Mi primer problema en *¡Acepta el reto!*

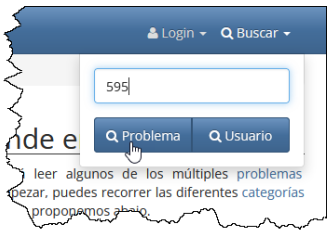
¡Acepta el reto![1] es una web que pone a disposición de sus usuarios un gran abanico de problemas de programación que pueden ser resueltos en C, C++ y Java. Las soluciones enviadas son automáticamente analizadas por un *juez automático*, que proporciona un veredicto sobre su corrección.

Para poder hacer envíos a los problemas disponibles, el primer paso es crearse una cuenta, si no tienes ya una. En la parte superior, el botón de "Login" despliega un pequeño formulario con un enlace de "Crear una cuenta" con el que, tras introducir unos pocos datos básicos, podrás crearte tu usuario en el portal.

Tendrás que validar la cuenta usando el mensaje de correo electrónico automático que se te envía, por lo que la cuenta de correo que pongas deberá poder recibir mensajes de cualquier sitio, algo que algunas cuentas académicas institucionales no permiten. Si no te llega el mensaje ¡pregunta en tu centro!

El siguiente paso es leer algunos de los problemas disponibles, elegir uno, resolverlo y ¡probar a enviarlo! Para empezar, te proponemos resolver uno muy sencillo, el problema 595 ¿En qué volumen? Todos los problemas en *¡Acepta el reto!* tienen un número que los identifica, en este caso el 595. Si conoces el número, puedes ir al enunciado del problema usando la búsqueda disponible, otra vez, en la parte superior.

[1] https://www.aceptaelreto.com



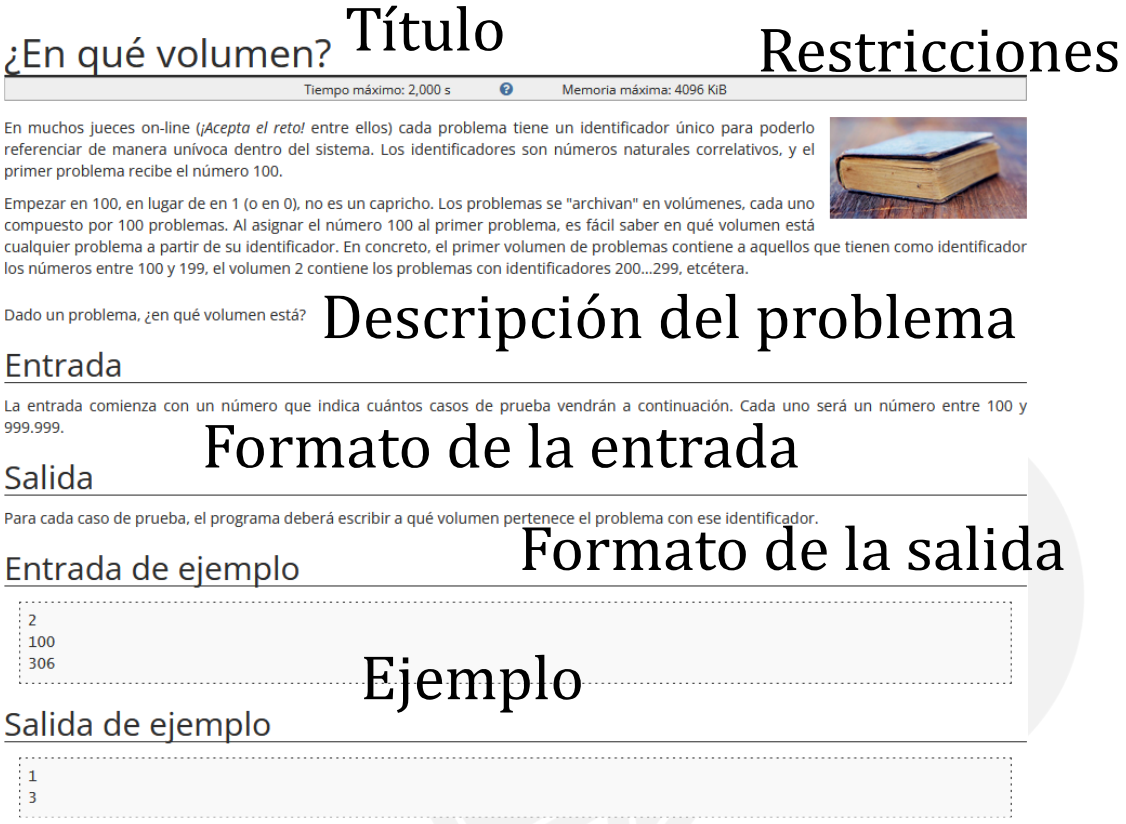
Los enunciados de todos los problemas en *¡Acepta el reto!* tienen el mismo esquema:

- Comienzan con un título y con restricciones de ejecución que deben cumplir las soluciones.

- Proporcionan una descripción del problema, normalmente dentro de una ambientación que hay que analizar para averiguar qué se nos está pidiendo.

- Luego se describe el formato de la entrada y de la salida con exactitud, indicando los posibles límites que existan en la entrada. Por ejemplo, nos pueden decir que nuestro programa nunca será probado con números mayores que un determinado valor. Algunas veces esos límites ¡son muy importantes! porque, junto con las restricciones indicadas antes, pueden forzarnos a tener que hacer soluciones específicas.

- Finalmente se da un pequeño ejemplo de entrada y de salida. El ejemplo de la entrada indica lo que *el juez automático* podría enviar a la solución por el teclado, y el ejemplo de la salida indica lo que se espera que el programa escriba para esa entrada.



El enunciado del problema 595 ¿En qué volumen? dice así:

> En muchos jueces on-line (*¡Acepta el reto!* entre ellos) cada problema tiene un identificador único para poderlo referenciar de manera unívoca dentro del sistema. Los identificadores son números naturales correlativos, y el primer problema recibe el número 100.
>
> Empezar en 100, en lugar de en 1 (o en 0), no es un capricho. Los problemas se "archivan" en volúmenes, cada uno compuesto por 100 problemas. Al asignar el número 100 al primer problema, es fácil saber en qué volumen está cualquier problema a partir de su identificador. En concreto, el primer volumen de problemas contiene a aquellos que tienen como identificador los números entre 100 y 199, el volumen 2 contiene los problemas con identificadores 200…299, etcétera.
>
> Dado un problema, ¿en qué volumen está?
>
> **Entrada**
>
> La entrada comienza con un número que indica cuántos casos de prueba vendrán a continuación. Cada uno será un número entre 100 y 999.999.
>
> **Salida**
>
> Para cada caso de prueba, el programa deberá escribir a qué volumen pertenece el problema con ese identificador.
>
> **Entrada de ejemplo**
>
> 2
> 100
> 306
>
> **Salida de ejemplo**
>
> 1
> 3

Lo que nos piden es que dado un número de problema digamos en qué volumen está. Si lo piensas un poco, te darás cuenta de que para saber el volumen en el que está un problema solo hay que dividir por 100 su identificador.

En *¡Acepta el reto!* es habitual que los programas tengan que contestar muchas veces a la misma pregunta. En este caso, eso significa que el *juez automático* no da un único número de problema para que escribamos su volumen, sino que nos dará muchos. La descripción de la entrada nos avisa de que, antes de leer el primer número de problema, tenemos que leer por cuántos problemas nos van a preguntar, y luego ya van todos ellos.

Para confirmar que hemos entendido el problema, podemos utilizar el ejemplo del enunciado. El primer número, el 2, nos dice que nos van a preguntar por dos problemas. Después va un 100, que es la primera de las dos preguntas. Nuestra intuición es que el problema 100 está en el volumen 1 (100/100 = 1). Si miramos en la salida de ejemplo, vemos que lo primero que tiene que escribir la solución es un 1, lo que nos dice que, de momento, estamos entendiendo bien el problema.

El siguiente número en la entrada del ejemplo es un 306. Nuestra intuición es que ese problema irá en el volumen 3. De nuevo, mirando en la salida del ejemplo, vemos que se espera que escribamos un 3, lo que confirma nuestra idea.

El enunciado muestra por separado la entrada de ejemplo (lo que el programa leerá por teclado) y su salida (lo que el programa escribirá). Es importante saber que *no* hay que leer toda la entrada de teclado y luego escribir toda la salida, sino que podemos ir leyendo cada *caso de prueba* y escribir la salida en ese momento, en lugar de esperar hasta el final. Eso significa que nuestro programa leerá el 2 y el 100, escribirá el 1 como respuesta, luego leerá el 306, escribirá el 3 de la respuesta y terminará.

En *¡Acepta el reto!* hay muchos problemas que siguen este mismo *esquema de la entrada*: primero el número de *casos de prueba* que hay que contestar, y luego esos casos de prueba. El programa por tanto lee el primer número, y luego va leyendo cada caso de prueba y contestándolo en el momento. Si estás empezando a programar, puede que aún no sepas hacer un programa que repita algo varias veces (para responder a múltiples casos de prueba), de manera que lo mejor es que uses un *esqueleto de solución*, aunque, de momento, no sepas lo que significa todo lo que aparece ahí. A continuación, te mostramos el esqueleto en C++ que te proponemos usar para empezar cuando el esquema de la entrada sea el que acabamos de ver.

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

void casoDePrueba() {

   // TU CÓDIGO AQUÍ

} // casoDePrueba

int main() {
   unsigned int numCasos;

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

   return 0;
}
```

Si sabes programar, no te resultará difícil entender lo que hace este código. Si no, quizá reconozcas algunas cosas (`#include <iostream>`, `int main()`) pero no entiendas otras. No te preocupes. Lo importante es que el código para contestar a un *caso de prueba* hay que ponerlo en la zona marcada y el resto se encargará de atender al esquema de la entrada que conocemos.

Quitando la ambientación lo que nos pide el problema es que, en cada caso de prueba, leamos un número y escribamos el resultado de dividirlo por 100. El código en C++ que hace eso es fácil:

```
int numProblema;    // Variable para guardar la entrada
cin >> numProblema; // Leemos un número del teclado
cout << numProblema / 100 << "\n"; // Escribimos el resultado
```

Aunque las razones no importan ahora, en *¡Acepta el reto!* usa `"\n"` en lugar de `endl` para cambiar de línea. Es importante no olvidar añadir el salto, para que la respuesta de cada caso de prueba vaya en su propia línea. Además, fíjate que nuestro problema no es muy amigable con el usuario. Si has hecho algunos problemas introductorios de programación, quizá sientas la tentación de resolver el problema con algo así:

```
int numProblema;    // Variable para guardar la entrada
cout << "Dime el número de problema: "; // ¡NO HAGAS ESTO!
cin >> numProblema; // Leemos un número del teclado
cout << "Ese problema está en el volumen "; // ¡NO HAGAS ESTO!
cout << numProblema / 100 << "\n"; // Escribimos el resultado
```

Si resuelves así el problema, el juez automático te dirá *que está mal* porque la salida de tu programa no será *exactamente* la que él espera. En el ejemplo de la salida del enunciado no aparece texto de ningún tipo salvo la respuesta en sí misma. Por tanto, ¡no escribas nada que el enunciado no muestre!

Ahora que sabemos cómo resolver un caso de prueba, nos falta meterlo en el *esqueleto de solución* que veíamos antes. Al final, la solución completa será algo así:

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

void casoDePrueba() {
   int numProblema;    // Variable para guardar la entrada
   cin >> numProblema; // Leemos un número del teclado
   cout << numProblema / 100 << "\n"; // Escribimos el resultado
} // casoDePrueba

int main() {
   unsigned int numCasos;

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

   return 0;
}
```

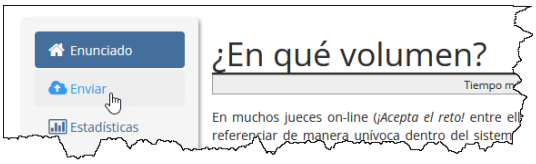
¡Pues ya lo tenemos! Antes de enviar tu solución al juez, es preferible que lo pruebes en tu ordenador por si has cometido algún error obvio. Utiliza tu entorno de desarrollo favorito y cuando lances el programa puedes escribir el ejemplo y confirmar que tu programa contesta lo que se espera. Fíjate que la entrada que escribas y su salida se entrelazarán. ¡Es lo esperado! Como decíamos antes, en los enunciados la entrada y la salida de ejemplo se muestran por separado por claridad, pero si lo ejecutas, el resultado será algo distinto. En este caso, al probar tu programa con el ejemplo deberías ver algo así:

```
2      ⌨  Número de casos de prueba
100    ⌨  Primer caso de prueba
1      💻  Salida del primer caso de prueba
306    ⌨  Segundo caso de prueba
3      💻  Salida del segundo caso de prueba
```

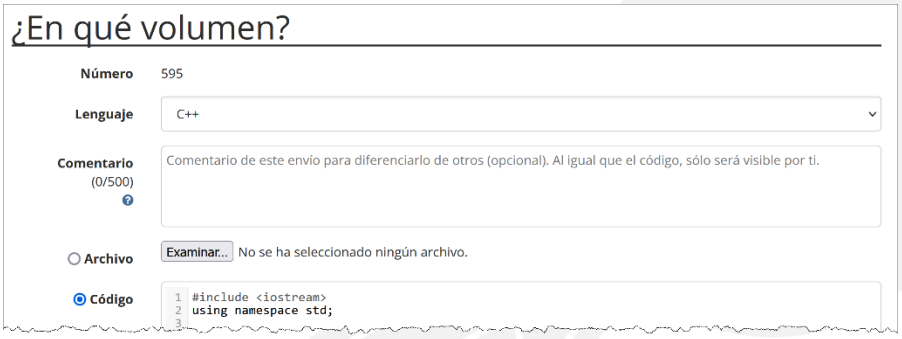
Para diferenciar lo que tienes que escribir tú y lo que genera el programa hemos añadido el icono ⌨ en las líneas escritas por ti, y el icono 💻 en las generadas por el programa, junto con una pequeña descripción de qué es cada línea.

En problemas más complicados seguramente quieras probar con más casos de los que muestra el enunciado. Piensa que el juez automático probará tu solución con muchísimos casos. El ejemplo del enunciado sirve para que confirmes que has entendido lo que se pide, pero no pretende ser una forma de probar exhaustivamente tu solución. Incluso aunque tu problema funcione con el ejemplo, puede ocurrir que luego el juez encuentre algún error que tú no has visto. ¡Es parte del juego!

Una vez que tengas cierta confianza de que la solución está bien, es el momento de enviarlo al juez. Para eso, después de meter tu usuario y tu contraseña, pulsa sobre "Enviar" a la izquierda del enunciado:



Esto te lleva a una página donde puedes o bien escribir directamente el código fuente de la solución, o bien enviar el archivo donde lo tienes.



Elige la opción que más te guste, confirma que tienes seleccionado a C++ como lenguaje, pulsa el botón "Enviar" y ¡buena suerte!

Si todo va bien, deberías ver tu primer envío aceptado (AC), junto con el tiempo y memoria que ha consumido su ejecución.

Si hay algún problema, el juez te emitirá un veredicto *de error*, que indica que algo ha ido mal y tu solución no le ha gustado. Hay varios veredictos posibles que dan una pequeña indicación de qué puede estar sucediendo:

- *Presentation error* (PE): ocurre si el programa parece estar bien, pero escribe espacios o saltos de línea sobrantes. Por ejemplo, si en la solución que hemos escrito hubiéramos puesto `"\n\n"` (escribiendo dos saltos de línea en lugar de solo uno) habríamos recibido este veredicto.

- *Wrong answer* (WA): ocurre si la salida del programa no coincide con lo esperado. Habríamos recibido este veredicto si, por ejemplo, hubiéramos dividido por 10 en lugar de por 100, o si hubiéramos añadido los mensajes de "Dime el número de problema:" y "Ese problema está en el volumen" que veíamos antes. El juez es poco amable y no da más pistas del error, ni te dice algún caso de prueba que te falle. ¡Eso tendrás que descubrirlo tú!

- *Compilation error* (CE): ocurre si el código de la solución tiene algún problema y no compila. Por ejemplo, si se nos olvida algún punto y coma, o no declaramos una variable recibiremos este veredicto. *¡Acepta el reto!* nos muestra en ese caso el error de compilación sufrido en la página del envío.

- *Run-time error* (RTE): el programa ha hecho una operación incorrecta y ha tenido que ser detenido antes de terminar. Nos habría ocurrido si, por ejemplo, en lugar de dividir por 100 hubiéramos dividido por 0. Es imposible dividir por 0, y si le pedimos a un ordenador que lo haga, detiene la ejecución del programa de manera abrupta porque no sabe como seguir.

- *Time-limit exceeded* (TLE): ocurre si el problema estaba tardando mucho en contestar a todos los casos de prueba y el juez automático se ha cansado de esperar. El tiempo es una de las restricciones que aparecen en la parte inicial del enunciado. En los problemas introductorios, es muy raro sufrir este veredicto, aunque a veces ocurre si se usa `endl` en lugar de `"\n"`. Y, desde luego, en problemas más difíciles este veredicto es un verdadero quebradero de cabeza.

- *Memory limit exceeded* (MLE): similar al anterior, pero si se consume mucha memoria. Es también poco habitual en los problemas introductorios.

Hay algunos otros veredictos posibles, pero son muy atípicos, por lo que no merece la pena describirlos. Recuerda que en la parte de Documentación de la web tienes información sobre todos ellos y alguna otra información que puede resultarte interesante.

Y ¡eso es todo! Ahora que tienes tu primer problema resuelto en el juez, ¿no tienes gusanillo de resolver otros? Si no sabes por dónde seguir, aquí van algunas ideas:

- 635 Cinquecento: es muy parecido al que hemos visto. Te resultará fácil descubrir que también hay que dividir por 100. Pero hay que hacer algunas otras operaciones aritméticas con el número leído. ¡Descúbrelas!

- 373 Cubos visibles: ¿cuántos cubitos son visibles en las caras de un cubo de Rubick de *n* cubitos por cara? Para calcularlo ¡no pienses en los que ves, si no en los que no! Todo es más fácil si calculas el número de cubitos total que forman el cubo, y restas los que *quedan dentro*.

- 621 La otra página: se puede resolver fácilmente si se utiliza un condicional (un `if`). Pero, como ejercicio intelectual, también se puede resolver con una expresión aritmética y, de hecho, ¡de varias formas!