Entrenamiento OIE 2023 Nivel Inicial

Sesión 3: Bucles simples

En la tercera sesión de entrenamiento practicaremos *bucles*. En la primera sesión, el código que escribimos se ejecutaba de forma lineal, de principio a fin. La segunda sesión sirvió para practicar *condicionales*, que nos permitían *saltar* secciones de código dependiendo de algún tipo de condición. Los *bucles* que practicamos ahora permiten *repetir la ejecución* de la misma sección de código varias veces. Esto significa que la ejecución *vuelve atrás*.

Hay varias formas de programar un bucle: `for`, `while` o `do-while`. Dependiendo de la situación, interesará más utilizar un tipo u otro.

Comprendiendo los bucles y *las funciones* se entienden finalmente *los esquemas de la solución* que hemos estado usando en las sesiones anteriores.

Todos 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. Con o sin estrella, no uses *prueba y error*. Piensa siempre antes de programar la solución. Primero resuelve el problema ¡y luego escribe el código!

Los problemas de esta sesión de entrenamiento son:

- 369 Contando en la arena
- 158 Los saltos de Mario
- 359 Timo en el cocedero de mariscos
- 165 Número hyperpar
- 170 Triángulo con piedras
- 119 Escudos del ejército romano
- 205 Números de Lychrel ★
- 190 Dividir factoriales ★
- 221 Entrando al cine ★

369 Contando en la arena

Categorías

- Bucles simples
- Esquema de la entrada con marca de fin

Resumen del enunciado

Nos dan un número n y hay que escribir n `1`'s consecutivos. Se debe terminar al leer un `0`.

Solución

La escritura de un `1` es sencilla:

```
cout << "1";
// Al ser un solo carácter, también valdría:
//cout « '1';
```

Necesitamos ejecutar esa línea n veces, y para eso hay que usar *un bucle*, que es una estructura de programación para *repetir* una sección de código varias veces. Se dice que el bucle *da vueltas* repitiendo su *cuerpo* una y otra vez hasta que termina.

Hay dos grandes tipos de bucles:

- Bucle `for`: se utiliza cuando el número de veces que hay que ejecutar el código *no depende de lo que ocurra en cada ejecución*. Dicho de otro modo, es el tipo de bucle que se usa cuando el número de repeticiones *se conoce antes de empezar el bucle*.
- Bucle `while`: se utiliza cuando el número de ejecuciones no se conoce de antemano, sino que es necesario descubrirlo en función de lo que ocurra al ir dando vueltas.

En este caso tenemos que dar n vueltas, que es un número desconocido al escribir el código, pero conocido durante la ejecución. Por tanto, usaremos un bucle `for`. Necesita un *contador* para saber cuántas vueltas llevamos dadas, y salir una vez alcanzado el número deseado:

```
1 // Solución parcial.  Sigue leyendo.
2 unsigned int n;
3 cin >> n;
4 for (unsigned int i = 0; i < n; ++i)
5     cout << "1";
6 cout << "\n";
```

El código anterior no está completo porque el problema pide que si se lee un `0` se finalice la ejecución. Antes de entrar en el bucle, necesitamos comprobar que n no es la *marca de fin*. Utilizando el *esqueleto de la solución* para este esquema de la entrada, el código completo queda:

```
1 #include <iostream>
2 using namespace std;
3
4 bool casoDePrueba() {
5
```

```
6       unsigned int n;
7
8       cin >> n;
9
10      if (n == 0) {
11          return false;
12      }
13      else {
14          for (unsigned int i = 0; i < n; ++i)
15              cout << "1";
16          cout << "\n";
17          return true;
18      } // if-else caso especial de fin
19
20  } // casoDePrueba
21
22  //-----------------------------
23
24  int main() {
25
26      while (casoDePrueba()) {
27      }
28
29      return 0;
30
31  } // main
```

La ejecución del programa comienza en *la función* `main`, que, con un bucle `while`, llama a la función `casoDePrueba()` una y otra vez mientras esta devuelva `true`. Dentro, leemos el número n igual que antes (línea 8) y miramos su valor. Si n resulta ser `0`, la función `casoDePrueba()` devuelve `false`, que indica al bucle del `main` que se ha alcanzado el final y hay que terminar. Si no, ejecuta el código principal de la solución (líneas 14-16) y termina devolviendo `true`, para indicar que el procesamiento debe continuar.

Información

En el resto de soluciones no pondremos el código completo, y nos preocuparemos solo del interior de `casoDePrueba()`, que es donde queda recluída la solución para cada caso de prueba.

158 Los saltos de Mario

Categorías

- Bucles simples
- Secuencias

Resumen del enunciado

Cada caso de prueba comienza con un valor n indicando la longitud de la secuencia de números que viene a continuación. Después aparecen n números. Hay que decir cuántos de esos números son mayores que el que tienen inmediatamente antes, y cuántos son menores. Si un número es igual al anterior, no se tiene en cuenta.

Solución

Necesitamos procesar *una secuencia* de números y hacer algún tipo de análisis de cada uno de sus valores. La longitud de la secuencia es conocida de antemano, por lo que utilizaremos un bucle `for` para recorrerla.

Hay un detalle importante. El procesamiento de cada elemento de la secuencia necesita el valor inmediatamente anterior. Eso significa que:

- Durante el procesamiento necesitamos guardar *dos* valores: el que acabamos de leer y el que leímos en la vuelta anterior, cada uno en una variable.
- Antes de terminar cada vuelta, tenemos que *transferir* el contenido de la variable donde guardamos el valor que acabamos de leer a la variable donde guardamos el anterior, dado que, al comenzar la vuelta siguiente, el actual será el anterior.
- El primer número de la secuencia *no tiene valor anterior* con el que comparar. El bucle dará una vuelta menos que la longitud de la secuencia y, antes de empezar, leeremos el primer valor que será el *anterior* al primero leído dentro del bucle (el segundo de la secuencia).

Con todo esto, el código final queda:

```
unsigned int n;
unsigned int anterior, actual;
unsigned int mayores, menores;

mayores = menores = 0;

cin >> n;

cin >> anterior;
for (unsigned int i = 1; i < n; ++i) {
    cin >> actual;
    if (actual > anterior)
        ++mayores;
    else if (actual < anterior)
        ++menores;

    // ¡Saltamos al siguiente!  Nos preparamos para
    // la próxima vuelta.
```

```
19      anterior = actual;
20  } // for
21
22  cout << mayores << " " << menores << "\n";
```

Cuidado

Aunque en este ejercicio no es un problema, hay que fijarse siempre en los detalles. Si n fuera 0 (no hay números en la secuencia) tendríamos un problema al leer siempre un primer número. Afortunadamente el enunciado nos dice que n será mayor que 0 por lo que esa situación no puede ocurrir.

Hazlo por tu cuenta

¿Qué habría que cambiar en el programa si n pudiera ser 0?

359 Timo en el cocedero de mariscos

Categorías

- Bucles simples
- Secuencias

Resumen del enunciado

Cada caso de prueba es una secuencia de números naturales acabados por un -1. Hay que decir si la suma total de los números (sin contar el -1) es igual, mayor o menor que la cantidad de números leída.

Solución

Tenemos que leer una secuencia de números hasta llegar a un valor especial, el -1, que marca el fin. Necesitamos un bucle para hacer esa lectura, pero en esta ocasión *no* conocemos de antemano cuántos números hay que leer, porque es la lectura del -1 lo que marca el final. Por tanto, necesitamos un bucle `while`.

La condición del bucle `while` indica lo que se tiene que cumplir para que se entre en el cuerpo y se dé otra vuelta más. En este caso, la condición es que el último valor leído no sea un -1, que *no* debe procesarse. Esto significa que *antes* de dar la primera vuelta tenemos que haber leído el primer número, y que lo último que hay que hacer antes de terminar es leer el siguiente número, que validaremos en la condición de la vuelta siguiente. En *pseudocódigo*:

```
leer v
mientras v no sea la marca de fin
    procesar el v
    leer v // para la siguiente vuelta
```

Información

Fíjate en la necesidad de *repetir* la línea que lee el valor de `v`. Hay gente que prefiere utilizar otros esquemas para este tipo de recorridos, que evitan esa repetición, aunque, a cambio, necesitan un `if` dentro del bucle, por ejemplo. En este caso, también se puede usar el operador coma.

El ejercicio lo que nos pide es que contemos cuántas vueltas damos al bucle, y que sumemos todos los valores leídos y los comparemos al final. El código completo queda:

```
1  int suma = 0;
2  int longitud = 0;
3  int v;
4  cin >> v;
5  while (v != -1) {
6      suma += v;
7      ++longitud;
8      cin >> v;
9  }
10
```

```
11 if (suma == longitud) {
12     cout << "Justo\n";
13 }
14 else if (suma > longitud) {
15     cout << "Suerte\n";
16 }
17 else {
18     cout << "Timo\n";
19 }
```

Para valientes

C++ proporciona el *operador coma* (`,`). De forma general, un operador es una función que recibe, normalmente, dos valores, hace un cálculo con ellos y genera un resultado. Por ejemplo el *operador suma* (`+`) recibe dos valores y devuelve su suma.
El *operador coma* recibe también dos valores, *ignora el primero* y genera como resultado el valor del segundo. Se ve más claro con un ejemplo:

```
a = 1, 3;
```

El código anterior tiene, a la derecha del símbolo `=`, una expresión que utiliza el operador coma. Este recibe dos valores (el `1` y el `3`) y devuelve como resultado de "su cálculo" el segundo. Eso significa que lo que se asignará realmente a la variable `a` será el `3`.
En principio esto no parece tener mucha utilidad y, para muchos, su existencia genera más problemas que ventajas porque un código como este:

```
float f = 1,3;
```

está asignando a `f` el valor `3`, y no el `1.3` como podría pensarse.
La utilidad del operador coma surge si lo que se pone como primer operando *tiene efectos secundarios*, es decir si hace algo más allá de calcular un valor. En general, crear expresiones que tengan efectos secundarios no se considera una buena práctica de programación. Pero en C/C++, muchas sentencias *resultan ser expresiones* que devuelven un valor, aunque luego no se utilice. En particular, `cin` *devuelve un valor* que normalmente se descarta. Esto nos permite evitar la doble lectura del código anterior usando el operador coma dentro de la condición del bucle:

```
while (cin >> v, v != -1) {
    suma += v;
    ++longitud;
}
```

El operador coma "calcula" el lado izquierdo (`cin >> v`) que tiene como efecto secundario la lectura del valor de `v`. Luego compara el valor que acaba de leerse con `-1` y es esa comparación lo que queda como resultado del conjunto, y lo que usará el `while` para decidir si da o no otra vuelta.

165 Número hyperpar

Categorías

- Bucles simples
- Extracción de dígitos
- Esquema de la entrada con marca de fin

Resumen del enunciado

Cada caso de prueba es un número y hay que decir si es o no *hyperpar*, es decir si todos sus dígitos son pares o no.

Solución

En este problema tenemos que leer un número y *extraer sus dígitos*. Para cada uno de sus dígitos, hay que mirar si es o no par. Si no lo es, sabemos que el número no es *hyperpar* y podemos dejar de mirar dígitos.

Para resolverlo, necesitamos un bucle que procese cada uno de los dígitos del número original. Dado que no sabemos de antemano cuántos dígitos tiene el número, el bucle será un `while`.

Para extraer los dígitos de un número lo habitual es utilizar el *operador módulo*. Al calcular el resto de dividir un número entre 10 nos estaremos quedando con el dígito de más a la derecha (conocido como el "menos significativo", por ser el de menor valor). Una vez procesado el dígito, se divide el número completo entre 10 (con división entera) para deshacernos del dígito procesado, y repetimos. El bucle lo terminamos cuando "nos quedemos sin número", es decir cuando el resultado de la división sea 0.

Por ejemplo, si queremos *sumar* todos los dígitos de un número n haríamos algo así:

```
unsigned int suma = 0;
while (n != 0) {
    suma += n % 10;
    n /= 10;
}
```

En el código anterior *recorremos* todos los dígitos hasta consumir el número completo. Para saber si un número es *hyperpar* lo que tenemos que hacer es algo parecido pero ahora en cuanto nos encontremos un dígito que sea impar *podemos terminar el bucle* porque sabremos que el número *no* es *hyperpar*. Estamos por tanto *buscando el primer dígito impar* y solo queremos seguir dando vueltas al bucle si *no* lo hemos encontrado. La condición del bucle será un poco más complicada porque para seguir dando vueltas se tienen que cumplir dos condiciones:

- No hemos consumido el número entero (no es 0)
- El siguiente dígito es par (no hemos encontrado un dígito impar)

Esto significa, además, que el bucle *puede terminar por dos motivos*:

- Hemos consumido el número completamente
- Hemos encontrado el primer dígito impar

Dependiendo de la razón por la que haya terminado el bucle, el número será o no *hyperpar*. Al final, el código queda:

```
int n;
cin >> n;
while((n != 0) && (n % 2 == 0))
    n /= 10;

if (n == 0)
    // Hemos consumido el número sin encontrar un dígito impar.
    // Es hyperpar.
    cout << "SI\n";
else
    // Hemos salido del bucle por encontrar un dígito impar.
    // No es hyperpar.
    cout << "NO\n";
```

Información

Hay otras muchas formas de resolver el problema escribiendo código equivalente. Es habitual ver soluciones que utilizan variables *booleanas* para hacer comprobaciones dentro del bucle, que utilizan `if`, `break`'s... La solución propuesta es mucho más concisa y elegante.

Cuidado

El enunciado del problema dice que la ejecución debe terminar si el valor leído es negativo. El código anterior debe por tanto ser adaptado al *esquema de la entrada* que utiliza una *marca de fin*.

Hazlo por tu cuenta

El enunciado *permite* que la entrada sea `0`. El número 0 es *hyperpar* porque todos sus dígitos son pares. Estamos aprovechando esto (quizá sin habernos dado cuenta) en la condición del bucle y su condición final. Resuelve el problema alternativo de decir si un número es *hyperimpar*, es decir si tiene todos sus dígitos impares. Tendrás que tener cuidado con el `0` como valor válido en la entrada.

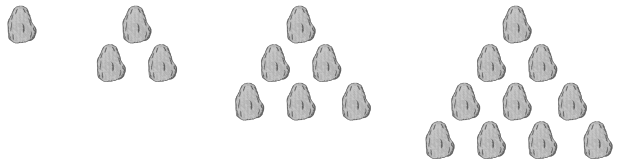
170 Triángulos con piedras

Categorías

- Bucles simples
- Matemáticas

Resumen del enunciado

Tenemos un conjunto de piedras (hasta 250.000.000) y las organizamos formando *un triángulo relleno* poniendo filas sucesivas de 1, 2, 3... piedras.



Dado el número de piedras que tenemos, hay que decir la longitud del lado del mayor triángulo que podemos formar, y cuántas piedras nos sobran.

Solución

Para resolverlo, la idea que podemos usar es *tantear*, simulando que vamos creando el triángulo poniendo más y más filas hasta que nos quedamos sin piedras suficientes para poner la siguiente. Cuando eso ocurra, paramos y escribimos el resultado.

Como no sabemos cuántas filas vamos a poder poner, necesitamos un bucle `while`. Entraremos en él si nos quedan piedras suficientes para la siguiente fila. Dentro, restamos las piedras consumidas con la fila que ponemos en esa vuelta, e incrementamos la longitud de la fila en preparación para la vuelta siguiente.

```
unsigned int numPiedrasDisponibles;
unsigned int longSigFila = 1;

cin >> numPiedrasDisponibles;
while(numPiedrasDisponibles >= longSigFila) {
    // Tenemos piedras suficientes para una fila más.  Las usamos.
    numPiedrasDisponibles -= longSigFila;
    ++longSigFila;
}
// No hemos podido poner la fila de longitud longSigFila.
// La última que hemos puesto (y la longitud del lado de
// nuestro triángulo) tiene una menos.
cout << longSigFila - 1 << ' ' << numPiedrasDisponibles << '\n';
```

Cuidado

El enunciado del problema dice que la ejecución debe terminar si el valor leído es 0. El código anterior debe por tanto ser adaptado al *esquema de la entrada* que utiliza una *marca de fin*.

Cuidado

¡No olvides escribir el espacio separando los dos números de la salida! Si no, sufrirás *Presentation Error*.

Hazlo por tu cuenta

Si tuvieras que resolver este problema con lápiz y papel para un número de piedras muy alto, el proceso de tanteo sería muy aburrido. Hay una forma mejor de hacerlo aprovechando la fórmula de los *números triangulares* o, con otro nombre, la fórmula de la suma de una sucesión aritmética. Esto permite calcular el resultado con expresiones en lugar de con un bucle. Resuelve el problema con esta idea y comprueba que el tiempo de ejecución es muchísimo menor.

119 Escudos del ejército romano

Categorías

- Bucles simples
- Matemáticas

Resumen del enunciado

Somos los responsables de un grupo de legionarios romanos que tenemos que colocar en formación. Hemos decidido que los organizaremos en varios grupos cuadrados (de $n \cdot n$ legionarios) y para eso intentaremos, en cada momento, crear el agrupamiento más grande posible con los legionarios que nos queden. Si tras crear un cuadrado nos siguen sobrando legionarios por colocar, repetiremos el proceso.

Cada formación de $n \cdot n$ legionarios se protege con escudos, colocando un escudo sobre la cabeza de cada legionario, y uno en cada flanco de la formación. Hay que decidir cuántos escudos necesitamos en total para el número de legionarios que tenemos.

Solución

Para resolver el problema necesitamos decidir el agrupamiento que haremos de nuestros legionarios. La forma en la que lo haremos se conoce como *solución voraz* porque una vez que colocamos un conjunto de legionarios en un grupo, no nos replantearemos si habría sido mejor colocarlos de otra forma.

Si tenemos n legionarios, lo que hay que hacer es crear el *cuadrado* más grande posible. Para eso, podemos calcular *la raíz cuadrada* del número, ignorando los decimales. Eso nos da el mayor número cuyo cuadrado es menor o igual que el número n que es precisamente lo que estamos buscando. Una vez que tenemos ese valor, "consumimos" a los legionarios que crearán esa formación, sumamos los escudos que necesitan y repetimos el proceso. Paramos cuando hayamos colocado a todos los legionarios.

Para programarlo necesitamos un bucle. Como no sabemos cuántas vueltas necesitamos dar (dependerá de cuándo hayamos colocado a todos los legionarios) necesitamos un `while`:

```
unsigned int numLegionarios;
unsigned int numEscudos = 0;

cin >> numLegionarios;
while(numLegionarios > 0) {
    // sqrt() requiere #include <cmath>
    unsigned int sigTam = sqrt(numLegionarios);
    // Colocamos a sigTam*sigTam legionarios, que nos
    // quitamos.
    numLegionarios -= sigTam * sigTam;
    // Necesitan escudos para arriba, y para los laterales,
    // es decir el "área" del cuadrado, y su perímetro.
    numEscudos += sigTam * sigTam + 4 * sigTam;
} // while

cout << numEscudos << "\n";
```

Cuidado

El enunciado del problema dice que la ejecución debe terminar si el valor leído es 0. El código anterior debe por tanto ser adaptado al *esquema de la entrada* que utiliza una *marca de fin*.

205 Números de Lychrel ★

Categorías

- Bucles simples
- Funciones
- Extracción de dígitos
- Capicúas

Resumen del enunciado

Dado un número natural, podemos conseguir un número nuevo, mayor que él, si le damos la vuelta y sumamos el valor al número original.

Este proceso podemos repetirlo una y otra vez hasta alcanzar un número capicúa. Para algunos números no se ha alcanzado un capicúa después de mucho probar, y a esos números se les conoce como *números de Lychrel*.

Cada caso de prueba es un número n y hay que decir cuántas vueltas hay que repetir el proceso para alcanzar un capicúa, y qué capicúa es, o escribir `Lychrel?` si se supera el número 1.000.000.000 sin haber encontrado un capicúa.

Solución

Para resolver el problema necesitamos *un bucle* que repita el proceso de dar la vuelta al número, y sumarlo al valor original. El bucle terminará cuando la suma sea capicúa. Dado que no sabemos en cuántas vueltas ocurrirá eso (ni si llegará a ocurrir) necesitamos un `while`.

En realidad, hay un detalle importante y es que *siempre hay que dar al menos una vuelta*. En el enunciado aparece como ejemplo el número 4994 que es capicúa. La salida para ese caso *no* es él mismo diciendo que no damos ninguna vuelta (por ser capicúa desde el principio) sino que resulta ser `Lychrel?` porque tras aplicar sobre él el proceso varias veces no se llega a ningún capicúa nuevo.

Si queremos un bucle tipo `while` que garantice que se ejecuta al menos una vez (es decir la condición se comprueba tras la primera vuelta) lo adecuado es utilizar el bucle `do-while`.

El bucle debe terminar si alcanzamos un número capicúa, o si hemos llegado a un número que supere un valor máximo permitido a partir del cual suponemos que el número es de Lychrel. Eso significa que nuestra condición tendrá que tener en cuenta ambas comprobaciones.

En *pseudocódigo*, lo que queremos programar es:

```
numVueltas = 0
leer n
do {
    sumar a n el propio n dado la vuelta
    incrementar numVueltas
} while(n menor que el limite y no es capicua)
```

Para programarlo, tenemos que buscar la forma de *dar la vuelta a un número*, es decir convertir, por ejemplo, el número 135 en el 531. Para hacer esto la idea es ir *extrayendo*

los dígitos del número original desde la derecha, y *añadir* los dígitos extraídos a un número nuevo, que vamos construyendo. El número original lo vamos dividiendo por 10 para retirar los dígitos que vamos "extrayendo", y el número nuevo lo vamos multiplicando por 10 para "empujar" los dígitos ya añadidos hacia la derecha antes de sumar cada nuevo dígito.

Para facilitar el uso del código, vamos a programar *una función* que reciba como parámetro el número a procesar, y devuelva el valor dado la vuelta. El código queda:

```
unsigned int darLaVuelta(unsigned int i) {

    unsigned int resultado = 0;

    while(i != 0) {
        resultado *= 10;
        resultado += i % 10;
        i /= 10;
    }

    return resultado;

} // darLaVuelta
```

Cuidado

En programación competitiva siempre hay que tener cuidado con los límites. En la mayoría de los ordenadores actuales, los enteros sin signo (`unsigned int`) usados en el código anterior pueden guardar valores hasta $2^{32} - 1$, es decir 4.294.967.295. Si utilizas el tipo `int`, el límite se reduce aproximadamente a la mitad, a $2^{31} - 1$ o 2.147.483.647.

En ambos casos se puede representar con bastante margen el número 1.000.000.009. Sin embargo, ese mismo número dado la vuelta se convierte en el 9.000.000.001 que ni un `int` ni un `unsigned int` pueden guardar. Afortunadamente, el enunciado nos pone un tope máximo a los números que tendremos que dar la vuelta, que no superarán el 1.000.000.000. Cualquier número menor o igual que él, dado la vuelta, sigue entrando en nuestros tipos de datos básicos, y no tendremos que preocuparnos de los límites de la representación.

En el *pseudocódigo* de la solución del problema también aparece la necesidad de saber si un número es *capicúa*. Lo será si al ser escrito en sentido opuesto (de derecha a izquierda) el número es el mismo. Para simplificar la programación, haremos una *función* que reciba un número y devuelva un *booleano* indicando si es o no capicúa. Curiosamente, para programarlo podemos apoyarnos en la función `darLaVuelta()` anterior:

```
bool capicua(unsigned int i) {

    return i == darLaVuelta(i);

}
```

Equipados con estas dos funciones, programar la solución del problema resulta ahora más sencillo:

```cpp
unsigned int num;
unsigned int numVueltas = 0;

cin >> num;

do {
    num += darLaVuelta(num);
    ++numVueltas;
} while((num <= 1000000000) && !capicua(num));

if (num > 1000000000)
    cout << "Lychrel?\n";
else
    cout << numVueltas << " " << num << "\n";
```

Información

Podría ocurrir que saliéramos del bucle por superar el umbral (1.000.000.000) y resultara que el número final fuera capicúa. En ese caso sabríamos que el número *no* es de Lychrel y podríamos escribir el número de vueltas dadas. No obstante, el enunciado nos dice que si se supera el límite hay que escribir `Lychrel?` incondicionalmente, por lo que no se hace la comprobación adicional.

190 Dividir factoriales ★

Categorías

- Bucles
- Matemáticas
- Límite de la representación

Resumen del enunciado

Nos dan dos números, n y m y hay que calcular el valor de $\frac{n!}{m!}$, donde $i!$ representa *el factorial de i*, es decir la multiplicación de todos los números entre 1 e i. Nos garantizan que *el resultado* será siempre menor que 2^{63} y que el numerador (m) será siempre mayor o igual que el denominador (n).

Solución

Calcular el *factorial* de un número es algo relativamente fácil con un bucle. Como nos piden la división de un factorial entre otro, es tentador calcular el factorial del numerador, el del denominador, y hacer la división. Podríamos incluso programar una función para calcular el factorial y llamarla dos veces.

El problema *son los límites*. El enunciado nos garantiza que *el resultado* será menor que 2^{63} pero no nos dice nada del valor de cada factorial, y es peligroso porque el factorial *crece muy deprisa*. El enunciado podría pedirnos la división del factorial de 1000 entre el factorial de 998 y calcular cualquiera de los dos ocasiona que desbordemos la representación de los enteros.

Para resolverlo, hay que pensar cómo lo haríamos con lápiz y papel, intentando ahorrar todo el trabajo posible. Siguiendo con el ejemplo anterior:

$$\frac{1000!}{998!} = \frac{1 \cdot 2 \cdot \ldots \cdot 998 \cdot 999 \cdot 1000}{1 \cdot 2 \cdot \ldots \cdot 998} = 999 \cdot 1000 = 999.000$$

La idea de fondo es que el denominador sirve para simplificar el numerador, y al final lo que tenemos que hacer es multiplicar entre sí los números desde *m+1* hasta n y escribir el resultado. Para eso necesitamos un bucle. El número de vueltas es conocido de antemano (dados n y m) por lo que usaremos un `for`.

Un último detalle importante es que la salida puede alcanzar números muy altos, hasta casi 2^{63}. Valores tan altos *no* entran en un entero habitual de los lenguajes de programación, por lo que tenemos que utilizar una representación más amplia. En C++ usar `long long` (o `unsigned long long`) es suficiente:

```
unsigned int num, den;
unsigned long long resultado = 1;
cin >> num >> den;

for (unsigned int i = den+1; i <= num; ++i)
    resultado *= i;
cout << resultado << "\n";
```

Información

Inicializar `resultado` a 1 es importante, porque es el *neutro de la multiplicación*. Podríamos inicializarlo con `num` o con `den+1` y ahorrarnos una vuelta del bucle, pero entonces tendríamos problemas para casos donde el numerador y el denominador *tienen el mismo valor* (algo que permite el enunciado) y que, en el código anterior, suponen que el bucle *no dé ninguna vuelta*.

Cuidado

El enunciado del problema dice que la ejecución debe terminar cuando el numerador sea menor que el denominador. El código anterior debe por tanto ser adaptado al *esquema de la entrada* que utiliza una *marca de fin*.

221 Entrando al cine ★

Categorías

- Bucles simples
- Secuencias

Resumen del enunciado

Cada caso de prueba es una secuencia de hasta 10.000 números. Hay que escribir `SI` si la secuencia se puede separar en una primera *subsecuencia* de números pares seguida de otra de números impares, y `NO` en otro caso. Si se puede, entonces hay que indicar también la longitud de la *subsecuencia* de números pares.

Se debe tener en cuenta que alguna de las dos *subsecuencias* podría ser vacía. Es decir, la entrada podría estar compuesta únicamente de, por ejemplo, números impares y se consideraría válida. En ese caso habría que escribir `SI 0`.

Solución

Hay que leer una secuencia cuya longitud se conoce de antemano, por lo que lo natural es utilizar un bucle `for`. La dificultad es que dentro tendremos que mirar una cosa u otra dependiendo del *estado*. Al principio estaremos en la primera parte de la secuencia, saltando números pares y buscando el primer impar. Si ya hemos encontrado un impar, estaremos saltándonos los impares buscando un potencial par que rompa la estructura deseada del conjunto.

Una última cosa importante es que si sabemos que el caso de prueba *no* está formado por una primera parte de números pares y una segunda de impares (porque ha aparecido un nuevo par) *hay que leer todos los números* que vayan después en lugar de contestar inmediatamente. De otra forma, dejaríamos números sin leer, que nos encontraríamos al leer el caso de prueba siguiente y desincronizaríamos todo el procesamiento.

Para saber en qué situación estamos dentro del `for` usaremos variables *booleanas* que nos digan dónde estamos.

```
unsigned int n;
unsigned int v;
unsigned int numPares;
bool imparEncontrado;
bool secuenciaCorrecta;

cin >> n;

imparEncontrado = false;
numPares = 0;
secuenciaCorrecta = true; // Mientras no se demuestre lo contrario

for (unsigned int i = 0; i < n; ++i) {

    cin >> v;
    if (imparEncontrado) {
        // Estamos en la sección de números impares.
```

```
18          // Todos deberían ser impares ya.
19          // La secuencia será correcta si antes lo era Y
20          // el número leído es impar.  En cualquier otro caso
21          // no lo estará.
22          secuenciaCorrecta &= ((v % 2) == 1);
23          // También valdría:
24          // if (v % 2 == 0) secuenciaCorrecta = false;
25      }
26      else {
27          // Estábamos en la parte de los pares.  ¿Seguimos en ella?
28          if (v % 2 == 0)
29              // Sí, y además tenemos un par más.
30              ++numPares;
31          else
32              // No.  Anotamos el cambio de sección.
33              imparEncontrado = true;
34      } // if-else imparEncontrado
35  } // for
36
37  if (secuenciaCorrecta)
38      cout << "SI " << numPares << "\n";
39  else
40      cout << "NO\n";
```