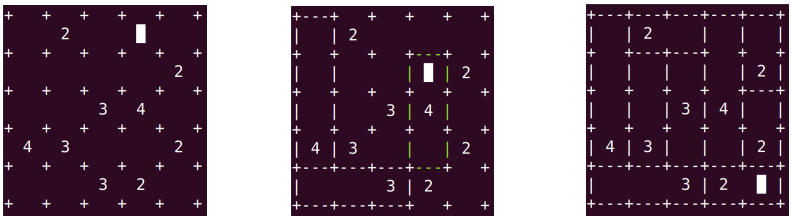
Fundamentos de la programación II

Práctica 1. Shikaku

Indicaciones generales:

- La línea 1 del programa y siguientes deben contener los nombres de los alumnos de la forma:
  `// Nombre Apellido1 Apellido2`
- **Lee atentamente** el enunciado e implementa el programa tal como se pide, con los métodos, parámetros y requisitos que se especifican. No puede modificarse la representación propuesta ni alterar los parámetros de los métodos pedidos, a excepción del modo de paso de dichos parámetros (out, ref,...), que debe determinar el alumno. Pueden implementarse todos los métodos adicionales que se consideren oportunos, especificando claramente su cometido, parámetros, etc.
- El programa, además de correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.
- La entrega se realizará a través del campus virtual, subiendo únicamente el archivo `Program.cs`, con el programa completo.
- El **plazo de entrega** finaliza el 6 de marzo.

Vamos a desarrollar el *Shikaku*[1], un puzle lógico que puede jugarse online en `https://es.puzzle-shikaku.com/`. Se desarrolla sobre una cuadrícula rectangular que contiene números en algunas de sus casillas, que llamaremos *pilares*. El objetivo es dividir la cuadrícula en rectángulos, cada uno exactamente con un pilar, y cuya área coincida con el valor de dicho pilar. A continuación se muestra un ejemplo con tres posibles estados del juego:



Inicialmente no hay ningún rectángulo marcado y el cursor está en la posición *(0,3)*; en el tablero intermedio hay dos rectángulos marcados (lados blancos) y otro que se está marcando (en verde), con el cursor en *(1,3)*. El tablero final muestra el puzzle resuelto.

La posición del cursor se mueve con las flechas de cursor. Para definir un nuevo rectángulo se sitúa el cursor en una cualquiera de sus esquinas y se pulsa *espacio*; después se desplaza el cursor a la esquina opuesta y vuelve a pulsarse *espacio*. Para facilitar la tarea, el rectángulo en curso se dibujará en verde desde que se marca la primera esquina hasta que se fija la segunda.

A lo largo del juego *siempre estará garantizado que los rectángulos marcados no se solapan* (invariante de la representación). Para ello, definimos los siguientes comportamientos:

- Si se intenta marcar la primera esquina de un nuevo rectángulo sobre otro ya existente, se elimina automáticamente este segundo (no se empieza a marcar uno nuevo). Así pues, se puede borrar un rectángulo pulsando espacio sobre una cualquiera de sus casillas.
- Cuando se está marcando un nuevo rectángulo y la segunda esquina se marca dentro de un rectángulo ya existente, se preserva el rectángulo existente y deja de marcarse el nuevo, que no se guarda de ningún modo.

Para representar el juego definimos los siguientes tipos:

```
struct Coor { // coordenadas en el tablero
    public int x,y; }

struct Pilar { // pilar en el tablero
    public Coor coor; // posición en el tablero
    public int val; } // valor

struct Rect {  // rectangulo determinado por dos esquinas
    public Coor lt, rb; }  // left-top, right-bottom
```

[1] Nikoli, 2011

```
struct Tablero { // tamaño, pilares, rectángulos marcados
    public int fils, cols; // num fils y cols del tablero
    public Pilar [] pils;  // array de pilares
    public Rect [] rects;  // array de rectángulos
    public int numRects; } // num de rectángulos definidos = prim pos libre en rect
```

El tipo `Coor` se utiliza para definir coordenadas en el tablero. `Pilar` almacenará la información de los pilares: posición y valor de cada número en el tablero. `Rect` contendrá los rectángulos marcados por el usuario, *definidos por su esquina superior izquierda (*`lt`*) e inferior derecha (*`rb`*)* (invariante de la representación). Por último, `Tablero` contiene el número de filas y columnas del tablero, el array de pilares y el array de rectángulos definidos por el usuario, así como un contador para estos últimos. Una vez cargado un tablero, durante el juego no se cambiará su número de filas ni columnas, ni los pilares. Solo cambiará la información referente a los rectángulos que va definiendo el jugador. Nótese que esta representación no utiliza una matriz bidimensional, como podría sugerir del aspecto visual del juego, pero será muy apropiada para implementar las operaciones necesarias.

El **estado del juego** incluye el tablero y dos coordenadas (`Coor`):

- `act`: posición actual del cursor.
- `ori`: posición de la *esquina origen* del posible nuevo rectángulo que se esté marcando. Esta variable además codifica el estado de la selección: cuando se marca la esquina inicial de un nuevo rectángulo se guarda su posición en esta variable. Pero cuando no hay selección en curso se asigna `ori.x = -1` (es una fila inexistente en el tablero, que denota precisamente que no hay ninguna esquina seleccionada).

Durante el desarrollo será útil visualizar información adicional sobre el estado del juego. Para ello definimos la constante global `const bool DEBUG = true` que hará que el renderizado muestre los rectángulos definidos, las esquinas (*ori*, *act*) del que se esté definiendo, así como otra información adicional que resulte útil. Para el estado intermedio del ejemplo anterior tendremos:

```
Rects: (0,0)-(3,0)  (4,0)-(4,2)
Ori: (3,3)     Act: (1,3)
```

Una vez finalizado y depurado el código, esta información se ocultará poniendo `DEBUG = false`.

Para desarrollar el programa implementaremos los siguientes métodos:

- `void LeeNivel(string file, Tablero tab)`: lee un nivel del archivo `file` y crea el tablero `tab` correspondiente. Se proporcionan más de 400 archivos de nivel tomados de `https://github.com/catlzy/shikaku-solver`, donde puede verse el formato utilizado: las dos primeras líneas son el número de filas y de columnas, y a continuación vienen las filas (una por línea) con los huecos y los pilares. Será útil el método `Split()` para trocear strings de acuerdo a un separador definido (blanco, en nuestro caso).

  El array de pilares debe tener exactamente una componente por pilar y el de rectángulos se *dimensionará* con *fils*×*cols* componentes.

- `Rect NormalizaRect(Coor c1, Coor c2)`: toma las coordenadas `c1` y `c2` como *esquinas opuestas* de un rectángulo, calcula las esquinas normalizadas `lt` y `rb` del mismo, y devuelve el rectángulo normalizado en nuestra representación. Por ejemplo, si `c1=(2,1)` y `c2=(0,1)` tendremos `lt=(0,1)` y `rb=(2,1)` (rectángulo de tamaño 3)[2].

  Cuando se está marcando un nuevo rectángulo, las posiciones `ori` y `act` no tienen ningún orden particular y este método de normalización será útil para mantener una representación unívoca de cada rectángulo.

- `void Render(Tablero tab, Coor act, Coor ori)`: dibuja el tablero en varias fases sucesivas:
  - Tablero vacío, solo con `'+'` en las intersecciones (ver los ejemplos iniciales).
  - Pilares, poniendo el número en el centro de la casilla correspondiente.
  - Rectángulos ya marcados.
  - Rectángulo en curso, en verde.
  - Información de depuración si `DEBUG` es `true`.
  - Cursor en su posición `act`.

[2]Nuestros rectángulos se definen en el plano $\mathbb{N} \times \mathbb{N}$ y dos posiciones siempre definen un rectángulo no nulo. Incluso si esas posiciones son la misma, definen un rectángulo de área 1.

Para dibujar los rectángulos utilizaremos un método auxiliar:

  - `void RenderRect(Rect r)`: dibuja el rectángulo `r` en pantalla escribiendo las conexiones horizontales "`---`" y verticales "`|`" en las posiciones correspondientes (véanse los dibujos de los ejemplos).

- `void ProcesaInput(char ch, Tablero tab, Coor act, Coor ori)`: procesa el input de usuario, que viene codificado en `ch`:

  - Con `'l','r','u','d'` cambia la posición del cursor `act` en la dirección correspondiente. Si está al borde de una dirección y se intenta mover en dicha dirección, no se modifica.

El método `LeeInput` se proporciona implementado en la plantilla. Transforma las teclas de cursor el caracteres de dirección (`'l','r','u','d'`) y la barra espaciadora en `'c'`. En este punto se puede implementar una primera versión del método `Main`. Leerá un archivo de nivel e implementa el bucle principal, que hará **en este orden**: lee el input del jugador (con `LeeInput`), modifica el cursor `act` y renderiza el nuevo estado.

A continuación implementaremos la gestión rectángulos con varios métodos:

- `bool Dentro(Coor c, Rect r)`: determina si la coordenada `c` está dentro del rectángulo `r`.
- `bool InterSect(Rect r1, Rect r2)`: determina si los rectángulos dados tienen intersección común.
- `void InsertaRect(Tablero tab, Coor c1, Coor c2)`: normaliza el rectángulo correspondiente a las coordenadas `c1` y `c2` y lo añade al tablero `tab`, siempre que dicho rectángulo no solape con ninguno de los existentes; en caso contrario no hace nada.
- `bool EliminaRect(Tablero tab, Coor c)`: busca en el tablero un rectángulo que contenga la coordenada `c`. Si existe lo elimina (será único de acuerdo a la representación) y devuelve `true`; en caso contrario devuelve `false`.

Una vez implementados estos métodos extenderemos la funcionalidad de `ProcesaInput` para poder crear y borrar rectángulos sobre la cuadrícula. El método `LeeInput` devuelve `'c'` cuando se pulsa la barra espacioadora. Ahorra `ProcesaInput`:

- Con `'c'` (que) comienza o termina la selección de un nuevo rectángulo. En este punto es donde hay que implementar los comportamientos del marcado explicados al principio, utilizando los métodos anteriores de inserción y eliminación de rectángulos.

Por último, implementaremos los métodos:

- `int AreaRect(Rect r)`: calcula el área del rectángulo `r`.
- `bool CheckRect(Rect r, Pilar [] p)`: busca un pilar del array de pilares `p` que quede dentro del rectángulo `r` y comprueba si el área de dicho rectángulo coincide con el valor del pilar. Razonar el comportamiento en el caso de que haya más de un pilar dentro del rectángulo.
- `bool FinJuego(Tablero tab)`: comprueba si el juego está resuelto con ayuda del método anterior.
- `bool JuegaNivel(String file)`: lee el nivel del archivo `file`, inicializa el estado del juego e implementa el bucle principal para jugar ese nivel. Devuelve `true` si se ha superado el nivel, `false` si se ha abortado el juego.
- `void Main()`: integra los métodos anteriores para ir avanzando los niveles a medida que el usuario los va superando. Implementar un menú de juego para interactuar con el jugador y gestionar el siguiente nivel a cargar.

Hasta aquí quedaría cubierta la funcionalidad básica que tiene que implementar la práctica necesariamente.

Extensiones del juego

Pueden implementarse de manera opcional algunas (o todas) las extensiones que se proponen a continuación:

- Mejorar la presentación en pantalla coloreando el interior de los rectángulos. Tiene que garantizarse que dos rectángulos con frontera común (segmento de longitud positiva) tengan distinto color. Debe conseguirse también que los pilares queden bien visibles. Curiosidad: ¿qué número mínimo de colores son necesarios para conseguir este coloreado?
- Utilizar archivos para gestionar un registro de jugadores y niveles superados por cada uno de ellos. Puede guardarse además una puntuación o registro de records para cada nivel, que puede ser por ejemplo, el tiempo requerido para completar el nivel.
- Implementar un método para generar tableros aleatorios de juego *que tengan solución*. Idea: ir generando rectángulos aleatorios que vayan cubriendo un tablero inicialmente vacío. ¿Podría garantizarse de algún modo la unicidad de la solución así generada? ¿Podría gradarse la dificultad del nivel de algún modo razonable y sencillo?
- (Dificil) Implementar un resolutor de Shikakus.