Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2019-2

9 y 10 de diciembre de 2019
Actividad Sumativa

Actividad 11

Recuperativa

Introducción

Estás terminando tu semestre y te llega un mensaje de los profesores:

¡Hola! Bienvenido al mundo PROGRÁMON.

Somos los profesores RUZ, FLORENZANO, OSSA y DOMINGUEZ.

Pero la gente nos llama PROFESORES PROGRÁMON.

Este mundo esta habitado por unas criaturas llamadas PROGRÁMON.

Para algunos, los PROGRÁMON son mascotas. Pero otros los usan para pelear.

Tu propia leyenda PROGRÁMON está a punto de comenzar.

Te espera un mundo de sueños y aventuras con los PROGRÁMON.

¡Adelante!

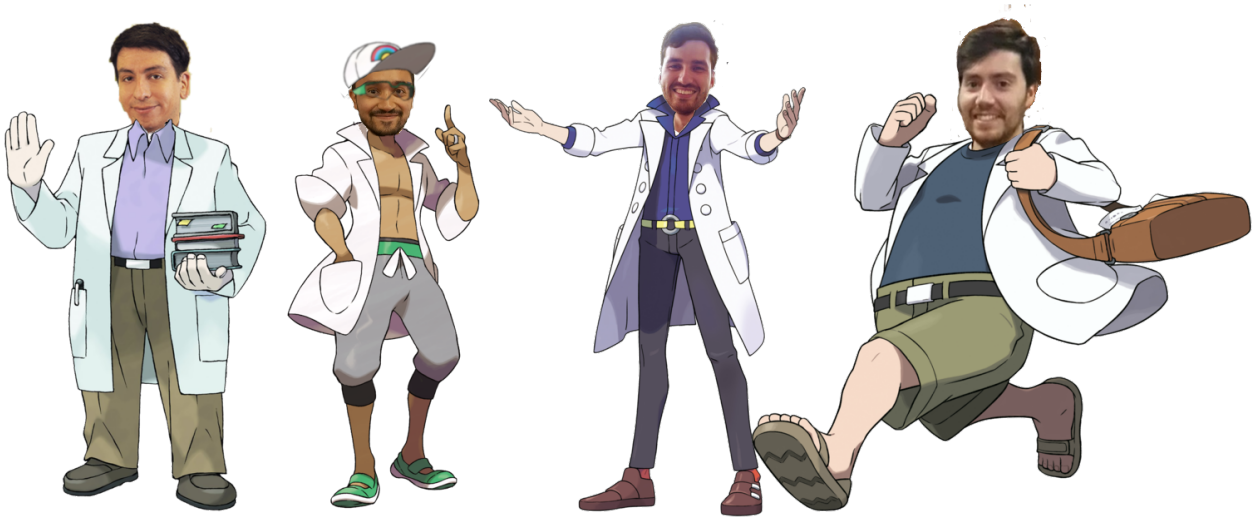

Figura 1: PROFESORES PROGRÁMON

Instrucciones y entrega

- **Lugar de entrega:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AC11`
- **Hora del *push*:** 10 de diciembre, 20:00

A continuación se presentan seis secciones con problemas de programación distintos, cada uno vale **dos puntos**. Cada uno está enfocado en un contenido específico del curso correspondiente a una **Actividad Formativa**. Tú solo debes desarrollar **tres** de estas secciones, con las cuales se te evaluará esta actividad.

Tu repositorio personal inicialmente solo contaba con diez carpetas de actividades, así que debes crear la carpeta `AC11` para entregar esta evaluación. **Dentro de ella, solo debes entregar los archivos correspondientes a las tres secciones que realizaste, y en sus carpetas respectivas. En caso de entregar más de tres secciones, se corregirán tres al azar de las entregadas**.

Por ejemplo, si decides realizar las partes 1, 4 y 5, la estructura de tus carpetas de actividades en tu repositorio debiese verse así:

```
Actividades
├── AC01
├── AC02
├── ...
├── AC10
└── AC11
    ├── 01
    │   ├── main.py
    │   └── ...
    ├── 04
    │   ├── main.py
    │   └── ...
    └── 05
        ├── main.py
        └── ...
```

Recomendaciones generales

- **Se recomienda comenzar por una sección de un contenido que conozcas bien.**
- **Si te quedas pegado en una sección, intenta avanzar con otra.**
- **Recuerda que son solo tres secciones a completar.**
- **Si así lo deseas, puedes entregar menos de tres partes. Recuerda que cada una que entregas solo vale dos puntos de nota.**
- **Recuerda estar atento a las *issues* en caso de que alguien ya haya preguntado una duda que tuviste.**

1. Iterables (Gimnasio tipo Hada)

Te da la bienvenida al primer gimnasio la líder **Hadani** y su fiel Clefairy. Hadani está obsesionada con el tipo Hada, tan misterioso y mágico, y te pide le ayudes con un programa que busca describir los PROGRÁMON de este tipo en comparación al resto de los tipos. Hadani te recompensará con la primera medalla si logras el siguiente desafío.

Hadani te entrega `main.py`, que tiene la base del programa objetivo. Debes completar las **seis funciones cortas** pendientes y que se describen a continuación. El detalle es que Hadani **no permite que utilices los ambientes de iteración** `for` y `while` para que aproveches al máximo la magia de los **iterables**. La palabra reservada `for` solo puede utilizarse dentro del contexto de una **estructura por comprensión**, y te recomienda el uso de las funciones `map`, `filter` y `reduce` para actuar sobre objetos iterables.

Para obtener los datos de los PROGRÁMON es necesitamos extraer información del archivo de texto que tiene datos separados por comas (`programon.txt`). Para eso necesitamos las funciones:

- `def obtener_lineas_archivo(ruta)`: **Esta función se entrega implementada, no debes alterarla**. Recibe la ruta (`str`) de un archivo, lee el archivo y retornar una lista con sus líneas de texto. Cada línea sigue la siguiente forma, donde donde se especifican en orden: el identificador, el nombre, el primer tipo, el segundo tipo[1], y luego las características de PROGRÁMON:

  `'1,Bulbasaur,Planta,Veneno,45,49,49,65,65,45'`

- `def obtener_programones(lineas)`: Esta función recibe una lista con líneas obtenidas en la función anterior y a partir de los datos obtenidos de las líneas, debe retornar una lista con instancias de PROGRÁMON creadas utilizando la *named tuple* entregada `Programon`.

Para consultar la información obtenida anteriormente deberás implementar las siguientes funciones:

- `def obtener_tipos(programones)`: Esta función recibe una lista con múltiples PROGRÁMON y retorna una lista con los tipos de todos los PROGRÁMON presentes en el iterable entregado. Nota, que cada PROGRÁMON tiene a lo más dos tipos distintos, y el resultado no puede tener repeticiones. Un ejemplo de esta consulta sería:

  ```
  obtener_tipos([
      PROGRaMON(id='39', nombre='Jigglypuff', tipo_1='Normal', tipo_2='Hada',...),
      PROGRaMON(id='183', nombre='Marill', tipo_1='Agua', tipo_2='Hada',...)
    ])
  ```

  Y un resultado esperado sería:

  `['Hada', 'Agua', 'Normal']`

- `def obtener_programones_de_tipo(programones, tipo)`: Esta función recibe una lista con múltiples PROGRÁMON y un *string* (`str`) que indica un tipo de PROGRÁMON. La función debe retornar una nueva lista que sólo contenga los PROGRÁMON del iterable entregado pero que sean del tipo indicado. Un ejemplo de esta consulta sería:

  ```
  obtener_programones_de_tipo([
      PROGRaMON(id='284', nombre='Masquerain', tipo_1='Bicho', tipo_2='Volador',...),
      PROGRaMON(id='301', nombre='Delcatty', tipo_1='Normal', tipo_2='',...),
      PROGRaMON(id='347', nombre='Anorith', tipo_1='Roca', tipo_2='Bicho',...)
    ],
    'Bicho'
  )
  ```

[1] Un PROGRÁMON puede no tener segundo tipo, en cuyo caso está vació ese campo de la línea.

Y un resultado esperado sería:

```
[
    PROGRaMON(id='284', nombre='Masquerain', tipo_1='Bicho', tipo_2='Volador',...),
    PROGRaMON(id='347', nombre='Anorith', tipo_1='Roca', tipo_2='Bicho',...)
]
```

- `def obtener_cantidad_por_tipo(programones, tipos)`: Esta función recibe una lista con múltiples PROGRÁMON y una lista de *strings* (`str`) que indican tipos de PROGRÁMON. La función retorna un `dict` que tiene como llaves los tipos entregados y como valor asociado la cantidad de PROGRÁMON de ese tipo que se encuentran en el iterable.[2] Un ejemplo de esta consulta sería:

```
obtener_cantidad_por_tipo([
    PROGRaMON(id='267', nombre='Beautifly', tipo_1='Bicho', tipo_2='Volador',...),
    PROGRaMON(id='269', nombre='Dustox', tipo_1='Bicho', tipo_2='Veneno',...)
  ],
  ['Planta', 'Bicho', 'Veneno', 'Volador']
)
```

  Y se obtendría:

```
{'Planta': 0, 'Bicho': 2, 'Veneno': 1, 'Volador': 1}
```

- `def obtener_caracteristica_promedio(programones, caracteristica)`: Esta función recibe un iterable con PROGRÁMON y un `str` con el nombre de una característica de PROGRÁMON. Para obtener el valor de un atributo de un objeto `obj` a partir de su nombre como *string* `attr`, se puede utilizar la función `getattr(obj, attr)`. La función a implementar debe retornar un decimal con el valor promedio de esa característica para los PROGRÁMON presentes en el iterable entregado.

  Un ejemplo de esta consulta sería:

```
obtener_caracteristica_promedio(
  [
    PROGRaMON(id='522', nombre='Blitzle', tipo_1='Eléctrico',..., velocidad='76'),
    PROGRaMON(id='524', nombre='Roggenrola', tipo_1='Roca',...,velocidad='116')
  ],
  'velocidad'
)
```

  La cual debe dar: `96.0`

- `def repr_lista(lista)`: recibe una lista y retorna un *string* (`str`)[3] que enumera en múltiples líneas cada elemento junto a su posición en la lista de forma ordenada. Un ejemplo:

```
repr_ranking(['Dragón', 'Hada', 'Bicho'])
```

  Cuyo resultado sería (como *string*, los `'\n '` representan los saltos de línea):

```
'''
1. Dragón\n
2. Hada\n
3. Bicho
'''
```

[2] La función anterior te puede ser de ayuda para realizar esta función.
[3] El método `str.join` que recibe iterables de *strings* puede ser de mucha ayuda.

Notas

- El código principal de `main.py` usa las funciones pedidas para que puedas ir viendo los resultados que arrojan.
- La función `def obtener_programones_de_tipo` te puede ser de ayuda para implementar `def obtener_cantidad_por_tipo`.
- Para obtener el valor de un atributo de un objeto `obj` a partir de su nombre como *string* `attr`, se puede utilizar la función `getattr(obj, attr)`.
- El método `str.join` que recibe iterables de *strings* puede ser de mucha ayuda para `def repr_lista`.

Requerimientos

- (0.4 pts) Implementar correctamente `def obtener_programones`
- (0.4 pts) Implementar correctamente `def obtener_tipos`
- (0.3 pts) Implementar correctamente `def obtener_programones_de_tipo`
- (0.3 pts) Implementar correctamente `def obtener_cantidad_por_tipo`
- (0.3 pts) Implementar correctamente `def obtener_caracteristica_promedio`
- (0.3 pts) Implementar correctamente `def repr_lista`

2. *Threading* (Gimnasio tipo Fantasma)

Para el segundo gimnasio, contamos a su cargo a la líder del **Brujaviera** junto a su querido Runerigus. Hace varios días están teniendo problemas con la organización del torneo anual de entrenadores **Fantasma** que ocurre en el gimnasio. Por lo que si la ayudas a simular su torneo con las herramientas de *threading*, te otorgará la segunda medalla.

El torneo es una competencia de eliminación directa: se enfrentan de a dos entrenadores, donde el ganador pasa a la siguiente ronda y el perdedor queda eliminado inmediatamente. Cada batalla es simple, se atacan sucesivamente en turnos hasta que uno de deshabilite. Los entrenadores que van ganando cada ronda se enfrentan según una estructura de árbol, hasta que haya un solo ganador. Por alta demanda, este año cuentan con 16 entrenadores que entraron al torneo, por lo que Brujaviera necesita coordinar las batallas para que se realicen en un orden tal que las batallas iniciales se realicen antes que las batallas que depende de ellas.

Se te entrega `main.py` **ya implementado** con el código base para realizar la simulación del torneo: este crea instancias de la clase `Entrenador` que representan a los participantes del torneo; y de la clase `Batalla` que representan el enfrentamiento entre dos participantes y que son los *threads* a coordinar en el programa. Luego se crea la estructura de árbol del torneo en la función `generar_batallas()`, y finalmente, se comienzan los *threads* de batallas.

En `clases.py` se entregan las clases a completar. Primero, esta la `class Entrenador` que se entrega **parcialmente implementada**. Cada instancia cuenta con un nombre, un valor de ataque, de defensa y de HP (*health points* o salud) de su PROGRÁMON. También cuenta con métodos: `sanar` que recupera el HP de su PROGRÁMON; y `atacar` que lo que debes completar en esta clase:

- `def atacar(self)`: Este método efectúa daño a un contrincante a partir de sus características. Además, deberá encargarse de revisar si hay un ataque crítico disponible según la instancia de `Critico`, que existe como atributo de clase. Para revisar si está disponible, puedes usar el método `is_set`, ya que este es un `Event` que se refrescará aleatoriamente cada cierto tiempo. Si no hay un ataque crítico, el daño se calcula tal como dicta el programa entregado. Pero si lo está, se encarga de mostrar en pantalla que hubo un crítico, reiniciar el valor del `Event critico`, y duplicar el daño calculado.

Similarmente, la `class Batalla` se te entrega a medias y la **deberás completar**. Tiene tres atributos principales: `oponente_1`, `oponente_2` y `ganador` que comienzan como `None`. Antes de comenzar la simulación, a cada batalla se asignan los atributos `oponente_1` y `oponente_2` a: dos instancias de `Entrenador` (lo que representa una batalla de etapa inicial entre entrenadores); o dos instancias de `Batalla` (que presenta una batalla que depende de otras batallas previas, de etapa posterior).

Debes completar los métodos:

- `def run(self)`: Este es el método del *thread* que se ejecuta al llamar a `start`. Aquí se debe revisar la naturaleza de los atributos `oponente_1` y `oponente_2` para asegurarse que sean instancias de `Entrenador` (para lograrlo, puedes usar la función `isinstance`[4]). Si lo son, entonces llama al siguiente método, `realizar_batalla`, pero si son instancias de `Batalla`, debe **esperar** a que haya un ganador en cada una de ellas. Cuando ocurra lo anterior, **reasigna el valor** de `oponente_1` y `oponente_2` a los respectivos entrenadores **ganadores** de las batallas que ya terminaron.
- `def realizar_batalla(self)`: Este método se ejecuta una vez que se puede iniciar la batalla entre dos entrenadores, y es el ciclo principal de la simulación. Mientras ambos entrenadores tengan HP mayor a 0, se debe simular un turno de batalla. Cada turno, aleatoriamente se escoge que entrenador ataca primero al otro. El primer entrenador ataca, y si el segundo entrenador sigue con HP mayor

[4]Para revisar si un objeto `obj` es instancia de una clase `cls`, puedes usar el método `isinstance(obj, cls)`.

a 0, entonces ataca de vuelta. Para efectos de la simulación, cada turno durará **1 segundo**, y continuarán ejecutándose turnos hasta que uno de los entrenadores se haya debilitado. Cuando esto ocurra, se debe asignar el valor de `ganador` al entrenador y se debe sanar antes de que pase a la siguiente batalla.

Además de coordinar que ciertas batallas se realicen antes de otras como descrito previamente, se debe asegurar que solo una **batalla ocurra a la vez**: el gimnasio de Brujaviera no es lo suficientemente grande para albergar más de una batalla concurrente. Además de completar los métodos `run` y `realizar_batalla`, puedes agregar atributos de instancia o de clase a la clase `Batalla` que estimes necesario para asegurar los requisitos de coordinación.

Toda situación que obligue a *threads* a esperar en el programa debe realizarse mediante un mecanismo de sincronización revisado en el curso, y no mediante un mecanismo de *busy waiting*[5]. Es decir, se espera se resuelva mediante *locks*, eventos, o métodos nativos de `Thread` como `join`.

Notas

- Lee la estructura del código de `main.py` para familiarizarte con el problema. Agrega `print`s de ser necesario para ver los valores de lo entregado.
- Mucho de los descrito anteriormente se encuentra ya implementado, revisa cuales son los aspectos faltantes.

Requerimientos

- (0.5 pts) Implementar la revisión de ataques críticos en `def atacar`.
- (0.5 pts) Asegura en `def run` que las batallas dependientes de otras batallas esperen para ser ejecutadas sin usar un mecanismo de *busy waiting*.
- (0.5 pts) Completar la simulación en `def realizar_batalla` siguiendo el flujo indicado y declarando un ganador.
- (0.5 pts) Utiliza algún mecanismo para asegurar que solo una batalla se ejecuta a la vez sin usar un mecanismo de *busy waiting*.

[5]Se refiere a no forzar una pausa por chequeo continuo de una condición. Así el programa no se ejecuta continuamente en ese tiempo de espera y el procesador se ocupa en trabajo efectivo.

3. Excepciones (Gimnasio tipo Bicho)

¡Oh no! El malvado **equipo P** ha soltado un montón de **Bugterpies** en los servidores de la PROGRÁDEX, causando problemas en los datos del sistema. Por esto el líder del gimnasio **Vicho** te pide que encuentres y atrapes los distintos errores del sistema.

Por suerte, posees en tu equipo un **Pydgey** especializado en derrotar PROGRÁMON de tipo **Bicho**. Para ganar la medalla, deberás completar los siguientes movimientos. Estos, son métodos de la clase `Pidgey` que aparece en el archivo `clases.py`:

- `def aire_afilatipo(programon)`: Recibe una instancia de `Programon` y verifica que no tenga más de dos tipos y que éstos no sean el mismo. En caso contrario levanta un `TypeError`.
- `def pico_taladraid(programon)`: Recibe una instancia de `Programon` y levanta un `IndexError` si su *id* no corresponde al rango de *ids* de la `generación` del PROGRÁMON. Podrás la correspondencia de generaciones y rangos de *ids* en el diccionario `indices_generaciones` entregado.
- `def remolinombre(programon)`: Recibe una instancia de `Programon` y revisa que su nombre no tenga la palabra `'bug'` entremedio. En caso contrario levanta la excepción personalizada `Bugterpie`. Nota, que la palabra `'bug'` dentro del nombre podría tener una mayuscula, como para `'Bugcanion'`, en cuyo caso también debe detectarse.

Excepción `Bugterpie`

Como habrás notado, también deberás construir la excepción `Bugterpie`, la cual se encuentra en el archivo `bugterpie.py`. Ésta deberá recibir la instancia de `Programon` que la lanzó e imprimir un mensaje cada vez que se levante y guardar el número de veces que se ha levantado. También, cuenta con un método que imprime la cantidad de `Bugterpies` encontrados hasta el momento. Este se llamará luego de haber capturado todas las excepciones.

Un ejemplo de consultas donde se levantaría el error y los *prints* esperados sería:

```
pydgey = Pydgey()
pydgey.remolinombre(Programon(416, 'Vespibug', ...))
pydgey.remolinombre(Programon(721, 'Bugcanion', ...))
```

```
El PROGRáMON N 416 es un Bugterpie
El PROGRáMON N 721 es un Bugterpie
En total has capturado 2 Bugterpies
```

Capturar errores

El flujo del programa es: una instancia de `Pydgey` es creada y en el método `encontrar_errores()` usa los métodos que levantan excepciones para revisar no hayan errores. Si se levantan errores, deberás capturarlos adecuadamente, y corregir la información del PROGRÁMON según el caso:

- Si el PROGRÁMON tiene más de dos tipos, debe quedarse con los primeros dos. En el caso que los dos primeros sean el mismo, deberás eliminar uno de ellos.
- Si el *id* del PROGRÁMON no corresponde a su generación, deberás cambiar su generación apropiadamente. No habrá PROGRÁMON con *id* superior al límite de la última generación ni inferior al de la primera.
- Si el nombre del PROGRÁMON contiene a la palabra `'bug'`, deberás reemplazarla con la palabra `'progra'`.

Notas

- Los métodos `str.lower` y `str.capitalize` que cambian las mayúsculas y minúsculas de un *string* te puede ser útil para el encontrar `Bugterpies` y corregir los nombres de los PROGRÁMON.
- Te recomendamos poner *prints* como los siguientes, para ayudarte a visualizar los errores y si fueron corregidos correctamente:

```
El PROGRáMON N 017: Pidbugtto es un Bugterpie
>> Nombre corregido: PROGRáMON N 017: Pidprogratto

El PROGRáMON N 031: Nidoqueen no corresponde a la generación 2
>> Generación corregida: PROGRáMON N 031: Nidoqueen generación 1

El PROGRáMON N 036: Clefable tiene uno o más tipos repetidos: ['Hada', 'Hada']
>> Tipo corregido: PROGRáMON N 036: Clefable tipo ['Hada']

El PROGRáMON N 047: Bugasect tiene más de dos tipos: ['Bicho', 'Planta', 'Acero',
                                                       'Bicho', 'Bicho']
>> Tipo corregido: PROGRáMON N 047: Bugasect tipo ['Planta', 'Bicho']
```

Requerimientos

- (0.75 pts) Levantar excepciones. Se levanta excepción si:
  - (0.25 pts) Un PROGRÁMON posee más de dos tipos o más de uno repetido.
  - (0.25 pts) El *id* de un PROGRÁMON no corresponde a su generación.
  - (0.25 pts) El nombre de un PROGRÁMON contiene la palabra *'bug'*.
- (0.5 pts) Construir la excepción *Bugterpie*
- (0.75 pt) Capturar excepciones y corregir correctamente.
  - (0.25 pts) Corregir errores de tipo.
  - (0.25 pts) Corregir errores de generación.
  - (0.25 pts) Corregir errores de nombre.

4. Listas Ligadas (Gimnasio tipo Roca) y Árboles (Gimnasio tipo Planta)

Luego de un arduo camino en busca del gimnasio de tipo Roca, te das cuenta de que se encuentra fusionado con el de tipo Planta. Al entrar te encuentras con **Erika** (**Planta**) y **Brock** (**Roca**), los líderes del gimnasio, quienes te cuentan que, al haber tantos PROGRÁMON, ahora les ha costado mucho ordenarlos. A Erika y a Brock se les ocurrió utilizar un `ProgramonTree` para mantenerlos ordenados, pero no saben como utilizarlo bien. Como eres un experto programador te proponen ayudarlos a terminar esta estructura, y si lo logras te darán las medallas de ambos gimnasios.

Figura 2: Gimnasios tipo **Planta** y **Roca**.

Para cumplir con esta misión y así ganar ambas medallas, te dan la siguiente información:

Clase `NodoProgramon`:

Esta clase representa a cada nodo del árbol, que es un PROGRÁMON diferente. Sus atributos son su `numero` (`int`), `nombre` (`str`), `hije_izquierdo` (`NodoProgramon`), `hije_derecho` (`NodoProgramon`); el primero representa el número asignado en la PROGRÁDEX, el segundo el nombre del PROGRÁMON, y los últimos otras instancias de `NodoProgramon` hijos.

Clase `ProgramonTree`:

Una instancia de `ProgramonTree` es una estructura que contiene instancias de `NodoProgramon` ordenados. Este solo tiene una referencia `raiz` al primero `NodoProgramon`, pero se encarga de agregar en orden las referencias entre nodos. En la Figura 3 se muestra un ejemplo de como se vería un `ProgramonTree`.

Como ves en el ejemplo, los `ProgramonTree` tienen un orden basado en el número de la PROGRÁDEX de cada en PROGRÁMON. A la izquierda de un `NodoProgramon` se encuentran solo otros nodos cuyo número son menores, y a la derecha solo con números mayores. Nota que esto se cumple para cada nodo.

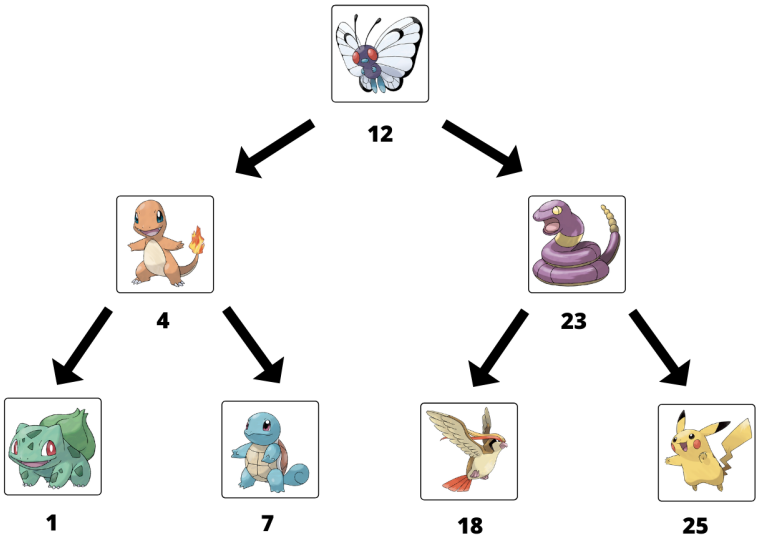

Figura 3: Ejemplo `ProgramonTree`. PROGRÁMON ordenados por número.

`programontree.json`

Este archivo contiene información de los `NodoProgramon` a cargar en el `ProgramonTree`. A continuación se muestra el archivo `programontree.json` del ejemplo:

```
[
    {"nombre": "Butterfree", "numero": 12},
    {"nombre": "Charmander", "numero": 4},
    {"nombre": "Ekanz", "numero": 23},
    {"nombre": "Bulbasaur", "numero": 1},
    {"nombre": "Pidgeot", "numero": 18},
    {"nombre": "Squirtle", "numero": 7},
    {"nombre": "Pikachu", "numero": 25}
]
```

Programa

Para facilitar las tareas a realizar, se especifican los siguientes métodos para ordenar los requisitos para llegar al objetivo final y conseguir ambas medallas.

- `def cargar_nodos(self, ruta):`

  Este método **se entrega implementado, y no debes alterarlo**. Recibe la ruta del archivo JSON y carga los datos del grafo utilizando el siguiente método `insertar_programon`.

- `def insertar_programon(self, programon):`

  Este método **si debes implementarlo**. Recibe una instancia de `NodoProgramon` y debe encargarse de insertarlo en el árbol en el lugar que corresponde. Esto significa, que dado un nodo donde podrías

anclarlo, debe ver como se compara el número que se desea insertar. Si es menor, lo insertará a la izquierda de él. Si ya tiene un nodo a la izquierda, debe repetir la comparación con ese hijo. Si es mayor, debe colocarlo a la derecha, reproduciendo lo mismo que el lado izquierdo. Deberás seguir este procedimiento hasta encontrar un espacio libre. **Puedes asumir que nunca se insertarán dos nodos del mismo PROGRáMON**.

Por ejemplo, supongamos el `ProgrmonTree` comienza vacío. Si se agrega un `NodoProgramon` de número 12, entonces queda a la raíz ya que el árbol está vacío. Si luego se agrega un `NodoProgramon` de número 23, se compara con el número de la raíz, y como es mayor, se agrega como hijo derecho de este. Ahora, si agrego un `NodoProgramon` de número 18 se vuelve a comparar con la raíz, encontrando que se debe anclar a la derecha. Pero la raíz ya tiene un nodo a la derecha, por lo que ahora debe comparar con ese (el de número 23). Como 18 es menor a 23, lo ancla a la izquierda del nodo.

- `def numero_programon(self, nombre):`

  Este método también **debes implementarlo**. Recibe el nombre de un PROGRÁMON que se encuentra en el `ProgramonTree` y debeencontrar a este PROGRÁMON en el dentro de el y retornar su número. **Puedes asumir que el PROGRáMON buscado siempre estará en el árbol**.

- `def ruta_programon(self, nombre):`

  Este método se te entrega parcialmente construido y **debes completarlo**. Recibe el nombre de un PROGRÁMON que se encuentra en el `ProgramonTree`. Tu misión es encontrarlo y retornar la ruta desde el nodo raíz hasta el PROGRÁMON como una lista en donde aparezcan los números de los PROGRÁMON por los cuales pasaste.

Notas

- Las clases entregadas implementan el método `__repr__` para que puedas imprimir de forma amigable el árbol que construyes.

Requerimientos

- (1 pt) Se implementa correctamente el método `def insertar_programon`.
- (0.4 pts) Se implementa correctamente el método `def numero_programon`.
- (0.6 pts) Se implementa correctamente el método `def ruta_programon`.

5. Grafos (Gimnasio tipo Normal)

Blanca es la líder de gimnasio de ciudad Trigal y tiene un Miltank endemoniado que representa la peor pesadilla de todo entrenador PROGRÁMON novato. Para derrotarla necesitas criar un PROGRÁMON con un movimiento (ataque) que normalmente no aprendería: un movimiento huevo.

Un PROGRÁMON es una criatura que puede atacar con poderes especiales de distintos tipos, a los que llamamos "movimientos". Cada especie de PROGRÁMON tiene dos listas de movimientos que puede aprender, **movimientos por nivel**, que corresponde a los movimientos que aprende de manera espontánea a lo largo de su vida y **movimientos huevo**, que son los movimientos que un PROGRÁMON puede heredar de su padre.

Normalmente, los PROGRÁMON tienen crías con otros miembros de su misma especie, sin embargo también es posible que dos PROGRÁMON de especies similares tengan un huevo. Se dice que dos PROGRÁMON son de especies similares, si ambos pertenecen a un mismo **grupo huevo**, y la cría que tengan será **de la especie de la madre**, pero tendrá los **movimientos del padre** que estén en su lista de movimientos huevo.

Cadenas de crianza

Dado que no siempre encontrarás un PROGRÁMON que tenga el movimiento que quieres en el mismo grupo huevo de la especie que buscas, podrás criar varias generaciones de PROGRÁMON, generando así una **cadena de crianza**.

Una cadena de crianza representa las parejas de PROGRÁMON que debes criar y en qué orden para conseguir que tu PROGRÁMON final obtenga un movimiento y se verán como una lista de nombres, donde el primer PROGRÁMON de la lista es el objetivo del movimiento huevo, mientras que el último es aquel que aprende el movimiento naturalmente.

Por ejemplo, imagina que en nuestro universo existieran únicamente tres PROGRÁMON: SKITTY, SPINDA y GOTHITA. Si quisiéramos conseguir un SKITTY con el movimiento *FAKETEARS*, podríamos seguir el siguiente proceso:

Gen I: GOTHITA: aprende el movimiento *FAKETEARS* naturalmente (al subir de nivel).

Gen II: GOTHITA (Padre) × **SPINDA** (Madre). Pueden criar porque comparten el grupo huevo *Humanlike* y tendrán un SPINDA que sabe *FAKETEARS*

Gen III: SPINDA (Padre) × **SKITTY** (Madre). Pueden criar porque comparten el grupo huevo *Field* y tendrán un SKITTY que sabe *FAKETEARS*

En la figura 4 se muestra el ejemplo mencionado, donde la estrella representa el movimiento *FAKETEARS* que se desea heredar.

Si representamos lo anterior como una cadena de crianza, obtenemos: `['SKITTY', 'SPINDA', 'GOTHITA']`. Pues el PROGRÁMON que queríamos que aprendiera el movimiento era SKITTY, el que lo aprendía naturalmente era GOTHITA y para llegar de uno a otro tuvimos que pasar por SPINDA.

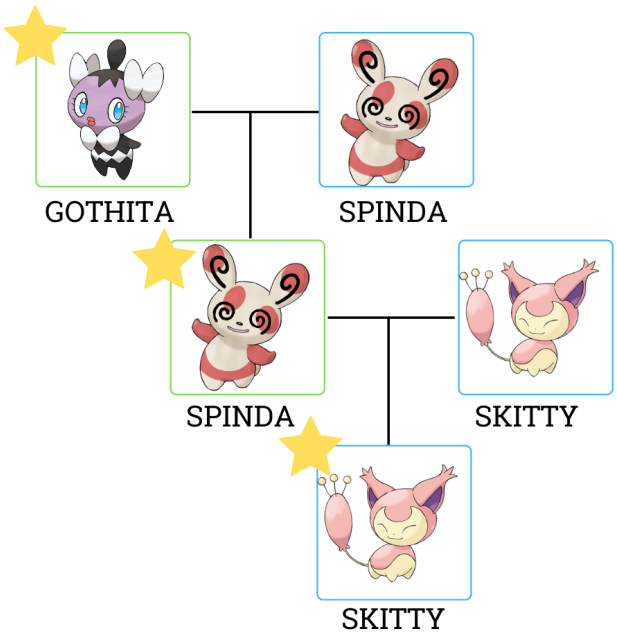

Figura 4: Ejemplo de crianza PROGRÁMON.

Programa

A continuación se muestra lo que tienes que hacer para descifrar como enseñar el movimiento que necesitas a tu PROGRÁMON objetivo.

- `def cargar_grafo(ruta)`

  Debes completar esta función. Esta recibe la ruta de la base de datos con los diferentes PROGRÁMON y carga el grafo con a partir de este utilizando diccionarios. Los datos se encuentra en el archivo `programon.json`, contiene los nombres, movimientos, movimientos de huevo y grupos de huevo, de cada PROGRÁMON y tiene el siguiente formato:

```
{
    "nombre": "BULBASAUR",
    "movimientos": [
        "SEEDBOMB",
        "TAKEDOWN",
        ...,
        "SWEETSCENT"
    ],
    "movs_huevo": [
        "SLUDGE",
        ...,
        "ENDURE"
    ],
    "grupos_huevo": ["Grass", "Monster"]
}
```

Considera que tu grafo va a tener dos tipos de nodos: `Programon` y `GrupoHuevo`, que son clases que se te entregan. Los nodos `Programon` tienen referencias a los `GrupoHuevo` a los que pertenecen y los nodos `GrupoHuevo` tienen referencias a los nodos `Programon` que pertenecen a estos. Por lo que es un grafo **NO dirigido**. Así, el grafo se puede entonces almacenar en diccionarios de las instancias de `Programon` y `GrupoHuevo`, donde cada instancia tiene referencias a los nodos a los que está conectado. Revisa el archivo `main.py` donde se da la idea de estas estructuras.

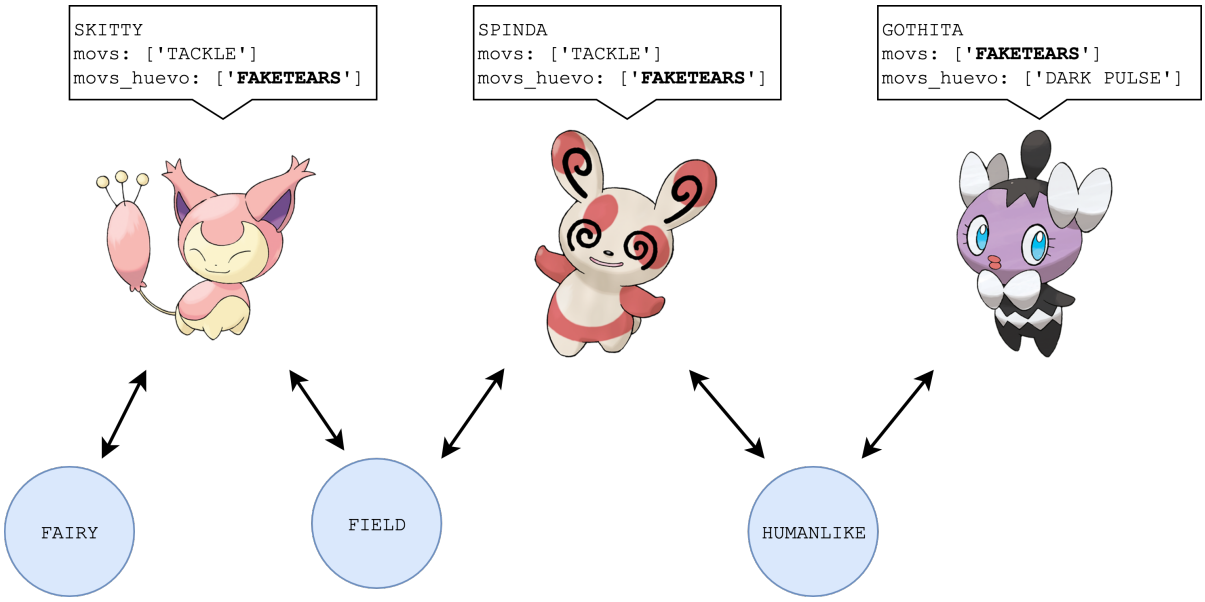

Figura 5: Ejemplo de grafo no dirigido.

- `def comprobar_cadena(grafo, cadena, movimiento)`

  Esta función **se te entrega implementada**, usa el diccionario de instancias que se crea en la función anterior y te puede servir de referencia para entender el problema. Recibe un grafo (`dict` de `Programon`), una cadena de crianza como una lista de *strings* y el movimiento (*string*) que quieres enseñar. Deberá retornar `True` si esta es una cadena válida o `False` si no lo es, indicando (mediante un `print` en pantalla) en qué punto la cadena falla.

- `def movimiento_posible(grafo, programon, movimiento)`

  Debes completar esta función. Esta recibe un grafo (`dict` de `Programon`), el nombre del PROGRÁMON (*string*) al que le quieres enseñar un movimiento y el movimiento (*string*) que se quiere enseñar. Retorna `True` en caso de que exista una cadena que permita el aprendizaje de este movimiento y `False` en caso contrario.

Requerimientos

- (1 pts) Se implementa correctamente la función `def cargar_grafo`.
- (1 pts) Se implementa correctamente la función `def movimiento_posible`.

6. I/O (Gimnasio tipo Eléctrico) y Serialización (Gimnasio tipo Dragón)

Ya teniendo seis medallas de gimnasio, solo te queda visitar dos ciudades colindantes. La primera de estas, es la **ciudad Binaria** que se encuentra en el archivo `Binaria.py`, donde se encuentra el líder **Raulsistor** con su **Magnezone**.

Para derrotarlo, deberás interceptar las ondas generadas por Magnezone que están dentro del archivo `magnezone.pkmn` mediante la siguiente función:

- `def obtener_movimiento_efectivo(path)`

  Esta función recibe el ruta (`path`) del archivo a desencriptar, y debe obtener un *string* a partir de él. Para hacerlo, debes leer y obtener los *bytes* del archivo y realizar el siguiente flujo:

  1. Por cada dos *bytes* del archivo, debes obtener su binario equivalente[6] de ocho dígitos y concatenarlos.
  2. El binario anterior de 16 dígitos resultante siempre tiene sus *bits* duplicados, por lo que deberás simplificarlo eliminando los pares.
  3. Del binario resultante anterior obtén su valor numérico[7]
  4. Finalmente, para ese valor numérico obtén el carácter ASCII equivalente utilizando el diccionario que se entrega en el archivo.

  Es decir, digamos el archivo solo tiene dos *bytes* `\x3c \xcf` :

```
\x3c  \xcf ⟶ 00111100 11001111
00111100 11001111⟶ 01101011
    01101011⟶ 107 ⟶ 'k'
```

Con los caracteres obtenidos y concatenados, obtendrás un mensaje que se imprimirá y te permitirá derrotar al Magnezone de Raulsistor.

Ya con tu medalla, te diriges a la **Ciudad Dragonización** que se encuentra en el archivo `Dragonizacion.py`, donde se encuentra el líder **Enzinni**, quien es también, el líder del malvado **equipo P**.

Figura 6: Enzinni del **equipo P**

Para derrotarlo, debes utilizar los movimientos obtenidos de las siguientes funciones:

[6] La función `bin` de Python puede ser de ayuda.
[7] La función `int` de Python tiene más de un argumento que puede ser de ayuda.

- `def obtener_programons_json(path)`

  Esta función recibe la ruta (`path`) de un archivo a deserializar mediante JSON. Esto deserializará una lista de `Programon`, cuya clase esta definida en el mismo archivo. Sin embargo, el malvado **equipo P** ha borrado todos los movimientos de tus PROGRÁMON, por lo que debes obtenerlos del archivo `movimientos.json`. El archivo contiene un diccionario con los movimientos dado el nombre de un PROGRÁMON, y de estos, debes elegir cuatro al azar para asignarle a cada PROGRÁMON. Retorna una lista de instancias de `Programon`.

- `def obtener_programons_pickle(path)`

  En esta función recibe la ruta (`path`) de un archivo a deserializar mediante `pickle`. Esto deserializará la lista de `Programon` de Enzinni. Pero Enzinni posee demasiados PROGRÁMON, la mayoría de estos, robados por el **equipo P**. Por lo que, deberás agregar el atributo `origen` a cada instancia de `Programon`. Deberá tener el valor `"robado"` en caso de estar en la lista de nombres de PROGRÁMON robados, y en caso de que no, tendrá el valor de `"Enzinni"`. El archivo `robados.poken` posee la lista de nombres de prográmon robados, debes deserializar su contenido con `pickle` (es una lista de *strings* con los nombres). Finalmente la función debe retornar la lista de instancias de `Programon` cuyo atributo `origen` sea `"Enzinni"`.

Notas

- Para rellenar un *string* numérico con ceros para alcanzar cierto largo, puedes usar el método `str.zfill`.
- Recuerda que para personalizar la deserialización de `json`, puedes usar una función *hook*.
- Recuerda que para personalizar la deserialización de `pickle` puedes usar el método `__setstate__` en la clase que quieres instanciar.
- Para sacar una muestra aleatoria de una lista, se puede utilizar el método `sample(lista, tamano)` de `random`.
- Puede encontrar como transformar *bytes* a *bits*, y *bits* a *bytes* en la ayudantía del tema.

Requerimientos

- (1 pt) Completar `def obtener_movimiento_efectivo`:
  - (0.5 pts) Obtener por cada par de *bytes* la representación binaria y eliminar los duplicados.
  - (0.5 pts) Obtener valor numéricos y caracteres correspondientes para formar mensaje.
- (0.5 pt) Completar `def obtener_programons_json` y deserializar correctamente
- (0.5 pt) Completar `def obtener_programons_pickle` y deserializar correctamente