Capítulo TRES

Subprogramas

```
#include<iostream>
void leerEnteroNoNegativo( int& n ) {
  do {
    cout << "escribe un entero no negativo:  " << flush;
    cin >> n;
  } while(n < 0);
}
void escribirCifrasInvertidas( int const n ) {
  if (n < 10) {
    cout << n;
  } else {
    cout << n  % 10;
    escribirCifrasInvertidas(n / 10);
  }
}
int numCifras(int const n) {
  if (n < 10) return 1;
  else return 1+ numCifras(n / 10);
}
int main (){
  int num;
  leerEnteroNoNegativo(num);
  cout << num  << " invertido es ";
  escribirCifrasInvertidas(num);
  cout << " y tiene " << numCifras(num)  << " cifras " << endl;
}
```

Resumen 99
3.1 Funciones . . . 99
3.2 Acciones o procedimientos . . . 101
3.3 Elementos avanzados . . . 101

Enunciados 103
3.1 Incierta igualdad de números reales . . . 103 131
3.2 Triángulos rectángulos enteros . . . 103
3.3 Ponle tú el título . . . 103 131
3.4 Descomposición de un número . . . 104
3.5 Palíndromos . . . 104
3.6 Cuentaletras . . . 105 131
3.7 Movimiento de una partícula dentro de una superficie . . . 106
3.8 Generación de primos . . . 107
3.9 Conjetura de Goldbach . . . 107
3.10 El factorial en la sociedad del futuro . . . 108
3.11 Sucesión bicicleta . . . 109 134
3.12 Cotejo de n-gramas . . . 109
3.13 Simulación de variables aleatorias . . . 110
3.14 Aproximación hacia π con dardos . . . 114
3.15 Conjetura para la formación de palíndromos . . . 114
3.16 Juegos perdedores ganan . . . 115
3.17 La tabla de Galton . . . 116
3.18 El retrato robot . . . 116
3.19 Número de ceros en que termina un factorial . . . 118 135
3.20 Representación de un número con palabras . . . 118
3.21 ¿Cuál es el mejor orden para recibir los datos de un polinomio? . . . 119 136
3.22 Calificación de oído . . . 120
3.23 Los cuadrados abominables y cáncer . . . 120
3.24 Codificaciones de plantas con cadenas . . . 121 138
3.25 Triángulos anidados . . . 122 143
3.26 Cálculo puntual de la matriz de mediotono de Judice-Jarvis-Ninke . . . 123 144
3.27 Dibujo de árboles mediante fractales . . . 124 146
3.28 Dragones y teselas . . . 126 148

Pistas 129

Soluciones 131

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.)

3.1 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.

3.2 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.

3.3 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

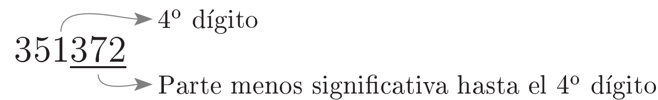
En este ejercicio proponemos la definición de funciones sencillas que permitan descomponer un número de múltiples formas.

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

Ejemplo

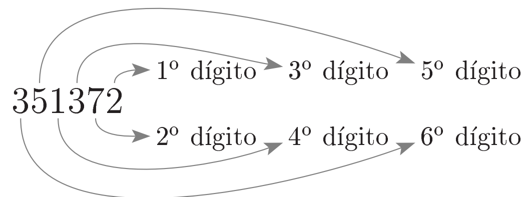
Parte menos significativa Escribe una función que devuelva la parte menos significativa hasta el n-ésimo dígito de un número.

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

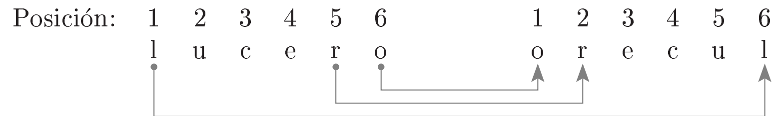
Un palíndromo es una palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda; por ejemplo, la palabra “anilina”, o las frases “anula la luz azul a la luna” y “sé verla al revés”, despreciando las diferencias entre mayúsculas y minúsculas, los espacios en blanco entre palabras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

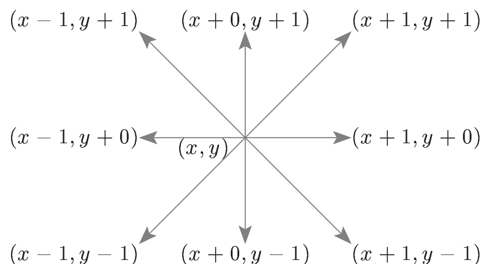
Si quieres encontrar más πnemotécnicos y πemas puedes consultar la página: `http://www.cilea.it/~bottoni/www-cilea/F90/piph.htm`.

Movimiento de una partícula dentro de una superficie

129

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

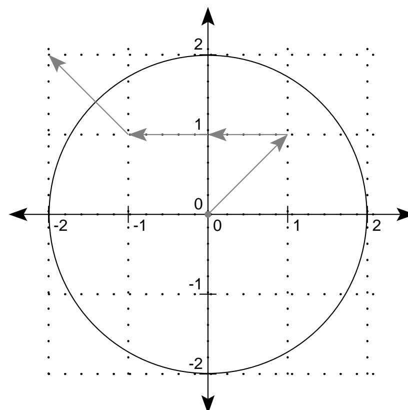
Movimientos dentro de un círculo Supongamos que tenemos una partícula que se mueve de forma aleatoria en el plano en las coordenadas enteras. La partícula puede pasar con la misma probabilidad a cualquiera de las posiciones en las que una o ambas de sus coordenadas actuales varíe de forma entera en -1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

3 8 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.

3 9 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.

3 11 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.

3 12 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:

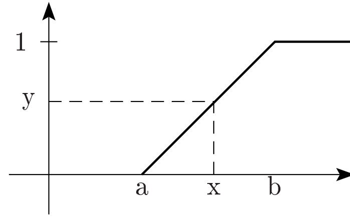


Figura 3.2: Función de distribución de una variable aleatoria continua

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

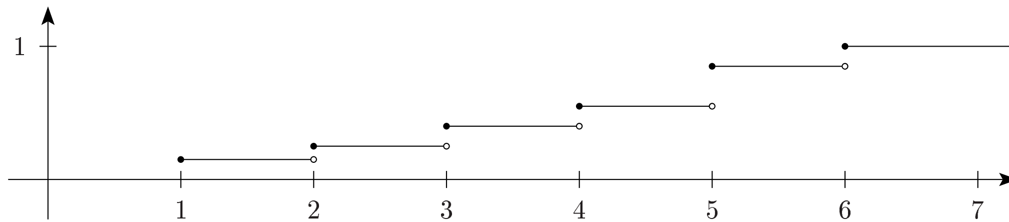


Figura 3.3: Función de distribución de una variable aleatoria discreta

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 $\min\{k \text{ tal que } P(k) \geq y\}$. 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 λ; 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*

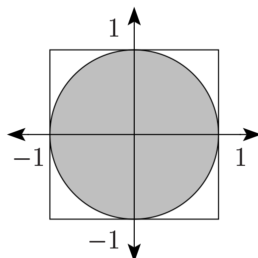
Escribe una función para simular un dado con infinitas caras, de forma que saque un 1 con probabilidad 1/2; un 2 con probabilidad 1/4; un 3 con probabilidad 1/8; etc.

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 π 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 π.

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.

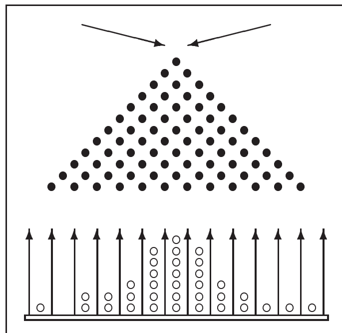


Figura 3.4: La tabla de Galton

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     \\\//////     |"""""""|     |||||||||
 |  O O  |     |-(. .)-|     |-(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

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?

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

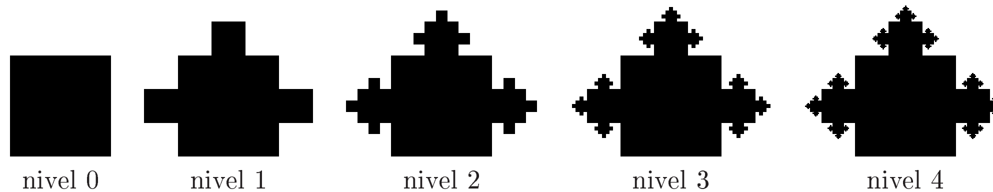
Un profesor, ya curtido por los años, se ha ido convenciendo por su experiencia de que el tamaño de las orejas está directamente relacionado con la inteligencia, y ésta con la nota que se obtiene en los exámenes. No entraré en los pormenores de su cuidadoso razonamiento, pero el caso es que ha llegado 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:

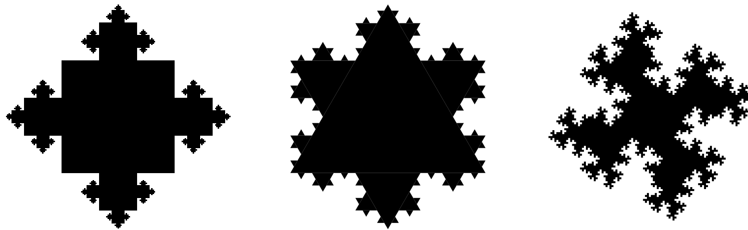


Es decir, en el nivel 0 tenemos un cuadrado de lado ℓ; en el nivel 1 pegamos un cuadrado de lado $\ell/3$ en tres lados del cuadrado anterior; en el nivel 2 pegamos un cuadrado de lado $\ell/9$ en tres lados (los que quedan libres) de los cuadrados añadidos en el nivel anterior; etc. Tal vez sea más 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`$|_{6}$`ITTHTAT`$|_{13}$`ITHTATB`$|_{20}$`DTHTATB`$|_{27}$`TB`$|_{29}$`DTTHTAT`$|_{36}$`ITHTATB`$|_{43}$`DTHTATB`$|_{50}$`TB`$|_{52}$`TT`$|_{54}$`ITHTATB`$|_{61}$`DTHTATB`$|_{68}$`TB`$|_{70}$. 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.

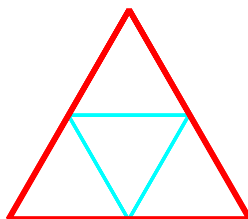
El *nivel* de una rama indica cuántas bifurcaciones hubo que tomar para llegar a ella. En el níspero, el tronco está en el nivel 0, la rama $[6, 29]$ está en el nivel 1, y la rama $[13, 20]$ está en el nivel 2. La *longitud* de una rama es el número de tramos que contiene.

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.

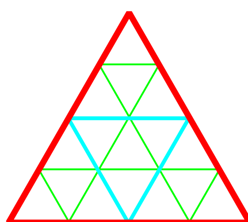
Un poco de historia Aristid Lindenmayer (1925–1989) presentó en 1968 una formalización matemática del crecimiento de las plantas que luego se ha dado en llamar sistemas de Lindenmayer o simplemente sistemas-L (*Lindenmayer Systems* o *L-systems* en inglés, para que lances una búsqueda en tu indexador 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?

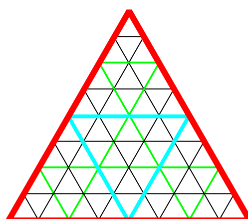


No es muy difícil. ¿Te atreves con este otro?



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

La *matriz de mediotono de Limb* de nivel 0, que denotaremos con L_0, es una matriz entera de tamaño 1×1 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

$$L_n = \begin{bmatrix} 4L_{n-1} & 4L_{n-1} + 3U_{n-1} \\ 4L_{n-1} + 2U_{n-1} & 4L_{n-1} + 1U_{n-1} \end{bmatrix}$$

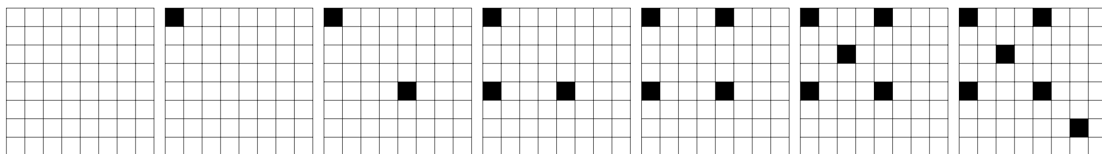
donde U_n es una matriz $2^n \times 2^n$ con todos sus elementos a 1.

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.)

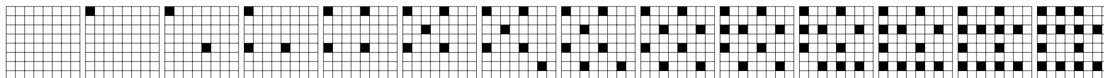
Un poco de historia Estas matrices pretenden solventar el problema de plasmar una imagen con múltiples tonos de gris en un dispositivo que sólo admite blanco o negro. Para articular esta exposición utilizaremos L_3, que es una matriz 8×8:

$$L_3 = \begin{bmatrix} 0 & 48 & 12 & 60 & 3 & 51 & 15 & 63 \\ 32 & 16 & 44 & 28 & 35 & 19 & 47 & 31 \\ 8 & 56 & 4 & 52 & 11 & 59 & 7 & 55 \\ 40 & 24 & 36 & 20 & 43 & 27 & 39 & 23 \\ 2 & 50 & 14 & 62 & 1 & 49 & 13 & 61 \\ 34 & 18 & 46 & 30 & 33 & 17 & 45 & 29 \\ 10 & 58 & 6 & 54 & 9 & 57 & 5 & 53 \\ 42 & 26 & 38 & 22 & 41 & 25 & 37 & 21 \end{bmatrix}$$

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

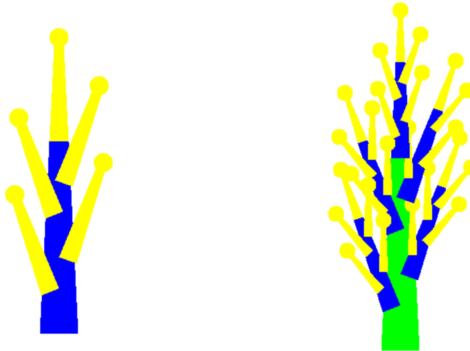
Muchas estructuras de la naturaleza se pueden dibujar en una computadora mediante el uso de fractales, no sólo *cánceres* como en el ejercicio 3.23. En este ejercicio usaremos un fractal sencillo que sirve para dibujar árboles similares a los siguientes:

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 $0 \leq \texttt{rojo}, \texttt{verde}, \texttt{azul} \leq 1$. (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

¿Te gusta el dibujo que aparece en la figura 3.5? Es un fractal, llamado la *curva del dragón* debido a

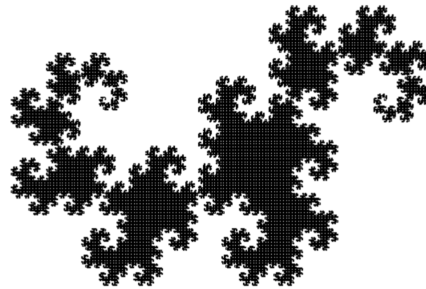


Figura 3.5: Curva del dragón

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.*

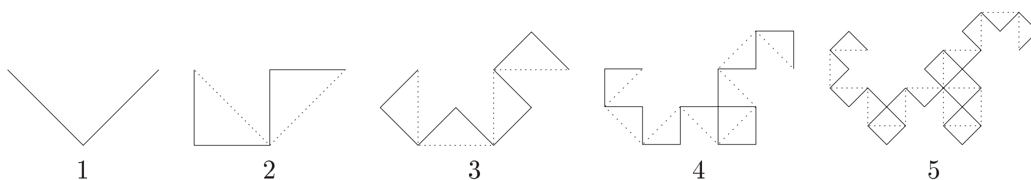


Figura 3.6: Pasos en la generación de una curva dragón

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.

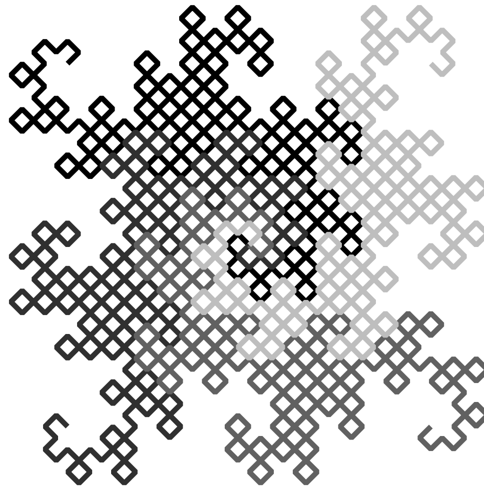
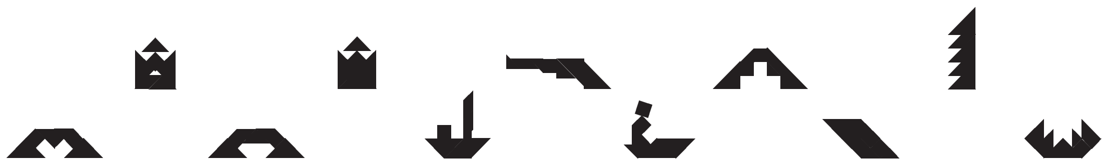


Figura 3.7: Cuatro dragones acoplados entre sí

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.

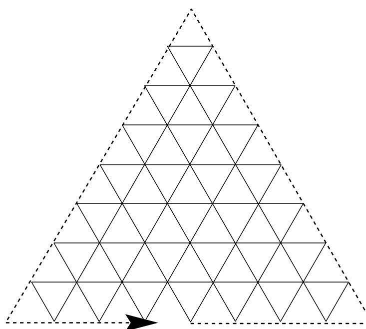
3.16a. El ejercicio 3.13 te será útil para definir las funciones aleatorias que necesitas utilizar en la simulación de los juegos que propone el ejercicio.

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,

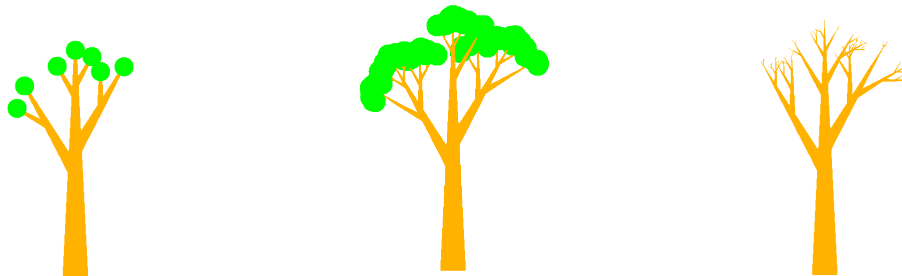
$$L_1 = \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}$$

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.



3.1 Incierta igualdad de números reales

103

Una primera idea consiste en cambiar la comparación $x = y$ por la siguiente,

$$|x - y| < \varepsilon$$

siendo ε 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);
```

3.3 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
```

3.6 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,

$$\texttt{grado(7, 24)} \leadsto \texttt{0}$$

y el caso recurrente se halla a partir del grado de multiplicidad del primero, dividido exactamente por el segundo:

$$\texttt{grado(2, 24)} \leadsto \texttt{1 + grado(2, 12)}$$

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 δ, el nuestro α, el intermedio será β, mientras que γ será un cuarto orden $a_0, \ldots, a_n, b$.

Tanto el orden α como el β 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 α 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 β 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 γ. 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 δ 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.

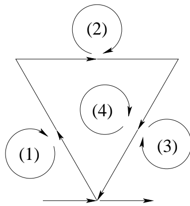
Triángulos anidados

122 129

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 π 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 π 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 8×8, que llamaremos L_3. Como $(4, 3)$ cae en la submatriz de 4×4 de la esquina superior izquierda (L_2) pasamos a buscarlo ahí. En este nuevo nivel, $(4, 3)$ cae en la submatriz de 2×2 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;
  }
}
```