Entrenamiento OIE 2023 Nivel Inicial

Sesión 4: Más bucles y bucles anidados

La cuarta sesión de entrenamiento continúa con los bucles y añade un ingrediente nuevo: bucles dentro de bucles. Los últimos problemas los utilizarán para generar algunas figuras, como hexágonos y tableros de ajedrez.

Varios de los problemas de esta serie utilizan como esquema de la entrada el que usa una *marca de fin*. Es decir, hay que ir leyendo casos hasta que uno de ellos, especial, marca el final. En las plantillas de las soluciones se anima a utilizar un esquema del tipo:

```
1  #include <iostream>
2  using namespace std;
3
4  bool casoDePrueba() {
5      Leer caso de prueba
6      if (es el caso de prueba que marca el final)
7          return false;
8      else {
9          // CÓDIGO PRINCIPAL AQUÍ
10         return true;
11     }
12 }
13
14 int main() {
15     while (casoDePrueba()) {
16     }
17     return 0;
18 }
```

En las soluciones suministradas en este cuadernillo *no* se proporciona el código completo. Nos limitamos a escribir el código de la lectura de la entrada del caso de prueba ("línea 5") y a procesarlo ("línea 9"), ignorando la comprobación de la condición de salida y el resto de la plantilla. Recuerda que deberás añadir esos elementos que faltan si quieres probar la solución en el juez.

Como siempre, los problemas propuestos están disponibles en *¡Acepta el reto!* Los marcados con una estrella (★) tienen un nivel de dificultad algo mayor que el resto y se dejan para el final.

Recuerda. Primero resuelve el problema ¡y luego escribe el código!

Los problemas de esta sesión de entrenamiento son:

- 360 Rellenando el agua de la fuente
- 297 Resistencias en serie

- 265 Suma descendente
- 151 ¿Es matriz identidad?
- 262 Ada, Babbage y Bernouilli
- 150 ¡A dibujar hexágonos! ★
- 162 Tableros de ajedrez ★

360 Rellenando el agua de la fuente

Categorías

- Bucles simples

Resumen del enunciado

El problema nos habla de las fuentes con circuitos cerrados de agua. Estos tienen una capacidad inicial de agua, que se va perdiendo poco a poco. Cuando baja de un umbral, se pone en marcha el proceso de rellenado para volver a la cantidad deseada y que el sistema no se estropee.

Cada caso de prueba es la capacidad del circuito de una fuente, la cantidad mínima que debería tener, y una serie de "pérdidas de agua" que se sufren. Hay que decir cuántas veces saltará el proceso de rellenado.

Solución

Para resolver el problema tenemos que *simular el proceso*. Llevamos en una variable la cantidad de agua que hay en la fuente y vamos restando progresivamente las pérdidas que nos indican. Cuando, tras una resta, la cantidad de agua baja del umbral, contamos un rellenado más y reiniciamos la cantidad al valor original.

Por si tras la lectura del enunciado quedan dudas, uno de los ejemplos que lo acompañan deja ver que si la cantidad de agua, tras una resta, es exactamente la del umbral, *no* hay que rellenar la fuente. Solo se hace cuando la cantidad de agua del circuito desciende por debajo del valor mínimo.

La cantidad de "pérdidas" que hay en el caso de prueba no se conoce con antelación. En lugar de eso, el formato de la entrada nos dice que debemos parar el proceso cuando leamos que la pérdida ha sido de -1. Tendremos que utilizar un `while` para leer las pérdidas hasta alcanzar la marca de fin. Hay que tener cuidado porque algún caso de prueba podría *no tener ninguna pérdida* y que en la entrada hubiera, directamente, un -1.

```
int capacidadInicial;
int capacidadMinima;
int capacidadActual;
int rellenados = 0;
int perdida;

cin >> capacidadInicial >> capacidadMinima;

capacidadActual = capacidadInicial;

cin >> perdida;
while (perdida != -1) {
    capacidadActual -= perdida;
    if (capacidadActual < capacidadMinima) {
        // < y no <= porque hay que tener menos agua del
        // mínimo para rellenar
        ++rellenados;
```

```
        capacidadActual = capacidadInicial;
    }
    cin >> perdida;
} // while

cout << rellenados << '\n';
```

Información

Recuerda que en estas soluciones normalmente no ponemos el código completo, sino únicamente el que resuelve cada caso de prueba. Revisa las plantillas de las soluciones si tienes dudas para escribir el código que falta.

297 Resistencias en serie

Categorías

- Bucles simples
- Condicionales

Resumen del enunciado

Cada caso de prueba nos proporciona el valor de un conjunto de resistencias, medidas en varias unidades múltiplos de un ohmio. Nos piden que demos la suma total utilizando como unidad la mayor posible que no requiera decimales.

Solución

Si el concepto de "resistencia" y de "ohmios" te confunde, puedes pensar el problema de otra manera. Piensa que nos dan longitudes en milímetros, centímetros, decímetros, metros, etcétera, y que tenemos que sumarlas y dar el resultado utilizando la mayor unidad de longitud posible. Así, por ejemplo, si nos piden que sumemos 99 centímetros y 10 milímetros, tendremos que decir que la suma total es de un metro. El problema es el mismo, pero con *ohmios*, que miden resistencia, en lugar de con longitudes.

Para resolverlo, lo más fácil es que al ir leyendo cada resistencia la convirtamos a la unidad más pequeña posible, en este caso *ohmios*. Siguiendo con el ejemplo de la longitud, sería equivalente a convertir cada longitud de la entrada a *milímetros*.

Una vez que tenemos la suma completa, tenemos que escribirlo en la mayor unidad posible. En el contexto de las longitudes, si la suma final son 1.000 milímetros, tendremos que escribir 1 metro. Para hacerlo, miramos si podemos escribir, con la mayor unidad posible, la suma total sin usar decimales. Esto podemos averiguarlo utilizando *el operador módulo*. Siguiendo con nuestra versión del problema de la longitud, si la suma acumulada es divisible por 1.000.000 entonces tendremos un número exacto de *kilómetros* y podemos utilizar esa unidad.

Si no se puede, probamos con la unidad inmediatamente por debajo, y así sucesivamente hasta encontrar la primera para la que el resto de la división sea 0.

Fijándonos un poco más en los detalles, la entrada nos da las resistencias indicando primero la cantidad y luego la unidad que usa, dentro de un conjunto de opciones descritas en el enunciado. La lista termina con un 0 (sin unidad) y *podría ser vacía*. La lectura de las resistencias las hacemos por tanto usando un `while` y vigilando la posibilidad de que no tengamos que ejecutarlo ninguna vez.

Además, la unidad usada en cada resistencia la leemos *como cadena* y la comparamos con las opciones posibles.

Con todo esto, el código final queda:

```
1 unsigned int total = 0;
2 unsigned int siguiente;
3 string unidad;
4
5 cin >> siguiente;
6 while (siguiente != 0) {
```

```
    cin >> unidad;

    if (unidad == "da")
        // Decaohmios.  Un decaohmio son 10 ohmios.
        siguiente *= 10;
    else if (unidad == "h")
        siguiente *= 100;
    else if (unidad == "k")
        siguiente *= 1000;
    else if (unidad == "M")
        siguiente *= 1000000;
    else if (unidad == "G") {
        siguiente *= 1000000000;
    }
    // Si no, unidad == "o", ohmios, y no
    // hay que hacer nada.

    total += siguiente;
    cin >> siguiente;
} // while

if ((total % 1000000000) == 0)
    cout << total / 1000000000 << " G\n";
else if ((total % 1000000) == 0)
    cout << total / 1000000 << " M\n";
else if ((total % 1000) == 0)
    cout << total / 1000 << " k\n";
else if ((total % 100) == 0)
    cout << total / 100 << " h\n";
else if ((total % 10) == 0)
    cout << total / 10 << " da\n";
else
    cout << total << " o\n";
```

Cuidado

Aunque en este ejercicio no es un problema, hay que fijarse siempre en los detalles. La unidad más alta que se puede utilizar en el enunciado son los *gigaohmios* que son 10^9 ohmios. ¡Ese es un número muy alto! El enunciado nos garantiza que la suma total nunca será mayor que 2 gigaohmios, lo que significa que nunca tendremos valores por encima de 2.000.000.000 que es una buena noticia porque significa que podremos utilizar enteros normales de 32 bits.

265 Suma descendente

Categorías

- Bucles simples

Resumen del enunciado

Cada caso de prueba es un número y nos piden que calculemos su "suma descendente". Si, por ejemplo, el número tiene 6 dígitos de la forma `ABCDEF` (donde cada letra es un dígito), su suma descendente será `ABCDEF + BCDEF + CDEF + DEF + EF + F`.

Solución

Para resolver el problema necesitamos un modo de quedarnos con *los últimos dígitos* de un número para poder ir consiguiendo cada sumando. Fíjate que, siguiendo con el ejemplo, el `ABCDEF`, `BCDEF`, `CDEF`, etcétera, son los últimos 6 dígitos del número, los últimos 5, los últimos 4, y así sucesivamente.

Eso se consigue con el *operador módulo*. El último dígito de un número podemos sacarlo si nos quedamos con el resto de dividir el número entre 10. Así, el último dígito de 4578 es el 8, porque el resto de dividir 4578 entre 10 es, precisamente, 8. Los dos últimos dígitos se consiguen con el resto de *dividir por 100*, los tres siguientes con el de 1000 y así sucesivamente.

Al definirnos lo que es la suma descendente, el enunciado parece que nos anima a conseguir primero los sumandos mayores y luego ir quitando dígitos por la izquierda. Pero programar la solución así es más difícil. Es más sencilla si *damos la vuelta a la suma* y vamos consiguiendo los sumandos del pequeño hacia el mayor. Esto se debe a que el más pequeño lo conseguimos, como hemos visto, con el módulo con 10, el siguiente con el módulo con 100, etcétera. En cada paso, conseguimos el nuevo valor con el que calcular el módulo (10, 100, 1000...) multiplicando por 10 el anterior.

Si lo hiciéramos al contrario, sería equivalente pero dividiendo por 10 el anterior. El problema es que hacerlo en ese sentido *nos exige saber el primer valor* por el que dividir, y para eso necesitamos saber la cantidad de dígitos del número. Es mucho más sencillo empezar por 10 e ir subiendo hasta que el valor siguiente sea demasiado grande.

Al final, lo que tenemos que hacer es *repetir un proceso* una cantidad de veces que, de antemano, no sabemos cuántas son. Por tanto utilizaremos un `while`. Necesitamos tres variables:

- El número leído cuya *suma descendente* estamos calculando.
- La suma acumulada hasta el momento.
- El valor con el que calculamos el módulo en cada vuelta, y que iremos multiplicando por 10.

Tendremos que dejar de dar vueltas *cuando hayamos sumado el número completo*. Eso ocurrirá cuando el resto de la división que calculemos sea el número original, algo que solo será cierto cuando el valor por el que dividamos sea *mayor* que el número.

En este problema hay que pensar especialmente el orden de los pasos. Es decir, hay que elegir bien cuando multiplicamos por 10 el divisor, y cuando sumamos.

```cpp
unsigned int num;
unsigned long long result = 0;
unsigned long long divisor = 1;

cin >> num;
while (num >= divisor) {
    divisor *= 10;
    result += num % divisor;
}
cout << result << "\n";
```

Cuidado

Estamos usando tanto para `result` como para `divisor` enteros grandes (de 64 bits) en lugar de los más habituales de 32. Para `result` no es necesario, realmente, porque, si lo piensas un poco, la salida nunca puede ser tan grande. No obstante, ponerlo así nos puede evitar algunos avisos (equivocados) del compilador sobre posibles desbordamientos.
En `result` si es importante. El enunciado nos dice que el número leído puede llegar a ser 1.000.000.000. Ante ese número, la última vuelta del bucle utilizará un divisor enorme de 10.000.000.000 que *no* entra en un entero de 32 bits, de ahí que necesitemos dar el salto al tamaño siguiente.

Hazlo por tu cuenta

En este problema, el formato de la entrada nos dice que hay que terminar cuando el valor leído sea 0, por lo que la adaptación del código anterior para enviarlo al juez es un poco más laboriosa.
Además, eso significa que nunca nos pedirán que calculemos el valor de la suma descendiente de 0 (porque es la marca de fin). Por tanto, el bucle que tenemos en el código *siempre se ejecutará al menos una vez*. Prueba a sustituir el `while` por un `do-while` para aprovecharlo.

Para valientes

Si se resuelve el problema utilizando enteros normales en lugar de los de 64 bits es posible que la solución ¡también funcione! Esto va en contra de nuestro análisis, porque al calcular el número 10.000.000.000 deberíamos sufrir desbordamiento y que el programa funcione de forma errónea.
La realidad es que al intentar guardar el número 10.000.000.000 en un entero de 32 bits, el valor que se guarda (por el desbordamiento) es el 1.410.065.408 (te dejaremos a ti que pienses por qué). Si se utiliza ese valor en lugar del 10.000.000.000 correcto, el cálculo del módulo *no cambia* y por tanto la suma funciona. Además, ese número será también mayor que el valor leído, y por tanto saldremos en la siguiente vuelta.
Esto significa que, incluso aunque usemos enteros normales, la solución funcionará aunque *conceptualmente estará mal*. ¡Hay cosas que a los jueces se les escapan!

151 ¿Es matriz identidad?

Categorías

- Bucles anidados

Resumen del enunciado

Cada caso de prueba nos da una *matriz cuadrada* y nos preguntan si es o no *matriz identidad*.

Una matriz cuadrada de tamaño n es un grupo de $n \times n$ números colocados en n filas y n columnas. Será *identidad* si los números de la *diagonal principal* (la que va de arriba a la izquierda a abajo a la derecha) son 1, y el resto son 0.

Solución

La entrada de cada caso comienza con el número n que indica el tamaño de la matriz. Luego le siguen n filas, cada una con n números, que forman la matriz completa.

Lo que tenemos que hacer es leer los $n \times n$ números y analizarlos para saber si, en conjunto, forman una matriz identidad. Para eso, podríamos hacer un bucle que se ejecutara $n \times n$ veces, dado que es la cantidad de números que tenemos. Pero en cada vuelta nos interesa saber en qué fila y en qué columna estamos, para saber si estamos o no en una posición de la *diagonal principal*.

Podríamos intentar averiguar la fila y la columna a partir del contador del bucle pero resulta mucho más sencillo si usamos *bucles anidados*. Un bucle anidado es un bucle dentro de otro. La idea es tener un *bucle externo* que se ejecutará una vez por cada fila, y para cada fila, tenemos dentro otro bucle que se ejecuta una vez por cada columna de la fila actual.

Información

El bucle externo se ejecuta una vez por fila, y el interno una vez por cada columna de la fila actual. ¿Por qué es así y no al contrario? ¿No podría ser que el bucle externo fuera por columnas y el interno por filas? No. Es así por *la entrada*. Los primeros n números que leeremos forman *la primera fila*. Eso significa que primero leeremos los n números de la primera fila, luego los n de la segunda, y así sucesivamente. Eso significa que lo que estamos haciendo es "para cada fila, leo sus columnas", y no "para cada columna, leo sus filas".

Al principio puede resultar difícil pensar en tener un bucle dentro de otro y suele ayudar "abstraer" lo que se hace dentro. Si lo ponemos en *pseudocógido* tendremos algo así:

```
para cada fila f:
  mirar si sus numeros son 0's salvo el que esta en la diagonal principal
```

En este código, tenemos el bucle externo que recorre por filas. Para programar el interior ("mirar si sus números son 0's salvo el que está en la diagonal principal") resulta que necesitamos recorrer todos los números de esa fila, y eso se consigue con un segundo bucle, que queda encerrado dentro del primero:

```
para cada fila f:
  para cada columna c:
    si el numero esta en la diagonal principal y no es un 1
       no es matriz identidad
    si el numero no esta en la diagonal principal y no es un 0
       no es matriz identidad
```

Fíjate que el bucle externo dará n vueltas, y, por cada una, el interno dará también n vueltas, lo que hace un total de $n \times n$ que coincide con la cantidad de números que tenemos en la matriz de la entrada.

Ahora nos queda saber si estamos o no en la diagonal principal. En la primera fila, la columna que está en la diagonal principal es la de más a la izquierda, es decir, la primera. Por su parte, en la segunda fila, la diagonal principal está en la segunda columna. Esto ocurre todo el tiempo, hasta llegar a la última fila, en donde la diagonal principal está en la última columna. Por tanto, para saber si estamos leyendo la posición de la diagonal principal basta mirar si la fila y la columna tienen el mismo número:

```
para cada fila f:
  para cada columna c:
    si f == c y el numero no es un 1
       no es matriz identidad
    si f != c y el numero no es un 0
       no es matriz identidad
```

Un último detalle importante es que si una matriz no es identidad, vamos a saberlo en cualquier momento intermedio, antes de leer la matriz entera. Aun así, *es importante leerla entera* porque si no, dejaremos el resto de la matriz en la entrada sin procesar, y estropearemos el procesamiento de los casos de prueba que vayan detrás. Lo que tenemos que hacer, por tanto, es *terminar siempre la lectura*, aunque podamos sentir la tentación de contestar a mitad y terminar. Necesitaremos recordar si, en algún momento, hemos demostrado que la matriz *no* es identidad, para escribir una cosa u otra al terminar la lectura completa.

Tras todo este análisis, escribir el código debería resultar sencillo:

```cpp
unsigned int tam;

cin >> tam;
bool identidad = true; // Mientras no se demuestre lo contrario
for (unsigned int f = 0; f < tam; ++f) {
    for (unsigned int c = 0; c < tam; ++c) {
        int v;
        cin >> v;
        if (f == c && v != 1)
            identidad = false;
        if (f != c && v != 0)
            identidad = false;
    }
}
if (identidad)
    cout << "SI\n";
else
```

```
18      cout << "NO\n";
```

Información

Si conoces el concepto de programación de *arrays*, puede que hayas pensado en resolver el problema utilizando uno. Aunque una solución así es correcta, observa que en este problema no necesitamos "volver atrás" para mirar varias veces el valor de las posiciones de la matriz. Eso nos permite ahorrarnos el array, porque no necesitamos mantener en memoria la matriz completa. Mirando un número cada vez (y recordando por donde vamos) podemos saber si es o no identidad sin demasiados problemas.

Hazlo por tu cuenta

Este problema se puede resolver de otras formas:

- Utilizando un único bucle desde 1 hasta $n \times n$ (o, mejor, desde 0 hasta $n \times n - 1$) y averiguando en cada paso la fila y la columna actual a partir del índice del bucle único.
- Dándonos cuenta de que una matriz identidad empieza en un 1, y luego tiene n 0's, y otro 1. Esto se repite $n - 1$ veces. Podemos ignorar las filas y las columnas utilizando esta idea.

Resuelve otra vez el problema usando las dos aproximaciones.

262 Ada, Babbage y Bernouilli

Categorías

- Bucles anidados
- Aritmética modular

Resumen del enunciado

Nos dan dos números, n y p, y hay que calcular:

$$1^p + 2^p + \ldots + n^p$$

Como el número resultante es muy grande, nos lo piden módulo 46.337.

Solución

En pseudocódigo, lo que tenemos que hacer es algo así:

```
resultado = 0;
para cada i desde 1 hasta n:
    resultado = resultado + potencia(i, p)
```

donde `potencia(i, p)` es i^p, es decir i multiplicado por él mismo p veces.

Para calcular una potencia, algunos lenguajes de programación proporcionan un operador especial, pero C, C++ o Java no lo hacen. A cambio, en ellos la *librería estándar* del lenguaje proporciona funciones matemáticas que nos permiten calcular potencias.

El problema es que el resultado *puede ser muy grande*. En particular, los límites del enunciado permiten que tanto n como p puedan llegar a ser 100, y 100^{100} es un número muy grande. Si utilizamos las funciones de la librería, perderemos precisión y el resultado no nos servirá. Debemos hacer el cálculo nosotros mismos.

Información

Esto *no* significa que utilizar las funciones de la librería del lenguaje sea mala idea de forma general. De hecho, cuando calculemos nosotros el valor, *también* vamos a sufrir desbordamiento inicialmente, y generaremos un resultado incorrecto. Pero lo arreglaremos después, gracias a la extraña solicitud del enunciado de pedirnos el resultado módulo 46.337. Ese arreglo no podemos hacerlo si usáramos la función del cálculo de la potencia proporcionada por el lenguaje, de ahí que hagamos su cálculo de forma manual.

Para calcular la potencia necesitamos *un segundo bucle* lo que nos lleva a tener *bucles anidados*:

```
1 // Solución errónea, sigue leyendo.
2 unsigned int n, p;
3 unsigned int r = 0;
4
5 cin >> n >> p;
```

```
for (unsigned int base = 1; base <= n; ++base) {
    unsigned int aux = 1; // Neutro de la multiplicación
    for (unsigned int exponente = 1; exponente <= p; ++exponente) {
        aux *= base;
    }
    r += aux;
}
cout << r << '\n';
```

El bucle externo recorre todas las *bases*, es decir los números de 1 a n. Para cada uno, calculamos la potencia con el *bucle interno* que multiplica tantas veces como diga el exponente (p).

Esta solución no es aún correcta porque el enunciado nos dice "dado que el resultado puede ser muy alto, se dará módulo 46.337". Alguien poco acostumbrado a este tipo de solicitudes podría verse tentado a calcular el módulo al final, una vez que se tiene el resultado completo:

```
cout << r % 46337<< '\n';
```

Hacerlo así supondrá una solución incorrecta. El uso del módulo nos lo dan *como ayuda* para evitar la necesidad de números grandes. Como se decía antes, el valor calculado puede llegar a ser muy alto, más allá del límite de la representación de los números enteros de 32 y de 64 bits. Si no hacemos nada, el valor calculado, por tanto, será incorrecto y calcular al final su módulo con 46.337 no ayudará en nada.

Como nos dice el enunciado, el uso del módulo nos ayuda porque *podemos irlo aplicando tras cada operación*. Al sumar, por ejemplo, la última potencia calculada, vamos quizá a conseguir un número que es mayor o igual que el 46.337. En lugar de ignorar esta posibilidad y seguir "dejándolo crecer", las matemáticas (y el enunciado) nos dicen que podemos calcular el módulo del resultado parcial con el número y el resultado final no se verá afectado. Esto es importante porque nos hace al número más pequeño, volviendo a "terreno conocido" lejos de los límites de la representación. Eso nos da otra vez espacio por arriba para la siguiente operación, de modo que no suframos desbordamiento al aplicarla. Calculando el módulo todo el tiempo, nos mantiene el tamaño del resultado bajo control y no sufriremos un desbordamiento que nos estropee el resultado.

Al final, el código queda:

```
unsigned int n, p;
unsigned int r = 0;

cin >> n >> p;
for (unsigned int base = 1; base <= n; ++base) {
    unsigned int aux = 1;
    for (unsigned int exponente = 1; exponente <= p; ++exponente) {
        aux *= base;
        aux %= 46337; // Volvemos a un valor pequeño
    }
    r += aux;
    r %= 46337;       // Volvemos a un valor pequeño
}
cout << r << '\n';
```

Para valientes

El número 46.337 es un número primo. No es necesario que lo sea para que la aritmética modular funcione, pero es muy habitual que se usen números primos en este tipo de situaciones. Si fuera un número compuesto, sería relativamente fácil que no todos los números pudieran terminar estando en la salida.
Pero el 46.337 no es un número primo cualquiera. Es el mayor número primo que es menor que la raíz cuadrada de 2^{31}. Esto permite que si vamos aplicando la aritmética modular como hemos hecho, la multiplicación de dos números *nunca* supere 2^{31} y por tanto entre dentro de los límites de la representación de los enteros con signo de la mayoría de los lenguajes de programación.
Hay muchos problemas donde se pide el resultado módulo un número, como aquí, pero con números mayores, como 1.000.000.007. Si en esos problemas hay que multiplicar, entonces se puede sufrir desbordamiento muy fácilmente, porque ese número por él mismo no entra en un entero normal. En esos problemas será por tanto necesario utilizar enteros de doble precisión.

Para valientes

En la resolución del problema hemos aprovechado las dos propiedades de la aritmética modular que nos dice el enunciado:

$$(a + b)\%k = ((a\%k) + (b\%k))\%k$$

$$(a \times b)\%k = ((a\%k) \times (b\%k))\%k$$

Es tentador extrapolar estas dos propiedades y pensar que también funcionan con la división. Pero *no es así*. El módulo interactúa con la división de forma contraproducente en este caso y en general:

$$(a \div b)\%k \neq ((a\%k) \div (b\%k))\%k$$

Esto no afecta a este problema, pero tenlo en cuenta si te encuentras problemas similares en el futuro.

150 ¡A dibujar hexágonos! ★

Categorías

- Bucles anidados
- Funciones

Resumen del enunciado

El objetivo del problema es dibujar hexágonos. Cada caso de prueba nos da la longitud del lado del hexágono, y el carácter con el que hay que pintarlo.

Por ejemplo, para lado 3 y * como símbolo hay que escribir:

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

Solución

Para "pintar" el hexágono tenemos que escribir cada línea, de una en una. No podemos "mover" el sitio donde escribiremos el próximo símbolo, por lo que necesitaremos "diseccionar" la figura para saber cómo conseguir pintarla de arriba abajo, y de izquierda a derecha.

Un análisis rápido nos permite llegar a la conclusión de que cada línea es un grupo de 0 o más espacios seguido de varios símbolos (asteriscos en el ejemplo). Para que el código quede más sencillo, resulta práctico hacer una subrutina que nos escriba una serie de caracteres seguidos, de manera que podamos utilizarla muchas veces. Esta subrutina tendrá dos parámetros, el caracter a escribir y el número de veces a hacerlo, y la programaremos con un bucle:

```
1 void escribeCaracter(char c, unsigned int veces) {
2
3     for (unsigned int i = 0; i < veces; ++i)
4         cout << c;
5
6 } // escribeCaracter
```

Información

El problema podríamos resolverlo sin utilizar esta subrutina, repitiendo el código una y otra vez, pero es menos elegante. También podríamos utilizar alguna otra forma de escribir el mismo caracter varias veces, como utilizar `string` y su constructor que recibe un carácter y el número de repeticiones.

Equipados con esta subrutina, ahora toca analizar la figura para saber cómo construirla. Para que resulte más sencillo de analizar, vamos a verla sustituyendo los espacios por puntos:

```
..***
.*****
*******
.*****
..***
```

En este caso, la longitud n del lado del hexágono es 3 y vemos que:

- La primera línea tiene 2 espacios y 3 asteriscos.
- La segunda línea tiene 1 espacio y 5 asteriscos.
- La tercera línea no tiene espacios y tiene 7 asteriscos.
- La cuarta línea es como la segunda, con 1 espacio y 5 asteriscos.
- La quinta línea es como la primera, con 2 espacios y 3 asteriscos.

Si lo ponemos en una tabla, tenemos:

```
             Espacios  Asteriscos
..***            2         3
.*****           1         5
*******          0         7
.*****           1         5
..***            2         3
```

Necesitamos *una expresión* que nos diga el número de espacios y de asteriscos que tenemos que escribir sabiendo la línea en la que estamos. Vemos que hay en realidad claramente dos fases. En la mitad superior de la figura el número de espacios desciende en uno entre dos líneas consecutivas, y el de asteriscos crece en dos. En la mitad inferior el proceso es opuesto, incrementándose el número de espacios de uno en uno y reduciéndose el de asteriscos de dos en dos. La fila central se puede incorporar en cualquiera de las dos mitades.

Para resolver el problema, por tanto, será más fácil mantener estas dos fases. Tendremos un primer bucle con las líneas de la mitad superior, y otro bucle con las de la mitad inferior. Incluiremos la línea central en la primera mitad, aunque podría hacerse, de forma equivalente, en la segunda.

En cada bucle calcularemos, según la línea en la que estemos, el número de espacios y de asteriscos. La primera línea tiene tantos asteriscos como la longitud del lado del hexágono que nos piden, y tiene un espacio menos. Por su parte, la segunda mitad es igual que la primera, pero pintada en sentido inverso (y sin la última línea que es la central de la figura). Una vez que somos capaces de pintar la primera mitad, podemos pintar la segunda usando el mismo bucle pero recorriendo los índices en sentido contrario.

Aprovechando la subrutina `escribeCaracter(...)` que ya tenemos, el código queda:

```
1  int tam;
2  char c;
3  cin >> tam >> c;
4
5  // Primera mitad.
6  for (int f = 0; f < tam; ++f) {
7      escribeCaracter(' ', tam - f - 1);
8      escribeCaracter(c, tam + 2*f);
9      cout << '\n';
```

```
10  } // for f
11
12  // Segunda mitad.
13  for (int f = tam-2; f >= 0; --f) {
14      escribeCaracter(' ', tam - f - 1);
15      escribeCaracter(c, tam + 2*f);
16      cout << '\n';
17  }
```

Cuidado

¡Acepta el reto! proporciona el veredicto *Presentation Error* si la solución es correcta *salvo por los espacios o saltos de línea*. En este problema eso significa que si te confundes en el número de espacios que escribes (o si, por ejemplo, decides escribir espacios al final de cada línea), el juez te evaluará con ese veredicto (PE) en lugar de aceptar tu solución. Si lo recibes, analiza dónde te estás equivocando al escribir los espacios.

162 Tableros de ajedrez ★

Categorías

- Bucles anidados

Resumen del enunciado

El objetivo del problema es dibujar tableros de ajedrez. Cada caso de prueba nos da la longitud de cada "escaque" (cuadrado del tablero) y el carácter con el que hay que pintar los escaques negros.

Por ejemplo, para lado 2 y `#` como símbolo hay que escribir:

```
|----------------|
|  ##  ##  ##  ##|
|  ##  ##  ##  ##|
|##  ##  ##  ##  |
|##  ##  ##  ##  |
|  ##  ##  ##  ##|
|  ##  ##  ##  ##|
|##  ##  ##  ##  |
|##  ##  ##  ##  |
|  ##  ##  ##  ##|
|  ##  ##  ##  ##|
|##  ##  ##  ##  |
|##  ##  ##  ##  |
|  ##  ##  ##  ##|
|  ##  ##  ##  ##|
|##  ##  ##  ##  |
|##  ##  ##  ##  |
|----------------|
```

Solución

A la vista de la figura es fácil darse cuenta de que vamos a tener que escribir varias veces seguidas el mismo caracter. Vemos, por ejemplo, que los "bordes" del tablero superior e inferior están compuestos por una serie de guiones, y luego cada escaque tiene varios espacios o varios `#`. Lo más fácil es empezar haciéndonos una subrutina para escribir el mismo caracter varias veces:

```
1 void escribeCaracter(char c, unsigned int veces) {
2
3     for (unsigned int i = 0; i < veces; ++i)
4         cout << c;
5
6 } // escribeCaracter
```

Información

El problema podríamos resolverlo sin utilizar esta subrutina, repitiendo el código una y otra vez, pero es menos elegante. También podríamos utilizar alguna otra forma de escribir el mismo caracter varias veces, como utilizar `string` y su constructor que recibe un carácter y el número de repeticiones.

Una vez leída la entrada de cada caso de prueba (tamaño del escaque y símbolo a usar) tenemos que pintar el borde. También tendremos que pintarlo al acabar. Ambas cosas son sencillas teniendo nuestro `escribeCaracter(...)`.

Pintar el interior del tablero es más complicado. Dado que *no podemos mover* el lugar donde escribimos, es necesario escribir línea a línea, incluídos los espacios para los escaques "blancos". Tenemos que pintar 8 filas de escaques, y para cada una, 8 escaques de colores alternos, lo que nos hace pensar que tendremos dos bucles anidados.

Pero la cosa no se queda ahí. Cada fila de escaques está, en realidad, formada por varias líneas (tantas como la longitud del escaque que nos han pedido). Tenemos por tanto *tres* bucles anidados. En *pseudocódigo*:

```
para cada una de las 8 filas de escaques
  para cada una de las lineas de la fila de escaques actual
    para cada una de las 8 columnas de escaques
      pintar el escaque
```

Para pintar el escaque necesitamos saber si es "blanco" (para escribir espacios) o si es "negro" (para pintar el símbolo que nos hayan dado). Para saberlo, podemos utilizar la fila y la columna en la que estamos. Si lo piensas un poco, puedes ver que si sumas el número de fila y el número de columna en la que está colocado un escaque, su *paridad* (si la suma es par o impar) determina su color. Piensa que si estás en un determinado escaque y te vas a la fila o a la columna siguiente, se incrementa en uno la suma de fila+columna, y por tanto su paridad se invierte (pasa de par a impar, o viceversa). Si te desplazas no una, sino dos posiciones, la suma se incrementará en 2, y la paridad *no cambia*, por lo que el color se mantendrá.

Con esta idea, y nuestro `escribeCaracter(...)` el código queda:

```
1  unsigned int tam;
2  char c;
3  cin >> tam;
4  cin >> c;
5
6  // Borde superior.
7  cout << '|'; escribeCaracter('-', tam * 8); cout << "|\n";
8
9  // Tablero.
10 for (unsigned int fila = 0; fila < 8; ++fila) {
11     for (unsigned int i = 0; i < tam; ++i) {
12         cout << '|';
13         for (unsigned int col = 0; col < 8; ++col) {
14             escribeCaracter(((fila+col)%2 == 0) ? ' ' : c, tam);
15         } // for columnas
16         cout << "|\n";
```

```
17      } // for líneas por fila de escaques
18  } // for fila
19
20  // Borde inferior.
21  cout << '|'; escribeCaracter('-', tam * 8); cout << "|\n";
```