Paradigmas de Programación

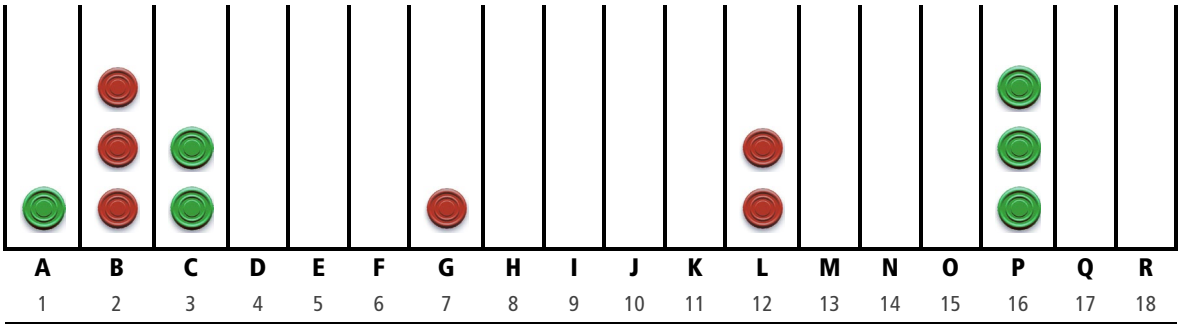
Práctica I – Curso 2024/25

Pargammon

1. Objetivo y Descripción General

El objetivo de la práctica es el desarrollo de una aplicación en lenguaje Python que permita **jugar y evaluar estrategias** para un **nuevo juego de tablero** denominado **Pargammon** en un entorno de texto (no gráfico).

El Pargammon es un mestizaje del juego del Parchís y del Backgammon, con reglas simplificadas. Se juega en un tablero de **N** columnas etiquetadas con letras mayúsculas del alfabeto anglosajón (N < 27), cada jugador dispone de **M** fichas iniciales y en cada jugada se tiran **D** dados en secuencia que indican el desplazamiento de la ficha elegida por el jugador. Los valores habituales de estos parámetros son N =18, M = 6 y D = 3 pero otras combinaciones pueden ser válidas. El número de jugadores es de dos, pero se está estudiando la posibilidad de lanzar versiones del juego con más jugadores[1]. En la siguiente imagen se muestra un posible estado del tablero en un momento dado de una partida (juegan los rojos, se muestra la secuencia de dados):



Dados: ⚀ ⚄ ⚂

Fig. 1

1.1. Reglas generales

[R1] Cada columna puede tener varias fichas apiladas, pero deben ser siempre del **mismo jugador**.

[R2] Al comienzo del juego todas las fichas del primer jugador se encuentran en la columna **A** y todas las del segundo jugador en la columna **B** (y si hubiera más jugadores se seguiría el mismo patrón)

[R3] Los jugadores alternan turnos. En cada turno se tiran **D** dados (valores al azar del 1 al 6), que se procesan de forma **secuencial**: Para cada dado el jugador primero decide **si lo va a usar o no** para mover una ficha. Si no lo usa se pasa al dado siguiente, en caso contrario indica la columna donde está la ficha que se va a mover, **se realiza el movimiento** (que consiste en desplazar la ficha hacia delante el valor que indique el dado) y se pasa a procesar el siguiente dado.

[R4] El **ganador** de la partida es el primer jugador que consigue **sacar** todas sus fichas del tablero. Para **sacar** una ficha del tablero se debe realizar un movimiento que lleve esa ficha a la (hipotética) columna que **está justo después de la última**. En el ejemplo sería la hipotética columna **S,** la columna (N+1)-ésima. Si el movimiento llevase la ficha más lejos entonces no sería válido.

1.2. Movimientos válidos

Un movimiento se indica o bien con el carácter **@**[2] o bien con la letra de una columna. Si se usa el carácter **@** significa que ese dado **se descarta** y no se va a usar para mover ficha.

Si se usa la letra de una columna su significado es que se debe coger la ficha superior de esa columna y desplazarla **hacia la derecha** tantas posiciones como indique el dado, para colocarla en la columna destino. En este movimiento no importa el contenido de las columnas intermedias (se puede suponer que la ficha *vuela* por encima de ellas), tan solo importa el contenido de la columna destino.

[1] Como se explica más adelante, el diseñar vuestro programa de forma que la adaptación a más de dos jugadores sea lo más sencilla posible no es algo obligatorio, sino una opción que puede proporcionar puntos extra.

[2] En el código ASCII la letra @ está justo antes de la A. Esto simplifica ligeramente las cosas al tratar este caso en el código.

Por lo tanto un movimiento es un carácter. Para que se considere **válido** se deben cumplir las siguientes condiciones:

[M1] El carácter identifica a una columna del tablero (la columna origen) o es el carácter @ (no hacer nada).

[M2] La columna origen debe contener una o más fichas del jugador que tiene el turno (se moverá la ficha superior)

[M3] La columna destino está en el tablero o es justo la siguiente a la última (en este caso se **sacaría** la ficha)

[M4] Si la columna destino está en el tablero (no se saca ficha), la columna destino debe o bien estar vacía, o bien contener fichas del jugador, o bien contener **una única ficha** de otro jugador.

1.3. Captura de piezas contrarias

Si se realiza un movimiento a una columna que contiene exactamente una ficha del jugador contrario, esa ficha se **captura**, lo que provoca que la ficha capturada se mueva a **la primera columna del tablero que esté vacía o contenga piezas del jugador de la ficha capturada.** Fijaros que de esta forma se respeta la regla de que cada columna solo contenga fichas de un mismo jugador.

Se puede suponer que siempre es posible encontrar una columna donde mover la ficha capturada, no es necesario que vuestro programa lo compruebe. Eso sí, es perfectamente posible (aunque poco probable) que la ficha capturada acabe en una columna **posterior** a la que estaba (no hay ningún problema con esa situación).

La captura es básicamente el único elemento de estrategia que existe en este juego, la idea es que conviene crear apilamientos de nuestras fichas para evitar que sean capturadas (retrocediendo posiciones) por el contrario.

1.4. Jugadas válidas

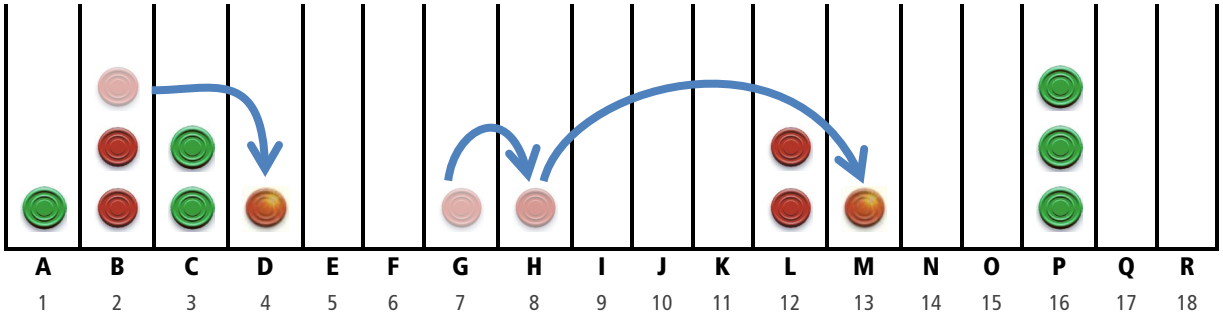
Una **jugada** es una secuencia de **D** movimientos (uno por cada dado). Como se ha visto en 1.2, un movimiento es o bien el carácter @ o la letra de una columna, por lo tanto una jugada es un string de D caracteres. Para que una jugada sea válida se tiene que cumplir:

[J1] Su longitud debe ser exactamente D caracteres.

[J2] Cada uno de sus movimientos (con el dado que le corresponda) debe ser válido.

[J3] La jugada con todos los caracteres **@** (no hacer nada) solo es válida si no existe ninguna otra jugada válida disponible.

Por ejemplo, en la **Fig. 1** de la página anterior juegan los rojos y la secuencia de dados es **1-5-2**. Fijaros que ninguna jugada puede comenzar con la columna **B**, porque eso supondría mover una ficha roja una posición a la derecha (columna **C**) y esa columna está ocupada por fichas del otro jugador. Si la jugada fuera **GHB** el resultado sería el siguiente:



Si la jugada fuera **GLQ** entonces el jugador rojo sacaría una ficha del tablero, ya que acabaría justo en la columna posterior a la última (la columna **S**, si existiera). Pero la jugada **LMR** no sería válida, ya que la ficha acaba dos posiciones después de la última (regla **[R4]**). Como referencia, las jugadas válidas para el juego de los rojos en la Fig. 1 son las siguientes:

@@B, @@G, @@L, @B@, @BB, @BG, @BL, @G@, @GB, @GL, @L@, @LB, @LG, @LL, @LQ, G@@, G@B, G@H, G@L, GB@, GBB, GBG, GBH, GBL, GH@, GHB, GHL, GHM, GL@, GLB, GLH, GLL, GLQ, L@@, L@B, L@G, L@L, L@M, LB@, LBB, LBG, LBL, LBM, LG@, LGB, LGL, LGM, LL@, LLB, LLG, LLM, LLQ, LM@, LMB, LMG, LML

2. Descripción Detallada

El objetivo de esta práctica es la de crear una aplicación Python en modo consola que permita jugar al Pargammon y también facilite el analizar características de su juego. La práctica se organiza en bloques que indican las capacidades del programa. Cada bloque amplía las capacidades del anterior, y (de forma aproximada) se corresponde a una parte de la nota de la práctica. Solo es obligatorio el primer bloque (y los requisitos obligatorios, claro):

1. **Juego Manual** (hasta 3 puntos): Con este bloque dispondremos de un programa que permita jugar al Pargammon a dos jugadores humanos, pidiendo las jugadas por teclado y verificando su corrección, y mostrando el estado y desarrollo de la partida.
2. **Juego Automático** (hasta 4 puntos): En este bloque añadiremos la capacidad de que alguno o ambos de los jugadores estén controlados por la máquina. Existirán dos tipos de jugadores automáticos: La máquina tonta escoge una jugada al azar de entre las jugadas posibles que sean válidas, y la máquina lista escoge la mejor jugada de entre las disponibles.
3. **Opción de Deshacer** (hasta 2 puntos): Se añade la opción de deshacer las jugadas realizadas.
4. **Múltiples Jugadores** (hasta 2 puntos): El programa está adaptado a la posibilidad de que existan múltiples jugadores. La puntuación de este bloque solo se tendrá en cuenta si se han desarrollado todos los anteriores, y puede proporcionar hasta un punto extra (si se realizan perfectamente todos los bloques es posible obtener una nota de 11.5 sobre 10 en la práctica).

2.1. Requisitos obligatorios

Para facilitar el análisis automatizado y las modificaciones solicitadas en la defensa, vuestro programa debe responder al siguiente esquema:

```python
from random import seed, randrange, choice

AZAR = 75  # Semilla del generador de números aleatorios

# … Otras constantes, funciones y clases …

class Pargammon(object):
    def __init__(self, n=18, m=6, d=3, fichas=('\u263a', '\u263b')):
        self.N = n            # Número de columnas
        self.M = m            # Número inicial de fichas
        self.D = d            # Número de dados
        self.FICHAS = fichas  # Caracteres de las fichas de cada jugador
        # … Resto de código de inicialización …

    def __repr__(self) -> str:
        """
        :return: Un string que indica el tablero y estado de la partida
        """
        # …

    def cambiar_turno(self) -> bool:
        """ Cambia de turno. Devuelve True si es fin de partida, False si no """
        # …
        # La tirada de dados se debe realizar con esta línea:
        self.dados = [randrange(6) + 1 for _ in range(self.D)]
        # …

    def jugar(self, txt_jugada: str) -> None | str:
        """ Intenta realizar la jugada indicada en el string txt_jugada
        :return: None si es válida o un string con un mensaje de error
        """
        # …
```

(*continúa en la siguiente página*)

```
def main():
    seed(AZAR)
    print("*** PARGAMMON ***")
    params = map(int, input("Numero de columnas, fichas y dados = ").split())
    juego = Pargammon(*params)
    # …
```

Los requisitos son los siguientes:

- El programa debe residir en un único fichero, **practica1.py**.
- Las única librería que se puede importar es **random**. Si desea utilizar alguna otra primero debe consultar al profesor.
- Tal como se indica en el esquema, debe definirse una clase, **Pargammon**, cuyo objetivo es almacenar y modificar el estado de la partida, mediante los métodos **jugar** y **cambiar_turno**. Pueden existir otros métodos auxiliares y clases extra que sirvan de apoyo, pero el modo concreto en que se juega una partida debe implementarse en la función **main**.
- Para conseguir que los resultados sean reproducibles se usa una semilla concreta (constante AZAR) para el generador de números aleatorios. Todas las operaciones con números aleatorios deben realizarse tal como se indican en el esquema o en apartados posteriores (En el Juego Básico la única de estas operaciones es la tirada de dados).
- El parámetro **fichas** de Pargammon indica los caracteres que van a representar las fichas de los jugadores. Por defecto se usan dos caracteres Unicode (☺ y ☻), pero se pueden cambiar por cualquier otro juego de caracteres que elijáis.

2.2. Juego manual

El objetivo de este primer bloque es obtener un programa que permita jugar una única partida de Pargammon a dos jugadores humanos. Para completar la función **`main`** del esquema hay que añadir el bucle principal del juego, que se repite hasta que se termine la partida (cuando el jugador actual haya sacado todas sus fichas del tablero). En el bucle principal se hacen tres cosas:

1. Imprimir el estado del juego (ejecutando **`print(juego)`**, que escribe la cadena devuelta por el método **`__repr__`**)
2. Pedir la jugada del jugador que tenga el turno en ese momento. Como el usuario puede introducir una jugada no válida, esta tarea debe implementarse mediante un bucle que solo termine cuando haya introducido una jugada correcta. La detección de si una entrada es válida o no la realiza el propio método **`jugar`**, que devuelve o bien `None` (jugada correcta) o un string con el mensaje de error que hay que mostrar por pantalla[3].
3. Cambiar de turno, llamando al método **`cambiar_turno`**. Este método también debe detectar si se ha terminado la partida (devuelve un valor booleano), para que el bucle principal pueda terminar.

Por último, cuando termina la partida se debe escribir por pantalla un mensaje indicando quien ha ganado.

Al escribir el código de los métodos de la clase Pargammon se debe tener en cuenta lo siguiente:

- Se pueden crear nuevos métodos y clases auxiliares, pero desde **`main`** solo se puede llamar a **`jugar`** y **`cambiar_turno`** (e implícitamente a **`__init__`** y **`__repr__`**).
- El método **`__repr__`** debe proporcionar una cadena que al mostrarse por pantalla tenga la misma apariencia e información que la del ejemplo que se muestra en la siguiente página (salvo los caracteres que representan a las fichas y a los dados, que pueden ser distintos).

[3] La forma en que devuelven valores los métodos `jugar` y `cambiar_turno` no son las más adecuadas desde el punto de vista del diseño, ya que realizan tareas extra (traducir un error a texto en el caso de `jugar` y detectar fin de partida en el caso de `cambiar_turno`) que no son consustanciales a su tarea principal, por lo que la cohesión de ambos métodos es débil. Pero se ha considerado que es más importante el tener un esquema lo más minimalista posible para que podáis enfocaros directamente en el problema planteado.

- El método **`jugar`** debe comprobar que la jugada sea válida según las reglas indicadas en las secciones 1.2 y 1.4, **con la excepción de la regla [J3]** (jugada vacía).
- Al detectar jugadas no válidas podéis elegir entre tres alternativas según el detalle con que **`jugar`** devuelve el mensaje de error: Especificar tipo de error y el movimiento concreto donde se ha detectado, tal como aparece en el ejemplo (esta alternativa suma ½ punto extra), el indicar únicamente el tipo de error (ni suma ni resta puntos), o simplemente devolver una única cadena "Jugada errónea" en todos los casos (resta ½ punto).

A continuación se muestra un ejemplo de partida completa (en azul las entradas del usuario):

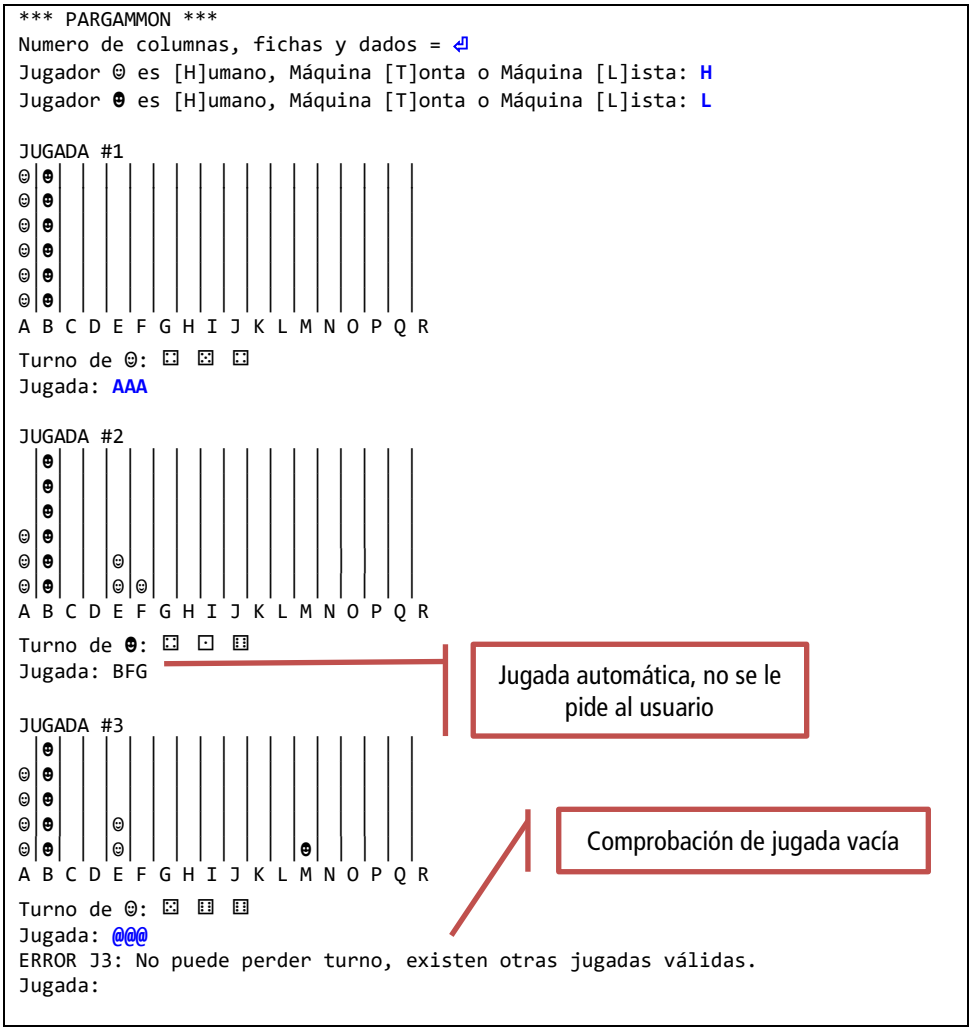
```
*** PARGAMMON ***
Numero de columnas, fichas y dados = 10 5 3

JUGADA #1
☺|☻| | | | | | | | |
☺|☻| | | | | | | | |
☺|☻| | | | | | | | |
☺|☻| | | | | | | | |
☺|☻| | | | | | | | |
A B C D E F G H I J
Turno de ☺: ⚃ ⚃ ⚃
Jugada: HOLA
ERROR J1: Debe indicar exactamente 3 movimientos.
Jugada: HUY
ERROR J2-M1: No existen columna(s) con estas letras: Y, U.
Jugada: ACA
ERROR J2-M2: Columna de origen C no tiene fichas del jugador.
Jugada: AAA

JUGADA #2
  |☻| | | | | | | | |
  |☻| | | | | | | | |
  |☻| | | | | | | | |
☺|☻| | |☺| | | | | |
☺|☻| | |☺|☺| | | | |
A B C D E F G H I J
Turno de ☻: ⚂ ⚀ ⚁
Jugada: BBC

JUGADA #3
☺|☻| | | | | | | | |
☺|☻| | |☺| | | | | |
☺|☻| | |☺|☻| | |☻| |
A B C D E F G H I J
Turno de ☺: ⚂ ⚁ ⚁
Jugada: AEE

JUGADA #4
  |☻| | | | | | | | |
  |☻| | | | | | | | |
☺|☻| | | | | | | | |
☺|☻| | | |☺| | |☻| |
A B C D E F G H I J
Turno de ☻: ⚂ ⚁ ⚁
Jugada: BBB

JUGADA #5
☺| | | | | | | | | |
☺| | | | | | |☻| | |
☺|☻| | | |☻| |☻|☻| |
A B C D E F G H I J
Turno de ☺: ⚃ ⚄ ⚀
Jugada: AAG
ERROR J2-M4: Movimiento G -> H, columna destino tiene más de una ficha contraria.
Jugada: AGJ

JUGADA #6
☺| | | | | | |☻| | |
☺|☻| | | |☻| |☻|☻| |
A B C D E F G H I J
Turno de ☻: ⚃ ⚄ ⚀
Jugada: FHH
ERROR J2-M3: Movimiento F -> L, columna destino fuera de rango.
Jugada: BHH

JUGADA #7
☺| | | | | | | |☻| |
☺| | | | | |☻| |☻|☻|
A B C D E F G H I J
Turno de ☺: ⚄ ⚄ ⚄
Jugada: ADG

JUGADA #8
  | | | | | | | |☻| |
☺| | | | | |☻| |☻|☻|
A B C D E F G H I J
Turno de ☻: ⚃ ⚃ ⚄
Jugada: @@H

JUGADA #9
  | | | | | | | |☻| |
☺| | | | | |☻| | |☻|
A B C D E F G H I J
Turno de ☺: ⚄ ⚃ ⚄
Jugada: @AG

Han ganado los ☺!
```

2.3. Juego automático

En este bloque incorporamos la opción de que alguno o todos los jugadores estén controlados por la máquina. Por tanto la primera modificación a realizar es que antes del comienzo del juego se debe pedir al usuario que indique para cada jugador si va a estar controlado por el usuario, por una máquina tonta o por una máquina lista.

En el bucle de juego se comprueba si el jugador que tiene el turno esta controlado o no por el usuario. Si lo está entonces se pide la jugada al usuario, en caso contrario la jugada se realiza automáticamente de la forma que se indica posteriormente.

Ejemplo de ejecución:



Para implementar el juego automático habrá que crear uno o varios métodos en la clase Pargammon para calcular y almacenar la lista de **todas las jugadas válidas posibles** para el jugador que tiene el turno.

Para calcular la lista de todas las jugadas válidas posibles se debe utilizar **recursividad**, ya que no conocemos a priori el número de dados que se van a utilizar ni se puede suponer que el número de dados, **D**, tiene un valor máximo concreto.

Es importante darse cuenta de que al generar una jugada añadiéndola movimientos, los movimientos válidos van a depender los movimientos realizados anteriormente, por lo que es necesario ir modificando la estructura de datos que almacena el contenido del tablero, y esas modificaciones se deben **deshacer** al ir considerando las distintas alternativas: Por lo tanto puede ser ventajoso el desarrollar este bloque teniendo en cuenta los requisitos del siguiente, donde se pide poder **deshacer jugadas**.

Una vez que se dispone de la lista de todas las jugadas posibles es sencillo implementar el juego en modo automático:

- La **máquina tonta** simplemente escoge una jugada al azar (usando `random.choice`)
- La **máquina lista** escoge la **mejor jugada** de la lista.

Para saber cual es la mejor jugada hay que definir un método para valorar las jugadas: El **valor de una jugada** se calculará como la **puntuación del jugador** actual **menos** la puntuación del otro jugador (o la suma de los puntos de los otros jugadores, en el caso de múltiples jugadores) **si se realizara esa jugada**. Cuanto más alto sea su valor mejor, ya que eso significa que el jugador actual obtendría una mayor ventaja respecto al resto de los jugadores.

Para definir la función que nos da la puntuación de un jugador en un estado concreto de la partida vamos a tener en cuenta la posición de sus fichas (cuanto más a la derecha mejor), si están apiladas o no, y el hecho de que conviene sacar cuanto antes fichas del tablero (para evitar que sean capturadas).

La **puntuación de un jugador** se define como la **suma de los valores de cada una de sus fichas**. El **valor de una ficha** es un **factor** multiplicado por su **posición** (el índice *1-based* de la columna donde se encuentra o bien N+1 si es una ficha sacada del tablero). Y el **factor** de una ficha es un número entero, 1 si la ficha es la única de su columna, 2 si está apilada con otras y 3 si es una ficha sacada del tablero.

Para aquellos que prefieran una descripción matemática, el valor de una jugada viene dado por la siguiente fórmula, donde j representa un jugador y $\mathbb{J}$ representa al conjunto de jugadores:

$$v_j = p_j - \sum_{\substack{\forall\, k \in \mathbb{J} \\ k \neq j}} p_k = 2p_j - \sum_{\forall\, k \in \mathbb{J}} p_k$$

Y la puntuación de un jugador se calcula como:

$$p_j = 3 \cdot (N+1) \cdot \mathcal{S}_j + \sum_{\forall c \in \mathbb{F}_j} \mathcal{A}(c) \cdot \mathcal{I}(c) \cdot \mathcal{N}(c)$$

Donde la variable c representa a una columna concreta del tablero, N es el número de columnas, $\mathbb{F}_j$ es el conjunto de columnas del tablero que contienen fichas del jugador j y el resto de funciones y términos se definen así:

- $\mathcal{I}(c)$ es el índice (*1-based*) de la columna c en el tablero (A → 1, B → 2, etc.)
- $\mathcal{N}(c)$ es el número de fichas que contiene la columna c
- $\mathcal{A}(c) = \begin{cases} 1 & \text{si } \mathcal{N}(c) = 1 \\ 2 & \text{si } \mathcal{N}(c) > 1 \end{cases}$ es el factor de apilamiento de las fichas en la columna
- $\mathcal{S}_j = M - \sum_{\forall c \in \mathbb{F}_j} \mathcal{N}(c)$ es el número de fichas que ha sacado el jugador j del tablero (M es el número de fichas iniciales)

Nota: Al implementar este bloque se debe añadir al código de `jugar` la comprobación de la regla **[R3]** (jugada vacía), ya que al disponer de la lista de jugadas posibles es factible la comprobación de si la única jugada disponible es el saltar turno.

2.4. Deshacer jugadas

En este bloque se tiene que añadir la posibilidad (para un jugador humano) de deshacer un número cualquiera de jugadas. La forma de indicarlo va a ser muy sencilla: Si el jugador humano en vez de una jugada proporciona una cadena de asteriscos, se entiende que quiere deshacer el número de jugadas dado por la longitud de la cadena. Hay que tener en cuenta las siguientes condiciones:

- Se puede suponer que el usuario siempre va a proporcionar o bien una cadena formada únicamente por asteriscos o una jugada. No es necesario realizar ninguna comprobación, si el primer carácter es un asterisco entonces estamos en el caso de deshacer jugadas. Tampoco es necesario comprobar el caso en que se pide deshacer más jugadas de las realizadas.
- Las jugadas a deshacer no distinguen entre jugadores humanos y automatizados: Si esta jugando un humano contra la máquina y el jugador introduce un único asterisco, entonces se deshará únicamente la jugada anterior (de la máquina), y como ahora el turno es de la máquina, volverá a realizar una jugada automáticamente y el turno volverá al jugador.
- **Importante**: Al deshacer jugadas, la jugada a la que se retrocede debe tener **la misma secuencia de dados** que tenía originalmente. A partir de ese momento si que es posible que la secuencia de dados difiera.

En la página siguiente se muestra un ejemplo del uso de esta opción:

```
*** PARGAMMON ***
Numero de columnas, fichas y dados = 12 5 4
Jugador ☺ es [H]umano, Máquina [T]onta o Máquina [L]ista: L
Jugador ☻ es [H]umano, Máquina [T]onta o Máquina [L]ista: H

JUGADA #1
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
A B C D E F G H I J K L
Turno de ☺: ⚀ ⚁ ⚂ ⚃
Jugada: AAEI

JUGADA #2
  |☻| | | | | | | | | | |
  |☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | |☺| | | | | | |
A B C D E F G H I J K L
Turno de ☻: ⚀ ⚃ ⚂ ⚃
Jugada: BBHC

JUGADA #3
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | |☺| | | |☻| | |
A B C D E F G H I J K L
Turno de ☺: ⚅ ⚄ ⚅ ⚅
Jugada: AAAG

JUGADA #4
  |☻| | | | | | | | | | |
  |☻| | | | | | | | | | |
  |☻| | |☺|☺|☺| |☻| | | |
A B C D E F G H I J K L
Turno de ☻: ⚃ ⚁ ⚀ ⚃
Jugada: BILB

JUGADA #5
  |  | | | | | | |☻| | | |
  |☻| | |☺|☺|☺|☻| | | | |
A B C D E F G H I J K L
Turno de ☺: ⚁ ⚀ ⚁ ⚁
Jugada: @EGJ

JUGADA #6
  |  | | | | |☺| |☻| | | |
  |☻| | | | |☺| |☻| | | |
A B C D E F G H I J K L
Turno de ☻: ⚀ ⚃ ⚃ ⚁
Jugada: ***

JUGADA #3
☺|☻| | | | | | | | | | |
☺|☻| | | | | | | | | | |
☺|☻| | | |☺| | | |☻| | |
A B C D E F G H I J K L
Turno de ☺: ⚅ ⚄ ⚅ ⚅
Jugada: AAAG

JUGADA #4
  |☻| | | | | | | | | | |
  |☻| | | | | | | | | | |
  |☻| | |☺|☺|☺| |☻| | | |
A B C D E F G H I J K L
Turno de ☻: ⚀ ⚃ ⚀ ⚁
Jugada: _
```

2.5. Múltiples jugadores

Este último bloque se considera cumplido si se han implementado correctamente todos los bloques anteriores y el juego funciona con cualquier número de jugadores proporcionando un valor (distinto al por defecto) al parámetro **fichas** en la creación del objeto de clase **Pargammon**. En el ejemplo siguiente se creado el objeto en el **main** con la línea `juego = Pargammon(*params, fichas="☺☻×")`

```
*** PARGAMMON ***
Numero de columnas, fichas y dados = 10 5 3
Jugador ☺ es [H]umano, Máquina [T]onta o Máquina [L]ista: H
Jugador ☻ es [H]umano, Máquina [T]onta o Máquina [L]ista: T
Jugador × es [H]umano, Máquina [T]onta o Máquina [L]ista: L

JUGADA #1
☺|☻|×| | | | | | | |
☺|☻|×| | | | | | | |
☺|☻|×| | | | | | | |
☺|☻|×| | | | | | | |
☺|☻|×| | | | | | | |
A B C D E F G H I J
Turno de ☺: ⚀ ⚁ ⚂
Jugada: AAA

JUGADA #2
  |☻|×| | | | | | | |
  |☻|×| | | | | | | |
  |☻|×| | | | | | | |
☺|☻|×|☺| | | | | | |
☺|☻|×|☺|☺| | | | | |
A B C D E F G H I J
Turno de ☻: ⚀ ⚃ ⚃
Jugada: @@B

JUGADA #3
  | |×| | | | | | | |
  |☻|×| | | | | | | |
  |☻|×| | | | | | | |
☺|☻|×|☺| | | | | | |
☺|☻|×|☺|☺| |☻| | | |
A B C D E F G H I J
Turno de ×: ⚃ ⚃ ⚀
Jugada: CCC

JUGADA #4
  |☻| | | | | | | | |
  |☻| | | | | | | | |
☺|☻|×| |☺| | | | | |
☺|☻|×| |☺|☺|×|☻|×| |
A B C D E F G H I J
Turno de ☺: ⚃ ⚃ ⚃
Jugada: AEE

JUGADA #5
  |☻| | | | | | | | |
  |☻|×| | | | | | | |
  |☻|×| | | | | | | |
☺|☻|×| | | |☺|☺|☻|×|
A B C D E F G H I J
Turno de ☻: ⚁ ⚃ ⚃
Jugada: @@B

JUGADA #6
  |☻|×| | | | | | | |
  |☻|×| | | | | |☻|×|
☺|☻|×| | | | | |☻|×|
A B C D E F G H I J
Turno de ×: ⚃ ⚃ ⚁
Jugada: @CF

JUGADA #7
  |☻| | | | | | | |×|
☺|☻|×| | | | | |☻|×|
☺|☻|×| | | | |☺|☻|×|
A B C D E F G H I J
Turno de ☺: ⚀ ⚃ ⚃
Jugada: AAE

JUGADA #8
  |☻| | | | | | | |×|
  |☻|×| | | | |☺|☻|×|
  |☻|×| | | | |☺|☻|×|
A B C D E F G H I J
Turno de ☻: ⚁ ⚁ ⚃
Jugada: B@@

JUGADA #9
  | | | | | | | | |×|
  |☻|×| | | | |☺|☻|×|
  |☻|×| |☻| | |☺|☻|×|
A B C D E F G H I J
Turno de ×: ⚁ ⚁ ⚂
Jugada: CFJ

…
```

3. Criterios de Evaluación

Los alumnos depositarán el código de la práctica en una tarea del Campus Virtual de la asignatura. El depósito del código no proporciona ninguna calificación, es simplemente un requisito previo para unificar la evaluación de subgrupos y realizar comprobaciones de autoría. La calificación se obtendrá en la **defensa**, en la que se propondrá una **modificación** del código presentado.

Importante: La defensa se llevará a cabo en los ordenadores del aula, sin acceso a internet, y usando el entorno de desarrollo **IDLE**.

Se deben cumplir las siguientes **condiciones** para que la práctica **sea evaluada** (en caso contrario se calificará con un cero):

- La práctica se debe realizar individualmente o en pareja (dos alumnos).
- Si se realiza en pareja, es posible que la calificación de ambos sea distinta, según el desarrollo de la defensa.
- El código presentado debe haber sido desarrollado **en su totalidad** por el/los alumno(s), sin ayudas humanas o de IA's.
- El código debe poder evaluarse sin errores durante la defensa. En el caso de errores triviales los alumnos deben ser capaces de corregirlos in situ inmediatamente.
- Durante la defensa se solicitará a los alumnos que **modifiquen su código** para implementar un cambio sencillo en las condiciones de la práctica. Para que se evalúe la práctica los alumnos deben ser capaces de realizar la modificación y obtener un código funcional.

En la evaluación de la práctica se tendrán en cuenta, entre otros, los siguientes aspectos:

- La correcta resolución del problema y la implementación de los distintos aspectos mencionados en el enunciado.
- La modularidad y estructuración de la solución (división en funciones y clases).
- La documentación interna. Deben existir al menos comentarios que indiquen el propósito y la forma en que se debe usar cada función y clase del programa, así como los comentarios adecuados para cada parte del código donde se vaya a realizar alguna tarea no trivial.
- El uso de las técnicas impartidas en la asignatura.

4. Presentación y Evaluación de la práctica

La evaluación de la práctica se divide en dos etapas:

1. Presentación electrónica del fichero ***.py** que contiene el código de la práctica. Se habilitará en el Campus Virtual de la asignatura una tarea de subida de ficheros cuya fecha límite será el **domingo 6 de abril a las 23:59.** Al principio de todos los ficheros debe aparecer un comentario con el nombre de quienes la han desarrollado.

2. Evaluación **presencial**, en laboratorio, ante el profesor. Se realizará en el lugar, día y hora correspondiente al horario de prácticas del subgrupo al que pertenezca durante la semana del 7 al 13 de abril.