Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2233 Programación Avanzada (2019-2)

Tarea 00

Entrega

- **Tarea**
  - **Fecha y hora:** sábado 17 de agosto de 2019, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T00/`
- `README.md`
  - **Fecha y hora:** lunes 19 de agosto de 2019, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T00/`

Objetivos

- Desarrollar algoritmos para la resolución de problemas complejos.
- Aplicar competencias asimiladas en *Introducción a la Programación* para el desarrollo de una solución a un problema.
- Procesar *input* del usuario de forma robusta, manejando potenciales errores de formato.
- Trabajar con archivos de texto para leer, escribir y procesar datos.
- Escribir código utilizando paquetes externos (*i.e.* código no escrito por el estudiante), como por ejemplo, módulos que pertenecen a la librería estándar de Python.
- Familiarizarse con el proceso de entrega de tareas y uso de buenas prácticas de programación.

Índice

1. Introducción a LegoSweeper

El **Doctor Pinto** busca incrementar su poder, por lo que intenta sabotear la perfecta gestión del **Mafioso Enzini** para que el próximo semestre sea contratado él como el nuevo jefe supremo. El sabotaje consiste en dejar bloques de Lego en distintas baldosas de la oficina de **Enzini**, con el objetivo de evitar que pueda llegar a su computador a tiempo para subir las notas del curso.

Luego de algunos días (y uno que otro pie adolorido), **Enzini** decide hacer algo al respecto y te pide a ti como alumno de *Programación Avanzada* que recojas todos los bloques de la oficina y así evites más accidentes. Para ayudarte con esa labor se te entrega un pequeño aparato que al ser utilizado sobre una baldosa te indicará el número de baldosas adyacentes que contienen bloques de Lego.

Como no quieres que tus pies sufran utilizando el aparato en baldosas al azar, decides utilizar tus conocimientos de *Introducción a la Programación* para crear un programa que simule esta situación y te ayude a practicar para cuando tengas que hacerlo realmente.

2. Flujo del juego

Tu objetivo es escribir un programa que permita a un usuario jugar una partida de LegoSweeper. La ejecución e interacción del juego será mediante consola, por lo que en ésta aparecerán todas las instrucciones para el jugador.

Al ejecutar tu programa se deberá abrir el Menú de inicio, el cual te dará la información pedida en la Subsección 3.1 y una opción para cerrar el programa. Si se inicia una partida, ya sea nueva o reanudar una ya existente, se le pedirá al jugador ingresar un **nombre de usuario** y se deberá abrir el Menú de juego, que debe permitirte descubrir baldosas. Esta última acción deberá seguir las reglas explicadas en Reglas, lo que mostrará más información del tablero para seguir descubriendo baldosas o terminar la partida en caso de seleccionar una baldosa que contenga una pieza de Lego.

En caso de terminar la partida se le debe mostrar al usuario su puntaje final junto a su **nombre de usuario**, guardarlo en los *rankings* del juego y luego devolver al jugador al Menú de inicio para que éste pueda salir del programa o seguir jugando.

3. Menús

Para que tu programa sea, ante todo, fácil e intuitivo de usar, decides hacer que la interacción por consola sea mediante **menús**. Cada menú muestra opciones disponibles al usuario, para luego recibir y procesar su decisión.

Todos los menús deben ser **a prueba de errores de usuario**, tener la opción de **volver atrás** o salir del juego cuando corresponda, y ser **fáciles** de utilizar. El formato de éstos queda a tu criterio, y puedes crear más submenús adicionales a los anteriores, siempre y cuando se cumpla con las funcionalidades mínimas.

Como mínimo deberás implementar los siguientes menús:

3.1. Menú de inicio

Éste es el menú que verá el jugador al iniciar el programa. Debe preguntar inicialmente si desea comenzar una **partida nueva**, **cargar una partida** ya existente o visualizar el ***ranking* de puntajes**. En el caso

de una partida nueva, deberás dar la opción de elegir el tamaño del tablero previamente a comenzar el juego, mientras que si se decide cargar una partida, se cargará la información del archivo correspondiente. Finalmente, si se pide ver el *ranking* de puntajes, se deberán mostrar los 10 puntajes más altos contenidos en el archivo `puntajes.txt`, el cual se explica con mayor profundidad en Puntajes.

3.2. Menú de juego

Este menú permitirá al jugador interactuar con el juego en sí. Debe mostrarse luego de cada jugada efectuada. Primero debe mostrar el estado actual del tablero y luego mostrar la lista de acciones que el jugador puede ejecutar. Dentro de las acciones, se debe ofrecer como mínimo las opciones de: **descubrir una baldosa**, **guardar la partida**, y **salir de la partida, con o sin guardar**.

Si se decide descubrir una baldosa, tu programa deberá manejar y entregar una respuesta apropiada en caso de que dicha baldosa ya esté descubierta. En caso de descubrir una baldosa que no contenga un Lego, deberás mostrar el tablero actualizado y el menú de juego nuevamente.

Hint: Para crear un menú a prueba de errores, se recomienda crear uno que sólo requiera *inputs* numéricos y que rechace cualquier *input* que no sea número[1] o se encuentre fuera del rango. De forma visual, un menú podría ser del estilo:

```
Seleccione una opción:
[1] Crear partida
[2] Cargar partida
[3] Ver ranking
[0] Salir

Indique su opción (0, 1, 2 o 3): (input de usuario)
```

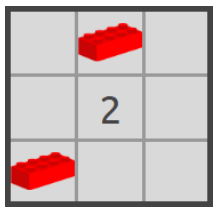
4. Reglas

El objetivo del juego es despejar todas las casillas del tablero que **no contienen Legos** (para que el **Mafioso Enzini** llegue a su computador). En cada turno, el jugador debe ingresar coordenadas que indiquen la casilla que quiere despejar, intentando encontrar todas las casillas que no contienen Legos. Tu programa deberá verificar si hay un Lego en la coordenada ingresada o no y, para cada caso, seguir las instrucciones indicadas.

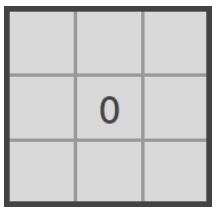
En el caso de que se ubique un Lego en la coordenada ingresada, el jugador pierde. A continuación, se deben mostrar todos los Legos que contenía el tablero en sus respectivas posiciones, se termina la partida y se calcula el puntaje correspondiente.

En caso de seleccionar una coordenada donde **no** hay un Lego, se debe mostrar el número de Legos que se encuentran en las 8 casillas que rodean dicha posición y permitir continuar jugando. A continuación se presentan algunos tipos de casillas y posibles disposiciones de los Legos que las rodean:

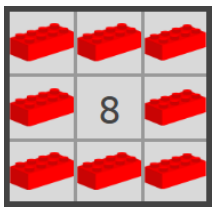
[1] Para esto, te puede ser de ayuda el método `isdigit()` de *strings*.



(a) Casilla rodeada por 2 Legos



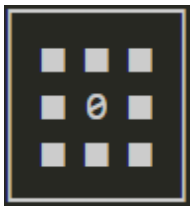
(b) Casilla sin Legos alrededor



(c) Casilla rodeada por 8 Legos

Figura 1: Visualización de ejemplo

(a) Casilla rodeada por 2 Legos



(b) Casilla sin Legos alrededor

(c) Casilla rodeada por 8 Legos

Figura 2: Visualización consola

Cuando todas las casillas que no contienen Legos quedan despejadas, se termina la partida y se procede a calcular el puntaje correspondiente.

5. Partida

5.1. Crear

Al momento de crear una partida se le debe pedir al usuario el tamaño del tablero. Éstos serán dos enteros, N y M que representan el largo y ancho del tablero, respectivamente. Luego, la cantidad de Legos L se verá definida por la siguiente fórmula[2]:

$$L = \lceil N \times M \times \texttt{PROB\_LEGO} \rceil$$

donde:

- N y M son enteros pertenecientes a $[3, 15]$[3].
- `PROB_LEGO` es una variable constante que se encuentra almacenada en el archivo `parametros.py`, el cual sera explicado más adelante.

Finalmente, cada Lego debe ubicarse en una baldosa aleatoria del tablero[4]. Debes asegurarte que hayan exactamente L Legos, todos en distintas posiciones del tablero. Cada baldosa puede contener como máximo un Lego.

[2] La función techo la pueden encontrar como la función `ceil` de la librería `math`.
[3] Debes comprobar que el N y M ingresado por el usuario se encuentren dentro del rango establecido.
[4] Para realizar esta funcionalidad pueden utilizar la librería `random`.

5.2. Guardar

Cuando se guarde una partida, se debe guardar toda la información relevante a ésta en un archivo de extensión `.txt` que tenga como nombre de archivo el nombre de usuario pedido al inicio del juego. Este archivo se debe guardar dentro de una carpeta llamada `partidas`. Finalmente se deben almacenar todas las posiciones que han sido descubiertas, con sus números respectivos y la posición en donde está cada Lego. El formato en el cual se guarde dicha información queda a tu criterio y debe describirse en el archivo `README.md`.

5.3. Cargar

Al momento de cargar una partida, se debe utilizar el nombre de usuario para cargar el archivo que contiene la partida. Si el archivo existe dentro de la carpeta `partidas`[5], se debe mostrar el Menú de juego con los datos de la partida cargada y permitir continuar la partida normalmente. En caso de que el usuario ingrese una partida que no existe, se debe avisar al usuario y volver al Menú de inicio.

6. Puntajes

Al finalizar una partida se le deberá asignar un puntaje al jugador. El puntaje se calculará de la siguiente manera:

$$puntaje = legos\_tablero \times celdas\_descubiertas \times \texttt{POND\_PUNT}$$

donde:

- *legos_tablero*: cantidad de Legos que contenía el tablero.
- *celdas_descubiertas*: cantidad de celdas ingresadas por el jugador que **no** contenían Legos.
- `POND_PUNT`: variable constante que se encuentra almacenada en el archivo `parametros.py`.

Por ejemplo, para el siguiente tablero el cálculo del puntaje seria:

$$\begin{aligned} puntaje &= 11 \times 25 \times 3 \\ &= 825 \end{aligned}$$

Figura 3: Tablero y su respectivo puntaje

Dichos puntajes deberán ser guardados en el archivo `puntajes.txt`. Nuevamente, el formato del archivo queda a tu criterio, pero éste deberá contener los puntajes de **todas** las partidas finalizadas hasta el momento.

[5]Para verificar esto se recomienda utilizar la función `isfile`, del módulo `os.path`.

7. Archivos entregados

Para facilitar tu trabajo **Enzini** te hace entrega de los siguientes módulos: `parametros.py` y `tablero.py`.

Estos módulos **no** deben ser modificados, ni debe escribirse código en ellos. Además, para acceder a los valores o funciones que éstos contienen debes **importarlos correctamente**[6].

7.1. `parametros.py`

La información de las constantes mencionadas anteriormente se encuentra en el archivo `parametros.py`, en donde cada línea almacena una constante con su respectivo valor.

En este caso el archivo contiene:

```
PROB_LEGO = 10 / 64
POND_PUNT = 3
```

Se verificará que la importación de `parametros.py` se realice correctamente dentro del programa, por lo que las constantes anteriores deben provenir de dicho archivo y no ser sobreescritas por ti.

7.2. `tablero.py`

Este módulo contiene la función `print_tablero(tablero, utf8)`, el cual permite imprimir el tablero actual de la partida. Esta función recibe los siguientes argumentos:

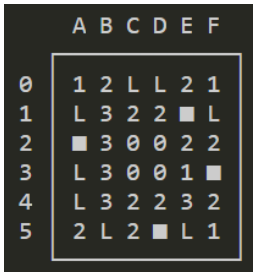
- `tablero`: lista de listas que contiene el estado del tablero. Esta lista debe seguir el siguiente formato:
  - Cada sub-lista representa una fila del tablero, donde cada elemento de la sub-lista representa una celda.
  - Las celdas que aún no han sido descubiertas se representan por un espacio (`' '`).
  - Las celdas que han sido descubiertas y **no** contienen un Lego, son representadas por el número de Legos que tiene alrededor (algún `int` o `str` entre `0` y `8`, incluidos).
  - Las celdas que contienen Legos son representadas con la `str 'L'`.
- `utf8`: booleano que indica si el tablero se imprimirá con caracteres UTF-8 o no, ya que dependiendo de la fuente que utiliza tu consola, el tablero puede presentar problemas para mostrar los caracteres UTF-8 o deformarse. El valor por defecto de esta variable es `True`, en dicho caso el tablero imprimirá los caracteres UTF-8, mientras que si es `False`, estos no se imprimirán. Por lo tanto, si es que tu tablero no se está mostrando correctamente te recomendamos dejar el valor de este argumento como `False`.

A continuación un ejemplo de un *input* y el *output* de cuando `utf8` es `True` y `False`, respectivamente:

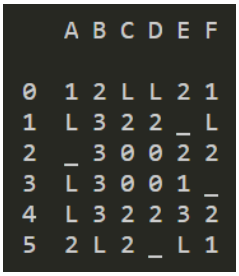
[6] Para más información revisar el siguiente **material**.

```
tablero = [
    ['1', '2', 'L', 'L', '2', '1'],
    ['L', '3', '2', '2', ' ', 'L'],
    [' ', '3', '0', '0', '2', '2'],
    ['L', '3', '0', '0', '1', ' '],
    ['L', '3', '2', '2', '3', '2'],
    ['2', 'L', '2', ' ', 'L', '1']
]
```

Figura 4: Lista de listas con el estado del tablero



(a) `print_tablero(tablero)`



(b) `print_tablero(tablero, False)`

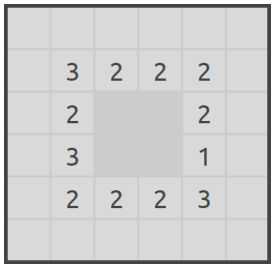
Figura 5: *Output* de su respectiva función

8. Bonus: Descubrimiento de celdas (2 décimas)

Como te podrás fijar, cuando una celda no contiene ningún Lego a su alrededor es "seguro" descubrir todas las baldosas adyacentes sin preocuparte de encontrar un Lego. Tu deber es automatizar este proceso, por lo que cada vez que se seleccione una casilla con 0 Legos adyacentes deberás despejar todas las casillas a su alrededor de forma instantánea en el mismo turno. Si es que alguna de ellas tampoco tiene Legos a su alrededor, se debe repetir este mismo proceso hasta que no se encuentren más celdas que cumplan dicha condición. Puedes realizar este *bonus* de manera recursiva o iterativa.

A continuación un ejemplo de dicha situación:

(a) Tablero antes de seleccionar alguna de las casillas con un 0



(b) Tablero donde se descubrieron las casillas automáticamente

No se dará puntaje parcial para ningún *bonus*. Para obtener el *bonus*, la funcionalidad se debe implementar de forma completamente correcta.

9. `.gitignore`

Cuando estés trabajando con repositorios, muchas veces habrán archivos y/o carpetas que no querrás subir a la nube. Por ejemplo, puedes estar trabajando con planillas de Excel muy pesadas, o tal vez estás utilizando un Mac y no quieres subir la carpeta `__MACOSX`, o el archivo `.DS_Store`, entre otros.

Una posible solución es simplemente tener cuidado con lo que subes a tu repositorio. Sin embargo esta "solución" es extremadamente vulnerable al error humano y podría terminar causando que subas muchos *gigabytes* de archivos y carpetas no deseados a tu repositorio.[7]

Para solucionar esto, `git` nos da la opción de crear un archivo `.gitignore`. Éste es un archivo **sin nombre, y con extensión `.gitignore`**, en el cual puedes detallar **archivos y carpetas a ser ignoradas por `git`**. Esto quiere decir que todo lo especificado en este archivo **no será subido a tu repositorio accidentalmente**, evitando los problemas anteriores.

En esta ocasión el uso de este archivo **no será evaluado**, pero se recomienda su realización para que aprendan a crearlo y utilizarlo ya que será evaluado en las siguientes tareas.

Se recomienda utilizar el archivo `.gitignore` para ignorar archivos en tu entrega, específicamente el enunciado, los archivos indicados en Archivos entregados y todos los que no sean pertinentes para el funcionamiento de tu tarea. El archivo `.gitignore` debe encontrarse dentro de tu carpeta `T00`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

10. `README`

Para todas las tareas de este semestre deberás redactar un archivo `README.md`, escrito en Markdown, que tiene por objetivo explicar su tarea y facilitar su corrección para el ayudante. Markdown es un *markup language* (como LaTeX o HTML) que permite crear un texto organizado y simple de leer. Pueden encontrar un pequeño tutorial del lenguaje en este ***link***.

Un buen `README.md` debe **facilitar la corrección de la tarea.** Una forma de lograr esto es explicar de forma breve y directa **qué cosas fueron implementadas, y que cosas no**, usualmente **siguiendo la pauta de evaluación**. Esto permite que el ayudante dedique más tiempo a revisar las partes de tu tarea que efectivamente lograste implementar, lo cual permite entregar un *feedback* más certero. Para facilitar la escritura del `README`, se te entregará un ***template*** a rellenar con la información adecuada.

Finalmente, como forma de motivarte a redactar buenos `README`s, todas las tareas tendrán **décimas de des-descuento** si el ayudante considera que tu `README` fue especialmente útil para la corrección. Estás décimas anulan décimas de descuento que les hayan sido asignadas hasta un máximo de cinco.

11. Descuentos

En todas las tareas de este ramo habrá una serie de descuentos que se realizarán para tareas que no cumplan ciertas restricciones. Estas restricciones están relacionadas con malas prácticas en programación, es decir, formas de programar que hace que tu código sea poco legible y poco escalable. Los descuentos tienen por objetivo que te vuelvas consciente de estas prácticas e intentes activamente seguirlas, lo cual a la larga te facilitará la realización de las tareas en éste y próximos ramos donde tengas que programar. Los descuentos que se aplicarán en esta tarea serán los siguientes:

[7] Lamentablemente basado en una historia real.

- **PEP8**: (hasta 4 décimas)

  PEP8 es la guía de estilo que se utiliza para programar en Python. Es una serie de reglas de redacción al momento de escribir código en el lenguaje y su utilidad es que permite estandarizar la forma en que se escribe el programa para sea más legible.[8] En este curso te pediremos seguir un pequeño apartado de estas reglas, el cual puede ser encontrado en la **guía de estilos**. Se descontarán hasta cuatro décimas por no seguir las reglas especificadas.

- `README`: (1 décima)

  Se descontará una décima si no se indica(n) los archivos principales que son necesarios para ejecutar la tarea o su ubicación dentro de su carpeta. También se descontará una décima si es que no se hace entrega de un `README`. Esto se debe a que este archivo facilita considerablemente la recorrección de las tareas.

- **Modularización**: (5 décimas)

  Se descontarán cinco décimas si es que hay uno o más archivos que **excedan las 600 líneas de extensión**. Esto se debe a que tener archivos extremadamente largos no solo entorpece la corrección, sino que también causan que el código sea muy difícil de analizar. En caso de que se quisiera arreglar algo, el tener archivos demasiado largos hace que se pierda mucho tiempo recorriendo el archivo.

- **Formato de entrega**: (hasta 5 décimas)

  Se descontarán hasta cinco décimas si es que no se siguen reglas básicas de la entrega de una tarea, como son el uso de groserías en su redacción, archivos sin nombres aclarativos, no seguir restricciones del enunciado, entre otros[9]. Esto se debe a que en próximos ramos (o en un futuro trabajo) no se tiene tolerancia respecto a este tipo de errores.

- **Adicionales**: (hasta 5 décimas)

  Se descontarán hasta cinco décimas a criterio del ayudante corrector en caso de que la tarea resulte especialmente difícil de corregir, ya sea por una multitud de errores o por que el programa sea especialmente ilegible. Este descuento estará correctamente justificado.

- ***Built-in* prohibido**: (entre 1 a 5 décimas)

  En cada tarea se prohibirán múltiples funcionalidades que Python ofrece y se descontarán entre una a cinco décimas si se utilizan, dependiendo del caso. Para cada tarea se creará una *issue* donde se especificará qué funcionalidades estarán prohibidas. Es tu responsabilidad leerla.

- **Entrega atrasada**: (entre 5 a 20 décimas)

  Las tareas serán recolectadas automáticamente y no se considerará ningún avance realizado después de la hora de entrega. Sin embargo, se puede optar por entregar la tarea de forma atrasada y se descontarán 5 a 20 décimas dependiendo de cuánto tiempo de diferencia haya entre la hora de entrega y la entrega atrasada.

- **Des-descuento**: (entre 1 a 5 décimas)

  Finalmente, se des-descontarán hasta 5 décimas por un `README` especialmente útil para la corrección de la tarea.

En la **guía de descuentos** se puede encontrar un desglose más específico y detallado de los descuentos.

[8] Símil a como la ortografía nos ayuda a estandarizar la forma es que las palabras se escriben.
[9] Uno de los puntos a revisar es el uso de *paths* relativos, para más información revisar el siguiente **material**.

12. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del juego será el pilar de la corrección, es decir, **sólo se corrigen tareas que se puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y `push` en sus repositorios.

Cuando se publique la distribución de puntajes, se señalará con color amarillo cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado de amarillo significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea. En tu `README` deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a esos ítems.

13. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.7.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería Python está prohibido. Pregunta en la *issue* especial del **foro** si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un archivo `README.md` **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. **Tendrás hasta 48 horas después del plazo de entrega** de la tarea para subir el `README` a tu repositorio.
- Tu tarea podría sufrir los descuentos descritos en la **guía de descuentos**.
- Entregas con atraso de más de 24 horas tendrán calificación mínima (1,0).
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0).