Entrenamiento OIE 2023 Nivel Inicial

Sesión 2: Condicionales

En esta segunda sesión de entrenamiento vamos a poner en práctica los *condicionales*. En la primera sesión, el código que escribimos era *secuencial*: definíamos algunas variables, leíamos del teclado, hacíamos cálculos y escribíamos el resultado. Todo se ejecutaba de principio a fin sin ningún tipo de *bifurcación*. Con los *condicionales* añadimos la posibilidad de *tomar decisiones* en función del estado de las variables y que el programa ejecute unas instrucciones u otras dependiendo de ellas.

El código condicional se consigue principalmente con la instrucción `if` (con o sin `else`) y será la que más utilicemos en los ejercicios de la sesión. En algunos casos también se puede hacer uso de `switch` o incluso del *operador ternario* (`?:`).

Como en la primera sesión, todos los ejercicios están disponibles en *¡Acepta el reto!* Algunos los marcamos con una estrella (★) por tener un nivel de dificultad algo mayor que el resto, normalmente no desde el punto de vista de programación sino de descubrir cómo se resuelven. En cualquier caso, para todos los problemas, la recomendación siempre es la misma: piensa antes de programar las soluciones y no uses *prueba y error*. ¡Primero resuelve el problema y luego escribe el código!

Los problemas de esta sesión de entrenamiento son:

- 217 ¿Qué lado de la calle?
- 313 Fin de mes
- 355 Gregorio XIII
- 427 Yo soy tu...
- 180 Triángulos
- 304 División euclídea ★
- 114 Último dígito del factorial ★
- 241 Me quiere, no me quiere ★
- 397 ¿Es múltiplo de 3? ★

217 ¿Qué lado de la calle?

Categorías

- Condicionales
- Esquema de la entrada con marca de fin

Resumen del enunciado

Justificado por la ambientación, cada caso de prueba es un número y hay que escribir `IZQUIERDA` si el número es *impar* o `DERECHA` si es *par*.

Solución

En este problema tenemos que hacer una cosa u otra dependiendo de si el número leído cumple o no una condición. Eso significa que tendremos *código condicional* que debe o no ejecutarse dependiendo de las circunstancias.

Dado que tenemos que o bien hacer una cosa o bien hacer otra, necesitaremos un `if` con su cláusula `else`. Lo que tenemos es que averiguar qué condición poner en el `if` para que, si es cierta, se ejecute el "caso sí", y si es falsa se ejecute el "caso no".

En la solución que te proponemos, hemos decidido que entraremos en el "caso sí" cuando el número leído sea *impar* (escribiremos `IZQUIERDA`). En *pseudocódigo* el código tendrá la siguiente estructura:

```
leer numPortal
si numPortal es impar entonces
    escribir "IZQUIERDA"
si no
    escribir "DERECHA"
```

Información

Pseudocódigo es una "descripción de alto nivel compacta e informal" de un algoritmo. Normalmente tiene el aspecto de ser un programa, pero en un "lenguaje de programación" inexistente. El objetivo es que se entienda la idea general (sus pasos, bifurcaciones, repeticiones, etcétera) pudiendo ahorrarse los detalles que no interesen en ese momento.

Tenemos que buscar la forma de escribir una *expresión booleana* que sea cierta si un número es impar, y falsa si es par. Para eso hay que pensar en las características que determinan la *paridad* de un número. Afortunadamente, es fácil recordar que los números impares son aquellos que, al ser divididos por 2, dan como resto 1. Por tanto, lo que hacemos es calcular el resto de dividir el número leído por 2, y si es 1 será impar. Al final nos queda:

```
1 // Solución parcial.  Sigue leyendo.
2 cin >> numPortal;
3 if (numPortal % 2 == 1) {
4     cout << "IZQUIERDA\n";
5 }
6 else {
```

```
    cout << "DERECHA\n";
}
```

Información

En C++, la condición del `if` debe ir entre paréntesis. Fíjate además en que el operador módulo (`%`) tiene *más precedencia* que el de comparación (`==`) de modo que primero se calcula el resto de la división y luego se compara. No es necesario poner paréntesis adicionales.

Cuidado

La aparición de dos signos de igual seguidos (`==`) *no* es un error. Es el *operador de comparación*, no una asignación. Construye una *expresión booleana* (es decir una expresión que se evalúa a cierto o a falso) en la que se comparan los dos valores de cada lado.

El código anterior no está completo. Este problema utiliza un *esquema de la entrada* distinto al que usaron los problemas de la sesión 1 del entrenamiento. El enunciado dice que hay que procesar casos de prueba hasta que encontremos uno especial que actúa como "marca de fin", en este caso un `0`. Tenemos que identificarlo y, cuando llegue, terminar la ejecución.

Para detectarlo, *antes* de mirar si el número leído es impar tenemos primero que mirar si es o no el `0` que marca el final. Si lo es, habrá que parar la ejecución, y si no, continuar con la comparación que ya tenemos.

Utilizando el *esqueleto de la solución* para este esquema de la entrada, el código completo queda:

```
#include <iostream>
using namespace std;

bool casoDePrueba() {

    unsigned int numPortal;

    cin >> numPortal;

    if (numPortal == 0) {
        return false;
    }
    else {
        if (numPortal % 2 == 1) {
            cout << "IZQUIERDA\n";
        }
        else {
            cout << "DERECHA\n";
        }
        return true;
    } // if-else caso especial de fin

} // casoDePrueba
```

```
24
25 //----------------------------
26
27 int main() {
28
29     while (casoDePrueba()) {
30     }
31
32     return 0;
33
34 } // main
```

La parte importante es el interior de `casoDePrueba()` (de la línea 6 a la 21). Si el número leído es `0`, se ejecuta la línea del `return false`. En este programa, eso ocasiona que la ejecución termine, aunque la razón por la que ocurre está más allá del objetivo de esta sesión de entrenamiento. Además, el "caso no" termina con un `return true` que sirve para indicar que el caso procesado no es el especial y el programa 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.

Hazlo por tu cuenta

En la solución propuesta miramos si el número es impar. También puede resolverse haciendo la condición mirando si el número es *par*. Inténtalo.

Para valientes

En C++, cualquier expresión se convierte automáticamente a un valor *booleano* (cierto o falso) dependiendo de su valor. Un 0 se considera falso (`false`) y cualquier otra cosa se considera cierta (`true`). Eso significa que el resultado de calcular el resto de dividir el número por 2 se puede reinterpretar como un *booleano* si no se compara con nada. Si el resto es 1 la expresión se considerará `true` y si es 0 se considerará `false`.
El resultado es que el código podríamos escribirlo así:

```
1 ...
2 if (numPortal % 2) {
3     cout << "IZQUIERDA\n";
4 }
5 else {
6     cout << "DERECHA\n";
7 }
8 ...
```

Para valientes

Si representamos un número entero *en binario*, el *bit menos significativo* indica la paridad. Si es 1 entonces el número es impar, y si es 0 es par. Al ser un *bit*, no puede tener más valores.
Dado un número, podemos quedarnos con su último bit si hacemos la *and a nivel de bits* (`&`). Mezclando esto con el código anterior, la condición del bucle podría reescribirse por:

```
...
if (numPortal & 1) {
    cout << "IZQUIERDA\n";
}
else {
    cout << "DERECHA\n";
}
...
```

313 Fin de mes

Categorías

- Condicionales

Resumen del enunciado

Cada caso de prueba son dos números. Si su suma es mayor o igual que 0 hay que escribir "SI" y si no lo es se debe escribir "NO".

Solución

Para poder resolver el problema necesitamos que el código *tome una decisión* en función de una comparación, para lo que necesitamos utilizar un `if`. Queremos comparar dos números y saber si uno es *mayor o igual que* el otro (0). En matemáticas, para eso se utiliza el símbolo $\geq$, que no está disponible en los teclados. Los lenguajes de programación utilizan dos símbolos para representar este operador, `>=` (sin espacio entre ellos).

En la condición del `if` tenemos que comparar la suma de dos números con el valor 0. Podemos calcular la suma en una variable auxiliar y comparar su valor con el 0, o podemos realizar la suma directamente en la condición.

```
1  int saldo, cambio;
2
3  cin >> saldo >> cambio;
4
5  if (saldo + cambio >= 0)
6      cout << "SI\n";
7  else
8      cout << "NO\n";
```

Los operadores aritméticos (como el de suma) tienen mayor precedencia que los de comparación (como el de menor o igual) por lo que se ejecutarán antes y no es necesario añadir paréntesis adicionales.

Hazlo por tu cuenta

Supón que no conoces la forma de escribir la comparación "menor o igual". ¿Cómo resolverías el problema? Piensa al menos en dos posibilidades distintas.

355 Gregorio XIII

Categorías

- Condicionales
- Operadores lógicos

Resumen del enunciado

Dado un año, se pide decir cuántos días tuvo (o tendrá) su mes de febrero. Debe usarse la regla de los años bisiestos del llamado "calendario gregoriano" (el actual) en el que un año es bisiesto si es divisible por 4, salvo que sea divisible por 100 en cuyo caso debe también serlo por 400.

Solución

El enunciado detalla la historia y necesidad del *calendario gregoriano* que redefinió los años bisiestos. La mayor parte de la gente piensa que los años bisiestos son aquellos cuyo número es divisible por 4 (2020, 2024, 2028, ...) pero eso es una verdad a medias. Esa regla tiene una excepción que, a su vez, tiene una excepción. En particular, todos los números que son divisibles por 100 lo son también por 4 y por tanto deberían ser años bisiestos. Sin embargo, la excepción nos dice que esos años *no* son bisiestos, *salvo* que también sean divisibles por 400 (esa es la excepción de la excepción) que entonces sí lo son. Así, por ejemplo el año 1900 o el 2100 no son bisiestos porque son divisibles por 100, pero el 2000 o el 2400 sí porque, pese a ser divisibles por 100, también lo son por 400.

Para realizar un programa que identifique los años bisiestos necesitamos incorporar la regla general, y también las excepciones. Una posibilidad sería utilizar varios `if` encadenados siguiendo un *pseudocódigo* de este estilo:

```
si el numero no es divisible por 4
    // Caso fácil
    No es bisiesto
si no
    // En principio lo es, pero hay que confirmar primero que
    // no es divisible por 100
    si el numero no es divisible por 100
        // Definitivamente lo es
        Es bisiesto
    si no
        // Uy, divisible por 100.  Estamos en la excepción.
        // Pero podemos estar en la excepción de la excepción.
        si el numero es divisible por 400
            // ¡La excepción de la excepción!
            Es bisiesto
        si no
            No es bisiesto
```

Aunque una solución de este estilo debería funcionar, resulta muy largo de escribir. Además tenemos repetida varias veces la decisión (las líneas `No es bisiesto` o `Es bisiesto`). Para mejorarlo, podemos utilizar *operadores lógicos* que mezclen varias condiciones *booleanas*, es decir los operadores `and` (`&&`) y `or` (`||`).

Reescribir el código anterior con *una* única expresión *booleana* compleja no es sencillo. Hay a personas a las que les resulta de ayuda "decirlo en español" usando "y" y "o" para unir las distintas condiciones que se deben cumplir. Este proceso hay que hacerlo, no obstante, con cuidado, porque en ocasiones el lenguaje natural puede jugar malas pasadas. Por ejemplo la frase "me duele la garganta cuando grito y cuando cojo frío" no significa que tengan que ocurrir *las dos cosas* (gritar *y* coger frío) para que me duela la garganta, sino que es suficiente con una de las dos. El sentido común convierte el "y" en un "o", que es realmente el conector correcto.

Otra alternativa es pensar en "conjuntos". Tenemos todos los posibles elementos (en este caso los años) y hay que seleccionar aquellos que nos interesen dando condiciones. Los elementos que cumplan una condición sencilla (por ejemplo ser múltiplo de 4) forman un subconjunto del total. Esos subconjuntos se pueden unir, que es equivalente a una "o" lógica de las condiciones de los dos subconjuntos, o se pueden hacer intersecciones, que es equivalente a una "y" lógica.

Sea como sea, en este problema nosotros empezaremos *por el final*, es decir por la excepción de la excepción. Los años que son divisibles por 400 lo son también por 4, por tanto *todos* los años divisibles por 400 son bisiestos. Ya tenemos un "subconjunto" del total de años para los que hay que escribir un "29".

Pero ¡nos quedan muchos! De todos los demás, serán bisiestos los que sean divisibles por 4 *y* no sean divisibles por 100. En esta regla ya *no* hay que meter la segunda excepción, porque esos años ya están considerados en el primer subconjunto. El año 2000 no está incluído en este bloque (por ser divisible por 4 y por 100), pero lo está en el primero.

La frase completa, escrita en lenguaje natural, quedaría "un año es bisiesto si es divisible por 400 **o** es divisible por 4 **y no** es divisible por 100". Ten en cuenta que en esta frase las "pausas" son importantes porque nos indican qué está agrupado con qué. Las dos frases siguientes *no* significan lo mismo:

- Un año es bisiesto si es divisible por 400 **o si** es divisible por 4 y no es divisible por 100
- Un año es bisiesto si es divisible por 400 o por 4 **y si** [además] es divisible por 100

De las dos, la correcta *es la primera*. Desde el punto de vista de la *expresión booleana* se diferencian en donde *van los paréntesis*:

- (Divisible por 400) *o* (divisible por 4 *y* no divisible por 100)
- (Divisible por 400 *o* divisible por 4) *y* (no divisible por 100)

La segunda, de hecho, ni siquiera tiene mucho sentido porque, como hemos dicho, todos los números divisibles por 400 son también divisibles por 4. Eso significa que el primer paréntesis da información sobrante y la comprobación de división por 400 se podría eliminar. Y ¡no solo por eso! Si algo es divisible por 400 irremediablemente también lo es por 100, de modo que la condición "divisible por 400 y no divisible por 100" no es cierta nunca.

Tras toda esta reflexión, escribir la solución del problema en un lenguaje de programación debería ya ser fácil.

```
unsigned int anyo;

cin >> anyo;

if ((anyo % 400 == 0) || ((anyo % 4 == 0) && (anyo % 100 != 0))) {
```

```
6       cout << "29\n";
7   }
8   else {
9       cout << "28\n";
10  }
```

Para evitar ambigüedad, se han añadido paréntesis para que quede claro el orden deseado de las operaciones. En realidad, debido a la precedencia de los operadores, ninguno de ellos es necesario (salvo, naturalmente, los que encierran a la condición completa en el `if`). El `or` (`||`) tiene *menos* prioridad que el `and` (`&&`) y por tanto si no hubiera paréntesis las condiciones se agruparían tal y como las queremos.

Hazlo por tu cuenta

Si no lo conoces, investiga sobre el *operador ternario* y resuelve el problema con él.

427 Yo soy tu...

Categorías

- Condicionales
- Comparación de cadenas

Resumen del enunciado

Cada caso de prueba son dos palabras, en concreto un nombre y un parentesco. Hay que escribir la frase "<nombre>, yo soy tu <parentesco>", salvo que sean "Luke" y "padre", en cuyo caso la frase sería "Luke, yo soy tu padre" que debe ser sustituída por "TOP SECRET".

Solución

En este problema lo que tenemos que hacer es leer dos cadenas y mirar si cada una de ellas es una específica ("Luke" y "padre"). Dependiendo del resultado de la comparación, tendremos que hacer una cosa u otra. Eso nos lleva a que necesitamos una *bifurcación* en el código, para realizar una tarea u otra, algo que conseguimos con un `if`.

Para saber si una cadena es igual a otra en C++, podemos utilizar el mismo operador de comparación que se utiliza para comparar números (`==`). Dado que necesitamos comparar *dos* cadenas y que ambas coincidan con una específica para escribir el caso especial ("TOP SECRET") necesitamos usar una `and` (`&&`) en la condición, para que esta sea cierta únicamente si *ambas* comparaciones lo son.

```
string nombre, parentesco;
cin >> nombre >> parentesco;

if (nombre == "Luke" && parentesco == "padre") {
    cout << "TOP SECRET\n";
}
else {
    cout << nombre << ", " << "yo soy tu " << parentesco << '\n';
}
```

Cuidado

El texto escrito debe coincidir *exactamente* con el solicitado en el enunciado, o recibirás un veredicto de *Wrong Answer* (respuesta incorrecta) o, en el mejor de los casos, *Presentation Error* (error de presentación). En particular, no debes olvidar la coma tras el nombre, el `\n` final o los espacios de separación.

Cuidado

La lectura de cadenas con `cin >> variable` lee hasta el primer separador (por ejemplo espacio o salto de línea). El enunciado nos asegura que ni el nombre ni el parentesco tendrán espacios, lo que evita complicaciones adicionales.

Peligro

En C++ es más cómodo leer cadenas utilizando el tipo `string` tal y como hemos hecho. Por herencia de C, también es posible leer en arrays de caracteres (`char nombre[101]`), especialmente si, como en este problema, el enunciado nos pone un límite al tamaño de la entrada. En ese caso *no* podemos comparar con el operador `==`, aunque las razones (al igual que el concepto de *array*) queda fuera del objetivo de esta sesión de entrenamiento.

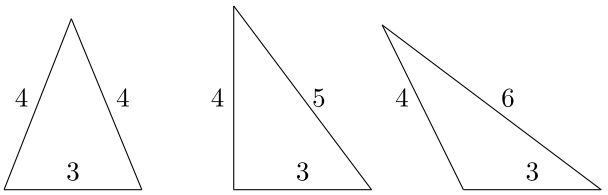
180 Triángulos

Categorías

- Condicionales
- Geometría

Resumen del enunciado

Nos dan las longitudes de los tres lados de un hipotético triángulo y hay que decir si es acutángulo, rectángulo, obtusángulo o es imposible formar un triángulo con esos tres segmentos.



Acutángulo Rectángulo Obtusángulo

Solución

Para programar la solución lo primero es saber cómo resolver el problema con lápiz y papel. Si nos dan la longitud de tres segmentos, ¿cómo podemos saber el tipo de triángulo que forman o incluso si pueden formar uno?

Para contestar a la última pregunta lo más fácil es imaginarse que tenemos un segmento muy largo (por ejemplo de longitud 100) y dos extremadamente cortos (por ejemplo de longitud 1). En ese caso, si colocamos cada uno de los dos cortos a un lado del segmento largo es fácil visualizar que no podrán llegar a tocarse, y será imposible cerrar el triángulo. Esto nos da la idea de que para poder crear un triángulo, el segmento *más largo* debe ser *más corto* que la suma de los otros dos, para que puedan alcanzarse.

Una vez que sabemos que el triángulo puede formarse, necesitamos identificar su tipo. Desde primaria sabemos que los triángulos *rectángulos* son especiales porque cumplen el teorema de Pitágoras: el cuadrado de la hipotenusa es igual a la suma del cuadrado de los catetos.

La hipotenusa es siempre el lado más largo de los tres. Para saber, por tanto, si un triángulo es rectángulo, podemos buscar la hipotenusa candidata (el segmento más largo) y mirar si su cuadrado es igual a la suma de los cuadrados de los otros dos. Si lo es, entonces los segmentos cumplen el teorema de Pitágoras y por tanto el triángulo será rectángulo.

Para diferenciar entre los otros dos tipos (acutángulo y obtusángulo) debemos, otra vez, hacer una labor de visualización mental. Si pensamos que tenemos un triángulo rectángulo y vamos poco a poco reduciendo su hipotenusa (sin cambiar las longitudes de los catetos originales) lo que ocurrirá es que los catetos "se cierran" convirtiendo al triángulo en *acutángulo*. Si, por el contrario, la hipotenusa original crece, "empujará" a los catetos hacia el exterior, abriendo el ángulo y convirtiendo al triángulo en obtusángulo. Puedes comprobarlo con la figura anterior.

Estas reflexiones nos llevan a la idea de solución:

- Ordenar las longitudes de los segmentos de menor a mayor para identificar el mayor.
- Si el mayor es más largo que la suma de los otros dos, no se puede formar un triángulo y hemos terminado.
- En otro caso, calculamos la hipotenusa que tendría un cuadrado rectángulo si tuviera como catetos a los segmentos más cortos.
  - Si coincide con el segmento largo, tenemos un triángulo rectángulo.
  - Si es mayor que el segmento largo (o, al contrario, *el segmento largo es más corto* que la hipotenusa), tendremos un triángulo acutángulo.
  - Si no, tendremos un triángulo obtusángulo.

Para ordenar las longitudes de los segmentos utilizaremos comparaciones entre ellos. Es algo laborioso, pero dado que el número de segmentos a ordenar es pequeño y siempre el mismo (3) es asumible. Empezaremos leyendo las longitudes de los segmentos en tres variables (por ejemplo a, b y c) y buscaremos la manera de conseguir que los valores acaben de tal forma que se cumpla que $a \leq b \leq c$.

Para ello, empezamos comparando los dos primeros segmentos, a con b. Si *están en desorden* (es decir si $a > b$) entonces *los intercambiamos* de modo que a pase a tener el valor de b y viceversa, y estarán ordenados entre sí. Este intercambio necesita *una variable auxiliar* donde guardar, temporalmente, el valor original de una de las dos variables antes de sustituir su valor por el de la otra. Piensa que lo que queremos hacer es equivalente a tener un vaso en cada mano y querer intercambiarlos: necesitamos, temporalmente, un lugar donde dejar uno de ellos para poder hacer el cambio.

Este proceso de comparar y, si están en desorden, intercambiar los valores se repite con las otras dos parejas, lo que garantizará que al acabar los tengamos en el orden correcto. Al final, terminaremos teniendo en la variable c la longitud del segmento más largo, que será la supuesta hipotenusa. A partir de ahí, hacemos las comparaciones para saber si se puede formar el triángulo y su tipo, tal y como hemos descrito antes.

```
unsigned int a, b, c;
unsigned int aux;

cin >> a >> b >> c;

// Los ordenamos.
if (a > b) {
    // a y b están en desorden.  Les damos la vuelta.
    aux = b;
    b = a;
    a = aux;
}
if (a > c) {
    aux = c;
    c = a;
    a = aux;
}
if (b > c) {
    aux = c;
    c = b;
    b = aux;
}

```

```
// Ya está.  a <= b <= c
if ((a + b) <= c) {
    cout << "IMPOSIBLE\n";
}
else {
    unsigned int hipotenusaCuadrado = a*a + b*b;
    if (hipotenusaCuadrado > c*c) {
        cout << "ACUTANGULO\n";
    }
    else if (hipotenusaCuadrado == c*c) {
        cout << "RECTANGULO\n";
    }
    else {
        cout << "OBTUSANGULO\n";
    }
} // if-else era imposible
```

Información

El intercambio del contenido de las variables lo estamos haciendo de manera artesanal. El proceso es tan habitual que C++ nos facilita el trabajo. Por ejemplo, para intercambiar a y b podríamos haber hecho:

```
if (a > b) {
    std::swap(a, b);
}
```

Dependiendo del compilador que utilices (y la versión de C++) es posible que tengas que incluir el fichero de cabecera `algorithm` o `utility`.

Fíjate que el teorema de Pitágoras lo utilizamos en su modo "canónico" de los cuadrados. Podríamos haber utilizado la raíz cuadrada y comparar directamente con c, pero dado que la raíz cuadrada añade riesgo de imprecisión y es más lenta que multiplicar dos enteros, en este caso es más práctico comparar directamente los cuadrados.

Hazlo por tu cuenta

Para saber si podemos formar un triángulo, o su tipo, lo único que nos interesa es el valor del segmento más largo. Los otros dos son "indistinguibles". Si nuestra ordenación estuviera mal y colocaramos a y b en desorden, nuestro código funcionaría de todas formas porque lo importante es que c termine siendo el segmento más largo.
Esto nos permite simplificar el problema porque en lugar de *ordenar* lo que necesitamos realmente es *encontrar el mayor de los tres*. Piensa en cómo resolver el problema con esta nueva idea.

Información

Ordenar valores es una tarea muy habitual en informática (y en programación competitiva en particular) que se utiliza como punto de partida para muchos algoritmos, como ha ocurrido aquí. Si tenemos solo 3 valores es asequible hacerlo de forma manual comparando todos con todos, pero si tenemos 5, 10 o 1.000 números el proceso se vuelve impracticable. El problema es tan importante que ha sido muy estudiado y existen muchas formas de ordenar elementos (el que hemos usado podría categorizarse como "ordenación por burbuja"). Haremos problemas para practicar la ordenación más adelante.

304 División euclídea ★

Categorías

- Condicionales
- Matemáticas

Resumen del enunciado

Cuando dividimos dos números a y b, obtenemos dos valores, el cociente q y el resto r. Para que el resultado sea correcto, se debe cumplir que $a = b \times q + r$.

Pero nos falta algo. Si no ponemos ninguna condición más, entonces podríamos decir que la división de 87 entre 17 es 0, con resto 87. Con esa respuesta se cumpliría el $a = b \times q + r$ pero es, claramente, incorrecta. Para que la divisón sea válida, se debe también cumplir algún tipo de condición *sobre el resto*.

La llamada *división euclídea* añade la restricción de que el resto no debe ser negativo y tiene que ser menor que el valor absoluto de b ($0 \leq r < |b|$). Esto suena natural, pero las cosas se tambalean cuando a, b o incluso ambos son negativos, especialmente porque los ordenadores normalmente *no* hacen división euclídea.

Dado dos números enteros, hay que dar el cociente y el resto de su división euclídea.

Solución

Para calcular la *división entera* entre dos números, muchos lenguajes de programación (como C, C++ o Java) utilizan el operador `/`. Y para calcular el resto, se utiliza el operador módulo (`%`).

El comportamiento de ambos operadores es desconcertante cuando el numerador o el denominador son negativos, porque el resultado *puede dar restos negativos*, algo que va en contra de la llamada *división euclídea* mencionada en el enunciado. En particular, la tabla siguiente muestra lo que devuelven el operador de división entera y del módulo ante las cuatro posibilidades de los signos del dividendo y divisor:

Dividendo	Divisor	Cociente	Resto
7	3	2	1
7	−3	−2	1
−7	3	−2	**−1**
−7	−3	2	**−1**

Los resultados desconcertantes son los dos últimos, donde el resto *es negativo*. Fíjate, no obstante, que en todos los casos la condición $a = b \times q + r$ es correcta, pero falla la restricción euclídea sobre el resto en los dos últimos.

Para resolver el problema, lo primero que tenemos que saber es el resultado que habría que dar en esos dos casos particulares. Necesitamos *arreglar el resto* para que sea positivo *sin estropear la división*. En el caso de $-7 \div 3$ lo que hacemos es *sumar el divisor (3) al resto* para sacarlo de los negativos, y al mismo tiempo restar uno al cociente para compensar:

$$-7 \div 3 \Rightarrow -7 = 3 \cdot -2 + -1 = 3 \cdot -2 + -1 + 3 - 3 = 3 \cdot -2 - 3 + (-1 + 3) = 3 \cdot -3 + 2$$

En el caso de $-7 \div -3$ el procedimiento es equivalente, pero ahora no podemos sumar el divisor al resto, porque el divisor *es negativo* (-3) y si lo sumamos, vamos a hacer al resto todavía más pequeño. Lo que tenemos que hacer es *restar* el divisor (restar un negativo es sumar su valor absoluto). Para que la división siga siendo correcta, ahora hay que *incrementar* el cociente:

$$-7 \div -3 \Rightarrow -7 = -3 \cdot 2 + -1 = -3 \cdot 2 + -1 + -3 - -3 = -3 \cdot 2 + -3 + (-1 - -3) = -3 \cdot 3 + 2$$

Si ampliamos la tabla anterior con el resultado *esperado* de acuerdo a las normas de la división euclídea que hemos usado ahora tendremos:

Dividendo	Divisor	Cociente	Resto	Cociente esperado	Resto esperado
7	3	2	1	2	1
7	−3	−2	1	−2	1
−7	3	−2	**−1**	**−3**	**2**
−7	−3	2	**−1**	**3**	**2**

Esto nos da las reglas que tenemos que aplicar en los casos especiales que, ahora ya sí, podemos llevar a un lenguaje de programación. Hay que añadir, además, la comprobación de que el denominador no sea 0, que el enunciado nos pide que tratemos de forma especial al no poder dividir por dicho valor.

```
int numerador, denominador;
int cociente, resto;

cin >> numerador >> denominador;

if (denominador == 0)
    cout << "DIV0\n";
else {
    cociente = numerador / denominador;
    resto = numerador % denominador;
    if (resto < 0) {
        // Hay que llevar el resto a los positivos, sumando
        // el denominador.  Pero el denominador puede ser negativo, y
        // si sumamos, estaremos restando y rompiéndolo más.
        // Hay que ver el signo del denominador.
        if (denominador > 0) {
            // Caso normal.  Sumamos el denominador al resto.  Eso
            // además significa que el cociente es uno menos.
            // a = b·q + r -> a = b·(q-1) + r + b
            resto += denominador;
            cociente -= 1;
        }
        else {
```

```
            // Caso invertido.  Hay que restar el denominador.  Eso
            // además significa que el cociente es uno más
            // a = b·q + r -> a = b·(q+1) + r - b
            resto -= denominador;
            cociente += 1;
        }
    } // if (resto < 0)
    cout << cociente << " " << resto << "\n";
} // if-else denominador == 0
```

114 Último dígito del factorial ★

Categorías

- Condicionales
- Matemáticas
- Acertijo

Resumen del enunciado

Cada caso de prueba es un número natural y hay que dar *el último dígito del factorial*. El factorial de un número n es la multiplicación de todos los números entre 1 y el propio n.

Solución

Este problema lo categorizamos de "acertijo" porque tiene una trampa oculta. Tras leer el enunciado, una reacción habitual puede ser buscar la forma de calcular el factorial del número leído de la entrada, y luego utilizar el *operador módulo* para obtener su último dígito calculando el resto de dividir el factorial entre 10.

El cálculo del factorial es un ejercicio muy habitual en los cursos de introducción a la programación. Hay dos aproximaciones principales. Una de ellas es iterativa y utiliza un bucle para *repetir* múltiples veces la multiplicación (primero por 1, luego por 2, luego por 3, y así hasta el número n). La segunda es una aproximación *recursiva* y el cálculo del factorial es uno de los ejemplos más típicos de este tipo de algoritmos.

Pero, en este momento, estamos dando por hecho que ninguna de las dos cosas las conocemos aún. ¡Estamos practicando condicionales! Y solo con condicionales *no* se puede calcular el factorial de un número. Eso es toda una indicación de que algo pasa en este ejercicio.

Cuidado

Si sabes programar el cálculo del factorial e intentas resolver el problema con esta idea, el juez automático te rechazará la solución. Hay dos causas principales:

- *Time Limit Exceeded* (TLE): calcular el factorial de un número alto require un bucle que dé potencialmente muchas vueltas. Eso es mucho más lento que la solución correcta que describiremos ahora.
- *Wrong Answer* (WA): si sorteas el problema del tiempo, la solución fallará porque no calculará bien el último dígito del factorial para números "grandes". El factorial crece muy deprisa. El de 24 (*24!*) tiene 24 dígitos, que es mucho más de lo que entra en un entero normal. El cálculo sufre *desbordamiento*.

Para resolver el problema hay que darse cuenta de que al calcular *10!* vamos a multiplicar por 10. Esa multiplicación nos añade un 0 como último dígito del factorial. Y, mejor aún, dado que para calcular el factorial de cualquier número mayor que 10 hay, también, que multiplicar en algún momento por 10, el factorial de *todos* los números a partir de 10 tienen como último dígito un 0.

Pero esto no acaba aquí. El factorial de 5 es:

$$5! = \mathbf{5} \cdot 4 \cdot 3 \cdot \mathbf{2} \cdot 1 = 120$$

La multiplicación de 5 y 2 da 10, por lo que *5!* también termina en 0, y lo mismo ocurrirá con todos los números mayores que él. ¡No hay necesidad de llegar a 10! No es casualidad que el ejemplo del enunciado del problema nos ponga los primeros valores, hasta el 4, ocultando lo que ocurre con el valor siguiente.

El resultado es que podemos resolver el problema con condicionales. Para *cualquier* número de la entrada mayor o igual que 5 el resultado es 0, y lo sabemos sin calcular el factorial. Para los números menores, podemos calcularlo en papel y cablear el valor en el código directamente, o hacer el cálculo en el propio programa. Un último detalle que puede pasar desapercibido es que la entrada *puede ser 0*. Aunque el enunciado no lo dice, $0! = 1$, que tenemos que meter también entre nuestras respuestas.

```
unsigned int i, respuesta;

cin >> i;

switch(i) {
    case 0:
    case 1:
        respuesta = 1;
        break;
    case 2:
        respuesta = 2;
        break;
    case 3:
        respuesta = 6; // 2*3
        break;
    case 4:
        respuesta = 4; // 2*3*4 = 24
        break;
    default:
        respuesta = 0;
} // switch

cout << respuesta << "\n";
```

Información

También se puede programar utilizando una secuencia de `if`'s, pero el `switch` es muy adecuado en este caso. Eso sí, ¡no olvides los `break`'s!

241 Me quiere, no me quiere ★

Categorías

- Condicionales
- Pensar

Resumen del enunciado

Tenemos en el suelo n *pétalos* que han salido de deshojar un conjunto de tréboles de 3 y 4 hojas. Hay que decir cuántos eran de 4, asumiendo que es la menor cantidad posible.

Matemáticamente, dado un número n, hay que buscar cómo escribirlo de la forma:

$$n = 3 \cdot t + 4 \cdot c$$

con t y c no negativos, y con el menor c posible, que es lo que hay que escribir. Si es imposible, hay que escribir `IMPOSIBLE`.

Solución

Para resolverlo, lo más fácil es probar varios números consecutivos para intentar sacar un patrón. Por ejemplo, si n es 15, entonces podemos asumir que esas hojas han salido de 5 tréboles (de 3 hojas) y no había ninguno de 4. También podrían haber salido de 3 tréboles de 4 hojas y uno de 3 ($3 \cdot 4 + 3 = 15$) pero dado que el enunciado nos pide que lo consigamos con el menor número de tréboles de 4 hojas posible, nos quedamos con la primera opción donde no había ninguno.

Si nos encontramos 16 pétalos, entonces podrían haber salido de 4 tréboles de 4 hojas. Pero también podrían haber salido de 4 tréboles de 3 hojas, y un único trébol de 4. Como antes, se busca la respuesta con el menor número de tŕeboles de 4 hojas, por lo que en este caso la respuesta sale de la última configuración, con solo 1.

Si nos encontramos 17 pétalos, la única posibilidad es que hayan salido de 3 tréboles de 3 hojas, y 2 de 4.

Por último, si nos encontramos 18 pétalos, entonces habrán salido de 6 tréboles de 3 hojas. Esto nos lleva a una situación equivalente a la del 15, donde no hay ningún trébol de 4 hojas. A partir de ahí, podemos intuir que se repite el ciclo.

Al final de lo que hay que darse cuenta es de que:

- Si hay $3 \cdot k$ hojas, entonces todos los tréboles eran de 3 hojas y tenemos que escribir un 0 (no había ninguno de 4).
- Si hay $3 \cdot k + 1$ hojas, entonces todos los tréboles eran de 3 hojas salvo 1, que era de 4.
- Si hay $3 \cdot k + 2$ hojas, entonces todos los tréboles eran de 3 hojas salvo 2, que eran de 4.

Para saber en qué situación de las 3 estamos, hay que saber el resto de dividir el número n de la entrada por 3, haciendo uso del *operador módulo*.

Hay que tener cuidado únicamente con los imposibles. Los casos de prueba del ejemplo del enunciado nos dan la idea de que con menos de 3 hojas hay que escribir `IMPOSIBLE`.

Pero hay una trampa más: también es imposible conseguir 5 hojas que vengan de tréboles de 3 o 4 hojas.

El resto de números sí es posible y se usa la respuesta calculada.

```
unsigned int numHojas;

cin >> numHojas;

if ((numHojas < 3) || (numHojas == 5))
    cout << "IMPOSIBLE\n";
else
    cout << numHojas % 3 << "\n";
```

Información

Los paréntesis alrededor de la expresión de comparación no son realmente necesarios porque los operadores relacionales tienen más prioridad que los lógicos, y se ejecutarán antes.

397 ¿Es múltiplo de 3? ★

Categorías

- Condicionales
- Pensar

Resumen del enunciado

Si ponemos seguidos el número 1, el 2, el 3, y así hasta un n conocido, formaremos un número gigante con un montón de dígitos. Dado el n nos preguntan si el número construído con este procedimiento es o no múltiplo de 3.

Solución

La solución inocente consiste en, de verdad, construir el número resultante de escribir, uno detrás de otro, todos los números desde el 1 hasta el n y usar el operador *módulo* para saber si es o no divisible por 3. Pero esta solución está condenada a fallar. El enunciado nos avisa de que el valor de n puede llegar a ser hasta 10^9. Poner, seguidos, todos los números desde 1 hasta 10^9 no solo lleva mucho tiempo, sino que da lugar a un número gigantesco que no entra, por mucho, en un entero normal de los lenguajes de programación. Eso significa que la construcción sufrirá *desbordamiento* y el cálculo del resto, si llegamos a poder hacerla antes de que se nos termine el tiempo, estará mal.

Hay que pensar una solución distinta. Estamos ante un problema en el que la solución más natural es demasiado lenta y debemos agudizar el ingenio para encontrar una aproximación más rápida. Si tuvieras que hacerlo con lápiz y papel, ¿te pondrías a escribir todos los números del 1 al n o intentarías buscar una aproximación más inteligente?

En cuanto n fuera un poco grande, seguramente intentarías buscar algún atajo. Para encontrarlo, hay que recordar que podemos saber si un número es divisible por 3 mirando si *la suma de sus dígitos* lo es. De hecho, esta regla ¡es todavía más general! El resto de dividir un número cualquiera k entre 3 da el mismo resultado que el resto de dividir por 3 *la suma de los dígitos de* k.

Eso nos lleva a poder reescribir el problema que nos ocupa. No es necesario construir el número gigante. Es suficiente con saber si la suma de sus dígitos es divisible por 3. Dicho de otro modo, la suma de los dígitos de 1, más la suma de los dígitos de 2, ... más la suma de los dígitos de n ¿da algo divisible por 3? Si es así, entonces el número que formaríamos será también divisible por 3.

Y ¿cuál es la suma de los dígitos de todos esos números? Pues ni siquiera hace falta calcularla, porque lo que nos interesa es si su suma es o no divisible por 3. Lo interesante es que por la manera en la que estamos construyendo el número es fácil ver un patrón:

n	Número construído	Suma de los dígitos	Módulo de la suma y 3
1	1	1	1
2	12	3	0
3	123	6	0
4	1234	10	1
5	12345	15	0
6	123456	21	0

La columna importante es la última, donde vemos el resto de dividir la suma de los dígitos del número construído entre 3. Vemos aparecer un patrón: 1,0,0; 1,0,0; ... El resto 1 indica que el número completo *no* es divisible por 3, y el 0 que sí lo es (puedes comprobarlo por tu cuenta). Analizando para qué n ocurre cada cosa podemos intuir que si el n de la entrada (último número usado para construir el número completo) es 1, 4, 7, ..., es decir si su resto con 3 da 1, el número completo *no* será divisible por 3. En otro caso, sí lo será.

Pero, ¿por qué ocurre esto? Cuando vamos añadiendo los números 1, 2, 3, ... etcétera para crear el número completo, primero incorporamos un número cuyo resto con 3 es 1, luego uno cuyo resto con 3 es 2, y finalmente uno cuyo resto con 3 es 0. Esto se repite en ciclo, porque al añadir el 4 volvemos a incorporar un número cuyo resto es 1, etcétera.

Si pensamos ahora en *sus dígitos*, que es lo que nos importa para nuestro problema, estamos haciendo lo mismo dado que, como hemos visto antes, el resto de dividir un k entre 3 es lo mismo que el resto de dividir por 3 la suma de sus dígitos. Esto significa que al añadir *al número completo* el número 1, añadimos dígitos cuyo resto con 3 es 1. Al añadir el número 2, añadimos dígitos cuyo resto con 3 es 2. Y finalmente al añadir el número 3, añadimos dígitos cuyo resto con 3 es 0. Y esto se repite a partir de ahí todo el tiempo.

Y de ahí surge el patrón. El primer número añade "resto 1", que hace que el número completo no sea divisible por 3. El siguiente añade "resto 2" que se "fusiona" con el anterior de resto 1, y pasan a formar, en conjunto, una suma de resto 0, que hace que el número completo sea divisible por 3. El siguiente número es en sí mismo múltiplo de 3, por lo que también lo son sus dígitos, que mantienen la divisibilidad del número completo.

Al final, la solución del problema es increíblemente sencilla de programar:

```
unsigned int n;

cin >> n;

if ((n % 3) == 1)
    cout << "NO\n";
else
    cout << "SI\n";
```

Hazlo por tu cuenta

Resuelve el problema utilizando el operador ternario `:?`