Cómo resolver los problemas para el juez automático[1]

Alberto Verdejo

Todos los problemas son aplicaciones de consola que leen los datos por la entrada estándar y escriben los resultados por la salida estándar. La entrada siempre se va a ajustar a lo que describa el enunciado, por lo que no hace falta preocuparse por eso. De igual forma, la salida debe ajustarse exactamente a lo que indique cada enunciado.

Para cada problema, el juez dispone de un conjunto de casos de prueba. Cuando recibe el código de una supuesta solución, lo compila y lo ejecuta, enviándole por la entrada estándar esos casos de prueba. El programa escribirá en la salida estándar los resultados, que serán comparados por el juez con los resultados correctos, especificados por el profesor. El veredicto emitido dependerá del resultado de esa comparación.

Es importante resaltar el hecho de que el juez automático exige que todos los programas acaben con éxito (resultado de ejecución 0). Esto supone que en las soluciones programadas en C o C++ es importante acabar la función `main` con un `return 0`. En otro caso, el juez dará como veredicto un error de ejecución.

Todos los enunciados contienen un ejemplo de entrada y un ejemplo de salida, que se muestran para ilustrar el problema y que se separan entre sí por claridad y simplicidad. Pero las soluciones no tienen que leer toda la entrada, procesarla y luego escribir toda la salida. Más bien al contrario. Hacerlo así requeriría que el programa utilizara más memoria para guardar toda la entrada antes de empezar a hacer algo útil. Más abajo aparecen los esquemas de programa que deben seguirse.

Los programas deben probarse antes de ser enviados al juez automático, compilándolos y ejecutándolos. Y no solamente con los ejemplos sencillos que aparecen en el enunciado (que sirven más para aclarar el problema que para comprobar la corrección de la solución) sino con otros desarrollados para intentar descubrir los errores que pueda haber en la solución.

Los ficheros con las implementaciones de las estructuras de datos vistas en clase pueden utilizarse directamente en un `#include` y no es necesario subirlos al juez cuando se envíe una solución. Están instalados junto con el compilador que utiliza el juez. Si es necesario modificar esa implementación, cambia el nombre del fichero antes y súbelo junto con el resto de ficheros cuando hagas el envío al juez.

Importante: Todos los ficheros enviados al juez deben contener un comentario en la primera línea con el *número de grupo* y los *nombres y apellidos* de los alumnos.

Esquemas de programas

Todos los problemas utilizan el mismo esquema: dado un caso de entrada hay que escribir algo sobre él. Para que se pueda probar con certeza que el programa funciona, este tendrá que ser probado con numerosos casos de entrada, y dar la respuesta correcta para todos ellos. Para hacerlo, hay tres alternativas o estilos de entrada:

1. Al principio de la ejecución, el programa recibe el número de casos de prueba que se utilizan.
2. El programa va leyendo casos de prueba hasta que se encuentra con un caso de prueba especial.
3. El programa va leyendo casos de prueba hasta que se alcanza el final de la entrada (no quedan más datos).

Dependiendo de si es una u otra alternativa el esquema general del programa será algo distinto. Pero en cualquier caso todos deben estar bien estructurados y comentados.

[1] Este documento es una adaptación de la documentación de ayuda que aparece en las webs `www.aceptaelreto.com` y `www.programa-me.com`.

Casos de prueba limitados

Para el primer tipo de entrada, el esquema debe ser el siguiente:

```
// Grupo XYZ, Fulano y Mengano

// Comentario general sobre la solución,
// explicando cómo se resuelve el problema

#include <iostream>
#include <...>

#include "..."  // propios o los de las estructuras de datos de clase

// función que resuelve el problema
// comentario sobre el coste, O(f(N)), donde N es ...
Solucion resolver(Datos datos) {
    ...
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    ⟨ leer los datos de la entrada ⟩

    Solucion sol = resolver(datos);

    ⟨ escribir sol ⟩
}

int main() {
    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i)
        resuelveCaso();
    return 0;
}
```

La función `main` simplemente tiene el bucle que recorre todos los casos. Para cada caso, se llama a la función `resuelveCaso` que lee los datos asociados a un caso de prueba, lo resuelve y escribe la solución. Para resolver un caso se utiliza la función `resuelve` que recibe los datos concretos del problema, ya construidos a partir de la entrada, y los procesa. Esta es la función principal, que debe ser la que mejor documentada esté y de la que se indicará cuál es su complejidad.

Casos de prueba ilimitados acotados por caso de prueba especial

Con el segundo tipo de entrada, la estructuración en funciones es la misma. Solamente cambia el bucle en la función `main` y cómo se detecta la condición de terminación en la función `resuelveCaso`:

```
 ... igual al caso anterior ...

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    ⟨ leer los datos de la entrada ⟩

    if (caso especial)
       return false;

    Solucion sol = resolver(datos);

    ⟨ escribir sol ⟩

    return true;
}

int main() {
    while (resuelveCaso());
    return 0;
}
```

Casos de prueba ilimitados

Para el tercer caso, solamente cambia la forma de detectar la terminación de los casos de prueba:

```
 ... igual al caso anterior ...

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    ⟨ leer los datos de la entrada ⟩

    if (!std::cin)  // fin de la entrada
       return false;

    Solucion sol = resolver(datos);

    ⟨ escribir sol ⟩

    return true;
}

int main() {
    while (resuelveCaso());
    return 0;
}
```

Ejemplo

Supón que el problema pide calcular la altura de un árbol binario (de números enteros no negativos), con la siguiente descripción de la entrada y la salida.

Entrada La entrada comienza con el número de casos que vienen a continuación. Cada caso consiste en una cadena de caracteres con la descripción de un árbol binario: el árbol vacío se representa con un punto (`.`); un árbol no vacío se representa con un `*` (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

Salida Para cada árbol se escribirá una línea con su altura.

Una solución correcta podría ser la siguiente (se muestra el contenido del fichero `solucion.cpp`):

```
// TAIS62, Isabel Pita y Alberto Verdejo

// Construye un árbol binario a partir de la entrada y después
// calcula su altura de forma recursiva

#include <algorithm>
#include <iostream>

#include "bintree_eda.h"

// lee un árbol binario de la entrada estándar
template <typename T>
bintree<T> leerArbol(T vacio) {
   T raiz;
   std::cin >> raiz;
   if (raiz == vacio) { // es un árbol vacío
      return {};
   } else { // leer recursivamente los hijos
      auto iz = leerArbol(vacio);
      auto dr = leerArbol(vacio);
      return {iz, raiz, dr};
   }
}

// dado un árbol binario, calcula su altura
// lineal en el número N de nodos del árbol, O(N)
unsigned int altura(bintree<char> const& arbol) {
   if (arbol.empty())
      return 0;
   else
      return 1 + std::max(altura(arbol.left()), altura(arbol.right()));
}

// resuelve un caso de prueba
void resuelveCaso() {
    auto arbol = leerArbol('.');
    int sol = altura(arbol);
    std::cout << sol << "\n";
}

int main() {
    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i)
        resuelveCaso();
    return 0;
}
```

Probar los programas antes de enviarlos al juez

Las aplicaciones que se envían al juez leen los datos de la entrada estándar y escriben los resultados por la salida estándar. Sin embargo, cuando la entrada del programa es larga, resulta muy conveniente leer los datos de un fichero durante la fase de prueba. De esta forma no es necesario introducir los datos por teclado en cada ejecución, sino que automáticamente el programa lee el fichero.

Para conseguirlo (sin tener que modificar el código al subirlo al juez) basta con introducir las siguientes instrucciones en el programa, que redireccionan a la entrada estándar el contenido de un fichero cuando no se está ejecutando el programa bajo el juez. El efecto es un programa que cuando se ejecute en el juez leerá los datos de la entrada estándar (como el juez requiere) pero cuando se ejecute localmente leerá los datos del fichero que se indica en las instrucciones.

El programa principal es el siguiente:

```
#include <fstream>

...

int main() {
    // ajustes para que cin extraiga directamente de un fichero
    #ifndef DOMJUDGE
      std::ifstream in("casos.txt");
      auto cinbuf = std::cin.rdbuf(in.rdbuf());
    #endif

    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i)
        resuelveCaso();

    // para dejar todo como estaba al principio
    #ifndef DOMJUDGE
      std::cin.rdbuf(cinbuf);
      system("PAUSE");
    #endif

    return 0;
}
```