registrar. Y a partir de él recorremos los demás sin visitar, tratando de acortar la distancia que nos separa de él:

```
Planeta masProximoPendiente(Planeta const planeta, Distancias const& distancias,
                            ConjuntoPlanetas const& planetasVisitados) {
  Planeta planetaMasProximo = mercurio;
  while (planetasVisitados.datos[planetaMasProximo]) {
    planetaMasProximo = siguientePlaneta(planetaMasProximo);
  }
  Planeta i = planetaMasProximo;
  while (i <= pluton) {
    if (!planetasVisitados.datos[i]) {
      if (valorEn(distancias, planeta, i) <
          valorEn(distancias, planeta, planetaMasProximo)) {
        planetaMasProximo = i;
      }
    }
    i = siguientePlaneta(i);
  }
  return planetaMasProximo;
}
```

Puesto que estamos ciñéndonos a los planetas no visitados, no es necesaria la salvedad `(planeta != i)` del apartado anterior, ya que el planeta en que estamos actualmente ya se ha visitado.

***Inspección de un planeta*** La inspección de un planeta debe arrojar un *sí* con probabilidad $\frac{1}{\texttt{resto}}$, siendo `resto` el número de planetas pendientes. Esto equivale a lanzar un dado de `resto` caras representando una cualquiera de ellas al *sí* y las demás al *no*:

```
int dado(int const numCaras) {
  return rand() % numCaras;
}
```

```
bool seEncuentraAqui(int const resto) {
  int n = dado(resto);
  return n == 0;
}
```

***¡A por ella!*** Por último realizaremos un programa que busque a la princesa. Nos colocamos sobre un planeta, en nuestro caso la Tierra, y vamos saltando siempre al planeta más próximo de entre los no visitados. Cada vez que vamos a un planeta en el que la princesa no está, debemos indicar que hemos estado en él para que no se vuelva a visitar.

```
Planeta buscaPrincesa(Distancias const& distancias, ostream& salida) {
  ConjuntoPlanetas planetasVisitados;
  iniciaConjuntoPlanetas(planetasVisitados);
  Planeta planetaActual = tierra;
  int faltanPorVisitar = numPlanetas;
  while (!seEncuentraAqui(faltanPorVisitar)) {
    planetasVisitados.datos[planetaActual] = true;
    faltanPorVisitar--;
    salida << "Estoy en " << planetaActual;
    planetaActual=masProximoPendiente(planetaActual, distancias, planetasVisitados);
    salida << " y me voy a " << planetaActual;
    salida << endl;
  }
  return planetaActual;
}
```

## 5.5 Derivadas y desarrollos en serie

228

***Derivada*** Teniendo en cuenta que la derivada $f'(x)$ se define como el límite siguiente,

$$\lim_{\Delta \to 0} \frac{f(x+\Delta) - f(x)}{\Delta}$$

nos conformaremos con una aproximación como la siguiente,

$$f'(x) = \frac{df}{dx} \simeq d(f,x) = \frac{f(x+\Delta) - f(x)}{\Delta}$$

eligiendo una constante $\Delta$ suficientemente pequeña (por ejemplo, $10^{-3}$). Dicha aproximación viene descrita por la siguiente función:

```
double const delta = 1e-3;
typedef double (*RenR)(double);
double derivada(RenR fun, double x) {
  return (fun(x+delta) - fun(x)) / delta;
}
```

Se precisa el requisito de que `fun` sea derivable.

***Derivada enésima*** La derivada enésima $f^{(n}(x)$ puede definirse mediante la llamada `derivadaEnesima(n, f, x)`, donde la función `derivadaEnesima` sigue la descripción (recursiva) del enunciado:

$$\begin{array}{rcl} \texttt{derivadaEnesima}(0,f,x) & = & f(x) \\ \texttt{derivadaEnesima}(n,f,x) & \simeq & \texttt{derivada}\big(\texttt{derivadaEnesima}(n-1,f,x)\big), \text{ para } n \geq 1 \end{array}$$

Transcribiendo esto a C++, tenemos lo siguiente:

```
double derivadaEnesima(int n, RenR fun, double x) {
  if (n == 0) {
    return fun(x);
  } else {
    return derivada(derivadaEnesima(n-1,fun,x));
  }
}
```

En este caso, el requisito necesario es que `fun` ha de ser derivable $n$ veces.

***Desarrollo en serie de Taylor*** Ahora se trata simplemente de aplicar la definición de derivada enésima, del apartado anterior. Con esta función definida, sólo queda expresar el desarrollo en serie de Taylor como un sumatorio, cuyos numeradores constan de llamadas a la función mencionada y las potencias $(x-a)^k$, que se calculan de forma acumulativa, y cuyos denominadores $(k!)$ se hallan también eficientemente de forma acumulativa. En resumidas cuentas, se tiene el siguiente programa:

```
double taylorCero(int n, RenR fun, double x) {
  double const a = 0; // El desarrollo es alrededor de "a"
  double potencia = 1;
  double acum = fun(a);
  double denom = 1;
  for (int i=1; i <= n; i++) {
    potencia = potencia * (x-a);
    denom = denom * i;
    acum = acum + derivadaEnesima (i,fun,a) * potencia / denom;
  }
  return acum;
}
```

Ahora, la función `taylorCero` exige que la función `fun` sea derivable $n$ veces, siendo naturalmente $n \geq 0$.

## 5.8 Un traductor de Morse

230

***Tabla: su tipo de datos*** Los códigos en Morse no responden a una regla genérica, así que definimos uno a uno los valores de la tabla de dichos códigos,

```
string const alfabetoMorse[] = {
  /* A = */".-",   /* B = */"-...", /* C = */"-.-.", /* D = */"-..",
  /* E = */".",    /* F = */"..-.", /* G = */"--.",  /* H = */"....",
  /* I = */"..",   /* J = */".---", /* K = */"-.-",  /* L = */".-..",
  /* M = */"--",   /* N = */"-.",   /* O = */"---",  /* P = */".--.",
  /* Q = */"--.-", /* R = */".-.",  /* S = */"...",  /* T = */"-",
  /* U = */"..-",  /* V = */"...-", /* W = */".--",  /* X = */"-..-",
  /* Y = */"-.--", /* Z = */"--.."
};
int const numLetras = 26;
```

de forma que la codificación de cada carácter `c` es `alfabetoMorse[c-'A']`.

***Traducción*** Para traducir una palabra a código Morse, la recorremos y construimos una nueva cadena con la codificación de cada letra. Es necesario insertar un espacio tras cada letra en Morse para poder

realizar después la traducción inversa:

```
string traducirAMorse(string const sajon) {
  string resultado = "";
  for (int i = 0; i < sajon.length(); i++) {
    if (sajon[i] >= 'A' && sajon[i] <= 'Z') {
      resultado = resultado + ⟨traducción en Morse de la letra sajon[i]⟩ + " ";
    }
  }
  return resultado;
}
```

La función que devuelve la traducción de cada letra en su codificación Morse es trivial tal y como hemos definido la tabla:

```
string letraMorse(char const c) {
  return alfabetoMorse[c-'A'];
}
```

***Traducción inversa*** La traducción inversa es algo más compleja debido a que cada letra en Morse puede ocupar más de un carácter. Para su realización aplicaremos el esquema clásico de recorrido, por lo que conviene considerar que el texto por traducir está compuesto por *palabras* separadas por un espacio en blanco o más; cada palabra es una secuencia de puntos y guiones. La visión que tendremos del texto por traducir será la siguiente,

| $b_1$ | $p_1$ | $b_2$ | $p_2$ | $b_3$ | $p_3$ | ..... | $b_n$ | $p_n$ |
|---|---|---|---|---|---|---|---|---|

donde cada $b_i$ es una secuencia de blancos, y $p_i$ es una secuencia de puntos y guiones. La cadena de blancos inicial podrá ser vacía, pero las que están entre dos palabras no podrán ser vacías; de lo contrario no podríamos separar fácilmente las palabras.

Para traducir el texto a caracteres normales iremos leyendo cada palabra hasta llegar al final; debemos llevar una variable `pos` que indique la posición del carácter actual que estamos leyendo. Pero, ¿cómo sabremos que hemos llegado al final? Para contestar a esta pregunta supondremos que tras la secuencia de letras en Morse que queremos traducir existe una palabra vacía; es decir, $p_n$ será una secuencia vacía y por tanto no será parte del texto que se traduce; simplemente es una marca final de texto. Además deberemos considerar que $b_n$ puede ser también vacía.

Llevaremos una variable `textoTraducido` en la que vamos acumulando el texto traducido hasta la última palabra analizada (sin incluir ésta).

```
string traducirASajon(string const morse) {
  string textoTraducido = "";
  string letraMorse;
  int pos;
  ⟨Leer la primera palabra de morse en letraMorse⟩
  ⟨pos será la posición del siguiente carácter⟩
  while (letraMorse != "") {
    textoTraducido = textoTraducido + ⟨letra codificada por letraMorse⟩;
    ⟨Leer la siguiente palabra de morse en letraMorse⟩
    ⟨pos será la posición del siguiente carácter⟩
  }
  return textoTraducido;
}
```

Para leer la primera palabra y las siguientes, el procedimiento es muy similar. La única diferencia es que al leer la primera palabra el parámetro `pos` es únicamente de salida (empezamos en la posición 0) y en el procedimiento de leer la siguiente el parámetro será de entrada y salida. Así podemos hacer que el procedimiento de leer la primera palabra llame directamente al de leer las siguientes:

```
void primeraPalabraMorse(string& letraMorse, string const& textoMorse, int& pos) {
  pos = 0;
  siguienteLetraMorse(letraMorse, textoMorse, pos);
}
```

Y ya podemos concentrarnos en `siguienteLetraMorse`. Para realizarla tendremos que saltar los blancos que puede haber delante de la palabra una vez situados sobre el primer carácter no blanco (o estar al final del texto), y buscar el siguiente carácter blanco (o el final del texto). Mientras buscamos ese carácter vamos almacenando en `letraMorse` los caracteres leídos.

```
void siguienteLetraMorse(string& letraMorse, string const& textoMorse, int& pos) {
  letraMorse = "";
  ⟨Saltar blancos y almacenar en pos la posición del siguiente carácter⟩
  while (pos < textoMorse.length() && esCaracterMorse(textoMorse[pos])) {
    letraMorse = letraMorse+textoMorse[pos];
    pos++;
  }
}
```

Para saltar blancos basta con buscar el primer carácter no blanco o el final del texto:

```
void saltarBlancos(string const& textoMorse, int& pos) {
  while (pos < textoMorse.length() && textoMorse[pos] == ' ') pos++;
}
```

La función `esCaracterMorse` simplemente comprueba que el carácter pasado como argumento es un punto o un guión.

Finalmente, para asociar a cada secuencia de caracteres Morse su letra correspondiente, buscamos en la tabla de la codificación Morse. Para calcular la letra a la que corresponda será necesario sumar a la posición encontrada el código ASCII de `'A'`:

```
char letraAlfa(string const& letraMorse) {
  int pos = 0;
  while (pos < numLetras && alfabetoMorse[pos] != letraMorse) pos++;
  return pos+'A';
}
```

En esta función estamos asumiendo que el parámetro real pasado `letraMorse` es válido.

## 5.14 Números de Eudoxus

243

Aunque es de apariencia inofensiva, este ejercicio ofrece interesantes posibilidades para lograr la eficiencia que se pide en el enunciado.

### 5.14.1 Solución trivial: recursión mutua

La solución más inmediata consiste sin duda en transcribir la definición recurrente dada en el par de funciones mutuamente recursivas que definen cada número de Eudoxus,

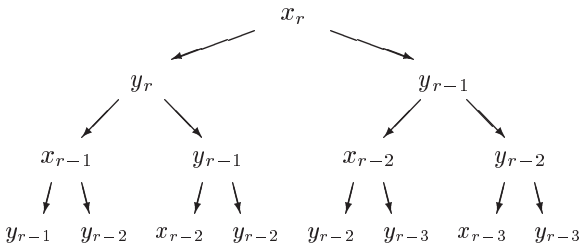
```
int eudoxusY(int const n);
int eudoxusX(int const n) {
  if (n == 0) return 1;
  else return eudoxusY(n) + eudoxusY(n-1);
}
int eudoxusY(int const n) {
  if (n == 0) return 0;
  else return eudoxusX(n-1) + eudoxusY(n-1);
}
```

y entonces generar la secuencia directamente mediante las llamadas a esas funciones:

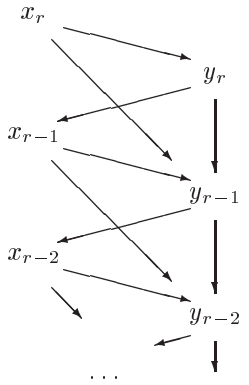
```
int const tamanyo = 12;
int main() {
  for (int i = 0; i < tamanyo; i++)
    cout << i << " -> " << eudoxusX(i) << " " << eudoxusY(i) << endl;
}
```

### 5.14 2 Primera tabulación

Una mirada al árbol de llamadas generado basta para comprender la gran cantidad de cálculos redundantes que se producen en el programa anterior:




En efecto, juntando los nodos iguales del árbol de llamadas se obtiene el siguiente grafo de dependencias:




Una técnica bien conocida para evitar la repetición de cálculos, especialmente en programas recursivos, es la tabulación [Bir86]: en nuestro caso se parte de un par de tablas (para los $x_i$ y los $y_i$ respectivamente). Entonces, cada $x_i$ o $y_i$ que se vaya a calcular se consulta previamente en la tabla, por si ya se hubiera hallado; y cada vez que se calcule uno, se consigna su valor en la tabla.

Para cada elemento de la tabla, debe saberse en todo momento si ya se ha calculado o todavía no; por consiguiente, las tablas tendrán esta estructura:

```
int const tamanyo = 12;
typedef struct Elem {
  bool calculado;
  int valor;
} Elem;
typedef Elem Tabla[tamanyo];
```

Ahora, las reformas en el programa anterior son sencillas:

```
int eudTablaY(int const n, Tabla tx, Tabla ty);
int eudTablaX(int const n, Tabla tx, Tabla ty) {
  if (tx[n].calculado) {
    return tx[n].valor;
  } else {
    int resultado;
    if (n == 0) {
      resultado = 1;
    } else {
      resultado = eudTablaY(n, tx, ty) + eudTablaY(n-1, tx, ty);
    }
    tx[n].calculado = true;
    tx[n].valor = resultado;
    return resultado;
  }
}
int eudTablaY(int const n, Tabla tx, Tabla ty) {
  if (ty[n].calculado) {
    return ty[n].valor;
  } else {
    int resultado;
    if (n == 0) {
      resultado = 0;
    } else {
      resultado = eudTablaX(n-1, tx, ty) + eudTablaY(n-1, tx, ty);
    }
    ty[n].calculado = true;
    ty[n].valor = resultado;
    return resultado;
  }
}
int main() {
  Tabla eudX, eudY;
  for (int i = 0; i < tamanyo; i++) {
    eudX[i].calculado = false;
    eudY[i].calculado = false;
  }
  for (int i = 0;i < tamanyo;i++) {
    cout << i << " -> "
         << eudTablaX(i, eudX, eudY) << " "
```

```
        << eudTablaY(i, eudX, eudY) << endl;
  }
}
```

### 5.14 3 Segunda tabulación

La tabulación propuesta economiza cálculos con respecto a la versión directa recursiva. No obstante, una ligera ojeada al entramado de las llamadas unidas basta para comprender que las tablas `eudX` y `eudY` se rellenan en orden ascendente, por lo que se pueden sustituir las funciones recursivas por un simple bucle:

```
int main() {
  int tablaX[tamanyo];
  int tablaY[tamanyo];
  tablaX[0] = 1; tablaY[0] = 0;
  for (int i = 1; i < tamanyo; i++) {
    tablaY[i] = tablaX[i-1] + tablaY[i-1];
    tablaX[i] = tablaY[i-1] + tablaY[i];
  }
  for (int i = 0; i < tamanyo; i++) {
    cout << i << " -> " << tablaX[i] << " " << tablaY[i] << endl;
  }
}
```

Dos observaciones:

- Es necesario tener la precaución de rellenar antes el elemento $y_r$ que el $x_r$ de cada par, debido a la dependencia del segundo con respecto al primero.
- En el programa se han simplificado las tablas, cuyos elementos ya no requieren información sobre si están hallados o no.

### 5.14 4 Versión iterativa

Una nueva observación nos lleva a una mejora más: el hecho de que, una vez escrito el par de valores $(x_i, y_i)$, sólo sirven para hallar el par siguiente, $(x_{i+1}, y_{i+1})$, hace que no valga la pena mantener la tabla, siendo suficiente con mantener en cada momento el par en uso:

```
int main() {
  int x = 1; int y = 0;
  cout << 0 << " -> " << x << " " << y << endl;
  for (int i = 1; i < tamanyo; i++) {
    int yy = x + y;
    x = yy + y;
    y = yy;
    cout << i << " -> " << x << " " << y << endl;
  }
}
```

### 5.14 5 Versión matricial

La versión anterior es casi inmejorable para calcular la secuencia descrita. Sin embargo, si en vez de la secuencia entera sólo se pidiera un cierto par $(y_i, x_i)$, es posible una mejora sustancial de tiempo, aunque a cambio de un pequeño incremento del espacio consumido. El método empleado explota la

misma idea que el ejercicio 2.2. Recomendamos recordarlo para entender mejor el fin que se persigue con las manipulaciones siguientes.

En primer lugar, sustituyendo $y_n$ por su definición en $x_n$, se tiene el par $(y_i, x_i)$ descrito como una combinación lineal *n*eta de $(y_{i-1}, x_{i-1})$,

$$\begin{array}{lcll} y_n & = & x_{n-1} + y_{n-1} & \text{si} \quad n \geq 1 \\ x_n & = & y_n + y_{n-1} & \text{si} \quad n \geq 1 \\ & & \{\text{sustituyendo } y_n\} & \\ & = & x_{n-1} + 2y_{n-1} & \text{si} \quad n \geq 1 \end{array}$$

lo que se puede expresar mejor en forma matricial:

$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_{n-1} \\ y_{n-1} \end{pmatrix}$$

Y aplicando una inducción sencilla, se tiene la expresión siguiente:

$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\equiv \text{ Primera columna de } \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}^n$$

Por lo tanto, hallar $(x_n, y_n)$ consiste en calcular dicha matriz, que se puede hacer de forma muy eficiente con un algoritmo similar al de la multiplicación a la rusa. Observa los siguientes hechos:

- Si $k$ es par, es decir $k = 2k'$ para algun entero $k'$, tenemos
$$A^k = A^{2k'} = (A^2)^{k'}.$$
- Si $k$ es impar, es decir $k = k' + 1$ para algun entero $k'$ par, tenemos
$$A^k = A^{k'+1} = A \cdot A^{k'}.$$

Así, un algoritmo para calcular la potencia `n` de una matriz `mat` es el siguiente:

```
typedef int Matriz[2][2];
void elevarPotencia(Matriz mat, int const n) {
  Matriz matAcum = {{1,0},{0,1}}; int k = n;
  while (k >= 1) {
    if (k%2 == 1) { // k es impar
      multiplicar(matAcum, mat);
      k--;
    } else { // k es par
      multiplicar(mat, mat);
      k = k / 2;
    }
  }
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      mat[i][j] = matAcum[i][j];
    }
  }
}
```

Falta realizar un procedimiento que multiplique dos matrices, de forma que el primer parámetro es de entrada (un factor) y salida (el resultado), e integrar los dos procedimientos en un programa:

```
void multiplicar(Matriz a, Matriz b) {
  Matriz c;
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < 2; j++) {
      c[i][j] = 0;
      for (int k = 0; k < 2; k++) {
        c[i][j] = c[i][j] + a[i][k] * b [k][j];
      }
    }
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      a[i][j] = c[i][j];
    }
  }
}
int main() {
  Matriz matriz = {{1, 2}, {1, 1}};
  elevarPotencia(matriz, tamanyo-1);
  cout << tamanyo-1 << " -> "
       << matriz[0][0] << " " << matriz[1][0] << endl;
}
```

El coste de esta solución es el de la potencia de la matriz, esto es, lineal, si se halla trivialmente mediante un bucle, o logarítmico, en su versión más eficiente.

## 5 17 Códigos de trasposición utilizando rejillas

248 254

Como indica el enunciado, la codificación con rejilla es una forma de cifrado por transposición. Las rejillas son una forma gráfica e intuitiva de definir una transposición. Utilizaremos este hecho para construir una solución que tiene una parte de cifrado con transposiciones genéricas y otra parte de construcción de rejillas y extracción de la transposición a la que equivalen.

### 5.17.1 Cifrado y descifrado con transposiciones

En álgebra, el concepto de reordenación de $n$ elementos recibe el nombre de *permutación*. Una permutación sólo entiende de posiciones y nunca de valores de los elementos; se debe limitar a decir "el elemento que está en la posición 0 va a la posición $\sigma_0$, el que está en la posición 1 va a la posición $\sigma_1 \ldots$" Como no puede llevar dos elementos a la misma casilla, todos los $\sigma_i$ deben ser distintos entre sí. Finalmente, como los $\sigma_i$ deben estar entre 0 y $n-1$, ambos incluidos, una permutación es una función biyectiva entre los enteros en el rango $[0, n-1]$. La forma más sencilla de escribir una permutación es con la secuencia de los *destinos*: $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$. Algunos ejemplos: la permutación $(1, 0)$ intercambia dos elementos; $(0, 1, \ldots, n-1)$ no hace nada, deja todo donde estaba; $(1, 2, 3, 4, 0)$ desplaza 5 elementos a la derecha y el último, que se habrá salido, pasa a ocupar la primera posición; $(5, 4, 3, 2, 1, 0)$ produce una simetría central; $(0, 4, 10, 14, 6, 7, 12, 13, 1, 5, 11, 15, 2, 3, 8, 9)$ es la permutación que engendra la rejilla que sirve de ejemplo en el enunciado.

Si suponemos que nuestras permutaciones van a intercambiar `maxLongitud` elementos, podemos definirlas así:

```
typedef struct Permutacion {
  int longitud;
  int destino[maxLongitud];
} Permutacion;
```

El campo `longitud` debe ser menor o igual que `maxLongitud` e indica cuántos elementos reordena esta permutación, el segmento inicial utilizable en `destino`. De nuevo, la permutación del ejemplo, esta vez en C++:

```
Permutacion const permutacionEjemplo
= { 16,
    {  0, 4, 10, 14,
       6, 7, 12, 13,
       1, 5, 11, 15,
       2, 3,  8,  9
    }
};
```

Una permutación sólo se puede aplicar a bloques de datos con su misma longitud. Como nuestros datos van a ser caracteres, definimos lo siguiente,

```
typedef struct Bloque {
  int longitud;
  char mensaje[maxLongitud];
} Bloque;
```

que tiene la misma estructura que `Permutacion`. Ahora podemos definir un procedimiento que aplique una `Permutacion` a un `Bloque` y lo reordene para producir otro bloque:

```
void permuta(Bloque& resultado, Permutacion const& perm, Bloque const& datos) {
  resultado.longitud = datos.longitud;
  for (int i = 0; i < datos.longitud; i++) {
    resultado.mensaje[perm.destino[i]] = datos.mensaje[i];
  }
}
```

Observa que la permutación se utiliza para indexar el array del `resultado` porque su propósito es indicar el `destino` de cada uno de los `datos`. La forma de deshacer una permutación es justamente utilizar la permutación para indexar en los datos:

```
void despermuta(Bloque& resultado, Permutacion const& perm, Bloque const& datos) {
  resultado.longitud = datos.longitud;
  for (int i = 0; i < datos.longitud; i++) {
    resultado.mensaje[i] = datos.mensaje[perm.destino[i]];
  }
}
```

La comprobación de que el código

```
permuta(d1, perm, d0);
despermuta(d2, perm, d1);
```

hace que `d0` y `d2` contengan los mismos datos es muy sencilla: el elemento `d2.mensaje[i]` proviene de `d1.mensaje[perm.destino[i]]`, que a su vez es `d0.mensaje[i]`.

Cifrar y descifrar no es más que permutar y *despermutar* una secuencia de datos. Para tratar una secuencia arbitrariamente larga, hay que dividirla en `Bloques`. Puede no haber suficientes datos para completar el último bloque, pero entonces supondremos que se añaden suficientes espacios como para satisfacer exactamente nuestra demanda. Podemos cifrar una secuencia así:

```
void cifra(ostream& out, Permutacion const& perm, istream& in) {
  Bloque datos;
  datos.longitud = perm.longitud;
  while (!in.eof()) {
    in >> datos;
    Bloque resultado;
    permuta(resultado, perm, datos);
    out << resultado;
  }
}
```

Descifrar es prácticamente igual: basta con delegar en `despermuta` en vez de `permuta`.

Para leer y escribir `Bloques` hemos utilizado el extractor (`>>`) y el insertador (`<<`). Obviamente, es deber nuestro añadir las definiciones para el tipo `Bloque`. Ya hemos comentado que, en la lectura, ante la falta de datos, completaremos con espacios:

```
istream& operator>>(istream& in, Bloque& bloque) {
  int i = 0;
  while (i < bloque.longitud && !in.eof()) {
    in.get(bloque.mensaje[i]);
    i++;
  }
  while (i < bloque.longitud) {
    bloque.mensaje[i] = ' ';
    i++;
  }
  return in;
}
```

Escribir un bloque es más sencillo porque siempre está completo. Cierto es que parte de sus datos pueden ser espacios añadidos, pero es imposible evitarlos porque son indistinguibles de los del mensaje y, además, están entremezclados con otros datos ayudando a sostener un orden fundamental que perderíamos:

```
ostream& operator<<(ostream& out, Bloque const& bloque) {
  for (int i = 0; i < bloque.longitud; i++) out << bloque.mensaje[i];
  return out;
}
```

### 5.17.2 Rejillas y derivación de permutaciones

Esta parte de la solución se centra en las rejillas. Necesitamos un tipo de datos con el que representarlas.

Empezamos ahora una reflexión que nos conducirá a este objetivo. Una rejilla tiene huecos que caracterizamos por su posición:

```
typedef struct Hueco {
  int fila;
  int columna;
} Hueco;
```

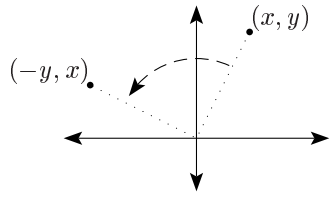
Contaremos desde la esquina superior izquierda, empezando en 0, filas hacia abajo y columnas hacia la derecha. Pero, además, una rejilla *es* sus huecos; así que la podríamos definir como una ristra de `Huecos`, de nuevo con una estructura similar a la de una `Permutacion`:

```
typedef struct RejillaComoHuecos {
  int usados;
  Hueco huecos[maxHuecos];
} RejillaComoHuecos;
```

La rejilla del ejemplo sería ésta:

```
RejillaComoHuecos const rejillaEjemplo = {
  4,
  { {0,0}, {1,0}, {2,2}, {3,2} }
};
```

Girar una rejilla 90 grados en sentido contrario al de las agujas del reloj es una operación fundamental. Toda la dificultad recae en el giro de un `Hueco`. El giro de la rejilla tiene que ser respecto a su centro; pero se obtiene el mismo resultado girando con respecto a la esquina superior izquierda y luego deslizando el resultado hacia abajo hasta que los bordes de la rejilla se sobrepongan. (La magnitud del desplazamiento es justamente uno menos que la longitud del lado de la rejilla.)




En lo referente a un hueco con posición $(x, y)$ en una rejilla de lado $l$, al girar respecto a la esquina superior izquierda, pasa a la posición $(-y, x)$ (que tiene fila negativa porque cae por encima del borde superior de la rejilla) y luego, al deslizarlo hacia abajo, termina en la posición $(-y + (l - 1), x)$.

```
void rota(Hueco& hueco, int const lado) {
  int const fila0 = hueco.fila;
  hueco.fila = lado - hueco.columna - 1;
  hueco.columna = fila0;
}
```

Girar una rejilla compuesta de huecos es aplicar el procedimiento anterior a cada hueco.

Llegamos ahora al punto clave de esta reflexión: la conversión de una rejilla en un permutación. Pero para atenernos a lo expuesto en el enunciado, vamos tratar el problema de codificar con una rejilla compuesta de huecos. Tenemos que recorrer los huecos de arriba a abajo y de izquierda a derecha rellenándolos con las sucesivas letras del mensaje. Pero, ¿cómo sabemos que estamos recorriendo los huecos en ese orden? Podríamos obligar a que la ristra de huecos que define una rejilla esté reordenada de forma que, al recorrerla secuencialmente, visitemos los huecos en el orden adecuado. Es fácil construir rejillas ateniéndose a esta restricción; el problema viene a la hora de girar. Si recorremos la ristra de huecos girando uno por uno y conservando su orden, el resultado ya no cumple la restricción. Si giramos la rejilla del ejemplo del enunciado, obtenemos la ristra $((3, 0), (3, 1), (1, 2), (1, 3))$; al recorrerla secuencialmente, visitamos de izquierda a derecha y de abajo a arriba. Esto ocurre siempre, y reordenar los huecos para que cumplan la restricción merece alguna consideración.

Llegados a este punto, tenemos dos alternativas: o bien decidimos hacer una interpretación laxa del enunciado y concedemos que las rejillas rotadas se visitan en otro orden, o bien buscamos otra

implementación que nos facilite este detalle. La primera opción es perfectamente razonable; lo único realmente importante en cuanto a la forma de recorrer los huecos es mantener la consistencia entre cifrado y descifrado. Pero aquí optaremos por la segunda, porque no podemos dejar que un enunciado nos derrote.

Nuestra representación de las rejillas será la matriz de ocupación, la matriz que en el enunciado está marcada con (5.1).

```
typedef struct Rejilla {
  int lado;
  int rotacion[maxLado][maxLado];
} Rejilla;
```

La rejilla del ejemplo del enunciado queda así:

```
Rejilla const rejillaEjemplo
= { 4,
    { {0, 2, 3, 3},
      {0, 2, 1, 1},
      {3, 3, 0, 2},
      {1, 1, 0, 2}
    }
};
```

Con esta representación es muy fácil convertir una rejilla en una permutación respetando el orden prescrito. Supongamos que tenemos una rejilla de lado $l$. Vamos a recorrerla de arriba a abajo y de izquierda a derecha. El primer 0 que encontremos es el hueco en que debemos escribir la primera letra de nuestro mensaje, en el segundo 0, la segunda letra, etc. ¿Cuántos 0 tiene una rejilla de lado $l$? Justamente $l^2/4$, cantidad a la que llamaremos $k$. En ese recorrido, en el hueco del primer 1 que encontremos hay que escribir la letra de la posición $k$, en el segundo 1, la letra $k+1$, etc. En general, en el $j$-ésimo $i$ que encontremos en una rejilla hay que escribir la letra de la posición $i \cdot k + j$ (suponiendo que $j$ empieza a contar desde 0). Por otra parte, como el mensaje cifrado con una rejilla se muestra recorriéndola también de arriba a abajo y de izquierda a derecha, resulta que lo escrito en el hueco $(x, y)$, acaba en la posición $x \cdot l + y$. En definitiva, una rejilla que tiene en la posición $(x, y)$ el $j$-ésimo $i$ define una permutación que envía el elemento de la posición $x \cdot l + y$ a la posición $i \cdot k + j$.

```
void hazPermutacion(Permutacion& perm, Rejilla const& rejilla) {
  int const area = rejilla.lado * rejilla.lado;
  int const salto = area / 4;
  int posiciones[] = {0, salto, 2*salto, 3*salto};
  perm.longitud = area;
  for (int x = 0; x < rejilla.lado; x++) {
    for (int y = 0; y < rejilla.lado; y++) {
      perm.destino[posiciones[rejilla.rotacion[x][y]]] = x*rejilla.lado + y;
      posiciones[rejilla.rotacion[x][y]]++;
    }
  }
}
```

### 5.17 3 Construcción de rejillas

Vamos a intentar construir de forma aleatoria rejillas de lado par. Aunque algunas de las afirmaciones que haremos son incorrectas para casillas con lado impar, todo el código que hemos construido hasta

ahora y que vamos a construir funciona para cualquier paridad; trata tú de argumentar por tu cuenta el motivo de esta buena suerte.

La rejilla en discordia tiene lado $2l$. Para que el resultado sea correcto, sabemos que tenemos que garantizar las condiciones de *intersección vacía* y de *exhaustividad*. Supongamos que en cierto momento tenemos una rejilla que cumple la condición de intersección vacía pero no la de exhaustividad. Entonces, al unir los huecos de las cuatro rotaciones resulta que hay alguna posición no perforada. Basta perforar cualquiera de estas posiciones para conseguir una rejilla que sigue cumpliendo la condición de intersección vacía. Si iteramos este proceso, necesariamente terminaremos con una rejilla que además cumple la condición de exhaustividad; es más, lo conseguiremos justamente en $l^2$ pasos empezando con una rejilla sin un solo agujero.

Para implementar la idea del párrafo anterior vamos a fijar el convenio de que una casilla con $-1$ indica una posición sin perforar en una `Rejilla`:

```
void vaciaRejilla(Rejilla& rejilla) {
  for (int i = 0; i < rejilla.lado; i++) {
    for (int j = 0; j < rejilla.lado; j++) {
      rejilla.rotacion[i][j] = -1;
    }
  }
}
```

Además, cada vez que perforemos un hueco, no sólo vamos a poner un 0 en la casilla elegida, sino un 1 en el hueco rotado 90 grados... Así, la rejilla irá acumulado las cuatro versiones giradas:

```
void rejillaAleatoria(Rejilla& rejilla, int const lado) {
  int const area = lado*lado;
  rejilla.lado = lado;
  vaciaRejilla(rejilla);
  int const totalHuecos = (area + 3) / 4;
  Hueco actual = {0, 0};
  for (int i = 0; i < totalHuecos; i++) {
    visitableAleatoria(actual, area - 4*i, rejilla);
    for (int j = 0; j < 4; j++) {
      rejilla.rotacion[actual.fila][actual.columna] = j;
      rota(actual, lado);
    }
  }
}
```

Gracias a que nuestras `Rejillas` son la unión de los cuatro giros, para encontrar una posición invisible para todas, basta con encontrar un $-1$. Buscarlo de forma aleatoria es un poco más difícil porque hay que elegirlo sólo entre los $-1$ que `quedan`. Por eso hay que recorrer la rejilla revisando $-1$ hasta que demos con el que está en la posición que la suerte ha querido:

```
void visitableAleatoria(Hueco& hueco, int const quedan, Rejilla const& rejilla) {
  int saltos = random() % quedan;
  for (int i = 0; i < rejilla.lado; i++) {
    for (int j = 0; j < rejilla.lado; j++) {
      if (rejilla.rotacion[i][j] == -1) {
        if (saltos == 0) {
          hueco.fila = i;
```

```
            hueco.columna = j;
            return;
          }
          saltos--;
        }
      }
    }
  }
```

## 5.19 Implementación de números grandes

251 255

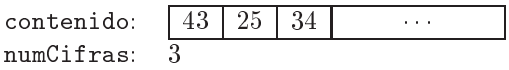
***Definición de tipo*** Todo el mundo piensa en los números enteros como números naturales a los que se les ha añadido una etiqueta que es el signo. Y así definimos también nosotros los enteros:

```
typedef enum {negativo, positivo} Signo;
typedef struct Entero {
  Natural natural;
  Signo signo;
} Entero;
```

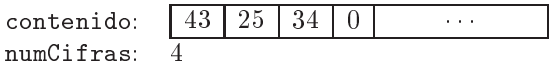
Usualmente, representamos los números naturales como secuencias de cifras en base 10. Ahora también un natural será una secuencia de enteros, pero en la base que consideremos adecuada. Implementaremos la secuencia con un array más un contador que nos indica su ocupación; fijamos el tamaño del array en la constante `longitudMaximaNatural`, y su contenido serán las cifras, es decir, enteros en un rango que va desde 0 hasta `base` $-$ 1. En su momento escogeremos el valor de la `base`. Por ahora lo único que necesitamos es que sea un número representable con los enteros del lenguaje; cuando diseñemos las operaciones iremos viendo las restricciones necesarias para `base`:

```
typedef struct Natural {
  unsigned int contenido[longitudMaximaNatural];
  int numCifras;
} Natural;
```

Todos los naturales tendrán al menos una cifra por lo que asumiremos que `numCifras` $\geq 1$. Representaremos los naturales al revés de lo usual; la cifra menos representativa estará en la posición 0. Por ejemplo, si la base elegida fuera 100, el número 342543 vendría representado así:




Además, no admitiremos que la cifra más representativa del natural sea cero (salvo si el natural es el propio 0). Por tanto, nunca surgirá la siguiente representación para el entero anterior:



***Operadores de comparación*** Las comparaciones entre enteros se reducen de forma natural a comparaciones entre naturales:

$$a = b \quad \Leftrightarrow \quad |a| = |b| = 0 \quad \text{o} \quad \begin{cases} |a| = |b| & \text{y} \\ \text{signo}(a) = \text{signo}(b) \end{cases}$$

$$a \leq b \quad \Leftrightarrow \quad |a| = |b| = 0 \quad \text{o} \quad \begin{cases} a \leq 0 \text{ y } b \geq 0\,, \\ a \leq 0,\ b \leq 0 \text{ y } |a| \geq |b| \quad \text{o} \\ a \geq 0,\ b \geq 0 \text{ y } |a| \leq |b| \end{cases}$$

Estas funciones se implementan mediante una simple distinción de casos, por lo que nos centraremos directamente en las comparaciones entre naturales.

La comparación de naturales de distinto número de cifras es sencilla: el más largo es el mayor. Cuando ambos tienen el mismo número de cifras, buscamos la primera que sea distinta en ambos números empezando por la más significativa; si no encontramos ninguna cifra distinta, los números son iguales, y en caso contrario el mayor será aquél cuya primera cifra distinta sea mayor:

```
bool operator<=(Natural const& a, Natural const& b) {
  if (a.numCifras > b.numCifras) {
    return false;
  } else if (a.numCifras < b.numCifras) {
    return true;
  } else {
    int pos = posPrimeraDistinta(a, b);
    return pos == -1 || a.contenido[pos] <= b.contenido[pos];
  }
}
bool operator>=(Natural const& a, Natural const& b) {
  if (a.numCifras > b.numCifras) {
    return true;
  } else if (a.numCifras < b.numCifras) {
    return false;
  } else {
    int pos = posPrimeraDistinta(a, b);
    return pos == -1 || a.contenido[pos] >= b.contenido[pos];
  }
}
bool operator==(Natural const& a, Natural const& b) {
  if (a.numCifras != b.numCifras) {
    return false;
  } else {
    int pos = posPrimeraDistinta(a, b);
    return pos == -1;
  }
}
```

Los operadores `>`, `<` y `!=` se definen de forma trivial negando los operadores `<=`, `>=` y `==` respectivamente.

Hemos usado una función que da la posición de la primera cifra distinta, y si todas las cifras son iguales, la función devolverá $-1$. Para ello recorremos el array empezando por la última posición ocupada y avanzando hacia el inicio:

```
int posPrimeraDistinta(Natural const& a, Natural const& b) {
  int i = a.numCifras-1;
  while (i >= 0 && a.contenido[i] == b.contenido[i])
    i--;
  return i;
}
```

***Suma y resta*** La suma de enteros se reduce a sumas y restas de naturales. Si los sumandos tienen el mismo signo se suman los naturales contenidos, y se mantiene el signo. Si tienen signo contrario se resta al natural mayor el menor y el signo es el del que contenga el natural mayor. Por tanto, usando funciones de suma y resta de naturales, la suma de enteros se reduce a una simple distinción de casos que omitimos aquí. La resta entre enteros se reduce a la suma de enteros: se cambia de signo el sustraendo y se invoca la operación de suma. Se deja también al lector la implementación de esta operación. Nos centraremos aquí en la suma y la resta de naturales.

Realizaremos el algoritmo de suma entre naturales que nos enseñaron en la escuela: sumar cifra a cifra; cuando la suma de ambas cifras supere (o iguale) la base, restamos la base a la cifra resultado y nos llevamos una para la suma siguiente. Para que sean más sencillos los cálculos completaremos el natural de menos cifras con ceros a la izquierda. Como deseamos trabajar sobre los parámetros de entrada, modificándolos, debemos hacer una copia de los mismos:

```
Natural operator+(Natural const& a, Natural const& b) {
  Natural resultado;
  Natural auxA = a;
  Natural auxB = b;
  int maximoCifras =
    auxA.numCifras < auxB.numCifras ?  auxB.numCifras :  auxA.numCifras;
  ⟨Completar auxA con ceros hasta maximoCifras⟩;
  ⟨Completar auxB con ceros hasta maximoCifras⟩;
  for (int i = 0; i < maximoCifras; i++) {
    ⟨Sumar las cifras en la posición i de auxA y auxB⟩
  }
  ⟨Establecer el número de cifras de resultado⟩
  return resultado;
}
```

Para igualar la longitud de `auxA` y `auxB`, recurrimos a un sencillo procedimiento que añade ceros, desde la última cifra de cada entero. Para sumar las cifras de la posición `i` debemos realizar la suma de ambas; pero esta suma puede sobrepasar a la base; en este caso debemos restar la base y llevarnos una para la cifra siguiente. Es posible que nos llevemos una de sumar la columna anterior: usaremos una variable entera `meLlevo` para este fin; debemos definirla con el valor inicial `0` antes de entrar en el bucle. Con todo esto la suma de las cifras de la posición `i` se refina así:

```
unsigned int suma = auxA.contenido[i] + auxB.contenido[i] + meLlevo;
if (suma >= base) {
  suma = suma-base;
  meLlevo = 1;
} else {
  meLlevo = 0;
}
resultado.contenido[i] = suma;
```

Por último debemos decidir cuál es el tamaño del nuevo número; para ello debemos tener en cuenta si nos llevamos alguna de la última suma. Si no nos llevamos ninguna, el tamaño del natural será el del natural más largo de los parámetros de entrada, a saber, `maximoCifras`; en caso contrario será una unidad mayor. En este último caso debemos tener cuidado porque podemos llegar a sobrepasar el tamaño máximo de los naturales. Hemos dejado al lector la elección del código en caso de que ocurra esta incidencia:

```
resultado.numCifras = maximoCifras;
if (meLlevo != 0) {
  resultado.numCifras++;
  if (resultado.numCifras > longitudMaximaNatural) {
    ⟨Se ha sobrepasado la capacidad de los enteros⟩
  }
  resultado.contenido[maximoCifras] = meLlevo;
}
```

Falta comentar un par de detalles que serán de importancia a la hora de escoger una base adecuada para nuestro problema. En primer lugar estamos sumando cifras,

```
auxA.contenido[i] + auxB.contenido[i] + meLlevo;
```

y ese resultado se debe reflejar en los enteros subyacentes, por lo que se ha de cumplir lo siguiente,

$$\texttt{auxA.contenido[i]} + \texttt{auxB.contenido[i]} + \texttt{meLlevo} \leq \texttt{UINT\_MAX}$$

lo que se consigue siempre que se tenga que

$$2 \cdot \texttt{base} - 1 \leq \texttt{UINT\_MAX}$$

Pasamos ahora a realizar la resta entre naturales. En primer lugar supondremos que los operandos cumplen las condiciones de la resta de naturales. Como el primero es mayor o igual que el segundo, tiene al menos tantas cifras como él, de forma que sólo habrá que completar con ceros este último. La resta es similar a la suma; en este caso nos llevamos una cuando la cifra del minuendo es inferior a la del sustraendo (más lo que nos llevábamos).

```
Natural operator-(Natural const& a, Natural const& b) {
  Natural resultado;
  Natural auxA = a;
  Natural auxB = b;
  completaConCeros(auxB, auxA.numCifras);
  int meLlevo = 0;
  for (int i = 0; i < auxA.numCifras; i++) {
    int resta = auxA.contenido[i] - (auxB.contenido[i]+meLlevo);
    if (resta < 0) {
      resta = resta + base;
      meLlevo = 1;
    } else {
      meLlevo = 0;
    }
    resultado.contenido[i] = resta;
  }
  ⟨Calcular el número de cifras de resultado⟩
  return resultado;
}
```

Antes se seguir conviene comprobar que la operación

```
auxA.contenido[i] - (auxB.contenido[i]+meLlevo)
```

está dentro del rango del tipo `int`, cosa que ocurre si $2 \cdot \texttt{base} - 1 \leq \texttt{UINT\_MAX}$: por un lado tenemos que $2 \cdot \texttt{INT\_MAX} + 1 = \texttt{UINT\_MAX}$, de lo que deducimos $\texttt{base} - 1 \leq \texttt{INT\_MAX}$; por otro tenemos que $-(1 + \texttt{INT\_MAX}) = \texttt{INT\_MIN}$; de todo esto podemos deducir fácilmente que, si $0 \leq x, y < \texttt{base}$, también se cumple esto:

$$\texttt{INT\_MIN} \leq x - (y + 1) \leq \texttt{INT\_MAX}$$

Falta saber cómo resolvemos el problema de calcular el número de cifras de `resultado`. Una posibilidad sería recorrer el contenido de `resultado` para encontrar la primera cifra que no es cero. Pero podemos dar una solución mejor si nos acordamos de esta posición cada vez que realizamos una resta básica. Para ello introducimos una variable nueva `maxPosCifraNoCero` en la que llevamos cuenta de la última operación realizada distinta de cero. El valor inicial de esta variable puede ser `0` puesto que estamos suponiendo que todos los enteros tienen al menos una cifra. Así antes de entrar en el bucle debemos introducir la siguiente línea:

```
int maxPosCifraNoCero = 0;
```

En el cuerpo del bucle debemos actualizar este valor siempre que el resultado de una resta no sea `0`. Para ello, en el cuerpo del bucle debemos introducir después del cálculo de `resta` las siguientes líneas:

```
if (resta != 0) {
  maxPosCifraNoCero = i;
}
```

Así antes del `return` debemos poner la línea

```
resultado.numCifras = maxPosCifraNoCero+1;
```

***Por 2, entre 2*** La multiplicación por 2 es bastante sencilla. Cualquier natural $n$ lo podemos expresar de la siguiente forma

$$n = \texttt{base} \cdot b + a$$

donde $0 \leq a < \texttt{base}$ es su cifra menos representativa cuando se representa en `base`. Ahora, si multiplicamos por 2 el miembro derecho, tenemos lo siguiente:

$$2 \cdot (\texttt{base} \cdot b + a) = \texttt{base} \cdot (2 \cdot b) + 2 \cdot a$$

Si $2 \cdot a < \texttt{base}$ la fórmula nos sugiere que, para multiplicar $n$ por 2, hemos de multiplicar $a$ por 2 y luego seguir multiplicando el resto del número. Aunque eso no es cierto si $2 \cdot a \geq \texttt{base}$, en este caso tenemos que $0 \leq 2a - \texttt{base} < \texttt{base}$, y también se cumple lo siguiente,

$$\texttt{base} \cdot (2 \cdot b) + 2 \cdot a - \texttt{base} + \texttt{base} = \texttt{base} \cdot (2 \cdot b + 1) + 2 \cdot a - \texttt{base}$$

lo que nos sugiere que podemos hacer algo similar: la cifra menos representativa del resultado es $2 \cdot a - \texttt{base}$ y nos llevamos una para seguir multiplicando. En resumen, basta con multiplicar todas las cifras por 2 empezando por las menos significativas. Si alguna de esas multiplicaciones es mayor o igual que la base que estamos considerando, habrá que restar dicha base al resultado obtenido y llevarnos una para la siguiente cifra. Si en la última multiplicación no nos llevamos ninguna, el número de cifras del resultado es el mismo que el del número de entrada; en caso contrario tendrá una cifra más que deberá ser obligatoriamente 1:

```
Natural multiplicaPorDosNatural(Natural const& a) {
  Natural resultado;
  int meLlevo = 0;
  for (int i = 0; i < a.numCifras; i++) {
    unsigned int mult = a.contenido[i]*2 + meLlevo;
    if (mult >= base) {
      mult = mult-base;
      meLlevo = 1;
    } else {
      meLlevo = 0;
    }
    resultado.contenido[i] = mult;
  }
  if (meLlevo == 0) {
    resultado.numCifras = a.numCifras;
  } else {
    resultado.numCifras = a.numCifras+1;
    if (resultado.numCifras > longitudMaximaNatural) {
      ⟨Se ha sobrepasado la capacidad de los enteros⟩
    }
    resultado.contenido[a.numCifras] = meLlevo;
  }
  return resultado;
}
```

La idea para dividir un número entre 2 consiste en invertir el proceso anterior y empezar a dividir las cifras más significativas entre 2; pero ese proceso puede ser, en general, más complejo. Sin embargo, si la base es par se puede comprobar fácilmente que las siguientes fórmulas son ciertas:

$$\begin{array}{ll} \text{si } b \text{ es par} & \lfloor (\texttt{base} \cdot b + a)/2 \rfloor = \texttt{base} \cdot \lfloor b/2 \rfloor + \lfloor a/2 \rfloor \\ \text{si } b \text{ es impar} & \lfloor (\texttt{base} \cdot b + a)/2 \rfloor = \texttt{base} \cdot \lfloor b/2 \rfloor + \lfloor (\texttt{base} + a)/2 \rfloor \end{array}$$

Así, podemos invertir el proceso anterior y empezar a dividir las cifras más significativas; si la cifra era impar nos llevábamos una para la cifra anterior, teniendo en cuenta que llevarnos una significa sumar la base. Además, si $a < \texttt{base}$, se tiene que $\lfloor (\texttt{base} + a)/2 \rfloor < \texttt{base}$, por lo que todas las cifras que van saliendo son correctas. Según el caso, el número de cifras del resultado es...

- ... igual al número de cifras del argumento, si la primera división era distinta de cero o bien si siendo cero el número original sólo tenía una cifra; el resultado en este caso es cero y hemos dicho que cero tiene una cifra;
- ... una menos, en caso contrario.

```
Natural dividePorDosNatural(Natural const& a) {
  Natural resultado;
  int meLlevo = 0; // resto de la división anterior
  for (int i = a.numCifras-1; i >= 0; i--) {
    unsigned int cifra = a.contenido[i] + meLlevo*base;
    resultado.contenido[i] = cifra / 2;
    meLlevo = cifra % 2;
```

```
  }
  if (resultado.contenido[a.numCifras-1] != 0 || a.numCifras == 1)
    resultado.numCifras = a.numCifras;
  else
    resultado.numCifras = a.numCifras-1;
  return resultado;
}
```

Y ya sólo falta comprobrar que las expresiones `a.contenido[i] + meLlevo*base` (en la división) y `a.contenido[i]*2 + meLlevo` (en la multiplicación) están dentro del rango `unsigned int`, cosa fácil de hacer con la restricción $2 \cdot \texttt{base} - 1 \leq \texttt{UINT\_MAX}$.

Como ya hemos asumido que la base es par, la comprobación de la paridad es bastante sencilla puesto que sólo tenemos que saber si la cifra menos significativa es par o no:

```
bool esNaturalImpar(Natural a) {
  return (a.contenido[0] % 2) != 0;
}
```

***Multiplicación y división*** La multiplicación de enteros también se reduce a la de naturales: se multiplican los naturales contenidos en los enteros y se aplica la conocida regla de los signos. Si los dos enteros tienen el mismo signo, el signo del resultado es positivo; será negativo en caso contrario. Al igual que en los casos anteriores dejamos la implementación de esta sencilla función al lector. La división entre enteros también se reduce a la de naturales; como esta operación no se suele especificar, empezamos definiéndola; el algoritmo admitirá dos enteros de entrada `dividendo` y `divisor` tales que $\texttt{divisor} \neq 0$, y devolverá dos enteros de salida `cociente` y `resto`, tales que se verifique lo siguiente:

$$\texttt{dividendo} = \texttt{divisor} \cdot \texttt{cociente} + \texttt{resto},\ 0 \leq \texttt{resto} < |\texttt{divisor}|$$

Si ambos enteros son positivos (o más precisamente no negativos) basta con realizar la división entera y devolver los naturales calculados con signo positivo. Pasamos a desarrollar los tres casos restantes:

- $\texttt{divisor} > 0,\ \texttt{dividendo} < 0$. Si realizamos la división (entre naturales) del valor absoluto del `dividendo` entre el `divisor`, obtenemos:

  $$|\texttt{dividendo}| = -\texttt{dividendo} = \texttt{divisor} * c + r$$

  Y cambiando de signo todo, tenemos lo siguiente:

  $$\texttt{dividendo} = -\texttt{divisor} * c - r = \texttt{divisor} * (-c) - r$$

  Si $r = 0$ ya hemos acabado: el `cociente` buscado será $-c$ y el `resto` el propio $r$. Si $r \neq 0$ sumamos y restamos el `divisor` en el último término,

  $$\texttt{divisor} * (-c) - r - \texttt{divisor} + \texttt{divisor} = \texttt{divisor} * (-c - 1) + \texttt{divisor} - r$$

  y ahora tenemos $0 \leq \texttt{divisor} - r < \texttt{divisor}$ por lo que el `cociente` buscado será $-(c + 1)$ y el `resto` será $\texttt{divisor} - r$.

- $\texttt{divisor} < 0,\ \texttt{dividendo} > 0$. Realizando la división de naturales del `dividendo` entre el valor absoluto del `divisor`, obtenemos:

  $$\texttt{dividendo} = |\texttt{divisor}| * c + r = \texttt{divisor} * (-c) + r$$

  Así, el `cociente` buscado será $-c$ y el `resto` $r$.

- `divisor` $< 0$, `dividendo` $< 0$. Si realizamos la división de naturales del valor absoluto del `dividendo` entre el del `divisor`, obtenemos:

$$-\texttt{dividendo} = |\texttt{dividendo}| = |\texttt{divisor}| * c + r = -\texttt{divisor} * c + r$$

Cambiando de signo ambos miembros, tenemos:

$$\texttt{dividendo} = \texttt{divisor} * c - r$$

Si $r = 0$ el cociente buscado será $c$ y el resto $r$; en caso contrario sumamos y restamos el divisor al miembro derecho,

$$\texttt{divisor} * c - r + \texttt{divisor} - \texttt{divisor} = \texttt{divisor} * (c - 1) + \texttt{divisor} - r$$

y, puesto que $0 \leq \texttt{divisor} - r < \texttt{divisor}$ el cociente buscado será $c - 1$ y el resto $\texttt{divisor} - r$.

Juntado todo lo anterior obtenemos el siguiente procedimiento:

```
void divide(Entero const& dividendo, Entero const& divisor,
            Entero& cociente, Entero& resto) {
  divide(dividendo.natural, divisor.natural, cociente.natural, resto.natural);
  resto.signo = positivo; // el resto es positivo en cualquier caso;
  if (dividendo.signo == positivo && divisor.signo == positivo) {
    cociente.signo = positivo;
  } else if (dividendo.signo == negativo && divisor.signo == positivo) {
    if (resto.natural != ⟨natural 0⟩) {
      cociente.natural = cociente.natural + ⟨natural 1⟩;
      resto.natural = divisor.natural + resto.natural;
    }
    cociente.signo = negativo;
  } else if (dividendo.signo == positivo && divisor.signo == negativo) {
    cociente.signo = negativo;
  } else { // (dividendo.signo == negativo && dividendo.signo == negativo)
    if (resto.natural != ⟨natural 0⟩) {
      cociente.natural = cociente.natural + ⟨natural 1⟩;
      resto.natural = divisor.natural - resto.natural;
    }
    cociente.signo = positivo;
  }
}
```

Antes de realizar las operaciones de multiplicación y división entre naturales, nos centramos en varias funciones auxiliares que han aparecido: la función que devuelve el natural 0,

```
Natural ceroNatural() {
  Natural resultado;
  resultado.contenido[0] = 0;
  resultado.numCifras = 1;
  return resultado;
}
```

y la función que devuelve el natural 1:

```
Natural unoNatural() {
  Natural resultado;
  resultado.contenido[0] = 1;
  resultado.numCifras = 1;
  return resultado;
}
```

Y vamos ahora con la multiplicación y división de enteros. Se podría pensar en hacer algoritmos usando las restas y sumas que ya tenemos de una forma sencilla; por ejemplo, el de la división se podría hacer de la siguiente forma:

```
cociente = 0; resto = dividendo;
while (resto > divisor) {
  cociente = cociente+1;
  resto = resto-divisor
}
```

Pero ese algoritmo es altamente ineficiente, como puedes comprobar tú mismo con un dividendo grande y un divisor pequeño; por tanto habrá que buscar algoritmos mejores. Para el producto usaremos el algoritmo de *multiplicación a la rusa* (véase el ejercicio 2.2), que hace uso de sumas y restas cualesquiera y de multiplicaciones y divisiones por 2:

```
Natural operator*(Natural const& a, Natural const& b) {
  Natural auxA = a;
  Natural auxB = b;
  Natural resultado = ceroNatural();
  Natural cero = ceroNatural();
  while (auxA != cero) {
    if (esNaturalImpar(auxA)) {
      resultado = resultado + auxB;
    }
    auxB = multiplicaPorDosNatural(auxB);
    auxA = dividePorDosNatural(auxA);
  }
  return resultado;
}
```

La división es más compleja. Para poder realizar bien este procedimiento conviene detallar el objetivo:

$$\texttt{dividendo} = \texttt{divisor} \cdot \texttt{cociente} + \texttt{resto},\ 0 \leq \texttt{resto} < \texttt{divisor}$$

¿Cómo lo conseguiremos? Haremos un bucle con una variable nueva `w`, que en principio será mayor que el divisor, de forma que en cada vuelta del bucle se verifique esto:

$$\texttt{dividendo} = \texttt{w} \cdot \texttt{cociente} + \texttt{resto},\ 0 \leq \texttt{resto} < \texttt{w}$$

Obviamente, pararemos cuando `divisor` $=$ `w`. Hemos dicho que `w` será mayor que `divisor`, por lo que deberemos ir disminuyendo `w`, y lo haremos como en el algoritmo anterior: dividiendo por 2. Además nos las apañaremos para que siempre que haya que dividir, `w` sea par. Si dividimos `w` por 2, para que siga siendo cierta la primera igualdad bastaría con multiplicar el cociente por 2. En realidad ése es el caso cuando `resto` $< \lfloor \texttt{w}/2 \rfloor$, puesto que sigue siendo cierta la desigualdad de la derecha; el problema lo tenemos precisamente cuando `resto` $\geq \lfloor \texttt{w}/2 \rfloor$. ¿Cómo se arregla este caso? Tenemos lo siguiente,

$$0 \leq \lfloor \texttt{w}/2 \rfloor \leq \texttt{resto} < w$$

y entonces, se cumple también lo siguiente:

$$0 \leq \texttt{resto} - \lfloor \texttt{w}/2 \rfloor < \texttt{w} - \lfloor \texttt{w}/2 \rfloor = \lfloor \texttt{w}/2 \rfloor$$

Ahora, para arreglar la igualdad de la izquierda, observamos lo siguiente:

$$\begin{aligned} \texttt{dividendo} &= \texttt{w} \cdot \texttt{cociente} + \texttt{resto} \\ &= \texttt{cociente} \cdot 2 \cdot \lfloor \texttt{w}/2 \rfloor + \texttt{resto} \\ &= (2 \cdot \texttt{cociente}) \cdot \lfloor \texttt{w}/2 \rfloor + \texttt{resto} + \lfloor \texttt{w}/2 \rfloor - \lfloor \texttt{w}/2 \rfloor \\ &= \underbrace{(2 \cdot \texttt{cociente} + 1)}_{\text{nuevo } \texttt{cociente}} \cdot \lfloor \texttt{w}/2 \rfloor + \underbrace{\texttt{resto} - \lfloor \texttt{w}/2 \rfloor}_{\text{nuevo } \texttt{resto}} \end{aligned}$$

Y el algoritmo se podría escribir así,

```
void divide(Natural const& dividendo, Natural const& divisor,
            Natural& cociente, Natural& resto) {
  ⟨Establecer el valor inicial de w, cociente y resto⟩
  while (w != divisor) {
    if (⟨w/2⟩ <= resto) {
      cociente = ⟨2*cociente + 1⟩;
      resto = resto - ⟨w/2⟩;
    } else {
     cociente = ⟨2*cociente⟩;
    }
    w = ⟨w/2⟩;
  }
}
```

que luego se puede mejorar sacando la división fuera del bucle; además, como la duplicación del `cociente` se realiza en las dos ramas del condicional también se puede sacar fuera, quedando el algoritmo así:

```
while (w != divisor) {
  w = dividePorDosNatural(w);
  cociente = multiplicaPorDosNatural(cociente);
  if (w <= resto) {
    cociente = cociente + unoNatural();
    resto = resto - w;
  }
}
```

Faltan las asignaciones iniciales a `w`, `cociente` y `resto`. Para hacerlo conviene darse cuenta de la condición que hemos impuesto en cada vuelta del bucle, que en particular ha de cumplirse en la primera. Haciendo que `cociente` valga 0 y `resto` valga lo mismo que `dividendo`, la igualdad se cumple evidentemente. Falta dar un valor inicial a `w`. Sabemos que en toda vuelta del bucle debe tomar un valor par hasta que se sale, es decir `w` debe ser de la forma $2^n \cdot$ `divisor` para algún $n \geq 0$ y además `w` $>$ `resto` $=$ `dividendo`, por lo que `w` puede ser el primer natural que cumpla esa condición. Para calcularlo utilizaremos un bucle de forma que, inicialmente, `w` $=$ `divisor`, y en cada vuelta se duplica `w`; el bucle parará precisamente cuando `w` $>$ `dividendo`:

```
Natural w = divisor;
while (w <= dividendo) {
  w = multiplicaPorDosNatural(w);
}
cociente = ceroNatural(); resto = dividendo;
```

***Elección de la base*** Las restricciones que hemos encontrado a la base son éstas:

- Debe ser un número par, para que las divisiones se puedan hacer de forma sencilla.
- Debe ser lo mayor posible, para desperdiciar lo mínimo posible de memoria.
- El doble de la base menos uno debe ser menor o igual que el mayor entero representable.

Por último se ha de tener en cuenta que, si escogemos la base de forma que sea una potencia de 2, las operaciones con cifras se pueden hacer fácilmente mediante operaciones de bits. Supongamos que tenemos las siguientes constantes:

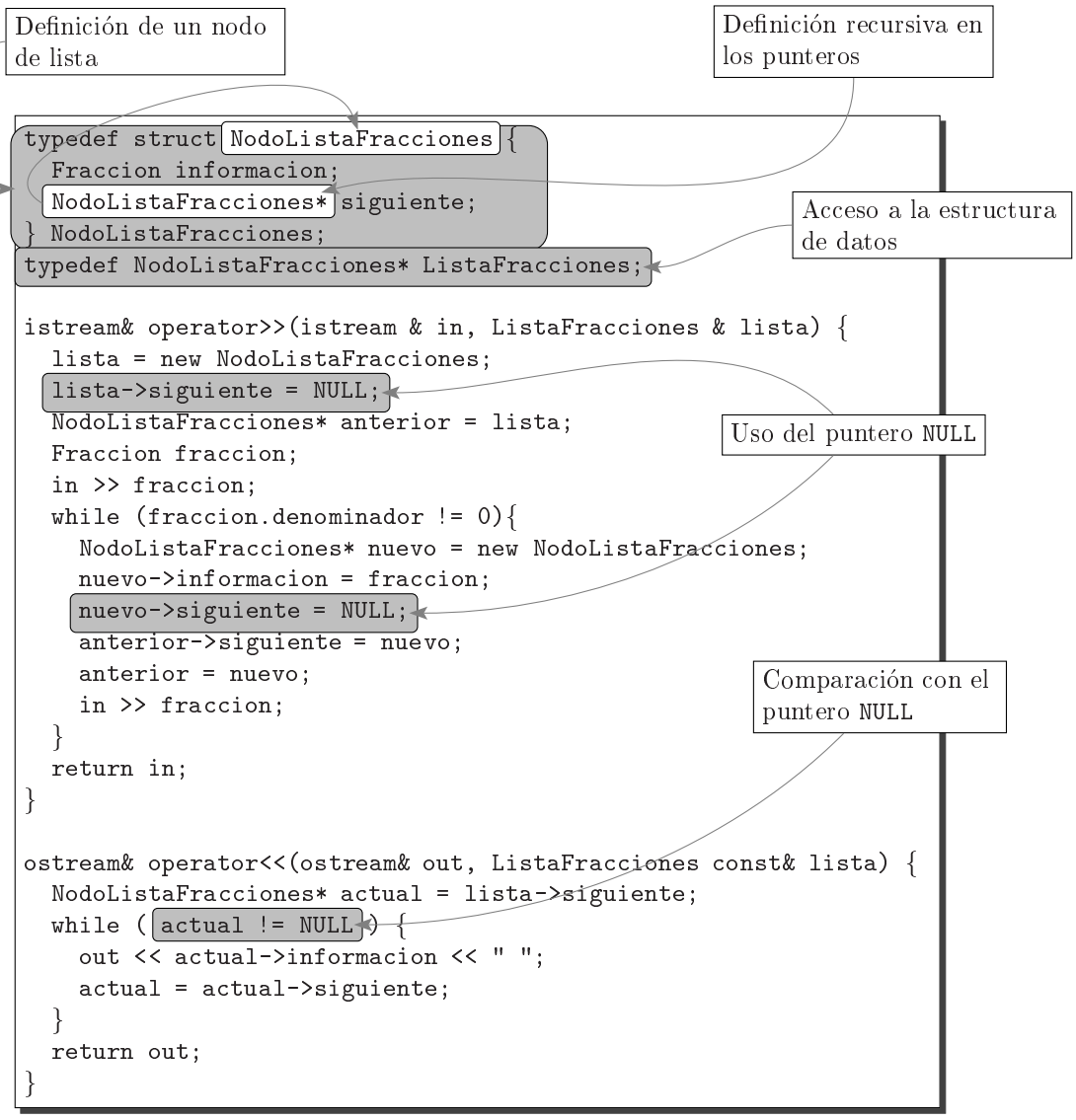
```
int const numBitsCifra = 31;
unsigned int const base = 1 << numBitsCifra;
```

Entonces la suma de dos cifras (más el acarreo) se puede hacer de la siguiente forma:

```
suma = auxA.contenido[i] + auxB.contenido[i] + meLlevo;
meLlevo = (suma & base) >> numBitsCifra;
suma = suma & ~base;
```

# 6 Capítulo SEIS

# Memoria dinámica




```
typedef struct NodoListaFracciones {
  Fraccion informacion;
  NodoListaFracciones* siguiente;
} NodoListaFracciones;
typedef NodoListaFracciones* ListaFracciones;

istream& operator>>(istream & in, ListaFracciones & lista) {
  lista = new NodoListaFracciones;
  lista->siguiente = NULL;
  NodoListaFracciones* anterior = lista;
  Fraccion fraccion;
  in >> fraccion;
  while (fraccion.denominador != 0){
    NodoListaFracciones* nuevo = new NodoListaFracciones;
    nuevo->informacion = fraccion;
    nuevo->siguiente = NULL;
    anterior->siguiente = nuevo;
    anterior = nuevo;
    in >> fraccion;
  }
  return in;
}

ostream& operator<<(ostream& out, ListaFracciones const& lista) {
  NodoListaFracciones* actual = lista->siguiente;
  while ( actual != NULL ) {
    out << actual->informacion << " ";
    actual = actual->siguiente;
  }
  return out;
}
```

## En este capítulo

Al igual que en el caso de la entrada y salida, hay dos formas de manipular la memoria dinámica en C++. La primera es herencia de C y está resuelta como es usual en este lenguaje: con una biblioteca de funciones, que en este caso se llama `malloc`. Permite un control muy fino, pero es fácil cometer deslices muy difíciles de detectar y que tendrán efectos catastróficos en nuestro programa.

La segunda está integrada en el lenguaje y es bastante más sencilla de usar y más segura. Nace para apoyar la programación orientada a objetos, que es la característica fundamental que distingue a C++ de C. Pero no se restringe a la parte de orientación a objetos del lenguaje, sino que se ha diseñado para que sustituya completamente a la librería `malloc`. Aún así, y aunque no es recomendable, es posible mezclar en un mismo programa ambos métodos. Obviamente en este libro usaremos exclusivamente la segunda forma.

## 6.1 Conceptos básicos

Como en otros lenguajes imperativos, la *memoria dinámica* surge de la cooperación entre dos partes del lenguaje: el concepto de *puntero* y la posibilidad de obtener memoria nueva de un espacio denominado *montículo*.

El concepto de puntero, de posición de memoria, se marca en C++ con un asterisco (`*`). El asterisco sirve tanto para indicar que un tipo es un puntero como para llegar al valor apuntado por un puntero.

Como operador de tipos, el asterisco es postfijo. El tipo puntero a `T` se escribe como `T*`. Así, para declarar una variable que sea un puntero a un entero basta con escribir lo siguiente:

```
int* punteroAEntero;
```

El asterisco como operador de expresiones es un operador prefijo. Así, para acceder al valor de una expresión `expr` que tiene tipo `T*` hay que escribir `*expr`; como es de esperar, esta última expresión tiene tipo `T`. Se denomina *desreferenciación* (o *indirección*) a esta operación. Una desreferencia es un elemento válido en la parte izquierda de una asignación,

```
*punteroAEntero = 10;
```

y también en la derecha:

```
int n = (*punteroAEntero) / 2;
```

Por supuesto, la ejecución de la asignación anterior justo después de la declaración de la variable `punteroAEntero` es un error garrafal; porque antes de poder referenciar un puntero es necesario que apunte a algo. Una variable de tipo puntero recién declarada, como cualquier otra variable, tiene un valor indeterminado y, por tanto, estará apuntando a una posición indeterminada. Si la desreferenciamos para obtener el valor al que apunta, el resultado será basura. Pero si la desreferenciamos para asignar a la posición a la que apunta, habremos modificado una zona de memoria desconocida y terminaremos notándolo, más adelante en la ejecución de nuestro programa, en forma de un comportamiento aberrante. Usar un puntero que no se ha definido es lo peor que le puede pasar a un programa.

Para obtener del montículo un nuevo espacio de memoria hay que utilizar el operador `new`. A este operador debe seguirle un tipo. Su comportamiento es el esperable: reserva la memoria necesaria para almacenar cualquier valor de ese tipo y devuelve un puntero a esta zona de memoria. Como ejemplo, para que la variable `punteroAEntero` apunte a una zona de memoria donde se puede almacenar un entero hay que hacer:

```
punteroAEntero = new int;
```

Una zona de memoria recién obtenida del montículo, al igual que una variable recién declarada, tiene

un valor indeterminado. Pero ahora que `punteroAEntero` ya apunta a una zona prevista para contener un entero, tiene completo sentido ejecutar la asignación siguiente:

```
*punteroAEntero = 10;
```

Así todo estará bien definido: la variable `punteroAEntero` tiene como valor una dirección de memoria legítima y preparada para contener un entero; además, hemos puesto el entero `10` en esa posición.

Se pueden copiar punteros, pasar punteros a subprogramas, devolver punteros desde una función, almacenar punteros en un registro o en un array, etc. Si escribrimos

```
int* otroPunteroAEntero = punteroAEntero;
```

tendremos dos variables apuntando a la misma dirección de memoria. La modificaciones que hagamos sobre esa dirección de memoria usando uno de los punteros será visible cuando accedamos con el otro.

Cuando una zona de memoria del montículo ya no está en uso, es conveniente indicarlo para que posteriormente se pueda reutilizar para albergar otros datos. El operador `delete` sirve para este propósito. Es una instrucción que debe ir seguida de una expresión cuyo resultado sea la dirección de memoria que deseamos devolver al montículo. Una vez que se ha devuelto una zona de memoria, ya no se puede utilizar, ni para leer de ella (porque posiblemente `delete` haya cambiado su valor para almacenar datos de administración interna), ni mucho menos para escribir (porque modificaríamos la información, supuestamente crítica, que haya podido registrarse en esa zona). Para evitar usar esa zona de memoria puede ser útil asignar el valor `NULL` (indefinido) al puntero borrado:

```
delete punteroAEntero;
punteroAEntero = NULL;
```

De todas formas es necesario tener cuidado, porque anular un puntero no preserva la dirección apuntada, sino sólo el acceso a la misma desde dicho puntero. Por ejemplo, las instrucciones anteriores han cortado el paso a `punteroAEntero`, pero esa misma dirección sigue siendo alcanzable ahora mediante `otroPunteroAEntero`.

Una zona de memoria reservada con `new` sólo se puede devolver una vez a la memoria libre con `delete`, sin importar que haya más de una variable apuntando a la misma zona.

## 6.2 Memoria dinámica y registros

Es difícil que los ejemplos del apartado anterior sean de utilidad. Utilizar el montículo para almacenar elementos de tamaño fijo es complicar innecesariamente lo que puede resolverse con variables locales. La memoria dinámica es un recurso útil cuando se aprovecha para almacenar estructuras de tamaño variable o de tamaño fijo pero desconocido hasta que no llega el momento de su construcción. Son las típicas estructuras de nodos enlazados y de arrays de tamaño variable. En este apartado veremos cómo definir y manejar estructuras de nodos enlazados en C++, y dedicaremos los dos apartados siguientes a la mezcla de arrays y memoria dinámica.

Las estructuras de nodos enlazados tienen necesariamente una definición recursiva: un nodo que contiene ciertos datos y un puntero a otro nodo que contiene datos y un puntero a otro nodo, etc. Como desafortunadamente la memoria de una computadora es finita, la concreción de uno de estos desplieges recursivos ha de tener un fin o un ciclo. Para marcar el fin se reserva un valor de puntero especial que se abstrae con la constante `NULL`; nunca es el resultado de un `new` salvo cuando no queda memoria en el sistema, contingencia que este operador indica devolviendo `NULL`.

Las estructuras recursivas simples se pueden definir de una sola vez en C++. La definición de los nodos de una lista enlazada normal sería así:

```
typedef struct Nodo {
  int valor;
  Nodo* sig;
} Nodo;
```

Si queremos dar un nombre más indicativo a `Nodo*` podemos añadir esto:

```
typedef Nodo* Lista;
```

Pero las estructuras recursivas múltiples necesitan un preámbulo con promesas para que el compilador no se queje al ver identificadores no definidos. Supongamos que queremos definir unas listas en donde se alternan nodos cuyo contenido es un valor entero o uno real; la definición

```
typedef struct Nodo1 {
  int valor;
  Nodo2* sig;
} Nodo1;
typedef struct Nodo2 {
  float valor;
  Nodo1* sig;
} Nodo2;
```

no satisface al compilador de C++ porque cuando usamos `Nodo2` por primera vez todavía no se ha definido. Se puede resolver ese conflicto de orden con la promesa de que habrá un tipo `Nodo2` que será un registro:

```
typedef struct Nodo2 Nodo2;
```

Para hacer el código más simétrico, se puede ensayar una definición como la que sigue:

```
typedef struct Nodo1 Nodo1;
typedef struct Nodo2 Nodo2;
struct Nodo1 {
  int valor;
  Nodo2* sig;
};
struct Nodo2 {
  float valor;
  Nodo1* sig;
};
```

Sigamos con el ejemplo de las listas enlazadas de enteros. Antes de ver alguna de las operaciones tradicionales, vamos a presentar un nuevo operador de C++. Supongamos que tenemos

```
Nodo* n = new Nodo;
```

y queremos rellenar sus campos. La forma correcta de hacerlo es ésta:

```
(*n).valor = 10;
(*n).sig   = NULL;
```

Los paréntesis alrededor de `*n` no se pueden eliminar porque la prioridad de la desreferenciación es más baja que la del acceso a los campos de un registro. Si escribimos

```
*n.valor = 10;
```

el compilador entenderá

```
*(n.valor) = 10;
```

que obviamente no es lo que queremos y además es incorrecto. Parecerá una locura que, para resolver una tarea tan normal como manipular los campos de un registro al que sólo podemos acceder con un puntero, haya que escribir unos paréntesis; pero es admisible porque hay un operador especial para esta tarea: el operador flecha `->`. La expresión `puntero->campo` es equivalente a `(*puntero).campo`. El ejemplo anterior se puede reescribir así:

```
n->valor = 10;
n->sig   = NULL;
```

Este operador es particularmente afortunado porque es un buen símil en texto de las flechas que, en los diagramas de memoria dinámica usuales, viajan de los punteros a los nodos apuntados.

Ahora ya sí podemos ver alguna de las operaciones usuales sobre listas enlazadas.

**Añadir al principio** Construcción de un nuevo nodo a la cabeza de una lista. No modifica la lista original que se pasa como segundo argumento.

```
Lista cons(int const valor, Lista sig) {
  Lista const l = new Nodo;
  l->valor = valor;
  l->sig   = sig;
  return l;
}
```

**Inserción al final** La inserción de un nodo al final de una lista puede modificar la lista original y, por tanto, hay que pasarla por referencia.

```
void alFinal(Lista& lista, int const valor) {
  Lista const nuevo = new Nodo;
  nuevo->valor = valor;
  nuevo->sig = NULL;
  Lista const ultimo = ultimoNodo(lista);
  if (ultimo == NULL) lista = nuevo;
  else ultimo->sig = nuevo;
}
```

La definición de la función `ultimoNodo`:

```
Lista ultimoNodo(Lista lista) {
  if (lista == NULL) return NULL;
  Lista aux = lista;
  while (aux->sig != NULL) aux = aux->sig;
  return aux;
}
```

**Insertador** La redefinición con sobrecarga del insertador para poder escribir listas.

```
ostream& operator << (ostream& out, Lista const lista) {
  for (Lista aux = lista; aux != NULL; aux = aux->sig) {
    out << aux->valor << ' ';
  }
  return out;
}
```

El interés de este código estriba en el uso del bucle `for` para recorrer una lista. Este uso es muy frecuente y demuestra la flexibilidad y legibilidad de dicho bucle cuando no se abusa de él.

## 6.3 Punteros y arrays

Ahora que se ha presentado el concepto de puntero en C++ ha llegado el momento de resolver el misterio de los arrays que se pasan siempre por referencia. La respuesta es breve pero tiene unas amplias implicaciones en todo el lenguaje: un array es indistiguible de un puntero. Parafraseando: un array de tipo `T` es indistinguible, salvo en detalles nimios, de un puntero a objetos de tipo `T`.

En un sentido, tenemos que un array es un puntero. Cuando pasamos un array a una función lo que realmente está haciendo el compilador es pasar un puntero al inicio de la memoria del array. Por esto, es imposible pasar un array por valor.

En el otro sentido, tenemos que un puntero es un array, afirmación con unas consecuencias bastante notorias. A un puntero se le puede aplicar el operador de indexación de los arrays:

```
int sumaArray(int* arr, int const len) {
  int sum = 0;
  for(int i = 0; i < len; i++) sum = sum + arr[i];
  return sum;
}
```

Sintácticamente no hay diferencia entre apuntar a un objeto o a un array de objetos. Podríamos llamar a la función `sumaArray` con un array:

```
int tresEnteros[] = {1, 2, 3};
int const sum = sumaArray(tresEnteros, 3);
```

o con un puntero a un solo entero:

```
int* const unEntero = new int;
*unEntero = 10;
int const sum = sumaArray(unEntero, 1);
```

Salvo por comentarios del programa o por el contexto, es imposible saber si un puntero apunta a un solo elemento o a un array. Lo curioso es que las dos llamadas anteriores a `sumaArray` son correctas, incluso si la función se hubiera declarado con un array de tamaño arbitrario,

```
int sumaArray(int arr[], int const len);
```

o con un array de un cierto tamaño fijo:

```
int sumaArray(int arr[N], int const len);
```

C++ ignora la restricción en el tamaño porque espera simplemente un puntero.

## 6.4 Manejo de arrays

Para estar a la altura de unos punteros que pueden comportarse como arrays, el operador `new` tiene que saber reservar memoria para una cantidad arbitraria de elementos consecutivos. Si queremos reservar memoria para almacenar un array de `n` elementos de tipo `T`, en la llamada a `new` hay que añadir detrás del tipo, rodeado por unos corchetes, el valor `n`:

```
T* arr = new T[n];
```

Para devolver una zona que se ha reservado para un array no se puede escribir simplemente,

```
delete arr;
```

sino que hay que informar al compilador de la condición de array (sin importar el tamaño) de esta zona añadiendo unos corchetes justo detrás de `delete`:

```
delete[] arr;
```

Esta capacidad de C++ para *construir* arrays de un tamaño arbitrario es muy útil en la práctica. La ilustraremos implementando unos arrays potencialmente infinitos.

Al construir un array potencialmente infinito hay que especificar el valor que inicialmente tendrán todas sus casillas. Como todas contienen lo mismo, no es necesario reservar memoria para guardar el valor de ninguna: sólo hay que recordar ese valor inicial. Pero cuando se empiecen a definir valores para las casillas habrá que guardarlos; lo haremos de la forma más sencilla posible. Supongamos que nos han dado valores nuevos para las casillas en las posiciones $k_0, \ldots, k_{n-1}$; supongamos además que $0 \leq k_i$ y que $k_i < k_{i+1}$. Lo más simple es reservar un array de tamaño $k_{n-1} + 1$ (los arrays se indexan desde 0) y guardar el valor dado para la posición $k_i$ en la casilla $k_i$. Las casillas que no hayan recibido un valor explícito se rellenarán con el valor inicial.

Cuando nos enfrentamos a la redefinición de una casilla $k$ surgen dos posibilidades. Primero, que $k \leq k_{n-1}$; entonces basta con cambiar la entrada $k$ del array; y segundo, que $k > k_{n-1}$; entonces habrá que reservar memoria para un array con $k + 1$ casillas, preservar el valor de las que ya existían, rellenar las nuevas con el valor inicial y poner en la posición $k$ el nuevo valor. Ya que tenemos que *extender* nuestro array, es mejor hacerlo con un poco de holgura: al menos se duplica el tamaño del array anterior, es decir, se construye un array en donde existe la posición $k$ y la posición $2k_{n-1} + 1$.

Veamos los detalles. En un array potencialmente infinito necesitamos un valor inicial, cuántos valores se están representando explícitamente y el array que se extiende:

```
typedef struct ArrayInfinito {
  double inicial;
  int longitud;
  double* datos;
} ArrayInfinito;
```

Construimos un array potencialmente infinito exactamente como lo hemos descrito anteriormente:

```
void creaArrayInfinito(ArrayInfinito& arri, double const inicial) {
  arri.inicial  = inicial;
  arri.longitud = 0;
  arri.datos    = NULL;
}
```

La consulta de un elemento es muy simple: si está entre los que tienen una definición explícita, accedemos al array `datos`, y en otro caso, devolvemos el valor inicial:

```
double valorEn(ArrayInfinito arri, int const i) {
  if (0 <= i && i < arri.longitud) return arri.datos[i];
  else return arri.inicial;
}
```

Hay una interrelación crítica entre este código, la construcción inicial y la definición que veremos enseguida: no se debería indexar el array `datos` si es `NULL`. Pero es `NULL` precisamente si no se ha definido ningún elemento, lo que a su vez es equivalente a que `longitud` sea 0. Por tanto, la condición del `if` tiene que ser necesariamente falsa.

La definición de una nueva componente es muy simple si delegamos al procedimiento `aseguraIndice` la tarea de asegurar que el índice que vamos a definir existe en el array:

```
void ponEn(ArrayInfinito& arri, int const i, double const valor) {
  aseguraIndice(arri, i);
  arri.datos[i] = valor;
}

void aseguraIndice(ArrayInfinito& arri, int const i) {
  if (i < arri.longitud) return;
  ⟨Construcción del nuevo array de datos⟩
  ⟨Preservación de los viejos datos⟩
  ⟨Asignación del valor por defecto⟩
  ⟨Lo nuevo sustituye a lo viejo⟩
}
```

La construcción del nuevo array de datos:

```
int const nuevaLongitud = max(i+1, 2*arri.longitud);
double* const nuevosDatos = new double[nuevaLongitud];
```

La preservación de los viejos datos:

```
if (arri.datos != NULL) {
  for (int i = 0; i < arri.longitud; i++) nuevosDatos[i] = arri.datos[i];
  delete[] arri.datos;
}
```

Para que las casillas no asignadas tengan el valor `arri.inicial`, hemos de asignar a las posiciones *nuevas* ese valor:

```
for (int i = arri.longitud; i < nuevaLongitud; i++) {
  nuevosDatos[i] = arri.inicial;
}
```

Y cómo lo nuevo sustituye a lo viejo:

```
arri.longitud = nuevaLongitud;
arri.datos    = nuevosDatos;
```

## 6.1 Cálculo de la matriz completa de mediotono de Judice-Jarvis-Ninke

Consulta el enunciado 3.26 porque éste es el momento para resolverlo de otra forma: calculando todos los elementos de la matriz simultáneamente. Haz un procedimiento que rellene una matriz con $2^n \times 2^n$ elementos con la matriz de mediotono $L_n$. Como en el ejercicio parejo, intenta tanto una solución recursiva como otra iterativa. (Véase la pista 6.1a.)

## 6.2 Polígonos

Un polígono puede representarse por la secuencia ordenada de sus vértices, que a su vez pueden indicarse mediante puntos en $\mathbb{R}^2$.

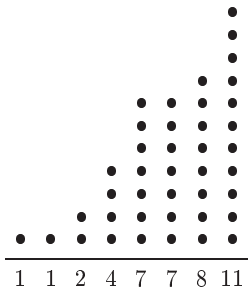
***Tipo de datos*** Diseña un tipo de datos que permita almacenar un polígono de un número de vértices cualquiera.

***Perímetro*** Escribe una función que calcule el perímetro de un polígono.

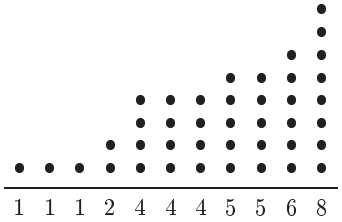
***Área*** Escribe una función que calcule el área de un polígono. (Véase la pista 6.2a.)

## 6.3 Listas conjugadas

Consideremos un *histograma* entero no decreciente representado mediante una lista en que se refieren las alturas de sus columnas:




Si hacemos que las columnas pasen a ser filas y viceversa, se obtienen el histograma y la lista representativa siguientes:




Se propone desarrollar un procedimiento que construya la lista transpuesta a partir de la primera; esto es:

$$[1\ 1\ 2\ 4\ 7\ 7\ 8\ 11] \rightsquigarrow [1\ 1\ 1\ 2\ 4\ 4\ 4\ 5\ 5\ 6\ 8]$$

(de longitud 8) (de longitud 11)

## 6.4 Tiempo de conexión a una máquina

Cierto sistema operativo admite la conexión simultánea de varios usuarios y recuerda cuándo ha ocurrido cada conexión y desconexión desde que se arrancó por primera vez. El sistema operativo nos da todos estos datos de muy buena gana y los guarda en un fichero organizado como ilustra el siguiente ejemplo:

```
Entra  david   18:24 9-2-2000
Sale   susana  18:20 9-2-2000
Sale   david   14:07 9-2-2000
Entra  david   10:46 9-2-2000
Entra  susana   9:12 9-2-2000
Sale   susana  12:07 8-1-2000
Entra  susana  19:12 7-1-2000
```

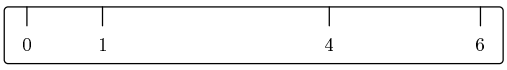
Cada línea refleja una contingencia. Salen ordenadas de más reciente a más antigua. Cada línea tiene (1) el tipo de suceso (`Entra` indica que ha empezado un conexión y `Sale` indica que ha terminado), (2) el nombre del usuario, (3) la hora en que ocurrió el evento y (4) la fecha.

Escribe una función que calcule el tiempo total que un cierto usuario ha estado conectado a este sistema operativo. Siguiendo el ejemplo anterior, y suponiendo que esas siete líneas reflejan toda la historia de esta máquina, Susana ha estado conectada 26 horas y 3 minutos (la última conexión duró 9 horas y 8 minutos y la anterior 16 horas y 55 minutos), y David está conectado actualmente, pero antes utilizó el sistema durante 3 horas y 21 minutos.

## 6.5 Reglas golombinas

330

Una regla *golombina* (en adelante RG) es una varilla con marcas en algunas posiciones enteras. Por ejemplo, en la siguiente RG esas posiciones señalan los puntos {0, 1, 4, 6}:




El interés que suscitan estas reglas es que, con unas pocas marcas, se puede medir cualquier cantidad que coincida con la distancia entre dos señales. Por ejemplo, con la regla anterior se pueden medir las cantidades 0 $(= 0 - 0)$, 1 $(= 1 - 0)$, 2 $(= 6 - 4)$, 3 $(= 4 - 1)$, 4 $(= 4 - 0)$, 5 $(= 6 - 1)$ y 6 $(= 6 - 0)$.

***Tipo de datos*** Define un tipo de datos apropiado para registrar una regla golombina, así como procedimientos de lectura y escritura.

***Distancias nuevas (en un paso)*** Escribe un subprograma que averigüe el conjunto de distancias nuevas medibles con una RG dada. Por ejemplo, para la regla anterior, resultan las distancias nuevas 2 $(= 6-4)$, 3 $(= 4 - 1)$ y 5 $(= 6 - 1)$.

***Distancias nuevas (iterando)*** El proceso de añadir nuevas medidas a las marcadas se puede iterar, pudiéndose completar muchas más. Por ejemplo, con una regla cuyas marcas son $\{0, 1, 6\}$, puede medirse también la distancia 5, ya que $5 = 6 - 1$. Pero una vez que consideramos 5 como una medida que se puede obtener con la regla, también es posible medir 4, ya que $4 = 5 - 1$; este proceso puede continuar tantas veces como se desee. Realiza un procedimiento que complete una RG añadiendo todas las posibles medidas nuevas.

***Bibliografía*** S. W. Golomb fue el primero que estudió estas reglas y sus propiedades; de ahí su nombre. Sobre este tema y otros relacionados, puede consultarse [Dew86].

## 6 6 Solitario búlgaro 333

Consideremos un mazo de $n$ cartas, siendo $n$ un número triangular; esto es, $n = 1 + 2 + 3 + \ldots + k$ para algún $k \in \mathbb{N}$. (Véase el ejercicio 1.5.) Se reparte la totalidad de las cartas en un número arbitrario $m$ de montones, cada uno de ellos con una cantidad arbitraria $c_i$ ($i \in \{1, ..., m\}$) de cartas.

El lote de montones se puede reorganizar así: se toma una carta de cada montón (con lo que desaparecerán los unitarios), y con todas ellas se forma uno nuevo, que se agrega al lote. Por ejemplo, la operación descrita transforma los montones de `1`, `1`, `8` y `5` cartas, en otros de `7`, `4` y `4`, repectivamente:

$$[1\ 1\ 8\ 5] \rightsquigarrow [7\ 4\ 4]$$

El desarrollo del juego consiste en llevar a cabo la reorganización descrita cuantas veces sea necesario hasta que haya un montón de 1 carta, otro de 2 cartas..., otro de $k-1$ cartas y, finalmente, otro de $k$ cartas. Por ejemplo, partiendo de la situación [5 7 3], las reorganizaciones sucesivas evolucionan como sigue:

$$\begin{array}{ll} & [5\ 7\ 3] \\ \rightsquigarrow & [4\ 6\ 2\ 3] \\ \rightsquigarrow & [3\ 5\ 1\ 2\ 4] \\ \rightsquigarrow & [2\ 4\ 1\ 3\ 5] \\ \rightsquigarrow & [1\ 3\ 2\ 4\ 5] \\ \rightsquigarrow & [2\ 1\ 3\ 4\ 5] \\ \rightsquigarrow & [1\ 2\ 3\ 4\ 5] \end{array}$$

Surge la duda de si esas acciones conducen efectivamente a una configuración final y, por consiguiente, el juego termina sea cual fuera la distribución inicial del mazo; simplemente, aceptaremos que así ocurre ya que su demostración rebasa el alcance de este ejercicio. Además, aunque el orden entre los montones es irrelevante en el juego, si se disponen en hilera y el nuevo montón se agrega siempre al final, se llega a una configuración de $1, 2, \ldots, k$ cartas, en orden creciente.

En este ejercicio se propone desarrollar un programa que simule este juego.

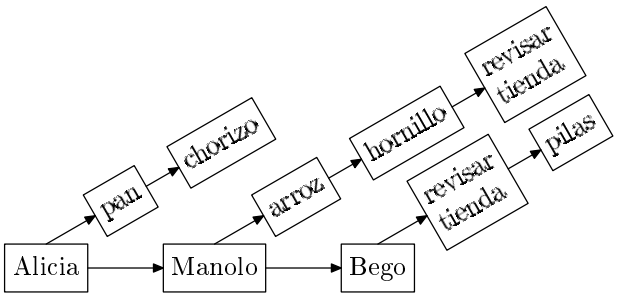
***Bibliografía*** Este enunciado proviene de [Gar83]. Se plantean en él una serie de actividades "ilimitadamente ilimitadas" (parafraseando a John H. Conway) pero que, paradójicamente, concluyen de forma inevitable en un número finito de pasos. Ésta es la idea que ya anticipa el título del artículo mencionado: *Tareas que es forzoso concluir, por mucho que se quiera evitarlo.*

El problema que se enuncia aquí con tanta sencillez está emparentado con los números ordinales transfinitos y otros interesantes conceptos en torno al infinito.

## 6 7 Una listilla con listas

Otoño. Hemos quedado todos para ver qué hacemos el fin de semana. Rápidamente nos hemos puesto de acuerdo en el monte que queremos subir esta vez; es uno de los que nos está esperando desde hace tiempo, el Vignemale. Tenemos que hacer los preparativos repartiendo las tareas ya que tenemos que pasar una noche en la montaña. Alicia aparece con un portátil dispuesta a utilizarlo a toda costa. "He pensado —dice— que ya está bien de hacer a mano listas de compras y tareas. Voy a desarrollar un programa que nos ayude a confeccionar una lista de las cosas que tenemos que hacer y quiénes nos encargaremos de hacerlas." No dejan de oírse comentarios críticos (¡donde esté un lápiz y un papel!, ¿estás aburrida?, ¡pringaos!, ...), pero Alicia lo tiene claro.

***Una gran idea*** Ha pensado crear una lista con los miembros de la pandilla montañera. Cada elemento de esa lista, a su vez, apuntará a la lista de compras o tareas que tiene asignadas. Por lo tanto hablamos de dos tipos de elementos (elemento montañero y elemento tarea o compra) que estarán relacionados. La siguiente figura ilustra *grosso modo* cómo sería dicha estructura de datos:




Escríbele a Alicia los tipos de datos que le permitan representar la estructura de datos que ha ideado.

***Manos a la obra*** Escribe sendos subprogramas que permitan añadir y borrar montañeros de la lista. También será necesario un subprograma que añada una tarea a un montañero y otro que lo libere de una tarea. Ten en cuenta que antes de borrar a un montañero habrá que distribuir sus tareas entre los demás. Elige tú el criterio de distribución.

***A ti te toca ...*** Escribe un subprograma que muestre las tareas que tiene asignadas cada montañero.

***Guardamos la lista*** El sector crítico resalta que, si todas las semanas que haya salida va a haber que reescribir la lista completa, no ven la utilidad del programa de Alicia. Cierto, pero para evitar esta situación Alicia ha previsto almacenar la información de la estructura de datos en un fichero en disco. De esta manera, se podrá reutilizar la información de una lista creada con anterioridad, borrando lo que no convenga y añadiendo la información necesaria. En el fichero habrá dos tipos de líneas: un tipo de línea con información del montañero que, por ejemplo, comience con un carácter "*" y un tipo de línea para la tarea o compra. La estructura de la figura quedaría en el fichero:

```
* Alicia
pan
chorizo
* Manolo
arroz
hornillo
revisar tienda
* Bego
...
```

Escribe un subprograma que, siguiendo la convención descrita, vuelque a un fichero la información de la estructura de datos en la memoria. Asimismo, escribe otro subprograma que reconstruya la estructura de datos en la memoria a partir de la información de un fichero.

***El programa completo*** Ya sólo queda escribir un programa que integre los subprogramas de los apartados anteriores para que Alicia se pueda lucir.

## 6.8 De cómo podar setos

Un seto crece, unas ramas se alargan más que otras y, al final, siempre quedan puntas que repugnan a nuestra decencia; por eso, el jardinero las recorta hasta llegar al nivel que han alcanzado todas las ramas. De igual manera un árbol binario completo puede crecer, unas ramas se alargarán más que otras y al final siempre quedan puntas que impiden su completitud; por eso, escribiremos un subprograma para recortar las ramas desparejas en un árbol hasta que lo volvamos a dejar completo. Hay que tener cuidado de no podar el árbol más de lo necesario para que quede un árbol completo tan grande como sea posible.

## 6.9 Cuadrados mágicos

323 338

Se denomina *cuadrado mágico* a una colocación de $n^2$ números en filas y columnas formando un cuadrado, con $n$ números en cada fila y en cada columna, de tal forma que la suma de los elementos de cada fila, cada columna y las dos diagonales del cuadrado dé el mismo valor. Habitualmente los números utilizados para colocar en un cuadrado de lado $n$ son aquéllos comprendidos entre el 1 y el $n^2$. Los cuadrados

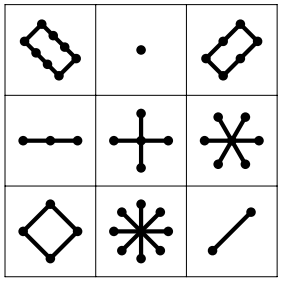


Figura 6.1: Cuadrado mágico de Luo Shu

mágicos han fascinado desde la antigüedad y podemos encontrar ejemplos de ellos en escritos antiguos de culturas tan diversas como la árabe, la china o la india. Uno de los primeros cuadrados mágicos conocidos (siglo X) es el de Luo Shu (figura 6.1) que aparece en una leyenda china grabado sobre la concha de una tortuga. Numerosos artistas también se han maravillado por su abstracta *belleza* y los han incluido en sus obras. Albrecht Dürer (1478–1521) realizó el grabado *Melancholia* en 1514, donde puede encontrarse el siguiente cuadrado:

| | | | |
|---|---|---|---|
| 16 | 3 | 2 | 13 |
| 5 | 10 | 11 | 8 |
| 9 | 6 | 7 | 12 |
| 4 | 15 | 14 | 1 |

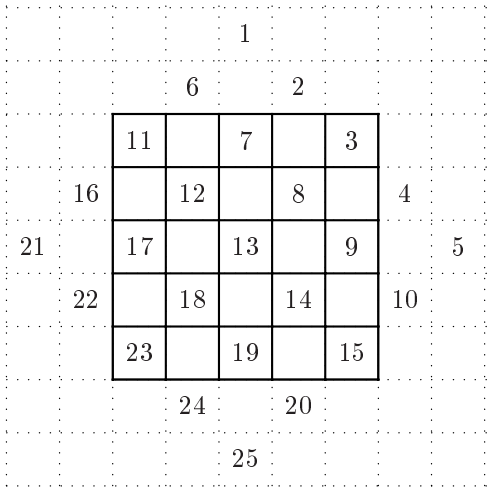
¿Puedes ver la fecha del grabado en el cuadrado? En una de las puertas de la catedral de *La Sagrada Familia* en Barcelona, el arquitecto Antoni Gaudí (1852–1926) incluyó el siguiente,

| | | | |
|---|---|---|---|
| 1 | 14 | 14 | 4 |
| 11 | 7 | 6 | 9 |
| 8 | 10 | 10 | 5 |
| 13 | 2 | 3 | 15 |

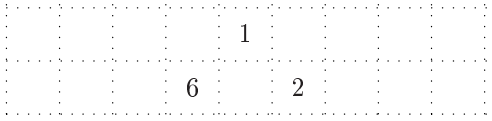
que no es un cuadrado mágico ordinario (ya que no contiene los números entre el 1 y el 16) pero que conserva la propiedad de que sus filas, columnas y diagonales suman lo mismo, 33 en este caso.

***Comprueba cuadrado*** Define un subprograma que permita comprobar que un cuadrado relleno de números es en efecto un cuadrado mágico.

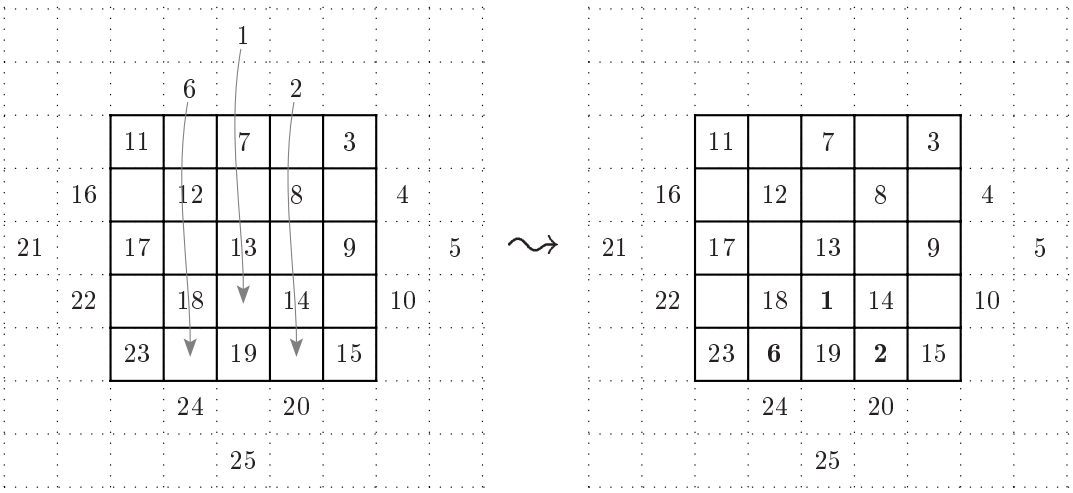
***Cuadrados mágicos de lado impar*** Hay un bonito método para la construcción de cuadrados mágicos de lado impar que se puede explicar con un ejemplo: supongamos que se desea generar un cuadrado mágico cuyo lado tiene un número impar de elementos, por ejemplo, un cuadrado de lado cinco. Primero dibujamos el cuadrado de dimensión $5 \times 5$ dentro de un cuadrado de dimensión mayor. Luego escribimos los números que irán en el cuadrado mágico (en este caso del 1 al 25) en orden creciente, siguiendo las diagonales de izquierda a derecha comenzando por la casilla superior, como indica el dibujo siguiente:




Hemos punteado las casillas que no pertenecen al cuadrado que deseamos construir. Para completar el cuadrado mágico tenemos que poner *dentro* del cuadrado los números que están *fuera* del cuadrado central. Fijémonos en la parte superior:




Si imaginamos que éstos se desplazan por el interior del cuadrado hasta tocar el fondo del mismo, comprobamos cómo pasan a ocupar unos huecos en el cuadrado. Hemos puesto en **negrita** los números que entran en el cuadrado:




Con el resto de los elementos que están fuera realizamos operaciones análogas: se desplazan por su base hasta tocar el lado opuesto del cuadrado al que se encuentra, obteniendo así el cuadrado mágico deseado.

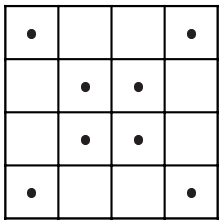
| | | | | |
|---|---|---|---|---|
| 11 | **24** | 7 | **20** | 3 |
| **4** | 12 | **25** | 8 | **16** |
| 17 | **5** | 13 | **21** | 9 |
| **10** | 18 | **1** | 14 | **22** |
| 23 | **6** | 19 | **2** | 15 |

Teniendo el cuenta el método descrito, escribe un subprograma que permita generar cuadrados mágicos de dimensión impar. (Véase la pista 6.9a.)

***Cuadrados con lado múltiplo de cuatro*** Un método propuesto en el siglo XIV por Ibn Qunfudh (1330-1407) para cuadrados cuyo lado es múltiplo de cuatro consta de dos pasos:

- Se *marcan* de forma adecuada algunas de las casillas del cuadrado que se va a construir.
- Se *rellena* el cuadrado con los números de una forma simple.

Describimos primero la operación de relleno con un ejemplo: supongamos que tenemos un cuadrado de dimensión cuatro previamente marcado (luego veremos cómo realizar este paso); el marcaje de dicho cuadrado es el siguiente:



Para rellenar el cuadrado tenemos que recorrer sus casillas en orden lexicográfico (de izquierda a derecha y de arriba hacia abajo) contando de uno en uno. En las casillas marcadas escribiremos el número de orden que corresponda.

En nuestro caso concreto, la primera casilla en orden lexicográfico es la de la esquina superior izquierda. Como dicha casilla está marcada escribimos el 1 sobre ella. La siguiente casilla es la de su derecha, y como no está marcada no escribimos el 2. La tercera casilla tampoco está marcada, pero la cuarta sí por lo que escribimos el 4. En el cuadrado siguiente se puede ver el resultado:

| | | | |
|---|---|---|---|
| 1 | | | 4 |
| | 6 | 7 | |
| | 10 | 11 | |
| 13 | | | 16 |

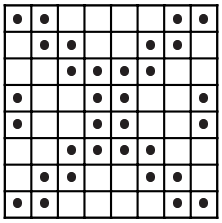
Para acabar de rellenar el cuadrado, que ya tiene una tanda de números, vamos a realizar una operación muy parecida a la anterior. Esta vez recorremos las casillas del cuadrado de derecha a izquierda y de abajo hacia arriba, contando de uno en uno y escribiendo el número de orden en los huecos vacíos.

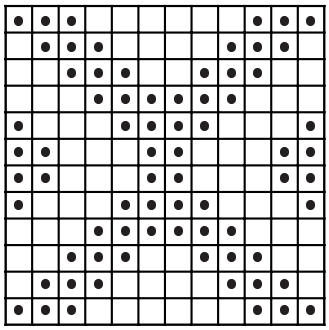
En el caso concreto que estamos considerando, la primera casilla del recorrido de derecha a izquierda y de abajo hacia arriba es la del extremo inferior derecho. Como esta casilla está ya ocupada, no escribimos nada. La siguiente casilla del recorrido es la que está a su izquierda, que está vacía, y por tanto escribimos en ella el número 2. Lo mismo ocurre con la siguiente casilla a su izquierda: como no está ocupada anotamos el 3. La siguiente ya está numerada, y por tanto no tenemos que escribir nada. En el cuadrado siguiente puedes ver el resultado final. Los números escritos en el segundo recorrido están en **negrita** para que se aprecien mejor:

| | | | |
|---|---|---|---|
| 1 | **15** | **14** | 4 |
| **12** | 6 | 7 | **9** |
| **8** | 10 | 11 | **5** |
| 13 | **3** | **2** | 16 |

Como puedes comprobar hemos construido un cuadrado mágico de forma muy sencilla, aunque eso sí, partiendo de un cuadrado ya marcado.

El marcado de los cuadrados cuyo lado es múltiplo de 4 sigue una pauta muy concreta. Ya conoces el marcado del cuadrado de lado 4, son únicamente las diagonales lo que marcamos. Vamos a darte el marcado de los cuadrados de lado 8 y 12 y tendrás que encontrar el patrón general:





Descubre cuál es el patrón general para la definición de marcados para cuadrados de lado múltiplo de cuatro. Siguiendo las indicaciones del método descrito anteriormente (marcado y rellenado) escribe un subprograma que permita generar cualquier cuadrado mágico de dimensión múltiplo de cuatro.

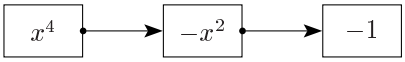
***Bibliografía*** Aparte de los dos métodos que propone el ejercicio existen otras formas para la construcción de cuadrados mágicos. En [Cha99] podrás encontrar, entre otros, el método del marcado. En [dG84] podrás encontrar la construcción que hemos dado para cuadrados de lado impar y muchos otros juegos matemáticos. En Internet, te invitamos a visitar el siguiente enlace: `http://mathforum.org/alejandre/magic.square.html`.

***Un poco de historia*** A partir del siglo XV son muchos los matemáticos europeos que se interesaron por los cuadrados mágicos, y a ellos se les atribuye el descubrimiento de ciertos métodos de construcción. Sin embargo, los matemáticos árabes, ya en el siglo X, conocían métodos de construcción de cuadrados mágicos.

## 6.10 Implementación de polinomios

345

Un polinomio se puede ver como una lista de monomios:




***Tipo de datos*** Define un tipo de datos adecuado para representar monomios y polinomios.

***Suma y resta de polinomios*** Diseña un subprograma que sume dos polinomios y otro que los reste.

***Multiplicación de polinomios*** Diseña un subprograma que multiplique dos polinomios.

***División de polinomios*** Diseña un subprograma que divida dos polinomios.

## 6.11 Máximos y mínimos

352

Se dispone de varias secuencias de números enteros, y se desea hallar el mínimo de los máximos elementos de las secuencias. Por ejemplo, para las secuencias siguientes,

16, 49, 21, 2, 58, 15;
50, 71, 23, 40, 81, 34, 23, 75;
23, 53, 25, 15;
45, 90, 12

los máximos son, respectivamente, 58, 81, 53, 90, y el mínimo de ellos es 53.

Lo que se pide es un procedimiento para ese cálculo, suponiendo que los datos están consignados en una lista de listas.

***Bibliografía*** El enunciado de este ejercicio (tomado de [BH87]) tiene una generalización conocida para trabajar con listas de listas de listas..., invirtiendo la elección: máximo, mínimo, máximo, etc. Esta generalización se llama *minimax* (o *maximín*), y su campo de aplicación son los juegos: un jugador debe escoger el mejor movimiento, teniendo en cuenta que, seguidamente, el otro jugador realizará el movimiento más ventajoso para él y por tanto el más perjudicial para el primer jugador. En la solución de este ejercicio se propone una mejora para truncar la búsqueda que recibe el nombre de *poda alfa-beta*, y puede consultarse en libros especializados en algoritmos, como [AHU98, BB00, Man89, Sed89].

## 6.12 Caminos en la red de metro

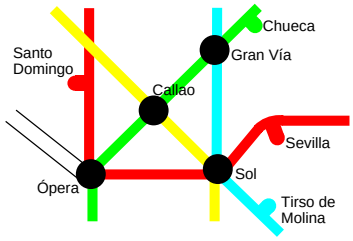
323

Dada la complejidad creciente de la red de metro, nos gustaría tener un programa que nos ayudara a la hora de encontrar un camino para ir de un punto de la ciudad a otro.

***Representación de datos*** Explica cómo podemos representar los datos que intervienen en el problema de forma adecuada para poder manejarlos en un programa. Escribe un programa que permita definir una red de metro.

***Búsqueda de caminos*** Escribe un programa que, dada una red de metro y dos puntos $P_1$ y $P_2$ pertenecientes a dicha red, indique todas las formas posibles de ir de $P_1$ a $P_2$ sin hacer ciclos, es decir, sin pasar más de una vez por una misma estación.

***Ejemplo*** Consideremos el siguiente fragmento de una red de metro:




Si $P_1$ es *Gran Vía* y $P_2$ es *Santo Domingo*, los caminos posibles *sin ciclos* para ir de $P_1$ a $P_2$ en esta red son los siguientes:

*Gran Vía* → *Callao* → *Ópera* → *Santo Domingo*
*Gran Vía* → *Sol* → *Ópera* → *Santo Domingo*
*Gran Vía* → *Callao* → *Sol* → *Ópera* → *Santo Domingo*
*Gran Vía* → *Sol* → *Callao* → *Ópera* → *Santo Domingo*

Desde luego, en esta red podemos pensar en otros caminos que pasan dos veces por la misma estación y por tanto dan lugar a ciclos:

*Gran Vía* → *Callao* → *Ópera* → *Sol* → *Callao* → ...
ciclo

Pero estos caminos deberían evitarse, ya que suponen una vuelta (ineficaz), además de dar lugar a posibles caminos infinitos. (Véase la pista 6.12a.)

El problema que acabamos de proponer es muy general y puede ser aplicado a muchas otras situaciones. Podemos aplicarlo en otras *redes*, como líneas de autobuses, trenes, carreteras o incluso calles. También la escala del problema es perfectamente adaptable: igual podemos encontrar los caminos posibles para ir de nuestra casa a la panadería (sin salirnos del barrio) que para ir en avión desde Madrid a Sri-Lanka.

***Bibliografía*** Las *redes* (de metro, de carreteras...) de las que habla este ejercicio son representables mediante *grafos*. En muchos libros de programación se pueden estudiar en detalle técnicas generales para explorar, recorrer y obtener información de grafos. Mira por ejemplo [Man89, BB96, BB00].

## 6.13 El planificador que no planifique un mal planificador será

En este ejercicio te proponemos implementar una de las diversas tareas que lleva a cabo un sistema operativo (SO). Un poco de teoría te ayudará a entender de qué tarea se trata. Ya sabes que un SO multitarea aprovecha los tiempos muertos de la Unidad Central de Proceso (CPU), los tiempos muertos de los periféricos y el espacio de memoria principal no ocupado. En esencia, consiste en cargar en la memoria principal varios programas e ir asignando a cada proceso asociado a esos programas la CPU. De esta manera, se pueden aprovechar los tiempos muertos de CPU que se producen al ejecutar un proceso, ejecutando otros. Varios procesos pueden ir avanzando en su ejecución sin necesidad de que finalice completamente uno para poder comenzar la ejecución del siguiente. Se dice que un proceso entra en estado de *bloqueo* cuando la CPU no puede continuar trabajando con él porque está esperando la realización de una operación de entrada o salida. Se dice que un proceso está *preparado o ejecutable* cuando la CPU puede iniciar o continuar su ejecución. Un proceso está *activo* si la CPU lo está ejecutando.

Si en un determinado momento existen varios procesos preparados y la CPU está inactiva, el SO debe elegir uno para darle el turno de ejecución. El módulo del SO que se encarga de solventar este problema se denomina planificador (*scheduler* en inglés) y lo hace aplicando un algoritmo de planificación. Pues bien, es precisamente un planificador lo que tienes que implementar; en concreto, deberás hacer un programa que simule su funcionamiento.

***Tipos de datos*** Declara una estructura de datos para almacenar la información asociada a un proceso, básicamente: identificador del proceso y estado (preparado, bloqueado o activo). Además, podrás añadir todos aquellos campos que consideres necesarios.

***Planificación de petición circular*** Escribe un subprograma que simule la asignación de tiempo de CPU de un planificador que utilice el algoritmo de planificación de petición circular. Con este algoritmo, a cada uno de los procesos en la memoria se le asigna un intervalo de tiempo de duración fija. Se cambia el contexto de un proceso a otro, de forma rotatoria, conforme se van consumiendo esos intervalos.

Piensa en la estructura de datos más adecuada para representar los procesos que están en la memoria y a los que se les asignará tiempo de CPU según el algoritmo de planificación circular. Además, deberás simular la incorporación de un proceso a dicha estructura y una forma de determinar cuándo ha acabado la ejecución de un programa.

***Tiempo compartido*** En los sistemas operativos de tiempo compartido, se cambia la asignación de tiempo de CPU de un proceso a otro no sólo cuando al proceso activo se le acaba el tiempo, sino también en el instante en que quede bloqueado porque debe realizar una operación de entrada o salida.

Escribe un subprograma que contemple este nuevo caso. Ten en consideración que deberás simular la posible aparición de una operación de entrada o salida en el proceso activo.

***Planificación de asignación de prioridades*** Escribe un subprograma que simule la asignación de tiempo de CPU de un sistema operativo que utilice el algoritmo de planificación de asignación de prioridades. En este caso, se da el turno de ejecución al proceso preparado con mayor prioridad. Existen varios criterios de asignación de prioridades. Siguiendo uno de ellos, para que el proceso de mayor prioridad (por tanto activo) no monopolice el uso de la CPU, cada cierto tiempo (cada 40 ms por ejemplo) se le baja su prioridad. Así, se conmuta a otro proceso cuando alguno de la cola de espera supere la prioridad del proceso activo.

Piensa en la conveniencia de modificar la estructura de datos que definiste en el primer apartado añadiendo algún campo.

## 6.14 Simulación de una cola múltiple

Se desea implantar un gran supermercado en el barrio más populoso de una ciudad. Como es natural, se desea ofrecer un servicio de calidad al menor coste posible; y, para ello, se debe hacer una previsión de afluencia de público con la información que se tiene. Concretamente, en este estudio se desea averiguar el número ideal de cajas de cobro que se van a necesitar, de forma que los clientes nunca esperen un tiempo excesivo, pero también de forma que no se contrate más personal de caja que el necesario.

Como pasa a menudo, el estudio puede hacerse tan preciso (y complejo) como se desee. Nosotros vamos a plantear distintas situaciones que recogen una parte de la realidad de más simple a más compleja. Por ejemplo, el comportamiento de los clientes se recoge en distintos aspectos:

- Las llegadas de los clientes.
- El tiempo que tardan en llenar el carrito de la compra.
- El tiempo que tardan en ser atendidos en la caja.

Esta información vendrá dada por sendas variables aleatorias que se pueden modelar con mayor o menor grado de realismo, como veremos después. El número de cajas, la longitud de sus colas individuales y su flexibilidad al abrir y cerrar son aspectos que también se pueden detallar en mayor o menor medida. En los siguientes párrafos veremos distintas descripciones de nuestro estudio, de más sencillo a más complejo.

***Primera versión: simulación de una cola simple*** En esta primera versión, se considera que los clientes llegan al supermercado, a intervalos de tiempo aleatorios, según una función de distribución exponencial (véase el ejercicio 3.13) de media $\lambda_1$ (en segundos); por el momento, este parámetro es fijo, independiente de la hora del día de que se trate. Imaginamos que su comportamiento consiste en ir directamente a la (única) caja, ponerse en la cola y pagar, empleando en pagar un tiempo medio $\lambda_2$. También imaginamos que los clientes se consideran bien atendidos si no han de esperar en la cola más allá de un cierto tiempo, al que vamos a llamar *paciencia*.

En este apartado, se trata de efectuar una simulación, con los datos $\lambda_1, \lambda_2$, y *paciencia*. La simulación deberá representar una jornada completa. El programa debe hallar la siguiente información:

- El número total de clientes que han sido atendidos en una jornada.
- El tiempo medio de espera de los clientes en la cola.
- El número de clientes que han tenido que esperar más de un cierto tiempo crítico (*paciencia*).
- La longitud máxima que ha alcanzado la cola.

***Segunda versión: simulación de una cola múltiple*** La siguiente versión de nuestro supermercado no consta de una caja única, sino de $k$ cajas abiertas, numeradas de uno en adelante, con un máximo de $N$, que son las que caben en el local. Cada caja puede tener una cola de longitud arbitraria.

Los clientes se comportan como antes, por lo que se mantienen los datos $\lambda_1, \lambda_2$, y *paciencia*. Pero ahora eligen la caja aleatoriamente, entre las $k$ abiertas, sin preferencias por ninguna de ellas. Debe observarse que, en este caso, hay que manejar $k+1$ posibles eventos:

| | | |
|---|---|---|
| Evento 0 | : | La llegada de un cliente nuevo al supermercado. |
| Evento 1 | : | La salida de la caja 1. |
| ... | | |
| Evento $k$ | : | La salida de la caja $k$. |

Lo que se pide es efectuar distintas simulaciones para distintos números de cajas abiertas (entre 1 y $N$). En cada simulación, interesa recoger la misma información que antes. Esta información se deberá registrar en un archivo, para ser usada posteriormente por otros programas.

***Una mejora: decisión sobre el tamaño de la hilera de cajas*** Teniendo en cuenta la información recogida en el apartado anterior, se desea que sea un programa el que decida el menor tamaño de la hilera $k$, tal que no se haya agotado la paciencia de más de un 1% de los clientes.

Seguidamente, el comportamiento del supermercado con ese $k$, deberá ratificarse con varias (p.e., 100) simulaciones, aceptándose ese $k$ como decisión final sólo si más del 95% de ellas han resultado exitosas (es decir, no se ha agotado la paciencia de más de un 1% de los clientes).

***Otra mejora: clientes con distinta cantidad de compra*** La realidad es que los clientes no van directamente a las cajas, sino que se entretienen más o menos, llenando su carrito: a efectos de simulación, a cada cliente que llega se le asigna un tiempo de compra. El resultado es una estructura de clientes en estado de *paseo* por las galerías del supermercado. En esta estructura ingresan todos los clientes. Lógicamente, no van a ser atendidos en el orden en que llegaron, sino en el que terminan sus compras.

Así pues, los posibles eventos son ahora los siguientes:

**Llegada de un cliente al supermercado** Este evento se rige por una variable aleatoria exponencial de parámetro $\lambda_1$.

Además, al llegar, cada cliente decide el número de artículos que va a comprar (por ejemplo, uniformemente, entre 1 y 50). Esto supone un tiempo de estancia en los pasillos más o menos prolongado:

$$5 \text{ minutos} + 0.5 \text{ minutos} \times \text{núm. de artículos}$$

En realidad, los tiempos habría que simularlos con variables aleatorias.

Cuando llega un cliente se introduce en los pasillos del supermercado hasta que le toca ir a una caja a pagar.

**Fin de compra de un cliente para colocarse en la cola de una caja** Cuando un cliente termina de llenar el carrito, sale de los pasillos y se dirige a la caja que elija a pagar.

**Salida de un cliente de una caja** El tiempo de atención de cada cliente en la caja es aleatorio y depende del número de artículos que ha comprado:

$$1.5 \text{ minutos} + 0.1 \text{ minutos} \times \text{núm. de artículos}$$

***Eventos de funciones de distribución arbitrarias*** Otro aspecto interesante es que los clientes no se distribuyen por igual a lo largo del día, sino que hay horas *flojas* y horas *punta*, con distintas características de afluencia.

Para reflejar esta realidad, es sencillo modificar nuestros programas para que operen con funciones de distribución exponenciales, pero dependientes de parámetros $\lambda_1(t)$ variables, según la hora $t$ en que se hallen las variables aleatorias correspondientes. Más aún, las funciones de distribución podrían ser arbitrarias (no necesariamente exponenciales), dependientes del tiempo.

Lo que se propone aquí es permitir que nuestro estudio opere con las funciones de distribución planteadas:

- Exponenciales, de parámetro fijo $\lambda$ (ya sea fijo o no).
- Arbitrarias. En este caso, se asumirá que las funciones de distribución son de variables aleatorias del intervalo $[0, \infty)$, y continuas (además de crecientes entre 0 y 1, como es lógico).

Y como aplicación, adapta nuestro estudio para simular la situación concreta que sigue:

- El horario es de 10 a 20 horas.
- Los tiempos medios de llegada entre clientes (en segundos) son los siguientes, según la hora del día ($t$, en horas):

$$\text{tiempo medio a las } t \text{ horas } = \begin{cases} 20-(t-10)\cdot 5 & \text{segundos} & \text{si } 10 \le t \le 13 \\ 5 & \text{segundos} & \text{si } 13 \le t \le 14 \\ 5+(t-14)\cdot 5 & \text{segundos} & \text{si } 14 \le t \le 20 \end{cases}$$

***Llegadas selectivas a las cajas de cobro*** La descripción anterior es bastante aceptable. Pero la realidad es que los clientes no suelen escoger una cola aleatoriamente, sino que procuran situarse en la más vacía. En este apartado se propone esta variante: que los clientes se ponen en la cola que menos personas tenga.

***Hilera de cajas dinámica*** Otra variante interesante consiste en efectuar la simulación manteniendo abiertas sólo las cajas que resulten necesarias. Esto lo podemos precisar así: cuando un cliente llega a ser atendido en una caja, se comprueba su paciencia y, en caso de haberse agotado, se abre una caja nueva. Recíprocamente, cuando queda una caja vacía, se cierra.

De este modo, el programa podrá ofrecer como salida un informe con la evolución del número de cajas que han estado abiertas en los diferentes momentos de la jornada. Esta información será crucial para permitir a los ejecutivos de la empresa decidir sobre la contratación por turnos.

***Control regular del supermercado*** Por otro lado, puede interesar efectuar observaciones del supermercado a intervalos de tiempo fijos, establecidos independientemente de las llegadas o salidas de clientes. Dichas observaciones pueden consistir en tomar medidas o en simples *fotografías* del supermercado. Esto requiere introducir un nuevo evento, con su cronómetro propio, que controle lo necesario para efectuar las observaciones.

Se pide que, a intervalos de tiempo fijos (por ejemplo, cada 5 minutos), el programa ofrezca una representación del supermercado con los datos más relevantes,

- Número de clientes atendidos hasta el momento.
- Frecuencia de entrada de los clientes.
- Ocupación de las galerías.
- Número de colas y sus tamaños.
- Etcétera.

indicando el comportamiento de las cajas a distintas horas del día.

Una aplicación interesante es ayudar en la toma de decisiones sobre contratación de cajeros o cajeras, ya que dicha contratación se puede hacer por turnos dependiendo de las horas.

***Bibliografía*** La simulación de colas está ampliamente tratada en muchos libros de investigación operativa. En [POAR97] (apartado A.2.3) puede encontrarse una introducción sencilla, siguiendo el método del acontecimiento crítico para la simulación de colas sencillas.

## 6.15 Un diccionario electrónico bilingüe

Se desea desarrollar un programa que gestione un diccionario bilingüe, digamos de inglés-español y viceversa para fijar ideas. En los siguientes apartados se plantean diferentes posibilidades, de más sencilla a más complicada, concentrando la atención en varios aspectos de interés.

***Operaciones básicas*** En términos generales, las estructuras de datos necesarias pueden organizarse como dos tablas independientes, cada una de ellas indicada para traducir en un sentido. Las operaciones serán pues las típicas de las tablas: de creación (vacía), de inserción de un nuevo elemento, de supresión (aunque ésta no se va a usar en realidad) y de consulta, que es el uso más típico de un diccionario.

Además, se han de incluir operaciones de almacenamiento del diccionario en un dispositivo permanente, y recuperación del mismo.

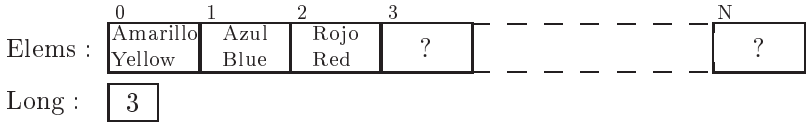
Suponiendo que tenemos definida una estructura de datos adecuada para manejar un diccionario, describe el perfil (la cabecera) que podrían tener las operaciones básicas descritas.

***Aplicaciones del usuario*** Las primeras operaciones que ha de permitir nuestro diccionario son la consulta de un término en español para conocer su traducción al inglés, y viceversa. Además, se desea desarrollar otra aplicación que, a partir de un texto en español almacenado en un archivo de disco (`*.esp`), genere otro (`*.ing`), resultante de traducirlo al inglés término a término, ignorando la sintaxis y cualesquiera otras consideraciones lingüísticas. También se debe generar la operación inversa, traductora del inglés al español. Desarrolla los programas descritos.

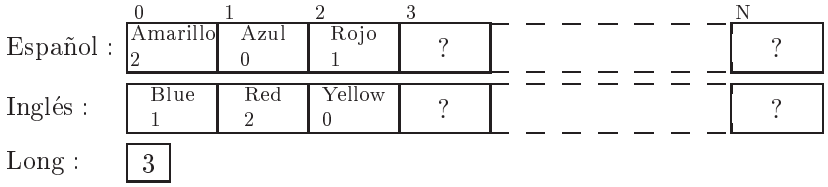
***Implementaciones sencillas*** Una primera implementación sencilla de una tabla es mediante una lista ordenada de pares $\langle clave, valor \rangle$, donde lo usual para nosotros es que las claves sean términos en español y los valores términos en inglés:

| Claves | Valores |
|---|---|
| Amarillo | Yellow |
| Azul | Blue |
| Rojo | Red |

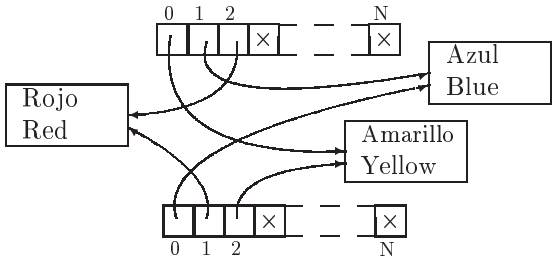
Sobre las claves, se tienen definidas sendas relaciones de orden (el lexicográfico) y de equivalencia para facilitar la búsqueda. A su vez, la lista ordenada se puede implementar mediante un array junto con su nivel de ocupación, como se ve en la siguiente figura:



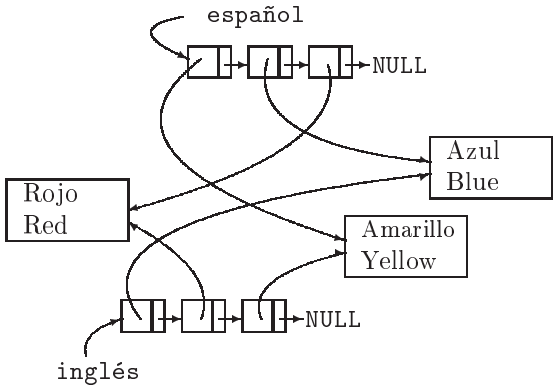

El problema con este diccionario es que, para traducir en *ambos* sentidos eficientemente, haría falta tener los elementos ordenados según *dos* relaciones de orden distintas, lo que no es posible. Una solución es mantener una lista para cada idioma, añadiendo a cada término la posición de su homólogo en el otro idioma:




Por otra parte, el gasto de memoria (superfluo) derivado de la parte vacía del diccionario se puede evitar sustituyendo los términos por punteros:



o, mejor aún, usando listas enlazadas en vez de arrays:

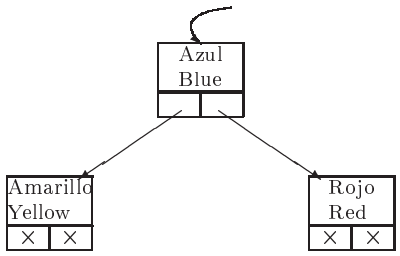



Se plantea ahora desarrollar una de las implementaciones descritas.

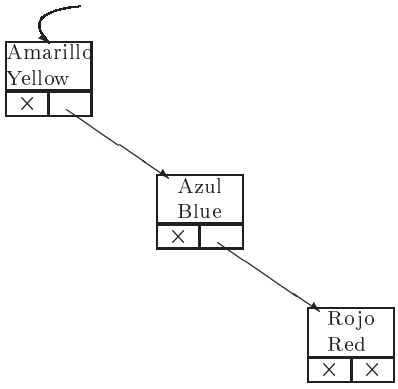
Si se sabe que se va a manejar siempre un diccionario pequeño, ¿cuál de las implementaciones mencionadas recomendarías? ¿Por qué? ¿Y en un diccionario *cerrado* (o sea, que no va a requerir incluir ningún término nuevo)?

***Implementaciones eficientes*** Los apartados anteriores deberían llevarnos a la conclusión de que cada una de las implementaciones anteriores de las tablas posee alguna operación con una eficiencia pobre, por lo que hace tiempo que han surgido otras estructuras de datos más convenientes.

Una de las implementaciones más corrientes de las tablas es a base de árboles binarios de búsqueda:



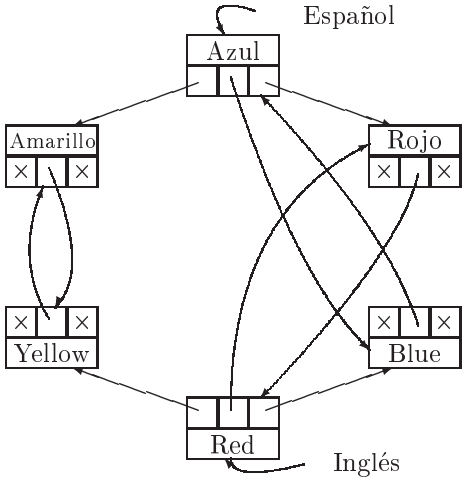

El inconveniente es que, dependiendo del orden en que se hayan efectuado las inserciones y supresiones de elementos, pueden aparecer configuraciones poco equilibradas,



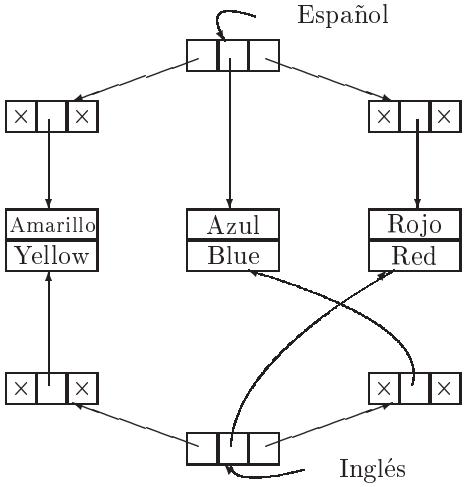
con un comportamiento ineficiente. Por eso surgen los árboles equilibrados.

Estas implementaciones de las tablas son ampliamente conocidas y pueden encontrarse en casi cualquier libro sobre estructuras de datos.

El problema, ya mencionado, de la duplicación de los órdenes de las claves se resuelve nuevamente usando dos tablas (árboles) en vez de una, ya sea cruzando las referencias




o compartiendo los pares de términos:




Desarrolla un módulo para el diccionario basado en una de las tres implementaciones descritas.

## 6.16 El sinónimo absurdo

En un diccionario de sinónimos y antónimos hay, asociada a cada palabra, además de su definición, si acaso es oportuno, una lista de sinónimos y otra de antónimos. Para el propósito de este enunciado, obviaremos la propia definición de la palabra y nos concentraremos en estas dos listas.

***Estructura de datos*** Define una estructura de datos adecuada para almacenar un diccionario de sinónimos y antónimos. Implementa las operaciones para añadir una nueva palabra, un nuevo sinónimo o un nuevo antónimo.

***Definición de clausura*** A *las sinónimas de mis sinónimas son mis sinónimas* lo llamaremos *clausura de sinónimas*. Formalmente, la clausura de sinónimas de una palabra $p$ es el menor conjunto que contiene a $p$ y a las clausuras de sus sinónimas. Es decir, si denotamos con $\bar{p}$ a la clausura de $p$ y $p_0, \ldots, p_n$, que son las palabras que aparecen en la lista de sinónimas de $p$, se define así:

$$\bar{p} = \{p\} \cup \bar{p}_0 \cup \cdots \cup \bar{p}_n.$$

Haz una función que calcule la clausura de sinónimas de una palabra en un diccionario.

***Definición de absurdo*** Un *absurdo* en un diccionario de sinónimos y antónimos es una secuencia de palabras $p_0, \ldots, p_n$ de forma que $p_i$ es sinónima de $p_{i+1}$ para $i = 0, \ldots, n-1$, y además $p_0$ y $p_n$ son antónimas. Escribe una función que encuentre todos los absurdos en un diccionario.

***Definición de incoherencia*** Una incoherencia es un par de términos $p$ y $q$ de forma que $q$ es una sinónima (alternativamente, antónima) de $p$ pero $p$ no es una sinónima (antónima) de $q$. Haz una función que calcule todas las incoherencias de un diccionario.

***En busca de una estructura para encontrar absurdos*** Tal vez, la estructura que concebiste originalmente para este problema es muy adecuada para añadir nuevas palabras, o nuevos sinónimos o antónimos, a un diccionario. Si es así, puede que no sea una buena estructura de datos para buscar absurdos. Investiga, en ese caso, otras estructuras más eficientes para nuestro problema.

## 6.17 La foto de un árbol

323

Encuadramos, disparamos y todo recomienza cuando volvemos de la tienda, a los pocos días, y en ese primer recorrido por nuestros recuerdos encontramos que la foto salió corrida o, peor aún, deformada, irreconocible, compuesta de manchas de colores y, por tanto, inútil a nuestra vaga memoria. Hace unos días o unos años, el tiempo es así, le hicimos una foto a este libro y el resultado lo puede usted ver en la portada. Sin duda salió mal, pensará, pero le vamos a explicar que no fue así. Primero, porque, si lo intenta, comprobará que no saca una foto mejor; que, por mucho que ajuste y perfeccione, su foto de la portada no superará a la portada que ya ve. Segundo, porque la cámara que nosotros usamos es un tanto especial, sólo sirve para fotografiar árboles y fíjese en el resultado que saca. No busque, no la encontrará en las tiendas, pero aquí le explicaremos cómo construirse una.

Nuestra cámara no servirá para fotografiar árboles físicos sino conceptuales. Se dice que abundan ya más que los primeros; será culpa del avance del mal. Toda organización jerárquica es un árbol conceptual: el ejército, la iglesia, la policía, casi todas las empresas, y en general siempre que hay alguien que manda sobre otros, que a su vez mandan sobre otros, etc. Si dibujamos puntos para las personas y trazos que unen su dependencia, tendremos un árbol. En los discos de las computadoras modernas también hay árboles conceptuales; son los sistemas de ficheros, formados por carpetas y subcarpetas (antes directorios y subdirectorios) junto con los ficheros que contienen. Si dibujamos puntos para las carpetas y archivos, y trazamos la pertenencia inmediata, tendremos un árbol. Y así podríamos seguir: la bolsa, una red de computadoras, la división del mundo en tratados, países, autonomías, etc. A partir de ahora, nos centraremos en los sistemas de ficheros.

Nuestra cámara dibuja un rectángulo por cada hoja del árbol; en el caso del sistema de ficheros, un rectángulo por cada fichero. Para decidir el área (que no proporciones) y el color se fija en dos atributos; usualmente, el área viene determinada por el tamaño mientras que el color lo decide el tipo del fichero (si es un programa ejecutable, código fuente, una imagen, un documento, etc.) El atributo del área hay que acumularlo propagándolo por el árbol hasta la raíz; por tanto, el área de una carpeta será la suma de las áreas de todas las carpetas y ficheros que contenga. En la figura 6.2 puede verse un trozo de un sistema de ficheros; los tamaños están entre paréntesis, los que se calculan acumulando, en *cursiva*.

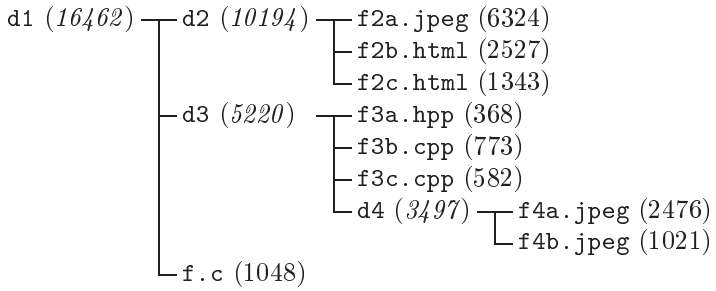



Figura 6.2: Árbol con una parte de un sistema de ficheros

### 6.17 1 Parte y vuelca

Vamos a construir el modelo de cámara conocido como *parte y vuelca*. Produce la imagen con un mecanismo de dos pasos: primero *parte* un rectángulo a lo largo de uno de sus lados, y luego *vuelca* cada uno de los subrectángulos y les aplica el mismo proceso recursivamente. De esta forma, los niveles del árbol se irán construyendo por subdivisión horizontal o vertical, de forma alternada, y la estructura de agrupamiento se hará visible. Vamos a concretar los detalles.

La zona de dibujo es una región rectangular, con los lados paralelos a los ejes de coordenadas. Como cuenta el ejercicio 1.6, este tipo de regiones puede delimitarse con un par de vértices diametralmente opuestos. Pero para este problema, es más cómodo definirlas a partir del vértice inferior izquierdo, $(v_0, v_1)$, la anchura, $\ell_0$, y la altura, $\ell_1$. Tenemos que fotografiar en esta región un árbol que empieza en un nodo $N$ del que salen $k$ ramas que llamaremos $N_0, \ldots, N_{k-1}$. Los nodos $N_i$ tienen tamaño $t_i$ para cada $i$ entre 0 y $k-1$, respectivamente; el nodo $N$ tiene tamaño $t$ que necesariamente ha de ser igual a $\sum_{i=0}^{k-1} t_i$.

El proceso de partición se realiza a lo largo de uno de los ejes; más que partir, es cortar en lonchas nuestro rectángulo a lo largo de ese eje, porque hay que sacar tantos trozos como ramas. El grosor de una loncha viene determinado por la importancia relativa de su rama correspondiente. Supongamos que tenemos que cortar por el eje X. La base del rectángulo es el segmento que empieza en $(v_0, v_1)$ y se extiende hasta $(v_0 + \ell_0, v_1)$. El grosor de la loncha $i$-ésima deber ser proporcional a $t_i/t$; por tanto, su anchura real es $\ell_0 t_i/t$, cantidad que llamaremos $\ell^i$. Estas lonchas realmente generan $k$ puntos de corte en la base; sus coordenadas serán $(v^i, v_1)$, donde $v^i$ viene definida por $v_0 + \sum_{j=0}^{i-1} \ell^j$. Si ahora cortamos hacia arriba empezando en los puntos $(v^i, v_1)$, obtendremos $k$ rectángulos, todos con la misma altura, $\ell_1$, y cada uno con la anchura adecuada a su tamaño, $\ell^i$.

La figura 6.3a resume este proceso. Se parte de una región cuadrada en la que se quiere plasmar el árbol de la figura 6.2. El tamaño de la raíz es 16462, conseguido por acumulación a partir de las tres ramas, de tamaños 10194, 5220 y 1048, que parten de ella. Hay que cortar el cuadrado en lonchas por su base, es decir, horizontalmente (las lonchas se van sacando horizontalmente, aunque los cortes se realizan en sentido vertical). Las anchuras relativas de estas lonchas son 10194/16462, 5220/16462 y 1048/16462;

si suponemos que el cuadrado tiene anchura 1 ($\ell_0 = 1$), estos valores son también las anchuras absolutas. Los puntos de corte caen en las posiciones 0, $10194/16462 \approx 0.6192$ y $(10194 + 5220)/16462 \approx 0.9363$, lo que, a ojo, confirma a la figura 6.3a como un buen inicio de nuestra fotografía.

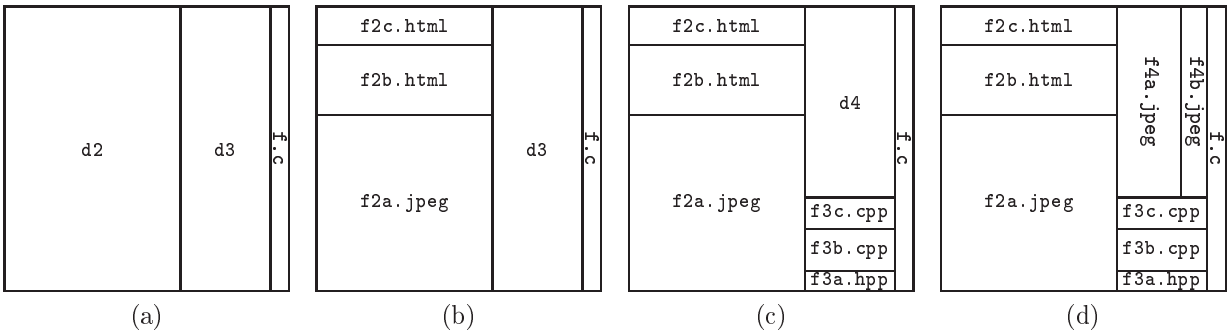



Figura 6.3: Proceso de construcción de una fotografía

Una vez que tenemos un rectángulo para cada una de las ramas, el proceso sigue recursivamente: cada rama se dibuja en su rectángulo. Pero antes hay que *volcar* cada uno de los rectángulos, para que la subdivisión ocurra en el sentido vertical. En las figuras 6.3b y 6.3c puede verse cómo las subdivisiones de las regiones de `d2` y `d3` tienen lugar a lo largo del eje vertical. Todos los pasos recursivos vuelcan los rectángulos, de forma que los recorridos horizontales y verticales se alternan; por eso, el recorrido del rectángulo de `d4` se resuelve horizontalmente como puede comprobarse en la figura 6.3d.

La idea intuitiva de intercambiar el papel de los ejes para *volcar* se plasma formalmente de una manera muy sencilla: las ecuaciones para calcular los puntos de corte sólo difieren en que hay que intercambiar 0 por 1 en los subíndices. Éste es el motivo por el que hemos usado una notación tan atípica para los vértices y hemos llamado $\ell_0$ y $\ell_1$ (de *lado*) a la anchura y la altura. Cuando el punto crítico del algoritmo es poder intercambiar fácilmente los ejes, no tiene mucho sentido usar una notación que remarque innecesariamente sus diferencias.

Todo lo relativo al atributo del tamaño ya está resuelto. El otro atributo, el del color, suele ser bastante más sencillo de tratar porque basta una tabla que asigne colores en función del tipo de fichero. En la fotografía de la figura 6.4 hemos usado el color para los documentos, para las imágenes, y para los códigos fuente.

### 6.17.2 Manual de instrucciones

"Ahora ya tiene la cámara modelo corta-y-vuelca en sus manos. Úsela. Saque fotografías de sus árboles favoritos. Si no sabe por donde empezar, le sugerimos el sistema de ficheros de su computadora; su tamaño y lo variado de su contenido le asegurará algunas sorpresas. Nosotros es lo primero que hicimos; apuntamos a la carpeta donde empieza este libro, disparamos y salió su portada. (Véanse las pistas 6.17a y 6.17b.)

### 6.17.3 Bibliografía

Las fotografías que aquí hemos sacado se conocen con el nombre de *treemaps*, mapas de árboles. Los inventaron Johnson y Schneiderman [JS91, Shn92] para resolver el problema de mostrar una jerarquía enorme de forma que sea comprensible su estructura y la importancia relativa de sus elementos. El algoritmo del apartado 6.17.1 se debe a ellos; posteriormente se han inventado otros algoritmos que intentan subsanar algunas de sus deficiencias. Por ejemplo, como las proporciones de los rectángulos

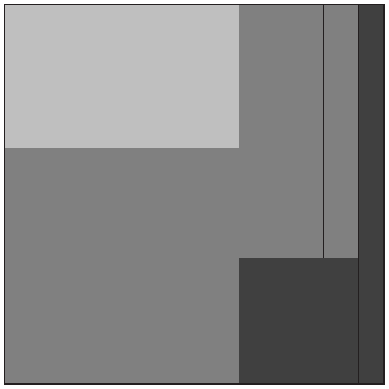


Figura 6.4: La fotografía con colores

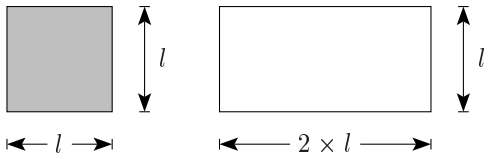
varían mucho, resulta muy difícil apreciar las áreas relativas; en [BHvW00] se describe un algoritmo que intenta construir rectángulos lo *más cuadrados posible*. También ocurre que, cuando las jerarquías son muy profundas, resulta difícil percibir el orden de inclusión; en [vWvdW99] se explica como colorear los rectángulos para producir una sensación de volumen que ayuda a comprender la estructura.

Dos muestras de las aplicaciones actuales de estos algoritmos: una, el programa SequoiaView que muestra el treemap de un sistema de ficheros (`http://www.win.tue.nl/sequoiaview/`); otra, *Map of the Market*, una visión de la evolución de la bolsa, que utiliza el área del rectángulo para la importancia del valor bursátil y el color para la tendencia actual (`http://www.smartmoney.com/marketmap/`).

## 6.18 Baldosas de jardín

324 355

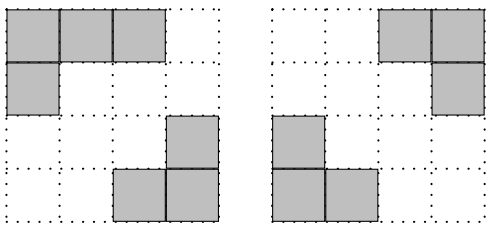
Los padres de Carmencita quieren embaldosar el jardín de su casa y para ello han comprado dos tipos de baldosas: las negras, que son cuadradas, y las blancas, que son el doble de largas que las baldosas negras:




La idea es utilizar las baldosas negras para hacer diseños y las baldosas blancas para completar los huecos. Después de algunos bocetos para dar con un diseño que les gustase, los padres de Carmencita se dan cuenta de que el trabajo no es tan sencillo como parecía: ¡algunos diseños hechos con las baldosas negras dejan huecos que no se pueden cubrir con baldosas blancas!

Viendo que sus padres estaban muy preocupados, Carmencita se puso a pensar en el problema y les prometió hacer un programa en C++ que los pudiese ayudar.

***Ejemplo*** En los siguientes diseños aparecen problemas que Carmencita tendrá que considerar.



En el diseño de la izquierda es fácil observar que el hueco dejado para las baldosas blancas tiene un número de casillas insuficientes. Si consideramos el tamaño de la baldosa negra como la unidad de casilla, tiene que haber un número par de cuadros en un hueco, ya que las baldosas blancas tienen el doble de área que las negras.

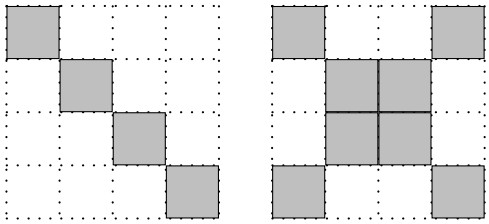
Pero aquí no acaban los problemas, en el diseño de la derecha vemos una situación más complicada: el hueco dejado tiene un número par de casillas, pero, por más que se intente, no se puede rellenar con baldosas blancas.

***Representación de los datos*** Piensa detenidamente en las estructuras de datos que necesitas para poder resolver el problema. Hay que ser capaz de realizar diseños con baldosas negras y comprobar si los huecos dejados por dichos diseños son o no rellenables con baldosas blancas.

***Comprobación para los huecos*** Busca una condición suficiente (no necesaria) que asegure que un diseño no es rellenable con baldosas blancas. Escribe un subprograma que compruebe esta condición. (Véase la pista 6.18a.)

***Búsqueda de huecos*** La comprobación que propone el apartado anterior tiene que realizarse en cada uno de los huecos que dejen las baldosas negras. Por tanto, necesitamos detectar la presencia de los diversos huecos dejados por baldosas negras en un diseño. Realiza un subprograma que se encargue de esta tarea.

***Ejemplo*** En los siguientes diseños aparecen huecos inconexos. En el de la izquierda hay dos huecos inconexos, y ninguno de ellos se puede cubrir con baldosas blancas. En el de la derecha hay cuatro huecos inconexos que sí pueden cubrirse con baldosas blancas.

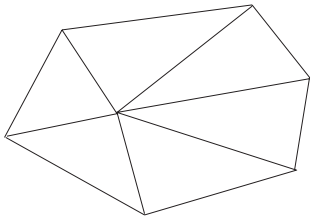


Para este apartado consulta la pista 6.18b.

***Resolución*** Escribe el programa en C++ que Carmencita prometió a sus padres para hacer diseños y dar soluciones, si es posible.

**6.1a.** La clave es rellenar la matriz en el orden adecuado. El cuadrante superior izquierdo contiene toda la información; para construir el resto de la matriz basta replicarlo sumándole una constante. Por eso, lo propio es atacar primero la construcción de este cuadrante.

**6.2a.** Dado un polígono y un punto en su interior, se pueden considerar todos los triángulos formados por cada lado del polígono con ese punto. El área del polígono es igual a la suma de las áreas de todos los triángulos juntos:



Pero, ¿es necesario que el punto sea interior al polígono?

**6.9a.** Para generar un cuadrado de dimensión $n$ no es necesario *utilizar* una matriz de dimensión mayor. Si se piensa un poco, podemos colocar *directamente* aquellos números que en principio están fuera del cuadrado mágico.

**6.12a.** La *detección* de ciclos es fundamental para garantizar la corrección y la finalización del algoritmo. Por suerte, es tarea fácil: tendremos un ciclo si pasamos dos veces por la misma estación.

**6.17a.** Es conveniente subdividir este programa en los siguientes pasos:

⟨*Obtención de un árbol, con pesos y tipos en las hojas*⟩
⟨*Acumulación de los pesos en los nodos intermedios del árbol*⟩
⟨*Generación de la fotografía*⟩

Si no sabes resolver el primer punto, consulta la pista 6.17b; si no te es suficiente, tendrás que recurrir a los manuales de tu sistema de programación.

Para la ⟨*Generación de la fotografía*⟩ te será de utilidad un buen tipo para definir los rectángulos; déjate llevar por la notación matemática del enunciado y por esta pista, y utiliza

```
typedef struct Rectangulo {
  float vertice[2];
  float lado[2];
} Rectangulo;
```

De esta forma, podrás guardar el lado por el que hay que cortar en un parámetro de tipo `int` llamado, por ejemplo, `direccion`. Volcar, ya sabemos, es elegir el otro lado, que se calcula fácilmente con $1 -$ `direccion`.

**6.17b.** Todos los sistemas operativos tienen una forma de recorrer sus sistemas de ficheros. Vamos a revisar cómo se hace en muchas de las variantes de Unix.

Lo primero es saber si el punto en el que estamos es un directorio o un fichero. La función

```
int lstat(char* nombre, struct stat* info);
```

deja en el parámetro `info` un montón de información referente a la entrada llamada `nombre`. El tipo `struct stat` está definido como

```
struct stat {
  mode_t    st_mode;
  long int  st_size;
  ⟨Muchas otras cosas que no nos interesan⟩
};
```

El campo `st_mode` codifica, de una forma un poco larga de explicar, la categoría. Afortunadamente, hay una función predicado para cada posible categoría; nos interesan `S_ISDIR`, que es cierta si la entrada es un directorio, y `S_ISREG`, que es cierta si es un fichero *normal*; en cualquier otro caso lo mejor es hacer como si no lo hubiéramos visto.

El campo `st_mode` también codifica si el fichero es ejecutable. Pero para extraer esta información hay que hacer `st_mode & S_IXUSR`; si el resultado es distinto de 0, el fichero es ejecutable.

El campo `st_size` es el tamaño en octetos (bytes) del fichero, valor con el que podremos completar el atributo de tamaño para las hojas de nuestro árbol.

Para poder hacer uso de todo lo anterior en un programa hay que incorporar

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

Una vez que hemos encontrado un directorio, tendremos que recorrer los elementos que contiene. Un directorio se abre con

```
DIR* opendir(char* nombreDirectorio);
```

No necesitamos saber nada de la forma interna del tipo `DIR`; nos limitaremos a leer las sucesivas entradas con la función

```
struct dirent* readdir(DIR* dir);
```

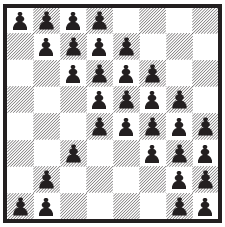
hasta que devuelva `NULL`, valor que indica que ya no queda nada más en el directorio; entonces, lo cerraremos con esta función:

```
int closedir(DIR* dir);
```

Tampoco hay que saber mucho del tipo `struct dirent`: tiene un campo `d_name` con el nombre de la entrada que acabamos de leer.

Para usar estas funciones hay que incluir el fichero de cabecera `dirent.h`.

**6.18a.** Supongamos que utilizamos un tablero de ajedrez como mesa de diseño. Indicamos la colocación de baldosas negras con peones negros; se desea cubrir el resto del tablero con baldosas blancas. Aunque estéticamente tiene un gusto dudoso, considera el ejemplo de diseño que propone el tablero siguiente:



Los peones negros delimitan tres huecos diferentes que deberían llenarse con baldosas blancas. El hueco inferior y el hueco de la izquierda pueden ser llenados con baldosas blancas. Sin embargo, el hueco de la derecha no puede ser cubierto con baldosas blancas.

¿Observas alguna propiedad que puedas utilizar para diferenciar los huecos que no son rellenables con baldosas blancas?

**6.18b.** Para realizar la búsqueda de huecos inconexos en el diseño de baldosas negras necesitas unas buenas estructuras de datos que te ayuden en tu labor.

Cada casilla perteneciente al tablero de diseño es única y podemos nombrarla, por ejemplo, por sus coordenadas e imaginarla como una bola con nombre. Un mecanismo para encontrar los distintos huecos inconexos es el siguiente: al principio, todas las casillas no ocupadas por baldosas negras están en una misma caja inicial. Elegimos una casilla cualquiera de la caja y la ponemos en otra caja nueva vacía. Sacamos de la caja inicial las casillas vecinas a la que se encuentra en la caja nueva y las metemos en la caja nueva; hacemos lo mismo con las vecinas de las vecinas, y así sucesivamente. Cuando ninguna casilla más entre en la caja nueva, todos las que estén en ella forman un hueco conexo. Si aún quedan casillas en la caja inicial, entonces tomamos una de ellas y volvemos a comenzar con otra caja nueva vacía. Al final, tendremos una o varias cajas nuevas y, en cada una de ellas, un conjunto casillas que forman un hueco conexo sobre el que podremos aplicar la prueba de comprobación correspondiente.

## 6.1 Cálculo de la matriz completa de mediotono de Judice-Jarvis-Ninke

301 323

Aunque pueda parecer contradictorio, porque hay que calcular toda la matriz en vez de una sola entrada, este ejercicio es más sencillo que el 3.26. Intentaremos entonces una solución un poco más complicada de lo necesario pero que nos permitirá explorar las posibilidades que nos proporciona el compartir memoria.

### 6.1.1 Submatrices

Imaginemos que queremos calcular $L_2$ y que tenemos $L_1$ en el cuadrante superior izquierdo:

$$\begin{bmatrix} 0 & 3 & ¿? & ¿? \\ 2 & 1 & ¿? & ¿? \\ ¿? & ¿? & ¿? & ¿? \\ ¿? & ¿? & ¿? & ¿? \end{bmatrix}$$

Hay que efectuar los siguientes pasos para convertir esta situación en una matriz $L_2$ correcta: (1) multiplicar por 4 el cuadrante superior izquierdo, (2) copiarlo al resto de cuadrantes, y (3) sumar las constantes 1, 2 y 3 a los elementos de los cuadrantes oportunos. Podríamos condensar cada una de estas operaciones en un procedimiento con una buena ristra de parámetros. Por ejemplo, la operación que copia una región cuadrada de tamaño `m` × `m` desde la posición (`io`, `jo`) de la matriz `mato` a la posición (`id`, `jd`) de `matd` tendría la siguiente cabecera:

```
void subcopia(Matriz matd, int const id, int const jd,
              Matriz mato, int const io, int const jo,
              int const m);
```

Pero un subprograma con siete parámetros es una indicación de que algo va mal. Posiblemente no hemos capturado en forma de tipo un concepto fundamental, en nuestro caso, el de submatriz: un trozo de una matriz.

No vamos a distinguir entre una matriz y una submatriz. Una matriz es siempre una parte

```
typedef struct Region {
  int i;     // Primera fila
  int j;     // Primera columna
  int dim;   // Tamaño
} Region;
```

de una malla de números bidimensional

```
typedef int** MallaMatriz;
```

Juntándolos tenemos:

```
typedef struct Matriz {
  Region region;
  MallaMatriz datos;
} Matriz;
```

La extracción de una submatriz genera un objeto que comparte la malla de números pero que se refiere a otra región:

```
Matriz submatriz(Matriz const& mat, int const i, int const j, int const m) {
  Matriz sub;
  sub.region.i   = mat.region.i + i;
  sub.region.j   = mat.region.j + j;
  sub.region.dim = m;
  sub.datos      = mat.datos;
  return sub;
}
```

Con esta organización, para hacer la copia de submatrices, ya no necesitamos un subprograma repleto de parámetros. Basta con poder copiar una matriz en otra:

```
void copia(Matriz& matd, Matriz mato);
```

La funcionalidad de `subcopia` la conseguimos con la composición de `copia` y `submatriz`:

```
matd = submatriz(matd, id, jd, m);
copia(matd, submatriz(mato, io, io, m));
```

En las operaciones que trabajen con más de una matriz, tendremos que resolver el problema de la coherencia de los tamaños. Para la copia, vamos a decantarnos por traspasar sólo el mínimo de las dos matrices:

```
void copia(Matriz& matd, Matriz mato) {
  int const m = min(matd.region.dim, mato.region.dim);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      matd.datos[matd.region.i+i][matd.region.j+j]
         = mato.datos[mato.region.i+i][mato.region.j+j];
    }
  }
}
```

Las operaciones para sumar o multiplicar por una constante todos los elementos de una matriz son triviales (aquí tienes la suma):

```
void suma(Matriz& matd, int const s) {
  int const m = matd.region.dim;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      matd.datos[matd.region.i+i][matd.region.j+j] += s;
    }
  }
}
```

Finalmente queda cómo construir y destruir una de estas matrices. Cuando queramos una matriz *independiente* del resto, que no comparta su malla, deberemos crearla:

```
Matriz crea(int const m) {
  Matriz mat;
  mat.region.i   = 0;
  mat.region.j   = 0;
  mat.region.dim = m;
  mat.datos      = new int*[m];
  for (int i = 0; i < m; i++) mat.datos[i] = new int[m];
  return mat;
}
```

Liberar la memoria es más delicado. Sólo se debería hacer sobre una matriz original y nunca si tiene submatrices en uso:

```
void libera(Matriz& mat) {
  for (int i = 0; i < mat.region.dim; i++) delete[] mat.datos[i];
  delete[] mat.datos;
  mat.datos = NULL;
}
```

### 6.12 El objetivo

La explicación de cómo pasar de $L_1$ a $L_2$ que hemos visto al principio de la sección anterior es la clave para poder construir un $L_n$ arbitrario. Llamaremos a este método "elevación del primer cuadrante". Dependiendo de en qué extremo empecemos a pensar, obtendremos una solución iterativa o recursiva.

Por el extremo $L_n$ llegamos a la solución recursiva. Nos ponemos en la tesitura de querer $L_n$ y descubrimos que alguien nos tiene que dar $L_{n-1}$:

```
void rLimb(Matriz& matd, int const n) {
  if (n == 0) {
    matd.datos[matd.region.i][matd.region.j] = 0;
  } else {
    rLimb(matd, n-1);
    int m = 1 << (n-1);  // 2^{n-1}
    ⟨Elevación del primer cuadrante de tamaño m⟩
  }
}
```

Luego ya es sólo cosa de "elevar el primer cuadrante" (supuestamente de tamaño $m$):

```
Matriz origen = submatriz(matd, 0, 0, m);
multiplica(origen, 4);
Matriz destino = submatriz(matd, 0, m, m);
copia(destino, origen);  suma(destino, 3);
destino = submatriz(matd, m, 0, m);
copia(destino, origen);  suma(destino, 2);
destino = submatriz(matd, m, m, m);
copia(destino, origen);  suma(destino, 1);
```

Para llegar a la solución iterativa hay que empezar por $L_0$. Empezamos construyendo $L_0$ (trivial) y luego elevamos y elevamos hasta llegar a $L_n$:

```
void iLimb(Matriz& matd, int const n) {
  matd.datos[matd.region.i][matd.region.j] = 0;
  int m = 1;
  int i = 0;
  while (i < n) {
    ⟨Elevación del primer cuadrante de tamaño m⟩
    m <<= 1;
    i++;
  }
}
```

## 6.5 Reglas golombinas

302

***Tipo de datos*** Antes de dar una definición concreta del tipo datos es preferible desarrollar los apartados siguientes. Veamos qué es lo que necesitan del tipo de datos, y a partir de ahí daremos su definición.

- Para calcular las medidas nuevas debemos recorrer el conjunto de medidas con dos bucles anidados. El de mayor nivel recorriendo todos los elementos y el de menor nivel recorriendo sólo los mayores. Una buena implementación para ello consistiría en tener una lista enlazada ordenada de forma ascendente.
- Debemos mezclar las medidas nuevas en la antigua regla. El hecho de disponer de listas enlazadas ordenadas de forma creciente también ayuda a este fin.

En definitiva, implementaremos las RG como listas enlazadas ordenadas de forma creciente sin repeticiones. Para facilitar los procedimientos, tendrán cabecera; el elemento en la cabecera podrá ser cualquiera. Para distinguirla del resto de los elementos ponemos $-1$.

```
typedef struct Medida {
  int numero;
  Medida* siguiente;
} Medida;
typedef Medida* ReglaGolombina;
```

Una regla vacía tendrá únicamente la cabecera:

```
ReglaGolombina reglaVacia() {
  ReglaGolombina resultado = new Medida;
  resultado->numero = -1;
  resultado->siguiente = NULL;
  return resultado;
}
```

Para comprobar que una regla es vacía bastará acceder al elemento siguiente a la cabecera y comprobar que es `NULL`.

```
bool esVacia(ReglaGolombina const regla) {
  return regla->siguiente == NULL;
}
```

Para comprobar si un elemento está o no en la regla, la recorremos hasta encontrar el primer elemento mayor o igual al buscado (si existe); si no existe tal elemento recorremos la lista hasta llegar al final sin haber encontrado nada.

```
bool esta(int const n, ReglaGolombina const regla) {
  Medida* actual = regla->siguiente;
  while (actual != NULL && actual->numero < n) {
    actual = actual->siguiente;
  }
  return (actual != NULL && actual->numero == n);
}
```

La inserción de medidas en la regla hará un recorrido similar; es necesario *acordarse* del elemento anterior para poder insertar justo delante del elemento encontrado:

```
void insertar(int const n, ReglaGolombina& regla) {
  Medida* actual = regla->siguiente;
  Medida* anterior = regla;
  while (actual != NULL && actual->numero < n) {
    anterior = actual;
    actual = actual->siguiente;
  }
  ⟨Añadir n⟩
}
```

A la hora de añadir el elemento `n`, hemos de insertarlo únicamente si no estaba ya en la regla:

```
if (actual == NULL || actual->numero > n) {
  Medida* aux = new Medida;
  aux->numero = n;
  aux->siguiente = actual;
  anterior->siguiente = aux;
}
```

El procedimiento para mostrar la regla la recorrerá imprimiendo los elementos en el fichero:

```
ostream& operator<<(ostream& out, ReglaGolombina const regla) {
  Medida* actual = regla->siguiente;
  out << "[";
  while (actual != NULL) {
    out << actual->numero << " ";
    actual = actual->siguiente;
  }
  out << "]";
  return out;
}
```

Para leer la regla de un fichero, leemos los elementos y utilizamos el procedimiento `insertar`.

```
istream& operator>>(istream& in, ReglaGolombina& regla) {
  regla = reglaVacia();
  while (!in.eof()) {
    int n;
    in >> n;
    insertar(n, regla);
  }
  return in;
}
```

No debemos olvidar un procedimiento para liberar la memoria, que en este caso es similar al del ejercicio del solitario búlgaro (ejercicio 6.6).

***Distancias nuevas (en un paso)*** Las distancias medibles con una RG son aquéllas que son diferencia de dos medidas en la RG

```
ReglaGolombina calcularNuevas(ReglaGolombina const regla) {
  ReglaGolombina resultado = reglaVacia();
  ⟨para cada i en la ReglaGolombina regla⟩ {
    ⟨para cada j > i en la ReglaGolombina regla⟩
      if (!esta(j-i, regla)) {
        insertar(j-i, resultado);
      }
    }
  }
  return resultado;
}
```

Al haber implementado las RG como listas crecientes, el pseudocódigo que aparece en el subprograma anterior se concreta fácilmente:

```
ReglaGolombina calcularNuevas(ReglaGolombina const regla) {
  ReglaGolombina resultado = reglaVacia();
  for (Medida* auxI = regla->siguiente; auxI != NULL; auxI = auxI->siguiente) {
    for(Medida* auxJ = auxI->siguiente; auxJ != NULL; auxJ = auxJ->siguiente) {
      int i = auxI->numero;
      int j = auxJ->numero;
      if (!esta(j-i,regla)) {
        insertar(j-i,resultado);
      }
    }
  }
  return resultado;
}
```

***Distancias nuevas (iterando)*** Para completar una RG calcularemos las medidas que se pueden añadir y las añadiremos. Observemos que el número de medidas que se pueden añadir a una RG según nuestro enunciado es finito.

```
void completar(ReglaGolombina& regla) {
  ReglaGolombina nuevasMedidas;
  do {
    nuevasMedidas = calcularNuevas(regla);
    anyadirNuevas(regla, nuevasMedidas);
  } while (!esVacia(nuevasMedidas));
}
```

Veamos cómo resolver el procedimiento `anyadirNuevas`. Una implementación sencilla consistiría en recorrer la lista de las medidas nuevas e insertar cada medida en la regla:

```
void anyadirNuevas(ReglaGolombina& regla, ReglaGolombina const nueva) {
  Medida* auxN = nueva->siguiente;
  while (auxN != NULL) {
    insertar(auxN->numero, regla);
    auxN = auxN->siguiente;
  }
}
```

Pero observemos que este procedimiento es ineficiente puesto que para insertar un elemento en una RG debemos recorrerla. Podemos aprovechar el hecho de que las RG están implementadas como listas ordenadas ascendentemente para mejorar esa solución. Realizaremos un procedimiento que mezcla los elementos de la regla original y las medidas nuevas. Recorre la regla y la lista de las medidas nuevas de forma simultánea con dos punteros: `auxR` y `auxN` respectivamente. Antes de empezar conviene tener en cuenta los siguientes hechos que simplifican el proceso de mezcla:

- La intersección de la regla original con las medidas nuevas es vacía.
- La máxima medida de las reglas nuevas es menor que la mayor medida de la regla original y por tanto siempre llegará antes al final el puntero que recorre la lista de medidas nuevas (`auxN`).

```
void anyadirNuevas(ReglaGolombina& regla, ReglaGolombina const nueva) {
  Medida* auxR = regla->siguiente;
  Medida* antR = regla;
  Medida* auxN = nueva->siguiente;
  while (auxN != NULL) {
      if (auxR->numero < auxN->numero) {
      antR = auxR;
      auxR = auxR->siguiente;
    } else {// auxR->numero < auxN->numero,
            // ya que la intersección entre regla y nueva es vacía
      Medida* aux = new Medida;
      aux->numero = auxN->numero;
      aux->siguiente = auxR;
      antR->siguiente = aux;
      antR = aux;
      auxN = auxN->siguiente;
    }
  }
}
```

## 6 6 Solitario búlgaro 303

La descripción inicial del juego se puede expresar mediante el siguiente seudocódigo:

⟨*Elegir el tamaño del mazo y formar el lote inicial*⟩
`while (!`⟨*se alcance una situación final*⟩`) {`
  ⟨*Reorganizar las cartas del modo descrito*⟩
`}`

Procederemos descendentemente, sustituyendo las acciones sin desarrollar por llamadas a los subprogramas correspondientes:

```
int main() {
  Lote lote = formarLoteInicial();
  while (!esSituacionFinal(lote)) {
    cout << lote << endl;
    reorganizar (lote);
  }
  cout << "Configuración final:  " << lote << endl;
}
```

Y surge así la necesidad de escoger una representación para el `lote` de montones y desarrollar los subprogramas descritos. Vamos por partes:

### 6.6.1 El lote de montones

La información necesaria del lote de montones consiste en los tamaños de los mismos, ya que no es necesario en ningún momento conocer el valor de las cartas. Es decir, se requiere un entero positivo (estrictamente) por cada montón. Por otra parte, el lote consiste en un número de montones desconocido *a priori* y variable en las sucesivas reorganizaciones.

En resumen, una estructura apropiada para el lote de montones es una lista de enteros:

```
typedef struct Monton {
  int num;
  Monton* sig;
} Monton;
typedef Monton* Lote;
```

Además, para una mayor comodidad, usaremos en todo momento listas con cabecera. Una lista vacía tendrá siempre un elemento (la cabecera) conteniendo un elemento que no puede aparecer en la lista (un $-1$ es un valor adecuado para tal fin):

```
Lote loteVacio() {
  Lote lista = new Monton;
  lista->sig = NULL;
  lista->num = -1;
  return lista;
}
```

Al estar definiendo una estructura dinámica se debe prever un procedimiento para liberar la memoria que ocupa. Para ello debemos realizar un recorrido borrando los nodos (incluyendo la cabecera). Puesto que se debe avanzar en el recorrido antes del borrado de cada nodo (de otra forma perderíamos la estructura), debemos usar una variable auxiliar para *acordarnos* del nodo visitado anteriormente y borrarlo:

```
void liberar(Lote& lote) {
  Monton* actual = lote;
  while (actual != NULL) {
    Monton* aux = actual;
    actual = actual->sig;
    delete aux;
  }
  lote = NULL;
}
```

No podemos olvidar añadir la línea

```
liberar(lote);
```

al final del `main` y, en general, cuando `lote` ya no sea necesario.

### 6.6.2 Condiciones iniciales

Una forma de establecer la situación inicial consiste en hallar el tamaño del mazo y luego repartir las cartas en montones construyendo así la lista inicial:

```
Lote formarLoteInicial() {
  int n = calcularTamanyo();
  return formarLoteIncial(n);
}
```

Optamos por dejar al azar la elección del tamaño del mazo, así como el reparto inicial (consúltese el ejercicio 3.13). Ahora, veamos el modo de llevar a cabo los dos pasos:

1. Escoger el tamaño del mazo.

   Como $n$ debe ser un número triangular ($n = 1+2+\ldots+k$ para algún $k \geq 1$), empezamos por elegir el número $k$ descrito, y luego hallamos $n = \frac{k(k+1)}{2}$. Definiremos una constante `maxK` que nos defina el valor máximo de cartas que habrá en un determinado montón. Hemos de tener cuidado para que las operaciones no se salgan de rango; en particular se debe cumplir que `maxK`$($`maxK`$+1) \leq$ `INT_MAX`.

   ```
   int const maxK = ⟨69 por ejemplo⟩;
   int calcularTamanyo() {
     int k = dado(maxK);
     return (k*(k+1))/2;
   }
   ```

2. Formar la lista de montones.

   Este reparto puede llevarse a cabo de diferentes formas. Partiendo de la lista vacía, una posibilidad consiste en ir generando cada montón con un número de cartas aleatorio, entre 1 y las que queden aún por repartir.

   ```
   Lote formarLoteInicial(int const total) {
     Lote lista = loteVacio();
     int quedan = total;
     do {
       int monton = dado(quedan);
       ⟨Añadir monton a la lista⟩
       quedan = quedan - monton;
     } while(quedan>0);
     return lista;
   }
   ```

   Vamos a añadir los nuevos elementos generados al final de la lista; para ello necesitamos acceder al último elemento de la lista. Una primera solución podría consistir en hacer un procedimiento que devolviera dicho elemento, pero eso es altamente ineficiente puesto que ese procedimiento debería recorrer toda la lista. Siendo algo más avispados podemos introducir una variable `ultimo` que en cada vuelta apunte al último nodo generado de la lista y así podremos insertar cómodamente detrás del mismo. No debemos olvidarnos de actualizar `ultimo` de forma que apunte al nuevo nodo. Para ello introducimos un procedimiento `anyadirMonton` que añade un elemento tras uno dado:

```
void anyadirMonton(Monton* const ultimo, int const n) {
  Monton* aux = new Monton;
  aux->num = n;
  aux->sig = NULL;
  ultimo->sig = aux;
}
```

No debemos olvidar actualizar `ultimo` para que apunte al último elemento de la lista. Así pues, ⟨*Añadir* `monton`...⟩ queda como sigue:

```
anyadirMonton(ultimo, monton);
ultimo = ultimo->sig;
```

Por último no debemos olvidarnos de declarar la variable `ultimo` como un puntero a `Monton` y de darle un valor inicial antes de entrar en el bucle. Puesto que estamos usando listas con cabeceras, el último nodo de una lista vacía es precisamente la cabecera. Así antes del `do` deberemos introducir la siguiente línea

```
Monton* ultimo = lista;
```

El procedimiento para *mostrar el lote* consiste trivialmente en un recorrido de la lista:

```
ostream& operator<<(ostream& out, Lote const& lote) {
  Monton* aux = lote->sig;
  out << "[";
  while (aux != NULL) {
    out << aux->num << " ";
    aux = aux->sig;
  }
  out << "]";
  return out;
}
```

### 6.6.3 Reorganización del mazo

Para *reorganizar el mazo*, recorremos la lista de montones. A cada montón le quitamos una carta, y aquéllos que se queden con 0 cartas serán eliminados de la lista. Es necesario llevar un contador para saber cuántas cartas hemos eliminado para, al final, añadir una nueva celda con ese número de cartas.

```
void reorganizar(Lote& lote) {
  Monton* actual = lote->sig;
  int numEliminados = 0;
  while (actual != NULL) {
    actual->num--;
    numEliminados++;
    if (actual->num == 0) {
      ⟨Borrar actual⟩
    }
    ⟨Avanzar actual⟩
  }
  ⟨Añadir numEliminados⟩
}
```

Como podemos observar aparecen problemas nuevos:

- ⟨*Borrar* `actual`⟩

  Borrar el elemento actual (en caso de contener un 0). Para realizar esta tarea fácilmente necesitamos una variable que en cada vuelta del bucle apunte al elemento previo a `actual` en la lista, al que llamaremos `anterior`. Así pues, ⟨*Borrar* `actual`⟩ queda como sigue:

  ```
  anterior->sig = actual->sig;
  delete actual;
  ```

- ⟨*Avanzar* `actual`⟩

  En principio, para avanzar `actual` bastaría con hacer `actual = actual->sig` . Pero aparece un problema: si entramos en el condicional `if (actual->num == 0) {...}` y borramos `actual`: no tenemos ninguna referencia al elemento al que apuntaba `actual`. Afortunadamente, tenemos el puntero `anterior`, de forma que `anterior->sig` apunta al nodo que, antes del borrado, era el siguiente a `actual`. Por tanto, si hemos borrado `actual`, la instrucción ⟨*Avanzar* `actual`⟩ debe hacer que éste valga `anterior->sig`. Por añadidura tenemos que `anterior` apunta al nodo anterior a `actual`.

  Si no hemos borrado el elemento, debemos mover `actual` a `actual->sig`; puesto que movemos `actual`, será necesario también mover `anterior`.

  La manera más fácil de realizar esto es distribuir la acción de ⟨*Avanzar* `actual`⟩ en el condicional anterior, quedando éste así:

  ```
  if (actual->num == 0) {
    anterior->sig = actual->sig;
    delete actual;
    actual = anterior->sig;
  } else {
    anterior = actual;
    actual = actual->sig;
  }
  ```

- ⟨*Añadir* `numEliminados`⟩

  Para realizar esta acción basta observar que cuando salimos del bucle, `anterior` apunta al último elemento de la lista. Así pues, esta acción queda como sigue:

  ```
  anyadirMonton(anterior,numEliminados);
  ```

Para acabar el procedimiento `reorganizar` debemos declarar y definir adecuadamente la variable `anterior`. Recordando otra vez que estamos utilizando listas con cabecera, el nodo anterior al primero de la lista será precisamente la cabecera. Así pues, antes del `while` deberemos introducir la siguiente línea:

```
Monton* anterior = lote;
```

### 6.6.4 ¿Es esto el fin?

Y ya sólo falta la función que comprueba si la situación a la que hemos llegado es final o no. Esta función deberá averiguar si el `lote` consiste en una secuencia ascendente de la forma 1, 2, 3...

Habrá que buscar el primer elemento que no se corresponda con su posición en la lista, y si no existe tal elemento estaremos en una *situación final*:

```
bool esSituacionFinal(Lote const lote) {
  int n = 1;
  Monton* aux = lote->sig;
  while(aux != NULL && aux->num == n) {
    aux=aux->sig;
    n++;
  }
  return aux == NULL;
}
```

## 6.9 Cuadrados mágicos

305 323

El tipo de datos en el que vamos a representar los cuadrados mágicos es una matriz cuadrada de dimensión arbitraria, es decir, es una matriz dinámica, cuyo tamaño lo estableceremos en tiempo de ejecución, y por tanto será necesario crearla con un `new`. La definición más sencilla es como un registro formado por el contenido propiamente dicho más el tamaño del cuadrado:

```
typedef struct CuadradoMagico {
  int** contenido;
  int tamanyo;
} CuadradoMagico;
```

Para hacer el resto del ejercicio más independiente del tipo de datos elegido, iremos definiendo subprogramas de acceso y modificación al mismo. En todos ellos aparecerá de forma natural el tipo `Posicion` del cuadrado:

```
typedef struct Posicion {
  int fila, columna;
} Posicion;
```

Como hemos decidido que la matriz que representa el cuadrado sea dinámica, debemos realizar un procedimiento que libere la memoria de un cuadrado mágico:

```
void liberar(CuadradoMagico & cuadrado) {
  for (int fila = 0; fila < cuadrado.tamanyo; fila++) {
    delete [] cuadrado.contenido[fila];
  }
  delete [] cuadrado.contenido;
  cuadrado.contenido = NULL;
}
```

***Comprueba cuadrado*** Realizaremos una función que compruebe si un cuadrado `esMagico` o no. Para realizar la comprobación es necesario sumar las filas, las columnas y las diagonales.

Una solución sencilla consistiría en recorrer la matriz por filas y sumar cada una de ellas, recorrer la matriz por columnas y sumar cada una de ellas y, finalmente, recorrer las diagonales y sumarlas. También podemos economizar cálculos y recorrer una única vez la matriz realizando las sumas adecuadas. La idea consiste en llevar un contador por cada fila, columna y diagonal. Recorremos toda la matriz y en cada casilla incrementamos el contador de su fila, su columna y el de las diagonales a las que pertenezca; por último el cuadrado será mágico si todos los contadores son iguales. Con esto, la función `esMagico` queda como sigue:

```
bool esMagico(CuadradoMagico const& cuadrado) {
  Contadores contadores = ⟨valor inicial para los contadores⟩;
  Posicion pos;
  for (pos.fila = 0; pos.fila < cuadrado.tamanyo; pos.fila++) {
    for (pos.columna = 0; pos.columna < cuadrado.tamanyo; pos.columna++) {
      ⟨Incrementar el contador de la fila⟩
      ⟨Incrementar el contador de la columna⟩
      ⟨Incrementar el contador de la diagonal derecha⟩
      ⟨Incrementar el contador de la diagonal izquierda⟩
    }
  }
  return ⟨true ⇔ todos los contadores son iguales⟩;
}
```

Para poder refinar el código es necesario saber la definición de tipos que vamos a utilizar para los contadores. Quizá la forma más sencilla es utilizar un array de tamaño $2N+2$ donde $N$ es el tamaño del cuadrado que se está analizando; las $N$ primeras $0 \leq i < N$ almacenarán los contadores de las filas, las siguientes $N$ los de las columnas y las dos últimas posiciones almacenarán el contador de las diagonales:

$$\begin{array}{cl} 0 \leq i < N & \text{el contador de la fila } i \\ N \leq i < 2N & \text{el contador de la columna } i - N \\ i = 2N & \text{el contador de la diagonal izquierda} \\ i = 2N + 1 & \text{el contador de la diagonal derecha} \end{array}$$

Así la definición de tipos para los contadores será un array, pero comoquiera que el tamaño no está prefijado deberemos llevar también un contador del tamaño:

```
typedef struct Contadores {
  int* contenido;
  int numeroContadores;
} Contadores;
```

La función que define inicialmente los contadores debe hacerlo con el valor 0:

```
Contadores iniciarContadores(CuadradoMagico const& cuadrado) {
  Contadores resultado;
  resultado.numeroContadores = 2*cuadrado.tamanyo+2;
  resultado.contenido = new int[resultado.numeroContadores];
  for (int i = 0; i < resultado.numeroContadores; i++) {
    resultado.contenido[i] = 0;
  }
  return resultado;
}
```

Puesto que el array de los contadores se está creando de forma dinámica, es necesario prever la liberación de la memoria ocupada por los mismos:

```
void liberar(Contadores & contadores) {
  delete [] contadores.contenido;
  contadores.contenido = NULL;
}
```

Será necesario invocar a este procedimiento antes de abandonar la función `esMagico`, por lo que debemos

cambiar el `return` de la función anterior por las siguientes instrucciones:

```
bool esMagico = ⟨true ⇔ todos los contadores son iguales⟩;
liberar(contadores);
return esMagico;
```

Los procedimientos que incrementan los contadores son muy parecidos entre sí. Los que incrementan el contador de una fila o una columna simplemente deben añadir el valor de la posición actual al contador de la fila o columna. Los que incrementan los contadores de las diagonales deben comprobar además si la posición actual está en una diagonal. A continuación se da el procedimiento que incrementa el valor de la diagonal derecha:

```
bool estaEnDiagonalDerecha(Posicion const& posicion,
                           CuadradoMagico const& cuadrado) {
  return (posicion.fila+posicion.columna) == cuadrado.tamanyo-1;
}
void incrementarContadorDiagonalDerecha(Posicion const& posicion,
                                        Contadores& contadores,
                                        CuadradoMagico const& cuadrado) {
  if (estaEnDiagonalDerecha(posicion,cuadrado)) {
    contadores.contenido[2*cuadrado.tamanyo+1] += valorEn(posicion,cuadrado);
  }
}
```

La función que comprueba si todos los contadores son iguales debe recorrerlo hasta encontrar uno que sea diferente a los anteriores:

```
bool sonTodosContadoresIguales(Contadores const& contadores) {
  int valor = contadores.contenido[0];
  for (int i = 1; i < contadores.numeroContadores; i++) {
    if (contadores.contenido[i] != valor) return false;
  }
  return true;
}
```

En la función `incrementarContadorDiagonalDerecha` se usa la función `valorEn`. Ésta es complementaria al procedimiento `ponerEn` que se define más adelante.

***Cuadrados mágicos de lado impar*** El método descrito nos propone, en esencia, colocar unos ciertos números dentro de la matriz siguiendo un patrón y el resto trasladarlos de forma adecuada. Para realizar esto podemos imaginar una matriz grande dentro de la cual está el cuadrado mágico que queremos rellenar. Véase figura 6.5.

Empezamos por la diagonal más a la derecha y colocamos los números del 1 al 5. El único número que cae dentro del cuadrado es el 3 en la fila 0 y columna 4, el número 2 está en la fila $-1$ y columna 3, el uno en la fila $-2$ y columna 2, el 4 en la fila 1 y columna 5 y, por último, el 5 en la fila 2 y columna 6. Si nos fijamos en las casillas en las que hemos colocado cada número observamos lo siguiente:

$$\mathit{columna} - \mathit{fila} = 4$$

Y haciendo lo mismo para las siguientes diagonales, observamos que:

$$\mathit{columna} - \mathit{fila} = 4, 2, 0, -2, -4$$

respectivamente. Así podemos identificar cada diagonal por la diferencia $\mathit{columna} - \mathit{fila}$.

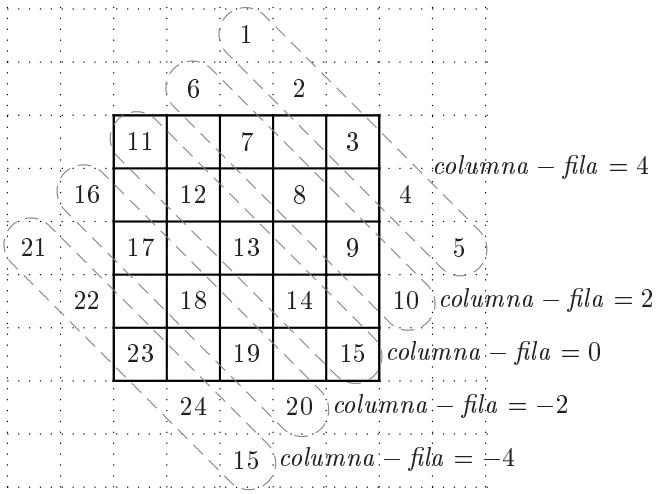



Figura 6.5: Método para cuadrados mágicos de lado impar

En general tendremos $N$ diagonales, siendo $N$ el tamaño del cuadrado mágico, y cada diagonal tendrá $N$ elementos. La primera diagonal verificará que $columna - fila = N - 1$; el primer elemento de esa diagonal estará en la fila $-\lfloor N/2 \rfloor$ y la columna vendrá dada por la fórmula que identifica la diagonal. Para calcular la siguiente diagonal será necesario restar 2 y para calcular la fila del primer elemento de esa diagonal será necesario sumar 1 a la fila del primer elemento de la diagonal anterior. Así podemos realizar la función que devuelve un cuadrado mágico de tamaño impar:

```
CuadradoMagico construirMagicoImpar(int const tamanyo) {
  CuadradoMagico resultado = ⟨cuadrado vacío de tamaño tamanyo⟩;
  int primeraFila = -(tamanyo/2);
  int diagonal = tamanyo-1;
  int valor = 1;
  for (int i = 0; i < tamanyo; i++) {
    for (int j = 0; j < tamanyo; j++) {
      Posicion posicion;
      posicion.fila = primeraFila+j;
      posicion.columna = diagonal+posicion.fila;
      ⟨Poner valor en la posicion del cuadrado⟩
      valor++;
    }
    primeraFila++;
    diagonal = diagonal-2;
  }
  return resultado;
}
```

La construcción de un cuadrado vacío se resuelve fácilmente con dos bucles anidados, uno recorriendo las filas y otro las columnas, dando un valor inicial a cada casilla del cuadrado:

```
CuadradoMagico construirVacio(int const tamanyo) {
  CuadradoMagico resultado;
  resultado.tamanyo = tamanyo;
  resultado.contenido = new int*[tamanyo];
  for (int i = 0; i < tamanyo; i++) {
    resultado.contenido[i] = new int[tamanyo];
    for (int j = 0; j < tamanyo; j++) {
      resultado.contenido[i][j] = 0;
    }
  }
  return resultado;
}
```

Para completar el procedimiento falta refinar el código para poner un valor en una casilla del cuadrado. En principio habría que usar un cuadrado de dimensión mayor al requerido para, a continuación, trasladar los valores que no están dentro del cuadro. Pero si pensamos un poco más podemos hacer las dos operaciones juntas: si la posición en que queremos poner el valor está dentro del cuadrado, se pone el valor en él; en caso contrario hacemos la traslación adecuada:

```
void ponerEn(int const valor, Posicion const& pos, CuadradoMagico& cuadrado) {
  Posicion posAux = pos;
  if (⟨posAux no está dentro del cuadrado⟩) {
    posAux = ⟨posición a la que debe trasladarse posAux dentro de cuadrado⟩
  }
  cuadrado.contenido[posAux.fila][posAux.columna] = valor;
}
```

La traslación es simple: si un cuadrado no está dentro es porque la fila está por *encima* o por *debajo*, o la columna está a la *derecha* o a la *izquierda*. En todos los casos las traslaciones son de longitud `cuadrado.tamanyo` dentro de la misma fila o columna:

```
Posicion trasladarPosicion(Posicion const& pos, CuadradoMagico const& cuadrado) {
  Posicion resultado = pos;
  if (resultado.fila < 0) {
    resultado.fila = resultado.fila + cuadrado.tamanyo;
  } else if (resultado.fila >= cuadrado.tamanyo) {
    resultado.fila = resultado.fila - cuadrado.tamanyo;
  } else if (resultado.columna < 0) {
    resultado.columna = resultado.columna + cuadrado.tamanyo;
  } else if (resultado.columna >= cuadrado.tamanyo) {
    resultado.columna = resultado.columna - cuadrado.tamanyo;
  }
  return resultado;
}
```

La función para averiguar si una determinada posición está *dentro* del cuadrado es bastante simple:

```
bool estaDentro(Posicion const& pos, CuadradoMagico const& cuadrado) {
  return (0 <= pos.fila && pos.fila < cuadrado.tamanyo) &&
         (0 <= pos.columna && pos.columna < cuadrado.tamanyo);
}
```

***Cuadrados con lado múltiplo de cuatro*** Si quisiéramos marcar las componentes de un cuadrado y luego escribir los números necesitaríamos complicar el tipo de datos de los cuadrados mágicos. Para ello introducimos una constante distinta de cero con la que identificar las posiciones marcadas:

```
int const valorMarcado = 69;
```

Entonces, para marcar una casilla basta con poner el valor en esa posición, y para comprobar si una casilla está marcada basta con ver si está ese valor en la posición:

```
void marcarPosicion(Posicion const& posicion, CuadradoMagico& cuadrado) {
  cuadrado.contenido[posicion.fila][posicion.columna] = valorMarcado;
}
bool esPosicionMarcada(Posicion const& posicion, CuadradoMagico const& cuadrado) {
  return cuadrado.contenido[posicion.fila][posicion.columna] == valorMarcado;
}
```

Teniendo en cuenta estas dos funciones, la construcción del cuadrado mágico se puede dividir en dos: primero el marcado y, tras él, el proceso de relleno; esto es:

```
CuadradoMagico construirMagicoMultiplo4(int const tamanyo) {
  CuadradoMagico resultado;
  resultado = construirVacio(tamanyo);
  marcarCasillas(resultado);
  rellenarCuadrado(resultado);
  return resultado;
}
```

Tal y como se puede ver en la figura 6.6, para realizar el marcado conviene distinguir las cuatro diagonales: superior izquierda (SI), superior derecha (SD), inferior izquierda (II) e inferior derecha (ID). De cada una de esas diagonales hay tantas como $\lfloor N/4 \rfloor$ siendo $N$ el tamaño del cuadrado. Entonces el

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SI | • | • | • | | | | | | | | • | • | SD | • |
| | SI | • | • | • | | | | | | | • | • | SD | • | |
| | | SI | • | • | • | | | | | • | • | SD | • | | |
| | | | SI | • | • | • | | | • | • | SD | • | | | |
| | | | | SI | • | • | • | • | • | SD | • | | | | |
| • | | | | | SI | • | • | • | SD | • | | | | | • |
| • | • | | | | | SI | • | SD | • | | | | | • | • |
| • | • | • | | | | | SI | • | | | | | • | • | SD |
| • | II | • | | | | | • | • | | | | | ID | • | • |
| II | • | | | | | • | • | II | • | | | | | ID | • |
| • | | | | | • | • | II | • | • | • | | | | | ID |
| | | | | • | • | II | • | ID | • | • | • | | | | |
| | | | • | • | II | • | | | ID | • | • | • | | | |
| | | • | • | II | • | | | | | ID | • | • | • | | |
| | • | • | II | • | | | | | | | ID | • | • | • | |
| • | • | II | • | | | | | | | | | ID | • | • | • |

Figura 6.6: Construcción del marcado por diagonales

procedimiento de marcado deberá marcar todas las diagonales:

```
void marcarCasillas(CuadradoMagico& cuadrado) {
  int numDiagonales = cuadrado.tamanyo/4;
  for (int diagonal = 0; diagonal < numDiagonales; diagonal++) {
    ⟨Marcar las diagonales⟩
  }
}
```

En cada una de las diagonales hay exactamente $\lfloor N/2 \rfloor$ elementos. Podemos rellenar todas las diagonales a la vez; para ello recorremos todas las posiciones de cada diagonal, empezando por los extremos y avanzando hacia el centro:

```
Posicion diagSupDcha, diagSupIzq;
Posicion diagInfDcha, diagInfIzq;
⟨valores iniciales a diagSupDcha, diagSupIzq, diagInfDcha y diagInfIzq⟩
for (int j = 0; j < cuadrado.tamanyo/2; j++) {
  ⟨Marcar diagSupDcha, diagSupIzq, diagInfDcha y diagInfIzq⟩
  ⟨Avanzar diagSupDcha, diagSupIzq, diagInfDcha y diagInfIzq⟩
}
```

Los valores iniciales de cada una de las diagonales son los siguientes:

| | fila | columna |
|---|---|---|
| diagSupDcha | 0 | tamanyo − diagonal − 1 |
| diagSupIzq | 0 | diagonal |
| diagInfDcha | tamanyo − 1 | tamanyo − diagonal − 1 |
| diagInfIzq | tamanyo − 1 | diagonal |

El marcado de cada diagonal es muy sencillo: simplemente se debe marcar cada una de las posiciones `diagSupDcha`, `diagSupIzq`, `diagInfDcha` y `diagInfIzq`.

Cada diagonal empieza en su extremo y va hacia el centro; hay que tener en cuenta que cuando la columna de la diagonal sobrepasa el eje central del cuadrado, la posición de la diagonal va a la primera o última columna dependiendo de si es una diagonal izquierda o derecha respectivamente. Así, el procedimiento de `avanzarDiagonales` queda como sigue:

```
void avanzarDiagonales(Posicion& diagSupDcha, Posicion& diagSupIzq,
                       Posicion& diagInfDcha, Posicion& diagInfIzq,
                       int const tamanyo) {
  diagSupDcha.fila++; diagSupIzq.fila++;
  diagInfDcha.fila--; diagInfIzq.fila--;
  diagSupDcha.columna--; diagInfDcha.columna--;
  diagSupIzq.columna++; diagInfIzq.columna++;
  if (diagSupIzq.columna >= tamanyo/2) diagSupIzq.columna = 0;
  if (diagInfIzq.columna >= tamanyo/2) diagInfIzq.columna = 0;
  if (diagSupDcha.columna < tamanyo/2) diagSupDcha.columna = tamanyo-1;
  if (diagInfDcha.columna < tamanyo/2) diagInfDcha.columna = tamanyo-1;
}
```

Falta por último el procedimiento que pone los números en el cuadrado ya marcado. Según el procedimiento descrito en el enunciado, es necesario recorrer el cuadrado dos veces: una de arriba abajo, y otra de abajo arriba. Pero si profundizamos un poco más en la solución propuesta descubrimos que un solo recorrido es suficiente. Para implementar la solución propuesta en el enunciado, es necesaria una variable

`valor`, con valor inicial 1, que se vaya incrementando a medida que avanzamos por cada casilla. En las casillas marcadas se escribe su `valor`. En el segundo recorrido hacemos lo mismo pero empezando por abajo, y ponemos el valor de la variable `valor` cuando llegamos a las casillas no marcadas. Si observamos el resultado, podemos obtener lo mismo con dos variables auxiliares: `valorMarcado` y `valorNoMarcado`. La primera hace el papel del primer recorrido: empieza valiendo 1, se va incrementando en cada casilla y su valor se pone en las casillas no marcadas. La segunda hace el papel del segundo recorrido: empieza valiendo $N^2$ ($N$ es el tamaño del cuadrado), disminuye en cada paso y su valor se pone en las casillas no marcadas:

```
void rellenarCuadrado(CuadradoMagico& cuadrado) {
  int valorMarcado = 1;
  int valorNoMarcado = cuadrado.tamanyo*cuadrado.tamanyo;
  Posicion pos;
  for (pos.fila = 0; pos.fila < cuadrado.tamanyo; pos.fila++) {
    for (pos.columna = 0; pos.columna < cuadrado.tamanyo; pos.columna++) {
      if (esPosicionMarcada(pos, cuadrado)) {
        ponerEn(valorMarcado, pos, cuadrado);
      } else {
        ponerEn(valorNoMarcado, pos, cuadrado);
      }
      valorMarcado++;
      valorNoMarcado--;
    }
  }
}
```

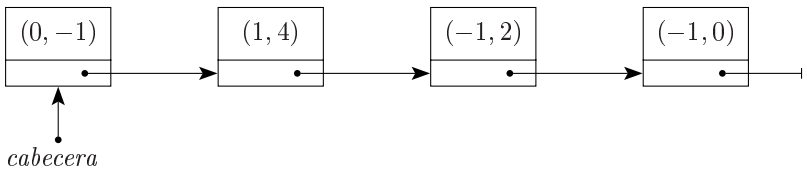
## 6.10 Implementación de polinomios

309

***Tipo de datos*** Realizaremos una implementación de polinomios basada en memoria dinámica. Representaremos los polinomios como listas enlazadas de *monomios*. Éstos vienen caracterizados por su grado (entero positivo) y por el coeficiente, que supondremos un racional. Como es habitual ordenar los monomios mediante su grado, consideraremos que las listas que representan los polinomios están así ordenadas; además, en nuestra implementación no existirán monomios con coeficiente nulo. En particular el polinomio 0 se implementará mediante una lista vacía.

Pronto veremos que, para desarrollar las operaciones de suma, resta y multiplicación, poco importa si el orden de los monomios es creciente o decreciente; pero en la división es mejor disponer de los monomios en orden decreciente.

Por último, las listas que representan los polinomios van a tener *cabecera*, es decir, un nodo especial, al principio, que sirve para dar acceso a la lista de monomios: el grado del monomio de la cabecera será $-1$. La cabecera nos permitirá realizar las operaciones de forma homogénea, evitando distinguir entre listas vacías o no en cada operación. Así, el polinomio $x^4 - x^2 - 1$ se representará mediante la lista



y ninguna otra lista representará a ese polinomio. La formalización de estos tipos es bastante sencilla: la lista de monomios se realiza enlazando celdas,

```
typedef struct CeldaMonomio {
  Monomio monomio;
  CeldaMonomio* siguiente;
} CeldaMonomio;
```

y el polinomio apunta sencillamente a la cabecera de la lista:

```
typedef CeldaMonomio* Polinomio;
```

Un monomio se puede definir como una tupla: una fracción que representa su coeficiente y un entero que representa su grado:

```
typedef struct Monomio {
  Fraccion coeficiente;
  int grado;
} Monomio;
```

Se deja la definición del tipo `Fraccion`, junto a todas sus operaciones, como ejercicio pendiente para el lector. Puesto que estamos definiendo una estructura en memoria dinámica, conviene dar un procedimiento que libere la memoria ocupada por un polinomio. Éste deberá recorrer la lista de monomios del polinomio liberando la memoria ocupada por los mismos:

```
void liberar(Polinomio& polinomio) {
  CeldaMonomio* actual = polinomio;
  while (actual != NULL) {
    CeldaMonomio* aux = actual;
    actual = actual->siguiente;
    delete aux;
  }
}
```

***Suma y resta de polinomios*** Las características de la memoria dinámica en C++ obligarían a una implementación en la que muchas de las operaciones con polinomios se materializarían en procedimientos. Así por ejemplo, la suma tiene esta cabecera:

```
void sumar(Polinomio const a, Polinomio const b, Polinomio& resultado);
```

Sin embargo este tipo de implementación pierde abstracción. Todos pensamos en las operaciones entre polinomios exactamente igual que en las operaciones entre enteros: ambas son funciones en sus respectivos dominios. Por ello parece más conveniente implementar las operaciones como aquéllas para enteros (véase ejercicio 5.19):

```
Polinomio operator+(Entero const& sumandoA, Entero const& sumandoB);
```

Parece que, con esta decisión, hemos resuelto todos nuestros problemas. Pero nada más lejos de la realidad: ahora, se podrían escribir expresiones como ésta: `p1 + p2 + p3`. ¿Qué problema hay? (Cinco minutos para la reflexión...)

Al evaluar esta expresión estamos construyendo de forma implícita el polinomio resultado de evaluar `p1 + p2` (al que llamaremos `auxSuma`) y a éste le agregamos `p3`. ¿Ves ya el problema? (Haremos otra

pausa antes de seguir...)

Efectivamente, hemos creado una estructura dinámica; el polinomio `auxSuma` y la memoria que ocupa no ha sido liberada. La memoria dinámica que no se libera siempre acaba siendo fuente de innumerables desgracias. Así que la expresión `p1 + p2 + p3` no se puede evaluar directamente; es necesario hacerla por pasos:

```
Polinomio auxSuma = p1 + p2;
Polinomio resultado = auxSuma + p3;
liberar(auxSuma);
```

O sea, que seguimos perdiendo abstracción, ya que seguimos sin poder utilizar las funciones como desearíamos.

Con todo esto vemos que es imposible realizar una implementación en C++ de las operaciones que nos satisfaga plenamente. Ambas soluciones tienen sus inconvenientes y sus ventajas: con la primera, se pierde abstracción y es más *incómoda* de usar que la segunda, pero parece más conveniente para no dejar basura en la memoria. En el resto del desarrollo optamos por esta solución.

Así pues, realizaremos un procedimiento para la suma, que consiste en recorrer simultáneamente los sumandos (los polinomios `a` y `b`) generando a la vez el resultado (el polinomio `resultado`, inicialmente nulo):

```
void sumar(Polinomio const a, Polinomio const b, Polinomio& resultado) {
  resultado = ⟨polinomio nulo⟩;
  ⟨Recorrer a y b, generando resultado⟩
}
```

Hablemos ahora del recorrido simultáneo de estos tres polinomios. Para ello, llevaremos tres punteros auxiliares: `auxA` y `auxB`, señalando a los nodos actuales de los polinomios sumandos, y `antR`, apuntando al último nodo creado del polinomio `resultado` que estamos calculando.

Inicialmente, `resultado` es el polinomio nulo (es decir, una lista vacía consistente únicamente en la cabecera), por lo que `antR` apuntará a dicho nodo cabecera. En cambio, tanto `auxA` como `auxB` apuntan inicialmente al primer nodo *útil* de las listas `a` y `b`. En cuanto al recorrido en sí, habrá que avanzar hasta agotar *todos* los términos (monomios) de `a` y `b`. Pero vamos a avanzar primero hasta terminar con uno de ellos, y luego recorreremos el otro, ya sólo para copiarlo:

```
CeldaMonomio* auxA = a->siguiente;
CeldaMonomio* auxB = b->siguiente;
CeldaMonomio* antR = resultado;
while (auxA != NULL && auxB != NULL) {
  ⟨Avanzar un término, comparando los grados de auxA y auxB⟩
}
if (auxA != NULL) antR->siguiente = copiarListaMonomios(auxA);
if (auxB != NULL) antR->siguiente = copiarListaMonomios(auxB);
```

El cuerpo del primer bucle avanza comparando los grados de ambos monomios: si el grado señalado por `auxA` es menor que el de `auxB`, lo añadimos a la suma y avanzamos el puntero `auxA`. Lo mismo con `auxB`, en su caso. Cuando ambos grados son iguales, añadimos el monomio del mismo grado, sumando los coeficientes, y avanzamos en ambos monomios. Así pues, el seudocódigo

⟨*Avanzar un término, comparando los grados de* `auxA` *y* `auxB`⟩

se refina así:

```
if (auxA->monomio.grado>auxB->monomio.grado) {
  anyadirMonomio(antR, auxA->monomio);
  auxA = auxA->siguiente;
} else if (auxA->monomio.grado<auxB->monomio.grado) {
  anyadirMonomio(antR, auxB->monomio);
  auxB = auxB->siguiente;
} else { // Los grados son iguales
  anyadirMonomio(antR, auxA->monomio + auxB->monomio);
  auxA = auxA->siguiente;
  auxB = auxB->siguiente;
}
```

Vamos con los detalles pendientes. Para añadir los monomios que siguen a `auxA` o `auxB`, recurrimos a una función que devuelva una copia de una lista de monomios dada. Su implementación sólo exige recorrer la lista dada, generando una copia de cada celda. Usaremos una variable auxiliar `auxL` para recorrer la lista de entrada. Para crear la nueva lista añadimos los monomios de la lista accedida a la nueva, y para ello es útil un puntero al último nodo generado de la nueva lista que llamaremos `ultimoNueva`:

```
CeldaMonomio* copiarListaMonomios(CeldaMonomio* const listaMonomios) {
  CeldaMonomio* auxL = listaMonomios;
  CeldaMonomio* ultimoNueva;
  ⟨Primer valor para ultimoNueva⟩
  while (auxL != NULL) {
    anyadirMonomio(ultimoNueva,auxL->monomio);
    auxL = auxL->siguiente;
  }
  ⟨Devolver el primer nodo de la lista generada⟩
}
```

Han aparecido dos nuevos problemas, ya menores: en primer lugar, hemos de dar un valor inicial a la variable `ultimoNueva` y, por otro lado, debemos devolver un puntero a la primera celda generada. Para atender al primer asunto creamos una cabecera y hacemos que `ultimoNueva` apunte a esa cabecera:

```
Polinomio nueva = ⟨polinomio nulo⟩;
CeldaMonomio* ultimoNueva = nueva;
```

Además, así resolvemos el problema de ⟨*Devolver el primer nodo...*⟩ puesto que éste es el nodo siguiente de la cabecera. Por último, debemos eliminar de la memoria la cabecera.

```
CeldaMonomio* aux = nueva->siguiente;
delete nueva;
return aux;
```

Ahora, la generación de un ⟨*polinomio nulo*⟩ se puede encapsular en una función que construya, sencillamente, un polinomio nulo, es decir, una lista con sólo la cabecera:

```
Polinomio polinomioCero() {
  Polinomio polinomio = new CeldaMonomio;
  polinomio->monomio.grado = -1;
  polinomio->monomio.coeficiente = fraccionCero;
  polinomio->siguiente = NULL;
  return polinomio;
}
```

La llamada sería `polinomioCero()`. Obsérvese que estamos usando una constante de tipo `Fraccion` llamada `fraccionCero`.

En cuanto a la adición de un monomio indicada como `anyadirMonomio(...)`, se trata de añadir una celda, de contenido el monomio dado, y mover el puntero a esa nueva celda. Hay que tener cuidado de no introducir monomios con coeficiente 0, que pueden surgir durante la suma:

```
void anyadirMonomio(CeldaMonomio*& nodo, Monomio monomio) {
  if (monomio.coeficiente != fraccionCero) {
    CeldaMonomio* actual = new CeldaMonomio;
    actual->monomio = monomio;
    actual->siguiente = NULL;
    nodo->siguiente = actual;
    nodo = actual;
  }
}
```

Y también ha aparecido una función para sumar monomios. Éstos sólo se pueden sumar cuando tienen el mismo grado; entonces, basta con sumar los coeficientes, y el grado es el mismo:

```
Monomio operator+(Monomio const& a, Monomio const& b) {
  Monomio resultado;
  resultado.coeficiente = a.coeficiente + b.coeficiente;
  resultado.grado = a.grado;
  return resultado;
}
```

La resta se resuelve fácilmente usando la suma: sólo hay que cambiar los monomios del segundo operando e invocar a la suma. Debemos tener cuidado de no estropear ningún polinomio de los de entrada, por lo que la operación de cambio de signo deberá generar un polinomio nuevo, auxiliar, que a la postre deberemos borrar antes de abandonar la función:

```
void restar(Polinomio const a, Polinomio const b, Polinomio& resultado) {
  Polinomio aux;
  cambiarSignoPolinomio(b, aux);
  sumar(a, aux, resultado);
  liberar(aux);
}
```

El procedimiento `cambiarSignoPolinomio` devuelve una copia del primer polinomio en el segundo. Puesto que es similar a `copiarListaMonomios`, se omite aquí su implementación.

***Multiplicación de polinomios*** Para realizar la multiplicación usaremos la propiedad distributiva, que coincide con el algoritmo habitual:

$$p(x)(a_0 + a_1x + \cdots + a_nx^n) = p(x)a_0 + p(x)a_1x + \cdots + p(x)a_nx^n$$

Por tanto recorremos los monomios del segundo polinomio, el multiplicador, los vamos multiplicando por el primero y acumulando la suma en `resultado`.

```
void multiplicar(Polinomio const a, Polinomio const b, Polinomio& resultado) {
  resultado = polinomioCero();
  CeldaMonomio* auxB = b->siguiente;
  while (auxB != NULL) {
    ⟨Acumular en resultado el producto del monomio auxB por el polinomio a⟩
    auxB = auxB->siguiente;
  }
}
```

A la hora de ⟨*Acumular en* `resultado`...⟩ habría que realizar una instrucción como ésta:

```
resultado = resultado + a*auxB->monomio;
```

Pero debido a nuestra decisión de implementar las operaciones como procedimientos (para bien o para mal) nos vemos obligados a guardar los resultados intermedios en variables auxiliares (ése es el inconveniente).

```
Polinomio aux1;
multiplicar(a, auxB->monomio, aux1);
Polinomio aux2 = resultado;
sumar(aux1, aux2, resultado);
```

Pero nos damos cuenta (y ésta es la ventaja) de que hemos creado dos nuevos polinomios que no se vuelven a usar, y por tanto es justo y necesario y es nuestro deber liberar su memoria:

```
liberar(aux1);
liberar(aux2);
```

A la hora de realizar esta operación ha aparecido una función para multiplicar un monomio por un polinomio, y el resultado debe ser un polinomio nuevo. Para ello, recorremos el polinomio con un puntero auxiliar `auxP` y vamos añadiendo a un polinomio nuevo el resultado de multiplicar el monomio de entrada del procedimiento por el monomio al que apunte `auxP`. Para poder añadir el monomio resultante al polinomio resultado necesitamos un puntero al último monomio del polinomio `resultado`:

```
void multiplicar(Polinomio const polinomio, Monomio const& monomio,
                 Polinomio &resultado) {
  resultado = polinomioCero();
  CeldaMonomio* antR = resultado;
  CeldaMonomio* auxP = polinomio->siguiente;
  while (auxP != NULL) {
    anyadirMonomio(antR, auxP->monomio * monomio);
    auxP = auxP->siguiente;
  }
}
```

La multiplicación de monomios sigue una regla muy sencilla: se multiplican los coeficientes y se suman los exponentes. Esto es:

```
Monomio operator*(Monomio const& a, Monomio const& b) {
  Monomio resultado;
  resultado.coeficiente = a.coeficiente * b.coeficiente;
  resultado.grado = a.grado + b.grado;
  return resultado;
}
```

***División de polinomios*** Realizaremos el algoritmo de división aprendido en la escuela:

$$
\begin{array}{rrrrr|l}
x^4 & +0x^3 & +0x^2 & +0x & +1 & x-1 \\
\cline{6-6}
-x^4 & +x^3 & & & & x^3+x^2+x+1 \\
\cline{1-2}
 & x^3 & +0x^2 & +0x & +1 & \\
 & -x^3 & +x^2 & & & \\
\cline{2-3}
 & & x^2 & +0x & +1 & \\
 & & -x^2 & +x & & \\
\cline{3-4}
 & & & x & +1 & \\
 & & & -x & +1 & \\
\cline{4-5}
 & & & & 2 &
\end{array}
$$

(Para que se entienda mejor cómo funciona nuestro programa, en cada nivel bajamos todo el polinomio restante en vez del término siguiente.)

Empezamos asignando a un polinomio `resto` el dividendo, siendo el polinomio `cociente` el polinomio 0. Tras añadir cada nuevo monomio en el divisor, se verificará la regla de oro:

$$\texttt{dividendo} = \texttt{cociente} \cdot \texttt{divisor} + \texttt{resto}$$

Acabaremos cuando el grado del `resto` sea menor que el del `divisor`. Nosotros consideramos definida la división cuando el grado del polinomio divisor es mayor que cero (no es un polinomio constante). Hemos de devolver dos polinomios: el cociente y el resto.

El algoritmo clásico para dividir los polinomios consiste en lo siguiente:

```
void dividir(Polinomio const dividendo, Polinomio const divisor,
             Polinomio& cociente, Polinomio& resto) {
  cociente = polinomioCero();
  resto = ⟨dividendo⟩
  while (grado(resto) >= grado(divisor)) {
    Monomio actual = ⟨Dividir los monomios de mayor grado del resto y el divisor⟩
    ⟨Añadir actual a cociente⟩
    ⟨Restar a resto el producto del monomio actual por el polinomio divisor⟩
  }
}
```

A continuación iremos refinando poco a poco este procedimiento. En primer lugar debemos definir la función `grado`, cuya implementación es sencilla y se deja al lector.

Para asignar el dividendo al resto podríamos hacer `resto = dividendo`; pero, haciéndolo, `resto` y `dividendo` compartirían memoria y los cambios que tuvieran lugar sobre el `resto`, tendrían también efecto sobre el `dividendo`. Necesitamos pues una copia del `dividendo`:

```
resto = copiarPolinomio(dividendo);
```

Para dividir los monomios de mayor grado de `resto` y `divisor` necesitamos un procedimiento adicional que divida monomios, y puesto que la lista de monomios de cada polinomio está ordenada de forma decreciente, los monomios de mayor grado son los que están al principio. Por tanto el monomio `actual` se obtiene de la siguiente forma:

```
Monomio actual = resto->siguiente->monomio / divisor->siguiente->monomio;
```

Necesitamos añadir el monomio `actual` al `cociente`, y para ello es conveniente tener un puntero al

último nodo del mismo, al que llamaremos `ultimoCociente`. El valor inicial de este puntero debe ser la cabecera del cociente; por tanto antes del bucle deberemos declarar esta variable de la siguiente forma,

```
CeldaMonomio* ultimoCociente = cociente;
```

y para añadir el monomio `actual` al cociente y actualizar `ultimoCociente` podemos proceder así:

```
anyadirMonomio(ultimoCociente, actual);
```

Para refinar la última parte del bucle deberíamos hacer una instrucción como ésta,

```
resto = resto - divisor*actual;
```

pero al haber realizado las operaciones como procedimientos debemos realizar dicha expresión por pasos y liberar la memoria de los polinomios auxiliares que calculemos:

```
Polinomio aux1, aux2;
multiplicar(divisor, actual, aux1);
restar(resto, aux1, aux2);
liberar(resto);
liberar(aux1);
resto = aux2;
```

Por último falta la función para dividir monomios: ahora la regla es dividir los coeficientes y restar los exponentes. Fijémonos en que, al no admitir monomios con coeficientes a cero, siempre se pueden dividir los monomios:

```
Monomio operator/(Monomio const& a, Monomio const& b) {
  Monomio resultado;
  resultado.coeficiente = a.coeficiente / b.coeficiente;
  resultado.grado = a.grado - b.grado;
  return resultado;
}
```

## 6.11 Máximos y mínimos

309

Se van a dar dos soluciones, una siguiendo literalmente el procedimiento descrito y otra, más eficiente, basada en una interesante observación. Ambas soluciones operan con una lista de listas de enteros, que se define como sigue:

```
typedef struct NodoEntero {
  int inf;
  NodoEntero* sig;
} NodoEntero;
typedef NodoEntero* ListaEnteros;
typedef struct NodoLista {
  ListaEnteros inf;
  NodoLista* sig;
} NodoLista;
typedef NodoLista* ListaListas;
```

No debemos olvidar los procedimientos para liberar la memoria dinámica: escribiremos uno para las listas de enteros y otro para las listas de listas:

```
void liberar(ListaEnteros& lista) {
  NodoEntero* actual = lista;
  while (actual != NULL) {
    NodoEntero* aux = actual;
    actual = actual->sig;
    delete aux;
  }
  lista = NULL;
}
void liberar(ListaListas& lista) {
  NodoLista* actual = lista;
  while (actual != NULL) {
    NodoLista* aux = actual;
    actual = actual->sig;
    liberar(aux->inf);
    delete aux;
  }
  lista = NULL;
}
```

### 6.11.1 Solución primera, ineficiente

Una primera solución consiste sencillamente en seguir literalmente el método descrito: recorrer completamente todas las secuencias hallando sus máximos, y seleccionar el mínimo entre ellos.

```
⟨Se parte de un mínimo ficticio, minimo = "infinito"⟩
while (⟨queden listas⟩) {
  ⟨Hallar el elemento máximo de una lista, maximo⟩
  if (⟨maximo < minimo⟩) {
    ⟨se sustituye el valor de minimo por el de maximo⟩
  }
}
```

El procedimiento se puede expresar con pocos detalles técnicos:

```
int maxMin(ListaListas const lista) {
  int minimo = INT_MAX;
  ListaListas auxLista = lista;
  while (auxLista != NULL) {
    int maximo = maximoLista(auxLista->inf);
    if (maximo < minimo) minimo = maximo;
    auxLista = auxLista->sig;
  }
  return minimo;
}
```

Finalmente, para terminar, queda por describir la función `maximoValor`, que halla el máximo elemento de una lista de enteros:

```
int maximoLista(ListaEnteros const lista) {
  int maximo = INT_MIN;
  ListaEnteros auxLista = lista;
  while (auxLista != NULL) {
    if (auxLista->inf > maximo) maximo = auxLista->inf;
    auxLista = auxLista->sig;
  }
  return maximo;
}
```

### 6.11.2 Solución eficiente, truncando la búsqueda

La solución dada puede mejorarse si se hace una observación algo sutil: con el ejemplo anterior, el máximo elemento de la primera secuencia es 58; o sea, que al repasar la segunda secuencia buscando el máximo, cuando se supere (o se iguale) el 58 (al encontrar el 71) no hará falta seguir, porque esta cantidad (u otra superior, como el 81, en el resto de esa secuencia) ya no podrá mejorar a 58 como mínimo de los máximos.

Este hecho permite mejorar el programa cambiando la función `maximoValor` (de una lista) por otra que, dada una `lista` de enteros y un entero (`z`), halla directamente *el mínimo entre el entero z y el máximo entero de la lista*, interrumpiendo la búsqueda cuando se sabe que va a ser infructuosa:

```
int minEntMaxLista(int const z, ListaEnteros const lista) {
  int maximo = INT_MIN;
  ListaEnteros auxLista = lista;
  while (auxLista != NULL && maximo < z) {
    if (auxLista->inf > maximo) maximo = auxLista->inf;
    auxLista = auxLista->sig;
  }
  return maximo < z ?  maximo :  z;
}
```

La mejora en la eficiencia se produce sustituyendo, en la función `maxMin`, la llamada

```
maximoValor(auxLista->inf);
```

por esta otra:

```
minEntMaxLista(minimo, auxLista->inf);
```

Por otra parte, es sencillo comprobar que el resultado de la función anterior es siempre menor o igual que el entero $z$ de partida, con lo que sobra la comparación posterior:

```
if (maxFila < minimo) ⟨...⟩
```

De esta forma, el bucle `while` se simplifica del siguiente modo,

```
while (auxLista != NULL) {
  minimo = minEntMaxLista(minimo, auxLista->inf);
  auxLista = auxLista->sig;
}
```

y la función `maxMin` queda finalmente así:

```
int maxMin(ListaListas const lista) {
  int minimo = INT_MAX;
  ListaListas auxLista = lista;
  while (auxLista != NULL) {
    minimo = minEntMaxLista(minimo, auxLista->inf);
    auxLista = auxLista->sig;
  }
  return minimo;
}
```

## 6 18 Baldosas de jardín

321 324

### 6.18.1 Representación de datos

Para realizar los diseños se necesita tener un tablero. Llamaremos casilla a cada una de las unidades que componen el tablero de diseño. Una baldosa negra ocupa una casilla, y una blanca dos. Definiremos un tablero de dimensión variable para poder realizar cualquier diseño:

```
typedef struct Posicion {
  int fila;
  int columna;
} Posicion;
typedef struct Tablero {
  Casilla** contenido;
  Posicion maxPosicion;
} Tablero;
```

Como estamos usando arrays de dimensión dinámica, debemos utilizar un procedimiento para liberar la memoria ocupada por un tablero cuando ya no sea necesario:

```
void libera(Tablero& tablero) {
  for (int i = 0; i < tablero.maxPosicion.fila; i++) {
    delete [] tablero.contenido[i];
  }
  delete [] tablero.contenido;
  tablero.contenido = NULL;
}
```

Las casillas pueden clasificarse en aquéllas ocupadas por baldosas negras, aquéllas que se desea cubrir con baldosas blancas y aquéllas que no pertenecen al diseño que deseamos realizar (por ejemplo si el jardín tiene forma triangular). Además, para definir las casillas se tienen que haber considerado las operaciones que deseamos realizar sobre el tablero. La comprobación de si un hueco de casillas se puede cubrir con baldosas blancas, como veremos más adelante, no requiere más información que la posición de cada una de las casillas que se desean cubrir con baldosas blancas; por el contrario, para detectar los diferentes huecos que se deben rellenar con baldosas blancas en un diseño, se necesita recorrer el tablero y *marcar* las casillas blancas ya visitadas. Con todo esto se define la siguiente enumeración para describir las casillas:

```
typedef enum Casilla {
  negra, blanca, noPertenece, marcada
} Casilla;
```

Las operaciones habituales de entrada y salida para los tipos `Casilla`, `Posicion` y `Tablero` se recogen en el siguiente código:

```
char const casillaBlanca = 'B';
char const casillaNegra = 'N';
char const casillaNoPertenece = '#';
char const casillaMarcada = 'b';
Casilla valorCasilla(char const c) {
  switch(c) {
  case casillaBlanca:
    return blanca;
  case casillaMarcada:
    return marcada;
  case casillaNegra:
    return negra;
  case casillaNoPertenece:
    return noPertenece;
  }
}
char casillaCaracter(Casilla const casilla) {
  char const nombreCasilla[] = {'N', 'B', '#', 'b'};
  return nombreCasilla[casilla];
}
istream& operator>>(istream& in, Posicion& posicion) {
  in >> posicion.fila;
  in >> posicion.columna;
  return in;
}
ostream& operator<<(ostream& out, Posicion const& posicion) {
  out << "(" << posicion.fila << "," << posicion.columna << ")";
  return out;
}
istream& operator>>(istream& in, Tablero& tablero) {
  in >> tablero.maxPosicion;
  tablero.contenido = new Casilla*[tablero.maxPosicion.fila];
  for (int i = 0; i < tablero.maxPosicion.fila; i++) {
    tablero.contenido[i] = new Casilla[tablero.maxPosicion.columna];
    for (int j = 0; j < tablero.maxPosicion.columna; j++) {
      char c;
      in >> c;
      tablero.contenido[i][j] = valorCasilla(c);
    }
  }
  return in;
}
```

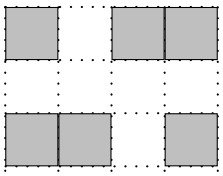
```
void muestraBorde(int const ancho, ostream& out) {
  out << "|";
  for (int j = 0; j < ancho; j++) {
    out << "-";
  }
  out << "|" << endl;
}
ostream& operator<<(ostream& out, Tablero const& tablero) {
  muestraBorde(tablero.maxPosicion.columna, out);
  for (int i = 0; i < tablero.maxPosicion.fila; i++) {
    out << "|";
    for (int j = 0; j < tablero.maxPosicion.columna; j++) {
      out << casillaCaracter(tablero.contenido[i][j]);
    }
    out << "|" << endl;
  }
  muestraBorde(tablero.maxPosicion.columna, out);
  return out;
}
```

### 6.18 2 Comprobación para los huecos

Como se verá en la solución del apartado 6.18b, encontrar una colocación adecuada de las baldosas blancas para un determinado diseño es una tarea costosa. Además, como se ha visto en algunos ejemplos, puede que no siempre exista dicha colocación. Quizás, se pueden determinar, con poco esfuerzo, algunos diseños que no se pueden rellenar.

En efecto, como apunta la pista 6.18a, si realizamos los diseños sobre tableros de ajedrez se puede comprobar que toda baldosa blanca abarca un escaque negro y otro blanco. Por tanto, todo hueco que se pueda cubrir con baldosas blancas tiene que tener el mismo número de escaques blancos y negros.

Esta propiedad es necesaria, pero no suficiente. Es decir, ningún hueco que no tenga el mismo número de escaques blancos y negros puede cubrirse con baldosas blancas, pero puede haber huecos con el mismo número de escaques blancos y negros que no pueden rellenarse con baldosas blancas, por ejemplo:



Añadimos la definición del tipo `Componente` para describir un hueco de casillas o, dicho con más propiedad, una *componente conexa* de casillas que no están ocupadas por baldosas negras.

```
typedef struct Componente {
  bool** contenido;
  Posicion maxPosicion;
} Componente;
```

Los procedimientos correspondientes para liberar memoria y escribir `Componente`s son éstos:

```
void libera(Componente& componente) {
  for (int i = 0; i < componente.maxPosicion.fila; i++) {
    delete [] componente.contenido[i];
  }
  delete [] componente.contenido;
  componente.contenido = NULL;
}
ostream& operator<<(ostream& out, Componente const& componente) {
  muestraBorde(componente.maxPosicion.columna, out);
  for (int i = 0; i < componente.maxPosicion.fila; i++) {
    out << "|";
    for (int j = 0; j < componente.maxPosicion.columna; j++) {
      if (componente.contenido[i][j]) out << "B";
      else out << " ";
    }
    out << "|" << endl;
  }
  muestraBorde(componente.maxPosicion.columna, out);
  return out;
}
```

Afortunadamente, comprobar que un hueco tiene el mismo número de casillas que corresponden a escaques blancos y negros se traduce en una propiedad muy sencilla: un hueco rodeado por baldosas negras no se podrá cubrir con baldosas blancas si no tiene la misma cantidad de casillas *pares* que *impares* (o sea, cuyas coordenadas dan una suma par o impar respectivamente). El código para comprobar que una componente no es rellenable es el siguiente:

```
bool noRellenable(Componente const& componente) {
  int sumaPar = 0, sumaImpar = 0;
  for (int i = 0; i < componente.maxPosicion.fila; i++) {
    for (int j = 0; j < componente.maxPosicion.columna; j++) {
      if (componente.contenido[i][j]) {
        if ((i+j) % 2 == 0) sumaPar++;
        else sumaImpar++;
      }
    }
  }
  return sumaPar != sumaImpar;
}
```

### 6.18.3 Búsqueda de huecos

Aunque cualquiera puede hacerlo sobre el papel, la búsqueda de los diferentes huecos dejados por un diseño no es una tarea sencilla.

Partiendo de una casilla identificada como blanca, hay que encontrar todas las casillas blancas a las que *se puede ir* desde la primera. Una solución posible es definir un subprograma recursivo que, considerando una determinada casilla blanca de origen, añade a una componente todas las casillas blancas que son *vecinas* de la casilla origen; con cada una de las casillas vecinas se repite el proceso.

Esta idea necesita identificar las casillas que ya han sido consideradas en la búsqueda para no entrar

en ciclos. Para eso se utiliza el valor `marcada` definido en la enumeración `Casilla`. El subprograma que busca una componente en un tablero a partir de una posición es `buscaComponente`. El subprograma `esBlanca` recoge todos los detalles relativos a los límites del tablero:

```
bool esBlanca(Tablero const& tablero, Posicion const& posicion) {
  return posicion.fila >= 0 && posicion.fila < tablero.maxPosicion.fila &&
    posicion.columna >= 0 &&  posicion.columna < tablero.maxPosicion.columna &&
    tablero.contenido[posicion.fila][posicion.columna] == blanca;
}
void buscaComponente(Tablero& tablero, Componente& componente,
                     Posicion const& posicion) {
  if (esBlanca(tablero, posicion)) {
    tablero.contenido[posicion.fila][posicion.columna] = marcada;
    insertaCasillaComponente(componente, posicion);
    buscaComponente(tablero, componente,
                    (Posicion) {posicion.fila+1, posicion.columna});
    buscaComponente(tablero, componente,
                    (Posicion) {posicion.fila, posicion.columna+1});
    buscaComponente(tablero, componente,
                    (Posicion) {posicion.fila-1, posicion.columna});
    buscaComponente(tablero, componente,
                    (Posicion) {posicion.fila, posicion.columna-1});
  }
}
```

Para manejar los componentes de forma abstracta y modular, añadimos dos operaciones más, una para creación y otra para inserción.

```
Componente componenteVacio(Posicion const maxPosicion) {
  Componente resultado;
  resultado.maxPosicion = maxPosicion;
  resultado.contenido = new bool*[maxPosicion.fila];
  for (int i = 0; i < maxPosicion.fila; i++) {
    resultado.contenido[i] = new bool[maxPosicion.columna];
    for (int j = 0; j < maxPosicion.columna; j++) {
      resultado.contenido[i][j] = false;
    }
  }
  return resultado;
}
void insertaCasillaComponente(Componente& componente, Posicion const& posicion) {
  componente.contenido[posicion.fila][posicion.columna] = true;
}
```

### 6.18 4 Resolución

No queda ya mucho por hacer. De los apartados anteriores tenemos subprogramas que permiten buscar componentes conexos de casillas blancas y que indican si no son rellenables. Para comprobar que el diseño total sobre un tablero se puede rellenar, tenemos que buscar todas los componentes que deben rellenarse con baldosas blancas. Para ello, se busca tantas veces como sea necesario una casilla blanca y se calcula su componente conexo:

```
Posicion buscaCasillaBlanca(Tablero const& tablero) {
  Posicion resultado;
  resultado.fila = -1; resultado.columna = -1;
  for (int i = 0; i < tablero.maxPosicion.fila; i++) {
    for (int j = 0; j < tablero.maxPosicion.columna; j++) {
      if (tablero.contenido[i][j] == blanca) {
        resultado.fila = i; resultado.columna = j;
        return resultado;
      }
    }
  }
  return resultado;
}
void calculaComponente(Tablero& tablero, Componente& componente,
                       Posicion const& posicion) {
  componente = componenteVacio(tablero.maxPosicion);
  buscaComponente(tablero, componente, posicion);
}
```

Un diseño sobre un tablero no será rellenable si alguno de sus huecos no es rellenable.

```
bool noRellenable(Tablero const& tablero) {
  Tablero tableroAux = copiaTablero(tablero);
  bool imposible = false;
  Posicion posicion = buscaCasillaBlanca(tableroAux);
  while (!imposible && posicion.fila >= 0) {
    Componente componente;
    calculaComponente(tableroAux, componente, posicion);
    imposible = noRellenable(componente);
    posicion = buscaCasillaBlanca(tableroAux);
    libera(componente);
  }
  libera(tableroAux);
  return imposible;
}
```

Puesto que hemos marcado el parámetro `tablero` como `const&`, necesitamos hacer copias del tablero cuando tengamos que hacer modificaciones de su estado, como ocurre en `calculaComponente`.

```
Tablero copiaTablero(Tablero const& tablero) {
  Tablero copia;
  copia.maxPosicion = tablero.maxPosicion;
  copia.contenido = new Casilla*[tablero.maxPosicion.fila];
  for (int i = 0; i < tablero.maxPosicion.fila; i++) {
    copia.contenido[i] = new Casilla[tablero.maxPosicion.columna];
    for (int j = 0; j < tablero.maxPosicion.columna; j++) {
      copia.contenido[i][j] = tablero.contenido[i][j];
    }
  }
  return copia;
}
```

Para poder mostrar una solución necesitamos definir el tipo adecuado y operaciones para manejarla. En este caso utilizamos una matriz de caracteres y la secuencia "`[]`" para marcar baldosas blancas colocadas en horizontal y "$\begin{smallmatrix}\texttt{n}\\\texttt{u}\end{smallmatrix}$" para marcar las colocadas en vertical. Por ejemplo, observa la siguiente solución generada a un diseño:

```
|----|
|NNNn|
|N[]u|
|n[]N|
|uNNN|
|----|
```

Los guiones "`-`" y las barras "`|`" delimitan el tablero, la letra "`N`" indica la colocación de baldosas negras. Hay dos baldosas blancas colocadas en horizontal (centro del diseño) y dos baldosas blancas colocadas en vertical, una en el extremo superior derecho y otra en el extremo inferior izquierdo.

El código que define el tipo `Solucion` es:

```
typedef struct Solucion {
  char** contenido;
  Posicion maxPosicion;
} Solucion;
```

Su correspondiente procedimiento para liberar la memoria es:

```
void libera(Solucion& solucion) {
  for (int i = 0; i < solucion.maxPosicion.fila; i++) {
    delete [] solucion.contenido[i];
  }
  delete [] solucion.contenido;
  solucion.contenido = NULL;
}
```

Y su procedimiento para mostrar los datos es:

```
ostream& operator<<(ostream& out, Solucion const& solucion) {
  muestraBorde(solucion.maxPosicion.columna, out);
  for (int i = 0; i < solucion.maxPosicion.fila; i++) {
    out << "|";
    for (int j = 0; j < solucion.maxPosicion.columna; j++) {
      out << solucion.contenido[i][j];
    }
    out << "|" << endl;
  }
  muestraBorde(solucion.maxPosicion.columna, out);
  return out;
}
```

El siguiente procedimiento genera una solución vacía sobre la que se irán añadiendo las baldosas:

```
Solucion solucionVacia(Posicion const& maxPosicion) {
  Solucion resultado;
  resultado.maxPosicion = maxPosicion;
  resultado.contenido = new char*[maxPosicion.fila];
  for (int i = 0; i < maxPosicion.fila; i++) {
    resultado.contenido[i] = new char[maxPosicion.columna];
    for (int j = 0; j < maxPosicion.columna; j++) {
      resultado.contenido[i][j] = ' ';
    }
  }
  return resultado;
}
```

El algoritmo para dar una solución en un tablero potencialmente rellenable es el siguiente: se coloca en el tablero una baldosa blanca, las dos casillas que se ocupan se consideran fuera del diseño, se comprueba si existe solución para el resto del tablero. Si es así la baldosa se mantiene y si no se cambia de orientación. Si no es posible encontrar solución con ninguna de las posiciones posibles de la baldosa blanca, es que el tablero no es rellenable y el programa termina. Esta descripción es recursiva y tiene un alto coste. Por ello llamaremos a la función `noRellenable` para filtrar diseños no rellenables. Usaremos el siguiente procedimiento:

```
void calculaSolucion(Tablero const& tablero, bool& encontrada, Solucion& solucion) {
  Tablero aux = copiaTablero(tablero);
  solucion = solucionVacia(tablero.maxPosicion);
  Posicion posicion={0,0};
  encontrada = false;
  if (!noRellenable(tablero)) {
    calculaSolucion(aux, encontrada, solucion, posicion);
  }
  libera(aux);
}
```

El algoritmo recursivo queda como sigue:

```
void calculaSolucion(Tablero& tablero, bool& encontrada,
                     Solucion& solucion, Posicion const& posicion) {
  if (!estaDentro(tablero, posicion)) {
    encontrada = true;
  } else {
    switch(tablero.contenido[posicion.fila][posicion.columna]) {
    case blanca:
      tablero.contenido[posicion.fila][posicion.columna] = noPertenece;
      if (esBlanca(tablero, (Posicion){posicion.fila, posicion.columna+1})) {
        tablero.contenido[posicion.fila][posicion.columna+1] = noPertenece;
        solucion.contenido[posicion.fila][posicion.columna] = '[';
          solucion.contenido[posicion.fila][posicion.columna+1] = ']';
          calculaSolucion(tablero, encontrada, solucion, siguiente(tablero, posicion));
      }
      if (!encontrada &&
          esBlanca(tablero, (Posicion){posicion.fila+1, posicion.columna})) {
```

```
        tablero.contenido[posicion.fila+1][posicion.columna] = noPertenece;
        solucion.contenido[posicion.fila][posicion.columna] = 'n';
        solucion.contenido[posicion.fila+1][posicion.columna] = 'u';
        calculaSolucion(tablero, encontrada, solucion, siguiente(tablero, posicion));
      }
      break;
    case negra:
      solucion.contenido[posicion.fila][posicion.columna] = casillaNegra;
      calculaSolucion(tablero, encontrada, solucion, siguiente(tablero, posicion));
      break;
    case noPertenece:
      if (solucion.contenido[posicion.fila][posicion.columna]==' ') {
        solucion.contenido[posicion.fila][posicion.columna] = casillaNoPertenece;
      }
      calculaSolucion(tablero, encontrada, solucion, siguiente(tablero, posicion));
      break;
    }
  }
}
```

Puesto que estamos calificando de `noPertenece` a las casillas que ocupamos con baldosas blancas, necesitamos diferenciar éstas de aquellas casillas que no pertenecen al diseño a la hora de mostrar la solución. Por este detalle, dentro del caso `case noPertenece` tenemos que preguntar si el contenido es *vacío*,

```
if (solucion.contenido[posicion.fila][posicion.columna]==' '
```

para asegurarnos de que se trata de una casilla que `noPertenece` al diseño, y marcarla como tal en la solución.

Para terminar con el cálculo de la solución, detallamos los siguientes subprogramas que son utilizados por el recursivo `calculaSolucion`:

```
Posicion siguiente(Tablero const& tablero, Posicion const& posicion) {
  Posicion resultado = posicion;
  resultado.columna++;
  if (resultado.columna >= tablero.maxPosicion.columna) {
    resultado.columna = 0;
    resultado.fila++;
  }
  return resultado;
}
bool estaDentro(Tablero const& tablero, Posicion const& posicion) {
  return posicion.fila >=0 && posicion.columna >=0 &&
    posicion.fila < tablero.maxPosicion.fila &&
    posicion.columna < tablero.maxPosicion.columna;
}
```

Finalmente, para tener el programa completo basta con incluir las llamadas a las bibliotecas externas `iostream` y `fstream`, reordenar de forma adecuada las definiciones de tipos y subprogramas que aparecen a lo largo de la solución y definir la función `main`:

```
int main(int argc, char* argv[]) {
  ifstream entrada(argv[1]);
  Tablero tablero;
  entrada >> tablero;
  cout << tablero;
  bool existe;
  Solucion solucion;
  calculaSolucion(tablero, existe, solucion);
  if (existe) {
    cout << "Se puede:" << endl;
    cout << solucion;
  } else {
    cout << "No se puede" << endl;
  }
}
```

En este caso hemos optado por cargar el diseño del tablero de un fichero externo que se pasa como parámetro. Dicho fichero contiene en la primera línea las dimensiones del tablero y en las siguientes una descripción del diseño, utilizando "`#`" para las casillas que no pertenecen al diseño, "`N`" para las ocupadas por baldosas negras, y "`B`" para las que deseamos ocupar con baldosas blancas. Carmencita cumplió con su palabra.

[AF00] Marcelo Alonso y Edward J. Finn. *Física*. Addison-Wesley (2000).

[AH01] Jörg Ardnt y Chistoph Haenel. *Pi Unleashed*. Springer (2001).

[AHU98] A. V. Aho, J. E. Hopcroft y J. D. Ullman. *Estructura de datos y algoritmos*. Addison-Wesley (1998).

[AR95] Owen Astrachan y David Reed. "The Applied Apprenticeship Approach to CS1". *SIGCSE Bulletin* **27**(1), 1–5 (1995).

[Bal02] Philip Ball. *Bright Earth: Art and the Invention of Color*. Farrar, Straus & Giroux (2002).

[BB96] Gilles Brassard y Paul Bratley. *Fundamentals of algorithmics*. Prentice-Hall (1996).

[BB00] Gilles Brassard y Paul Bratley. *Fundamentos de algoritmia*. Prentice-Hall (2000).

[Bec82] Petr Beckmann. {*A history of*} $\pi$. Golem Press (1982).

[Ben84] John Bentley. "Programming Pearls". *Communications of the ACM* (enero 1984).

[BH87] R. S. Bird y J. Hughes. "The Alpha-Beta Algorithm: An Exercise in Program Transformation". En *Information Processing Letters*, volumen 24, páginas 53–57. North-Holland (1987). International Summer School on Constructive Methods in Computer Science. Marktoberdorf, Germany.

[BHvW00] Mark Bruls, Kees Huizing y Jarke J. van Wijk. "Squarified Treemaps". En *Proccedings of Joint Eurographics and IEEE TCVG Symposium on Visualization*, páginas 33–42 (2000).

[Bir86] R. Bird. "Tabulation techniques for recursive programs". *ACM Computing Surveys* **12**(4), 403–417 (diciembre 1986).

[Cha99] Jean-Luc Chabert, editor. *A History of Algorithms: From the Pebble to the Microchip*. Springer (1999).

[Dew85a] A. K. Dewdney. "Cinco piezas sencillas para bucle y generador de números aleatorios". *Investigación y Ciencia* **105**, 94–99 (junio 1985).

[Dew85b] A. K. Dewdney. "Yin y Yang: recurrencia o iteración, la torre de Hanoi y las argollas chinas". *Investigación y Ciencia* **100**, 102–107 (enero 1985).

[Dew86] A. K. Dewney. "Búsqueda de una regla invisible que ayude a los radioastrónomos en la medición de nuestro planeta". *Investigación y Ciencia* **113**, 104–108 (febrero 1986).

[dG84] Miguel de Guzmán. "Juegos matemáticos en la enseñanza". En *Actas de las IV Jornadas sobre Aprendizaje y Enseñanza de las Matemáticas*, Santa Cruz de Tenerife (septiembre 1984). Sociedad Canaria de Profesores de Matemáticas Isaac Newton. Accesible en `http://www.mat.ucm.es/deptos/am/guzman/juemat/juemat.htm`.

[Dic45] L. R. Dice. "Measures of the amount of ecologic association between species". *Ecology* **26**, 297–302 (1945).

[Dox00] Apostolos Doxiadis. *El tío Petros y la conjetura de Goldbach.* Ediciones B (2000).

[dP63] Giovanni Battista della Porta. *De Furtivis Literarum Notis.* Libri Naples (1563).

[dPC00] Dionisio de Pedro Cursá. *Teoría completa de la música.* Real Musical (2000). dos volúmenes.

[dvOS00] Mark de Berg, Marc van Kreveld, Mark Overmars y Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer Verlag (mayo 2000).

[Eli75] Peter Elias. "Universal Codeword Sets and Representations of the Integers". *IEEE Transactions on Information Theory* **21**(2), 194–203 (marzo 1975).

[Eli93] Norbert Elias. *El proceso de la civilización: investigaciones sociogenéticas y psicogenéticas.* Fondo de Cultura Económica (1993).

[Enz01] Hans Magnus Enzensbergen. *El diablo de los números: un libro para todos aquéllos que temen a las matemáticas.* Siruela (2001).

[Euc91] Euclides. *Elementos. Libros I-IV.* Gredos (1991).

[Euc94] Euclides. *Elementos. Libros V-IX.* Gredos (1994).

[Euc96] Euclides. *Elementos. Libros X-XIII.* Gredos (1996).

[Fen96] Peter Fenwick. "Punctured Elias codes for variable-length coding of the integers". Informe técnico, University of Auckland (1996).

[Fey98] Richard P. Feynman. *Física.* Addison-Wesley (1998).

[Fri98] Jeffrey E. F. Friedl. *Mastering Regular Expressions. Powerful Techniques for Perl and other Tools.* O'Reilly (1998).

[FvDFH97] James D. Foley, Andries van Dam, Steven K. Feiner y John F. Hughes. *Computer Graphics. Principles and Practice.* Addison-Wesley (1997).

[Gar83] Martin Gardner. "Tareas que es forzoso concluir por mucho que se quiera evitarlo.". *Investigación y Ciencia* **85**, 100–105 (octubre 1983).

[Gar86] Martin Gardner. *Festival mágico-matemático.* Alianza (1986).

[Gar87] Martin Gardner. *Carnaval matemático.* Alianza Editorial, S. A (1987).

[Gar88] Martin Gardner. *Ruedas, vida y otras diversiones matemáticas.* Labor (1988).

[Gar94] Martin Gardner. *Nuevos pasatiempos matemáticos.* Alianza (1994).

[Gar95] Martin Gardner. *¡Ajá! Inspiración ¡ajá!.* Labor (1995).

[Góm99] Víctor Gómez Pin. *La tentación pitagórica: ambición filosófica y anclaje matemático.* Síntesis (1999).

[Gol91] D. Goldberg. "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys* **23**(1), 5–48 (1991).

[Gru84] Fred Gruenberger. "De cómo manejar números de miles de cifras y de por qué nos es necesario". *Investigación y Ciencia* **93**, 110–115 (junio 1984).

[Gui75] Geneviève Guitel. *Histoire Comparée des Numerations Écrites*. Flammarion (1975).

[Gut00] Ron Gutman. "Exploiting 64-bit parallelism". *Dr. Dobb's Journal* (septiembre 2000).

[Hei00] Ernst A. Heinz. *Scalable Search in Computer Chess. Algorithmic Enhancements and Experiments at High Search Depths*. Computational Intelligence. Vieweg Verlag (2000).

[Hil99] Raymond Hill. *A first course in coding theory*. Oxford Clarendon Press (1999).

[HLL+92] D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps, C. A. Rodger y J. R. Wall. *Coding theory: the essentials*. Marcel Dekker (1992).

[Hof99] Douglas R. Hofstadter. *Gödel, Escher, Bach, an Eternal Golden Braid*. Basic Books (1999).

[Hof01] Douglas R. Hofstadter. *Gödel, Escher, Bach, un Eterno y Grácil Bucle*. Tusquets (2001).

[Ifr01] Georges Ifrah. *Historia universal de las cifras: la inteligencia de la humanidad contada por los números y el cálculo*. Espasa Calpe (2001).

[JJN74] J. N. Judice, J. F. Jarvis y W. Ninke. "Using Ordered Dither to Display Continuous Tone Pictures". En *Proceedings of the Society for Information Display*, páginas 161–169 (1974).

[JS91] Ben Johnson y Ben Shneiderman. "Treemaps: a space-filling approach to the visualization of hierarchical information structures". En *Procceedings of the 2nd International IEEE Visualization Conference*, páginas 284–291 (October 1991).

[KL93] K. Koyama y T. W. Lai. "An optimal mastermind strategy". *Journal of Recreational Mathematics* **25**, 251–256 (1993).

[Knu77] Donald K. Knuth. "The computer as master mind". *Journal of Recreational Mathematics* **9**, 1–6 (1976–77).

[Knu95] Donald E. Knuth. *El arte de programar ordenadores*, volumen 2, Algoritmos seminuméricos. Reverté (1995).

[Man89] Udi Manber. *Introduction to Algorithms: a Creative Approach*. Addison-Wesley (1989).

[Man97] Benoît Mandelbrot. *La geometría fractal de la naturaleza*. Tusquets (1997).

[Mar00] Luisa Marqués, editor. *Fotografiando las matemáticas*. Carroggio (2000).

[Mor84] B. J. T. Morgan. *Elements of simulation*. Chapman and Hall (1984).

[MP01] Óscar Martín Sánchez y Cristóbal Pareja Flores. "Two reflected analyses of Lights Out". *Mathematics Magazine* **74**(4), 295–304 (2001).

[MvV00] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press (2000).

[O'R01] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press (febrero 2001).

[Par01] Juan Parrondo. "Perder + perder = ganar. Juegos de azar paradójicos". *Investigación y Ciencia* **298** (julio 2001).

[Pas65] Blaise Pascal. Des caractères de divisibilité des nombres de déduits de la somme de leurs chiffres, oeuvres complètes (1665).

[Pas01] Michel Pastoureau. *Blue: The History of a Color*. Princeton University Press (2001).

[Per88] Ya. I. Perelman. *Fun with Maths and Physics*. Mir Publishers (1988).

[POAR97] Cristóbal Pareja Flores, Manuel Ojeda Aciego, Ángel Luis Andeyro Quesada y Carlos Rossi Jiménez. *Desarrollo de algoritmos y técnicas de programación en Pascal*. Ra-Ma (1997).

[Pom83] Carl Pomerance. "A la búsqueda de los números primos". *Investigación y Ciencia* **77**, 80–99 (febrero 1983).

[PR86] Heinz-Otto Peitgen y Peter H. Richter. *The Beauty of Fractals: Images of Complex Dynamical Systems*. Springer Verlag (1986).

[PS91] Franco P. Preparata y Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer Verlag (enero 1991).

[PV87] L. Pardo y T. Valdés. *Simulación. Aplicaciones prácticas en la empresa*. Díaz de Santos, S. A. (1987).

[Rim93] Steve Rimmer. *Bit-Mapped Graphics*. Windcrest/MacGraw-Hill (1993).

[Sed89] Robert Sedgewick. *Algorithms*. Addison-Wesley (1989).

[Shn92] Ben Shneiderman. "Tree Visualization with Tree-Maps: a 2-d Space-Filling Approach". *ACM Transactions on Graphics* **11**(1), 92–99 (1992).

[Sue92] Cayo Suetonio Tranquilo. *Vidas de los doce césares*. Gredos (1992).

[Tay15] Brook Taylor. *Methodus incrementorum directa et inversa*. Londini: Impensis Gulielmi Innys (1715).

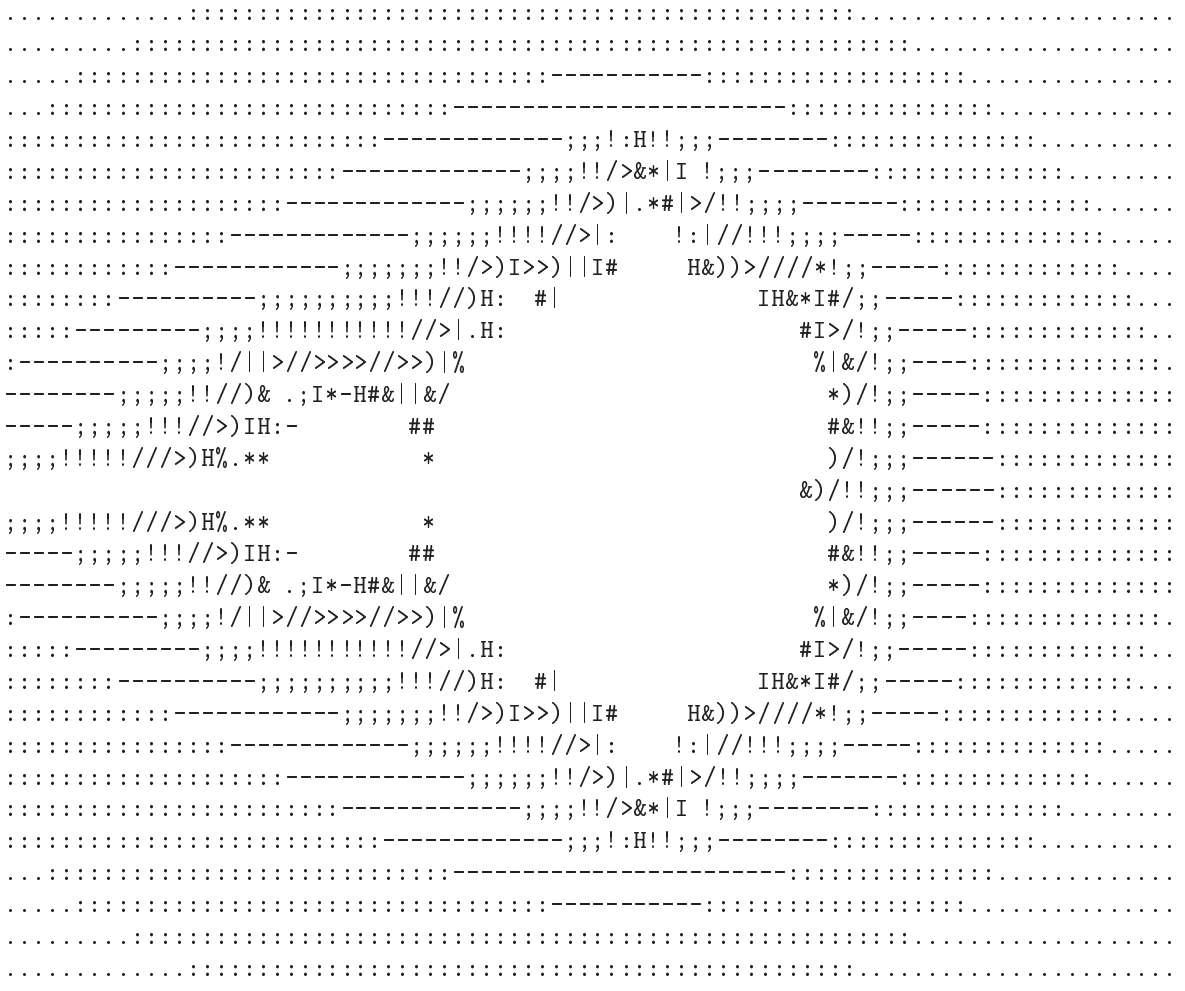
[Tho00] James R. Thompson. *Simulation: a Modeler's Approach*. John Wiley & Sons (2000).

[vWvdW99] Jarke J. van Wijk y Huub van de Wetering. "Cushion Treemaps: Visualization of Hierarchical Information". En *Proceedings of the 1999 IEEE Symposium on Information Visualization* (1999).

[Wol96] Stephen Wolfram. *Cellular Automata an Complexity: collected papers*. Reading, MA: Addison-Wesley (1996).

[Yak77] S. J. Yakowitz. *Computational probability and simulation*. Addison-Wesley Publishing Company (1977).

```
main(k){float i,j,r,x,y=-16;while(puts(""),y++<15)for(x
=0;x++<84;putchar(" .:-;!/>)|&IH%*#"[k&15]))for(i=k=r=0;
j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);}
```



Trabajé el aire,
se lo entregué al viento:
voló, se deshizo,
se volvió silencio.

Por el ancho mar,
por los altos cielos,
trabajé la nada,
realicé el esfuerzo,
perforé la luz,
ahondé el misterio.

Para nada, ahora,
para nada, luego:
humo son mis obras,
ceniza mis hechos.

...y mi corazón
que se queda en ellos.

Ángel González