

# Trabajo Práctico - Programación Imperativa

## Buscaminas

Introducción a la Programación - Primer cuatrimestre de 2025

Fecha límite de entrega: Viernes 06 de Junio - 23:59 hs

### Introducción

El objetivo de este Trabajo Práctico es aplicar los conceptos de programación imperativa vistos en la materia para resolver ejercicios utilizando el lenguaje de programación Python.

El trabajo consiste en implementar el famoso juego **Buscaminas**.

**Buscaminas** es un juego de lógica que se popularizó con los sistemas operativos Microsoft Windows. El juego consiste en un tablero o campo minado, formado por una cuadrícula de celdas, en la que se colocan minas en ubicaciones o coordenadas aleatorias.

Inicialmente, todas las celdas están cerradas, lo que significa que el jugador no sabe si una celda contiene una mina o no. El jugador abre una celda haciendo clic sobre ella: si la celda abierta contiene una mina, el jugador pierde. Si en cambio la celda no contiene una mina, se mostrará un número que indica cuántas ubicaciones vecinas (adyacentes en forma horizontal, vertical o diagonal) contienen minas. Además, si el número de celdas vecinas con minas resulta ser cero, se abrirán automáticamente todas las celdas vecinas, generando un efecto en cascada. La información sobre el número de minas en las celdas vecinas permite al jugador decidir, mediante razonamiento lógico, cuáles celdas contienen o no contienen minas. El jugador gana cuando se han abierto todas las celdas que no contienen minas, es decir, las únicas celdas cerradas restantes son aquellas que contienen minas.

Además de abrir celdas, el jugador puede interactuar con el juego bloqueando o desbloqueando celdas cerradas; una celda bloqueada es una celda que el jugador sospecha que contiene una mina; cuando está bloqueada, la celda no se puede abrir, lo que evita que se abra accidentalmente.

Notar que una celda tiene como adyacentes a todas aquellas que son vecinas. Por ejemplo, a continuación se resalta en verde las celdas adyacentes a la celda amarilla. Notar que figuran las coordenadas de cada celda (primer componente indica la fila, segundo componente indica la columna).

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

En el campus de la teórica pueden encontrar un proyecto `buscaminas.zip` el cual contiene los siguientes archivos:

- **interfaz\_buscaminas.py**: este archivo no deben modificarlo. Contiene la lógica para manejar la interfaz gráfica del juego, llamando a las funciones que ustedes deben implementar.
- **buscaminas.py**: contendrá la lógica del juego. Las funciones a implementar ya están en el archivo, sin funcionalidad. No deben cambiar los nombres de estas funciones, pero si pueden (y deben) agregar las funciones auxiliares que consideren necesarias para implementar la funcionalidad de forma modular.
- **tests\_materia.py**: un ejemplo de caso de test (el mismo que figura en el enunciado) para la mayoría de los ejercicios.

## Pautas de Entrega

Para la entrega del trabajo práctico se deben tener en cuenta las siguientes consideraciones:

- El trabajo se debe realizar en grupos de tres estudiantes *de forma obligatoria*.  
No se aceptarán trabajos de menos ni más integrantes.  
Ver aviso en el campus de la materia donde indica el link para registrar los grupos.
- Se debe implementar un conjunto de casos de test para todos los ejercicios (no es obligatorio para las funciones auxiliares que definan ustedes).
- Los programas deben pasar con éxito los casos de test entregados por la materia (en el archivo `tests_materia.py`), sus propios tests, y un conjunto de casos de test “secretos”.
- El archivo con el código fuente debe tener nombre `buscaminas.py`.
- No está permitido alterar los nombres de las funciones a implementar ni los tipos de datos. Deben mantenerse tal cual como descargan.
- Pueden definir todas las funciones auxiliares que requieran.
- Deben tipar las variables y funciones que definan.
- No está permitido utilizar técnicas no vistas en clase para resolver los ejercicios. En el campus, pueden encontrar un listado de todas las funciones válidas a usar. Si usan funciones fuera de esa lista en algún ejercicio, se desaprobará el TP y deberán re-entregar automáticamente.
- Recordar que no se acepta un número de integrantes diferente a tres, pero aceptamos que queden menos en caso de que alguien abandone la materia. Esto debe avisarse con anticipación y dejarlo por escrito como comentario en el archivo de `buscaminas.py`. Las personas que abandonen no pueden entregar de forma individual.
- El TP no tiene nota, sino Aprobado/Desaprobado.
- Todos los integrantes del grupo deben poder responder preguntas tanto de la especificación de cualquier ejercicio, como a la forma puntual de cómo lo implementaron y los test que definieron.

Se evaluarán las siguientes características:

- **Correctitud:** todos los ejercicios deben estar bien resueltos, es decir, deben respetar su especificación.
- **Declaratividad:** los nombres de las funciones que se definan, así como los nombres de las variables, deben ser apropiados al problema que se resuelve. Los nombre de variables declarativos ayudan a la legibilidad del código.
- **Modularización:** evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (por el enunciado o por ustedes mismos). Pueden agregar más funciones para separar el código en tareas específicas y facilitar su lectura.
- **Documentación:** Las funciones deben tener su documentación (Python docstring<sup>1</sup>) que describa **qué** hace y **cómo** se usa. En caso de sentencias complejas, analizar si es conveniente además realizar comentarios para otros programadores en su código (usando `#`).
- **Testing:** Todos los ejercicios del enunciado deben tener sus propios casos de test que pasen correctamente. Un ejercicio sin casos de test se considera *no resuelto*. Los casos de test deben cubrir distintos escenarios de uso de las funciones y **deben tener una cobertura de al menos 95 % de líneas de código, y al menos 95 % de cobertura de ramas**. Esto se verá en la teórica de Testing de caja Blanca y en su clase de laboratorio asociado, utilizando las herramientas `unittest` y `coverage`.

<sup>1</sup>Ver lineamientos de Python: <https://peps.python.org/pep-0287/>. También pueden referir al estilo Google: <https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings>

## Método de entrega

Se espera que el trabajo grupal lo realicen de forma incremental utilizando un sistema de control de versiones, como Git. Para la entrega, sólo vamos a trabajar con GitLab (pueden ser `git.exactas.uba.ar` si tienen usuario, o deben crearse un usuario en `gitlab.com`). El repositorio debe ser privado, deben estar todos los integrantes del grupo y, además, deben agregar con rol “Developer” un usuario de los docentes:

- Si usan `git.exactas.uba.ar`: Deben agregar al usuario *docentes.ip*
- Si usan `gitlab.com`: Deben agregar al usuario *ip.dc.uba*

La entrega debe ser realizada *únicamente* por un integrante del grupo. Debe realizarlo quien haya anotado al grupo en el link para registrar los grupos. La entrega consta de la URL del repositorio y un commit particular. Los docentes descargarán el código para ese commit. En caso que el link o el commit no sean válidos, o no se pueda descargar porque no agregaron el integrante docente con el rol apropiado, el trabajo estará desaprobado, pero tendrán la oportunidad de re-entregar en fecha de recuperatorio.

Antes de enviar el trabajo, se recomienda fuertemente:

- Clonar el repositorio con el link y commit proporcionados. Chequear que ambos son válidos.
- Ejecutar los casos de test de la materia y sus propios casos de test, sobre el repositorio recién clonado.
- Todos los casos de test deben pasar satisfactoriamente, y el archivo con la implementación debe poder cargarse sin generar errores.

## Tipos de datos

Para especificar los problemas a resolver, usaremos:

- `EstadoJuego`: como un renombre del tipo `dict[str, Any]`.

`estado: EstadoJuego` es un diccionario que guarda diferentes valores asociados al estado actual del Juego. Contiene 6 claves, cuyos valores se resumen a continuación:

- `estado['filas'] : Z`: representa la cantidad de filas del tablero.
- `estado['columns'] : Z`: representa la cantidad de columnas del tablero.
- `estado['minas'] : Z`: representa la cantidad de minas del tablero.
- `estado['juego_terminado'] : Bool`: representa si el juego ya finalizó (ya sea porque ganó o perdió el usuario).
- `estado['tablero'] : seq(seq(Z))`: representa el tablero (matriz de enteros) en el cual puede haber -1 para representar si hay mina, o un número que representa la cantidad de minas adyacentes en cada celda. El mismo no es visible al usuario.
- `estado['tablero_visible'] : seq(seq(String))`: representa una matriz de String con el valor que ve el usuario para cada celda.

Además, se usarán las siguientes constantes para `estado['tablero_visible']`, sobre las que sugerimos algunos valores:

- `BOMBA:str = '💣' # chr(128163)`
- `BANDERA:str = '🚩' # chr(127987) o chr(128681)`
- `VACIO:str = ' ' # espacio, es decir, len(VACIO) == 1`

Pueden cambiar el valor de la constante, pero no el nombre de la variable. Por ejemplo, pueden elegir si usar banderas blancas `chr(127987)` o rojas `chr(128681)`, pero no pueden cambiar el nombre `BANDERA`. Prueben cambiar el `VACIO = chr(10054)`.

## Uso de Random

Se puede usar la librería **random**. Por esto, encontrarán que se importa (`import random`) la misma en el template que descargan.

Algunos usos posibles de la librería son los siguientes:

- `n:int = random.randint(a,b)`: genera un número aleatorio  $n$  entre  $a$  (inclusive) y  $b$  (inclusive)
- `e = random.choice(s)`: dada una secuencia  $s$ , devuelve un elemento  $e$  aleatorio de  $s$ .
- `seleccion:list = random.sample(s, n)`: devuelve una lista con  $n$  elementos distintos al azar de una secuencia  $s$ . Por ejemplo dados  $s = \text{range}(1, 100)$  y  $n = 3$ , un resultado posible es `seleccion = [88, 54, 38]`

## Especificación

A continuación se especifican todos los ejercicios que se deben programar en Python.

**Aclaración:** Se debe implementar de forma obligatoria el primer problema de cada ejercicio, el cual se encuentra en el template a descargar. En los ejercicios que hay definidos más de un problema, se utilizan de forma auxiliar para reutilizar algunas especificaciones. Deben considerar si es necesario implementar estos problemas o no.

**Aclaración:** Por simplificación, cuando decimos “posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z} \dots$ ” estamos diciendo que dada una matriz  $m$  se cumple:

$$0 \leq i \wedge i < |m| \wedge 0 \leq j \wedge j < |m[0]|$$

Por lo tanto,  $m[i][j]$  no se indefine.

### Ejercicio 1

```
problema colocar_minas (in filas:  $\mathbb{Z}$ , columnas:  $\mathbb{Z}$ , minas:  $\mathbb{Z}$ ) :  $seq(seq(\mathbb{Z}))$  {
  requiere: {  $filas > 0 \wedge columnas > 0 \wedge 0 < minas \wedge minas < columnas * filas$  }
  asegura: {  $|res| = filas \wedge |res[0]| = columnas$  }
  asegura: {  $es\_matriz(res) = True$  }
  asegura: { La cantidad de posiciones válidas  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que  $res[i][j] = -1$  es igual a  $minas$  }
  asegura: { La cantidad de posiciones válidas  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que  $res[i][j] = 0$  es igual a  $filas * columnas - minas$  }
  asegura: { Las posiciones  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que  $res[i][j] = -1$  son seleccionadas con probabilidad uniforme utilizando la librería random }
}

problema es_matriz (in t:  $seq(seq(T))$ ) : Bool {
  requiere: { True }
  asegura: {  $|t| > 0 \wedge |t[0]| > 0$  }
  asegura: { Todos los elementos de  $t$  tienen tamaño  $|t[0]|$  }
}
```

Ejemplo *colocar\_minas*:

```
entrada: filas = 2, columnas = 2, minas = 1
salida: res = [[-1,0],
               [ 0,0]]
```

### Ejercicio 2

```
problema calcular_numeros (inout tablero:  $seq(seq(\mathbb{Z}))$ ) {
  requiere: {  $es\_matriz(tablero) = True$  }
  requiere: { Todas las celdas de tablero son -1 ó 0 }
  requiere: { Hay al menos una celda en tablero con valor -1, y al menos una celda con valor 0 }
  modifica: { tablero }
  asegura: { Para toda posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que  $tablero@pre[i][j] \neq -1$ , se cumple que:
               $tablero[i][j]$  es igual a la cantidad de celdas con valor -1 en  $tablero@pre$ 
              que son adyacentes a la celda  $tablero@pre[i][j]$  }
  asegura: { Para toda posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que  $tablero@pre[i][j] = -1$ , se cumple que:
               $tablero[i][j] = tablero@pre[i][j]$  }
  asegura: { tablero tiene las mismas dimensiones que tablero@pre }
}
```

Ejemplo *calcular\_numeros*:

```

entrada: tablero = [[0,-1],
                    [0, 0]]
salida: tablero = [[1,-1],
                  [1, 1]]

```

### Ejercicio 3

```

problema crear_juego (in filas:  $\mathbb{Z}$ , in columnas:  $\mathbb{Z}$ , in minas:  $\mathbb{Z}$ ) : EstadoJuego {
  requiere: {  $filas > 0 \wedge columnas > 0 \wedge minas > 0 \wedge minas < filas * columnas$  }
  asegura: {  $res['filas'] = filas$  }
  asegura: {  $res['columnas'] = columnas$  }
  asegura: {  $res['minas'] = minas$  }
  asegura: { Todas las celdas de  $res['tablero_visible']$  tienen valor igual a VACIO. }
  asegura: {  $res['juego_terminado'] = False$  }
  asegura: {  $estado\_valido(res) = True$  }
}

problema estado_valido (in estado: EstadoJuego) : Bool {
  requiere: { True }
  asegura: {  $estructura\_y\_tipos\_validos(estado)$  }
  asegura: { La cantidad de posiciones válidas  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que  $estado['tablero'][i][j] = -1$  es igual a  $estado['minas']$  }
  asegura: { Dado un  $t \in seq(seq(\mathbb{Z}))$  se cumple que  $estado['tablero'] = calcular\_numeros(t)$  }
  asegura: {  $estado['juego_terminado'] = True \iff$ 
    (  $todas\_celdas\_seguras\_descubiertas(estado['tablero'], estado['tablero\_visible']) = True$  ó
    existe una posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  tal que se cumple que:
       $estado['tablero\_visible'][i][j] = BOMBA$  ) }
  asegura: { Para cualquier posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ ,
    si  $estado['tablero\_visible'][i][j] = BOMBA$  entonces se cumple que  $estado['tablero'][i][j] = -1$  }
  asegura: { Para cualquier posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ ,
    si  $estado['tablero\_visible'][i][j]$  no es BOMBA, VACIO ni BANDERA, entonces se cumple que:
       $estado['tablero\_visible'][i][j] = str(estado['tablero'][i][j])$  }
}

problema estructura_y_tipos_validos (in estado: EstadoJuego) : Bool {
  requiere: { True }
  asegura: {  $res = True \iff$ 
    (  $estado['filas'] \in \mathbb{Z} \wedge estado['filas'] > 0 \wedge$ 
       $estado['columnas'] \in \mathbb{Z} \wedge estado['columnas'] > 0 \wedge$ 
       $estado['minas'] \in \mathbb{Z} \wedge$ 
       $estado['minas'] > 0 \wedge estado['minas'] < (estado['columnas'] * estado['filas']) \wedge$ 
       $estado['juego_terminado'] \in Bool \wedge$ 
       $estado['tablero'] \in seq(seq(\mathbb{Z})) \wedge$ 
      Los valores de las celdas de  $estado['tablero']$  deben ser valores
        entre  $-1$  y  $8$  (ambos inclusive)  $\wedge$ 
       $estado['tablero\_visible'] \in seq(seq(String)) \wedge$ 
      Los valores de las celdas de  $estado['tablero\_visible']$  deben ser VACIO,
        BOMBA, BANDERA, o un número entre  $0$  y  $8$  (ambos inclusive)  $\wedge$ 
      La cantidad de claves de  $estado$  es igual a  $6 \wedge$ 
       $son\_matriz\_y\_misma\_dimension(estado['tablero'], estado['tablero\_visible']) \wedge$ 
       $|estado['tablero']| = estado['filas'] \wedge$ 
       $|estado['tablero'][0]| = estado['columnas']$  ) }
}

problema son_matriz_y_misma_dimension (in t1, t2 :  $seq(seq(T))$ ) : Bool {
  requiere: { True }
  asegura: {  $res = True \iff (es\_matriz(t1) \wedge es\_matriz(t2) \wedge |t1| = |t2| \wedge |t1[0]| = |t2[0]|)$  }
}

```

```

problema todas_celdas_seguras_descubiertas (in tablero: seq<seq<Z>>, tablero_visible: seq<seq<String>>) : Bool {
  requiere: {son_matriz_y_misma_dimension(tablero, tablero_visible)}
  asegura: {res = True  $\iff$  para toda posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  se cumple que:
    (tablero[i][j] = -1  $\wedge$  (tablero_visible[i][j] = VACIO  $\vee$  tablero_visible[i][j] = BANDERA)  $\vee$ 
    (tablero[i][j]  $\neq$  -1  $\wedge$  tablero_visible[i][j] = str(tablero[i][j])) }
}

```

Ejemplo crear\_juego:

```

entrada: filas = 2, columnas = 2, minas 1
res: {'filas' = 2, 'columnas' = 2, 'minas' = 1,
      'tablero' = [[-1,1],
                   [ 1,1]],
      'tablero_visible' = [[' ', ' '],
                           [ ' ', ' ']],
      'juego_terminado' = False}

```

#### Ejercicio 4

```

problema obtener_estado_tablero_visible (in estado: EstadoJuego) : seq<seq<String>> {
  requiere: {estado_valido(estado)}
  asegura: {res = una copia de estado['tablero_visible']}
}

```

Ejemplo obtener\_estado\_tablero\_visible:

```

entrada: estado = {'filas' = 2, 'columnas' = 2, 'minas' = 1,
                  'tablero' = [[-1,1],
                               [ 1,1]],
                  'tablero_visible' = [[' ', '1'],
                                       [ ' ', ' ']],
                  'juego_terminado' = False}
salida: res = [[' ', '1'],
               [ ' ', ' ']]

```

#### Ejercicio 5

```

problema marcar_celda (inout estado: EstadoJuego, in fila: Z, columna: Z) {
  requiere: {estado_valido(estado)}
  requiere: {(fila, columna) es una posición válida en estado['tablero']}
  modifica: {estado}
  asegura: {Si (estado@pre['juego_terminado'] = True  $\vee$ 
    (estado@pre['tablero_visible'][fila][columna]  $\neq$  VACIO  $\wedge$ 
    estado@pre['tablero_visible'][fila][columna]  $\neq$  BANDERA)), entonces:
    estado['tablero_visible'] = estado@pre['tablero_visible']}
  asegura: {Si estado@pre['tablero_visible'][fila][columna] = VACIO, entonces:
    estado['tablero_visible'][fila][columna] = BANDERA  $\wedge$ 
    para toda posición  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  válida tal que  $i \neq fila \wedge j \neq columna$ ,
    se cumple que estado['tablero_visible'][i][j] = estado@pre['tablero_visible'][i][j]}
  asegura: {Si estado@pre['tablero_visible'][fila][columna] = BANDERA, entonces:
    estado['tablero_visible'][fila][columna] = VACIO  $\wedge$ 
    para toda posición  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  válida tal que  $i \neq fila \wedge j \neq columna$ ,
    se cumple que estado['tablero_visible'][i][j] = estado@pre['tablero_visible'][i][j]}
  asegura: {estado['filas'] = estado@pre['filas']}
  asegura: {estado['columnas'] = estado@pre['columnas']}
  asegura: {estado['minas'] = estado@pre['minas']}
}

```

```

asegura: {estado['juego_terminado'] = estado@pre['juego_terminado']}
asegura: {estado['tablero'] = estado@pre['tablero']}
asegura: {estado_valido(estado)}
}

```

Ejemplo marcar\_celda:

```

entrada: estado = {'filas' = 2, 'columnas' = 2, 'minas' = 1,
                  'tablero' = [[-1,1],
                               [ 1,1]],
                  'tablero_visible' = [[' ',' '],
                                       [' ',' ']],
                  'juego_terminado' = False},
      fila = 0,
      columna = 0
salida: estado = {'filas' = 2, 'columnas' = 2, 'minas' = 1,
                 'tablero' = [[-1,1],
                              [ 1,1]],
                 'tablero_visible' = [['🚩',' '],
                                       [' ',' ']],
                 'juego_terminado' = False}

```

## Ejercicio 6

```

problema descubrir_celda (inout estado: EstadoJuego, in fila:  $\mathbb{Z}$ , columna:  $\mathbb{Z}$ ) {
  requiere: {estado_valido(estado)}
  requiere: {(fila, columna) es una posición válida en estado['tablero']}
  requiere: {estado['tablero_visible'][fila][columna] = VACIO}
  modifica: {estado}
  asegura: {Si estado@pre['juego_terminado'] = True, entonces:
            estado['tablero_visible'] = estado@pre['tablero_visible']}
  asegura: {Si estado@pre['tablero'][fila][columna] = -1, entonces:
            estado['juego_terminado'] = True  $\wedge$ 
            para toda posición  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  válida se cumple que:
              (estado['tablero_visible'][i][j] = BOMBA  $\iff$  estado@pre['tablero'][i][j] = -1)  $\wedge$ 
              estado['tablero_visible'][i][j] = estado@pre['tablero_visible'][i][j]  $\iff$  estado@pre['tablero'][i][j]  $\neq$  -1
            }
  asegura: {Si estado@pre['tablero'][fila][columna]  $\neq$  -1, entonces:
            (estado['juego_terminado'] = True  $\iff$ 
             todas_celdas_seguras_descubiertas(estado['tablero'], estado['tablero_visible']) = True)}
  asegura: {Si estado@pre['tablero'][fila][columna]  $\neq$  -1, entonces:
            para toda secuencia  $s$  en
            caminos_descubiertos(estado@pre['tablero'], estado@pre['tablero_visible'], fila, columna) se cumple que:
              para todo  $0 \leq x < |s|$  vale que:
                estado['tablero_visible'][s[x]0][s[x]1]  $\neq$  BANDERA y
                estado['tablero_visible'][s[x]0][s[x]1] = str(estado@pre['tablero'][s[x]0][s[x]1]) y
              para toda posición válida  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  que no pertenece a ninguna secuencia  $s$ , se cumple que:
                estado['tablero_visible'][i][j] = estado@pre['tablero_visible'][i][j]
            }
  asegura: {estado['filas'] = estado@pre['filas']}
  asegura: {estado['columnas'] = estado@pre['columnas']}
  asegura: {estado['minas'] = estado@pre['minas']}
  asegura: {estado['tablero'] = estado@pre['tablero']}
  asegura: {estado_valido(estado)}
}

```

```

problema caminos_descubiertos (in tablero: seq(seq( $\mathbb{Z}$ )), in tablero_visible: seq(seq(String)), in f, c:  $\mathbb{Z}$ ) : seq(seq( $\mathbb{Z} \times \mathbb{Z}$ ))
{

```



```

requiere: {True}
asegura: {res contiene todos los caminos posibles contenidos en tablero, es decir, secuencias de posiciones válidas
( $i, j$ )  $\in \mathbb{Z} \times \mathbb{Z}$ , sin repetidos, tal que cada camino  $s$  perteneciente a res cumple:
    La secuencia  $s$  tiene al menos un elemento  $\wedge$ 
    El primer elemento de  $s$  es  $(f, c) \wedge$ 
    Si  $\text{tablero}[f][c] > 0$ , entonces  $|s| = 1 \wedge$ 
    Si  $\text{tablero}[f][c] = 0$ , entonces:
        Si  $|s| > 1$ , entonces todos los elementos de  $s$  salvo el último, tienen que ser posiciones en tablero
        con valor igual a 0 (es decir, para todo  $0 < x < |s| - 1$  vale que  $\text{tablero}[s[x]_0][s[x]_1] = 0$ )  $\wedge$ 
        No hay elemento  $(i, j) \in \mathbb{Z} \times \mathbb{Z}$  en la secuencia  $s$  tal que  $(i, j)$  es posición válida en
        tablero y  $\text{tablero.visible}[i][j] = \text{BANDERA}$ 
    Los elementos contiguos de  $s$  son posiciones válidas y adyacentes en tablero
}
}

```

Ejemplo *descubrir\_celda*:

```

entrada: estado = {'filas' = 3, 'columnas' = 3, 'minas' = 3,
                  'tablero' = [[ 2,-1, 1],
                               [-1, 3, 1],
                               [-1, 2, 0]],
                  'tablero_visible' = [[' ', ' ', ' '],
                                       [' ', ' ', ' '],
                                       [' ', ' ', ' ']],
                  'juego_terminado' = False},
fila = 2,
columna = 2
salida: estado = {'filas' = 3, 'columnas' = 3, 'minas' = 1,
                  'tablero' = [[ 2,-1, 1],
                               [-1, 3, 1],
                               [-1, 2, 0]],
                  'tablero_visible' = [[' ', ' ', ' '],
                                       [' ', '3', '1'],
                                       [' ', '2', '0']],
                  'juego_terminado' = False}

```

## Ejercicio 7

```

problema verificar_victoria (in estado: EstadoJuego) : Bool {
    requiere: {estado_valido(estado)}
    asegura: {res = True  $\iff$  todas_celdas_seguras_descubiertas(estado['tablero'], estado['tablero_visible']) = True}
}

```

Ejemplo *verificar\_victoria*:

```

entrada: estado = {'filas' = 2, 'columnas' = 2, 'minas' = 1,
                  'tablero' = [[-1,1],
                               [ 1,1]],
                  'tablero_visible' = [[' ', '1'],
                                       ['1', '1']],
                  'juego_terminado' = False},
salida: res = True

```

## Ejercicio 8

```

problema reiniciar_juego (inout estado: EstadoJuego) {

```

```

    requiere: {estado_valido(estado)}
    modifica: {estado}
    asegura: {estado['filas'] = estado@pre['filas']}
    asegura: {estado['columnas'] = estado@pre['columnas']}
    asegura: {estado['minas'] = estado@pre['minas']}
    asegura: {estado['juego_terminado'] = False}
    asegura: {estado['tablero'] ≠ estado@pre['tablero']}
    asegura: {Todas las celdas de estado['tablero_visible'] tienen valor igual a VACIO.}
    asegura: {estado_valido(estado)}
}

```

Ejemplo reiniciar\_juego:

```

entrada: estado = 'filas' = 2, 'columnas' = 2, 'minas' = 1,
              'tablero' = [[-1,1],
                          [ 1,1]],
              'tablero_visible' = [[ ' ', '1'],
                                  [ ' ', ' ']],
              'juego_terminado' = False
salida: estado = 'filas' = 2, 'columnas' = 2, 'minas' = 1,
              'tablero' = [[ 1,1],
                          [-1,1]],
              'tablero_visible' = [[ ' ', ' '],
                                  [ ' ', ' ']],
              'juego_terminado' = False

```

### Ejercicio 9

```

problema guardar_estado (in estado: EstadoJuego, in ruta_directorio:String) {
    requiere: {ruta_directorio es una ruta válida de un directorio.}
    requiere: {estado['juego_terminado'] = False}
    requiere: {estado_valido(estado)}
    asegura: {En ruta_directorio se guardan exactamente 2 archivos:
              tablero.txt: se guarda el estado de estado['tablero'], separando por comas (',') los valores del tablero.
              tablero_visible.txt: se guarda el estado de estado['tablero_visible'], separando por comas (',') los valores
del tablero_visible, guardando '*' en lugar de BANDERA y guardando '?' en lugar de VACIO}
    asegura: {La cantidad de líneas de cada archivo guardado es igual a estado['filas'], considerando una línea cuando
la cantidad de caracteres sea mayor a 0.}
    asegura: {La cantidad de comas (',') por línea es igual a estado['columnas'] - 1; habiendo una coma entre dos
valores.}
    asegura: {Las posiciones y valores guardados en los archivos deben corresponder con las posiciones y valores de
estado['tablero_visible'] y estado['tablero']}
    asegura: {los archivos guardados no contienen espacios}
}

```

Ejemplo *guardar\_estado*:

```
entrada: estado = {'filas' = 2, 'columnas' = 2, 'minas' = 1,
                  'tablero' = [[-1,1],
                               [ 1,1]],
                  'tablero_visible' = [[['🚩', '1'],
                                         [' ', ' ']],
                  'juego_terminado' = False},
ruta_archivo= './save'
```

salida:

```
.save/tablero.txt:
-1,1
1,1
.save/tablero_visible.txt:
*,1
?,?
```

**Aclaración:** Al trabajar con paths, si `path1` y `path2` son strings, es recomendable usar en python `os.path.join(path1, path2)` en lugar de `path1+path2`.

## Ejercicio 10

problema `cargar_estado` (out `estado`: EstadoJuego, in `ruta_directorio`:String) : Bool {  
  **requiere:** {*ruta\_directorio* es una ruta existente de un directorio.}  
  **asegura:** {Si en *ruta\_directorio* no existe alguno de los archivos *tablero.txt* ó *tablero\_visible.txt*, entonces **res** = *False*}  
  **asegura:** {Si en *ruta\_directorio* existen los archivos *tablero.txt* y *tablero\_visible.txt*, pero no cumplen las siguientes condiciones, entonces **res** = *False*. Condiciones:  
    La cantidad de líneas de cada archivo guardado es igual a *estado['filas']*, considerando una línea cuando la cantidad de caracteres sea mayor a 0.  
    La cantidad de comas (',') por línea es igual a *estado['columnas']* - 1.  
    En *tablero.txt* debe haber al menos un -1, y los valores deben ser -1 ó corresponder a la cantidad de -1 que hay en las posiciones adyacentes, considerando al contenido del archivo como una matriz  
    En *tablero\_visible.txt*, solo puede haber números (entre [0; 8]), '\*' (representa *BANDERA*) y '?' (representa *VACIO*), además de comas (',') para separar valores. En caso de haber números, estos deben corresponder a los valores en la misma posición en el archivo *tablero.txt* (la *i*ésima línea corresponde a la *i*ésima fila de tablero).}  
  **asegura:** {Si en *ruta\_directorio* existen los archivos *tablero.txt* y *tablero\_visible.txt* y cumplen las condiciones anteriores, entonces:  
    **res** = *True* ∧  
    *estado['filas']* = cantidad de líneas en *tablero.txt* ∧  
    *estado['columnas']* = 1+ cantidad de comas (',') por línea en *tablero.txt* ∧  
    *estado['minas']* = cantidad de -1 en *tablero.txt* ∧  
    *estado['juego\_terminado']* = *False* ∧  
    *estado['tablero']* = es igual, posición a posición, a los valores en *tablero.txt* ∧  
    *estado['tablero\_visible']* = es igual, posición a posición, a los valores en *tablero\_visible.txt*, cambiando '\*' por *BANDERA* y '?' por *VACIO*.  
  }  
  **asegura:** {*estado\_valido*(*estado*)}  
}

**Aclaración:** En el template van a encontrar la función  
`existe_archivo(in ruta_directorio:String, nombre_archivo:String): Bool`  
que dado un string, retorna True si y solo si el archivo con nombre `nombre_archivo` existe en el directorio `ruta_directorio`. Pueden utilizarla para resolver este problema.

Ejemplo *cargar\_estado*:

```
entrada: estado = {'filas' = 50, 'columnas' = 15, 'minas' = 150,
                  'tablero' = [[1,7],
                               [8,0]],
                  'tablero_visible' = [[' ', '1'],
                                       ['1', '1']],
                  'juego_terminado' = False},
ruta_archivo= './save/'
```

Y en ruta\_archivo están los siguientes archivos:

.save/tablero.txt:


-1,1

1,1

.save/tablero\_visible:

\*,1

?,?

```
salida: res = True,
        estado = {'filas' = 2, 'columnas' = 2, 'minas' = 1,
                  'tablero' = [[-1,1],
                               [ 1,1]],
                  'tablero_visible' = [[', '1'],
                                       [' ', ' ']],
                  'juego_terminado' = False}
```