Entrenamiento OIE 2023 Nivel Inicial

Sesión 1: Variables y expresiones

El objetivo de la primera sesión de entrenamiento es "romper el hielo" con la programación y el lenguaje C++, y coger algo de práctica con las herramientas básicas necesarias para resolver los problemas propuestos.

En las sesiones de entrenamiento vamos a resolver problemas disponibles en un juez en línea llamado *¡Acepta el reto!* Si no lo has hecho aún, deberías leer el documento Mi primer problema en *¡Acepta el reto!*, donde se explica qué es un juez online, se analiza el esquema de los enunciados y se resuelve un primer problema siguiendo un *esqueleto de solución*. Todos los problemas de esta primera sesión de entrenamiento utilizan exactamente el mismo esqueleto, porque todos tienen el mismo *esquema de la entrada*. Aquí asumiremos que has leído el documento anterior, has sido capaz de seguirlo para resolver, de forma guiada, un primer problema, y sabes cómo interpretar un enunciado y cómo usar el *esqueleto de solución* descrito.

En las sesiones de entrenamiento se proponen una serie de problemas a resolver para practicar los conceptos que son el foco de la sesión. El nivel de dificultad es variado y aquellos que se consideran "retos", con una dificultad algo mayor, se dejan para el final y se marcan con una estrella (★).

Los problemas planteados en esta sesión requieren que se conozcan únicamente los siguientes conceptos del lenguaje:

- Variables, tipos básicos y límite de la representación.
- Entrada/salida por consola: saber leer valores de teclado (enteros o cadenas) y saber escribirlos en pantalla.
- Expresiones aritméticas para operar con números.

Si se conocen los condicionales (`if`), bucles (`for`) y el formateo de la salida (`std::setw` y `std::setfill`) al resolver algunos de los problemas puede resultar natural su uso. Sin embargo no son obligatorios y te animamos a que, si los has usado, vuelvas a resolver los problemas sin ellos, usando *únicamente* variables y expresiones.

En la mayor parte de los problemas lo más difícil es saber *cómo se resuelven* y no tanto *cómo se programan*. Antes de empezar a escribir código, es importante pararse a pensar, buscar una forma de resolver el problema con lápiz y papel y una vez que las ideas estén claras, convertirlas a un lenguaje de programación es la parte fácil. ¡Primero resuelve el problema y luego escribe el código!

293 Artópodos

Categorías

- Expresiones

Resumen del enunciado

El problema nos habla de los *artrópodos*, animales invertebrados que, dependiendo de su tipo, tienen un número distinto de patas. En particular:

- Los **insectos** tienen **6 patas**
- Los **arácnidos** tienen **8 patas**
- Los **crustáceos** tienen **10 patas**
- Las **escalopendras** tienen **2 patas por anillo**, y un número variable de anillos

Cada caso de prueba nos dice cuántos *insectos*, *arácnidos*, *crustáceos* y *escalopendras* tenemos. Estas últimas son todas de la misma especie, y nos dicen también cuántos anillos tiene cada individuo. Nos preguntan cuántas patas tienen en total todos ellos.

Solución

Antes de empezar a programar la solución, lo importante es tener claro cómo resolveríamos el problema con lápiz y papel. Para cada caso de prueba que hay que resolver nos proporcionan cinco números. Por comodidad, vamos a representar cada uno de esos valores con una letra (como en las ecuaciones matemáticas) para podernos referir a cada valor cómodamente después:

- i : número de insectos (cada uno tiene 6 patas)
- a : número de arácnidos (cada uno tiene 8 patas)
- c : número de crustáceos (cada uno tiene 10 patas)
- e : número de escalopendras
- s : número de segmentos que tiene cada una de las escalopendras anteriores. Cada segmento tiene dos patas

Con esta información, para saber el número total de patas tenemos, simplemente, que multiplicar:

$$(6 \cdot i) + (8 \cdot a) + (10 \cdot c) + (e \cdot 2 \cdot s)$$

El último sumando es especial, porque el número de patas que tiene cada una de las e escalopendras no lo sabemos con antelación (como pasa con los demás), sino que tenemos que calcularlo multiplicando por 2 el número de segmentos que tienen $(2 \cdot s)$.

Hemos puesto los paréntesis para evitar dudas pero, en realidad, no hacen falta. En las fórmulas matemáticas y en las expresiones escritas en un lenguaje de programación como C++, la multiplicación *tiene prioridad* sobre la suma. Eso significa que primero se hacen todas las multiplicaciones, y luego se hacen todas las sumas. El resultado es que la fórmula anterior también podemos escribirla sin paréntesis:

$$6 \cdot i + 8 \cdot a + 10 \cdot c + e \cdot 2 \cdot s$$

Si miramos ahora el primer caso de prueba del ejemplo del enunciado veremos:

```
1 1 1 1 15
```

Según la descripción de la entrada del enunciado, eso significa que nuestros valores son:

- $i = 1$ (número de insectos)
- $a = 1$ (número de arácnidos)
- $c = 1$ (número de crustáceos)
- $e = 1$ (número de escalopendras)
- $s = 15$ (número de segmentos de cada escalopendra)

Si usamos esos números en nuestra fórmula:

$$6 \cdot i + 8 \cdot a + 10 \cdot c + e \cdot 2 \cdot s = 6 \cdot 1 + 8 \cdot 1 + 10 \cdot 1 + 1 \cdot 2 \cdot 15 = 54$$

Nuestra solución "con lápiz y papel" es 54, que coincide con el resultado que nos dice el ejemplo del enunciado.

Información

Normalmente no nos entretendremos tanto explicando el enunciado. Lo hacemos esta vez porque es nuestro primer problema. En lo sucesivo, se espera que este análisis lo hagáis vosotros y solo nos detendremos en casos que puedan resultar confusos o conflictivos.

Para resolver el problema en un lenguaje de programación necesitamos *leer y guardar* cada uno de esos números, para poder operar con ellos después. Para eso necesitamos *variables de tipo entero* para guardar números sin decimales (`int`). Una vez que tenemos el *hueco en memoria* para guardar los datos, leemos valores de teclado y los guardamos en ellas. Finalmente calculamos el valor de la fórmula anterior, y escribimos el resultado.

El "corazón" de la solución de este problema, en C++, queda:

```
1 int insectos, aracnidos, crustaceos, escolopendras, segmentos;
2
3 cin >> insectos >> aracnidos >> crustaceos;
4 cin >> escolopendras >> segmentos;
5
6 cout << 6*insectos + 8*aracnidos + 10*crustaceos +
7         2*escolopendras*segmentos << "\n";
```

Hay algunas cosas a destacar:

- Por claridad, en lugar de utilizar como nombres de variables (línea 1) las letras anteriores (*i*, *a*, etcétera) hemos utilizado los nombres completos, de modo que el código sea más autoexplicativo.
- Hemos leído los datos solo en dos líneas, aprovechando que podemos encadenar varias lecturas en el mismo `cin` (líneas 3 y 4). También podríamos haber hecho lo mismo con 5 líneas distintas, una por variable, o incluso en una sola.
- Hemos calculado el número total de patas en el mismo sitio que la escribimos (líneas 5 y 6). También podríamos haber calculado el valor para guardarlo en una variable,

y luego escribir su contenido. Como la línea quedaba muy larga, la hemos partido en dos. A C++ no le molesta.

- Después de escribir el número total de patas, pedimos al programa que cambie de línea escribiendo un `\n`, para que el resultado del caso de prueba siguiente no quede pegado con el actual. Además, usamos `\n` y no `std::endl` como puedes haber visto. Para este problema no es muy importante, pero en *¡Acepta el reto!* es mejor acostumbrarse a usar `\n` porque es algo más rápido, y podría ser importante en problemas más difíciles.

El código anterior no está completo. Hay que meterlo en el *esqueleto de solución* que comentábamos antes que está descrito en el documento Mi primer problema en *¡Acepta el reto!* El código completo queda:

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

void casoDePrueba() {

    int insectos, aracnidos, crustaceos, escolopendras, segmentos;

    cin >> insectos >> aracnidos >> crustaceos;
    cin >> escolopendras >> segmentos;

    cout << 6*insectos + 8*aracnidos + 10*crustaceos +
            2*escolopendras*segmentos << "\n";

} // casoDePrueba

//------------------------------

int main() {

    unsigned int numCasos;

    cin >> numCasos;

    for (unsigned int i = 0; i < numCasos; ++i) {
        casoDePrueba();
    }

    return 0;

} // main
```

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.

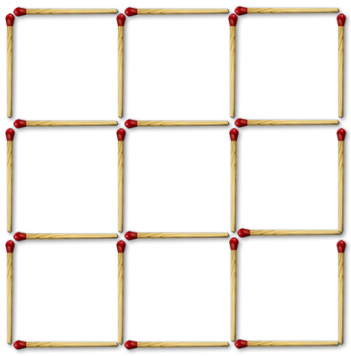
340 Cuadrados con cerillas

Categorías

- Expresiones

Resumen del enunciado

Usando cerillas podemos crear formas, como una cuadrícula:

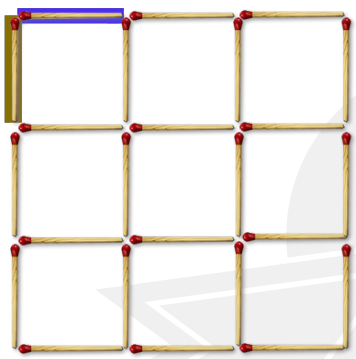


Nos dan el número de cuadrados a lo ancho y a lo alto y nos preguntan cuántas cerillas necesitamos.

Solución

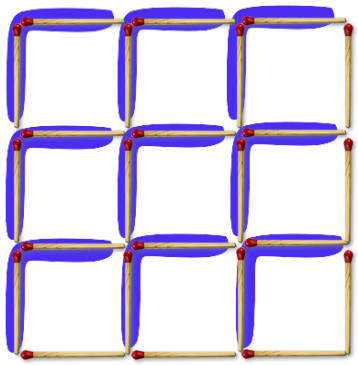
La parte más complicada de este problema es saber cómo resolverlo con lápiz y papel. Tenemos que buscar una fórmula que nos permita calcular el total, y para eso hay que buscar el "patrón" que ocurre en la figura. Hay varias formas de buscar ese patrón, y cada una lleva a una fórmula diferente aunque, matemáticamente, luego todas serán equivalentes.

La dificultad de ver el patrón es que casi todas las cerillas forman parte de *dos* cuadrados, y eso aparentemente complica calcular el total. Como lo que es fácil de hacer es *contar cuadrados*, tenemos que intentar "asignar" cada cerilla a un único cuadrado y no a dos, y buscar un patrón en esa asignación. Con esta idea en mente, vemos que las dos cerillas de la esquina superior izquierda de la figura tenemos que asignárselas irremediablemente al cuadrado de esa esquina, porque solo se usan en él.



Eso significa que al primer cuadrado de la figura le hemos asignado "como suyas" la cerilla que tiene encima y la que tiene a su izquierda. Si hacemos lo mismo con todos los

demás cuadrados, podemos comprobar que ninguna cerilla acaba asignada a dos cuadrados distintos:



Esta asignación significa que cada cuadrado que añadimos en una nueva fila por arriba o columna por la izquierda nos añade dos cerillas nuevas, y reutiliza dos de las que ya había. Es importante darse cuenta, eso sí, de que los cuadrados que quedan en el lado derecho e inferior quedan sin cerrar. En la figura, vemos que las cerillas de esos dos laterales ¡están sin asignar a ningún cuadrado! Por tanto tenemos que contarlas de forma independiente.

Al final, la cuenta que nos sale es que cada cuadrado añade a la figura dos cerillas (que serán potencialmente reutilizadas por otros) y luego tenemos que terminar de "cerrar" la figura poniendo una cerilla más por cada cuadrado a lo ancho, y por cada cuadrado a lo alto. Si tenemos f filas y c columnas de cuadrados, la fórmula que estamos buscando es:

$$2 \cdot c \cdot f + c + f$$

Puedes confirmar que funciona con los ejemplos que aparecen en el enunciado.

Cuidado

Que una fórmula parezca servir para resolver los ejemplos del enunciado ¡no es garantía de nada! La fórmula podría estar mal pero, casualmente, funcionar con los ejemplos, como pasa en este problema con:

$$2 \cdot (f^2 + f)$$

donde ¡ni siquiera usamos el número de columnas! Con esto se pone de manifiesto que los ejemplos no sirven para probar exhaustivamente la solución y es posible que tu código funcione con ellos pero luego el juez te diga que no es correcto.

En este caso, no obstante, la fórmula a la que hemos llegado estará bien. Escribir el código en C++ que resuelve el problema es ya la parte fácil.

```
1 int filas, columnas;
2
3 cin >> columnas >> filas;
4
5 cout << 2*columnas*filas + columnas + filas << "\n";
```

117 La fiesta aburrida

Categorías

- Lectura de la entrada

Resumen del enunciado

En cada caso de prueba leemos una frase del estilo de "Soy Lotario" y hay que escribir "Hola, Lotario."

Solución

El caso de prueba tiene en la entrada *dos palabras*, una con el "Soy" y otra con el nombre que tenemos luego que repetir para saludarle. Lo que tenemos que hacer es *leer las dos palabras* aunque la primera, "Soy", no la utilicemos para nada. Una vez que tenemos las dos, escribimos la respuesta:

```
1 string soy;
2 string nombre;
3
4 cin >> soy; // Nos saltamos la primera palabra
5 cin >> nombre;
6
7 cout << "Hola, " << nombre << ".\n";
```

Cuidado

Dependiendo de la plataforma en la que programes, es posible que tengas que hacer `#include <string>` (además del habitual `iostream`) al haber utilizado el tipo `string`.

Hay algunas consideraciones interesantes:

- Tenemos que leer la primera palabra, "Soy", y lo hacemos en una variable local a la que hemos llamado `soy` (línea 1), aunque podría llamarse de cualquier otra forma. No podemos leer sin guardar el valor (o al menos no de manera sencilla), de ahí que necesitemos una variable para él.
- El valor leído de la primera palabra no lo usamos luego para nada. Es posible que tu compilador te haya dado incluso un aviso diciéndote que tienes una variable en la que escribes pero no lees. Podríamos ahorrarnos la variable `soy` si *reutilizamos la del nombre*. Podemos leer dos veces sobre la misma variable, pisando el primer valor leído con la segunda lectura y también nos serviría:

```
1 string nombre;
2
3 cin >> nombre >> nombre;
4
5 cout << "Hola, " << nombre << ".\n";
```

- **No** hace falta comprobar que realmente hemos leído, exactamente, `Soy`. Si en el enunciado nos dice que la entrada tendrá el esquema "Soy <nombre>" no tenemos

que preocuparnos de nada más: el juez lo garantiza. Es verdad que nuestra solución también funcionaría si en lugar de decirnos "Soy Aldonza" nos dijeran "Adios Aldonza", a lo que contestaríamos con un desconcertante "Hola, Aldonza.". Pero eso *no importa*. El juez siempre va a decirnos lo mismo, y no hay razón para complicar la solución con casos que nunca ocurrirán.

- Igual que el juez nos garantiza el formato de la entrada, también *nos exige de la misma forma el formato de la salida*. Si al escribir la salida se nos olvida la coma ("Hola Aldonza."), el punto ("Hola, Aldonza") o escribimos en minúscula ("hola, Aldonza.") la solución *estará mal* y el juez no nos la aceptará. De hecho, la solución *ni siquiera funcionará con los casos del ejemplo del enunciado*. Por tanto, ¡ten mucho cuidado con el formato de la salida!

191 Los problemas de ser rico

Categorías

- Expresiones

Resumen del enunciado

Tenemos un acuario enorme con varios compartimentos de distinto tamaño. Nos dan cuántos compartimentos hay, el tamaño, en litros, del más grande, y cuántos litros menos va teniendo cada compartimento con respecto al anterior. Nos piden el número total de litros del acuario.

Solución

En el enunciado nos ponen como primer ejemplo:

```
5 300 10
```

Por el formato de la entrada, eso significa que nuestro acuario tiene 5 compartimentos. En el más grande entran 300 litros, y el resto tienen 10 litros menos que el anterior. Esto implica que el segundo compartimento tendrá capacidad para 290 litros, el siguiente para 280, el siguiente para 270 y el último para 260. En total suman 1400, tal y como nos confirma la solución del ejemplo del enunciado.

Lo que hemos hecho para conseguir la suma total es ir calculando el tamaño de cada uno de los compartimentos del acuario, de uno en uno, repitiendo una y otra vez el proceso de restar 10 litros y sumarlo al acumulado. En programación, la repetición se consigue con un *bucle*, y se puede usar uno para desarrollar la solución.

Pero nosotros categorizamos este problema como un problema de *expresiones*, lo que indica que existe una fórmula para conseguir el resultado de forma directa. Aunque para este problema no afecta mucho, utilizar una fórmula es más rápido en ejecución que un bucle, y la diferencia puede resultar importante en problemas más avanzados. Piensa, por ejemplo, en tener que calcular el número de litros para un acuario de 1000 compartimentos... con lápiz y papel. Ir sumando una por una la capacidad de cada uno de los compartimentos es muy tedioso. Si eres tú quien tiene que hacer las sumas, ¿no se te agudiza el ingenio para intentar evitarlas?

Para que sea más fácil de explicar vamos a poner nombre a cada dato que tenemos:

- c : número de contenedores del acuario
- m : número de litros que entran en el contenedor de mayor capacidad
- d : número de litros que desciende la capacidad de un contenedor al siguiente

Información

Aunque el enunciado no lo dice, se puede asumir que los datos de la entrada no nos llevarán a tener compartimentos con capacidades negativas.

Para llegar a la fórmula que nos calcula, directamente, el número de litros, podemos seguir diferentes reflexiones aunque, al final, todas nos deberían llevar a una fórmula matemáticamente equivalente. Nosotros aquí describiremos una forma de llegar a ella, aunque puede que tú hagas llegado por otro camino.

Si todos los contenedores fueran de la capacidad del más grande, entonces la capacidad total sería $c \cdot m$. El problema es que los compartimentos van siendo cada vez más pequeños, y tenemos que *restar litros*. Del primer compartimento, el más grande, no hay que quitar litros. Del siguiente hay que quitar d, del siguiente $2 \cdot d$, del siguiente $3 \cdot d$ y así sucesivamente:

$$0 + d + 2 \cdot d + 3 \cdot d + ... + (c-1) \cdot d = d \cdot (0 + 1 + 2 + 3 + ... + (c-1))$$

Puede resultarte extraño que si tenemos c compartimentos en el acuario, nos quedemos en $(c-1) \cdot d$ en lugar de llegar a $c \cdot d$. La razón es que *tenemos el 0*. Del primer compartimento, el más grande, no hay que quitar litros, y eso nos deja $c-1$ compartimentos de los que sí.

La suma $1 + 2 + 3 + ... + k$ (hasta un valor k cualquiera) es lo que se llama *suma de una sucesión aritmética*. Se cuenta que, a los 10 años, Gauss evitó tener que sumar los 100 primeros números por un castigo que les puso su profesor utilizando un truco. Se dio cuenta de que la suma del primer y último número de la sucesión da el mismo resultado que la suma del segundo y el penúltimo, que, a su vez, es igual que la suma del tercero y el antepenúltimo, etcétera. Eso le llevó a una fórmula:

$$1 + 2 + 3 + ... + k = \frac{k \cdot (k+1)}{2}$$

Si sustituimos el k por nuestro $c-1$ (número de contenedores menos 1) llegamos a la solución del problema.

```
1 int numContenedores, mayor, decremento;
2 int resultado;
3
4 cin >> numContenedores >> mayor >> decremento;
5
6 resultado = numContenedores * mayor;
7 resultado -= decremento * (numContenedores - 1) * numContenedores / 2;
8
9 cout << resultado << "\n";
```

216 Goteras

Categorías

- Expresiones
- Formato de la salida

Resumen del enunciado

Obviando la ambientación, cada caso de prueba es un número indicando una cantidad de segundos y hay que escribirlo en formato `HH:MM:SS`, donde `HH` indica el número de horas, `MM` el de minutos y `SS` el de segundos. Hay que usar dos dígitos para cada elemento, y deben cumplirse las restricciones habituales (el número de segundos y minutos no puede ser mayor de 59).

Información

El enunciado nos garantiza que el número de segundos de la entrada no será mayor al número de segundos de un día.

Solución

Para resolver el problema hay que preocuparse de dos cuestiones:

- Averiguar el número de horas, minutos y segundos que supone una cantidad arbitraria de segundos
- Escribir el resultado con el formato pedido, poniendo ceros a la izquierda

Para la primera parte usaremos tres variables donde guardar el número de horas, minutos y segundos. Sabemos que en una hora hay 3600 segundos (60×60), y por tanto el número de horas completas se calcula dividiendo los segundos por ese 3600. Un detalle importante es que queremos hacer *división entera* dado que los decimales no nos importan, pues hacen referencia a la parte de la última hora no completa. Por ejemplo si nos piden escribir en el formato `HH:MM:SS` el tiempo que son 1800 segundos la división real nos daría 0'5 horas, pero con división entera es 0, pues hay "0 horas completas". Afortunadamente, en C++ si nuestras variables son de tipo entero (`int`) la división será automáticamente sin decimales, que es lo que queremos.

Una vez que tenemos el número de horas completas, tenemos que quitar el número de segundos que hemos contado y, con los que nos queden, calcular cuántos minutos hay. Como hemos quitado las horas completas, el número de minutos conseguidos será siempre menor que 60, por lo que cumpliremos las restricciones de la escritura de las horas.

Con esta idea, el código para calcular el número de horas, minutos y segundos queda:

```
1 // Primera parte:  separar el tiempo total en horas, minutos y segundos.
2 int numHoras, numMinutos, numSegundos, segundosTotales;
3
4 cin >> segundosTotales;
5
6 numHoras = segundosTotales / (60*60);
7 segundosTotales -= numHoras*60*60; // Quitamos las horas ya contadas
8
```

```
numMinutos = segundosTotales / 60;
segundosTotales -= numMinutos*60; // Sin los minutos ya contados

numSegundos = segundosTotales;
```

Con esto, es tentador usar las variables que tenemos y escribir el resultado:

```
cout << numHoras << ":" << numMinutos << ":" << numSegundos << "\n";
```

El código anterior no funcionará bien, porque *no usa dos dígitos para los números*. Por ejemplo para 61 segundos hará la conversión y escribirá `0:1:1` (0 horas, un minuto y un segundo) en lugar de `00:01:01` que es lo que nos piden.

Para escribir los números con dos dígitos podemos configurar la forma en la que funciona `cout`. Puedes buscar información sobre `std::setw` y `std::setfill`. Para practicar, no obstante, y dado que este problema se categoriza como "de expresiones", podemos hacer nosotros el trabajo.

Para escribir un número siempre con dos dígitos podemos *calcular por separado cada dígito*. Lo que tenemos que calcular es el dígito de las decenas y el dígito de las unidades del número original que, sabemos, es menor que 100 (tiene como mucho dos dígitos). Para calcular el dígito de las decenas basta con que *lo dividamos por 10*, otra vez ignorando los decimales (división entera). Eso nos da el primero de los dos dígitos. El dígito de las unidades podemos calcularlo *con el resto de la división por 10*. Por ejemplo, el resto de dividir 17 entre 10 es 7, que es precisamente el dígito buscado.

Cuidado

En el futuro, ¡no lo hagas así! Aunque aquí escribamos los números con dos dígitos de forma manual, hacer uso de las funcionalidades proporcionadas por el lenguaje es mucho más cómodo. ¡Es mejor usar `std::setw` y `std::setfill`!

Para calcular el resto de una división entera se utiliza el *operador módulo* o `%`. Con esta idea, el código para resolver cada caso de prueba queda:

```
// Primera parte:  separar el tiempo total en horas, minutos y segundos.
int numHoras, numMinutos, numSegundos, segundosTotales;

cin >> segundosTotales;

numHoras = segundosTotales / (60*60);
segundosTotales -= numHoras*60*60; // Quitamos las horas ya contadas

numMinutos = segundosTotales / 60;
segundosTotales -= numMinutos*60; // Sin los minutos ya contados

numSegundos = segundosTotales;

// Segunda parte:  escribimos los números con dos dígitos.
cout << numHoras / 10 << numHoras % 10 << ":"
     << numMinutos / 10 << numMinutos % 10 << ":"
     << numSegundos / 10 << numSegundos % 10 << "\n";
```

Hazlo por tu cuenta

El código anterior tiene algunas posibles mejoras. Por ejemplo, la variable `numSegundos` prácticamente no se utiliza. ¿Se te ocurre cómo escribir el programa sin ella? Además, las líneas 7 y 10 son poco elegantes. Ahora que has usado el operador módulo, ¿se te ocurre una manera de escribirlas de otra forma?

Hazlo por tu cuenta

Busca información sobre `std::setw` y `std::setfill` y resuelve el problema usándolos.

274 Semanas ★

Categorías

- Expresiones

Resumen del enunciado

El número de *semanas completas* que hay en un año depende del día de la semana en la que empiece dicho año. Por ejemplo en un año no bisiesto (de 365 días) entran 52 semanas y sobra un día. Pero si el primer día del año es, por ejemplo, sábado, la primera semana completa del año empezará el 3 de enero lunes, no el 1 (en esta explicación asumimos que las semanas empiezan en lunes, no en domingo como ocurre en algunos lugares del mundo). Además, la última semana acabará en sábado, y tampoco estará completa. El resultado es que ese año tendrá en realidad 51 semanas completas, y luego dos que no lo son.

Si, además, generalizamos el problema, el número de semanas completas realmente también depende del número de días que tenga una semana (en nuestro calendario siempre son 7) y del número de días que tenga el año (en nuestro calendario son 365 o 366).

En este problema, cada caso de prueba es el número de días que tiene un año, el número de días que tiene una semana, y el día de la semana del primer día de un año, y hay que decir cuántas semanas completas tiene ese año.

Solución

Para explicar la forma de resolver el problema es mejor que nos centremos en nuestro calendario de 7 días semanales y 365 días al año. Así podremos hablar de días de la semana (lunes, martes, ...) y que la explicación resulte más natural. Una vez entendido el proceso, podemos dar el salto a calendarios distintos con semanas y años de longitudes diferentes.

Si un año empieza en lunes, entonces todos los días, desde el primero, forman parte de semanas completas. Lo único de lo que tenemos que preocuparnos es de saber qué ocurre al final del año, porque podemos dejar la última semana sin completar. Para saber cuántas semanas completas tenemos en este caso es suficiente con dividir el número de días del año entre la longitud de la semana, es decir 365/7, usando división entera, para descartar la última semana que no se completa. Con nuestro calendario, eso nos da las 52 semanas (y el día sobrante) que mencionábamos antes.

Si el año no comienza en lunes, sin embargo, hay que pensar un poco más. En este caso, los días iniciales del año, hasta el primer lunes, son días desperdiciados que no forman parte de una semana completa. Por ejemplo, si el año comienza en viernes (el quinto día de la semana) hay tres días iniciales del año (el viernes, sábado y domingo) que no son útiles por no estar en una semana completa. Eso nos deja solo 362 días posibles para estar en semanas completas. Es ese número el que tenemos que dividir por 7, con división entera, para saber el número total de semanas completas.

La parte difícil del problema por tanto es saber cuántos días hay que quitar a los días del año por culpa de la primera semana que no está completa. Siguiendo con nuestro calendario de 7 días semanales, sabemos que:

- Si el año empieza en lunes (día 1 de la semana) no hay que quitar ningún día.

- Si el año empieza en martes (día 2 de la semana) hay que quitar 6 días.
- Si el año empieza en miércoles (día 3 de la semana) hay que quitar 5 días.
- ...

Si lo ponemos en una tabla:

Primer día de la semana	1	2	3	4	5	6	7
Número de días a quitar	0	6	5	4	3	2	1

Lo que tenemos que hacer es buscar una fórmula que nos convierta cada número de la primera fila en el correspondiente de la segunda. Ignorando, de momento, la primera columna, en el resto lo que hacemos es "dar la vuelta al número" alrededor del número de días de la semana. Fíjate que si sumas los dos números de cada columna (salvo en la primera) ¡siempre sale 8! Por tanto, para saber el número de días a quitar podemos restar a 8 el número del primer día de la semana.

i	1	2	3	4	5	6	7
$8-i$	**7**	6	5	4	3	2	1

Nos queda arreglar la primera columna, donde nuestra fórmula no sirve. Lo que tenemos que hacer es buscar una forma de convertir un 7 en un 0 sin que afecte a los números del 1 al 6. Y la forma de conseguirlo es usar el *operador módulo*. Si calculamos el resto de dividir el resultado que tenemos ahora por 7, los números entre 1 y 6 no se ven afectados, y el 7 se convierte en un 0:

i	1	2	3	4	5	6	7
$(8-i)\%7$	**0**	6	5	4	3	2	1

¡Ya tenemos la expresión! Esto nos permite calcular los días iniciales sobrantes, que tenemos que quitar al número total de días del año. El resultado lo dividimos de forma entera por 7 para saber las semanas completas y tenemos el resultado que nos están pidiendo.

Información

En lugar de utilizar el operador módulo para resolver el problema del caso donde el año empieza el primer día de la semana, se podría utilizar un condicional (`if`). Pero lo que queremos es practicar el uso de expresiones, de ahí que hayamos buscado la alternativa de usar el operador módulo.

Lo único que falta es *generalizar* nuestras fórmulas, porque hemos asumido que tenemos 7 días a la semana por simplicidad, cuando en el problema puede darnos un valor distinto. Pero una vez que se tiene la idea general con nuestras semanas de 7 días, deducir la fórmula completa no te costará mucho.

```
1 unsigned int anyo, semana, inicio;
2 cin >> anyo >> semana >> inicio;
3
```

```
4 // Quitamos a los días del año los que se pierden
5 // en la primera semana.
6 anyo -= (semana + 1 - inicio) % semana;
7
8 cout << anyo / semana << "\n";
```

Información

En este código, las variables se declaran `unsigned int` que significa que guardarán solo *enteros sin signo* (números naturales). Esto es posible porque ninguno de los tres datos tendrá nunca valores negativos.
De todas formas, no importa si tu solución usa `int`, porque los números de la entrada no son tan grandes como para que la diferencia sea importante. Utilizar `int` o `unsigned int` termina siendo una cuestión de preferencia personal aunque en un concurso, donde la velocidad escribiendo el código puede ser importante, tener que escribir el `unsigned` puede resultar un poco latoso.

Cuidado

Usamos `anyo` en lugar de `año` para la variable que guarda el número de días del año porque los símbolos no ingleses pueden deparar sorpresas si se utilizan en identificadores en los lenguajes de programación y normalmente es mejor evitarlos. Es habitual convertir la ñ en `ny`, `ni` (`anio`) o incluso en, simplemente, `n`, aunque esta última sustitución no queda muy bien en este caso.

332 Pesando carretas ★

Categorías

- Expresiones
- Matemáticas

Resumen del enunciado

Tenemos cinco carretas cuyo peso deseamos conocer. Para eso, nos dan el peso de *cada pareja de carretas*, es decir la suma de lo que pesa la primera carreta y la segunda, la primera y la tercera, etcétera. Eso nos da un total de 10 pesos (hay 10 parejas distintas de 5 carretas) y hay que dar los pesos individuales.

Solución

Como siempre, para resolver este problema lo primero es sentarse a pensar e intentar resolverlo con lápiz y papel. En este caso, es difícil mirar algunos de los ejemplos del enunciado y deducir rápidamente la solución, porque 10 números son muchos y encontrar el proceso a seguir no es trivial.

Una cuestión importante es que para nosotros las carretas *son indistinguibles*. Los pesos de las parejas nos las dan ordenadas de menor a mayor (esto nos vendrá bien, como veremos) y hay que escribir el peso de las carretas de mayor a menor. El detalle importante es que no nos preguntan qué carretas son las que están pesadas en cada uno de los números de la entrada, y eso es algo que podemos ahorrarnos deducir para los 10 pesos.

Para explicar cómo resolver el problema, vamos a denominar s_0, s_1, ... s_9 a cada uno de los 10 números que nos dan en la entrada. Usamos la letra s para indicar que es una *suma* de dos pesos. Por comodidad, asignaremos a s_0 el peso de la primera suma que nos dan, a s_1 el segundo, etcétera. Dado que están ordenados, también lo estarán nuestros s_x, algo que será importante más adelante.

Información

Numeramos los pesos del 0 al 9 porque al programar es muy habitual numerar las cosas empezando por 0, y no por 1 como se hace en el día a día. Es un detalle poco importante aquí y podríamos haber numerado de 1 a 10 sin que cambiara nada.

Denominaremos p_0, p_1, p_2, p_3, p_4 a los pesos de las 5 carretas, que son los valores que tenemos que descubrir. Como el enunciado nos pide que escribamos los pesos finales ordenados de mayor a menor, tendremos que controlar cómo los guardamos. Por coherencia con los s_x, vamos a buscar la forma de rellenar nuestros p_x también ordenados de menor a mayor, y luego los escribiremos desde el último al primero.

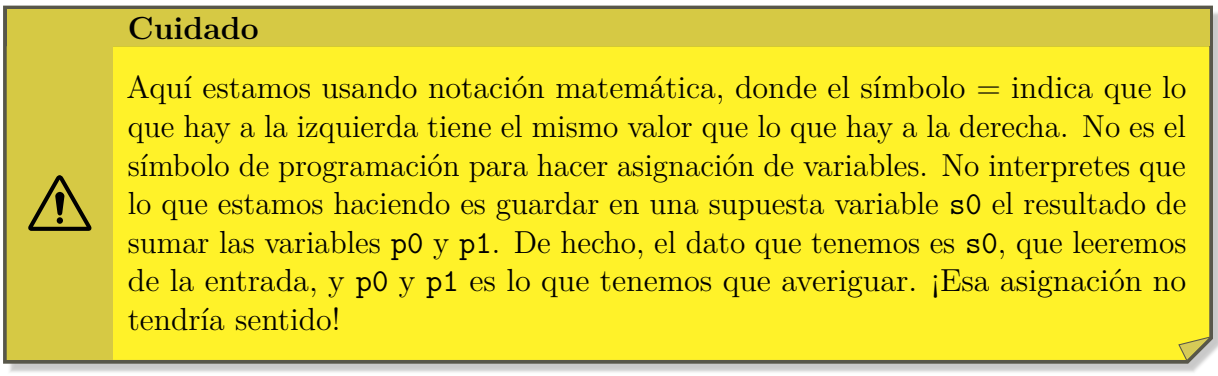
En esencia, lo que nos plantea este problema es un *sistema de ecuaciones* donde tenemos 5 incógnitas (nuestras p_x) y 10 ecuaciones (con las 10 sumas). El problema es que *no sabemos* qué pareja de $p_x + p_y$ están detrás de cada s_z, y por tanto ¡ni siquiera podemos escribir el sistema de ecuaciones! Tenemos que hacer uso de nuestras dotes de deducción para intentar extraer información de los datos que tenemos que nos permitan resolver el problema.

El primer dato importante es que en los 10 pesos de la entrada están los pesos de cada pareja de carretas. Eso significa que *cada carreta ha sido pesada cuatro veces*, haciendo pareja con cada una de las otras 4. El dato que nos proporciona esto es que la suma de todos los pesos de la entrada s_x es cuatro veces la suma de todas las carretas, los p_x que tenemos que descubrir. Escrito de otra forma, lo que sabemos es que:

$$p_0 + p_1 + p_2 + p_3 + p_4 = \frac{s_0 + s_1 + ... + s_9}{4}$$

No parece mucho pero ¡por algo hay que empezar! Otra cosa adicional es que antes hemos dicho que no sabemos qué carretas se están sumando en cada s_x pero eso es una verdad a medias. En realidad la suma más pequeña (s_0) tiene que ser, irremediablemente, el resultado de sumar los pesos de las dos carretas más ligeras, cuyos pesos hemos decidido llamar p_0 y p_1. Esto mismo ocurre con la suma más grande (s_9) que irremediablemente tiene que venir del peso de las dos carretas más pesadas:

$$s_0 = p_0 + p_1$$
$$s_9 = p_3 + p_4$$



Cuidado

Aquí estamos usando notación matemática, donde el símbolo = indica que lo que hay a la izquierda tiene el mismo valor que lo que hay a la derecha. No es el símbolo de programación para hacer asignación de variables. No interpretes que lo que estamos haciendo es guardar en una supuesta variable `s0` el resultado de sumar las variables `p0` y `p1`. De hecho, el dato que tenemos es `s0`, que leeremos de la entrada, y `p0` y `p1` es lo que tenemos que averiguar. ¡Esa asignación no tendría sentido!

Las dos ecuaciones anteriores ¡son muy útiles! Nos relacionan dos datos que conocemos (s_0 y s_9) con cuatro de los cinco que no, y eso podemos usarlo en la primera ecuación para *quitar cuatro de las cinco incógnitas*:

$$p_0 + p_1 + p_2 + p_3 + p_4 = \frac{s_0 + s_1 + ... + s_9}{4}$$

$$(p_0 + p_1) + p_2 + (p_3 + p_4) = \frac{s_0 + s_1 + ... + s_9}{4}$$

$$s_0 + p_2 + s_9 = \frac{s_0 + s_1 + ... + s_9}{4}$$

$$p_2 = \frac{s_0 + s_1 + ... + s_9}{4} - s_0 - s_9$$

Fíjate que en esta fórmula todo lo que queda a la derecha *son datos que conocemos* por lo que nos da una forma de calcular p_2, una de nuestras 5 incógnitas. ¡Ya solo nos quedan 4!

Igual que hemos deducido antes que $s_0 = p_0 + p_1$ (y simétricamente para s_9) ¿hay alguna otra pareja de pesos que sepamos que están escondidos en algún s_x concreto? La respuesta es que sí. Pensando un poco, puedes confirmar que se cumple que el segundo peso más

ligero de la entrada tiene irremediablemente que estar compuesto por la suma de los pesos de la carreta que menos pesa, y la tercera. Simétricamente ocurre lo mismo con el segundo peso más alto de la entrada:

$$s_1 = p_0 + p_2$$
$$s_8 = p_2 + p_4$$

En este momento, estas ecuaciones tienen una ventaja para nosotros: el problema nos da los valores de s_1 y de s_8, y nosotros ya sabemos calcular p_2, por lo que gracias a ellas también sabemos calcular p_0 y p_4:

$$p_0 = s_1 - p_2$$
$$p_4 = s_8 - p_2$$

¡Ya tenemos tres! ¿Cómo conseguimos las dos que nos faltan? Pues, en realidad, de dos ecuaciones que ya teníamos arriba:

$$s_0 = p_0 + p_1$$
$$s_9 = p_3 + p_4$$

Estas dos ecuaciones las utilizamos al principio para conseguir calcular p_2. Pero ahora que ya sabemos calcular también p_0 y p_4 podemos usarlas para despejar p_1 y p_3 que eran los dos únicos pesos que nos faltaban.

```
unsigned int p0, p1, p2, p3, p4;
unsigned int s0, s1, dummy, s8, s9;
unsigned int sumaTodos;

cin >> s0 >> s1;
sumaTodos = s0 + s1;
cin >> dummy; sumaTodos += dummy; // s2
cin >> dummy; sumaTodos += dummy; // s3
cin >> dummy; sumaTodos += dummy; // s4
cin >> dummy; sumaTodos += dummy; // s5
cin >> dummy; sumaTodos += dummy; // s6
cin >> dummy; sumaTodos += dummy; // s7
cin >> s8 >> s9;
sumaTodos += s8 + s9;

sumaTodos /= 4; // sumaTodos = p0 + p1 + p2 + p3 + p4
p2 = sumaTodos - s0 - s9;

p0 = s1 - p2;
p4 = s8 - p2;

p1 = s0 - p0;
p3 = s9 - p4;

// Escribimos, de mayor a menor
cout << p4 << " " << p3 << " " << p2 << " " << p1 << " " << p0 << "\n";
```

583 Encuesta comprometedora ★

Categorías

- Expresiones
- Matemáticas
- Límite de la representación

Resumen del enunciado

Si se hace una encuesta con una pregunta comprometedora (por ejemplo "¿lleva usted ahora mismo rotos los calcetines?" o "¿hace más de tres días que no se ducha?") muy poca gente estará dispuesta a contestar con la respuesta poco aceptada socialmente ("sí" en este caso). Para poder hacer este tipo de encuestas sin comprometer a la verdad, se puede utilizar una técnica curiosa. Se pide al entrevistado que, antes de contestar, tire una moneda y mire el resultado, sin decirlo. Si sale, por ejemplo, cara, entonces *deberá contestar la respuesta comprometedora*, aunque sea falsa. Si sale cruz, entonces deberá responder con sinceridad. Con este modo de proceder, si un entrevistado nos dice que lleva los calcetines rotos no sabremos si lo dice de verdad, o porque le ha salido cara en la moneda. Por tanto, su dignidad no se pondrá en entredicho.

En cada caso de prueba nos dan cuánta gente ha contestado con la respuesta comprometedora y cuánta lo ha hecho con la otra y hay que decir el porcentaje real de personas que podemos deducir que cae dentro de la respuesta comprometedora (llevan los calcetines rotos, o hace más de 3 días que no se ducha).

Información

El enunciado nos dice que podemos confiar en que la moneda será "perfecta" (a la mitad de los encuestados les sale cara, y a la otra les sale cruz) y que la selección de dichos encuestados no tiene sesgo y podemos extrapolar los resultados a la población total.

Solución

La solución de este problema es muy corta, pero necesitamos pensar antes un poco. El enunciado nos dice que la moneda es perfecta, lo que significa que *a la mitad de los encuestados les saldrá cara*. Eso significa que al menos la mitad de ellos contestarán con la respuesta comprometedora ("¡sí llevo los calcetines rotos!"). Como son respuestas forzadas *tenemos que quitarlas* para no contarlas, porque estropean la estadística. El resto de respuestas comprometedoras sí serán ciertas, aunque no podremos saber de qué encuestados vinieron.

En el enunciado nos ponen dos ejemplos, dando primero cuántos contestaron con la respuesta comprometedora y después cuántos lo hicieron con la aceptada socialmente:

```
100 0
50 50
```

En el primer caso (`100 0`), de los 100 entrevistados *todos* contestaron con la respuesta comprometedora. En realidad, a la mitad del total de encuestados (50) les salió cara, y dieron esa respuesta de manera forzada. Por tanto, de esas 100 respuestas "sí", solo

tenemos garantía de que 50 sean verdad. Eso significa que tenemos a 50 personas que han respondido con sinceridad a la respuesta comprometedora y a 0 la otra. El porcentaje es fácil de sacar: podemos deducir que el 100% de la población lleva los calcetines rotos.

En el segundo caso (`50 50`) procedemos igual. Hay 100 encuestados en total, e irremediablemente hay 50 que se han visto obligados a contestar que "sí" porque les ha salido cara. Eso no nos deja a nadie contestando "sí" con sinceridad, y por tanto estamos en la situación opuesta: el 0% de la población (es decir, nadie) hace más de tres días que no se ducha.

El patrón de solución que emerge es calcular la suma de los dos números para saber la población total. La mitad de esa suma tendrá cara en la moneda, y hay que restar ese valor al primer número, con la cantidad de gente que contesta con la respuesta comprometedora. Una vez hecho eso tendremos los datos reales y podemos, ahora sí, sacar el porcentaje con ellos.

Para calcular el porcentaje, tenemos que dividir la cantidad de gente que contestó la pregunta comprometedora entre el total de encuestados que respondieron con sinceridad y multiplicar por 100. Si no multiplicamos por 100, tendremos un valor entre 0 y 1, y no entre 0 y 100 como se espera con el porcentaje.

Para poder expresar el cálculo matemáticamente, pongamos nombres a los datos que tenemos:

- c : número de gente que dan la respuesta comprometedora ("sí llevo los calcetines rotos")
- a : número de gente que da la respuesta aceptada socialmente
- $t = c + a$: número total de entrevistados

Con estas definiciones, el valor que tenemos que escribir es:

$$100 \cdot \frac{c - \frac{t}{2}}{\frac{t}{2}}$$

Fíjate que no usamos a (gente que da la respuesta aceptada socialmente), aunque su uso está escondido en el cálculo de t (número total de entrevistados).

Si traducimos esta fórmula a un lenguaje de programación hay que tener cuidado en dos aspectos. El primero es el orden de las operaciones. Nos piden el porcentaje sin decimales, para lo que basta con hacer la división entera. Pero hay que tener cuidado y hacer la división *después* de multiplicar. Si primero calculamos la división:

$$\frac{c - \frac{t}{2}}{\frac{t}{2}}$$

el numerador será siempre menor que el denominador y con división entera esto será siempre 0. Tenemos que multiplicar el numerador por 100 y luego ya hacer la división.

Información

También podríamos hacer la división normal, con decimales, para lo que necesitaríamos un tipo distinto al entero, y una vez calculado el porcentaje quitar los decimales.

El segundo problema es más complicado, y está relacionado con el anterior. El enunciado nos dice que podemos llegar a preguntar hasta 10^9 personas. ¡Eso es mucha gente! En el caso más extremo, puede ocurrir que *todas* contesten con la respuesta comprometedora. El resultado es que el valor que tendremos que calcular será:

$$\frac{100 \cdot 500000000}{500000000}$$

La multiplicación del numerador es demasiado grande como para que el resultado nos entre dentro de una variable de tipo `int`. Necesitamos utilizar un tipo de datos para guardar números especialmente grandes, como por ejemplo `long long`, donde el resultado de la multiplicación entra con mucha holgura.

Para hacer la multiplicación con ese tipo de datos, es suficiente con decirle al compilador que queremos que el número 100 lo considere no de tipo `int` sino de tipo `long long` y eso podemos pedirlo añadiendo `ll` (dos eles minúsculas) junto al valor.

```
unsigned int comprometida, aceptada; // Respuestas de cada tipo
unsigned int total;

cin >> comprometida >> aceptada;

total = comprometida + aceptada;

// A la mitad les sale cara y contestan obligados.
// No contestan con sinceridad la respuesta comprometida,
// ni podemos usarlos para el porcentaje.
comprometida -= total/2;
total -= total/2;

cout << 100ll * comprometida / total << '\n';
```

Hazlo por tu cuenta

Si analizas la fórmula que estamos calculando, es posible *simplificarla* para ahorrarnos algunas operaciones. Reescribe la solución usando una fórmula más corta.