

## Práctica integradora de Recorridos

**ATENCIÓN:** Se recomienda realizar esta práctica íntegramente en papel, y recurrir a la computadora solamente cuando así lo solicite el enunciado.

Gobs-Man es un clon argentino de un popular juego de arcade de principios de los 80, que está realizado en Gobstones. En los siguientes ejercicios implementaremos diversos procedimientos y funciones que trabajan distintos aspectos del juego (no implementaremos el juego en sí, sino procedimientos y funciones que podrían servir para el mismo). Gobs-Man es una criatura que vive en un tablero del cual desconocemos su ancho y su alto, el cual varía de un nivel a otro. Gobs-Man puede ser programado para moverse de una celda a otra del tablero, para lo cual podemos utilizar las siguientes primitivas:

```
procedure MoverGobsManAl_(dirección)
{-
    PROPÓSITO: Mueve a Gobs-Man a la celda vecina en la dirección dada.
                El cabezal queda sobre Gobs-Man.

    PRECONDICIONES:
        * Existe una celda vecina en la dirección dada.
        * El cabezal se encuentra sobre Gobs-Man.

    PARÁMETROS:
        * dirección: Dirección - Indica hacia dónde se moverá Gobs-Man.
-}
```

```
procedure LlevarGobsManAlBorde_(dirección)
{-
    PROPÓSITO: Mueve a Gobs-Man a la celda en el borde hacia la
                dirección dada. El cabezal queda sobre Gobs-Man.

    PRECONDICIONES:
        * El cabezal se encuentra sobre Gobs-Man.

    PARÁMETROS:
        * dirección: Dirección - Indica hacia dónde se moverá Gobs-Man.
-}
```

A medida que avancemos en los distintos ejercicios iremos introduciendo las reglas puntuales del juego, y nuevas primitivas que podrán utilizarse junto con las anteriormente mencionadas para solucionar el problema.

**ATENCIÓN:** No hay que implementar las primitivas salvo que así lo indique el enunciado, pero puede hacerse uso de ellas para solucionar todos los problemas planteados.

### Ejercicio 1)

Al comenzar un nuevo “nivel” del juego, en cada celda del tablero hay un “coco” (pequeños puntos amarillos) que son el alimento natural de los seres como Gobs-Man. El objetivo de Gobs-Man es precisamente comerse todos los cocos del nivel. Para poder hacer esto, contamos con la siguiente primitiva adicional a las anteriores:

```
procedure ComerCoco()
```



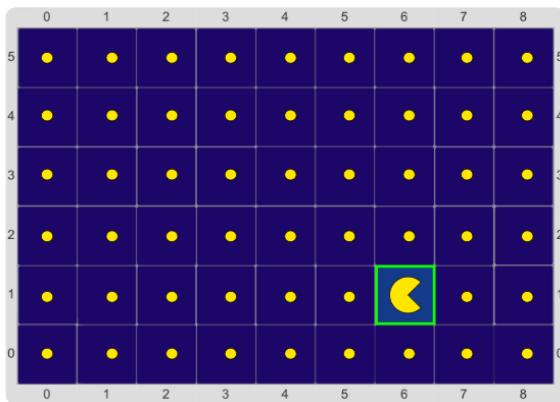
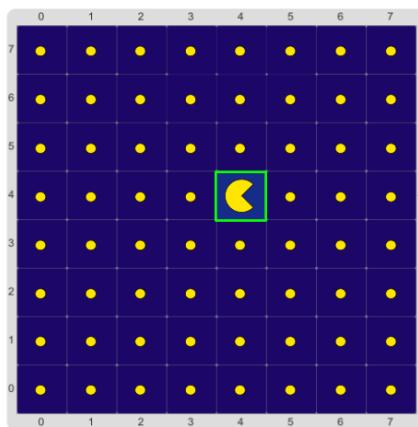
{- PROPÓSITO: Come el coco que haya en la celda actual.

PRECONDICIONES:

- \* Hay un coco en la celda actual.
- \* Gobs-Man está en la celda actual.

-}

Se desea implementar el procedimiento **ComerTodosLosCocosDelNivel**, que hace que Gobs-Man se coma absolutamente todos los cocos del nivel (tablero). Sabemos, como precondición de dicho procedimiento, que hay un coco en cada celda del tablero (incluida en la que inicia Gobs-Man). A continuación hay algunos posibles niveles de Gobs-Man. (Notar que son solo ejemplos, y que la solución tiene que funcionar en cualquiera de estos tableros, e incluso otros que cumplan las mismas características.)



## Ejercicio 2)

Gobs-Man también gusta de comer cerezas. En este caso queremos que Gobs-Man se coma absolutamente todas las cerezas del nivel. Ojo, a diferencia de los cocos, las cerezas no están en todas las celdas, sino que pueden aparecer en algunas celdas sí y en otras no, y nunca sabemos al arrancar un nivel en cuáles celdas estarán las cerezas. Para implementar esto necesitaremos unas nuevas primitivas:

**procedure ComerCereza()**

{- PROPÓSITO: Come la cereza haya en la celda actual.

PRECONDICIONES:

- \* Hay una cereza en la celda actual.
- \* Gobs-Man está en la celda actual.

-}

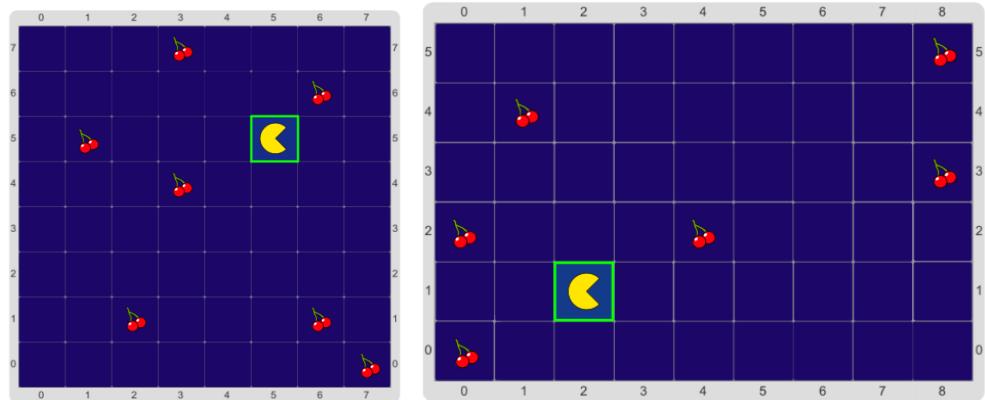
**function hayCereza()**

{- PROPÓSITO: Indica si hay una cereza en la celda actual.

PRECONDICIÓN: Ninguna.

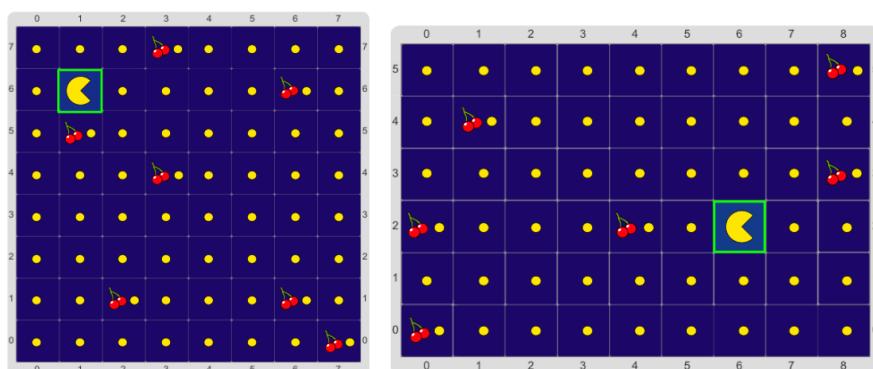
-}

Implementar el procedimiento **ComerTodasLasCerezasDelNivel**, que hace que Gobs-Man se coma todas las cerezas del tablero. Note que los tableros iniciales posibles de este ejercicio no tienen cocos, sino que en cada celda puede haber una cereza, o no haber nada, como muestran los ejemplos de tableros iniciales a continuación:



### Ejercicio 3)

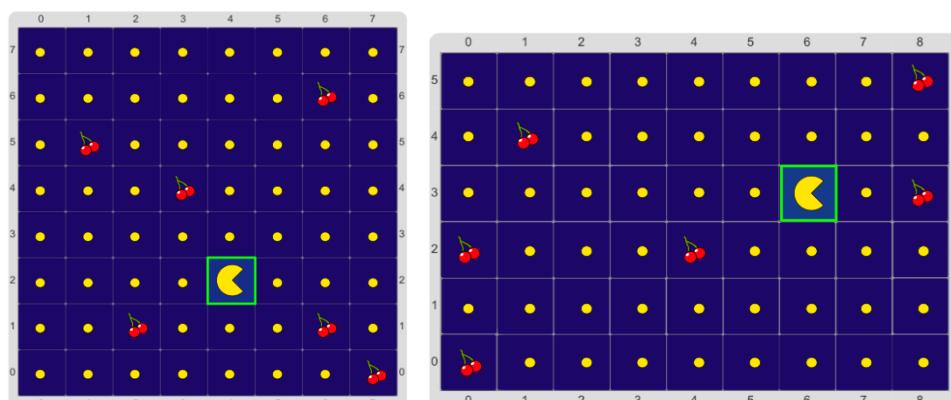
Ahora queremos trabajar sobre niveles que tienen tanto cerezas como cocos. En nuestros tableros iniciales habrá cocos en todas las celdas, y en algunas habrá adicionalmente una cereza. Gobs-Man quiere comerse absolutamente todo lo que encuentre en el nivel, y para indicarle cómo debe escribirse el código de **ComerTodoLoQueSeEncuentreEnElNivel**.



**Pista:** Si la solución requiere más de dos líneas de código, debe plantearse otra estrategia (pensar en qué cosas ya fueron realizadas anteriormente.)

### Ejercicio 4)

El programador del juego ha decidido hacer unos pequeños retoques a cómo inician los niveles. Ahora, en todas las celdas hay cocos, menos en aquellas donde hay una cereza. Es decir, en cada celda puede, o bien haber una cereza, o bien haber un coco (solo estará vacía la celda cuando Gobs-Man se haya comido todo lo de la celda, pero nunca al inicio del nivel). Modificar **ComerTodoLoQueSeEncuentreEnElNivel**, para que tenga en cuenta este cambio. En este caso, la solución no es tan sencilla como en el ejercicio anterior. A continuación, algunos posibles tableros iniciales de este caso:



**ATENCIÓN:** Notar que no se dispone de una primitiva **hayCoco** para solucionar el problema. Si la estrategia está realizada utilizando dicha primitiva, entonces es incorrecta.

**ATENCIÓN:** Si a esta altura las soluciones demandan el planteo de más de un recorrido, probablemente se estén planteando recorridos sobre filas o columnas. Una propuesta alternativa es plantear la solución en términos de un recorrido único sobre todas las celdas del tablero; de no hacerlo, los siguientes ejercicios podrían resultar súmamente complicados.

### Ejercicio 5)

Gobs-Man puede toparse en algún momento con un fantasma. Si lo hace, Gobs-Man sufre un paro cardíaco que la hace morir en la celda en donde vio al espectro. Sabiendo que existen ahora las siguientes primitivas:

```
procedure MorirGobsMan()
{-
    PROPÓSITO: Hace que Gobs-Man muera, dejando su cuerpo
    en la celda actual.

    PRECONDICIONES:
        * El cabezal se encuentra sobre Gobs-Man.

    -}
```

```
function hayFantasma()
{-
    PROPÓSITO: Indica si hay un fantasma en la celda actual.

    PRECONDICIÓN: Ninguna.

    -}
```

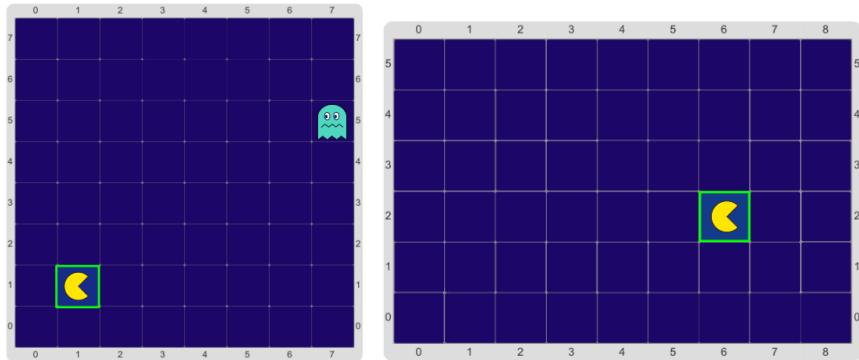
**ATENCIÓN:** Notar que una vez muerto, Gobs-Man no puede moverse. Es decir, los procedimientos que mueven a Gobs-Man tienen ahora una nueva precondición: Gobs-Man está vivo.

Se pide ahora hacer una prueba sobre un nivel vacío (es decir, en las celdas no hay cocos ni cerezas) donde Gobs-Man deberá moverse desde la celda más al Oeste y al Sur, hacia la celda más al Norte y al Este. Se garantiza que en algún lado del tablero habrá un fantasma, y Gob-Man debe morir en la celda en donde encuentre el mismo. Realizar entonces el procedimiento **RecorrerNivelMuriendoEnElFantasma**. Como es costumbre, dejamos algunos tableros iniciales:



### Ejercicio 6)

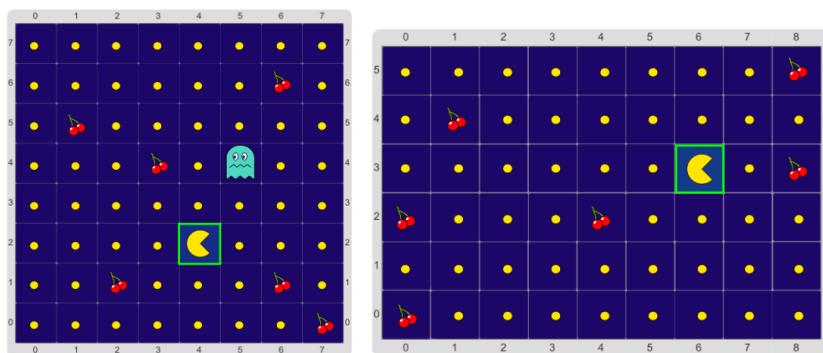
Si bien hemos logrado que Gobs-Man muera en el lugar correcto, también se desea contemplar los niveles en donde tal vez no haya un fantasma. Es decir, ahora queremos volver a recorrer el nivel, pero esta vez no tenemos la certeza de que hay un fantasma en el nivel. Si hay uno, Gobs-Man deberá morir allí, sino, deberá quedar vivo en la última celda del recorrido. Realizar **RecorrerNivelMuriendoSiHayFantasma**, que solucione dicho problema. Los tableros iniciales son idénticos a los anteriores, pero, el fantasma podría no estar, como muestra el segundo tablero de ejemplo:



### Ejercicio 7)

En este ejercicio queremos integrar todas nuestras soluciones al momento. Aunque probablemente no podamos reutilizar el código, sí podremos reutilizar las ideas de lo que venimos trabajando.

En este caso, el nivel comienza con un coco en cada celda, menos en las que hay cerezas, y tal vez, algún fantasma en alguna celda del tablero. Gobs-Man debe comer todos los cocos y cerezas que pueda, partiendo esta vez de la esquina Norte y Oeste, y yendo hacia el Sur y el Este. Al finalizar el nivel, Gobs-Man debe quedar en dicha esquina, si es que no se cruzó con ningún fantasma; si por el contrario el nivel tiene un fantasma, Gobs-Man deberá comer todo lo que tenga en el camino, hasta que se tope con el espectro, donde morirá y terminará el juego. Implementar entonces **JugarNivel** que realice lo mencionado.



### Ejercicio 8)

¿Qué pasa si queremos que Gobs-Man ahora recorra en sentido inverso, es decir, partiendo en la esquina Sur y Este, y yendo hacia el Norte y el Oeste? ¿Las soluciones dadas hasta ahora, permiten cambiar fácilmente ese detalle? Si la respuesta es negativa, pensar alguna forma de lograrlo. Pista, el truco está en pasar información a los procedimientos que realizan el recorrido celda a celda.

### Ejercicio 9)

Queremos realizar el juego, y probar que funcionen nuestras soluciones, pero el diseñador gráfico ha renunciado y no tenemos vestimentas ni primitivas que nos abstraigan de la representación. Por eso, debemos contentarnos con ver bolitas. Por suerte, todas nuestras soluciones anteriores asumen la existencia de ciertos procedimientos y funciones primitivas, por lo que bastará implementar las mismas para tener andando nuestro trabajo previo. Supondremos la siguiente representación:

- Gobs-Man estará representada por una bolita de color Azul si está vivo, y dos, si está muerto.
- Un coco estará representado por una bolita de color Negro.
- Una cereza estará representada por dos bolitas de color Rojo.
- Un fantasma estará representado por cinco bolitas de color Verde.

Se pide entonces implementar cada uno de los procedimientos y funciones primitivas mencionados en esta guía utilizando esta representación, y luego probar en la máquina las soluciones ya hechas en papel.



## Práctica integradora de Funciones Simples y Con Procesamiento, Alternativa de Expresiones y Variables

**ATENCIÓN:** Se recomienda realizar esta práctica íntegramente en papel, y recurrir a la computadora solamente cuando así lo solicite el enunciado.

Ms. Gobs-Man es la secuela del popular juego de video argentino desarrollado en Gobstones, Gobs-Man. Ms. Gobs-Man incorpora una serie de características que se esperan transformen al juego en un éxito inmediato.

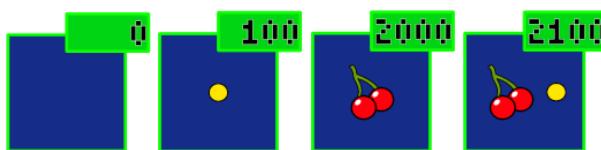
Esta vez el jugador se pondrá en la piel de Ms.Gobs-Man, la versión femenina de nuestro aclamado héroe come cocos. Como en la primer versión del juego, el objetivo será lograr que Ms. Gobs-Man se coma todos los cocos y las cerezas. Para ello contaremos con todas las primitivas que se presentaron en el primer juego, que ahora actúan sobre Ms.Gobs-Man, así como también la siguiente, que se agrega a las anteriores:

```
function hayCoco()  
{  
    PROPOSITO: Indica si hay un coco en la celda actual.  
    PRECONDICIONES: Ninguna.  
}
```

Esta vez, sin embargo, tanto los cocos como las cerezas otorgarán puntaje al jugador.

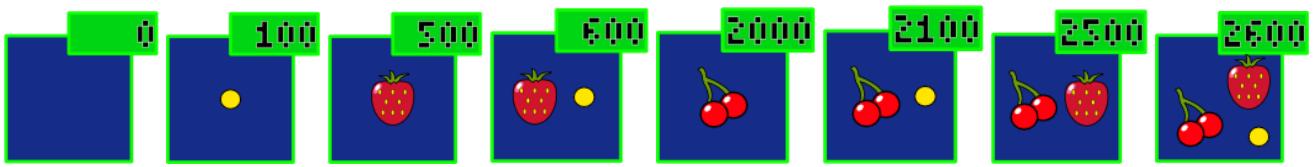
### Ejercicio 9)

Dado que el cabezal se encuentra en alguna celda, se espera poder determinar cuántos puntos obtendrá Ms. Gobs-Man si come todo lo que hay en dicha celda. Notar que la celda puede tener un coco, una cereza, ambos o estar vacía. Un coco otorga a Ms. Gobs-Man 100 puntos, y una cereza otorga 2000 puntos. Escribir la función: **puntajeAObtenerEnCeldaActual** que describe los puntos a obtener en la celda actual. A continuación se muestran algunas posibles celdas a analizar y los puntos que se deberían obtener:



### Ejercicio 10)

Se plantea ahora que además de cerezas y cocos, las celdas pueden contener frutillas (O fresas, si prefiere). Las fresas otorgan 500 puntos solamente, pero pueden encontrarse en cualquier celda, por lo que ahora tenemos las siguientes posibilidades:



Replantear la función `puntajeAObtenerEnCeldaActual` para tener en cuenta dicha situación. Si la estrategia elegida anteriormente fue buena, entonces este cambio no debería redundar en demasiado trabajo. Si por el contrario la solución no fue buena, llevará más esfuerzo (Si fuera este último caso, repensar si se está separando el problema en las subtareas correctas, o ver si se puede mejorar). ¡Ah, por cierto! Casi se nos olvida, también se cuenta con la primitiva siguiente:

```
function hayFrutilla()
{-
  PROPÓSITO: Indica si hay una frutilla en la celda actual.
  PRECONDICIONES: Ninguna.
-}
```

### Ejercicio 10)

Para mostrar esos cuadraditos verdes con los puntos que vimos en los ejemplos anteriores se utilizó una muy útil primitiva que nos fue proporcionada:

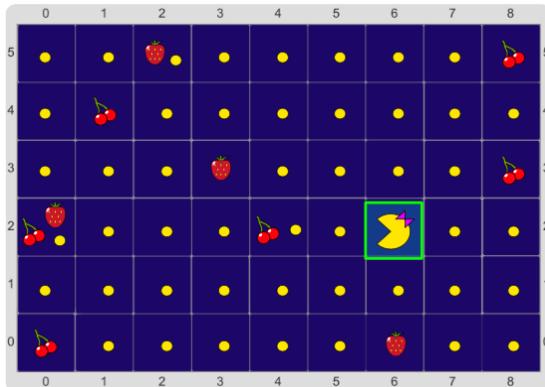
```
procedure MostrarPuntosEnPantalla(cantidadDePuntosAMostrar)
{-
  PROPÓSITO:
    Muestra en la pantalla la cantidad de puntos dados en como argumento.
  PRECONDICIONES: Ninguna.
  PARÁMETROS:
    * cantidadDePuntosAMostrar: Número
      Los puntos a mostrar en la pantalla.
  OBSERVACIÓN: Los puntos se muestran como un número en un recuadro
                verde en la esquina de la celda.
-}
```

Ahora queremos asegurarnos de poder mostrar los puntos correspondientes a lo que Ms. Gobs-Man efectivamente vaya a comer en la celda actual.

Crear el procedimiento `ComerLoQueHayEnLaCeldaYMostrarPuntos` que haga que Ms. Gobs-Man coma todo lo que hay en la celda en donde está parada, y que se muestre en dicha celda los puntos que se obtienen tras comer lo que allí había.

### Ejercicio 11)

Se desea saber cuántos puntos es posible obtener en un nivel determinado. Esto dependerá por supuesto de la cantidad de celdas que haya en dicho nivel, así como de que haya en cada celda (cocos, cerezas, frutillas, combinaciones de estas o nada). Se pide entonces realizar la función `cantidadDePuntosEnElNivel` que indique la cantidad total de puntos que se pueden obtener en el nivel. Por ejemplo, en el siguiente nivel se obtienen 18700 puntos (considerando que en el lugar en donde inicia Ms. Gobs-Man no hay nada). Puede asumirse que el cabezal se encuentra sobre Ms. Gobs-Man.



**Atención:** Para calcular los puntos no es necesario mover a Ms. Gobs-Man, sino solo el cabezal. Sin embargo, si movemos a Ms. Gobs-Man tampoco representará un problema, pues las funciones no tienen efecto, sino que describen valores.

### Ejercicio 12)

Es interesante poder determinar si Ms. Gobs-Man va a morir a causa de cruzarse con un fantasma o no. (Recordemos que en un nivel puede o no haber fantasmas). Se desea entonces la función `hayAlgúnFantasmaEnElNivel` que indica si hay un fantasma en el nivel. Por cierto, se puede asumir que el cabezal se encuentra sobre Ms. Gobs-Man.

**Pista:** Esta función es muy parecida a buscar un fantasma y luego morir, pero en lugar de morir debo indicar si encontré o no el fantasma. Notar que no se necesitan variables para resolver el problema.

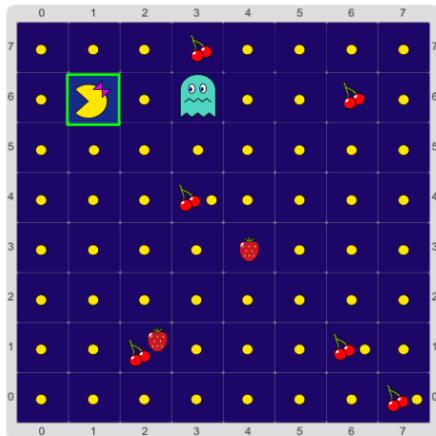
### Ejercicio 13)

Como Ms. Gobs-Man puede cruzarse con un fantasma en el camino, y en ese caso el juego termina en ese momento, los puntos totales que acumula Ms. Gobs-Man en un nivel no siempre son el total de las cosas que hay en el tablero, sino solamente aquellas que “come” hasta que encuentra el fantasma, si es que hubiera uno. En ese sentido, las direcciones hacia las cuales Ms. Gobs-Man realiza un recorrido comiendo lo que encuentra son importantes. Por ejemplo, si parte de la celda Sur-Oeste y se mueve primero al Este y luego al Norte, podría conseguir menos puntos (o más) que si parte de la celda Norte-Este y se mueve al Sur y al Oeste.

Por eso es interesante poder calcular cuantos puntos obtendrá Ms. Gobs-Man hasta toparse con un fantasma (si hubiera uno), si realiza un recorrido en dos direcciones determinadas, dadas por parámetro.

Escribir `cantidadDePuntosEnNivelHacia_Y_`, que dadas dos direcciones, `direcciónPrincipal` y `direcciónSecundaria`, indica cuántos puntos acumularía Ms.Gobs-Man en un recorrido en dicha dirección. Nuevamente, el cabezal arranca sobre Ms. Gobs-Man.

En el ejemplo siguiente, si el recorrido se realiza hacia el Este y el Sur (partiendo de la esquina Norte-Oeste) solo se obtendrán 2900 puntos, mientras que si se realiza hacia el Oeste y el Sur (partiendo de la esquina Norte-Este) se obtendrán 5000. Otras direcciones darán otros puntajes.



#### Ejercicio 14)

Se desea saber cuál de dos opciones de recorridos es más conveniente realizar. Por ejemplo, es mejor recorrer hacia el Norte y el Este, que hacia el Sur y el Este (siempre considerando mejor aquel recorrido en donde se obtienen más puntos). Para determinarlo, se pide escribir la función `esMejorRecorridoHacia_Y_QueHacia_Y_`, que dadas 4 direcciones, `dirPrincipal1`, `dirSecundaria1`, `dirPrincipal2`, `dirSecundaria2`, indica si un recorrido en `dirPrincipal1` y `dirSecundaria1` acumula efectivamente más puntos que un recorrido en `dirPrincipal2` y `dirSecundaria2`.

Si consideramos el ejemplo anterior, la función llamada como `esMejorRecorridoHacia_Y_QueHacia_Y_(Este, Sur, Oeste, Sur)` indica que es falso, pues en el recorrido Este-Sur se acumularían solamente 2900 puntos, mientras que en el Oeste-Sur serían 5000. En cambio, `esMejorRecorridoHacia_Y_QueHacia_Y_(Oeste, Sur, Este, Sur)` indica que es verdadero, por la misma razón.

#### Ejercicio 15)

Queremos también poder determinar si Ms. Gobs-Man ha logrado llegar a un punto en donde está cerca de finalizar el nivel, en particular, si completó más de la mitad del mismo.

Para esto, se pide implementar la función `másDeLaMitadDelNivelHacia_Y_`, que dadas 2 direcciones, `dirPrincipal` y `dirSecundaria`, indique si Ms. Gobs-Man pasó más de la mitad del nivel recorriendo hacia `dirPrincipal` y `dirSecundaria`. Notar nuevamente el cabezal está sobre Ms. Gobs-Man, y también contamos con esta útil primitiva:

```
function tamañoDelTablero()
{-
  PROPÓSITO: Denota el número total de celdas del tablero (nxm)
  PRECONDICIÓN: Ninguna.
-}
```

**Ayuda:** Tener en cuenta que Ms.Gobs-Man viene de las direcciones opuestas a aquellas hacia las cuales está recorriendo, y queremos saber cuántas celdas ya visitó.

#### Ejercicio 16)

El equipo de desarrollo se ha dado cuenta de que utilizar la misma representación en términos de bolitas para Ms. Gobs-Man que para Gobs-Man trae serias complicaciones. Por eso se pensó en una representación alternativa, que permita diferenciar mejor los elementos. Eso sí, algunas primitivas ahora son más complicadas y requieren operadores lógicos más complejos.

- Una bolita negra representa un coco.
- Dos bolitas negras representan una cereza.



- Tres bolitas negras en una celda indican que en la misma hay tanto un coco como una cereza.
- Una bolita roja representa una frutilla.
- Una bolita azul representa a Ms.Gobs-Man; dos, si estuviera muerta.
- Cinco bolitas azules representan un fantasma.
- Los puntos en una celda se representan con bolitas verdes (tantas como puntos).

Se pide cambiar las primitivas anteriormente realizadas en Gobs-Man para reflejar la nueva representación, así como también implementar las primitivas que son exclusivas de Ms. Gobs-Man.

**ATENCIÓN:** Los siguientes puntos son más avanzados, y requieren un buen manejo de las herramientas trabajadas hasta ahora. Además, requieren hacer uso de las funciones **filaActual** y **columnaActual** realizadas en la práctica. Se recomienda completar la práctica antes de continuar con los próximos ejercicios.

### Ejercicio 17)

No todo es diversión al programar a Ms.Gobs-Man, porque también tenemos que programar a los malos. En este caso el cabezal se encuentra sobre un fantasma, y queremos mover al fantasma hacia donde está Ms. Gobs-Man. Para ello, debemos calcular dónde está Ms.Gobs-Man y determinar hacia donde moverse el fantasma. Para ello contamos con las siguientes primitivas:

```
procedure PararCabezalEnMsGobsMan ()  
{- PROPÓSITO: Posiciona el cabezal sobre Ms. Gobs-Man.  
 PRECONDICIÓN: Ms. Gobs-Man está viva en el tablero.  
 -}
```

```
procedure MoverFantasmaAl(dirección)  
{- PROPÓSITO: Mueve al fantasma de la celda actual una celda hacia  
 la dirección dada.  
 PRECONDICIÓN: El cabezal se encuentra sobre un fantasma.  
 PARÁMETRO:  
 * dirección: Dirección - La dirección a la cual mover el fantasma.  
 -}
```

Realizar el procedimiento **MoverFantasmaHaciaMsGobsMan** que mueve el cabezal hacia Ms.Gobs-Man una celda, utilizando el siguiente criterio.

- Si Ms. Gobs-Man se encuentra en una fila y columna distinta a la de Ms.Gobs-Man, mueve el fantasma en diagonal hacia las direcciones en las que se encuentre Ms.Gobs-Man.
- Si Ms.Gobs-Man se encuentra en la misma fila que el fantasma, solo lo mueve una celda sobre la columna actual, en dirección a Ms.Gobs-Man.
- Si Ms.Gobs-Man se encuentra en la misma columna que el fantasma, solo lo mueve una celda sobre la columna actual, en dirección a Ms.Gobs-Man.

Realizar este procedimiento no es fácil, y es conveniente descomponer el problema en tareas mucho más pequeñas y simples. Es por eso que nuestro equipo de analistas ya ha planteado una serie de funciones que pueden ser útiles para solucionar el problema usando una estrategia top-down. A saber, se espera que se utilicen las siguientes funciones (y que sean implementadas también; ¿o alguien puede pensar que se programan solas?) para solucionar el procedimiento anteriormente mencionado:

- **elFantasmaDebeMoverseDeFila**, que indica si el fantasma no se encuentra en la misma fila que Ms. Gobs-Man.
- **elFantasmaDebeMoverseDeColumna**, que indica si el fantasma no se encuentra en la misma columna que Ms. Gobs-Man.



- **direcciónEnFilaAMoverElFantasma**, que dado que el fantasma no está en la misma fila que Ms. Gobs-Man, describe la dirección a la cual el fantasma se debería mover para quedar más cerca que Ms. Gobs-Man (Este u Oeste).
- **direcciónEnColumnaAMoverElFantasma**, que dado que el fantasma no está en la misma columna que Ms. Gobs-Man, describe la dirección a la cual el fantasma se debería mover para quedar más cerca que Ms. Gobs-Man (Norte o Sur).

A su vez, se recomienda realizar las siguientes funciones para solucionar las anteriores:

- **filaDondeEstáElFantasma**, que describe el número de fila donde se encuentra el fantasma.
- **columnaDondeEstáElFantasma**, que describe el número de columna donde se encuentra el fantasma.
- **filaDondeEstáMsGobsMan**, que describe el número de fila donde se encuentra Ms.Gobs-Man.
- **columnaDondeEstáMsGobsMan**, que describe el número de columna donde se encuentra Ms. Gobs-Man.

### Ejercicio 18)

Se desea contar con la función **elFantasmaSeComeráAMsGobsManAContinuación** , que indique si, tras mover el fantasma una única vez más, este alcanzará a Ms. Gobs-Man. Puede asumirse que el cabezal está sobre el fantasma.

## PANDEMIA:

Pandemia es el nombre de un juego de mesa en el que varios jugadores intentan evitar que una plaga mortal se esparza por todo el planeta. En nuestra versión del juego, que será implementada en Gobstones, cada celda del tablero representa una ciudad, que puede estar infectada o no (dada por el nivel de virulencia) y el tablero representa el mundo del juego

Tres enfermedades distintas compiten para intentar ser la pandemia global que destruya a la humanidad, estas son: **Dengue, Coronavirus y Sarampión**. El juego es colaborativo, así que en cada turno los jugadores llevan adelante acciones para combatir las enfermedades. El juego se gana al erradicar todas las enfermedades, y se pierde si alguna enfermedad se transforma en una pandemia, porque en ese caso habría que entrar en cuarentena.

Se cuenta con las siguientes operaciones primitivas:

- **sarampión, coronavirus, dengue**  
representan cada una de las enfermedades posibles.  
El resultado es un Color que representa la enfermedad correspondiente
- **hayVirusDe\_**  
que dado un parámetro con la enfermedad a controlar, indica si la ciudad actual está infectada por ella.
- **virulenciaDe\_**  
que dado un parámetro con la enfermedad a controlar, describe el nivel de virulencia de la misma en la ciudad actual como un número.  
Si la ciudad no está infectada con la enfermedad a controlar, describe 0.
- **enfermedadEnCiudadActual**  
describe la enfermedad que se encuentra en la ciudad actual.  
Como precondición, la ciudad debe estar infectada con alguna enfermedad.
- **IleguéALaÚltimaCiudad**  
indica si el cabezal se encuentra sobre la última ciudad del mapa.
- **AumentarEnfermedad\_En\_**  
que dado un parámetro con una enfermedad y un número, aumenta en la ciudad actual la virulencia de la enfermedad dada, tantas unidades como indique el número dado.  
Como precondición, la ciudad actual no debe estar infectada con otra enfermedad diferente.
- **BajarVirulenciaDe\_**  
que dado un parámetro con una enfermedad, disminuye en la ciudad actual en uno la virulencia de la enfermedad.  
Como precondición, la ciudad actual debe estar infectada por la enfermedad dada.
- **IrAPrimeraCiudad**  
posiciona el cabezal sobre la primera ciudad del mapa.

- **IrAPróximaCiudad**

posiciona el cabezal sobre la ciudad del mapa siguiente a la actual. Como precondición, debe existir una ciudad próxima.

1) Implementar la función **esCiudadSana**.

**Propósito:** Indicar cuando la ciudad actual no está infectada por ninguna enfermedad.

**Precondición:** Ninguna.

**Resultado:** ...

2) Implementar la el procedimiento **IrAPrimeraCiudadCon\_**.

**Propósito:** Posiciona el cabezal en la primer cuidad que esté infectada con la enfermedad dada

**Precondición:** Existe al menos una ciudad infectada con la enfermedad dada.

**Parámetros:** ...

3) Implementar la función **virulenciaLindanteTotalDe\_**.

**Propósito:** Describe la cantidad total de virulencia de la enfermedad dada que acecha a la ciudad actual, teniendo en cuenta las ciudades lindantes que existan en las direcciones ortogonales.

**Precondición:** Ninguna.

**Parámetros:** ...

**Resultado:** ...

# Introducción a la Programación

## Dominio de webvaluación 1

### 2020s2

#### Gobstong Us

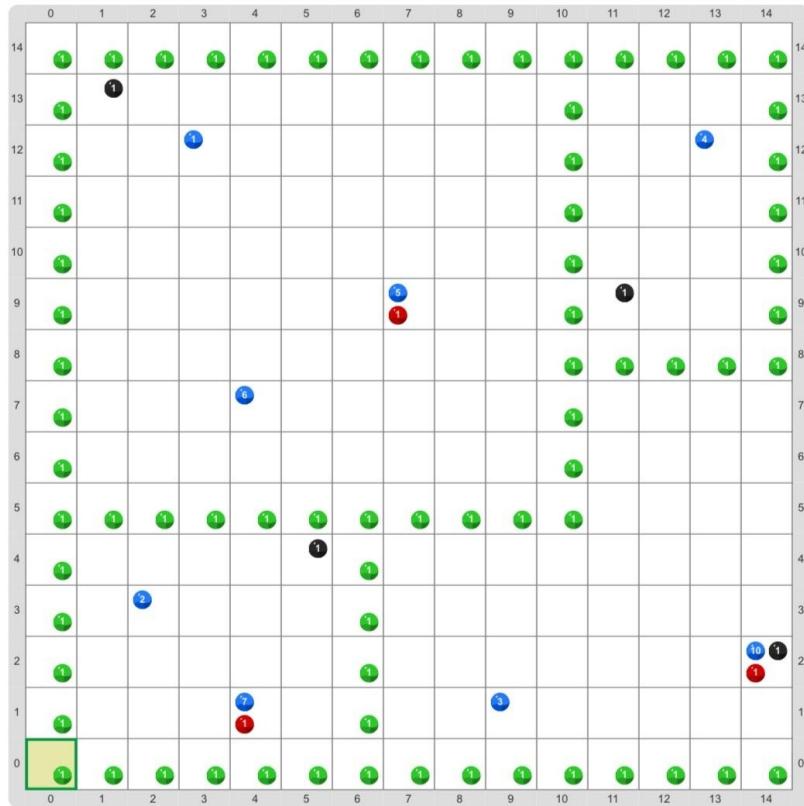
Gobstong Us es un videojuego multijugador en línea en donde los **Jugadores** toman el rol de **Tripulantes** de una nave que debe ser reparada o de **Impostores** que intentan engañar a los tripulantes haciéndose pasar por éstos, pero saboteando la nave y matando a los tripulantes en lugar de repararla. Así, el juego se transforma en una carrera entre tripulantes e impostores, donde ganarán los tripulantes si logran reparar la nave o descubrir quienes son los impostores antes de que los impostores logren matar a todos los tripulantes o sabotear completamente la nave. El juego se desarrolla en una nave que consiste en una serie de **habitaciones** y que pueden estar ubicadas de diversas formas según el nivel del cual se trate. Los impostores, a diferencia de los tripulantes, pueden moverse por las **rejillas de ventilación**, acortando el camino y permitiéndoles matar a más tripulantes rápidamente. Así, hay pistas que pueden indicar si un jugador cualquiera es un impostor o no. Por ej. si un jugador está sobre una rejilla de ventilación, es probable que sea un impostor, por lo que los otros jugadores tienden a identificarlo y expulsarlo de la nave antes de que mate a nadie o sabotee la nave. También puede pasar que quien se encuentra sobre una rejilla no sea un impostor, y entonces los jugadores lo expulsan injustamente de la nave.

Otra cuestión a tener en cuenta es que los impostores no matan a nadie frente a testigos, para evitar que los identifiquen. Así, sí en una habitación quedan solos dos jugadores y uno de ellos es un tripulante y otro un impostor, el impostor matará indefectiblemente al tripulante (para lo cual debe pararse primero a su lado). Así, otra forma de identificar personajes sospechosos es cuando estos se pegan entre sí. Dos personajes muy juntos puede entonces ser señal de que uno intenta matar al otro.

En la implementación actual el tablero representa la totalidad de la nave, y una serie de celdas con una bolita `colorPared()` marcan las paredes dividiendo la misma en sus habitaciones (*por simplicidad no hay puertas que comuniquen a las habitaciones entre sí en esta implementación*). Cada habitación posee **zonas**, dadas por aquellas celdas del tablero que la conforman. En una zona de una habitación puede haber un jugador (como máximo), el cual se identifica con bolitas de `colorJugador()` tantas bolitas como el número de identificador de ese jugador (no hay dos jugadores en todo el tablero con un mismo identificador, los identificadores son siempre mayores que cero). Aquellos jugadores que son impostores tienen además una bolita de `colorImpostor()`, que identifica tal característica; aquellos que no la tienen, son tripulantes (observar que por sí sola una bolita de este color NO alcanza para marcar un impostor; debe ser un complemento a la representación del jugador). Una única

bolita de color **colorRejilla()** en una celda indica una rejilla de ventilación. Es importante entender que el tablero representa la totalidad de la información, y no lo que vería cada jugador en particular.

A continuación hay un tablero de ejemplo que muestra algunos de los posibles niveles del juego y algunos jugadores en él (donde **colorImpostor()** es Rojo, **colorPared()** es Verde, **colorJugador()** es Azul y **colorRejilla()** es Negro)



En esta webvaluación lo que nos interesará es obtener información o modelar pequeños aspectos del juego, y no el juego en su totalidad. Para que mirar los elementos de una habitación sea sencillo, se brindan las siguientes primitivas:

```
// Para recorrer una habitación
IrAlPrincipioDeLaHabitaciónAl_Y_(dirPrincipal, dirSecundaria)
hayPróximaZonaEnLaHabitaciónAl_Y_(dirPrincipal, dirSecundaria)
IrAPróximaZonaEnLaHabitaciónAl_Y_(dirPrincipal, dirSecundaria)

// Para recorrer las diferentes habitaciones
IrAHabitaciónNúmero_(númeroDeHabitación)
cantidadDeHabitacionesEnLaNave()
```

Consignas 1 Webevaluacion:

- 1) Escribir la función `hayTripulanteAcá` que indica si en la zona actual hay un tripulante (recordar que los impostores son jugadores, pero no tripulantes...).
- 2) Escribir la función `cantidadDeTripulantesEnLaNave` que describe la cantidad total de tripulantes que hay en la nave.
- 3)Escribir la función `habráUnaExpulsiónInjusta` que indica si los jugadores van a determinar expulsar a alguien de la nave injustamente. Esto ocurre cuando en cualquier lugar de la nave un tripulante está sobre una rejilla de ventilación.
- 4) Escribir el procedimiento `AcercarseParaMatar` que hace que el impostor se mueva una posición en forma ortogonal (a una de las zonas lindantes) con la intención de acercarse al impostor que está cerca, para matarlo posteriormente. Para esto se presupone que en la zona actual hay un impostor y que en alguna de las celdas en diagonal a la actual (hacia cualquiera de las 4 diagonales) hay un tripulante, y que estos jugadores son los únicos dos en la habitación. Así, por ej. si el tripulante está en la diagonal Norte-Este, entonces el impostor debe moverse a la celda al Norte o al Este.
- 5) Se quiere saber si en algún lado hay más de un impostor agazapado esperando una víctima, es decir, si en una habitación hay 2 o más impostores y ningún tripulante. Se pide, entonces implementar la función `hayImpostoresAgazapadosEnAlgunaHabitación` que indica lo mencionado.

## Práctica integradora (1er parcial)

- Si bien el presente no es un examen, puede ser beneficioso intentar realizarlo dentro de un marco de tiempo equivalente al que tendrá en el mismo, en un ambiente sin interrupciones y de forma individual en papel.
- También recomendamos aspirar a escribir de forma prolífica, o pasar el contenido de hojas borradores a hojas finales para poder estimar correctamente durante el proceso los tiempos que esta tarea demandaría durante el examen.
- Los procedimientos procedimientos y funciones primitivos que son dados en el examen deberán ser implementados en caso de que se desee probar la solución en máquina, por lo que recomendamos que aspire a realizar esto solamente una vez terminado de realizar los ejercicios en papel.
- Aconsejamos leer el enunciado completo antes de empezar a resolver los ejercicios, ya que ayuda a comprender mejor el dominio.
- Recuerde que el parcial será a libro abierto, que puede consultar cualquier material (en papel) que haya sido escrito antes de comenzar el examen y usar sin definir todas las funciones y procedimientos vistos durante la cursada. Tenga esto en cuenta al desarrollar su solución.
- Pensar bien la estrategia a seguir, expresando la misma mediante procedimientos y funciones. Recordar escribir primero los contratos.

### Convención de Motoqueros

Convención de Motoqueros es una simulación del comportamiento de motoqueros a lo largo de una convención. En las convenciones, a los motoqueros les gusta interactuar con otros motoqueros y asociarse a clubes.

Convención de Motoqueros se puede modelar en Gobstones. Cada motoquero puede representarse con una celda del tablero. Un motoquero es vecino de otro si sus correspondientes celdas son lindantes en cualquiera de las 4 direcciones ortogonales. La imagen de la izquierda de la Figura 1 muestra una convención posible.

Durante una convención, se arman clubes. Es decir, grupos de motoqueros activos que están conectados entre sí. Dos motoqueros se dicen conectados cuando hay una secuencia de motoqueros activos vecinos que los unen. Podemos además asignarles números a cada club, para diferenciarlos unos de otros y determinar cosas como qué grupo tiene más

miembros, o cuál tiene menos. La imagen de la derecha de la Figura 1 muestra diversos grupos en la convención anterior, habiendo asignado un número a cada grupo.

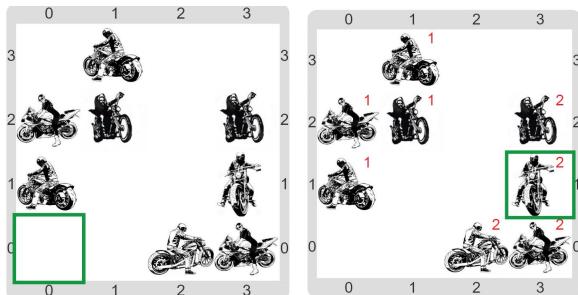


Figura 1: A la izquierda, una convención de motoqueros, con 13 motoqueros activos en total. A la derecha, la misma convención con los clubes marcados. Hay 2 clubes distintos en la convención, siendo ambos clubes de 4 miembros.

Se cuenta con los procedimientos y funciones primitivas que se enuncian a continuación:

- **elMotoqueroPerteneceAUnClub ()**  
PROPSITO: Indica si el motoquero actual pertenece a algún club.  
PRECONDICIÓN: El motoquero actual se encuentra activo.  
RESULTADO: Un valor de verdad.
- **clubAlQuePerteneceUnMotoquero ()**  
PROPSITO: Describe el número del club al cual pertenece el motoquero actual.  
PRECONDICIÓN: El motoquero actual se encuentra activo y pertenece a un club.  
RESULTADO: Un número de club válido.
- **IncluirEnElClubNúmero\_(número)**  
PROPSITO: Incluye al motoquero actual como perteneciente al club con el número dado.  
PRECONDICIÓN: El motoquero actual se encuentra activo y no pertenece todavía a ningún club.
- **elMotoqueroEstáActivo ()**  
PROPSITO: Indica si el motoquero actual está activo o no.  
PRECONDICIÓN: Ninguna.
- RESULTADO: Un valor de verdad.



## EJERCICIOS

**Ejercicio 1)** Realizar la función `cantidadDeClubesEnLaConvención()` que determine la cantidad total de clubes de motoqueros que hay en la convención.

Para ello se propone una estrategia *top-down* para contar los clubes: asignar un número de club distinto y consecutivo a cada club, hasta no tener más clubes. El último número de club asignado debería permitir determinar la cantidad de clubes. Para esto se propone realizar las siguientes funciones y procedimientos (aunque pueden requerirse o ser útil algunos adicionales):

**Ejercicio a)** La función `hayMotoquerosActivosSinClub()` que indica si en la convención hay algún motoquero activo que todavía no tenga un club.

**Ejercicio b)** El procedimiento `IrAMotoqueroActivoSinClub()` que asumiendo que hay un motoquero activo sin club, deja el cabezal en el mismo.

**Ejercicio c)** El procedimiento `AsignarClub_AMotoquerosConectados(númeroDeClub)` que marca al motoquero actual y a todos los motoqueros conectados a él como pertenecientes al club `númeroDeClub`. Dichos motoqueros no deben tener un club previamente asignado. Este problema es a su vez complejo, y se propone la siguiente estrategia para solucionarlo.

- Paso 1, marcar al motoquero actual como perteneciente al club, y también a todos sus vecinos activos que no tengan club asignado.
- Paso 2, buscar en la convención algún motoquero activo sin club que tenga como vecino un motoquero del club `númeroDeClub` e ir hasta el mismo.
- Repetir sucesivamente los pasos 1 y 2 hasta que no haya más motoqueros activos en la convención que sean vecinos de otros con el club `númeroDeClub`.

**Ejercicio d)** El procedimiento `AsignarClub_AVecinos(númeroDeClub)` que marca todos los motoqueros vecinos al motoquero actual que estén activos y sin club como pertenecientes al club número `númeroDeClub`.

**Ejercicio e)** La función `hayMotoqueroActivoYSinClubConVecinoEn_(númeroDeClub)` que indica si en la convención hay algún motoquero activo sin club todavía que tenga de vecino otro motoquero activo que tenga por club `númeroDeClub`.

**Ejercicio f)** `IrHastaElMotoqueroActivoYSinClubConVecinoEn_(númeroDeClub)` que posiciona el cabezal sobre un motoquero activo que no tenga club todavía y que tenga por vecino un motoquero en el club `númeroDeClub`.

**Ejercicio 2)** Escribir la función `clubConMásMotoqueros()` que determina cuál es el club que tiene más motoqueros activos en la convención. Se propone como estrategia marcar primero los clubes, y luego contabilizar la cantidad de miembros de cada club, para luego saber cuál tiene más. De esto surgen los siguientes procedimientos y funciones (nuevamente, realizar adicionales puede resultar útil):

**Ejercicio a)** El procedimiento `AsignarClubATodosLosMotoquerosActivos()` que asigna un club a cada motoquero activo de la convención.

**Ejercicio b)** El procedimiento `IrAAAlgúnMotoqueroDelClub_(númeroDeClub)` que posiciona el cabezal en algún motoquero activo miembro del club `númeroDeClub`.

**Ejercicio c)** La función `cantidadDeMiembrosDelClubActual()` que determina la cantidad de miembros que posee el club del cual el motoquero actual es miembro. El cabezal debe encontrarse sobre un motoquero que tenga un club asignado.

AYUDA: estructurarla como un recorrido sobre todos los motoqueros.



## Preparación para el Primer Parcial

- Si bien el presente no es un examen, puede ser beneficioso intentar realizarlo dentro de un marco de tiempo equivalente al que tendrá en el examen, en un ambiente sin interrupciones y de forma individual en papel.
- También recomendamos aspirar a escribir de forma prolífica, o pasar el contenido de hojas borradores a hojas finales para poder estimar correctamente durante el los tiempos que esta tarea demanda durante el examen.
- Los procedimientos procedimientos y funciones primitivos que son dados en el examen deberán ser implementados en caso de que se desee probar la solución en máquina, por lo que recomendamos que aspire a realizar esto solamente una vez terminado de realizar los ejercicios en papel.
- Aconsejamos leer el enunciado completo antes de empezar a resolver los ejercicios, ya que ayuda a comprender mejor el dominio.
- Recuerde que el parcial será a libro abierto, que puede consultar cualquier material (en papel) que haya sido escrito antes de comenzar el examen y usar sin definir todas las funciones y procedimientos vistos durante la cursada. Tenga esto en cuenta al desarrollar su solución.
- Pensar bien la estrategia a seguir, expresando la misma mediante procedimientos y funciones. Recordar escribir primero los contratos.

## Convención de Motoqueros

La Convención de Motoqueros es un juego que consiste en simular el comportamiento de motoqueros en una convención. Un motoquero puede estar activo o inactivo. A los motoqueros les gusta interactuar con otros motoqueros. Como producto de esta interacción algunos motoqueros pasan a la inactividad (duermen) y otros se hacen activos (despiertan).

La interacción se produce entre motoqueros activos que se mueven cerca unos de otros (decimos que son vecinos en ese caso). La interacción entre motoqueros activos sigue las siguientes pautas (de comportamiento):

- a. Todo motoquero activo con menos de dos vecinos activos se inactiva (no puede lucirse).
- b. Todo motoquero activo con más de tres vecinos activos pasa a la inactividad (le es imposible destacarse).
- c. Todo motoquero activo con exactamente dos o tres vecinos activos continúa activo.
- d. Todo motoquero inactivo con exactamente tres vecinos activos se convierte en activo (el grupo lo incentiva).

Estas pautas se aplican simultáneamente sobre todas los motoqueros de la convención. Cada una de estas aplicaciones simultáneas de las pautas se

conoce como una fase. Cada fase produce un nuevo estado de la convención.

La Convención de Motoqueros se puede modelar en Gobstones. Cada motoquero puede representarse con una celda del tablero. La misma tiene una bolita verde si el motoquero que representa está activo y ninguna verde si está inactivo.

Una motoquero es vecino de otro si sus correspondientes celdas son lindantes al N, NE, E, SE, S, SO, O u NO. La figura 1 muestra una convención posible y exhibe cómo evoluciona en una fase.

Además, durante la convención, se arman clubes. Es decir, grupos de motoqueros activos que están conectados entre sí. Dos motoqueros se dicen conectados cuando hay una secuencia de motoqueros activos vecinos que los unen.

Podemos además asignarles números a cada club, para diferenciarlos unos de otros y determinar cosas como qué grupo tiene más miembros, o cuál tiene menos.

La figura 2 muestra diversos grupos en una convención, habiendo asignado un número a cada grupo.

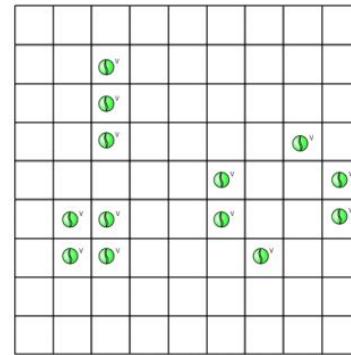
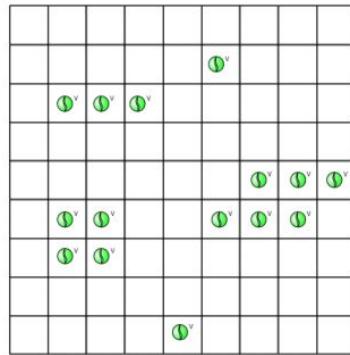


Figura 1: A la izquierdo se ve un posible estado inicial de una convención de motoqueros.  
A la derecha se ve la misma convención luego de la primer fase.

Para poder procesar la convención de motoqueros en Gobstones vamos a requerir de una serie de procedimientos y funciones primitivas que se enuncian a continuación:

#### **fueIncentivado()**

PROPÓSITO: Indica Verdadero si el motoquero de la celda actual fué incentivado en esta fase a partir de interactuar con otros motoqueros. Es decir, en la próxima fase pasará a estar activo.

PRECONDICIÓN: Ninguna.

#### **seAburrió(palo)**

PROPÓSITO: Indica Verdadero cuando el motoquero actual se aburrió en esta fase, ya sea porque no pudo lucirse o porque le fue imposible destacarse, y por tanto pasará a estar inactivo en la próxima fase.

PRECONDICIÓN: Ninguna.

#### **IncentivarAlMotoquero()**

PROPÓSITO: Incentiva al motoquero actual, de forma que en la próxima fase deberá estar activo.

PRECONDICIÓN: El motoquero actual no fue incentivado previamente en esta fase ni tampoco se aburrió.

#### **AburrirAlMotoquero()**

PROPÓSITO: Aburre al motoquero actual, de forma que en la próxima fase deberá estar inactivo.

PRECONDICIÓN: El motoquero actual no fue aburrido previamente en esta fase ni fue incentivado.

#### **ActualizarLaConvención()**

PROPÓSITO: Actualiza la convención de motoqueros, pasando a una próxima fase. Para ello, este procedimiento transforma a todos los motoqueros aburridos en motoqueros inactivos, a todos los motoqueros incentivados en motoqueros activos, y deja igual a aquellos motoqueros que no están ni aburridos ni incentivados. Al terminar, el humor de los motoqueros de la convención es neutro (es decir, no hay ningún motoquero incentivado o aburrido en la convención).

PRECONDICIÓN: Ninguna.

#### **MarcarMotoqueroComoPertenecienteAlClub\_(número)**

PROPÓSITO: Marca al motoquero actual como perteneciente al club número “número”.

PRECONDICIÓN: El motoquero actual se encuentra activo y no pertenece todavía a ningún club.

#### **elMotoqueroPerteneceAUnClub()**

PROPÓSITO: Describe Verdadero si el motoquero actual pertenece a algún club. Falso en caso contrario.

PRECONDICIÓN: El motoquero actual se encuentra activo.

#### **clubAlQuePerteneceUnMotoquero()**

PROPÓSITO: Describe el número al cual pertenece el motoquero actual.

PRECONDICIÓN: El motoquero actual se encuentra activo y pertenece a un club.

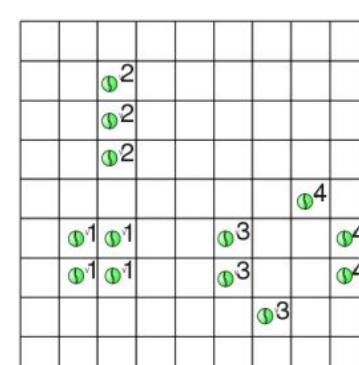


Figura 3. Hay 4 clubes distintos en la convención, siendo el club número 1 aquel que tiene más miembros (4 miembros)



## EJERCICIOS

### Ejercicio 1)

Escribir la función `elMotoqueroEstáActivo()` que retorna Verdadero si el motoquero sobre el cuál se encuentra el cabezal está activo y Falso en caso de que esté inactivo.

### Ejercicio 2)

Escribir la función `cantidadDeVecinosActivos()` que describe la cantidad de vecinos activos que tiene el motoquero sobre cuál se encuentra el cabezal.

### Ejercicio 3)

Escribir el procedimiento `ProcesarMotoquero()` que aplique las pautas de la fase al motoquero actual. Note que no se pueden aplicar las pautas de la fase de forma directa sobre un único motoquero, pues inactivarlo o activarlo en esta fase haría que, al procesar el motoquero vecino, este tenga menos o más vecinos activos que los que debería. Por esto, debemos simplemente aburrir al motoquero (si se va a inactivar en la próxima fase) o incentivarlo (si se va a activar la próxima fase).

### Ejercicio 4)

Escribir el procedimiento `Simular_Fases(cantidadDeFases)` que dado un número positivo `cantidadDeFases` simúle la interacción de la convención de motoqueros durante tantas fases como indique dicho número.

### Ejercicio 5)

Se desea realizar el procedimiento `cantidadDeClubEnLaConvención()` que determine la cantidad total de clubes de motoqueros que hay en la convención. Para ello se ha planteado una estrategia top-down que propone que para contar los clubes, lo más fácil es asignar un número de club distinto y consecutivo a cada club, hasta no tener más clubes. El último número de club asignado debería permitir determinar la cantidad de clubes. Para esto se propone realizar las siguientes funciones y procedimientos (aunque pueden requerirse o ser útil algunos adicionales):

**Ejercicio a)** La función `hayMotoquerosActivosSinClub()` que describe Verdadero si en algún lugar de la convención hay algún motoquero activo que no tenga todavía un club.

**Ejercicio b)** El procedimiento `IrAMotoqueroActivoSinClub()` que asumiendo que hay un motoquero activo sin club, deja el cabezal en el mismo.

**Ejercicio c)** El procedimiento `AsignarClub_AMotoquerosConectados(númeroDeClub)` que marca todos los motoqueros conectados al motoquero actual, y el actual, como pertenecientes al club con número “númeroDeClub”. Dichos motoqueros no deben tener un club previamente asignado. Este problema es a su vez complejo, y se propone la siguiente estrategia para solucionarlo. Paso 1, marcar al motoquero actual como perteneciente al club, y a sus vecinos activos. Paso 2, buscar en la convención algún motoquero activo sin club que tenga como vecino un motoquero del club “númeroDeClub” e ir hasta el mismo, repetir sucesivamente 1 y 2 hasta que no hayan más motoqueros activos en la convención que sean vecinos de otros con el club “númeroDeClub”.

**Ejercicio d)** El procedimiento `AsignarClub_AVecinos(númeroDeClub)` que marca todos los motoqueros vecinos al motoquero actual como pertenecientes al club número “númeroDeClub”.



**Ejercicio e)** La función **hayMotoqueroActivoConVecinoEn\_(númeroDeClub)** que retorna Verdadero si en la convención hay algún motoquero activo sin club todavía que tenga de vecino otro motoquero activo que tenga por club “númeroDeClub”.

**Ejercicio f)** El procedimiento **IrHastaElMotoqueroActivoConVecinoEn\_(númeroDeClub)** que posiciona el cabezal sobre un motoquero activo que no tenga club todavía y que tenga por vecino un motoquero en el club “númeroDeClub”

### Ejercicio 6)

Escribir el procedimiento **ClubConMásMotoqueros()** que determina cuál es el club que tiene más motoqueros activos en la convención. Se propone como estrategia marcar primero los clubes, y luego contabilizar la cantidad de miembros de cada club, para luego saber cuál tiene más. De esto surgen los siguientes procedimientos y funciones (Nuevamente, realizar adicionales puede resultar útil):

**Ejercicio a)** El procedimiento **AsignarClubATodosLosMotoquerosActivos()** que asigna un club a cada motoquero activo de la convención.

**Ejercicio b)** El procedimiento **IrAAAlgúnMotoqueroDelClub(númeroDeClub)** que posiciona el cabezal en algún motoquero activo miembro del club “númeroDeClub”.

**Ejercicio c)** La función **cantidadDeMiembrosDelClubActual()** que determina la cantidad de miembros que posee el club del cual el motoquero actual es miembro. El cabezal debe encontrarse sobre un motoquero que tenga un club asignado.



## Repasso Integrador

### Gimnasios (GobsGym)

GobsGym es una plataforma simple para registrar los accesos de los deportistas a los diferentes aparatos de un gimnasio y conseguir estadísticas de las rutinas efectuadas. Para modelar la plataforma, se introducen las estructuras de datos que se describen a continuación. Existen 3 tipos de Ejercicios que se ofrecen en la plataforma y se representan con el siguiente tipo de datos:

```
type TipoDeEjercicio is variant {
    case Musculación{}
    case Cardio{}
    case Fitness {}
}
```

De cada máquina deportiva conocemos su nombre, el tipo de ejercicio, el precio por uso del mismo, y una indicación sobre si el equipo se puede usar o está siendo reparada, por lo que no se permite su uso. Además tenemos un tipo para los Tickets de uso de las máquinas

```
type Máquina is record {
    /*
        PROPÓSITO: modela un máquina de GobsGym
        INV.REP.:
            * "nombre" no puede estar vacío
            * "precio" es >= 0
    */
    field nombre      // String
    field tipo        // TipoDeEjercicio
    field enReparación // Booleano
    field precio      // Número
}

type TicketDeUso is record {
    /*
        PROPÓSITO: modela el uso de una máquina por un deportista
        INV.REP.:
            * "númeroDeTarjeta" es un número >= 0
            * "nombreDeMáquina" no puede estar vacío
    */
    field númeroDeTarjeta // Número
    field nombreDeMáquina // String
}
```

El sistema se representa por una lista de las máquinas que están disponibles y la lista de los usos realizados por los deportistas, mediante el siguiente tipo:

```
type GobsGym is record {
    /*
        PROPÓSITO: modela una plataforma de registro de Las
                    máquinas usadas en el gimnasio
        INV.REP.: "máquinas" no contiene dos máquinas con el mismo nombre
    */
    field usosEfectuados // [TicketDeUso]
    field máquinas       // [Máquina]
```



}

En los ejercicios que siguen, programaremos algunos aspectos del sistema utilizando el lenguaje Gobstones.

## EJERCICIOS

### Ejercicio 1)

Definir la función `cantidadDeAccesosAl_RegistradosEn_(nombreDeMaquina, gobsGym)` que dado un nombre de máquina y un sistema GobsGym, describe la cantidad de contrataciones que se realizaron para esa máquina.

**Observación:** notar que si el nombre de la máquina no existe, debe describir al número 0.

### Ejercicio 2)

Definir la función `máquinasQueSeUsaronMásDe_RegistradosEn_(cantidad, gobsGym)` que dado un número y un sistema GobsGym retorna la lista de los nombres de las máquinas que registraron más tickets que la cantidad recibida.

### Ejercicio 3)

Definir `gymConAccesosDe_A_En_(númeroDeTarjeta, listaDeNombresDeMáquinas, gobsGym)` que describe el GobsGym resultante de intentar registrar el uso de la tarjeta dada por el número de tarjeta, que se supone existente, en cada uno de las máquinas cuyos nombres se reciben en la listaDeNombresDeMáquinas. Notar que solo se podrán registrar los tickets para máquinas que estén actualmente disponibles, es decir que NO estén en reparación.

### Ejercicio 4)

Definir la función `máquinasContratadasPor_En_(listaDeNrosDeTarjetas, gobsGym)` que describe una lista de listas de Máquinas. El primer elemento es la lista de todas las Máquinas que se usaron con el número de tarjeta que es el primer elemento de la `listaDeNrosDeTarjetas`, el segundo elemento son todos los Máquinas que se usaron con la tarjeta cuyo número es el segundo elemento de la lista `listaDeNrosDeTarjetas`.

Notar que si `listaDeNrosDeTarjetas` tiene `n` elementos, la lista resultante también tendrá `n` elementos, y cada uno será una lista de `Máquinas`.

### Ejercicio 5)

Definir la función `recaudaciónDeMáquinasParaFitnessEn_(gobsGym)` que describe un número que es la cantidad de dinero recaudado por las máquinas de tipo Fitness.

### Ejercicio 6)

Definir la función `hayMáquinaEnReparaciónConGananciaMayorA_En_(ganancia, gobsGym)` que describe si alguna máquina recaudó esa ganancia o un número mayor, y está actualmente siendo reparada.

## Práctica 8

### Interpretación de Enunciados Complejos

Introducción a la Programación  
1<sup>er</sup> Semestre de 2017

**Nota:** la presente práctica se compone por enunciados de parciales de años anteriores y su objetivo es entrenar al alumno en la resolución de enunciados complejos.

#### 1. Juego de la Vida

El *Juego de la Vida* consiste en simular la evolución de ciertas *células* que habitan el universo y que interactúan entre sí a lo largo del tiempo. Una célula puede estar *viva* o *muerta*. Como producto de esta interacción algunas células mueren y otras nuevas nacen. La interacción se produce entre células vivas que son *vecinas*. Una célula se dice vecina de otra si linda con la misma. La interacción entre células vivas sigue las siguientes *pautas de evolución*:

1. Toda célula viva con menos de dos vecinas vivas muere (escasez de población).
2. Toda célula viva con más de tres vecinas vivas muere (sobre población).
3. Toda célula viva con exactamente dos o tres vecinas vivas pasa a la siguiente generación.
4. Toda célula muerta con exactamente tres vecinas vivas se convierte en una célula viva (reproducción).

Estas pautas de juego se aplican *simultáneamente* sobre todas las células del universo. Cada una de estas aplicaciones simultáneas de las pautas de juego se conoce como un *tick*. Cada *tick* produce un nuevo universo.

El *Juego de la Vida* se puede modelar en GOBTONES de la siguiente manera. Cada célula del juego puede representarse con una *celda* del tablero. La misma tiene una bolita verde si la célula que representa está viva y ninguna bolita verde si está muerta. Una célula es vecina de otra si sus correspondientes celdas son lindantes al N, NE, E, SE, S, SO, O u NO. La figura 1(a) muestra un universo posible y 1(b) exhibe cómo evoluciona en un tick.

#### Ejercicio 1

Escribir las siguientes funciones:

1. *celulaViva*, que determine si una célula está viva o no.
2. *nroVecinas*, que determine el número de células vivas que son vecinas de la célula actual.

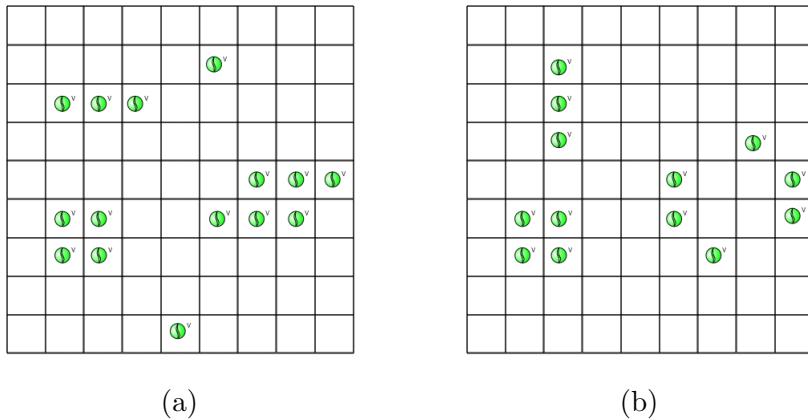


Figura 1: Ejemplo de universo y cómo evoluciona luego de un tick

### Ejercicio 2

Escribir `ProcesarCelula`, un procedimiento que aplique las pautas de juego a la celular actual.

**Nota:** Tener en cuenta que no se puede eliminar o reanimar una célula hasta tanto no se hayan aplicado las pautas de juego a **todas** las células del universo, pues sino las células de la nueva generación se confundirían con aquellas de la etapa actual, haciendo que se produzcan resultados erróneos al procesar las pautas. En consecuencia, en lugar de eliminar o reanimar una célula, la misma debe ser **marcada** como que va a ser eliminada o reanimada (una vez que se complete el procesamiento de las demás células). Para ello, puede asumir que cuenta con procedimientos `MarcarCelulaParaSerEliminada()` y `MarcarCelulaParaSerReanimada()` que marcan una célula para ser eliminada o reanimada más tarde, respectivamente.

### Ejercicio 3

Escribir `ActualizarUniverso`, un procedimiento que lee todas las marcas de eliminación/reanimación de células y las procesa (es decir, efectivamente elimina/reanima esas células). El resultado será un nuevo universo, en condiciones de poder evolucionar en un tick subsiguiente.

**Nota:** Estructurar su solución como un recorrido. Puede asumir la existencia de:

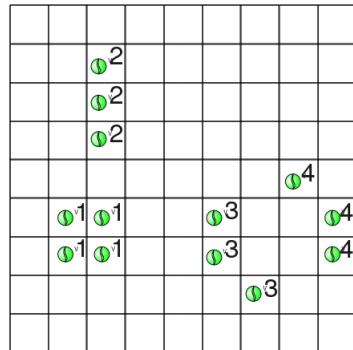
- Funciones `estaMarcadaParaSerEliminada()` y `estaMarcadaParaSerReanimada()`, que retorna un booleano indicando si esa célula está marcada para ser eliminada o reanimada, respectivamente.
- Procedimiento `EliminarCelula()` y `ReanimarCelula()`, que eliminan una célula si había allí una marca para eliminar y reaniman una célula si había allí una marca para reanimar. Además, estas operaciones quitan las marcas.

### Ejercicio 4

Escribir `Simular`, un procedimiento que dado un número entero positivo, simula esa cantidad de ticks sobre el universo.

Habiendo hecho una simulación, es de interés contabilizar la cantidad de *islotes* en un universo, una vez finalizada la simulación. Un islote es un conjunto de células vivas que están *conectadas*. Dos células vivas se dicen conectadas si hay una secuencia de células vivas lindantes entre ellas. Por ejemplo, en la figura 1(a) hay 5 islotes mientras que en 1(b) hay 4.

Para contabilizar la cantidad de islotes es conveniente marcarlos<sup>1</sup>, marcando todas las células vivas del mismo con, digamos, bolitas rojas. Es importante que dos islotes diferentes tengan marcas diferentes. A modo de ejemplo, el resultado de marcar todos los islotes del universo de la figura 1(b) podría ser:



### Ejercicio 5

Escribir `hayCelulasSinMarcar`, una función que determine si hay células activas sin marcar.

### Ejercicio 6

Escribir `computarNumeroDeIslotes`, una función que compute la cantidad de islotes que hay en el universo actual. Asumir la existencia de un procedimiento `MarcarIslote`, que dado un número positivo  $n$  y asumiendo que hay células sin marcar en el universo, marca el islote  $n$ -ésimo (colocando  $n$  bolitas rojas en cada célula del islote).

## 2. Buscaminas

*Buscaminas* es un juego que se juega sobre un tablero de *locaciones*. Cada locación puede estar tapada o destapada; también puede contener minas; al comienzo todas las locaciones están tapadas. El objetivo es adivinar dónde se encuentran ubicadas las minas. El problema es que las locaciones tapadas no dejan ver si hay una mina en la misma. Si se destapa una locación con una mina, la mina explota y se termina el juego. Por ello, hay que destapar locaciones que *no* tengan minas.

Una *partida* consiste en una serie de *jugadas*. Cada una indica una locación a destapar. Para evitar tener que seleccionar locaciones a ciegas, el juego brinda un mecanismo de asistencia: las locaciones pueden tener un número denominado *pista*. El mismo indica la cantidad de minas que hay en locaciones vecinas (N, NE, E, SE, S, SO, O, NO).

Cada celda del tablero puede verse como una locación. Al comienzo están todas tapadas. Se utilizan los siguientes códigos de colores para las locaciones: **Negro** (1=tapada; 0=destapada), **Rojo** (1=hay mina, 0=no hay mina) y **Azul** (pistas – 1 a 8 bolitas).

<sup>1</sup>Esta marca no guarda relación la marca para matar o reanimar células ya vista.

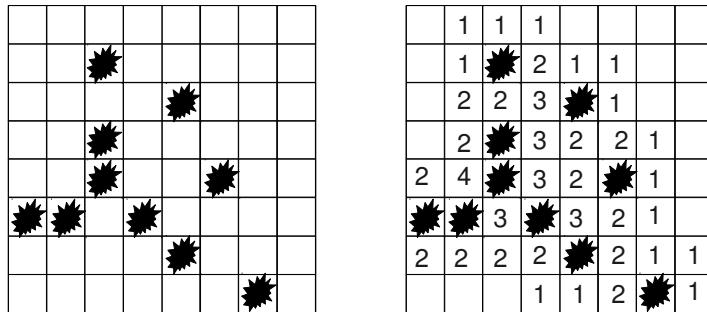


Figura 2: Ejemplo de tablero sin pistas y con pistas

### Ejercicio 7

Escribir:

- `contarMinasVecinas`, una función que retorna un número indicando el total de las minas que hay en locaciones vecinas.
- `PonerPista`, un procedimiento que pone una pista en la locación actual. Asumir que en la locación actual no hay pista ni mina.

### Ejercicio 8

Escribir `PonerPistas`, un procedimiento que pone las pistas en todas las locaciones del tablero (salvo aquellas que tienen minas). Puede asumir que hay al menos una locación con mina. Por ejemplo, dado el tablero izquierdo de la Fig. 2, debe arrojar como resultado el tablero derecho de esa figura.

Las pistas son una ayuda importante a la hora de saber si una locación tiene o no una mina. Como consecuencia de la información brindada por las pistas, para cada locación hay tres posibles juicios que pueden hacerse:

1. Seguro tiene una mina (STM).
2. Seguro no tiene mina (SNTM).
3. No se puede saber nada.

El objetivo de lo que sigue es escribir un procedimiento `InferirInfo` que analice las pistas y marque aquellas que STM y aquellas que SNTM. Esto permite al jugador tomar una decisión más informada sobre qué locación a destapar. Para poder definir `InferirInfo`, primero vamos a introducir algunos procedimientos y funciones auxiliares.

**Nota:** En lo que sigue se asume que las pistas ya fueron colocadas sobre el tablero.

### Ejercicio 9

Escribir un procedimiento `ProcesarLocaciónConPista` que revise las locaciones vecinas y marque aquellas que seguro tienen una mina (STM) con una bolita verde y aquellas que seguro no tienen una mina (SNTM) con dos bolitas verdes. Asumir que el cabezal se encuentra sobre

una locación destapada y con pista. **Ayuda:** Supongamos que la pista de la locación actual es  $n$ :

- Si hay exactamente  $n$  locaciones vecinas tapadas (sin importar si son STM o SNTM), todas las locaciones vecinas deben ser marcadas como STM (si es que ya no están marcadas como STM).
- Si hay  $n$  locaciones vecinas tapadas y marcadas como STM, entonces *todas* las demás locaciones vecinas (si las hubiere) se marcan como SNTM.

Puede asumir la existencia de una función `nroDeVecinasTapadas` que retorna un número indicando la cantidad de locaciones vecinas que están tapadas; y `nroDeVecinasMarcadasConSTM` que retorna un número indicando la cantidad de locaciones vecinas que tienen la marca de STM.

### Ejercicio 10

Escribir un procedimiento `ProcesarTodasLasLocacionesDelTablero` que procese todas las locaciones del tablero.

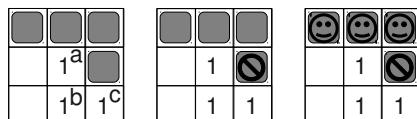
### Ejercicio 11

Escribir las siguientes funciones:

- `nroDeMarcasSTMoSNTMEnTablero` que retorna un número indicando el total de marcas (tanto STM como SNTM) que hay en el tablero.
- `hayNuevasMarcas` que determine si el procedimiento `ProcesarTodasLasLocacionesDelTablero` coloca alguna marca STM o SNTM nueva en el tablero procesado respecto al tablero a procesar.

### Ejercicio 12

Escribir un procedimiento `InferirInfo` que procese todas las locaciones con pistas poniendo las marcas de STM y SNTM hasta que no haya posibilidad de poner más marcas. Es **fundamental** observar que una vez que se procesen todas las locaciones del tablero, es posible que se presenten nuevas oportunidades para colocar marcas. Por ejemplo, en el tablero de abajo a la izquierda, asumiendo que las locaciones con pistas se recorren en el orden indicado con a,b,c, el tablero resultante sería el del medio. Notar que se ha marcado una locación como que STM. Ahora, si volvemos a procesar el tablero del medio, nos queda el del extremo derecho. Notar que las tres locaciones de la primera fila ahora están marcadas como que no tienen minas (SNTM – la carita).



### 3. SameGame

*SameGame* es un juego de un solo jugador que se juega sobre un tablero rectangular de casilleros. Inicialmente, cada casillero tiene un color (azul, negro, rojo o verde). Una *pieza* consiste de un grupo de *al menos* tres casilleros que están **unidos** y son todos del **mismo color**. Más precisamente, dos casilleros forman parte de la misma pieza cuando se puede ir desde uno hacia el otro recorriendo *exclusivamente* celdas del mismo color (con movimientos hacia el Norte, Este, Sur u Oeste). Notar que los casilleros que **no forman parte** de la pieza y son vecinos de algún casillero que **sí forma parte** de la pieza deben tener un color distinto a los casilleros de la pieza.

La Figura 3 exhibe un tablero ejemplo. El mismo tiene un total de tres piezas, indicadas con números (nota: los números no forman parte del tablero). El jugador selecciona una pieza cualquiera del tablero. Una vez hecho esto, los casilleros que componen la pieza explotan permitiendo *caer* a los casilleros que estaban encima de la pieza. Notar que a medida que las piezas explotan, algunos casilleros dejan de tener color. El jugador obtiene un *puntaje* por cada pieza que explota, que depende de la cantidad de celdas que componen la pieza explotada. El objetivo del juego es obtener el mayor puntaje posible. El juego termina cuando el tablero queda totalmente vacío o ya no hay piezas para seleccionar.

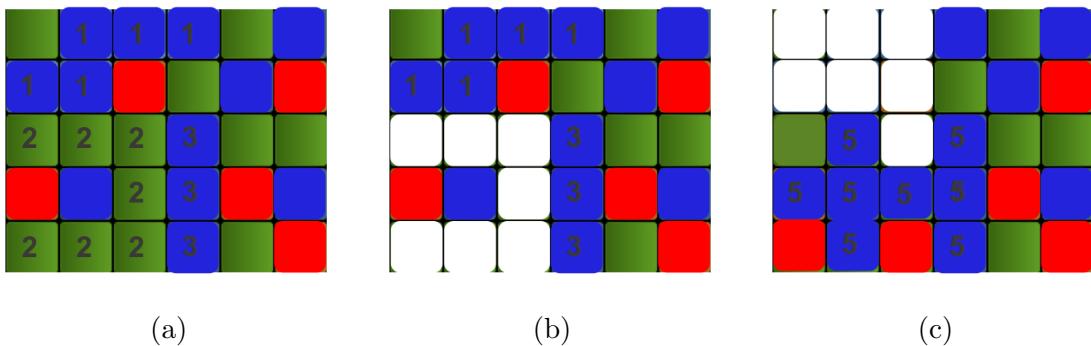


Figura 3: Tablero ejemplo con todas las piezas marcadas

Se utilizará el tablero, las celdas y las bolitas de GOBTONES para representar el tablero y los casilleros de *SameGame*. Por ejemplo, un casillero de color azul se representa con una celda con una bolita azul. Además, algunos casilleros pueden estar *marcados*. Los casilleros marcados se representan con **dos** bolitas del color del casillero. Por ejemplo, un casillero marcado de color rojo se representa con dos bolitas rojas.

#### Ejercicio 13

Escribir una función `esPieza` que retorna un booleano indicando si el casillero actual pertenece a una pieza. Para ello, alcanza con determinar si se verifica **cualquiera** de las siguientes condiciones:

1. El casillero actual tiene dos vecinos del mismo color que él.
2. El casillero actual tiene un vecino que cumple con el ítem anterior y tiene el mismo que el casillero actual.

**Nota:** puede asumir la existencia de las funciones `nroVecinos` que, dado un color  $c$ , denota la cantidad de celdas vecinas que tienen bolitas de color  $c$ , y `direccionConColor` que, dado un color  $c$ , denota la dirección de algún vecino que tienen bolitas de color  $c$ . La función `direccionConColor` es parcial; su precondición es que  $\text{nroVecinos}(c) > 0$ .

### Ejercicio 14

Escribir un procedimiento `BuscarCasilleroMarcado` que posiciona el cabezal de `GOBSTONEs` sobre el casillero que esté marcado. **Nota:** puede suponer que hay al menos un casillero marcado en el tablero.

Dado que una pieza se compone de casilleros, también puede marcarse una *pieza*. Para ello basta marcar cualquiera de los casilleros que la componen.

### Ejercicio 15

Escribir un procedimiento `ExplotarPiezaMarcada` que haga explotar la pieza que se encuentra marcada. Hacer explotar la pieza implica hacer explotar todos los casilleros que la componen. Por ejemplo, si la pieza marcada en la figura 3(a) es la 2, entonces el tablero resultando sería el de la Figura 3(b). En consecuencia el procedimiento debe:

1. Buscar el casillero marcado.
2. Marcar los vecinos que tengan el mismo color.
3. Explotar el casillero actual.
4. Repetir a)-c) hasta que no queden casilleros marcados.

**Nota:** puede asumir la existencia de la función `hayCasilleroMarcado` que indica si hay al menos un casillero marcado en el tablero, y la existencia de los procedimientos `MarcarCasillerosVecinos` que marca los casilleros vecinos al actual que tienen el mismo color y `ExplotarCasillero` que explota un casillero (i.e. deja la celda que lo modela vacía).

### Ejercicio 16

Escribir una función `puntajeDePiezaMarcada` que retorna el puntaje que se obtendría si se hiciera explotar la pieza actualmente marcada. El puntaje a retornar es  $n \times (n - 1)$  donde  $n$  es la cantidad de casilleros de la pieza. **Nota:** Puede hacer uso del ejercicio 9 de la práctica 2 (el mismo solicita realizar un procedimiento que cuente la cantidad de bolitas que haya en el tablero de un color dado).

### Ejercicio 17

Escribir un procedimiento `AplicarFuerzaDeGravedadACasillero` que mueve el casillero actual hacia el sur simulando la caída del casillero como efecto de la fuerza de gravedad. Para ello, la bolita del casillero actual debe moverse al sur, tantas veces como sea posible, sin que dos bolitas compartan el mismo casillero. **Nota:** puede asumir que el casillero actual no está vacío.

### Ejercicio 18

Escribir un procedimiento `AplicarGravedadAlTablero` que aplica la fuerza de gravedad a todos los casilleros, de forma tal que ningún casillero quede “flotando”. Por ejemplo, si se

explota la pieza 2 de la Figura 3(a), entonces el tablero resultando sería el de la Figura 3(c). El nuevo tablero tiene sola una pieza (que además es nueva).

Suponga que dispone de un procedimiento `InteligenciaArtificial`, que no tiene parámetros, cuyo objetivo es proveer una estrategia de juego. Para ello, `InteligenciaArtificial` marca la pieza del tablero que se debe explotar de acuerdo a la estrategia. Como precondición, ninguna pieza puede estar marcada y el juego no debe haber *finalizado*, i.e. aún deben quedar piezas en el tablero.

### Ejercicio 19

Escribir una función `simularJuego` que simula el juego con la estrategia de la inteligencia artificial y retorna el puntaje conseguido. **Nota:** puede suponer que ninguna pieza se encuentra marcada y que existe una función `juegoFinalizado` que devuelve un booleano indicando si quedan piezas.

## 4. Nurikabe

NURIKABE es un juego en el que un jugador debe dibujar un río en un mapa de forma tal de generar un conjunto de islas que satisfagan ciertas particularidades. El *mapa* es una grilla rectangular formada por celdas. Inicialmente algunas de estas celdas contienen un número y el resto se encuentran vacías (ver Figura 4(a)). Las celdas con número se llaman *indicadoras*. A medida que el juego transcurre, el jugador va *pintando* algunas de las celdas vacías con un lápiz negro; estas celdas negras van formando los segmentos del río, y dan origen a la formación de islas. Una *isla* es simplemente una región sin celdas negras<sup>2</sup> que está bordeada por el río o por los bordes del mapa (ver Figura 4(b)). El objetivo del juego es dibujar el río de forma tal que al finalizar se satisfagan las cuatro condiciones siguientes (ver Figuras 4(c) y 4(d)).

- Cada isla tiene una única celda indicadora (el resto de sus celdas están vacías).
- Cada isla tiene tantas celdas como el número que indica su única celda indicadora.
- El río es continuo, es decir, se debe poder recorrer todas las celdas negras con movimientos horizontales y verticales sin pasar por celdas blancas.
- No hay ninguna región de  $2 \times 2$  celdas que contenga sólo celdas negras.

Para representar el mapa vamos a utilizar el tablero. Cada celda pintada se representa poniendo una bolita negra, las celdas indicadoras tienen tantas bolitas azules como el número correspondiente, y las celdas vacías no tienen bolitas (ver Figura 5).

Para simplificar la tarea de programar el juego, nos conviene tener un procedimiento que nos permita marcar cada isla del mapa, usando bolitas **rojas**. Decimos que una celda está *marcada* si contiene una bolita roja y *desmarcada* en caso contrario.

### Ejercicio 20

Una celda es *marcable* cuando (a) es parte de una isla (b) está desmarcada, y (c) es lindante con alguna celda marcada. Escribir una función `hayCeldaMarcable` que indique si el tablero tiene alguna celda marcable.

---

<sup>2</sup>Es decir, contiene celdas blancas o celdas indicadoras.

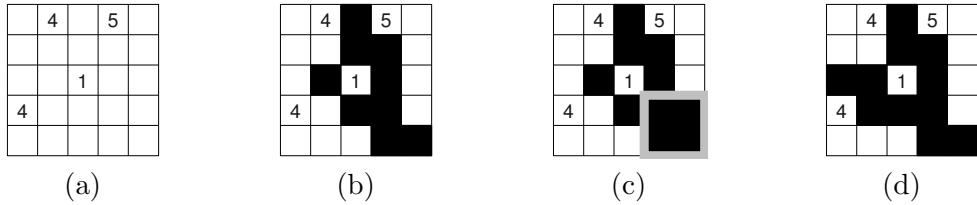


Figura 4: (a) Mapa inicial con cuatro casillas indicadoras. (b) Resolución parcial del mapa inicial con tres islas y un río que no es continuo; notar, además, que una de las islas contiene dos indicadores. (c) Solución del mapa inicial que es incorrecta porque tiene una región de  $2 \times 2$  celdas negras (remarcadas con gris), y porque la isla que contiene el indicador 5 tiene sólo 4 celdas. (d) Solución del mapa inicial que es correcta porque el río es continuo, cada isla tiene la misma cantidad de celdas que la que indica su única casilla indicadora, y no hay regiones de  $2 \times 2$  celdas negras.

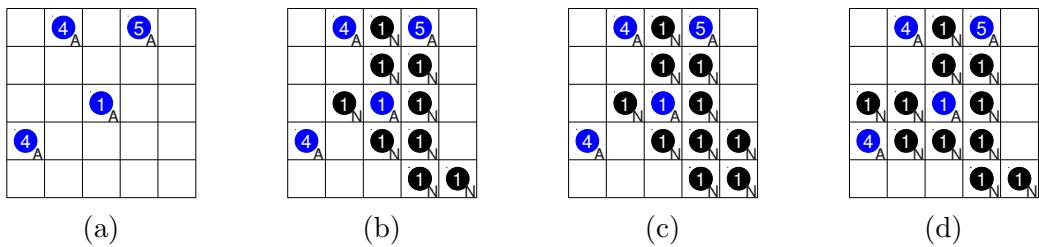
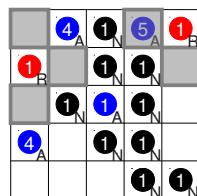


Figura 5: Representaciones de los mapas de la Figura 4 en Gobstones.

**Sugerencia:** escribir una función `esCeldaMarcable` que determine si la celda actual es marcable.

**Usar sin definir:** `lindanteMarcada(d)` que denota `True` cuando existe una celda en dirección `d` que además está marcada.<sup>3</sup>

**Ejemplo:** las celdas resaltadas son las **únicas** marcables, porque están desmarcadas, forman parte de una isla y son lindantes (al N, E, S u O) a celdas marcadas.



### Ejercicio 21

Escribir un procedimiento `MarcarIsla` que marque todas las celdas de la isla que contiene la celda actual. Para ello, **debe** comenzar marcando la celda actual, y luego **debe** marcar todas las celdas marcables hasta que no queden más celdas marcables.<sup>4</sup>

**Precondición:** no hay ninguna celda marcada en el tablero antes de la invocación del pro-

<sup>3</sup>Notar que `lindanteMarcada` no tiene precondición. En caso que la celda actual sea la última en dirección `d`, la función denota `False`.

<sup>4</sup>La estrategia propuesta funciona. No hace falta justificar este hecho, basta con implementar lo que se indica.

cedimiento. La celda actual forma parte de una isla.

**Usar sin definir:** IrACeldaMarcable que posiciona el cabezal en una celda marcable. Como precondición, debe haber al menos una celda marcable en el tablero.

**Ejemplos:** El tablero de la derecha resulta de aplicar MarcarIsla en el tablero de la izquierda cuando el cabezal se encuentra en la celda recuadrada.



Recordemos que para resolver un mapa hay que dibujar el río de forma tal que cada isla contenga exactamente una celda indicadora. Más aún, cada isla debe tener tantas celdas como el número que aparece en la celda indicadora (ver Figura 4 (d)). Las siguientes funciones se utilizan para verificar qué islas fueron resueltas.

### Ejercicio 22

Escribir la función `tamañoIsla` que denota la cantidad de celdas de la isla que contiene la celda actual.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte de una isla.

**Observación:** recordar la función `nroBolitasTotal(c)` (ejercicio 16, práctica 5).

**Ejemplos:** el tamaño de la isla de la izquierda es 10, el de la derecha es 6.



### Ejercicio 23

Escribir la función `nroIndicadoras` que denota la cantidad de celdas indicadoras de la isla que contiene la celda actual.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte de una isla.

**Ejemplo:** la isla de la izquierda tiene 2 indicadores, el de la derecha 1.



### Ejercicio 24

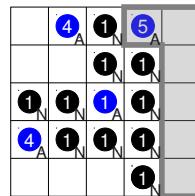
Escribir la función `indicador` que denota el valor del indicador de la isla que contiene la celda

actual.

**Precondición:** hay una única celda indicadora en la isla. No hay ninguna celda marcada antes de invocar la función. La celda actual forma parte de una isla.

**Usar sin definir:** IrAIIndicadorMarcado que posiciona el cabezal en alguna celda indicadora que está marcada. Como **precondición**, tiene que haber al menos una celda indicadora marcada.

**Ejemplo:** el indicador de la isla marcada es 5.

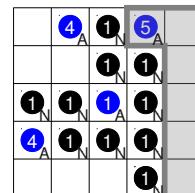
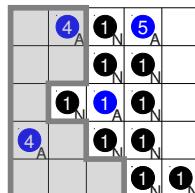
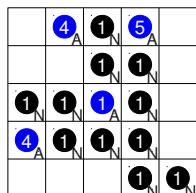


## Ejercicio 25

Decimos que una isla esta *resuelta* si (a) contiene exactamente una celda indicadora, y (b) la cantidad de celdas de la isla es igual al n mero de la celda indicadora. Escribir la funci n `quedanIslasSinResolver` que retorna true si el mapa contiene alguna isla no resuelta.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función.

**Ejemplos:** en el tablero de la izquierda todas las islas están resueltas; en el tablero del centro la isla marcada no está resuelta dado que contiene dos celdas indicadoras; en el tablero de la derecha la isla marcada no está resuelta porque tiene 6 celdas y su indicador es 5.



Para finalizar, la siguiente función se usará para determinar si el río no tiene regiones pintadas de  $2 \times 2$  celdas.

## Ejercicio 26

Un *lago* es una región de  $2 \times 2$  celdas del río (ver Figura 4 (c)). Escribir una función `estaEnLago` que indique si la celda actual es parte de un lago.

**Usar sin definir:** `hayRioAl(d)` que denota `True` si la celda en dirección `d` es parte del río; como precondición, debe ser posible moverse en dirección `d`. `hayRioDiag(d)` que denota `True` si la celda en dirección `d + siguiente(d)` es parte del río; como precondición, debe ser posible moverse tanto en dirección `d` como `siguiente(d)`.

**Ejemplo:** en el tablero de la izquierda la celda bajo el cabezal —recuadrada en gris— forma parte de un lago, mientras que en el tablero de la derecha la celda bajo el cabezal no forma parte de un lago.



## 5. Batalla Naval

La *Batalla Naval* es un juego conocido. Cada uno de dos jugadores cuenta con una flota de barcos que colocan sobre un tablero propio sin que el otro sepa dónde. Cada barco puede ocupar una o más celdas contiguas (cada una de ellas representa una “sección” del barco). Luego se turnan para disparar bombas sobre las celdas del tablero del oponente, anunciando en cada caso las coordenadas donde las mismas son arrojadas. Si en la coordenada anunciada hay una sección de un barco del oponente, entonces esa sección es destruida. El barco se declara “hundido” una vez que todas sus secciones son destruidas. Gana el primero que logra hundir todos los barcos de la flota del oponente.

Vamos a modelar este juego en GOBSTONEs. Para ello el tablero estará dividido en dos partes, a saber la *parte de datos* y la *parte de juego* (ver Fig. 6).

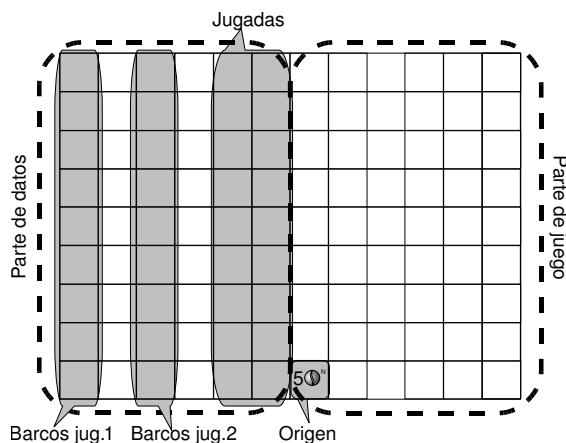


Figura 6: División del tablero

### Parte de datos

La parte de datos se usa para codificar los datos del juego y ocupa las primeras seis columnas. Las primeras 4 de estas están destinadas a codificar los barcos de ambos jugadores. La columna 1 codifica los barcos del jugador 1 mientras que la columna 3 los del jugador 2 (la finalidad de las columnas 2 y 4 será explicado en breve). Cada barco se representa a través de tres ingredientes:

- Una coordenada: indicada con bolitas azules (eje de las x) y bolitas verdes (eje de las y)

- Un tamaño: indicado con bolitas rojas
- Una dirección: indicada con bolitas negras (1 = norte, 2 = este, 3 = sur, 4 = oeste)

El resto de la parte de datos (columnas 5 y 6) codifica las jugadas, representadas como coordenadas. En estas columnas, si una celda corresponde con una jugada de un jugador entonces la celda lindante al sur tiene los datos de la jugada del oponente. Cuando hay más jugadas que el alto del tablero, estas continúan al tope de la columna 6.

Todas las coordenadas se miden con respecto al origen (esquina Sur-Oeste) de la parte de juego. La misma está indicada con 5 bolitas negras.

## Parte de juego

En la parte de juego se dibujarán los barcos y se jugará. Las bolitas azules marcarán los barcos del jugador 1 y las verdes los del jugador 2. Con bolitas rojas se demarcarán las secciones de barcos que han sido destruidas. Por limitaciones de espacio las flotas de ambos jugadores se colocan sobre la parte de juego (se asume que los barcos no se solapan).

## Problemas

### Ejercicio 27

Asuma que el cabezal se encuentra en una celda de la parte de datos que codifica un barco. Escribir un procedimiento *RenderBarco* que tome un color por parámetro y “dibuje” ese barco con el color dado en la parte de juego del tablero. Debe tenerse en cuenta que:

1. La primera sección del barco se marca con dos bolitas del color dado
2. La primera sección del barco también debe incluir las bolitas negras que indican la dirección
3. Las demás secciones del barco solamente contendrán una bolita del color dado

A modo de ejemplo, la figura 7 muestra cómo se dibujan los barcos sobre la parte de juego basándose en la codificación de la parte de datos. Se exhibe cómo quedaría el tablero al finalizar de dibujar todos los barcos de la parte de datos asumiendo que las celdas enumeradas del 1 al 6 contienen:

Celda	Azul (x)	Verde (y)	Negro (dir)	Rojo (tamaño)
1	0	2	1	3
2	2	4	3	2
3	5	8	3	5
4	1	0	2	3
5	4	2	4	2
6	1	6	2	4

**Nota:** Puede asumir la existencia de un procedimiento *IrACoordenadaBN* que toma dos números y ubica el cabezal en esa coordenada **relativa** al origen de la parte de juego. Ej. *IrACoordenadaBN(0,0)* se ubica sobre la celda que tiene 5 bolitas negras.

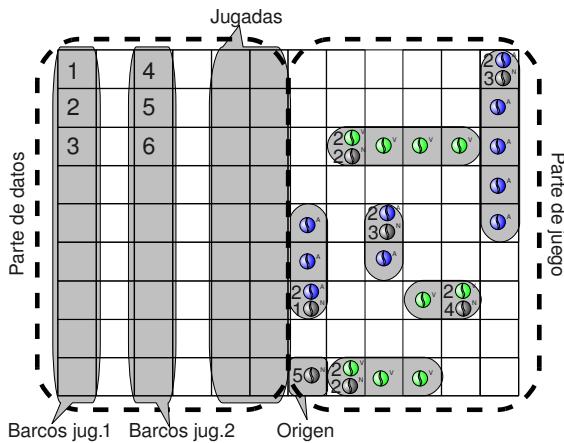


Figura 7: Dibujo de barcos en base a su codificación

### Ejercicio 28

Escribir un procedimiento *RenderBarcos* que tome un color por parámetro y recorre todos los barcos del jugador de ese color y los “dibuje” en la parte de juego utilizando el procedimiento anterior. Utilice las columnas libres de la parte de datos para demarcar (con seis bolitas negras) el barco que está siendo procesado actualmente. De esta manera, por cada barco que se encuentre codificado se deberá:

- poner seis bolitas negras en la celda lindante al Este
- dibujar el barco utilizando el procedimiento anterior,
- retornar a la celda inicial, buscando las seis bolitas negras,
- eliminar las seis bolitas negras.

**Nota:** Estructurarlo como recorrido. Puede asumir que el cabezal se encuentra en el tope de la columna 1 o 3 según el color sea azul o verde.

### Ejercicio 29

Escribir una función *estaHundido* que, asumiendo que el cabezal se encuentra en la celda de la parte de datos que codifica un barco, retorne un valor de verdad indicando si el mismo está hundido o no. Recordar que las secciones destruidas se indican con una bolita roja, y un barco se considera hundido si todas sus secciones han sido destruidas.

### Ejercicio 30

Escribir una función *perdio*, que reciba por parámetro un color (azul o verde) y determine si todos los barcos de ese color han sido hundidos. **Nota:** Estructurar como recorrido sobre las celdas de la parte de datos del tablero y apelar al ejercicio anterior.

### Ejercicio 31

Escribir un procedimiento *Disparo* que reciba un color y simule un disparo sobre los barcos

de ese color. La coordenada del disparo deberá tomarla de la celda actual. En caso de ser efectivo el disparo colocar una bolita roja en el lugar del impacto, indicando que se destruye esa sección de barco.

Nuevamente, se pide que utilice seis bolitas negras para demarcar el disparo actual que está siendo procesado.

### Ejercicio 32

Escribir una función *simularBN* que, partiendo de un tablero en el cual la parte de juego se encuentra en blanco, utilice la información contenida en la parte de datos para simular el juego y devuelva el color ganador.

Naturalmente deberá comenzar renderizando todos los barcos codificados en las columnas 1 y 3, para luego recorrer uno a uno los disparos.

La simulación termina cuando uno de los jugadores gana o bien cuando se acaban los disparos codificados (la celda que correspondería al siguiente disparo se encuentra en blanco o se alcanza el límite sur de la columna 6).

La función debe retornar el color de los barcos del jugador ganador, o bien el color negro para indicar que con esa secuencia de disparos ninguno de los dos jugadores llegó a hundir todos los barcos del contrincante.

# Introducción a la Programación – UNQ – 2<sup>do</sup> semestre de 2013

## Primer parcial – NURIKABE I

NURIKABE es un juego en el que un jugador debe dibujar un río en un mapa de forma tal de generar un conjunto de islas que satisfagan ciertas particularidades. El *mapa* es una grilla rectangular formada por celdas. Inicialmente algunas de estas celdas contienen un número y el resto se encuentran vacías (ver Figura 1(a)). Las celdas con número se llaman *indicadoras*. A medida que el juego transcurre, el jugador va *pintando* algunas de las celdas vacías con un lápiz negro; estas celdas negras van formando los segmentos del río, y dan origen a la formación de islas. Una *isla* es simplemente una región sin celdas negras<sup>1</sup> que está bordeada por el río o por los bordes del mapa (ver Figura 1(b)). El objetivo del juego es dibujar el río de forma tal que al finalizar se satisfagan las cuatro condiciones siguientes (ver Figuras 1(c) y 1(d)).

- Cada isla tiene una única celda indicadora (el resto de sus celdas están vacías).
- Cada isla tiene tantas celdas como el número que indica su única celda indicadora.
- El río es continuo, es decir, se debe poder recorrer todas las celdas negras con movimientos horizontales y verticales sin pasar por celdas blancas.
- No hay ninguna región de  $2 \times 2$  celdas que contenga sólo celdas negras.

El objetivo del presente parcial es programar parte del juego NURIKABE. Para representar el mapa vamos a utilizar el tablero. Cada celda pintada se representa poniendo una bolita negra, las celdas indicadoras tienen tantas bolitas azules como el número correspondiente, y las celdas vacías no tienen bolitas (ver Figura 2).

Para simplificar la tarea de programar el juego, nos conviene tener un procedimiento que nos permita marcar cada isla del mapa, usando bolitas **rojas**. Decimos que una celda está *marcada* si contiene una bolita roja y *desmarcada* en caso contrario.

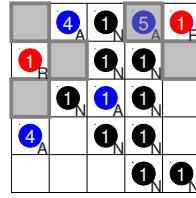
### EJERCICIO 1 (2 puntos)

Una celda es marcable cuando (a) es parte de una isla (b) está desmarcada, y (c) es lindante con alguna celda marcada. Escribir una función `hayCeldaMarcable` que indique si el tablero tiene alguna celda marcable.

**Sugerencia:** escribir una función `esCeldaMarcable` que determine si la celda actual es marcable.

**Usar sin definir:** `lindanteMarcada(d)` que denota `True` cuando existe una celda en dirección `d` que además está marcada.<sup>2</sup>

**Ejemplo:** las celdas resaltadas son las **únicas** marcables, porque están desmarcadas, forman parte de una isla y son lindantes (al N, E, S u O) a celdas marcadas.



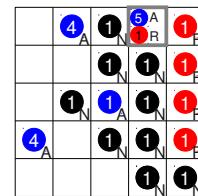
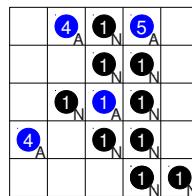
### EJERCICIO 2 (1 punto)

Escribir un procedimiento `MarcarIsla` que marque todas las celdas de la isla que contiene la celda actual. Para ello, **debe** comenzar marcando la celda actual, y luego **debe** marcar todas las celdas marcables hasta que no queden más celdas marcables.<sup>3</sup>

**Precondición:** no hay ninguna celda marcada en el tablero antes de la invocación del procedimiento. La celda actual forma parte de una isla.

**Usar sin definir:** `IrACeldaMarcable` que posiciona el cabezal en una celda marcable. Como precondición, debe haber al menos una celda marcable en el tablero.

**Ejemplos:** El tablero de la derecha resulta de aplicar `MarcarIsla` en el tablero de la izquierda cuando el cabezal se encuentra en la celda recuadrada.



Recordemos que para resolver un mapa hay que dibujar el río de forma tal que cada isla contenga exactamente una celda indicadora. Más aún, cada isla debe tener tantas celdas como el número que aparece en la celda indicadora (ver Figura 1 (d)). Las siguientes funciones se utilizan para verificar qué islas fueron resueltas.

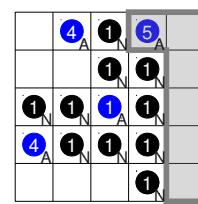
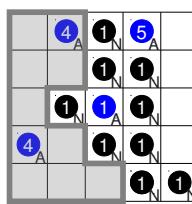
### EJERCICIO 3 (0.5 puntos)

Escribir la función `tamañoIsla` que denota la cantidad de celdas de la isla que contiene la celda actual.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte de una isla.

**Observación:** recordar la función `nroBolitasTotal(c)` (ejercicio 16, práctica 5).

**Ejemplos:** el tamaño de la isla de la izquierda es 10, el de la derecha es 6.



### EJERCICIO 4 (1 punto)

<sup>1</sup>Es decir, contiene celdas blancas o celdas indicadoras.

<sup>2</sup>Notar que `lindanteMarcada` no tiene precondición. En caso que la celda actual sea la última en dirección `d`, la función denota `False`.

<sup>3</sup>La estrategia propuesta funciona. No hace falta justificar este hecho, basta con implementar lo que se indica.

4	5
	1
4	

(a)

4	5
	1
4	

(b)

4	5
	1
4	

(c)

4	5
	1
4	

(d)

Figura 1: (a) Mapa inicial con cuatro casillas indicadoras. (b) Resolución parcial del mapa inicial con tres islas y un río que no es continuo; notar, además, que una de las islas contiene dos indicadores. (c) Solución del mapa inicial que es incorrecta porque tiene una región de  $2 \times 2$  celdas negras (remarcadas con gris), y porque la isla que contiene el indicador 5 tiene sólo 4 celdas. (d) Solución del mapa inicial que es correcta porque el río es continuo, cada isla tiene la misma cantidad de celdas que la que indica su única casilla indicadora, y no hay regiones de  $2 \times 2$  celdas negras.

4	5
	1
4	

(a)

4	1	5
	1	
1	1	
1	1	1
4	1	1

(b)

4	1	5
	1	
1	1	
1	1	1
4	1	1

(c)

4	1	5
	1	
1	1	
1	1	1
4	1	1

(d)

Figura 2: Representaciones de los mapas de la Figura 1 en Gobstones.

Escribir la función `nroIndicadoras` que denota la cantidad de celdas indicadoras de la isla que contiene la celda actual.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte de una isla.

**Ejemplo:** la isla de la izquierda tiene 2 indicadores, el de la derecha 1.

4	1	3
	1	
1	1	
1	1	1
4	1	1

4	1	5
	1	
1	1	
1	1	1
4	1	1

### EJERCICIO 5 (0.5 puntos)

Escribir la función `indicador` que denote el valor del indicador de la isla que contiene la celda actual.

**Precondición:** hay una única celda indicadora en la isla. No hay ninguna celda marcada antes de invocar la función. La celda actual forma parte de una isla.

**Usar sin definir:** `IrAIIndicadorMarcado` que posiciona el cabezal en alguna celda indicadora que está marcada. Como precondición, tiene que haber al menos una celda indicadora marcada.

**Ejemplo:** el indicador de la isla marcada es 5.

4	1	5
	1	
1	1	
1	1	1
4	1	1

### EJERCICIO 6 (2 puntos)

Decimos que una isla está resuelta si (a) contiene exactamente una celda indicadora, y (b) la cantidad de celdas de la isla es igual al número de la celda indicadora. Escribir la función `quedanIslasSinResolver` que retorna true si el mapa contiene alguna isla no resuelta.

**Precondición:** no hay ninguna celda marcada en el table-

ro antes de invocar la función.

**Ejemplos:** en el tablero de la izquierda todas las islas están resueltas; en el tablero del centro la isla marcada no está resuelta dado que contiene dos celdas indicadoras; en el tablero de la derecha la isla marcada no está resuelta porque tiene 6 celdas y su indicador es 5.

4	1	5
	1	
1	1	
1	1	1
4	1	1

4	1	5
	1	
1	1	
1	1	1
4	1	1

4	1	5
	1	
1	1	
1	1	1
4	1	1

Para finalizar, la siguiente función se usará para determinar si el río no tiene regiones pintadas de  $2 \times 2$  celdas.

### EJERCICIO 7 (3 puntos)

Un lago es una región de  $2 \times 2$  celdas del río (ver Figura 1 (c)). Escribir una función `estaEnLago` que indique si la celda actual es parte de un lago.

**Usar sin definir:** `hayRioAl(d)` que denota True si la celda en dirección `d` es parte del río; como precondición, debe ser posible moverse en dirección `d`. `hayRioDiag(d)` que denota True si la celda en dirección `d + siguiente(d)` es parte del río; como precondición, debe ser posible moverse tanto en dirección `d` como `siguiente(d)`.

**Ejemplo:** en el tablero de la izquierda la celda bajo el cabezal —recuadrada en gris— forma parte de un lago, mientras que en el tablero de la derecha la celda bajo el cabezal no forma parte de un lago.

4	1	5
	1	
1	1	
1	1	1
4	1	1

4	1	5
	1	
1	1	
1	1	1
4	1	1

# Introducción a la Programación – Algoritmos y Programación

## UNQ – 2<sup>do</sup> semestre de 2013 – Primer parcial – NURIKABE II

NURIKABE es un juego en el que un jugador debe dibujar un río en un mapa de forma tal de generar un conjunto de islas que satisfagan ciertas particularidades. El *mapa* es una grilla rectangular formada por celdas. Inicialmente algunas de estas celdas contienen un número y el resto se encuentran vacías (ver Figura 1(a)). Las celdas con número se llaman *indicadoras*. A medida que el juego transcurre, el jugador va *pintando* algunas de las celdas vacías con un lápiz negro; estas celdas negras van formando los segmentos del río, y dan origen a la formación de islas. Una *isla* es simplemente una región sin celdas negras<sup>1</sup> que está bordeada por el río o por los bordes del mapa (ver Figura 1(b)). El objetivo del juego es dibujar el río de forma tal que al finalizar se satisfagan las cuatro condiciones siguientes (ver Figuras 1(c) y 1(d)).

- Cada isla tiene una única celda indicadora (el resto de sus celdas están vacías).
- Cada isla tiene tantas celdas como el número que indica su única celda indicadora.
- El río es continuo, es decir, se debe poder recorrer todas las celdas negras con movimientos horizontales y verticales sin pasar por celdas blancas.
- No hay ninguna región de  $2 \times 2$  celdas que contenga sólo celdas negras.

El objetivo del presente parcial es programar parte del juego NURIKABE. Para representar el mapa vamos a utilizar el tablero. Cada celda pintada se representa poniendo una bolita negra, las celdas indicadoras tienen tantas bolitas azules como el número correspondiente, y las celdas vacías no tienen bolitas (ver Figura 2).

Para simplificar la tarea de programar el juego, nos conviene tener un procedimiento que nos permita marcar cada segmento del río, usando bolitas **rojas**. Decimos que una celda está *marcada* si contiene una bolita roja y *desmarcada* en caso contrario.

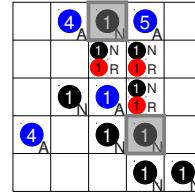
### EJERCICIO 1 (2 puntos)

Una celda es marcable cuando (a) es parte del río (b) está desmarcada, y (c) es lindante con alguna celda marcada. Escribir una función `hayCeldaMarcable` que indique si el tablero tiene alguna celda marcable.

**Sugerencia:** escribir una función `esCeldaMarcable` que determine si la celda actual es marcable.

**Usar sin definir:** `lindanteMarcada(d)` que denota `True` cuando existe una celda en dirección `d` que además está marcada<sup>2</sup>

**Ejemplo:** las celdas resaltadas son las **únicas** celdas marcables, porque están desmarcadas, forman parte del río y son lindantes (al N, E, S u O) a celdas marcadas.



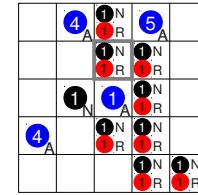
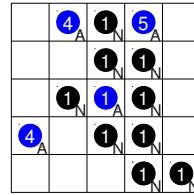
### EJERCICIO 2 (1 punto)

Escribir un procedimiento `MarcarSegmentoRío` que marque todas las celdas del segmento del río que contiene la celda actual. Para ello, **debe** comenzar marcando la celda actual, y luego **debe** marcar todas las celdas marcables hasta que no queden más celdas marcables.<sup>3</sup>

**Precondición:** no hay ninguna celda marcada en el tablero antes de la invocación del procedimiento. La celda actual forma parte del río.

**Usar sin definir:** `IrACeldaMarcable` que posiciona el cabezal en una celda marcable. Como precondición, debe haber al menos una celda marcable en el tablero.

**Ejemplos:** El tablero de la derecha resulta de aplicar `MarcarRío` en el tablero de la izquierda cuando el cabezal se encuentra en la celda recuadrada.



Recordemos que para resolver un mapa hay que dibujar el río de forma tal que cada isla contenga exactamente una celda indicadora, de forma tal que el río sea continuo y no tenga lagos. Las siguientes funciones se usan para verificar las propiedades del río.

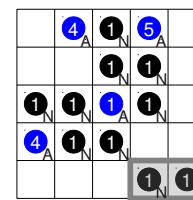
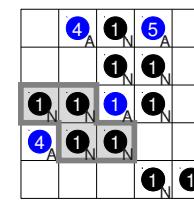
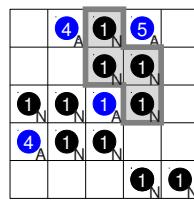
### EJERCICIO 3 (0.5 puntos)

Escribir la función `tamañoSegmentoRío` que denote la cantidad de celdas del segmento del río que contiene la celda actual.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte del río.

**Observación:** recordar la función `nroBolitasTotal(c)` (ejercicio 16, práctica 5).

**Ejemplos:** de izquierda a derecha, los tamaños de los segmentos del río son 4, 4 y 2.



<sup>1</sup>Es decir, contiene celdas blancas o celdas indicadoras.

<sup>2</sup>Notar que `lindanteMarcada` no tiene precondición. En caso que la celda actual sea la última en dirección `d`, la función denota `False`.

<sup>3</sup>La estrategia propuesta funciona. No hace falta justificar este hecho, basta con implementar lo que se indica.

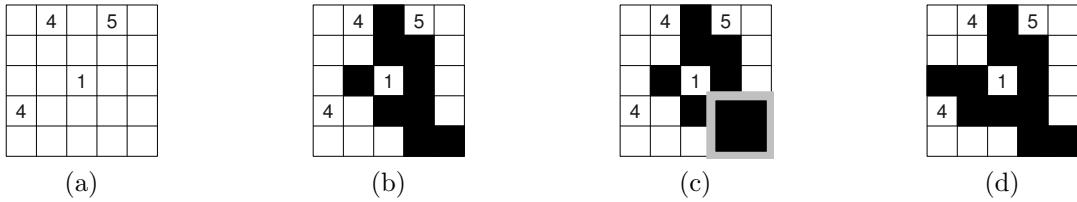


Figura 1: (a) Mapa inicial con cuatro casillas indicadoras. (b) Resolución parcial del mapa inicial con tres islas y un río que no es continuo; notar, además, que una de las islas contiene dos indicadores. (c) Solución del mapa inicial que es incorrecta porque tiene una región de  $2 \times 2$  celdas negras (remarcadas con gris), y porque la isla que contiene el indicador 5 tiene sólo 4 celdas. (d) Solución del mapa inicial que es correcta porque el río es continuo, cada isla tiene la misma cantidad de celdas que la que indica su única casilla indicadora, y no hay regiones de  $2 \times 2$  celdas negras.

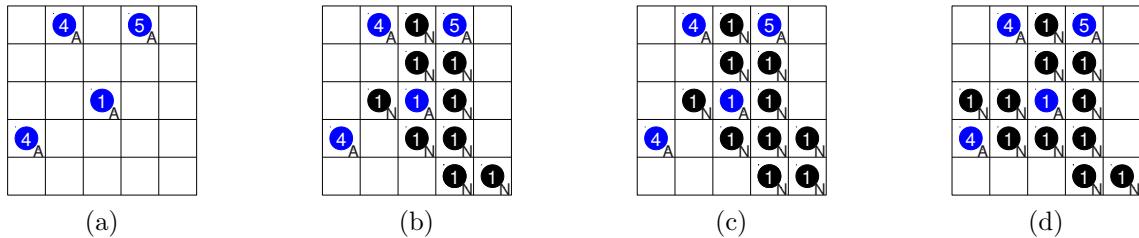


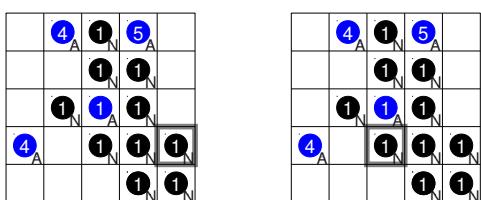
Figura 2: Representaciones de los mapas de la Figura 1 en Gobstones.

#### EJERCICIO 4 (3 puntos)

Un lago es una región de  $2 \times 2$  celdas del río (ver Figura 1 (c)). Escribir una función `estaEnLago` que indique si la celda actual es parte de un lago.

**Usar sin definir:** `hayRioAl(d)` que denota True si la celda en dirección `d` es parte del río; como precondition, debe ser posible moverse en dirección `d`. `hayRioDiag(d)` que denota True si la celda en dirección `d + siguiente(d)` es parte del río; como precondition, debe ser posible moverse tanto en dirección `d` como `siguiente(d)`.

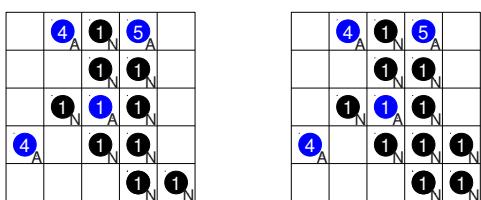
**Ejemplo:** en el tablero de la izquierda la celda bajo el cabezal —recuadrada en gris— forma parte de un lago, mientras que en el tablero de la derecha la celda bajo el cabezal no forma parte de un lago.



#### EJERCICIO 5 (1 punto)

Escribir la función `nroCeldasEnLago` que denota la cantidad de celdas del río que pertenecen a un lago.

**Ejemplo:** En el tablero de la izquierda hay 0 celdas en un lago, mientras que en el de la derecha hay 4.



#### EJERCICIO 6 (0.5 puntos)

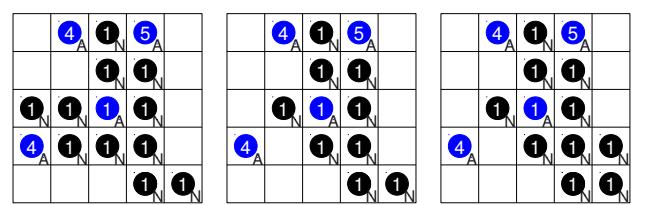
Escribir la función `rioCorrecto` que denota True si (a) el río es continuo y (b) no contiene lagos. En caso en que

no haya celdas pintadas, la función debe retornar True.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función.

**Usar sin definir:** `IrARio` que posiciona el cabezal en alguna celda del río; como precondition, debe haber al menos una celda del río. `nroCeldasRio` que denota la cantidad **total** de celdas que forman parte del río.

**Ejemplos:** en el tablero de la izquierda el río es correcto, el del centro no es correcto porque no es continuo y el de la derecha no es correcto porque tiene un lago.

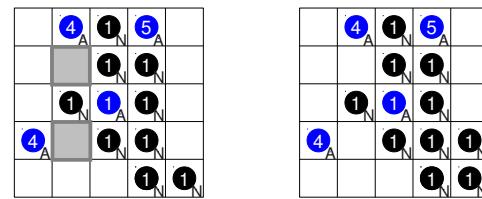


#### EJERCICIO 7 (2 puntos)

Decimos que una celda es conectora cuando (a) no esta pintada, (b) no es indicadora y (c) el río es correcto luego de pintarla (no importa si antes era correcto o no). Escribir una función `quedanCeldasConectoras` que denota True si hay al menos una celda conectora en el tablero.

**Precondición:** no hay ninguna celda marcada en el tablero antes de invocar la función.

**Ejemplos:** En el tablero de la izquierda las celdas marcadas son conectoras, mientras que el tablero de la derecha no contiene celdas conectoras.



# Práctica 10

## Listas de Registros

Introducción a la Programación  
2<sup>do</sup> Semestre de 2016

Los ejercicios que corresponden a los **contenidos mínimos** recomendados se encuentran marcados con el simbolo  $\circledast$ .

### 1. Ejercicios introductorios

#### Ejercicio 1

Suponga un pirata en una isla con un mapa. Su objetivo es encontrar el tesoro en esa isla. La isla esta modelada con el tablero de GOBTONES. El mapa es una lista cuyos elementos son registros de tipo **Indicacion** y el tesoro es una celda con 2 bolitas rojas. El punto de partida está indicado con una celda con 1 bolita roja.

```
# Un mapa es una lista de indicaciones
type Indicacion is record
{
    field direccion      # Dir
    field distancia       # número
}
```

Implementar la función **busquedaDelTesoro(mapa)**, que tome un mapa y retorna un booleano indicando si encontró o no el tesoro. Tenga en cuenta que el mapa puede tener errores (en el sentido que siguiendo sus indicaciones podría caer al agua – más allá de los límites del tablero hay agua) y el tesoro puede no existir.

#### Ejercicio 2

Utilizando el tipo **Persona** (práctica de registros), realice las siguientes funciones, suponiendo que no hay dos personas con el mismo número de DNI, donde usamos el tipo Padrón como renombre de listas de personas:

1. **perteneceDNI(padron, dni)**), recibe un padrón y un número de DNI y retorna true si hay una persona con dicho DNI en el padrón.
2. **personaConDNI(padron, dni)**, recibe un padrón y un número de DNI y devuelve la primer persona que tenga el DNI indicado. Puede suponer que existe alguna persona con dicho DNI.
3. **convivientes(padron, domicilio)**, recibe un padrón y un domicilio y retorna todas las personas que viven en dicho domicilio.

4. **sinPersonaConDNI(padron, dni)**, recibe un padrón y un número de DNI y retorna el padrón que se obtiene de sacar a la persona con dicho DNI.
5. **sinPersonasConDNI(padron, dnis)**, recibe un padrón y una lista con números de DNI y retorna el padrón que resulta de eliminar todas las personas correspondientes a los DNIs pasados como parámetro.
6. **nuevosDatosPersona(padron, dniViejo, persona)**, recibe un padrón, un dni y una persona y retorna el padrón que se obtiene de actualizar los datos la persona con el dni viejo para que coincidan con la nueva persona. En otras palabras, debe eliminarse aquella persona del padrón correspondiente al dni viejo, y agregarse la nueva persona.
7. **mudarFamilia(padron, domicilioAntiguo, domicilioNuevo)** que recibe una lista de personas y dos domicilios, y actualiza los datos de todas las personas con domicilio antiguo para que pasen a tener domicilio nuevo.
8. **dameUnaFamilia(padron)**, recibe un padrón y retorna una lista de personas convivientes.
9. **familias**, recibe un padrón y retorna la lista de todas las familias (que es una lista de lista de personas).
10. **consistente(padron)**, recibe un padrón y verifica que no hayan personas con el mismo dni que otras.

### Ejercicio 3

Declarar los tipos de registro **Alumno**, **Parcial** y **Curso** con la siguiente información:

- **Parcial** representa un parcial dado por un alumno, y sólo recuerda el tema (un numero) y la nota (otro número).
- Por cada alumno debe guardarse el legajo (un número) y la lista de parciales que rindió. En esa lista puede haber más de un parcial para el mismo tema, en caso que el alumno haya tenido que recuperar.
- Del curso se guarda la lista de alumnos y la lista de temas a evaluar (es una lista de números), por ejemplo “gobstones”, “recorridos” y “listas” (cada tema se corresponde con un número).

Implementar las siguientes funciones:

1. **nuevoCurso(temas)**, recibe una lista de temas y crea un curso con una lista vacía de alumnos.
2. **inscripcionAlumno(curso, legajo)**, recibe un curso y un legajo y agrega un alumno con ese legajo al curso, con una lista vacía de parciales.
3. **evaluacionAlumno(curso, legajo, tema, nota)**, recibe un curso, el legajo de un alumno, un tema y una nota, y retorna el curso en el que se registró que el alumno correspondiente al legajo rindió el tema con la nota correspondiente. Como precondición, el alumno correspondiente y el tema del parcial deben formar parte del curso. La

función **debe** denotar BOOM cuando no se cumpla la precondición. **Ayuda:** es particularmente importante la división en subtareas. Realizar todas las operaciones auxiliares que sean necesarias. Como mínimo, se sugiere tener una función que retorne el alumno correspondiente al legajo, otra que elimine al alumno correspondiente al legajo, y una que agregue un alumno cualquiera al curso.

4. **aprobo(curso, legajo, tema)**, recibe un curso, un número de legajo y un tema y devuelve un booleano indicando si el alumno correspondiente al legajo aprobó ese tema. Nótese que un alumno puede contener más de una evaluación para el mismo tema, con haber aprobado una de ellas es suficiente para estar aprobado.
5. **vaBien(curso)**, recibe un curso y devuelve un booleano que indica si el curso tiene más de la mitad de alumnos aprobados. Para considerarse aprobado un alumno tiene que aprobar la totalidad de los temas del curso. Dividir en subtareas

#### Ejercicio 4

Utilizando el tipo **Celda** (práctica de registros), realice los siguientes ejercicios.

```
/* El tipo Fila es un renombre de Lista de Celda.  
El tipo Tablero es un renombre de Lista de Fila. */
```

1. Escriba una función **leerFila** que denote la lista de **Celda** (**Fila**) que conforman la fila actual.
2. Escriba una función **leerTablero** que denote la lista de **Fila** (**Tablero**) que representan al tablero en su estado actual.
3. Escriba una función **vaciarFila** que dado una **Fila f** retorne la fila con las celdas vacías. Para ello, utilice la función **celdaVacia** que denota una celda vacía.
4. Escriba una función **llenarFila** que dado una **Fila f** y una **Celda c** retorna la fila con todas sus celdas incrementadas en la cantidad de bolitas que establece **Celda c**. Implemente una función **sumarCelda** que dado dos celdas retorne una tercera con la suma de bolitas de ambas celdas para cada color.
5. Escriba una función **llenarFilas** que dado un **Tablero t**, una **Celda c** y una lista de números de fila **fs**, retorne el tablero donde cada fila fue llenada con la **Celda c**.
6. Escriba un procedimiento **DibujarTablero** que dado un **Tablero t**, vacíe el tablero y ponga bolitas en él de acuerdo a la información en **t**. Para ello, programe un procedimiento **DibujarFila** y haga uso de el procedimiento **PonerCelda** (definido en la práctica de registros).

#### Ejercicio 5

Considerar las siguientes estructuras de registros e implementar las operaciones solicitadas.

```
/* Representa una cantidad de bolitas, contando azules,
   negras, rojas y verdes, respectivamente. */
type Bolitas is record
{
    field azules      # número
    field negras      # número
    field rojas       # número
    field verdes      # número
}

/* Representa una cantidad de bolitas de un color */
type CantidadDeColor is record
{
    field color        # Color
    field cantidad     # número
}
```

1. **cantidadDeColorEnTablero(c)**: una función que retorne un registro **CantidadDeColor** con la cantidad total de las bolitas de color **c** en el tablero.
2. **regionVacia()**: una función que denote un registro **Bolitas** que representa una región vacía (i.e., con todos los campos en cero).
3. **sumaBolitas(r1, r2)**: una función que, dados dos registros de **Bolitas**, retorne un registro nuevo con la suma de las bolitas de ambas regiones.
4. **bolitasEnColumna()**: una función que retorne un registro **Bolitas** con la suma total por bolitas de la columna de la celda del cabezal. Resolverlo de dos formas: 1. reusando la función **cantidadDeColorEnColumna** 2. haciendo un recorrido del tablero mientras acumula el resultado en una variable de registro **Bolitas**. ¿Qué solución prefiere y por qué?
5. **listaCromaticaDeColumna()**: una función que retorne una **lista** cuyos elementos son registros de **CantidadDeColor**, donde cada registro tiene la cantidad de **un color** en la columna. Notar que en XGOBSTONES, la lista resultante tiene sólo cuatro elementos. Sin embargo, la función debe ser correcta *independientemente* de la cantidad de colores.
6. **bolitasEnColumna** y **listaCromaticaDeColumna** tienen propósitos similares. ¿Qué diferencias puede encontrar? ¿Qué habría que modificar en cada una si la cantidad de colores aumenta?
7. **sublistaCromaticaDeColumna(colores)**: una función que, dada una lista de colores **colores**, retorne la sublista de la **listaCromaticaDeColumna()** con aquellos registros que corresponden a los **colores** pasados como parámetro. ¿Se le ocurre cómo se podría implementar una función análoga usando registros de tipo **Bolitas**? Discuta las deficiencias de los registros a la hora de representar listas, incluso cuando la cantidad de elementos es conocida.
8. **listaBolitasEnCeldas()**: una función que retorne una lista cuyos elementos son registros de **Bolitas**, donde cada registro tiene la cantidad de bolitas de cada color de una celda.

9. `listaCromaticaDeCelda()`: una función que retorna una lista cuyos elementos son registros de `CantidadDeColor`, donde cada registro tiene la cantidad de bolitas de un color. (Esta función tiene que funcionar independientemente de la cantidad de colores que tiene XGOBTONES)
10. `listaCromaticaDeCeldas()`: una función que retorna una lista cuyos elementos resultan de aplicar `listaCromaticaDeCelda()` en cada una de las celdas del tablero. Notar que cada elemento de la lista es a su vez una lista de registros `CantidadDeColor`. Discutir la conveniencia de invocar una función para obtener cada elemento interno de la lista.

## 2. WhileMart

Los directivos del supermercado WhileMart desean optimizar el número de cajas de sus instalaciones, de tal modo que sean mínimas para reducir los costos, pero suficientes para evitar la acumulación y demora de los clientes. Para ello se dispone de la información de los clientes de una sucursal en un día típico, que se modela mediante los siguientes registros:

```
type Cliente is record
{
    field horaIngresoCola      # número
    field cantProductos        # número
}

type Caja is record
{
    field numero                # número
    field clientesEsperando    # Lista de Cliente
}

type Super is record
{
    field horaActual            # número
    field lineaDeCajas          # Lista de Caja
    field clientesComprando     # Lista de Cliente
}
```

Para cada cliente, se conoce la hora a la que empezó o empezará a hacer cola (`horaIngresoCola`) y la cantidad de productos que compró (`cantProductos`). Cada caja del supermercado se identifica por un número de caja (`numero`) e incluye una lista de los clientes que están haciendo cola en esa caja (`clientesEsperando`).

Para simular un supermercado, se cuenta con el registro `Super`, que indica la hora actual en la simulación del supermercado (`horaActual`), la lista de todas las cajas (`lineaDeCajas`) y la lista de todos los clientes que todavía no comenzaron a hacer cola (`clientesComprando`).

Se asumen los siguientes hechos sobre el funcionamiento del supermercado:

- Cuando un cliente termina de comprar y comienza a hacer cola, elige la caja en la que haya menos personas esperando. En caso de empate, elige la caja que tenga el menor número de caja.
- La lista de clientes esperando en una caja representa una cola. El primer elemento de la lista representa el cliente que está siendo atendido.

- Las cajas procesan un producto por unidad de tiempo.
- Cuando una caja termina de procesar todos los productos del cliente que está siendo atendido, dicho cliente se retira y se pasa al siguiente cliente esperando en la cola, si lo hay.
- Se asumirá siempre que las líneas de cajas no contienen cajas con números repetidos.

Se solicita resolver los siguientes ejercicios. Tener en cuenta que *ninguno* de ellos involucra el tablero de XGOBTONES.

### Ejercicio 6

Implementar `cajaMenosOcupada(líneaDeCajas)` una función que, dada una `líneaDeCajas`, denota la caja de que está *menos ocupada*, es decir, aquella en la que hay menos clientes haciendo cola. En caso de empate, se elige la caja que esté menos ocupada y que tenga el número más chico. Por ejemplo, si la caja 1 está atendiendo un cliente, y las cajas 2 y 3 están libres, se elige la caja 2. Puede suponer que `líneaDeCajas` tiene alguna caja.

### Ejercicio 7

Implementar `ingresaColaCliente(líneaDeCajas, cliente)` una función que, dada una `líneaDeCajas` y un `cliente`, denota la línea de cajas que se obtiene después de que el cliente comienza a hacer cola. El cliente se ubica en la caja que esté menos ocupada, de acuerdo con el criterio del ejercicio anterior. Puede suponer que `líneaDeCajas` tiene alguna caja.

### Ejercicio 8

Implementar `avanzaCaja(líneaDeCajas, nroCaja)` una función que, dada una `líneaDeCajas` y un número `nroCaja`, denota la línea de cajas que se obtiene después de que la caja identificada por `nroCaja` procesa un producto. Si la caja en cuestión no tiene clientes, la línea de cajas debe quedar tal como estaba. Si ya se procesaron todos los productos del cliente que está siendo atendido, dicho cliente se retira de la caja. Como precondición, puede suponer que hay una caja identificada por `nroCaja`.

**Ayuda:** se recomienda tener funciones que busquen y actualicen una caja de la línea, a partir de su número.

### Ejercicio 9

Implementar `avanzaLíneaDeCajas(líneaDeCajas)` una función que, dada una `líneaDeCajas`, denota la línea de cajas que se obtiene después de que todas y cada una de las cajas procesan un producto, con las mismas observaciones que en el ejercicio anterior.

### Ejercicio 10

Implementar `finalDeLasCompras(whileMart)` una función que, dado un super `whileMart`, denota el super que se obtiene de hacer que todos los clientes cuya `horaIngresoCola` coincide con la hora actual del supermercado ingresen a alguna cola. Notar que cada cliente que ingresa a una cola lo hace según el criterio ya establecido (elige la caja menos ocupada). Puede suponer la existencia de (o implementar) `clientesQueIngresanALas(clientes, hora)` y `clientesQueNoIngresanALas(clientes, hora)` que, dadas una lista de `clientes` y un número `hora`, retornan las listas de clientes que ingresan y no ingresan a una cola a dicha hora, respectivamente.

### Ejercicio 11

Implementar `pasoDelTiempo(whileMart)` una función que, dado un super `whileMart`, denota el supermercado después de que pase una unidad de tiempo. Lo que se debe hacer es, en orden:

- Hacer que los clientes que terminaron sus compras en la hora actual pasen a estar esperando en las cajas.
- Procesar un producto en cada una de las cajas de la línea de cajas.
- Avanzar el reloj una unidad de tiempo.

### Ejercicio 12

Implementar `supermercadoVacio(whileMart)` una función que, dado un super `whileMart`, denota Verdadero si el supermercado está completamente vacío. Es decir, si no hay clientes comprando y no hay clientes esperando en ninguna caja.

### Ejercicio 13

Implementar `horaEnQueQuedaVacio(whileMart)` una función que, dado un super `whileMart`, denota la hora en la que el supermercado queda completamente vacío. *Nota:* Para determinar ese valor, simular el paso del tiempo hasta que el supermercado quede vacío.

## 3. Tuberías

Una *tubería* consiste de una serie de *piezas*. Cada pieza tiene una longitud y una dirección. Ambos elementos se modelan en XGOBTONES de la siguiente manera:

```
type Pieza is record
{
    field tamaño          # número
    field dirección       # Dir
}
# Tuberia es un renombre de Lista de Pieza.
```

El campo `tamaño` indica la longitud de la pieza y `dirección` su dirección.

Resolver los siguientes ejercicios (NB: ninguno de ellos requiere el uso del tablero):

### Ejercicio 14

Escribir una función `pipesACoordenadas(tuberia, inicio)` que, dada una tubería y una coordenada de inicio (como en la práctica de registros), retorne la lista de coordenadas por las que pasan las piezas de la tubería comenzando en la coordenada de inicio.

### Ejercicio 15

Escribir la función `esTuberiaValida(tuberia, inicio)` que, dada una tubería y una coordenada de inicio, indique si la misma es *válida*, es decir si:

1. Todas las coordenadas por las que pasa la tubería son positivas; y
2. La tubería pasa a lo sumo una vez por cada coordenada.

### Ejercicio 16

Un *tramo* de una tubería es una secuencia de piezas consecutivas y con la misma dirección. La *longitud de un tramo* es la suma de las longitudes de sus piezas. Escribir `estandarizar-Tuberia(tuberia,longitud)`, una función que dada una `tubería` y un número `longitud`, retorne la tubería en la que se rearma cada tramo de longitud `t` de la `tuberia` dada como argumento, utilizando piezas de longitud `longitud`. Ello implica rearmar cada tramo colocando:

- `t div longitud` piezas de longitud `longitud` y
- una de `t mod longitud`.

## 4. Mini Twitter

El presente conjunto de ejercicios tiene por finalidad simular la operatoria de *Twitter*, una de las plataformas sociales más populares de la actualidad. Esta plataforma consiste de una comunidad de *usuarios* que envían mensajes llamados *tweets*. Los usuarios tienen *seguidores*, y son estos seguidores los que pueden ver los tweets de un usuario.

Para modelar la mencionada plataforma, se introducen los registros que a continuación se describen. Cada usuario contiene la identificación del mismo, la lista de tweets que envió hasta el momento y la lista de identificadores de sus seguidores. Cada tweet viene dado por la fecha y hora en que se emitió y por el contenido en sí<sup>1</sup>. Finalmente, la plataforma en sí contiene la lista de usuarios y un número que servirá a la hora de dar de alta nuevos usuarios.

```
type Usuario is record
{
    field id          # número
    field tweets      # Lista de Tweet
    field seguidores # Lista de número
}

type Tweet is record
{
    field fecha       # número
    field hora        # número
    field mensaje     # número
}

type Twitter is record
{
    field usuarios    # Lista de Usuario
    field proximoId  # número
}

type _ is record
{
    field             #
    field             #
}
```

---

<sup>1</sup>Para simplificar, el contenido de un mensaje es un número en lugar de texto.

Se solicita resolver los siguientes ejercicios. Tener en cuenta que *ninguno* de ellos involucra el tablero de XGOBTONES.

### Ejercicio 17

Implementar `altaDeUsuario(twitter)` una función que, dado un `twitter`, retorna el `twitter` que se obtiene de dar de alta un nuevo usuario. El usuario agregado no tiene seguidores, ni tweets, y su id está dado por el valor del campo `proximoId(twitter)`. Este campo debe ser incrementado una vez que el alta se efectiviza, de modo tal que el próximo usuario que se agregue no repita ese número de identificación.

### Ejercicio 18

Implementar `bajaDeUsuario(twitter, id)` una función que, dado un `twitter` y un número `id`, retorna el `twitter` que se obtiene de dar de baja el usuario con identificación `id`. Además de eliminar el usuario de la lista de usuarios, tener en cuenta que *también* debe eliminarse `id` de la lista de los usuarios a los que sigue. Como precondición, suponer que hay un usuario con identificador `id`.

### Ejercicio 19

Implementar `seguirA(twitter, idSeguidor, idASeguir)` una función que, dado un `twitter`, y dos números `idSeguidor` e `idASeguir`, retorne el `twitter` que se obtiene de hacer que usuario con identificador `idSeguidor` siga al usuario con identificador `idASeguir`. Como precondición, puede suponer que `twitter` contiene usuarios con ambos identificadores y que el usuario con identificador `idSeguidor` no sigue al usuario con identificador `idASeguir`.

### Ejercicio 20

Implementar `tweetear(twitter, idEmisor, fecha, hora, mensaje)` una función que, dado un `twitter`, un número `idEmisor`, una `fecha`, una `hora` y un `mensaje`, retorne el `twitter` que se obtiene de agregar un nuevo tweet a la lista de tweets del usuario con identificación `idEmisor`. Puede suponer que el usuario con identificador `idEmisor` existe.

### Ejercicio 21

Implementar `tweetsVisiblesPorUsuario(twitter, id)` una función que, dado un `twitter` y un `id`, retorne la lista de todos los tweets que son visibles por el usuario con identificador `id`. Recordar que los tweets visibles son aquellos de todos los usuarios que `id` está siguiendo. Como precondición, puede suponer que el usuario `id` existe.

### Ejercicio 22

Implementar `usuariosMasPopulares(twitter)` una función que, dado un `twitter`, retorne la lista con los identificadores de todos los usuarios más populares, es decir, aquellos que tienen más seguidores.

## 5. Dime DB

El objetivo de esta sección es programar DIME DB, una base de datos películas de que permite que los usuarios califiquen las películas que vieron. Las calificaciones se usan para recomendar otras películas a los usuarios de acuerdo a sus gustos.

La base de películas es simplemente una lista que contiene todas las películas conocidas.

```
# DimeDB es un renombre de Lista de Pelicula
```

A la vez, una película se compone por su título, el nombre de su director y la lista de calificaciones que recibió. Es importante remarcar que pueden haber muchas películas con el mismo título, pero ningún director ha dirigido dos películas con el mismo título. Con respecto a las calificaciones, un usuario puede calificar muchas veces a una película, pero DIME DB guarda únicamente la última calificación que se haga. De esta forma, un usuario puede “actualizar” la nota de una película en el sistema.

```
type Pelicula is record
{
    field titulo          # número
    field director        # número
    field calificaciones # Lista de Calificacion
}
```

Por último, una calificación indica qué nota (del 0 al 10) puso un usuario. Cuanto más le haya gustado la película, más alta será la nota.

```
type Calificacion is record
{
    field usuario      # número
    field nota         # número del 0 al 10
}
```

Se solicita resolver los siguientes ejercicios. Tener en cuenta que ningún ejercicio involucra el uso del tablero XGOBTONES.

### Ejercicio 23

Implementar `calificar(db, titulo, director, usuario, nota)` una función que, dada una base de películas `db`, el `título` y `director` de una película, un identificador de `usuario` y una `nota`, retorna la base de películas en la que `usuario` calificó la película `título` del `director` con la `nota` indicada en la base `db`. Recordar que DIME DB guarda únicamente la última calificación del `usuario` para la película. En caso que `usuario` ya hubiera calificado la película en `db`, la calificación previa debe quitarse. Puede suponer la existencia de (o implementar) la función `reemplazarPorTituloDirector(db, pelicula)` que retorna el `DimeDB` que se obtiene de reemplazar la película `titulo(pelicula)` del `director(pelicula)` por `pelicula`. Puede suponer que la película `título` del `director` existe en `db` y `nota` es un valor del 0 al 10.

### Ejercicio 24

Implementar `usuariosDB(db)` una función que, dado una base de películas `db`, retorna la lista de todos los usuarios que hayan calificado al menos una película en `db`, sin repetidos. Recordar la función `sinDuplicados(lista)` que, dada una lista, retorna la lista sin repetidos.

### Ejercicio 25

Implementar `gustaronAlUsuario(db, usuario)` una función que, dada una base de películas

`db` y un nombre de `usuario`, retorna la lista de **todas** las películas de `db` que le gustaron a `usuario`. Consideramos que a un usuario le gustó una película cuando la calificó con una nota mayor o igual a 8.

### Ejercicio 26

Implementar `gustaronDeLaPelicula(db, titulo, director)` una función que, dada una base de películas `db` y el `título` y `director` de una película, retorne la lista con todos los usuarios de `db` a los que le gustó la película `titulo` del `director`. Como precondición, puede suponer que la pelicula `título` del `director` existe en `db`.

### Ejercicio 27

Implementar la función `siTeGustoPuedeQueTambienTeGuste(db, titulo, director)` que, dada una base de películas `db` y el `título` y `director` de una película, retorna la lista de películas que les gustó a **todos** los usuarios a los que también le gustó la película `titulo` del `director` (excluyendo del resultado la película `titulo` del `director`). Notar que la lista resultante contiene **todas** las películas de `db` (salvo por la película `titulo` del `director`) en el caso en que a ningún usuario le haya gustado la película `titulo` del `director`. Como precondición, puede suponer que la pelicula `título` del `director` existe en `db`

## 6. XMail

XMAIL es un programa simple para gestionar los e-mails de un usuario. Cada usuario almacena de manera conjunta todos los e-mails, tanto enviados como recibidos. Asimismo, utiliza distintas *etiquetas* para poder buscar, filtrar y generar vistas de los e-mails. Las etiquetas no están predeterminadas por el sistema, sino que cada usuario puede agregar las etiquetas que quiera a sus mensajes. Incluso, un mensaje puede tener distintas etiquetas de acuerdo al gusto del usuario. Por ejemplo, un correo puede tener las etiquetas: “yo”, “trabajo”, “jefe”, “compañeros”, “asado fin de año”. Cada e-mail se modela en XGobstones con el siguiente registro.

```
type Correo is record
{
    field idRemitente      # número
    field idDestinatarios  # lista de número
    field etiquetas        # lista de número
    field mensaje           # numero
}
```

El campo `idRemitente` identifica al usuario que envió el e-mail, `idDestinatario` es una lista de usuarios a los que se envió el e-mail, `etiquetas` es la lista de todas las etiquetas de este e-mail, y `mensaje` es el mensaje del correo. Como invariante de representación, se sabe que las listas de etiquetas y destinatarios no tienen repetidos<sup>2</sup>.

Tener que etiquetar cada e-mail manualmente es un trabajo tedioso y virtualmente imposible para un usuario que recibe cientos de e-mails por día. Para simplificar esta tarea, XMAIL le permite al usuario definir *reglas* de etiquetado que aplica automáticamente a cada

---

<sup>2</sup>Notar que se acepta una lista vacía de destinatarios (esto es útil para mensajes borrador), y que un usuario se envíe un e-mail a sí mismo.

e-mail enviado o recibido. De esta forma, por ejemplo, un usuario puede agregar una regla que etiquete como “recibido” todos los e-mails que él haya recibido, simulando de esta forma la bandeja de entrada. Las reglas son muy simples en esta primera versión de prototipo de la herramienta y se modelan con el siguiente registro de XGobstones.

```
type Regla is record
{
    field esRemitente      # booleano
    field idUsuario        # número
    field etiqueta         # número
}
```

El campo **etiqueta** es la etiqueta que será aplicada a todos los mensajes que cumplan la regla. El campo **idUsuario** es un nombre de usuario cualquiera y **esRemitente** es un booleano que denota si la regla se aplica cuando **idUsuario** es el remitente o cuando es un destinatario. Cómo se aplica una regla a un e-mail depende del valor del campo **esRemitente**. Si **esRemitente** denota true, entonces se agrega la etiqueta de la regla al e-mail cuando el remitente del mismo corresponde al campo **idUsuario** de la regla. Caso contrario, se agrega la etiqueta de la regla al e-mail cuando *alguno* de los destinatarios corresponde con el campo **idUsuario** de la regla. Por ejemplo, la regla con campos **esRemitente** <- True, **idUsuario** <- `messi@hace.lio`, **etiqueta** <- `enviados` puede ser usada por el usuario con identificador `messi@hace.lio` para simular la carpeta de e-mails enviados. (**Aclaración:** en XGOBTONES, **idUsuario** y **etiqueta** son números; por cuestiones de exposición—y para hacer referencial al mundial—es que los escribimos de la manera usual.)

Finalmente, XMAIL administra la cuenta de un usuario que tiene los siguientes datos.

```
type Usuario is record
{
    field id              # número
    field correos         # lista de Correo
    field reglas          # lista de Regla
}
```

El campo **id** es el nombre del usuario, **correos** contiene todos los correos del usuario, tanto enviados como recibidos, y **reglas** contiene todas las reglas que han de aplicarse automáticamente para etiquetar mensajes. Como invariante de representación, la única restricción es que todos los correos sean del usuario (es decir que es su remitente o uno de sus destinatarios). Notar, en particular, que pueden haber reglas duplicadas.

Se solicita escribir las siguientes funciones. Tener en cuenta que ninguna de ellas involucra el uso de tableros.

### Ejercicio 28

```
function nuevoUsuario(id) que, dado un número id, crea un nuevo usuario con dicho id, sin correos y con dos reglas. La primer regla indica que todos los correos recibidos deben etiquetarse con 0 (inbox), mientras que la segunda regla indica que todos los correos enviados han de etiquetarse con 1 (enviado).
```

### Ejercicio 29

```
function etiquetasRemitente(reglas, idRemitente) que, dada una lista de reglas y un
```

número **idRemitente**, retorna la lista con las etiquetas que se obtendrían al aplicar cada una de las **reglas** en e-mails cuyo remitente sea **idRemitente**.

**Nota:** no importa si la lista de etiquetas retornadas contiene repetidos.

**Nota:** la lista de reglas podría contener reglas que se aplican por destinatario; obviamente estas reglas serán ignoradas por la función.

### Ejercicio 30

**function envioCorreo(usuario, destinatarios, mensaje)** que, dado un **usuario**, una lista de números **destinatarios** y un número **mensaje**, retorna el nuevo usuario que se obtiene luego de enviar un correo a los **destinatarios** con el **mensaje** indicado por el **usuario** del parámetro. Recordar que XMAIL aplica **todas** las reglas de etiquetado sobre este mensaje, y que ninguna etiqueta debe aparecer duplicada.

**Suponer** que hay una función **etiquetasDestinatario(reglas,idDestinatario)** que retorna la lista de etiquetas que se obtiene si se aplican las reglas sobre un e-mail que tiene a **idDestinatario** como uno de los destinatarios.

### Ejercicio 31

**function busquedaPorEtiquetas(usuario, etiquetas)** que, dado un **usuario** y una lista de **etiquetas**, retorne la lista de correos del **usuario** que contienen **todas** las **etiquetas** pasadas como parámetro.

**Suponer** la existencia de una función **incluido(lista, sublist)** que denota verdadero si **todos** los elementos de **sublist** pertenecen a **lista**. Por ejemplo, **incluido([1,2,3], [3,1])** denota verdadero, mientras que **incluido([1,2], [3])** denota falso.

### Ejercicio 32

**function eliminacionEtiquetas(usuario, etiquetas)** que, dado un **usuario** y una lista de **etiquetas**, retorna el **usuario** que se obtiene si se eliminan todos los e-mails del **usuario** parámetro que contienen **alguna** de las **etiquetas** pasadas como parámetro.

**Recordar** la función **interseccion** del ejercicio 26 de la práctica de listas.

### Ejercicio 33

**function etiquetas(usuario)** que, dado un **usuario**, retorna la lista de todas las etiquetas de los e-mails del **usuario**. La lista resultante no debe tener elementos repetidos.

### Ejercicio 34

**function importacionCorreos(usuario, correos)** que, dado un **usuario** y una lista de **correos**, retorna el **usuario** que se obtiene de agregar al **usuario** parámetro todos los **correos** luego de aplicar las reglas del **usuario** parámetro.

## 7. Vuela-vuela

El presente conjunto de ejercicios tiene por finalidad simular el tráfico aéreo, en la visión de una Torre de Control que lleva a cabo el seguimiento de cada nave. Una *Torre de Control* consiste de una lista de *vuelos* y una lista que asocia un *plan de vuelos* (o ruta) con cada par de la forma (origen, destino). La lista de vuelos va a ir variando a medida que los vuelos despegan y aterrizan. Sin embargo, la lista que asocia planes de vuelo para cada par (origen,

destino) siempre es la misma: si estando en origen se quiere ir a destino, entonces habrá un único plan de vuelo que permita conectar esas dos coordenadas.

A su vez, cada *vuelo* tiene un número de vuelo (único), un origen y un destino (que determinan el plan de vuelo a ejecutar), la etapa actual del plan de vuelo en la que se encuentra la nave y el estado. La etapa del plan de vuelo va entre 1 y la longitud del plan de vuelo correspondiente, mientras que el estado es un color que puede ser

- Saliente (Verde): avión listo para despegar.
- En ruta (Azul): avión en vuelo.
- Aterrizado (Negro): avión llegó a destino.

Por último, un *plan de vuelos* consiste de una lista de direcciones N, S, E u O. Los siguientes registros modelan estos datos.

```
type TorreDeControl is record
{
    field vuelos          # Lista de Vuelo
    field planesDeVuelo   # Lista de PlanDestino
}

type Vuelo is record
{
    field numero           # número
    field origen            # Coord
    field destino           # Coord
    field estado             # Color
    field etapaEnPlanDeVuelo # número
}

type PlanDestino is record
{
    field origen            # Coord
    field destino           # Coord
    field planDeVuelo       # Lista de Dir
}
/* Coord se definió en la práctica
de registros. */
```

Se solicita resolver los siguientes ejercicios. Tener en cuenta que ningún ejercicio involucra el uso del tablero GOBONES con la *excepción* del último.

### Ejercicio 35

Implementar *vueloEnEstado(vuelos, estado)* una función que, dada una lista de vuelos y un color que representa un estado, retorna el primer vuelo en la lista que tiene el estado dado como parámetro. Como precondición, hay al menos un vuelo con dicho estado.

### Ejercicio 36

Implementar *despegarVuelo(torreDeControl)* una función que, dada una torre de control, retorna la torre de control que se obtiene de despegar al primer vuelo en *vuelos(torreDeControl)* que esté en estado “listo para despegar”. Notar que para ello, debe cambiar el

estado del vuelo a “en ruta” e indicar que se encuentra en la etapa 1 del plan de vuelo. Como precondición, hay al menos un vuelo en la torre de control cuyo estado es “listo para despegar”.

**Ayuda:** se sugiere dividir en subproblemas. En particular, es conveniente tener funciones que permitan 1. eliminar un vuelo existente de la torre de control a partir de su número, 2. agregar un vuelo a la lista de control, y 3. actualizar un vuelo de la torre de control, haciendo uso de 1. y 2.

### Ejercicio 37

Implementar `aterrizarVuelo(torreDeControl)` una función que, dada una torre de control, retorna la torre de control que se obtiene de aterrizar el primer vuelo en `vuelos(torreDeControl)` que esté en estado “en ruta” y que haya llegado a destino (la etapa coincide con la longitud del plan de vuelo). Notar que para ello debe cambiar el estado del vuelo a “aterrizado”. Como precondición, hay al menos un vuelo en condiciones de aterrizar. **Ayuda:** se sugiere una función que retorne el plan de vuelos de un avión, y otra que retorne todos los aviones que llegaron a destino, independientemente de su estado.

### Ejercicio 38

Implementar `coordenadaFutura(torreDeControl, nro_vuelo, tiempo)` una función que, dada una torre de control, un número de vuelo y un número que representa un tiempo en etapas, retorne la coordenada sobre la que estará sobrevolando el vuelo correspondiente a `nro_vuelo` dentro de `tiempo` etapas, contando desde la etapa actual. Como precondición, puede suponer que hay un vuelo “en ruta” con el número de vuelo y la etapa actual más `tiempo` no supera la longitud del plan de vuelo.

### Ejercicio 39

Implementar `hayColisionEnEtapa(torreDeControl, tiempo)` una función que, dada una torre de control y un número que representa un tiempo en etapas, indique si hay aviones cuyo estado es “en ruta” que colisionaran dentro de `tiempo` etapas, contando desde la etapa actual. Dos aviones colisionan en una etapa si pasan por la misma coordenada en esta etapa.

### Ejercicio 40

Implementar `hayColision(torreDeControl)` una función que, dada una torre de control, indique si hay aviones con estado “en ruta” que colisionaran en alguna etapa futura. Tener en cuenta que basta considerar solo  $n$  etapas futuras, donde  $n$  es la máxima etapa posible (la longitud del plan de vuelos más largo posible).

**Nota:** El siguiente ejercicio involucra el uso del tablero.

### Ejercicio 41

Implementar `t.RenderRadar(torreDeControl)` un procedimiento que, dada una torre de control, dibuja en `t` la ubicación actual de cada vuelo “en ruta”. Los vuelos deben codificarse con tantas bolitas verdes como indica su número de vuelo. Cada origen con una bolita azul y cada destino con una bolita negra (el origen y destino debe codificarse a lo sumo una vez en el tablero). Puede asumir la existencia (o implementarla) de una función `enTablero(t, coord)` que indica si una coordenada cae dentro del tablero `t`, como así también el procedimiento `t.PonerEnCoord(coord, color)` que pone una bolita del color indicado en la coordenada indicada (se asume que la coordenada cae dentro del tablero `t`).

# Introducción a la Programación - Práctica 11

## Ejercicios Integradores

### CONSEJOS:

- Leer el enunciado en su totalidad y pensar en la forma de resolver el ejercicio ANTES de empezar a construir código.
- Si un ejercicio no sale, se puede dejar para después y continuar con los ejercicios que siguen.
- Los ejercicios están pensados para ser hechos después de haber mirado la teórica correspondiente.

### EJERCICIOS:

1. Los directivos del supermercado **GobsMart** nos piden modelar el funcionamiento de las cajas de cobro en una sucursal.

Para esto usaremos los siguientes tipos:

```
type TipoDePago is variant{
    /* PROP: modelar Tipos de pago aceptados */
    case Tarjeta
    case MartPago
    case Efectivo
}

type Producto is record{
    /* PROP: modelar productos
       INV.REP.: precio > 0      */
    field nombre      //String
    field marca      //String
    field precio     //número
}

type Cliente is record{
    /* PROP: modelar clientes */

    field dni         //DNI
    field tipoDePago  //TipoDePago
    field productos   //#[Producto]
}

type Caja is record{
```

```

/* PROP: modelar cajas
   INV.REP.: número > 0
              facturado >= 0      */

field número          //número
field clientesEsperando // [Cliente]
field aceptaPagos     // [TipoDePago]
field esRápida         //Bool
field facturado        //número
}

```

Realizar las siguientes tareas:

- 1) Armado de ejemplos para visualizar el modelo y sus datos.
- 2) Implementar la funciones:
  - a) cantidadDeClientesEsperandoEn\_(caja) que describe la cantidad de clientes que están esperando en la caja dada
  - b) cajaMenosOcupadaDe\_(cajas) que describe la caja con menos clientes esperando de la lista de cajas
  - c) gobsMart\_conIngresoDe\_aCaja\_(cajas,cliente,númeroDeCaja) describe una lista de cajas actualizada, donde el cliente dado ingresó en el número de caja dada.
  - d) gobsMart\_conIngresosDe\_(cajas,clientes) describe una lista de cajas actualizada, donde los clientes dados ingresó cada uno a la caja menos ocupada.
  - e) caja\_conPrimerоФacturado(caja) describe la caja dada pero donde ya facturó al primero de sus clientes. Se puede asumir que el cliente eligió bien la caja de acuerdo a su tipo de pago.
  - f) gobsMart\_conCliente\_cambiaACaja\_(cajas,dniCliente,númeroDeCaja) describe la lista de cajas actualizada, donde el el cliente con el dni dado, se cambia al número de caja dado. Se puede asumir que existe un cliente con el dni dado esperando en alguna de las cajas del gobsMart

## GOBSTAGRAM

```
type TipoDePerfil is variant {
    /* PROPÓSITO: modelar los tipos de perfil */
    case Empresa {}
    case Persona {}
    case Mascota {}
}

type Perfil is record {
    /* PROPÓSITO: modelar un perfil
       INV.REP.: nombre no es vacío
    */
    field nombre      // String
    field fotos       // [Foto]
    field tipo        // TipoDePerfil
    field seguidores // [String]
}

type Foto is record {
    /* PROPÓSITO: modelar una foto
       INV.REP.: id no es vacío
    */
    field id          // String
    field tamaño      // Número
    field likes       // [String]
}

tamañoDeFoto_
/* PROPÓSITO: Describe el tamaño de la foto con
   el identificador dado
PRECONDICIONES: ninguna
PARÁMETROS: un String que identifica una foto
RESULTADO: Número
*/
perfilDe_En_
/* PROPÓSITO: Describe el perfil del gobster dado
   en el gobstagram dado
PRECONDICIONES: el gobster está en el gobstagram
PARÁMETROS: * un String que identifica un Perfil
             * una Lista de Perfiles
RESULTADO: Perfil
*/
*/
```

Ejercicios:

- 1) Escribir una función **perfil\_ConFoto\_** que describa el perfil dado con una nueva foto con el identificador dado. La nueva foto debe tener su tamaño calculado, y no tener ni likes.
- 2) Escribir una función **galeríasDePerfilesDe\_** que describa una lista con las listas de fotos de cada perfil del gobstagram dado que tienen al menos una foto. Recordar que un gobstagram es una Lista de Perfiles.
- 3) Escribir una función **es\_LikerEn\_** que indica si el gobster con el nombre dado dio al menos un like a fotos del Perfil dado.
- 4) Escribir una función **mascotasLikersDe\_** que describa la lista de nombres de gobsters de tipo mascota que dieron likes a alguna foto del gobstagram dado.

# Introducción a la Programación – UNQ

Ejercicio Integrador – 14/11/2018 Sistema Electoral

Nos solicitan escribir una aplicación para dar soporte al proceso electoral de un distrito . De dicho proceso electoral participan un conjunto de personas, llamadas electores , que emiten sus votos en distintas mesas . Para saber en qué mesa vota cada elector, la Justicia Electoral **diseña un padrón, que es simplemente un listado indicando qué mesa le corresponde a cada elector**. En este sistema electoral, **los electores se identifican únicamente a través de su número de DNI** (documento nacional de identidad), **mientras que cada mesa se identifica únicamente con un número.**

En el día de elección, cada votante emite su voto en la mesa que tiene asignada. Para evitar que un elector vote más de una vez, una autoridad de mesa anota el número de DNI de cada elector que emite su voto. **Este registro se hace en el mismo orden en el que cada elector vota, a fin de guardar estadísticas.** El secreto del voto no se rompe porque, a diferencia de lo que ocurre con algunas boletas electrónicas, las boletas que usa la Justicia Electoral no tienen ningún tipo de identificación.

A pesar de las bondades teóricas del sistema de votación, la Justicia electoral quiere modernizarlo, a fin de minimizar las apariciones de electores en múltiples centros de votación, la no asignación de electores al padrón, y la verificación central de la unicidad del voto. Es por este motivo que nos encomendaron la tarea de programar un sistema para llevar a cabo la confección del padrón y el seguimiento de las elecciones.

La información del sistema electoral se almacena en la siguiente estructura de Gobstones.

```
// Distrito modela el sistema electoral. Ninguna de sus listas tiene
repetidos.

type Distrito is record {
    field mesas      //lista de números de mesa
    field electores  //lista de números de DNI
    field padrón     //lista de Asignación
    field votaron    //lista de números
}
```

En esta estructura, **mesas contiene la lista (sin repetidos) con todos los números de mesas del distrito, electores contiene la lista (sin repetidos) con los número de DNI de todos los electores registrados para votar, padrón es una lista de Asignaciones que se usa para saber en qué mesa vota cada elector, y votaron es una lista que contiene los número de DNI de todos los electores que ya emitieron su voto, en el orden en el que el voto fue emitido (sin repetidos).** Vale la pena remarcar que la Justicia Electoral garantiza que todas las asignaciones (ver abajo) de un elector en una mesa corresponden a DNIs y números que pertenecen a electores y mesas , respectivamente. Asimismo, **la Justicia garantiza que cada elector esté asignado a lo sumo a una mesa, y esta asignación, de existir, aparece una única vez en el padrón . Sin embargo, durante el uso del sistema podría ocurrir que algunos electores no estén asignados a ninguna mesa.**

Para representar cada asignación de un lector a una mesa del padrón, se utiliza el siguiente tipo de Gobstones

```
//Representa la asignación de una mesa para un elector
type Asignación is record{
    field mesa      // número que identifica una mesa
    field elector   // número que representa el DNI de un elector
}
```

### Ejercicio 1

Escribir la función `asignaciónCompleta(distrito)` que, dado un `distrito`, describa verdadero si todos los electores del mismo fueron asignados a una mesa. Describir lo que falte del contrato.

### Ejercicio 2

Escribir la función `cambioDeMesa(distrito, mesa, dni)` que, dado un `distrito`, un número de `mesa` y el número de `DNI` de una persona, describa un nuevo distrito que resulta de cambiar la asignación actual del elector correspondiente al `DNI` por una en la que el elector vote en `mesa`.

Precondición: `mesa` pertenece a `mesas(distrito)`, `dni` pertenece a `electores(distrito)` y la persona correspondiente al `DNI` está asignada a alguna mesa en `distrito`.

### Ejercicio 3

Escribir la función `votaronDeMesa(distrito, mesa)` que, dado un `distrito` y un número de `mesa`, describa una lista con los números de `DNI` de todos los electores asignados a `mesa` que ya emitieron su voto. Los números de lista resultante deben aparecer en el mismo orden que emitieron su voto.

Precondición: `mesa` existe en `distrito`

### Ejercicio 4

Escribir la función `votosPorMesas(distrito)` que, dado un `distrito`, describa una lista en la que cada elemento es una lista de todos los votantes de una mesa. Describir lo que falte del contrato.

### Ejercicio 5

Escribir la función `sonDNIsDeElectoresConsecutivos(distrito, mesa)` que, dado un `distrito` y un número de `mesa`, describe verdadero si los números de `DNI` de todos los electores de la mesa (hayan votado o no) forman alguna permutación de un rango. En otras palabras, si leemos los números de `DNIs` de menor a mayor, estos tienen que ser consecutivos.

Precondición: `mesa` existe en `distrito` y tiene electores asignados.

### Ejercicio 6

Escribir la función `mesaConMasVotosConsecutivos(distrito)` que, dado un `distrito`, describe el número de la mesa en la que se emitieron la mayor cantidad de votos consecutivos en un lapso de tiempo sin que se emitieran votos en ninguna otra mesa.

Nota: recordar que la lista `votaron` está ordenada cronológicamente según se emitieron los votos.

Precondición: Existe al menos una mesa en el `distrito`

Pregunta 1

Sin responder aún

Puntúa como 10,00

Estudiantes de semestres anteriores llevaron adelante un proyecto para manejar los platos de comida de restaurantes. De cada plato se conoce su nombre, el precio y las críticas recibidas, que se representan con una escala del 1 al 10. También existe una clasificación de los platos, que permiten diferenciar Entradas, Principal y Postre.

```
type Restaurante record {
    /* PROPÓSITO: Modelar un restaurante
     * INV.REP.: no puede haber 2 platos con el mismo nombre
     */
    field nombre // String
    field platos // [Plato]
}

type Plato is record {
    /* PROPÓSITO: Modelar un plato de comida
     * INV.REP.: nombre no puede ser vacío
     */
    field nombre // String
    field tipo // TipoDePlato
    field precio // Número
    field críticasRecibidas // [Número]
}

type TipoDePlato is variant {
    /* PROPÓSITO: Modelar los distintos tipos de platos*/
    case Entrada {}
    case Principal {}
    case Postre {}
}
```

Escribir la función **restorante\_Con\_DePrecio\_** que dado un restaurante, un plato y un precio, describe al Restaurante luego de agregarle ese plato inicializado con ese precio.

Tamaño máximo de archivo: 5MB, número máximo de archivos: 1

[Archivos](#)

Puede arrastrar y soltar archivos aquí para añadirlos

Tipos de archivo aceptados

Archivos de imagen .ai .bmp .gdraw .gif .ico .jpe .jpeg .jpg .pct .pic .pict .png .svg .svgz .tif .tiff

[◀ Clase de webvaluación 2 \(25 de noviembre 2020, 18:00hs\)](#)[Ir a...](#)[Consultas de práctica en vivo ►](#)



**Pregunta 2**

Sin responder aún

Puntúa como 10,00

Estudiantes de semestres anteriores llevaron adelante un proyecto para manejar los platos de comida de restaurantes. De cada plato se conoce su nombre, el precio y las críticas recibidas, que se representan con una escala del 1 al 10. También existe una clasificación de los platos, que permiten diferenciar Entradas, Principal y Postre.

```
type Restaurante record {
    /* PROPÓSITO: Modelar un restaurante
     * INV.REP.: no puede haber 2 platos con el mismo nombre
     */
    field nombre // String
    field platos // [Plato]
}

type Plato is record {
    /* PROPÓSITO: Modelar un plato de comida
     * INV.REP.: nombre no puede ser vacío
     */
    field nombre // String
    field tipo // TipoDePlato
    field precio // Número
    field críticasRecibidas // [Número]
}

type TipoDePlato is variant {
    /* PROPÓSITO: Modelar los distintos tipos de platos*/
    case Entrada {}
    case Principal {}
    case Postre {}
}
```

Para detectar las entradas que deben ser evaluadas, escribir la función `entradasSinCríticasDe_` que dado un restaurante, describe la lista de platos de tipo Entrada que no han recibido críticas aún. Para la resolución de esta operación NO está permitido el uso de repetición indexada.

◀ Clase de webvaluación 2 (25 de noviembre 2020, 18:00hs)

Ir a...

Consultas de práctica en vivo ►



**Pregunta 3**

Sin responder aún

Puntúa como 10,00

Estudiantes de semestres anteriores llevaron adelante un proyecto para manejar los platos de comida de restaurantes. De cada plato se conoce su nombre, el precio y las críticas recibidas, que se representan con una escala del 1 al 10. También existe una clasificación de los platos, que permiten diferenciar Entradas, Principal y Postre.

```
type Restaurante record {
    /* PROPÓSITO: Modelar un restaurante
     * INV.REP.: no puede haber 2 platos con el mismo nombre
     */
    field nombre // String
    field platos // [Plato]
}

type Plato is record {
    /* PROPÓSITO: Modelar un plato de comida
     * INV.REP.: nombre no puede ser vacío
     */
    field nombre // String
    field tipo // TipoDePlato
    field precio // Número
    field críticasRecibidas // [Número]
}

type TipoDePlato is variant {
    /* PROPÓSITO: Modelar los distintos tipos de platos*/
    case Entrada {}
    case Principal {}
    case Postre {}
}
```

Escribir la función `rebajasDe_PesosFijoEn_` que dada una cantidad y un restaurante, describe a ese restaurante luego de rebajar el precio en la cantidad dada de cada uno de sus platos principales; los demás platos no deben sufrir modificación.

[◀ Clase de webvaluación 2 \(25 de noviembre 2020, 18:00hs\)](#)[Ir a...](#)[Consultas de práctica en vivo ►](#)



**Pregunta 4**

Sin responder aún

Puntúa como 10,00

Estudiantes de semestres anteriores llevaron adelante un proyecto para manejar los platos de comida de restaurantes. De cada plato se conoce su nombre, el precio y las críticas recibidas, que se representan con una escala del 1 al 10. También existe una clasificación de los platos, que permiten diferenciar Entradas, Principal y Postre.

```
type Restaurante record {
    /* PROPÓSITO: Modelar un restaurante
     * INV.REP.: no puede haber 2 platos con el mismo nombre
     */
    field nombre // String
    field platos // [Plato]
}

type Plato is record {
    /* PROPÓSITO: Modelar un plato de comida
     * INV.REP.: nombre no puede ser vacío
     */
    field nombre // String
    field tipo // TipoDePlato
    field precio // Número
    field críticasRecibidas // [Número]
}

type TipoDePlato is variant {
    /* PROPÓSITO: Modelar los distintos tipos de platos*/
    case Entrada {}
    case Principal {}
    case Postre {}
}
```

Escribir la función **es\_ConPostreExquisito** que dado una restaurante, indica si tiene algún plato de tipo Postre, que cumple con la condición de ser considerado exquisito; esta condición se da cuando tiene más de 10 críticas mayores a 8.

[◀ Clase de webvaluación 2 \(25 de noviembre 2020, 18:00hs\)](#)[Ir a...](#)[Consultas de práctica en vivo ►](#)



**Pregunta 5**

Sin responder aún

Puntúa como 10,00

Estudiantes de semestres anteriores llevaron adelante un proyecto para manejar los platos de comida de restaurantes. De cada plato se conoce su nombre, el precio y las críticas recibidas, que se representan con una escala del 1 al 10. También existe una clasificación de los platos, que permiten diferenciar Entradas, Principal y Postre.

```
type Restaurante record {
    /* PROPÓSITO: Modelar un restaurante
     * INV.REP.: no puede haber 2 platos con el mismo nombre
     */
    field nombre // String
    field platos // [Plato]
}

type Plato is record {
    /* PROPÓSITO: Modelar un plato de comida
     * INV.REP.: nombre no puede ser vacío
     */
    field nombre // String
    field tipo // TipoDePlato
    field precio // Número
    field críticasRecibidas // [Número]
}

type TipoDePlato is variant {
    /* PROPÓSITO: Modelar los distintos tipos de platos*/
    case Entrada {}
    case Principal {}
    case Postre {}
}
```

Se necesita disponer información sobre ciertos platos de los que solamente se conocen sus nombres. Escribir la función **losPlatosConNombresEn\_DeLos\_**, que dada una lista de nombres y una lista de restaurantes, describa una lista de los Platos de los restaurantes dados que tengan cada uno de los nombres de platos dados. O sea, el resultado tendrá una lista de platos cumpliendo lo pedido por cada nombre dado, en el mismo orden.

[◀ Clase de webvaluación 2 \(25 de noviembre 2020, 18:00hs\)](#)[Ir a...](#)[Consultas de práctica en vivo ►](#)



# Enunciado “Goba es vos” - 2021s1

El juego “Goba es vos” es un juego de acertijos donde existen diferentes elementos que se pueden ordenar de formas diversas y cuyo objetivo es completar cada acertijo según un conjunto de reglas que pueden variar. Los elementos son de 2 **clases**: los **textos** y los **objetos**. Los **textos** pueden ser **nombres**, **conectivos** o **atributos** y permiten conformar **afirmaciones** (tanto en forma horizontal como en forma vertical). Las diferentes afirmaciones alteran las reglas del juego. Las afirmaciones más simples tienen la forma “nombre-conectivo-atributo”. Los **objetos** son elementos que resultan afectados por las reglas.

Por ejemplo, en el siguiente tablero se pueden observar cuatro tipos de objetos (Goba, una bandera, varias piedras y varias paredes) y cuatro afirmaciones, conformadas por los textos “Goba”, “Bandera”, “Pared”, “Piedra” (como nombres), “es” (como conectivo), “vos”, “ganar”, “frenar” y “empujar” (como atributos). Las afirmaciones son: “Goba es vos”, “Bandera es ganar”, “Pared es frenar” y “Piedra es empujar”.



0	1	2	3	4	5	6	7
7 1 	1 	2 			5 1 	6 	7 
6 			8 2				
5 4 	4 2	4 	4 2	4 	4 2	4 	4 
4 1 	1 		8 2				
3 			8 2			2 	
2 4 	4 	4 	4 	4 	4 	4 	8 1
1 						1 	
0 						8 	

Los diferentes elementos se grafican en el tablero según la siguiente representación.

- El número de bolitas negras indica la **clase** de elemento (o sea, si se trata de un **texto**, o de un **objeto**). Una bolita negra indica un texto, 2 bolitas negras indican un objeto.
- Para los textos, se distingue mediante bolitas de diferentes colores si se trata de nombres, conectivos o atributos, siendo la cantidad de bolitas indicativo del elemento específico. Los nombres se indican con bolitas azules, los conectivos con bolitas verdes y los atributos con bolitas rojas.
- Para los objetos se utilizan bolitas azules y rojas. Las bolitas azules indican cuál es el objeto en cuestión. Las bolitas rojas identifican atributos del objeto.

Los elementos utilizan las siguientes cantidades para identificarse (cuáles bolitas se utilizan depende del tipo de elemento, como se describió antes).

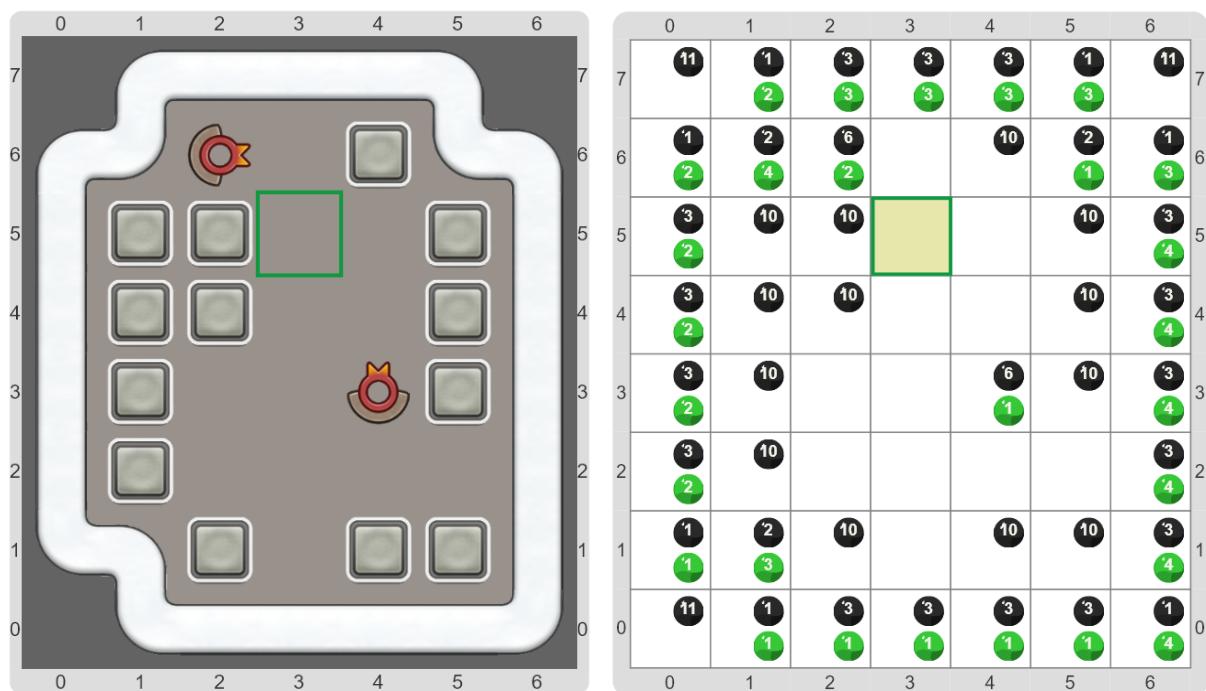
- Goba se representa con 1 bolita.
- La Bandera se representa con 2 bolitas.
- La Pared se representa con 4 bolitas.
- La Piedra se representa con 8 bolitas.
- El conectivo “es” se representa con 1 bolita.
- El atributo “vos” se representa con 1 bolita.
- El atributo “ganar” se representa con 2 bolitas.
- El atributo “frenar” se representa con 4 bolitas.
- El atributo “empujar” se representa con 8 bolitas.

# Primer parcial - 2021s1

## Parte 2

El juego *Quell* es un juego de acertijos donde el jugador maneja una burbuja en un escenario que contiene diversos elementos. Algunos de los elementos que se pueden encontrar en el tablero son **bordes**, **paredes**, **trampas** y **dispositivos**. Los bordes pueden ser **esquinas**, **ángulos** o **límites** rectos. Las trampas pueden ser **espinas**, **multiespinas** o **espinas rotables**. Los dispositivos pueden ser **interruptores**, **portales** o **compuertas**. Algunos de estos elementos tienen diferentes atributos, según su naturaleza. Por ejemplo, los bordes, las espinas, las espinas rotables y los interruptores tienen una dirección que indica hacia dónde apunta el elemento. Las compuertas tienen un atributo que indica si están abiertas o cerradas, y los portales tienen atributos que codifican su identidad y portal de destino.

En el siguiente tablero podemos encontrar un escenario básico de *Quell*, donde podemos observar una serie de bordes que conforman un espacio cerrado, algunas paredes, y un par de espinas. Notar que los bordes apuntan hacia adentro de la zona de juego.



Los diferentes elementos se grafican en el tablero según la siguiente representación.

- *El tipo de elemento se codifica con bolitas del color dado por la función tipo.* La cantidad depende de qué elemento se trate.
- Las direcciones se codifican con bolitas del color dado por la función dirección. La cantidad depende de qué dirección se trate, según el código dado por la función códigoDeDir\_
- Los elementos utilizan diferentes cantidades para identificarse. Para cada elemento, existe una función que indica la cantidad de bolitas a utilizar. Así, existen funciones esquina, ángulo, límite, espinas, etc.

# Introducción a la Programación

Recuperatorio 2 - 2021s1 - Gobwarts

Luego de recibir un encargo para programar un juego basado en las historias de Harry Potter, se decide usar Gobstones para hacerlo. Esta evaluación cubre algunas partes de la tarea encargada. Para modelar los elementos de este dominio se utilizan los siguientes tipos. Al modelar una lista de magos debe suponerse que no existen dos magos con el mismo nombre en la misma.

```
type Mago is record {
    /* PROPÓSITO: modelar un mago de
       Hogwarts
    INV.REP.:
        * el nombre no es vacío
        * no hay dos habilidades con la
            misma denominación. */
    field nombre      // String
    field casa        // Casa
    field puntos      // Número
    field rol         // Rol
    field habilidades // [ Habilidad ]
}

type Habilidad is record {
    /* PROPÓSITO: modelar una habilidad
    INV.REP.:
        * la denominación no es vacía
        * no hay dos características
            iguales. */
    field denominación // String
    field disciplina   // Disciplina
    field nivel         // Número
    field características // [ String ]
}

type Casa is variant {
    /* PROPÓSITO: Modelar las casas de
       Hogwarts. */
    case Griffindor {}
    case Slytherin {}
    case Ravenclaw {}
    case Hufflepuf {}
}

type Rol is variant {
    /* PROPÓSITO: Modelar los distintos
       tipos de roles. */
    case Director {}
    case Profesor {}
    case Estudiante {}
    case Personal {}
}

type Disciplina is variant {
    /* PROPÓSITO: Modelar los distintos
       tipos de disciplinas.*/
    case Transformaciones {}
    case Encantamientos {}
    case ArtesOscuras {}
    case Pociones {}
    case Herbología {}
    case CriaturasMágicas {}
    case Otra {}
}
```

Las habilidades de un mago combinan sus diferentes magias, pociónes, criaturas que domina y otras habilidades (como juegos o dominio de objetos). Además cada mago comienza con 0 puntos para su casa, y puede ganar o perder puntos según las acciones que desarrolle a lo largo de su estadía en la Escuela.