Trabajo Práctico - Programación Imperativa

Batalla naval

Introducción a la Programación - Segundo cuatrimestre de 2025

Fecha límite de entrega: Miércoles 12 de Noviembre durante la clase (se cierra el envío al terminar la clase)

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 **Batalla naval**. Para quien no lo conozcan, es un juego de mesa de dos jugadores en el que cada jugador tiene a su cargo una flota de barcos y el objetivo es hundir los barcos del oponente. En cada turno un jugador puede lanzar un torpedo a una posición del terreno enemigo y este le debe responder si el torpedo impactó alguno de sus barcos o no.

Este TP consta de 3 partes:

- Parte I: resolver los 5 problemas que se presentan al final de este documento. La fecha límite para resolverlo es el día 11/11/2025. Deben contemplar todas las condiciones detalladas en este documento.
- Parte II: el miércoles 12/11/2025 deben asistir *todos* los integrantes del grupo al taller de su turno. Durante ese taller, deberán implementar funcionalidad extra que será dada en el momento. Al finalizar el taller, deben enviar el TP completo (lo realizado en la Parte I más lo implementado durante el taller). Al terminar el taller se deshabilitará el envío del TP.
- Parte III: el lunes 17/11 durante el taller recibirán el TP de otro grupo y deberán analizar ciertos aspectos que les pediremos. Luego, en la teórica del 20/11 se debatirá sobre esto.

Reglas del juego

Cada *jugador* cuenta con un *tablero* que el otro *jugador* no puede ver. Cada *tablero* tiene dos *grillas* con las mismas *dimensiones*. Una de las *grillas* es la *grilla local* donde el *jugador* ubica sus propios *barcos*. La otra *grilla* es la *grilla del oponente* que empieza vacía y el *jugador* deberá ir actualizando con la información que obtenga en cada turno para replicar en esa *grilla* la *grilla local* del oponente. Antes de comenzar a jugar, cada *jugador* ubica sus *barcos* en su *grilla local*. Cada *barco* puede ocupar varias *posiciones* y puede haber barcos de distintos *tamaños* (el *tamaño* del barco indica la cantidad de *posiciones* de las *grillas* que ocupa). Las *posiciones* ocupadas por un *barco* deben ser todas seguidas en línea recta,

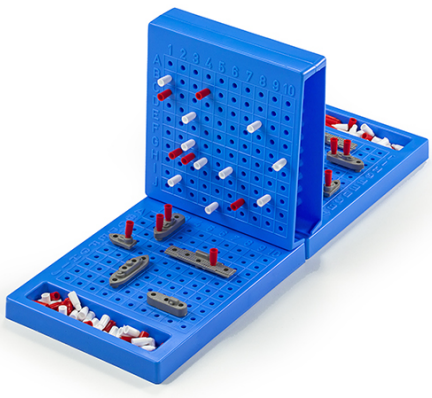

Figura 1: Imagen de ejemplo de un juego de batalla naval.

horizontal o vertical pero no diagonal. Además, los *barcos* no se pueden ubicar de forma que una *posición* ocupada por un *barco* quede adyacente a otra *posición* ocupada por otro *barco*. Una vez ubicados todos los *barcos* de ambos *jugadores*, el juego comienza. Empieza el *jugador* 1 y se van turnando. El turno de cada *jugador* consiste en realizar un ataque lanzando un torpedo. Cuando un *jugador* lanza un torpedo, lo hace indicando la *fila* (por medio de una *letra*) y la *columna* (por medio de un *número*) de la *posición* a la cual lanzó el torpedo. El otro *jugador* responde *tocado* si el torpedo impactó una *posición* ocupada por uno de sus *barcos* o *agua* si no. Tras cada ataque, el *jugador* atacado debe marcar en su *grilla local* la *posición* que fue atacada (para saber cuáles de sus *barcos* siguen a flote) y el *jugador* que atacó debe marcar en su *grilla del oponente* la misma *posición* con algún indicador del resultado del ataque (que le permita saber cuáles *posiciones* de esa *grilla* ya atacó y en cuáles de ellas descubrió algún *barco*). Si el torpedo impactó un *barco* y además con ese impacto ya fueron atacadas todas las *posiciones* ocupadas por ese *barco* entonces en lugar de *tocado*, el *jugador* atacado debe responder *hundido*. El juego termina cuando, tras un ataque, todos los *barcos* del *jugador* atacado fueron hundidos.

En la figura 1 se muestra una versión del juego a modo de ilustración. El juego tiene *dimensiones* 10×10 (así que se usan las *letras* A-J para las *filas* y los *números* 1-10 para las *columnas*) y 5 *barcos* (uno de *tamaño* 2, dos de *tamaño* 3, uno de *tamaño* 4 y uno de *tamaño* 5). La *grilla local* es la de abajo, donde están ubicados los *barcos* del *jugador*. Los indicadores rojos sobre los *barcos* en dicha *grilla* son las *posiciones* que fueron atacadas por el oponente. Tres de los *barcos* del *jugador* fueron descubiertos aunque ninguno fue hundido todavía (a todos les queda al menos una *posición* que no fue atacada todavía). La *grilla del oponente* es la de arriba, donde el *jugador* intenta replicar la *grilla local* del oponente. Los indicadores rojos son las *posiciones* donde el *jugador* lanzó torpedos y el oponente indicó allí la presencia de uno de sus *barcos*. Los indicadores blancos son las *posiciones* donde el *jugador* lanzó torpedos y el oponente indicó allí la ausencia de *barcos*.

Archivos

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

- **interfaz_batallaNaval.py**: este archivo no deben modificarlo, salvo alguna prueba básica. Contiene la lógica para manejar la interfaz gráfica del juego, llamando algunos de los procedimientos y funciones que ustedes deben implementar. Tengan en cuenta que, dado que la implementación solicitada no es completa, la interfaz tampoco lo es. Solo permitirá jugar con un jugador, y luego del disparo que elijan, la PC como jugador 2 efectuará un disparo sobre su tablero.
- **batallaNaval.py**: contendrá la lógica del juego. Los procedimientos y funciones a implementar ya están en el archivo, sin funcionalidad. **No deben cambiarles los nombres**, pero sí pueden (y deben) agregar los procedimientos y funciones auxiliares que consideren necesarios para implementar las funcionalidades pedidas de forma modular.
- **biblioteca.py**: Este archivo contiene tipos, funciones y procedimientos ya implementados. No deben modificar este archivo.
- **cubrimiento.py**: Script para obtener cubrimiento de líneas y ramas de forma simple. No deben modificar este archivo.
- **tests_materia.py**: un ejemplo de caso de test en formato unittest 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 los procedimientos y 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 `batallaNaval.py`.
- No está permitido alterar los nombres de los procedimientos y funciones a implementar ni los tipos de datos. Deben mantenerse tal cual como descargan.
- Pueden definir todos los procedimientos y funciones auxiliares que requieran.
- Deben tipar las variables, las funciones y los parámetros que definan.

- No está permitido utilizar técnicas no vistas en clase para resolver los ejercicios. En el campus, pueden encontrar un listado de todos los procedimientos y funciones válidos a usar, así como también los operadores y otras construcciones del lenguaje (como alternativas y repeticiones). Si usan alguna construcción 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 deben comentarlo en el formulario cuando envían el TP. 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 variables, parámetros, procedimientos y funciones que se definan deben ser apropiados al problema que se resuelve. Los nombres declarativos ayudan a la legibilidad del código. La estructura del código también debería comunicar de manera acorde la intención. Para ello, se espera que eviten el uso de `break` o de múltiples `return`s así como también se espera que eviten manifestar síntomas del síndrome del miedo al booleano.
- **Modularización**: evitar repetir código innecesariamente y usar adecuadamente los procedimientos y funciones definidos previamente (por el enunciado o por ustedes mismos). Pueden agregar más procedimientos y funciones para separar el código en tareas específicas y facilitar su lectura. Se recomienda evitar a toda costa más de un nivel de anidamiento en el código (si tienen que tabular 3 veces seguidas, probablemente les esté faltando una subtarea).
- **Documentación**: Deben usar docstrings. Recuerden que un comentario con # no es documentación. Buscamos que su código empiece a cumplir un estándar básico de calidad: describan parámetros, retorno y **qué** hace (no **cómo** lo hace) cada función. En caso de tenerlas, cuáles son sus precondiciones. Los tests no requieren docstrings, pero sí respeten el estándar de legibilidad con buenos nombres de funciones y variables. Si lo consideran necesario agreguen aclaraciones de comentarios para el equipo de programación que lea su código con decisiones de diseño e implementación. Es obligatorio contar al menos con esta documentación para los cinco problemas que deben implementar. Para las funciones auxiliares que implementen, es aconsejable. Un ejemplo podría ser el siguiente:

```
def esBisiesto(año: int) -> bool:
    """
    Indica si el año *año* es bisiesto.

    PRE: True

    Args:
        año (int): El número de año para el cual se indica si es bisiesto.

    Returns:
        True si el año *año* es bisiesto y False si no.
    """
    return (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0)
```

- **Testing**: Todos los ejercicios del enunciado (sólo los procedimientos y funciones que se piden, no los procedimientos y funciones auxiliares que implementen) 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 los procedimientos y funciones y **deben tener una cobertura de al menos 95 % de líneas de código, y al menos 95 % de cobertura de ramas para todo el archivo batallaNaval.py, no por procedimiento o función**. Esto se verá en la teórica de Testing de caja Blanca y en su clase de laboratorio asociado, utilizando las herramientas `unittest` y `coverage`.

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. Les 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.
- Deben incluir todo archivo/carpeta necesario para ejecutar los casos de test.
- Todos los casos de test deben pasar satisfactoriamente, y el archivo con la implementación debe poder cargarse sin generar errores.

Enunciado

Gran parte de la implementación ya está resuelta. Se piden implementar algunas funciones y algunos procedimientos particulares para validar ciertas características de las estructuras utilizadas para representar los elementos el juego.

Tipos de datos

Para especificar los problemas a resolver usaremos los siguientes tipos de datos[1]:

- Renombre Celda = VACÍO | AGUA | BARCO
- Renombre ResultadoDisparo = NADA | TOCADO
- Renombre Grilla = $seq\langle seq\langle Celda\rangle\rangle$
- Renombre Tablero = Grilla × Grilla
- Renombre Dimensiones = $\mathbb{Z} \times \mathbb{Z}$
- Renombre Posición = Char × $\mathbb{Z}$
- Renombre Jugador = UNO | DOS
- Renombre Barco = $\mathbb{Z}$
- Renombre BarcoEnGrilla = $seq\langle Posición\rangle$
- Renombre Dirección = ARRIBA | ABAJO | IZQUIERDA | DERECHA
- Renombre EstadoJuego = Dimensiones × $seq\langle Barco\rangle$ × $seq\langle Jugador\rangle$ × Tablero × Tablero

El tipo **Celda** se usa para representar el contenido de una *posición* en una *grilla* (**VACÍO** si no tiene nada, **AGUA** si tiene un indicador de *agua* y **BARCO** si tiene un indicador de *barco*).

El tipo **ResultadoDisparo** se usa para representar el resultado de haber efectuado un disparo (**NADA** si el disparo no alcanzó un barco, **TOCADO** si le dieron a un barco).

El tipo **Grilla** se usa para representar una *grilla* como una secuencia de *filas* de la *grilla*, donde cada fila es, a su vez, otra secuencia pero esta vez de las *posiciones* de dicha *fila* (representadas a partir de elementos del tipo **Celda**). Para ser una *grilla* válida debe ser una matriz (es decir, todas las filas deben tener la misma longitud), tener al menos una *columna* y tener entre 1 y 26 *filas* (no podría tener más porque sólo podemos usar 26 *letras* para identificar las *filas*).

El tipo **Tablero** se usa para representar el *tablero* de un *jugador* a partir de las dos *grillas* (primero la *grilla local* y luego la *grilla del oponente*). Para ser un *tablero* válido ambas *grillas* deben ser válidas y tener las mismas *dimensiones*.

El tipo **Dimensiones** se usa para representar las *dimensiones* del juego (es decir, la cantidad de *filas* y *columnas* que deben tener las cuatro *grillas*) a partir de dos números enteros (primero la cantidad de *filas* y luego la cantidad de *columnas*).

El tipo **Posición** se usa para representar las coordenadas de una *posición* en una *grilla* a partir de la *fila* (dada por una *letra*) y la columna (dada por un *número*) de dicha *posición*. Para ser una *posición* válida la *letra* debe estar entre ord('A') y ord('Z'), inclusive y la *columna* ser mayor a 0.

El tipo **Jugador** se usa para representar a cada uno de los *jugadores* (**UNO** para el *jugador* 1 y **DOS** para el *jugador* 2).

El tipo **Barco** se usa para representar un modelo de *barco* disponible en el juego (no uno de los *barcos* de un *jugador* ya posicionado en la *grilla local* del mismo) a partir de su *tamaño*.

El tipo **BarcoEnGrilla** se usa para representar un *barco* de un *jugador* ya posicionado en la *grilla local* del mismo a partir de la secuencia de *posiciones* que dicho *barco* ocupa en la *grilla*. Para que sea un *barco* válido todas las *posiciones* deben ser seguidas en línea recta, horizontal o vertical pero no diagonal.

El tipo **Dirección** se usa para representar las cuatro direcciones de *posiciones* adyacentes a una *posición* de una *grilla*.

El tipo **EstadoJuego** se usa para representar un juego en progreso a partir de las *dimensiones* de las *grillas*, la lista de *barcos* disponibles de cada *jugador*, el *jugador* al que le toca y los *tableros* de cada uno (primero el del *jugador* 1 y después el del *jugador* 2). Para ser un estado del juego válido las 4 *grillas* deben tener las mismas *dimensiones* que el estado y los *barcos* que estén en las *grillas* deben ser los que están disponibles según el estado (en caso de que ya se hayan colocado).

[1] Recordar que usaremos la siguiente sintaxis para definir tipos enumerativos:
Renombre <**NOMBRE_DEL_TIPO**> = <**CASO_1**> | <**CASO_2**> | ... | <**CASO_n**>

Tipos, procedimientos y funciones ya definidos

Todos los tipos mencionados en la sección anterior ya se encuentran implementados:

```
Celda = VACÍO | AGUA | BARCO    # contenido de una celda
ResultadoDisparo = NADA | TOCADO    # resultado de un disparo
Grilla = list[list[Celda]]      # una grilla, dada por una matriz de celdas
Tablero = tuple[Grilla,Grilla]       # un tablero, dado por la grilla local y la grilla del oponente
Dimensiones = tuple[int,int]         # cantidad de filas (alto) y cantidad de columnas (ancho) de las grillas
Posición = tuple[str,int]            # una ubicación de una grilla, dada por una letra y un número
Jugador = UNO | DOS             # identificador de jugador
Barco = int                     # definición de barco (sólo su tamaño)
BarcoEnGrilla = list[Posición]  # un barco ubicado en la grilla (lista de posiciones que ocupa en la grilla)
Dirección = ARRIBA | ABAJO | IZQUIERDA | DERECHA
EstadoJuego = tuple[
    Dimensiones,                # dimensiones de las grillas
    list[Barco],                # barcos disponibles
    list[Jugador],              # turno
    Tablero,                    # tablero jugador 1
    Tablero                     # tablero jugador 2
]
```

También se encuentran implementados procedimientos y funciones para los siguientes problemas. Si bien no es obligatorio utilizarlos, pueden simplificar bastante el trabajo. Se recomienda leerlos detenidamente para entender cómo usarlos y en qué escenarios serían útiles y luego volver a revisarlos al empezar a plantear la estrategia de solución de cada ejercicio.

`problema esLetraVálida` (in letra: Char) : Bool {
    `requiere nada:` { True }
    `asegura EsUnaLetraEntreAYZ:` { resultado = true $\iff$ |letra| = 1 $\land$ ord('A') $\leq$ ord(letra) $\leq$ ord('Z') }
}

`problema númeroDeFila` (in letra: Char) : $\mathbb{Z}$ {
    `requiere LaLetraEsVálida:` { esLetraVálida(letra) }
    `asegura:` { resultado = ord(letra) - ord('A') + 1 }
}

`problema siguienteLetra` (in letra: Char) : Char {
    `requiere LaLetraEsVálida:` { esLetraVálida(letra) }
    `requiere LaLetraNoEsLaÚltima:` { ord(letra) < ord('Z') }
    `asegura:` { resultado = chr(ord(letra) + 1) }
}

`problema anteriorLetra` (in letra: Char) : Char {
    `requiere LaLetraEsVálida:` { esLetraVálida(letra) }
    `requiere LaLetraNoEsLaPrimera:` { ord(letra) > ord('A') }
    `asegura:` { resultado = chr(ord(letra) - 1) }
}

`problema direccionesOrtogonales` () : $seq\langle Dirección \rangle$ {
    `requiere nada:` { True }
    `asegura:` { resultado = $\langle ARRIBA, ABAJO, IZQUIERDA, DERECHA \rangle$ }
}

`problema cantidadDeApariciones` (in elementoBuscado: T, in lista: $seq\langle T \rangle$) : $\mathbb{Z}$ {
    `requiere nada:` { True }
    `asegura:` { El resultado es la cantidad de apariciones del elemento 'elementoBuscado' en la lista 'lista'. }
}

`problema unoSiCeroSiNo` (in condición: Bool) : $\mathbb{Z}$ {
    `requiere nada:` { True }
    `asegura:` { (condición = true $\to$ resultado = 1) $\land$
        (condición = false $\to$ resultado = 0) }

}

problema esMatrizVálida (in matriz: $seq\langle seq\langle T\rangle\rangle$) : Bool {
  requiere nada: { True }
  asegura TodasLasFilasSonDeLaMismaLongitud: { resultado = true $\iff$
    |matriz| = 0 $\vee$
      Para toda fila 'fila' de tipo $seq\langle T\rangle$ en 'matriz' se cumple:
        |fila| = |matriz[0]|
  }
}

problema anchoDimensiones (in dimensiones: Dimensiones) : $\mathbb{Z}$ {
  requiere nada: { True }
  asegura: { resultado = dimensiones_1 }
}

problema altoDimensiones (in dimensiones: Dimensiones) : $\mathbb{Z}$ {
  requiere nada: { True }
  asegura: { resultado = dimensiones_0 }
}

problema mismasDimensiones (in dimensiones1: Dimensiones, in dimensiones2: Dimensiones) : Bool {
  requiere nada: { True }
  asegura TodasLasFilasSonDeLaMismaLongitud: { resultado = true $\iff$
    (anchoDimensiones(dimensiones1) = anchoDimensiones(dimensiones2)
    $\wedge$
    altoDimensiones(dimensiones1) = altoDimensiones(dimensiones2))
  }
}

problema esGrillaVálida (in grilla: Grilla) : Bool {
  requiere nada: { True }
  asegura EsMatrizVálidaConAlMenosUnaColumnaYEntre1Y26Filas: { resultado = true $\iff$
    (esMatrizVálida(grilla)
    $\wedge$
    $1 \leq$ cantidadDeFilasGrilla(grilla) ≤ 26
    $\wedge$
    $1 \leq$ cantidadDeColumnasGrilla(grilla))
  }
}

problema dimensionesGrilla (in grilla: Grilla) : Dimensiones {
  requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
  asegura: { resultado = (|grilla|, $|\text{grilla}_0|$) }
}

problema cantidadDeFilasGrilla (in grilla: Grilla) : $\mathbb{Z}$ {
  requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
  asegura: { resultado = altoDimensiones(dimensionesGrilla(grilla)) }
}

problema cantidadDeColumnasGrilla (in grilla: Grilla) : $\mathbb{Z}$ {
  requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
  asegura: { resultado = anchoDimensiones(dimensionesGrilla(grilla)) }
}

problema grillaVálidaEnJuego (in grilla: Grilla, in estadoDeJuego: EstadoJuego) : Bool {
  requiere nada: { True }
  asegura EsMatrizVálidaYCoincidenLasDimensionesConLasDelJuego: {
    resultado = true $\iff$
      esMatrizVálida(grilla)

```
        ∧
        mismasDimensiones(dimensionesGrilla(grilla), dimensionesEstadoJuego(estadoDeJuego))
    }
}

problema grillaLocal (in tablero: Tablero) : Grilla {
    requiere nada: { True }
    asegura: { resultado = tablero_0 }
}

problema grillaOponente (in tablero: Tablero) : Grilla {
    requiere nada: { True }
    asegura: { resultado = tablero_1 }
}

problema dimensionesEstadoJuego (in estadoDeJuego: EstadoJuego) : Dimensiones {
    requiere nada: { True }
    asegura: { resultado = estadoDeJuego_0 }
}

problema cantidadDeFilasEstadoJuego (in estadoDeJuego: EstadoJuego) : ℤ {
    requiere nada: { True }
    asegura: { resultado = altoDimensiones(dimensionesEstadoJuego(estadoDeJuego)) }
}

problema cantidadDeColumnasEstadoJuego (in estadoDeJuego: EstadoJuego) : ℤ {
    requiere nada: { True }
    asegura: { resultado = anchoDimensiones(dimensionesEstadoJuego(estadoDeJuego)) }
}

problema barcosDisponibles (in estadoDeJuego: EstadoJuego) : seq⟨Barco⟩ {
    requiere nada: { True }
    asegura: { resultado = estadoDeJuego_1 }
}

problema turno (in estadoDeJuego: EstadoJuego) : Jugador {
    requiere nada: { True }
    asegura: { resultado = estadoDeJuego_2[0] }
}

problema tableroDeJugador (in estadoDeJuego: EstadoJuego, in jugador: Jugador) : Tablero {
    requiere nada: { True }
    asegura: { (jugador = UNO → resultado = estadoDeJuego_3) ∧
        (jugador = DOS → resultado = estadoDeJuego_4) }
}

problema cambiarTurno (inout estadoDeJuego: EstadoJuego) {
    requiere nada: { True }
    modifica: {estadoDeJuego}
    asegura estadoJuegoSoloActualizaUnaCelda: {
      Todas las componentes de 'estadoDeJuego' permanecen iguales a sus respectivas
        componentes en 'estadoDeJuego@pre', excepto por el turno que es el opuesto
        a 'turno(estadoDeJuego@pre)'
    }
}

problema esPosiciónVálida (in posición: Posición) : Bool {
    requiere nada: { True }
    asegura LaLetraEsVálidaYElNúmeroEsMayorA0: { resultado = true ⟺
      (esLetraVálida(letraDePosición(posición))
      ∧
```

```
        númeroDePosición(posición) > 0)
      }
}

problema letraDePosición (in posición: Posición) : Char  {
      requiere nada: { True }
      asegura: { resultado = posición_0 }
}

problema númeroDePosición (in posición: Posición) : ℤ  {
      requiere nada: { True }
      asegura: { resultado = posición_1 }
}

problema mismaPosición (in posición1: Posición, in posición2: Posición) : Bool  {
      requiere nada: { True }
      asegura MismaLetraYMismoNúmero: { resultado = true ⟺
        (letraDePosición(posición1) = letraDePosición(posición2)
        ∧
        númeroDePosición(posición1) = númeroDePosición(posición2))
      }
}

problema hayPosiciónAdyacenteEn (in posición: Posición, in posiciones: seq⟨Posición⟩) : Bool  {
      requiere nada: { True }
      asegura: { resultado = true ⟺
        Para alguna posición 'otraPosición' de tipo Posición en 'posiciones' se cumple:
          sonPosicionesAdyacentes(posición, otraPosición)
      }
}

problema sonPosicionesAdyacentes (in posición1: Posición, in posición2: Posición) : Bool  {
      requiere nada: { True }
      asegura MismaFilaYUnaColumnaDeDiferenciaOViceversa: { resultado = true ⟺
        (
          letraDePosición(posición1) = letraDePosición(posición2)
        ∧
          |númeroDePosición(posición1) - númeroDePosición(posición2)| = 1
        ) ∨ (
          númeroDePosición(posición1) = númeroDePosición(posición2)
        ∧
          |númeroDeFila(letraDePosición(posición1)) - númeroDeFila(letraDePosición(posición2))| = 1
        )
      }
}

problema esPosiciónVálidaEnGrilla (in posición: Posición, in grilla: Grilla) : Bool  {
      requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
      asegura: { resultado = true ⟺
        (esPosiciónVálida(posición)
        ∧
        númeroDeFila(letraDePosición(posición)) ≤ cantidadDeFilasGrilla(grilla)
        ∧
        númeroDePosición(posición) ≤ cantidadDeColumnasGrilla(grilla))
      }
}

problema sonPosicionesVálidasEnGrilla (in posiciones: seq⟨Posición⟩, in grilla: Grilla) : Bool  {
      requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
```

```
        asegura: { resultado = true ⟺
          Para toda posición 'posición' de tipo Posición en 'posiciones' se cumple:
            esPosiciónVálidaEnGrilla(posición, grilla)
        }
}

problema celdaEnPosición (in grilla: Grilla, in posición: Posición) : Celda  {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        asegura: { resultado = grilla[númeroDeFila(letraDePosición(posición))-1][númeroDePosición(posición)-1]}
}

problema primeraPosiciónEnGrilla (in grilla: Grilla) : Posición  {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        asegura: { resultado = ('A',1) }
}

problema esLaÚltimaPosiciónEnGrilla (in posición: Posición, in grilla: Grilla) : Bool  {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        asegura: { resultado = true ⟺
          (númeroDeFila(letraDePosición(posición)) = cantidadDeFilasGrilla(grilla)
          ∧
          númeroDePosición(posición) = cantidadDeColumnasGrilla(grilla))
        }
}

problema posiciónSiguienteEnGrilla (in posición: Posición, in grilla: Grilla) : Posición  {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        requiere laPosiciónNoEsLaÚltimaDeLaGrilla: { ¬esLaÚltimaPosiciónEnGrilla(posición, grilla) }
        asegura SiEsLaÚltimaPosiciónDeLaFilaDevuelveLaPrimeraDeLaPróximaFila: {
          númeroDePosición(posición) = cantidadDeColumnasGrilla(grilla) →
            resultado = (siguienteLetra(letraDePosición(posición)), 1)
        }
        asegura SiNoDevuelveLaPróximaDeLaMismaFila: {
          númeroDePosición(posición) < cantidadDeColumnasGrilla(grilla) →
            resultado = (letraDePosición(posición), númeroDePosición(posición) + 1)
        }
}

problema hayPosiciónAdyacenteAl (in grilla: Grilla, in posición: Posición, in dirección: Dirección) : Bool  {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        asegura: { dirección = ARRIBA →
          resultado = númeroDeFila(letraDePosición(posición)) < cantidadDeFilasGrilla(grilla)
        }
        asegura: { dirección = DERECHA →
          resultado = númeroDePosición(posición) < cantidadDeColumnasGrilla(grilla)
        }
        asegura: { dirección = ABAJO →
          resultado = númeroDeFila(letraDePosición(posición)) > 1
        }
        asegura: { dirección = IZQUIERDA →
          resultado = númeroDePosición(posición) > 1
        }
}

problema posiciónAdyacenteAl (in grilla: Grilla, in posición: Posición, in dirección: Dirección) : Posición  {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
```

```
      requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
      requiere existeUnaPosiciónAdyacente: { hayPosiciónAdyacenteAl(grilla, posición, dirección) }
      asegura: { dirección = ARRIBA →
       resultado = (siguienteLetra(letraDePosición(posición)), númeroDePosición(posición))
      }
      asegura: { dirección = DERECHA →
       resultado = (letraDePosición(posición), númeroDePosición(posición)+1)
      }
      asegura: { dirección = ABAJO →
       resultado = (anteriorLetra(letraDePosición(posición)), númeroDePosición(posición))
      }
      asegura: { dirección = IZQUIERDA →
       resultado = (letraDePosición(posición), númeroDePosición(posición)-1)
      }
}

problema cambiarCeldaGrilla (inout grilla: Grilla, in posición: Posición, in contenido: Celda) {
      requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
      requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
      modifica: {grilla}
      asegura LaCeldaCambió: { celdaEnPosición(grilla, posición) = contenido }
      asegura LasDemásCeldasNoCambiaron: {
       Para toda posición 'posición2' de tipo Posición distinta a 'posición' que sea válida en la grilla 'grilla' se cumple:
         celdaEnPosición(grilla, posición2) = celdaEnPosición(grilla@pre, posición2)
      }
}

problema sonPosicionesVálidas (in posiciones: seq⟨Posición⟩) : Bool {
      requiere nada: { True }
      asegura: { resultado = true ⟺
       Para toda posición 'posición' de tipo Posición en 'posiciones' se cumple:
         esPosiciónVálida(posición)
      }
}

problema pertenecePosición (in posiciónBuscada: Posición, in posiciones: seq⟨Posición⟩) : Bool {
      requiere nada: { True }
      asegura: { resultado = true ⟺ posiciónBuscada ∈ posiciones }
}

problema todasEnLaMismaFila (in posiciones: seq⟨Posición⟩) : Bool {
      requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(posiciones) }
      asegura: { resultado = true ⟺
       |posiciones| = 0 ∨
         Para toda posición 'posición' de tipo Posición en 'posiciones' se cumple:
           letraDePosición(posición) = letraDePosición(posiciones[0])
      }
}

problema todasEnLaMismaColumna (in posiciones: seq⟨Posición⟩) : Bool {
      requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(posiciones) }
      asegura: { resultado = true ⟺
       |posiciones| = 0 ∨
         Para toda posición 'posición' de tipo Posición en 'posiciones' se cumple:
           númeroDePosición(posición) = númeroDePosición(posiciones[0])
      }
}

problema columnasConsecutivasAscendentes (in posiciones: seq⟨Posición⟩) : Bool {
      requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(posiciones) }
```

```
        asegura: { resultado = true ⟺
          |posiciones| = 0 ∨
            Para todo índice 'i' de tipo ℤ entre 1 y |posiciones|-1 (inclusive) se cumple:
              númeroDePosición(posiciones[i]) = númeroDePosición(posiciones[i-1]) + 1
        }
}

problema columnasConsecutivasDescendentes (in posiciones: seq⟨Posición⟩) : Bool {
        requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(posiciones) }
        asegura: { resultado = true ⟺
          |posiciones| = 0 ∨
            Para todo índice 'i' de tipo ℤ entre 1 y |posiciones|-1 (inclusive) se cumple:
              númeroDePosición(posiciones[i]) = númeroDePosición(posiciones[i-1]) - 1
        }
}

problema filasConsecutivasAscendentes (in posiciones: seq⟨Posición⟩) : Bool {
        requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(posiciones) }
        asegura: { resultado = true ⟺
          |posiciones| = 0 ∨
            Para todo índice 'i' de tipo ℤ entre 1 y |posiciones|-1 (inclusive) se cumple:
              númeroDeFila(letraDePosición(posiciones[i])) = númeroDeFila(letraDePosición(posiciones[i-1])) + 1
        }
}

problema filasConsecutivasDescendentes (in posiciones: seq⟨Posición⟩) : Bool {
        requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(posiciones) }
        asegura: { resultado = true ⟺
          |posiciones| = 0 ∨
            Para todo índice 'i' de tipo ℤ entre 1 y |posiciones|-1 (inclusive) se cumple:
              númeroDeFila(letraDePosición(posiciones[i])) = númeroDeFila(letraDePosición(posiciones[i-1])) - 1
        }
}

problema posicionesOrdenadas (in posiciones: seq⟨Posición⟩) : seq⟨Posición⟩ {
        requiere nada: { True }
        asegura mismaLongitud: { |posiciones| = |resultado| }
        asegura correspondenciaUnoAUno: {
          Para toda posición 'posición' de tipo Posición se cumple:
            cantidadDeApariciones(posición, posiciones) = cantidadDeApariciones(posición, resultado)
        }
        asegura elResultadoEstáOrdenado: {
          Las posiciones de 'resultado' están ordenadas bajo algún criterio
        }
}

problema tamañoBarco (in barco: BarcoEnGrilla) : ℤ {
        requiere elBarcoEsVálido: { esBarcoVálido(barco) }
        asegura: { resultado = |barco| }
}

problema esBarcoVálido (in barco: BarcoEnGrilla) : Bool {
        requiere nada: { True }
        asegura: { resultado = true ⟺
          (tamañoBarco(barco) > 0 ∧
          sonPosicionesVálidas(barco) ∧
            (esBarcoVálidoHorizontal(barco)
            ∨
            esBarcoVálidoVertical(barco))
```

```
        )
      }
}

problema sonBarcosVálidos (in barcos: seq⟨BarcoEnGrilla⟩) : Bool {
      requiere nada: { True }
      asegura: { resultado = true ⟺
        (
          Para todo barco 'barco' de tipo BarcoEnGrilla en 'barcos' se cumple:
            esBarcoVálido(barco)
        )
        ∧
        ¬hayBarcosSuperpuestosOAdyacentes(barcos)
      }
}

problema esBarcoVálidoHorizontal (in barco: BarcoEnGrilla) : Bool {
      requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(barco) }
      asegura: { resultado = true ⟺
        (todasEnLaMismaFila(barco)
        ∧
        columnasConsecutivasAscendentes(posicionesOrdenadas(barco)))
      }
}

problema esBarcoVálidoVertical (in barco: BarcoEnGrilla) : Bool {
      requiere LasPosicionesSonVálidas: { sonPosicionesVálidas(barco) }
      asegura: { resultado = true ⟺
        (todasEnLaMismaColumna(barco)
        ∧
        filasConsecutivasAscendentes(posicionesOrdenadas(barco)))
      }
}

problema hayBarcosSuperpuestosOAdyacentes (in barcos: seq⟨BarcoEnGrilla⟩) : Bool {
      requiere losBarcosSonVálidosPorSeparado: {
        Para todo barco 'barco' de tipo BarcoEnGrilla en 'barcos' se cumple:
          esBarcoVálido(barco)
      }
      asegura: { resultado = true ⟺
        Para algún número 'i' de tipo ℤ en el rango [0,|barcos|) se cumple:
          colisionaBarco(barcos[i], subseq(barcos, i+1, |barcos|-1))
          ∨
          hayBarcoAdyacente(barcos[i], subseq(barcos, i+1, |barcos|-1))
      }
}

problema colisionaBarco (in barco: BarcoEnGrilla, barcos: seq⟨BarcoEnGrilla⟩) : Bool {
      requiere losBarcosSonVálidosPorSeparado: {
        esBarcoVálido(barco) ∧
        Para todo barco 'otroBarco' de tipo BarcoEnGrilla en 'barcos' se cumple:
          esBarcoVálido(otroBarco)
      }
      asegura: { resultado = true ⟺
        Para algún barco 'otroBarco' de tipo BarcoEnGrilla en 'barcos' se cumple:
          colisionaBarcoCon(barco, otroBarco)
      }
}

problema hayBarcoAdyacente (in barco: BarcoEnGrilla, barcos: seq⟨BarcoEnGrilla⟩) : Bool {
```

```
    requiere losBarcosSonVálidosPorSeparado: {
      esBarcoVálido(barco) ∧
      Para todo barco 'otroBarco' de tipo BarcoEnGrilla en 'barcos' se cumple:
        esBarcoVálido(otroBarco)
    }
    asegura: { resultado = true ⟺
      Para algún barco 'otroBarco' de tipo BarcoEnGrilla en 'barcos' se cumple:
        sonAdyacentesEnAlgunaPosición(barco, otroBarco)
    }
}

problema colisionaBarcoCon (in barco1: BarcoEnGrilla, in barco2: BarcoEnGrilla) : Bool {
    requiere losBarcosSonVálidos: {
      esBarcoVálido(barco1) ∧ esBarcoVálido(barco2)
    }
    asegura: { resultado = true ⟺
      Para alguna posición 'posición' de tipo Posición en 'barco1' se cumple:
        pertenecePosición(posición, barco2)
    }
}

problema sonAdyacentesEnAlgunaPosición (in barco1: BarcoEnGrilla, in barco2: BarcoEnGrilla) : Bool {
    requiere losBarcosSonVálidos: {
      esBarcoVálido(barco1) ∧ esBarcoVálido(barco2)
    }
    asegura: { resultado = true ⟺
      Para alguna posición 'posición' de tipo Posición en 'barco1' se cumple:
        hayPosiciónAdyacenteEn(posición, barco2)
    }
}

problema sePuedeColocarBarco (in barco: BarcoEnGrilla, in grilla: Grilla) : Bool {
    requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
    asegura: { resultado = true ⟺
      esBarcoVálido(barco)
      ∧
      sonPosicionesVálidasEnGrilla(barco, grilla)
      ∧
      ¬colisionaBarcoEnGrilla(barco, grilla)
      ∧
      ¬quedaBarcoAdyacenteAOtroBarcoEnGrilla(barco, grilla)
    }
}

problema colisionaBarcoEnGrilla (in barco: BarcoEnGrilla, in grilla: Grilla) : Bool {
    requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
    requiere elBarcoEsVálido: { esBarcoVálido(barco) }
    requiere elBarcoQuedaDentroDeLaGrilla: { sonPosicionesVálidasEnGrilla(barco, grilla) }
    asegura: { resultado = true ⟺
      Para alguna posición 'posición' de tipo Posición en 'barco' se cumple:
        celdaEnPosición(grilla, posición) = BARCO
    }
}

problema quedaBarcoAdyacenteAOtroBarcoEnGrilla (in barco: BarcoEnGrilla, in grilla: Grilla) : Bool {
    requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
    requiere elBarcoEsVálido: { esBarcoVálido(barco) }
    requiere elBarcoQuedaDentroDeLaGrilla: { sonPosicionesVálidasEnGrilla(barco, grilla) }
    asegura: { resultado = true ⟺
      Para alguna posición 'posición' de tipo Posición en 'barco' se cumple:
```

```
        hayBarcoAdyacenteEnGrilla(posición, grilla)
    }
}

problema hayBarcoAdyacenteEnGrilla (in grilla: Grilla, in posición: Posición) : Bool {
    requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
    requiere laPosiciónEsVálidaEnLaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
    asegura: { resultado = true ⟺
      Para alguna dirección 'dirección' de tipo Dirección se cumple:
        hayBarcoAl(grilla, posición, dirección)
    }
}

problema hayBarcoAl (in grilla: Grilla, in posición: Posición, in dirección: Dirección) : Bool {
    requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
    requiere laPosiciónPerteneceALaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
    asegura: { resultado = true ⟺
      hayPosiciónAdyacenteAl(grilla, posición, dirección)
      ∧  celdaEnPosición(grilla, posiciónAdyacenteAl(grilla, posición, dirección)) = BARCO
    }
}

problema colocarBarcos (inout estadoDeJuego: EstadoDeJuego, in barcosAColocar: seq⟨BarcoEnGrilla⟩, in jugador: Jugador) {
    requiere juegoVálido: { esEstadoDeJuegoVálido(estadoDeJuego) }
    requiere losBarcosSonVálidos: { sonBarcosVálidos(barcosAColocar) }
    requiere grillaLocalVacía: { esGrillaVacía(grillaLocal(tableroDeJugador(estadoDeJuego, jugador))) }
    requiere sePuedenColocarLosBarcos: {
      Para todo barco 'barco' de tipo BarcoEnGrilla en 'barcosAColocar' se cumple:
        sePuedeColocarBarco(barco, grillaLocal(tableroDeJugador(estadoDeJuego, jugador)))
    }
    modifica: {estadoDeJuego}
    asegura estadoJuegoSoloActualizaGrillaLocalDeJugador: {
      Todas las componentes de 'estadoDeJuego' permanecen iguales a sus respectivas componentes
        en 'estadoDeJuego@pre', excepto por la grilla local del jugador 'jugador'.
    }
    asegura seColocanBarcosEnGrilla: {
      Para cada posición 'posición' de cada barco en 'barcosAColocar' se cumple:
        celdaEnPosición(grillaLocal(tableroDeJugador(estadoDeJuego, jugador)), posición) = BARCO
    }
    asegura celdasQueNoCambian: {
      Para cada posición 'posición' de la grilla local del jugador 'jugador' que no esté contenida
        en ningún barco de 'barcosAColocar' se cumple:
          celdaEnPosición(grillaLocal(tableroDeJugador(estadoDeJuego, jugador)), posición) = VACÍO
    }
}
```

Ejercicios

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.

Ejercicio 1

problema cantidadDeBarcosDeTamaño (in barcos: $seq\langle BarcoEnGrilla\rangle$, tamaño: $\mathbb{Z}$) : $\mathbb{Z}$ {
requiere losBarcosSonVálidos: { sonBarcosVálidos(barcos) }
asegura: { El resultado es la cantidad de barcos de tamaño 'tamaño' en la lista 'barcos'. }
}

Ejemplo:
cantidadDeBarcosDeTamaño(⟨
$\langle('H',3),('H',4),('H',5)\rangle$,
$\langle('F',4),('E',4)\rangle$,
$\langle('B',4),('B',3),('B',2)\rangle$
⟩,2) = 1

Ejercicio 2

problema nuevoJuego (in cantidadDeFilas: $\mathbb{Z}$, in cantidadDeColumnas: $\mathbb{Z}$, in barcosDisponibles: $seq\langle Barco\rangle$) : EstadoJuego
{
requiere cantidadDeFilasVálida: { cantidadDeFilas $\geq 1 \wedge$ cantidadDeFilas ≤ 26 }
requiere cantidadDeColumnasVálida: { cantidadDeColumnas > 0 }
requiere cantidadDeBarcosVálida: { |barcosDisponibles| > 0 }
asegura: { resultado = (
(cantidadDeFilas, cantidadDeColumnas),
barcosDisponibles,
$\langle UNO\rangle$,
nuevoTablero(cantidadDeFilas, cantidadDeColumnas),
nuevoTablero(cantidadDeFilas, cantidadDeColumnas)
)}
}

problema nuevoTablero (in cantidadDeFilas: $\mathbb{Z}$, in cantidadDeColumnas: $\mathbb{Z}$) : Tablero {
requiere cantidadDeFilasVálida: { cantidadDeFilas $\geq 1 \wedge$ cantidadDeFilas ≤ 26 }
requiere cantidadDeColumnasVálida: { cantidadDeColumnas > 0 }
asegura laGrillaLocalEsUnaGrillaVacía: {
grillaLocal(resultado) = grillaVacía(cantidadDeFilas, cantidadDeColumnas)
}
asegura laGrillaOponenteEsUnaGrillaVacía: {
grillaOponente(resultado) = grillaVacía(cantidadDeFilas, cantidadDeColumnas)
}
}

problema grillaVacía (in cantidadDeFilas: $\mathbb{Z}$, in cantidadDeColumnas: $\mathbb{Z}$) : Grilla {
requiere cantidadDeFilasVálida: { cantidadDeFilas $\geq 1 \wedge$ cantidadDeFilas ≤ 26 }
requiere cantidadDeColumnasVálida: { cantidadDeColumnas > 0 }
asegura laGrillaEsVálida: { esGrillaVálida(resultado) }
asegura laGrillaTieneLasDimensionesPedidas: {
mismasDimensiones(dimensionesGrilla(resultado), (cantidadDeFilas, cantidadDeColumnas))
}
asegura laGrillaEstáVacía: { esGrillaVacía(resultado) }
}

problema esGrillaVacía (in grilla: Grilla) : Bool {
requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
asegura: { resultado = true $\iff$
Para toda posición 'posición' de tipo Posición que sea válida en la grilla 'grilla' se cumple:
celdaEnPosición(grilla, posición) = VACÍO
}
}

Ejemplo:
nuevoJuego(2,2,⟨2⟩) = (

(2,2),⟨2⟩,⟨UNO⟩,(
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩,
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩
),(
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩,
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩
)
)

Ejercicio 3

`problema esEstadoDeJuegoVálido` (in estadoDeJuego: EstadoJuego) : Bool {
`requiere nada:` { True }
`asegura`: { resultado =
cantidadDeFilasEstadoJuego(estadoDeJuego) $\geq 1 \wedge$
cantidadDeFilasEstadoJuego(estadoDeJuego) $\leq 26 \wedge$
cantidadDeColumnasEstadoJuego(estadoDeJuego) $> 0 \wedge$
$|\text{estadoDeJuego}_2|| = 1 \wedge$
|barcosDisponibles(estadoDeJuego)| $> 0 \wedge$
tableroVálidoEnJuego(tableroDeJugador(estadoDeJuego, UNO), estadoDeJuego) $\wedge$
tableroVálidoEnJuego(tableroDeJugador(estadoDeJuego, DOS), estadoDeJuego) $\wedge$
coincidenPosicionesAtacadas(tableroDeJugador(estadoDeJuego, UNO), tableroDeJugador(estadoDeJuego, DOS))
}
}

`problema tableroVálidoEnJuego` (in tablero: Tablero, in estadoDeJuego: EstadoJuego) : Bool {
`requiere nada:` { True }
`asegura`: { resultado =
grillaVálidaEnJuego(grillaLocal(tablero), estadoDeJuego) $\wedge$
grillaVálidaEnJuego(grillaOponente(tablero), estadoDeJuego) $\wedge$
coincidenBarcosEnGrilla(barcosDisponibles(estadoDeJuego), grillaLocal(tablero))
}
}

`problema coincidenBarcosEnGrilla` (in barcos: $seq\langle Barco\rangle$, in grilla: Grilla) : Bool {
`requiere laGrillaEsVálida:` { esGrillaVálida(grilla) }
`asegura`: { resultado = mismosElementos(barcos, tamaños(barcosEnGrilla(grilla))) }
}

`problema tamaños` (in barcos: $seq\langle BarcoEnGrilla\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
`requiere losBarcosSonVálidos:` { sonBarcosVálidos(barcos) }
`asegura mismaLongitud`: { |barcos| = |resultado| }
`asegura correspondenciaUnoAUno`: {
Para todo número 'tamaño' de tipo $\mathbb{Z}$ se cumple:
cantidadDeApariciones(tamaño, resultado) = cantidadDeBarcosDeTamaño(barcos, tamaño)
}
}

`problema mismosElementos` (in lista1: $seq\langle T\rangle$, in lista2: $seq\langle T\rangle$) : Bool {
`requiere nada:` { True }
`asegura losElementosDeLaPrimeraAparecenEnLaSegunda`: {
Para todo elemento 'elemento' de tipo T se cumple:
cantidadDeApariciones(elemento, lista1) = cantidadDeApariciones(elemento, lista2)
}
}

`problema coincidenPosicionesAtacadas` (in tablero: Tablero, in tableroOponente: Tablero) : Bool {
`requiere nada:` { True }

`asegura`: { resultado = true $\iff$
para toda posición 'p' válida en $tablero_1$, si dicha posición representa una celda con valor diferente a VACÍO, entonces la posición 'p' en $tableroOponente_0$ representa una celda con el mismo valor que la celda en posición 'p' en $tablero_1$
$\wedge$
para toda posición 'p' válida en $tableroOponente_1$, si dicha posición representa una celda con valor diferente a VACÍO, entonces la posición 'p' en $tablero_0$ representa una celda con el mismo valor que la celda en posición 'p' en $tablero_1$
$\wedge$
Sea 'n1' la cantidad de celdas en $tablero_1$ que tienen valor distinto a VACÍO, y 'n2' la cantidad de celdas en $tableroOponente_1$ que tienen valor distinto a VACÍO, entonces se cumple que $0 \leq n1 - n2 \leq 1$ }
}

Ejemplo:
esEstadoDeJuegoVálido(((4,4),$\langle 2, 2\rangle$,$\langle DOS\rangle$,(⟨
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle BARCO, VACIO, VACIO, VACIO\rangle$,
$\langle BARCO, VACIO, BARCO, VACIO\rangle$,
$\langle VACIO, VACIO, BARCO, VACIO\rangle$
⟩,⟨
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$
⟩),(⟨
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, BARCO, BARCO, BARCO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$
⟩,⟨
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO\rangle$
⟩))) = False (no es válido porque la grilla local del jugador 2 tiene un barco de tamaño 3 en lugar de tener dos barcos de tamaño 2)

Ejercicio 4

`problema disparar EnPosición` (inout estadoDeJuego: EstadoJuego, in posición: Posición) : ResultadoDisparo {
`requiere juegoVálido`: { esEstadoDeJuegoVálido(estadoDeJuego) }
`requiere laPosiciónEsVálidaEnLaGrilla`: {
esPosiciónVálidaEnGrilla(posición, grillaOponente(tableroDeJugador(estadoDeJuego, turno(estadoDeJuego))))
}
`requiere posiciónNoAtacada`: {
celdaEnPosición(grillaOponente(tableroDeJugador(estadoDeJuego, turno(estadoDeJuego))), posición) = VACÍO
}
`modifica`: {estadoDeJuego}
`asegura estadoJuegoSoloActualizaUnaCeldaYElTurno`: {
Todas las componentes de 'estadoDeJuego' permanecen iguales a sus respectivas componentes
en 'estadoDeJuego@pre', excepto por:
- la celda ubicada en la posición 'posición' de la grilla del oponente del jugador 'turno(estadoDeJuego@pre)'que siempre se modifica
- la celda ubicada en la posición 'posición' de la grilla local del jugador atacado que a veces se modifica, y
- el turno, donde 'turno(estadoDeJuego)'$\neq$ 'turno(estadoDeJuego@pre)'
}
`asegura disparoHaceAgua`: {
Sea t2 el turno opuesto a 'turno(estadoDeJuego@pre)', si resultado = NADA, entonces:
celdaEnPosición(grillaLocal(tableroDeJugador(estadoDeJuego@pre, t2)), posición) = VACÍO

∧
celdaEnPosición(grillaLocal(tableroDeJugador(estadoDeJuego, t2)), posición) = AGUA
∧
celdaEnPosición(grillaOponente(tableroDeJugador(estadoDeJuego, turno(estadoDeJuego))), posición) = AGUA
}
asegura disparoAlBarco: {
Sea t2 el turno opuesto a 'turno(estadoDeJuego@pre)', si resultado = TOCADO, entonces:
celdaEnPosición(grillaLocal(tableroDeJugador(estadoDeJuego@pre, t2)), posición) = BARCO
∧
celdaEnPosición(grillaLocal(tableroDeJugador(estadoDeJuego, t2)), posición) = BARCO
∧
celdaEnPosición(grillaOponente(tableroDeJugador(estadoDeJuego, turno(estadoDeJuego))), posición) = BARCO
}
}

Ejemplo:
Sea estado = (
(2,2),⟨2⟩,⟨UNO⟩,(
⟨⟨$BARCO, VACIO$⟩, ⟨$BARCO, VACIO$⟩⟩,
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩
),(
⟨⟨$BARCO, BARCO$⟩, ⟨$VACIO, VACIO$⟩⟩,
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩
)
)
y posición = ("B",1), entonces

dispararEnPosición(estado,posición) = $NADA$

Y el nuevo valor de estado es:
(
(2,2),⟨2⟩,⟨DOS⟩,(
⟨⟨$BARCO, VACIO$⟩, ⟨$BARCO, VACIO$⟩⟩,
⟨⟨$VACIO, VACIO$⟩, ⟨$AGUA, VACIO$⟩⟩
),(
⟨⟨$BARCO, BARCO$⟩, ⟨$AGUA, VACIO$⟩⟩,
⟨⟨$VACIO, VACIO$⟩, ⟨$VACIO, VACIO$⟩⟩
)
)

Ejercicio 5

problema barcosEnGrilla (in grilla: Grilla) : $seq\langle BarcoEnGrilla\rangle$ {
requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
requiere hayUnaÚnicaFormaDeConstruirBarcos: {
Para toda posición 'posición' de tipo Posición que sea válida en la grilla 'grilla' se cumple:
noHayMásDeUnaFormaDeConstruirUnBarcoDesde(grilla, posición)
}
asegura losBarcosEncontradosSonVálidos: { sonBarcosVálidos(resultado) }
asegura lasPosicionesOcupadasPorBarcosEstánOcupadasEnLaGrilla: {
Para todo barco 'barco' de tipo BarcoEnGrilla en 'resultado' se cumple:
sonPosicionesVálidasEnGrilla(barco, grilla)
∧
posicionesOcupadasEnGrilla(grilla, barco)
}
asegura lasPosicionesOcupadasEnLaGrillaEstánOcupadasPorBarcos: {
Para toda posición 'posición' de tipo Posición que sea válida en la grilla 'grilla' y para la cual
el contenido de ella en la grilla 'grilla' sea igual a BARCO se cumple:

```
            algúnBarcoOcupaLaPosición(resultado, posición)
        }
}

problema noHayMásDeUnaFormaDeConstruirUnBarcoDesde (in grilla: Grilla, in posición: Posición) : Bool {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónPerteneceALaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        asegura: { resultado = true ⟺
          ¬sePuedeConstruirBarcoHorizontalDesde(grilla, posición)
          ∨
          ¬sePuedeConstruirBarcoVerticalDesde(grilla, posición)
        }
}

problema sePuedeConstruirBarcoHorizontalDesde (in grilla: Grilla, in posición: Posición) : Bool {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónPerteneceALaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        asegura: { resultado = true ⟺
          celdaEnPosición(grilla, posición) = BARCO
          ∧ (
            hayBarcoAl(grilla, posición, DERECHA)
            ∨
            hayBarcoAl(grilla, posición, IZQUIERDA)
          )
        }
}

problema sePuedeConstruirBarcoVerticalDesde (in grilla: Grilla, in posición: Posición) : Bool {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere laPosiciónPerteneceALaGrilla: { esPosiciónVálidaEnGrilla(posición, grilla) }
        asegura: { resultado = true ⟺
          celdaEnPosición(grilla, posición) = BARCO
          ∧ (
            hayBarcoAl(grilla, posición, ARRIBA)
            ∨
            hayBarcoAl(grilla, posición, ABAJO)
          )
        }
}

problema posicionesOcupadasEnGrilla (in grilla: Grilla, in posiciones: seq⟨Posición⟩) : Bool {
        requiere laGrillaEsVálida: { esGrillaVálida(grilla) }
        requiere lasPosicionesSonVálidasEnLaGrilla: { sonPosicionesVálidasEnGrilla(posiciones, grilla) }
        asegura: { resultado = true ⟺
          Para toda posición 'posición' de tipo Posición en 'posiciones' se cumple:
            celdaEnPosición(grilla, posición) = BARCO
        }
}

problema algúnBarcoOcupaLaPosición (in barcos: seq⟨BarcoEnGrilla⟩, in posición: Posición) : Bool {
        requiere losBarcosSonVálidos: { sonBarcosVálidos(barcos) }
        asegura: { resultado = true ⟺
          Para algún barco 'barco' de tipo BarcoEnGrilla se cumple:
            barco ∈ barcos ∧ posición ∈ barco
        }
}
```

Sugerencias:

1. Para acumular todos los barcos plantear una estrategia que vaya recorriendo las posiciones de la grilla que están ocupadas por barcos y recordar en una variable las posiciones ya visitadas como para asegurarse de pasar una única

vez por cada una. En cada iteración del recorrido agregar al resultado parcial el barco formado por todas las posiciones conectadas a la que se está recorriendo y agregar a la lista de posiciones ya visitadas todas las posiciones que conforman a dicho barco.

2. Para construir el barco completo desde una posición ocupada por una parte del mismo plantear un recorrido sobre las direcciones que por cada dirección agregue al resultado parcial todas las posiciones contiguas a la actual hacia esa dirección que están ocupadas por un barco. **Recordar que los barcos no se pueden ubicar de forma que una posición ocupada por un barco quede adyacente a otra posición ocupada por otro barco así que si hay dos posiciones contiguas marcadas como BARCO entonces ambas pertenecen al mismo barco**.

Ejemplo:

barcosEnGrilla($\langle$
$\langle VACIO, VACIO, VACIO, VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle BARCO, VACIO, VACIO, BARCO, BARCO, BARCO, VACIO\rangle$,
$\langle BARCO, VACIO, VACIO, VACIO, VACIO, VACIO, VACIO\rangle$,
$\langle VACIO, BARCO, BARCO, BARCO, VACIO, BARCO, VACIO\rangle$,
$\langle VACIO, VACIO, VACIO, VACIO, VACIO, BARCO, VACIO\rangle$,
$\rangle$) = $\langle$
$\langle('D', 2), ('D', 3), ('D', 4)\rangle$,
$\langle('B', 6), ('B', 5), ('B', 4)\rangle$,
$\langle('D', 6), ('E', 6)\rangle$,
$\langle('C', 1), ('B', 1)\rangle$
$\rangle$