Práctica 3
Uno Solo

Fecha de entrega: 28 de marzo

Fuente: Wikimedia.org

1. Historia y breve descripción del juego

El solitario conocido como Uno Solo o Senku es un juego de tablero abstracto en el que normalmente se juega con un tablero con agujeros y clavijas o bolas que se colocan en dichos agujeros.

Según Wikipedia, la primera evidencia del juego data de la corte de Luis XIV, según el grabado hecho en 1687 por Claude Auguste Berey donde aparece Anne de Rohan-Chabot, Princesa de Soubise, jugando a dicho solitario. La edición de agosto de 1687 de la revista literaria francesa "Mercure galant" contenía una descripción del tablero, de las reglas y de problemas de ejemplo.

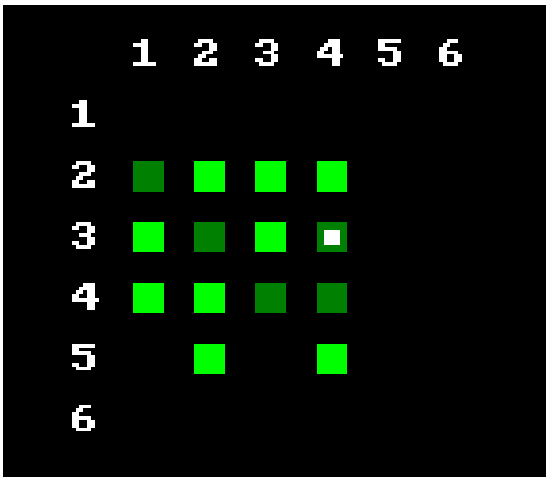
En este solitario, se empieza con un tablero en modo de rejilla con piezas colocadas sobre las casillas. El jugador debe mover una pieza en cada turno. Las piezas sólo pueden moverse "saltando" sobre otra ficha hasta una casilla libre de forma que la pieza sobre la que se salta es eliminada del tablero, como en las damas. Sólo se puede saltar una pieza, y el salto debe ser en horizontal o en vertical, nunca en diagonal. El objetivo del juego es eliminar todas las piezas, dejando sólo una en el tablero. Aunque nosotros implementaremos una variante de este solitario en la que la última pieza debe acabar en una casilla específica.

Existen muchas variantes de este juego que parten de formas concretas del tablero (como por ejemplo el tablero en cruz que muestra la primera imagen. Nuestro tablero podrá tener formas más libres como veremos más adelante.

2. El programa

El programa implemtará la dinámica del juego, y para dibujar el tablero utilizará caracteres con distintos colores de fondo para las fichas.

Para mantener el estado del tablero el programa usa un array bidimensional de celdas, de tamaño `DIM` x `DIM` (en los ejemplos de este enunciado `DIM=6`). Las celdas pueden tener valor `NULA`, `VACIA` o `FICHA`, que representan respectivamente el estado de una celda del tablero que no se puede utilizar en esta partida (es decir, no se pueden colocar fichas sobre él), una celda vacía y una celda en la que hay una ficha.



La imagen anterior representa un tablero donde las celdas en negro son nulas (no forman parte del tablero "jugable" en esta partida), las que están en verde apagado están vacías y las que están en verde brillante son celdas con ficha. La celda que tiene el símbolo en blanco en el centro representa la meta de esta partida.

Tal y como muestra la imagen anterior en cada turno el jugador deberá introducir la ficha que quiere mover introduciendo **primero la fila y luego la columna**. Si la fila y columna corresponden a una celda con ficha que se puede mover, el programa preguntará en qué

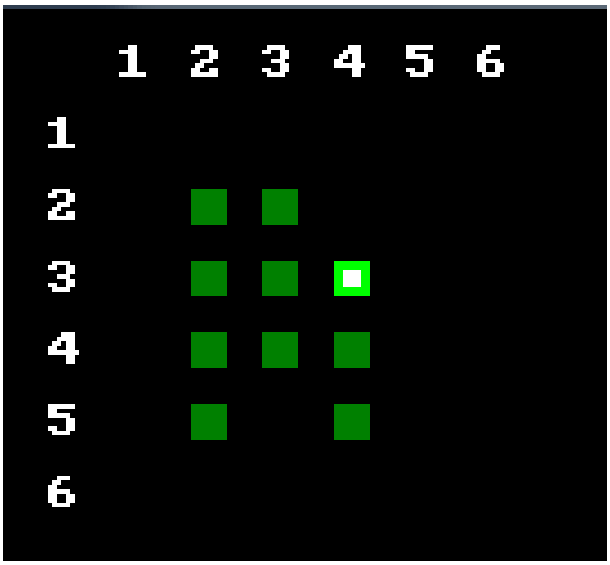
dirección quiere mover la ficha indicándole **sólo aquellas opciones posibles** para dicha ficha. De forma general el jugador introducirá

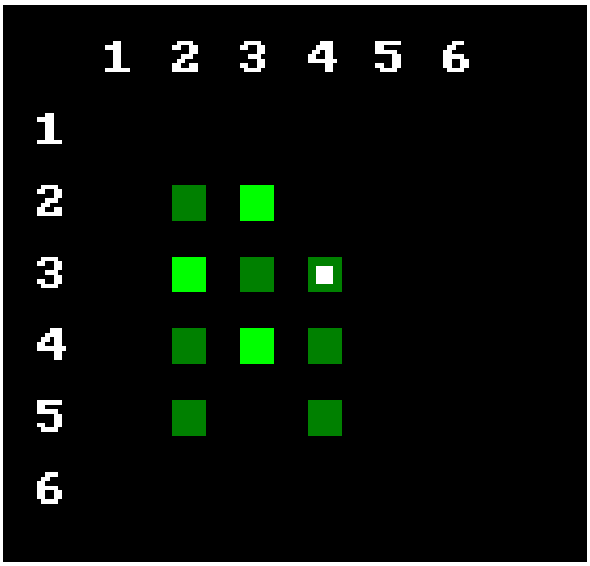
- 1 para mover Arriba
- 2 para mover a la Derecha
- 3 para mover Abajo
- 4 para mover a la Izquierda
- 0 si ha decidido que no quiere mover esa ficha y quiere seleccionar otra

Si la celda seleccionada por el jugador no es válida el programa indicará **la razón por la cual no es válida**. A saber:

- Se trata de una **posición fuera del tablero**. Por ejemplo, cualquier celda que incluya la fila/columna 0 o la fila/columna 7 en la imagen anterior.
- Se trata de una **celda sin ficha**, ya sea porque está vacía o porque es una celda nula del tablero. Por ejemplo, las celdas (1,1) o (2,1) de la imagen anterior.
- Se trata de una celda con una **ficha que no se puede mover**. Por ejemplo, la celda (2,4) o la celda (4,2) en la imagen anterior.

Además, el juego deberá informar si se llega a un punto en el que sobre el tablero queda una única ficha que está situada en la celda de meta (en ese caso el jugador habrá ganado) o en el que no se pueden mover más fichas (en ese caso el jugador habrá perdido). Las siguientes imágenes ilustran ambos casos.





3. Versión 1: Cargar, mostrar y guardar el juego

La primera versión, para tomar contacto con la práctica, el programa se encargará de cargar, mostrar y guardar en un fichero de texto el juego.

El juego se representa mediante un tipo de datos estructurado `tSoloPeg` con campos para:

- El tablero, que describimos más adelante.
- La fila y la columna en la que se encuentra la meta.
- El número de bolas iniciales del tablero.
- El número de movimientos realizados en la partida (que es igual al número de bolas eliminadas en el tablero).
- El estado del juego, que representamos mediante un enumerado para los posibles estados, a saber: `BLOQUEO, GANA, JUGANDO.`

Por su parte el tablero es una matriz bidimensional cuadrada de elementos de tipo `tCelda`, y dimensión constante `DIM`. Los posibles valores del tipo enumerado `tCelda` son: `NULA, VACIA y FICHA.` Aprovechando la posibilidad C++ de dar valor a los enumerados, asignaremos los valores enteros `0, 2 y 10` (respectivamente) a cada uno de los valores mencionados. Estos valores los usaremos para pintar el tablero como veremos más adelante.

3.1. Visualización del juego

A la hora de visualizar el tablero ten en cuenta que cada celda usa un espacio con un color de fondo que depende del valor de la celda. Salvo la meta, que usa el carácter `char`(254) en color blanco y el fondo será el correspondiente a celda vacía o con ficha.

Al dibujar el tablero por filas, ten en cuenta que las celdas de una fila aparecen separadas por un espacio con fondo negro, y entre filas se deja una línea. Cada fila empieza por el número de fila (índice + 1).

Sobre el tablero se mostrará también el número de bolas iniciales, el número de movimientos realizados, y los números de las columnas (índice + 1) del tablero.

Colores de fondo en la consola

Por defecto, el color de los caracteres es blanco, mientras que el color de fondo es negro. Podemos cambiar esos colores utilizando rutinas que son específicas de Windows, por lo que debemos ser conscientes de que el programa no será portable a otros sistemas.

En nuestro programa el color de los caracteres siempre es blanco, pero queremos poder cambiar el color de fondo para representar cada celda con un color diferente.

La biblioteca `windows.h` incluye subprogramas para manejar la consola. Uno de ellos es `SetConsoleTextAttribute()`, que permite ajustar los colores de fondo y primer plano (línea de los caracteres). Incluye en el programa esa biblioteca y este procedimiento:

```
void colorFondo(int color) {
   HANDLE handle = GetStdHandle(STD_OUTPUT_HANDLE);
   SetConsoleTextAttribute(handle, 15 | (color << 4));
}
```

Basta proporcionar un color para el fondo (0 a 15) y este procedimiento lo establecerá, con el color de primer plano siempre en blanco (15).

El color negro se representa con un 0, el verde apagado con el 2 y el verde brillante con el 10. Estos tres valores se deben establecer como los valores del enumerado `tCelda` (`NULA, VACIA y FICHA`), así, para mostrar cada celda primero se llama a `colorFondo` con el valor del enumerado de la celda.

3.2. Carga y guardado del juego

También queremos poder cargar y guardar el juego en mitad de una partida. Como podemos cambiar la dimensión del tablero para generar diferentes versiones del juego (de 3x3 a 10x10), los archivos de tableros deben comenzar con una línea que indique cuál es esa dimensión (`DIM`). En el caso de la carga, el tablero sólo se cargará si esa dimensión coincide con la dimensión actual de tablero en el programa. A continuación se guarda el número de bolas iniciales del tablero, y la fila y la columna de la meta.

Para el tablero, el archivo tiene cada fila dispuesta en una línea, y dentro de cada línea los valores están separados por tabuladores.

Tras el tablero se guardarán el número de movimientos realizados hasta ese momento en la partida. El archivo termina con esos puntos; no hay centinela (no se necesita).

Utiliza el siguiente ejemplo (archivo `soloPeg6.txt`) para hacer pruebas:

```
6   ← Dimensión
8   ← Nº de bolas iniciales
4   ← Fila de la meta
2   ← Columna de la meta
0    0    0    0    0    0
0    0    0    0    0    0
10   10   2    0    0    0      Tablero cuadrado
10   10   2    10   0    0      de DIM x DIM
2    10   10   0    0    0
10   0    0    0    0    0
0   ← Nº de movimientos
```

3.3. Implementación

Como hemos dicho anteriormente, esta primera versión del programa nos debe servir para probar que hemos implementado correctamente los subprogramas que nos permiten mostrar, cargar y guardar un juego. Por ello, la función `main()` tendrá el código necesario para comprobar que estos subprogramas funcionan correctamente, pero en esta práctica vamos a tener una primera toma de contacto con los módulos, organizando la implementación en tres archivos.

Mi primer proyecto multiarchivo

Crea un proyecto vacío y añade dos archivos de código C++ (`mainP3.cpp` y `soloPeg.cpp`) y uno de cabecera (`soloPeg.h`).

- `soloPeg.h`: Contendrá las declaraciones de los tipos de datos que necesitemos para el juego y las librerías necesarias para definir esos tipos de datos, así como los prototipos de los subprogramas que puede usar `main`. Los ficheros .h se denominan de cabecera (o *header* en inglés).
- `soloPeg.cpp`: Contendrá las implementaciones de los subprogramas cuyos prototipos estén definidos en `soloPeg.h`. Los ficheros .cpp (código fuente C++) son los archivos que implementan los prototipos del correspondiente archivo de cabecera. Tendremos un fichero .cpp por cada archivo de cabecera, cuya primera línea debe incluir el fichero .h correspondiente. Es decir, el archivo `soloPeg.cpp` debe comenzar con la directiva `#include "soloPeg.h".` A continuación incluirá las librerías (por ejemplo, `iomanip` o `windows.h`), las declaraciones (no declaradas en `soloPeg.h`) de constantes, tipos y prototipos necesarios para implementar lo subprogramas declarados en la cabecera, y las implementaciones de todos los prototipos.

```
// archivo soloPeg.h

#include <string>
…
// constantes y tipos
…
typedef struct {…} tSoloPeg;

// prototipos

bool cargar(…);
void guardar(…);
void mostrar(…);
…
```

```
// archivo soloPeg.cpp

#include "soloPeg.h"
#include <iostream>
…
// constantes y tipos (no declarados
…
// prototipos           en soloPeg.h)
…
// implementación de los prototipos
// declarados en soloPeg.h y aquí

void mostrar(…){ … }
…
```

- `mainP3.cpp`: Único archivo de un proyecto que puede definir la función `main` (punto de inicio de la ejecución de la aplicación). En este caso deberá incluir mediante la directiva `#include` el archivo de cabecera `soloPeg.h`, para que las declaraciones realizadas en `soloPeg.h` sean visibles, es decir, se puedan invocar. Además, este fichero también deberá incluir todas las librerías que precise el código del main para funcionar correctamente. En general sólo contienen la implementación de la función `main`, pero se puede añadir alguna función auxiliar, referente a la interacción con el usuario.

```
// archivo mainP3.cpp
// autores

#include <iostream>
…
#include "soloPeg.h"

// prototipos

bool partida(tSoloPeg & peg);

int main() {
  tSoloPeg peg;
  …
  if (cargar(peg,…)){
    partida(peg);
    guardar(peg,…);
  }
  …
  return 0;
}

bool partida(tSoloPeg & peg) {
  mostrar(peg);
}
…
```

Debes implementar, al menos, los siguientes subprogramas.

En `soloPeg`:

- ✓ `void mostrar(tSoloPeg const& peg)`: Muestra el número de movimientos realizados y el tablero con la partida según se ha descrito anteriormente.
- ✓ `bool cargar(tSoloPeg &/*sal*/ peg, string const& nombre)`: Intenta cargar desde el archivo llamado `nombre` el juego completo según el formato especificado anteriormente. Si no se puede cargar el archivo devuelve `false`, y `true` en caso contrario.
- ✓ `void guardar(tSoloPeg const& peg, string const& nombre)`: Guarda en el fichero `nombre` el juego que recibe como parámetro según el formato especificado anteriormente.

En `mainP3`:

- ✓ `bool partida(tSoloPeg &/*ent/sal*/ peg)`: Dado un juego ya inicializado, guía el desarrollo de una partida del juego. En esta versión simplemente lo muestra.

4. Versión 2: Implementando el juego

Aunque vamos a poder jugar partidas sucesivas al Uno Solo, vamos a empezar por implementar una partida del solitario.

4.1. La mecánica del juego

Suponiendo un tablero ya inicializado (por ejemplo cargado de `soloPeg6.txt`), el usuario deberá seleccionar una ficha del tablero especificando su fila y columna. Si la ficha no es válida, se le indicará por qué, y si es válida, se le indicarán los movimientos posibles. Una vez seleccionado un movimiento posible, se realizará el movimiento en el tablero y se volverá a mostrar el mismo. Los detalles de este funcionamiento ya se han explicado en la sección 2 de este enunciado.

Para gestionar los movimientos (jugadas), definiremos en el archivo de cabecera `soloPeg.h` un tipo de datos estructurado `tMovimiento` con campos para:

- La fila y la columna origen del movimiento. Hay que tener en cuenta que la numeración mostrada al usuario (de 1 a `DIM`) no corresponde con la interna del array (de 0 a `DIM-1`).
- La dirección escogida para el movimiento, que representamos mediante un entero sin signo de un byte (`uint8`): 0 (arriba), 1 (derecha), 2 (abajo) y 3 (izquierda). Además definimos las constantes `NUM_DIRS = 4`, y `const string NOMB_DIRS[NUM_DIRS] = {"Arriba", "Derecha", "Abajo", "Izquierda"};`

Para indicar al usuario las posibles direcciones en las que se puede mover la ficha elegida, según la situación del tablero, como sólo hay cuatro (`NUM_DIRS`) direcciones posibles, usaremos un array de booleanos de `NUM_DIRS` posiciones (define el tipo `tPosibles`) que representarán las direcciones en el orden: arriba, derecha, abajo e izquierda. Por tanto, si tenemos una variable `tPosibles posibles`, `posibles[dir]` indica si la ficha se puede mover en la dirección `NOMB_DIRS[dir]`.

El desarrollo de una jugada (movimiento de una ficha) consistirá en los siguientes pasos:

1. Leer el movimiento especificado por el usuario:
   - Fila y columna de una ficha del tablero con posibles movimientos (o 0 para salir)
   - Dirección del movimiento (una de entre las posibles o 0 para cambiar de ficha).
2. Ejecutar el movimiento:
   - Actualizar las fichas del tablero.
   - Incrementar el número de movimientos realizados.
   - Recalcular el estado del juego según el movimiento: `BLOQUEO, GANA, JUGANDO.` Este paso requiere comprobar varias cosas:
     - Si el usuario ha ganado la partida, es decir, si sólo queda una bola en el tablero y ésta está colocada en la celda de meta, el estado resultante será `GANA.`
     - En otro caso, hay que comprobar si todavía queda al menos una ficha con movimientos posibles. Si no es así, el usuario ha perdido y el estado resultante será `BLOQUEO`.

Además, si en el paso 1 el usuario introduce como fila un 0, el programa le preguntará si desea salvar la partida, si es así, se deberá solicitar el nombre del archivo e invocar al subprograma `guardar` que implementamos en la versión anterior.

4.2. Implementación

Para llevar a cabo esta versión deberás implementar, al menos, los siguientes subprogramas.

En `mainP3`:

- ✓ `bool partida(tSoloPeg &/*ent/sal*/ peg):` Dado un juego ya inicializado, realiza el bucle de movimiento de ficha explicado anteriormente. En cada vuelta del bucle mostrará el estado del tablero, leerá y ejecutará un movimiento, y terminará cuando el estado del tablero deje de ser `JUGANDO` (devolviendo `true`; es decir el juego ha terminado), o el usuario seleccione salir (devolviendo `false`).
- ✓ `bool leerMovimiento(tSoloPeg const& peg, tMovimiento &/*sal*/ mov):` Lee un movimiento válido y lo devuelve. Para ello, pregunta al usuario qué ficha quiere mover, muestra los movimientos posibles, y guarda la información dada por el usuario en `mov`. Si el usuario desea salir se devuelve `false`, y si se ha seleccionado un movimiento válido devuelve `true`.

En `soloPeg`:

- ✓ `bool posiblesMovimientos(tSoloPeg const& peg, tMovimiento mov, tPosibles/*sal*/ posibles):` Dada una casilla (la del movimiento) marca en `posibles` las direcciones en las que se podría mover. Si ninguna es posible devuelve `false`, y en otro caso `true`.
- ✓ `bool ejecutarMovimiento(tSoloPeg &/*ent/sal*/ peg, tMovimiento const& mov):` Dado un movimiento, se realiza el movimiento en el juego, modificándolo según se ha descrito.

4.3. Jugando varias partidas

Una partida consiste en la ejecución de la secuencia de pasos descrita en el apartado anterior hasta que el juego alcanza uno de los estados finales (`BLOQUEO, GANA`).

Ahora debemos permitir que nuestro programa juegue tantas partidas como el usuario desee. Seguirá la siguiente secuencia:

1. Se preguntará al usuario si quiere jugar una nueva partida o salir del programa.
2. En caso de que quiera jugar le debe permitir elegir cargar el tablero que desee. Si la carga se ha realizado con éxito, la partida comienza.
3. La partida proseguirá hasta que se alcance uno de los estados finales. Cuando eso suceda se volverá al punto 1.

5. Versión 3: Generación aleatoria de tableros

Vamos a añadir a nuestro programa la posibilidad de poder generar tableros de manera aleatoria. La versión que vamos a implementar, no es más que jugar al juego pero "al revés": partiendo de un tablero con todas las posiciones nulas excepto una única ficha que está en la meta, se irán generando **movimientos inversos** válidos que irán creando nuevas fichas en lugar de hacerlas desaparecer.

En nuestro programa se preguntará al usuario el número de bolas que se quieren poner (a más bolas más complejo será resolver el juego), y se intentará generar un tablero con ese número de bolas (movimientos inversos).

Las siguientes figuras representan la generación de un tablero mediante los dos primeros movimientos inversos:

La generación de tableros que vamos a implementar consiste en los siguientes pasos:

1. Inicialización del tablero con todas las posiciones nulas.
2. Fijar la meta en una posición aleatoria, y colocar una ficha en esa posición.
3. Mientras no se llegue al número de movimientos indicado por el usuario, se elegirá al azar una de las fichas, se buscará un movimiento inverso válido (si hay más de uno se elegirá también al azar), y se realizará el movimiento. Se considera que un movimiento inverso es válido si las dos siguientes celdas en esa dirección están vacías o nulas.

En las figuras, se puede ver que la meta elegida aleatoriamente se encuentra en la posición (3,4). Después, el primer movimiento inverso será usando una ficha aleatoria (que será la única existente, la ficha inicial) y con una dirección aleatoria (en este caso hacia abajo). Así, se colocan dos fichas en las siguientes posiciones en esa dirección, y la posición anteriormente ocupada por la ficha se indica como `VACIA` (no como `NULA`). Para el segundo movimiento inverso se elige de nuevo una ficha aleatoriamente, en el ejemplo (5,4), y una dirección aleatoria (en este caso sólo son válidos los movimientos hacia la derecha y hacia la izquierda). Igual que antes, se deja como `VACIA` la posición ocupada por la ficha, y se colocan fichas en las dos siguientes posiciones en esa dirección.

IMPORTANTE: Aunque el usuario elija un número M de bolas para generar el tablero, puede ocurrir que se llegue a un punto en que, aunque no se hayan hecho todos los movimientos pedidos por el usuario, se tenga que detener la generación porque no se pueda

realizar ningún movimiento inverso válido. Para no complicar la implementación, se considerará que si se elige aleatoriamente una ficha para hacer un movimiento inverso y ésta no tiene ningún movimiento válido, se detendrá la generación del tablero. En la parte opcional de la práctica se pide mejorar este criterio de parada.

5.1. Implementación

Deberás implementar, al menos, los siguientes subprogramas (en `soloPeg`):

- ✓ `void generarTablero(toloPeg &/*sal*/ peg, int numBolas):` Se intenta generar un tablero con el número de bolas dado. Si al elegir una ficha para generar un movimiento no tiene movimientos posibles, se parará la generación del tablero. Usará los siguientes subprogramas.
  Para poder ver si el tablero se está generando correctamente, este subprograma deberá mostrar cada uno de los pasos de la generación sin borrar la consola.
- ✓ `bool movimientoInverso(tSoloPeg &/*ent/sal*/ peg):` Intenta realizar un movimiento inverso seleccionando una ficha y eligiendo entre uno de sus posibles movimientos. Si ha podido realizar un movimiento devuelve `true`, y en caso contrario devuelve `false`.
- ✓ `bool posiblesMovsInv(tSoloPeg const& peg, tMovimiento mov, tPosibles /*sal*/ posibles):` Dada una casilla (la del movimiento) marca en `posibles` las direcciones en las que se podría mover al revés. Si ninguna es posible devuelve `false`, y en otro caso `true`.
- ✓ `void ejecutarMovInv(tSoloPeg &/*ent/sal*/ peg, tMovimiento const& mov):` Dado un movimiento, se realiza en el juego el movimiento inverso.

6. Partes opcionales

6.1. Nuevo criterio de parada en la generación aleatoria de tableros

Modifica la práctica para que la generación aleatoria de tableros sólo termine cuando se llegue al número de bolas indicado por el usuario, o cuando no haya ninguna ficha que tenga movimientos inversos posibles.

6.2. Deshacer movimientos

Una funcionalidad útil en el juego sería poder deshacer los N últimos movimientos realizados. Para ello, en `tSoloPeg` se deberán guardar los N últimos movimientos realizados en una lista de tamaño variable.

Entrega de la práctica

La práctica se entregará a través del Campus Virtual, en la tarea **Entrega de la Práctica 3** que permitirá subir el archivo Practica3.zip con los tres archivos de la práctica: `mainP3.cpp`, `soloPeg.cpp` y `soloPeg.h`. Asegúrate de poner el nombre de los miembros del grupo en un comentario al principio de cada archivo. Fin del plazo de entrega: **28 de marzo a las 23:55.**