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

IIC2233 Programación Avanzada (2024-1)

Tarea 1

Entrega

- **Tarea y** `README.md`
  - **Fecha y hora oficial (sin atraso):** lunes 25 de marzo de 2024, 20:00.
  - **Fecha y hora máxima (2 días de atraso):** miercoles 27 de marzo de 2024, 20:00.
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T1/`
  - **Pauta de corrección**: en este enlace.
- **Ejecución de tarea**: La tarea será ejecutada **únicamente** desde la terminal del computador. Además, durante el proceso de corrección, se cambiará el nombre de la carpeta "`T1/`" por otro nombre y se ubicará la terminal dentro de dicha carpeta antes de ejecutar la tarea. **Los *paths* relativos utilizados en la tarea deben ser coherentes con esta instrucción.**

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 biblioteca estándar de Python.
- Familiarizarse con el proceso de entrega de tareas y uso de buenas prácticas de programación.
- Aplicar conceptos de programación orientada a objetos (POO) para resolver un problema.

Índice

1. *DCCiudad* **3**

2. Red de Metro **3**
2.1. Rutas más largas 4

3. Flujo del programa **6**
3.1. Parte 1 - Funcionalidades 6
3.2. Parte 2 - Menú 11
3.2.1. Menú de Acciones 11

4. Archivos **12**
4.1. Código inicial 12
4.2. `data/**.txt` 13
4.3. `dcciudad.pyc` 14

5. `.gitignore` **14**

6. Importante: Corrección de la tarea **15**

7. Restricciones y alcances **15**

1. *DCCiudad*

Has comenzado a aventurarte en el mundo de la programación y, en busca de un mayor conocimiento, decides viajar a la ciudad de *DCCiudad*, donde se dice que se encuentra el mayor de los secretos de la programación. Sin embargo lo primero que encuentras en tu búsqueda es a la alcaldesa de la ciudad, **Gatochico**, conocida por muchos como "Por Fijar", y se ve bastante preocupada. Intentas preguntarle por el gran secreto y le mencionas tu interés por la programación, pero inesperadamente, al escucharte, te implora tu ayuda.

Resulta que con el inicio del semestre, la ciudad se ha sumido en un caos de proporciones épicas. Los gatos chicos, emocionados por su nueva vida universitaria, han colapsado todos los medios de transporte, especialmente el metro. Como los gatos no quieren ir apretados como en latas de sardinas, **Gatochico** ha lanzado una campaña de emergencia y te encarga la modelación de diferentes redes de metro de su ciudad, así como realizar diversas consultas que la misma alcaldesa o sus ciudadanos tienen sobre la red. Con esto, **Gatochico** podrá determinar cuál red de metro es la indicada para construir en su ciudad y poder darles un mejor transporte a los gatitos.

Dado lo anterior, y como forma de poner en práctica todo lo aprendido en "IIC1103 - Introducción a la programación" para llevarlo a un nivel más avanzado, has decidido aceptar este proyecto. Por lo cual, ahora deberás desarrollar un programa que permita modelar y realizar consultas sobre una red de metro.

2. Red de Metro

Para poder desarrollar la red de metro de *DCCiudad*, deberás poder simular los túneles de la red con un programa en Python. Para esto, ocuparemos una forma de trabajar estos túneles como una lista de listas, es decir, una matriz. Esta matriz tiene forma de un cuadrado, con `n` filas y `n` columnas. Cada fila y columna representa una estación de metro, y las celdas de cada fila representan la existencia de un túnel entre 2 estaciones. Para referirnos a una fila o columna de la matriz, utiliza la notación $fila_i$ donde i es un número entero mayor o igual a 0 que indica la fila de la matriz, por ejemplo, $fila_0$ hace referencia a la primera fila de la matriz, la fila número 0; y $columna_2$ hace referencia a la tercera columna de la matriz, la columna número 2.

Finalmente, cada celda de esta matriz pueden tener 2 posibles estados:

- `1`: Representa la existencia de un túnel que permite viajar desde la estación de la $fila_i$ a la estación de la $columna_j$.
- `0`: Representa que **no** existe un túnel que permita viajar desde la estación de la $fila_i$ a la estación de la $columna_j$.

En este tipo de representación, puede ocurrir que existan conexiones desde la $fila_i$ a la $columna_i$, es decir, que el metro salga de una estación y llegue inmediatamente a esa misma estación.[1]

En este tipo de matriz, el orden de las filas y columnas siempre serán iguales, es decir, si la $fila_0$ representa la información de la estación `"X"`, entonces la $columna_0$ también almacenará la información de la estación `"X"`. La diferencia es que cada fila representa los túneles que permiten **salir** de una estación, mientras que cada columna representa los túneles que permiten **llegar** a dicha estación.

Por temas de estandarización, para una matriz de medidas `NxN` el origen de la red estará ubicada en la esquina superior izquierda, siendo esta la coordenada `(0, 0)`. En el lado opuesto, la esquina inferior derecha será la última celda ubicada en la coordenada `(N - 1, N - 1)` siendo `N` la cantidad total de

[1] Imagina que es un mini viaje turístico donde el metro sale a mostrar la ciudad y vuelve al mismo punto de partido sin pasar por ninguna estación intermedia.

estaciones representada en esta matriz. El primer número corresponde a la fila de la matriz, mientras que el segundo número corresponde a la columna.

A continuación se muestra un ejemplo de una red de 4 estaciones cuyo nombres son: `"A"`, `"B"`, `"C"` y `"D"`. Se muestran las conexiones con la forma de matriz y una forma visual de ver dichas conexiones.

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	0	1	1	0
1 (B)	1	0	0	1
2 (C)	0	0	0	1
3 (D)	1	0	0	0

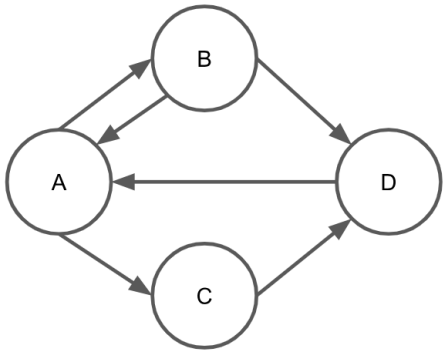

Figura 1: Red de Metro de *DCCiudad* con 4 estaciones.

Tal como se muestra en el ejemplo, en la coordenada `(0, 0)` no existe conexión desde la estación `"A"` hasta la estación `"A"`. En la coordenada `(2, 3)` está la celda con el valor 1, lo que implica que existe un túnel que permite viajar desde la estación `"C"` hasta la estación `"D"`.

Realizando un análisis a nivel de fila, podemos observar que la $fila_0$ representa los túneles que salen de la estación `"A"`. En este caso, desde dicha estación se puede llegar a la estación `"B"` y `"C"`. En cambio, la $columna_3$ representa los túneles que permiten llegar a la estación `"D"`, en este caso, las estaciones `"B"` y `"C"` pueden llegar a la estación `"D"`.

2.1. Rutas más largas

Actualmente, la matriz que hemos confeccionado solo permite ver túneles directos entre una estación y otra, pero a veces es necesario poder ver si existe forma de llegar a una estación más lejos.

Por ejemplo, en la siguiente red podemos observar que desde la estación `"A"` podemos llegar a `"B"` y a `"C"`, y desde estas 2 últimas estaciones podemos llegar a `"D"`:

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	0	1	1	0
1 (B)	0	0	0	1
2 (C)	0	0	0	1
3 (D)	0	0	0	0

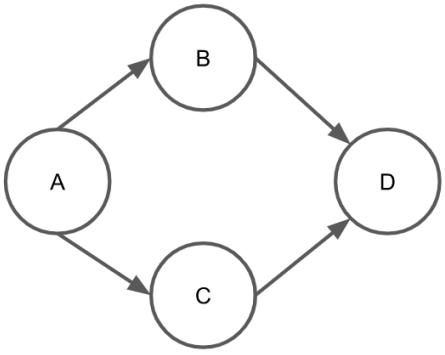

Figura 2: Red de Metro de *DCCiudad* con 4 estaciones.

No obstante, estas matrices tienen un mayor potencial. Existe el proceso de elevar una matriz, que consiste

en multiplicar dicha matriz consigo misma tantas veces como indica el exponente de la potencia[2]. Cuando elevamos una matriz a 2 o 3, nos genera las siguientes matrices:

Matriz Original

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	0	1	1	0
1 (B)	0	0	0	1
2 (C)	0	0	0	1
3 (D)	0	0	0	0

Matriz elevada a 2

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	0	0	0	2
1 (B)	0	0	0	0
2 (C)	0	0	0	0
3 (D)	0	0	0	0

Matriz elevada a 3

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	0	0	0	0
1 (B)	0	0	0	0
2 (C)	0	0	0	0
3 (D)	0	0	0	0

Figura 3: Red de Metro de *DCCiudad* y su versión elevada a 2 y 3.

Observando la matriz elevada a 2, podemos apreciar que hay un 2 en la celda `(0, 3)`. Entonces la interpretación ahora es: existen 2 caminos para llegar desde la estación `"A"` hasta la estación `"D"`, pero pasando por 2 túneles.

En cambio, observando la matriz elevada a 3 podemos ver que solo hay `0`, es decir, no existe camino que pase por 3 túnel y permite conectar 2 estaciones.

Esta propiedad de las matrices se puede extender para cualquier entero positivo. Por lo tanto, una matriz elevada a `Z` nos indicará la cantidad total de caminos posibles para llegar desde una estación a otra, pero pasando por `Z` túneles en cada camino.

Para efectos de esta evaluación, **Gatochico** te entregó un archivo que ya tiene implementado la función para elevar matrices al exponente que tu desees. Además, te exigió que la ocupes en tu tarea para resolver una o más consultas específicas. Más información de esta función en la sección de `dcciudad.pyc`.

Ejemplo de rutas más largas

A continuación se realizará un ejemplo con otra red de 4 estaciones, y se elevará su matriz a 2, 3 y 4.

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	0	1	1	1
1 (B)	0	0	0	1
2 (C)	0	0	0	1
3 (D)	1	0	0	0

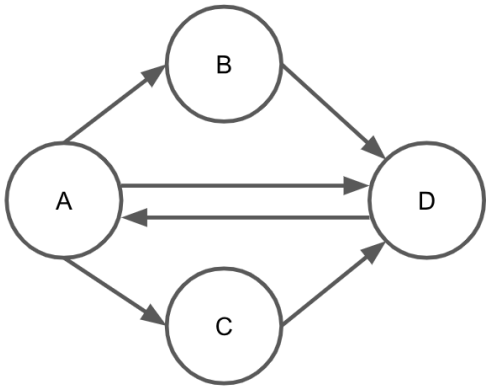

Figura 4: Red original para ejemplo de matriz elevada.

[2]Para entender el proceso de multiplicación de matrices, te recomendamos ver este enlace.

Matriz elevada a 2

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	1	0	0	2
1 (B)	1	0	0	0
2 (C)	1	0	0	0
3 (D)	0	1	1	1

Matriz elevada a 3

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	2	1	1	1
1 (B)	0	1	1	1
2 (C)	0	1	1	1
3 (D)	1	0	0	2

Matriz elevada a 4

	0 (A)	1 (B)	2 (C)	3 (D)
0 (A)	1	2	2	4
1 (B)	1	0	0	2
2 (C)	1	0	0	2
3 (D)	2	1	1	1

Figura 5: Matriz elevada a 2, 3 y 4.

Observaciones

- En la matriz elevada a 2, podemos ver que existe un 2 en la coordenada `(0, 3)`. Esto quiere decir que existen 2 caminos que pasan por 2 túneles y que permiten llegar a la estación `"D"` partiendo desde la estación `"A"`. Si observamos la red original, justamente existe el camino `"A -> B -> D"` y `"A -> C -> D"`.
- En la matriz elevada a 3, podemos ver que existe un 2 en la coordenada `(0, 0)`. Esto quiere decir que existen 2 caminos que pasan por 3 túneles para llegar a la estación `"A"` partiendo desde esa misma estación. Si observamos la red original, justamente existe el camino `"A -> B -> D -> A"` y `"A -> C -> D -> A"`.
- En la matriz elevada a 4, podemos ver que existe un 1 en la coordenada `(3, 2)`. Esto quiere decir que existe 1 camino que pasa por 4 túneles y permite llegar a la estación `"C"` desde la estación `"D"`. Si observamos la red original, justamente existe el camino `"D -> A -> D -> A -> C"`.

3. Flujo del programa

Tu objetivo en esta evaluación estará dividido en completar 2 partes:

Parte 1 Completar un archivo con la clase `RedMetro`, la cuál permitirá: abrir, analizar y responder diferentes consultas de *DCCiudad*.
Esta parte será corregida automáticamente mediante el uso de *tests*.

Parte 2 Confeccionar un menú por consola, que permita interactuar con una red de metro de *DCCiudad*.
Esta parte será corregida manualmente por el cuerpo docente.

3.1. Parte 1 - Funcionalidades

En esta parte, deberás completar diversos métodos de la clase indicada para trabajar con la red de metro de *DCCiudad*. La corrección completa de esta partes será mediante *tests*. Es por esto que debes asegurarte que cada método se pueda ejecutar correctamente, que el archivo no presente errores de sintaxis, y no cambiar el nombre y ubicación del archivo.

Para cada clase y método indicado a continuación, **no puedes modificar su nombre, agregar nuevos argumentos o argumentos por defecto**. En caso de no respetar lo indicado, la evaluación presentará un fuerte descuento. Puedes crear nuevos atributos, nuevos métodos, archivos y/o funciones si estimas conveniente. También se permite crear funciones en otro módulo y que el método que pedimos completar únicamente llame a esa función externa. El requisito primordial es que debes mantener el formato de los métodos y atributos informados en este enunciado.

Adicionalmente, para apoyar el desarrollo de esta parte, se provee de una batería de *tests* donde podrán revisar distintos casos con su respuesta esperada. Estos *tests* públicos corresponden a un segundo chequeo de la evaluación, es decir, no son representativos de todos los casos posibles que tiene una función/método. Por lo tanto, **es responsabilidad del estudiantado confeccionar una solución que cumplan con lo expuesto en el enunciado y que no esté creada solamente a partir de los *tests* públicos**. De ser necesario, el estudiantado deberá pensar en nuevos casos que sean distintos a los *tests* públicos. **No se aceptarán supuestos que funcionaron en los *tests* públicos, pero van en contra de lo expuesto en el enunciado.**

A continuación se describe la clase que deberás completar la clase `RedMetro`. **Importante**: para todos los métodos de esta clase donde se reciba el nombre de una estación de la red, **puedes asumir que el nombre de esta estación existe en la red**, es decir, es una estación válida. Además, pueden existir estaciones "cíclicas", es decir, el metro sale de dicha estación y llega inmediatamente a ella misma.

Modificar `class RedMetro`:
Clase que representa la red de Metro. Registra todas las conexiones existentes en la red como una lista de listas de *int*, junto con los nombres de cada estación de metro.

- No modificar `def __init__(self, red: list, estaciones: list) -> None`:
  Inicializa una instancia de `RedMetro` y guarda como atributo las conexiones de la red de metro y el nombre de cada estación.
  - `self.red` corresponde a los túneles de la red de metro, es decir, sus conexiones directas. Es una lista de listas donde en cada posición, `(i, j)`, hay un número natural que representa la existencia de un túnel que permite viajar desde la estación `i` a la estación `(j)`. Si un valor es `1` implica que existe un túnel, es decir, una conexión directa desde la estación `i` a la estación `j`. En cambio, si es `0`, implica que no existe dicho túnel, es decir, no exista una conexión directa. Esta matriz siempre será cuadrada.
  - `self.estaciones` corresponde a un lista de largo `N` con los nombres de cada estación en la red de metro. Esta lista tiene el mismo orden que `self.red`, es decir, la lista de la posición `i` en `self.red` corresponde a las conexiones de la estación cuyo nombre corresponde al elemento `i` de `self.estaciones`.
- Modificar `def informacion_red(self) -> list`:
  Una consulta estándar de toda red es conocer la cantidad de estaciones y túneles que se disponen. Por este motivo, este método retorna la cantidad de estaciones y la cantidad de túneles que salen de cada estación. Para esto, se retorna una lista de 2 elementos:
  - El primer elemento es un número entero que indica la cantidad de estaciones.
  - El segundo elemento es una lista de largo `N`, donde `N` es la cantidad de estaciones, y en cada posición `i` de la lista debe incluirse un número entero que indique la cantidad de túneles que salen de la estación `i`.

  Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0], [1, 1, 1, 0]]`, este método debe retornar: `[4, [1, 1, 0, 3]]`.
- Modificar `def agregar_tunel(self, inicio: str, destino: str) -> int`:
  Una acción básica de toda red de metro es poder crear nuevos túneles que conecten 2 estaciones de metro. Para lograr esto, este método modifica la red original (`self.red`) para crear una vía de metro entre 2 estaciones, es decir, agregar un túnel desde la estación `inicio` hasta la estación `destino`. Luego, retorna la cantidad total de túneles que salen de la estación `inicio`. En caso que ya exista un túnel desde `inicio` y `destino`, este método retorna `-1`.

Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`, si pedimos agregar el túnel desde `"A"` hasta `"D"`, este método retorna un `3` y la nueva red quedará del siguiente modo: `[[0, 1, 1, 1], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`. Otro caso, si pedimos agregar un túnel desde `"A"` hasta `"B"`, dado que ya existe dicho túnel, entonces la red no se modifica y se retorna `-1`.

- Modificar `def tapar_tunel(self, inicio: str, destino: str) -> int:`
Otra acción básica de toda red de metro es poder tapar túneles ya existentes entre 2 estaciones de metro. Para lograr esto, este método modifica la red original (`self.red`) para tapar una vía de metro, es decir, eliminar el túnel que va desde la estación `inicio` hasta la estación `destino`. Luego, retorna la cantidad total de túneles que salen de la estación `inicio`. En caso que no exista un túnel desde `inicio` y `destino`, este método retorna `-1`.

  Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`, si pedimos eliminar el túnel desde `"A"` hasta `"C"`, este método retorna un `1` y la nueva red quedará del siguiente modo: `[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`. Otro caso, si pedimos tapar un túnel desde `"A"` hasta `"D"`, dado que no existe dicho túnel, entonces la red no se modifica y se retorna `-1`.

- Modificar `def invertir_tunel(self, estacion_1: str, estacion_2: str) -> bool:`
A veces, es necesario invertir la dirección del metro de un túnel, es decir, que si la conexión permitía viajar desde `"A"` hasta `"B"`, ahora el túnel será al revés, es decir, viajar desde `"B"` hasta `"A"`. Para lograr esto, este método modifica la red original (`self.red`) para invertir la dirección de un túnel entre dos estaciones. Puedes asumir que `estacion_1` y `estacion_2` siempre serán distintas. En esta situación ocurren 4 casos:

  - Existe túnel de `estacion_1` a `estacion_2` y existe el túnel de `estacion_2` a `estacion_1`. En este caso no se modifica la red y se retorna `True`.
  - Existe túnel de `estacion_1` a `estacion_2`, pero no existe túnel el opuesto. En este caso se elimina el túnel de `estacion_1` a `estacion_2` y se agrega el túnel de `estacion_2` a `estacion_1`. Luego se retorna `True`.
  - No existe un túnel de `estacion_1` a `estacion_2`, pero si existe el túnel opuesto. En este caso se agregar el túnel de `estacion_1` a `estacion_2` y se elimina el túnel de `estacion_2` a `estacion_1`. Luego se retorna `True`.
  - No existe túnel de `estacion_1` a `estacion_2` y tampoco el túnel opuesto. En este caso, no se modifica la red y se retorna `False`.

  Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`, si pedimos invertir el túnel desde `"A"` hasta `"B"`, este método retorna `True` y la nueva red quedará del siguiente modo: `[[0, 0, 1, 0], [1, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`. Otro ejemplo, si pedimos invertir un túnel desde `"A"` hasta `"D"`, dado que no existe ninguno túnel entre esas dos estaciones, entonces la red no se modifica y se retorna `False`.

- Modificar `def nivel_conexiones(self, inicio: str, destino: str) -> str:`
Una consulta de interés para los usuarios es saber qué tan lejos están dos estaciones. Para lograr esta consulta, este método verifica el nivel de conexión desde la estación `inicio` hasta la estación `destino`. Las respuestas posibles son:

  - `"túnel directo"`: existe un túnel desde la estación `inicio` hasta la estación `destino`.

- `"estación intermedia"`: no existe túnel desde la estación `inicio` hasta la estación `destino`, pero solo con una estación intermedia, se puede llegar desde `inicio` hasta `destino`.
- `"muy lejos"`: no existe túnel directo desde la estación `inicio` hasta la estación `destino` y tampoco existe una estación intermedia que permita llegar desde `inicio` hasta `destino`, pero existe alguna ruta, más larga, que sí permite llegar a `destino` desde `inicio`.
- `"no hay ruta"`: desde la estación `inicio`, no existe forma de llegar a la estación `destino`.

Para este método **deberás** utilizar la función `alcanzable` que se te proveerá en el archivo `dcciudad.pyc`. Se explicará con más detalle este archivo y función en la sección de Archivos.

Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0]]`, el nivel de conexión desde `"A"` hacia `"B"` es `"túnel directo"`, el nivel de conexión desde `"A"` hacia `"C"` es `"estación intermedia"`, el nivel de conexión desde `"A"` hacia `"D"` es `"muy lejos"`, y el nivel de conexión desde `"D"` hacia `"B"` es `"no hay ruta"`.

- Modificar `def rutas_posibles(self, inicio: str, destino: str, p_intermedias: int) -> int:`

Una consulta muy recurrente, y un poco extraña, de los usuarios del metro es saber la cantidad de rutas posibles que hay entre una estación de metro y otra, pero solo considerando las rutas tengan una cantidad específica de estaciones intermedias. Por ejemplo, si existiera una ruta como la siguiente: `"A -> B -> C -> D"`, esta es solo contabilizada cuando se pregunte por 2 estaciones intermedias. Para lograr esta consulta, este método retorna la cantidad posibles de rutas existentes para llegar desde `inicio` hasta `destino`, pero que contengan exactamente `N` estaciones intermedias en dicha ruta, donde ese `N` está dado por el argumento `p_intermedias`, cuyo valor siempre será un entero mayor o igual a 0. En caso que no existan rutas con las `N` paradas intermedias solicitadas, se debe retornar `0`.

Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 1, 0], [1, 0, 0, 1], [0, 0, 0, 1], [1, 0, 0, 0]]`, las rutas posibles desde `"A"` hasta `"D"` con 1 parada intermedias son 2 (`"A -> B -> D"` y `"A -> C -> D"`), las rutas posibles desde `"A"` hasta `"B"` con 2 estación intermedias son 1 (`"A -> B -> A -> B"`), y las rutas posibles entre `"D"` y `"D"` con 2 estación intermedias son 2 (`"D -> A -> B -> D"` y `"D -> A -> C -> D"`).

- Modificar `def ciclo_mas_corto(self, estacion: str) -> int:`

Una consulta que los usuarios han preguntado mucho es: *¿Existe forma de volver a la misma estación de partida?* Para contestar esta duda, se implementará un método que buscará si existe o no una ruta que permita llegar desde una estación a sí misma. En particular, este método recibirá el nombre de una estación y retornará la cantidad mínima de estaciones intermedias para llegar a la misma estación indicada. En caso de no existir una ruta que permita partir desde una estación y llegar a esta misma, se debe retornar `-1`.

Para este método **deberás** utilizar la función `elevar_matriz` que se te proveerá en el archivo `dcciudad.pyc`. Se explicará con más detalle este archivo y función en la sección de Archivos.

- *Hint 1*: la diagonal de la matriz indica si existe una conexión desde una estación a ella misma, es decir, la existencia de ciclos.
- *Hint 2*: Si una matriz de `N` nodos es elevada `N` veces y todavía no se encuentra ciclos en la estación solicitada, entonces dicha estación no tiene una ruta para llegar a ella misma.

Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 0, 1, 1], [1, 0, 0, 1], [0, 0, 0, 1], [1, 0, 0, 0]]`, el ciclo más corto de `"A"`

es `1` (`"A -> D -> A"`), el ciclo más corto de `"C"` es `2` (`"C -> D -> A -> C"`), y el ciclo más corto de `"B"` es `-1` dado que no existe estación que permita llegar desde `"B"` hasta sí mismo.

- Modificar `def estaciones_intermedias(self, inicio: str, destino: str) -> list:`
  Muchas veces es necesario saber por cuales estaciones se pueden transitar cuando se quiere llegar a un destino. Este método se encargará de retornar una lista con el nombre de todas las estaciones intermedias que hay en las rutas que parten en `inicio` y terminan en `destino`. En este caso, solo se van a considerar rutas que tienen únicamente 1 estación intermedia. Además, no importa el orden de las estaciones en la lista a retornar, mientras contenga todas las estaciones esperadas.

  Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 1, 0], [1, 0, 0, 1], [0, 0, 0, 1], [1, 0, 0, 0]]`, las estaciones intermedias desde `"A"` hasta `"D"` son `["B", "C"]` (`"A -> B -> D"` y `"A -> C -> D"`), las estaciones intermedias desde `"D"` hasta `"C"` es `["A"]` (`"D -> A -> C"`), las estaciones intermedias desde `"C"` hasta `"D"` es `[]` (no hay ruta con solo 1 estación intermedia).

- Modificar `def estaciones_intermedias_avanzado(self, inicio: str, destino: str) -> list:`
  Este método es una versión avanzada de `estaciones_intermedias`. En este caso, solo se van a considerar rutas que tienen únicamente 2 estaciones intermedias. Por lo cual, este método debe retornar una lista de listas, donde cada sublista contendrá el nombre de las dos estaciones intermedias que hay en las rutas que parten en `inicio` y terminan en `destino`. Además, no importa el orden de las estaciones en cada sublista y tampoco importa el orden de las listas a retornar.

  Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C", "D"]` y sus conexiones son: `[[0, 1, 1, 0], [1, 0, 0, 1], [0, 0, 0, 1], [1, 0, 0, 0]]`, las estaciones intermedias desde `"B"` hasta `"D"` son `[["A", "B"], ["A", "C"]]` (`"B -> A -> B -> D"` y `"B -> A -> C -> D"`), las estaciones intermedias desde `"D"` hasta `"A"` es `[["A", "B"]]` (`"D -> A -> B -> A"`), las estaciones intermedias desde `"C"` hasta `"D"` es `[]` (no hay ruta con solo 2 estaciones intermedias).

- Modificar `def cambiar_planos(self, nombre_archivo: str) -> bool:`
  Método que carga el archivo `nombre_archivo` para extraer la información de otra red. Luego, actualiza los atributos de la instancia, `self.red` y `self.estaciones`, con la información de esta nueva red. La red cargada debe ser almacenada con el mismo formato que el original, es decir, una lista de listas, donde cada celda es el *int* correspondiente. Luego de actualizar los atributos correctamente, este método debe retornar `True`.

  En caso que el archivo solicitado no exista, este método debe retornar `False` y no sobre-escribir ningún atributo.

- Modificar `def asegurar_ruta(self, inicio: str, destino: str, p_intermedias: int) -> list:`
  Hay ocasiones que, por presupuesto, se deben cerrar algunos túneles de la red de metro. No obstante, el gobierno debe asegurar que exista una ruta mínima para llegar desde `inicio` hasta `destino` pasando por `N` estaciones intermedias, donde ese `N` está dado por el argumento `p_intermedias`. Este último parámetro siempre será un número entero igual o mayor a 0.

  Este método se debe encargar de **copiar la red de metro** e ir cerrando túneles de tal modo que la ruta más corta desde la estación `inicio` hasta la estación `destino` contenga **exactamente** la cantidad de estaciones indicadas por `p_intermedias`. En este método **no se permite crear nuevos túneles**, la única modificación que puede tener la red es tapar túneles, es decir, cambiar `1` por `0`. Finalmente, se debe retornar la nueva red de metro que satisface la ruta indicada. En caso de no existir una solución que permita que la ruta más corta desde la estación `inicio` hasta la estación

`destino` contenga la cantidad de estaciones indicadas por `p_intermedias`, se debe retornar una lista vacía.

Ejemplo: si tenemos una red con las siguientes estaciones `["A", "B", "C"]` y sus conexiones son: `[[0, 1, 1], [0, 0, 1], [0, 0, 0]]`. Esta red presenta 2 rutas para llegar desde `"A"` hasta `"C"` (`"A -> C"` y `"A -> B -> C"`) Si ejecutamos `asegurar_ruta("A", "C", 1)`, entonces la nueva red de metro debe eliminar la conexión `"A -> C"` para que la ruta más corta entre `"A"` y `"C"` contenga exactamente 1 estación intermedia. Por lo tanto, la red resultante es: `[[0, 1, 0], [0, 0, 1], [0, 0, 0]]`.

Hint: se recomienda apoyarse de la función `elevar_matriz` para ir verificando la existencia de rutas entre dos estaciones que contengan una cantidad específica de estaciones intermedias.

3.2. Parte 2 - Menú

En esta parte deberás implementar un menú que permita interactuar con una red de metro y con una estación de metro en particular. Para esto, se ejecutará tu programa mediante un archivo `main.py`. Esta ejecución deberá aceptar argumentos de línea de comando para poder indicar el nombre de la red a cargar y el nombre de una estación de la red cargada.

Por ejemplo, en la consola, se escribirá "`python3.11 main.py plano_1 cobreloa`". Lo que implica que la red a cargar es "`data/plano_1.txt`" y la estación de metro es `"cobreloa"`. Para trabajar con argumentos de línea de comando, te recomendamos investigar sobre el funcionamiento de `sys.argv` de Python en internet.

Respecto a los argumentos de línea de comando, puedes asumir que siempre será una palabra por argumento y que el orden será el mismo que el indicado en el primer párrafo. No obstante, debes realizar las siguiente validaciones:

- El nombre de la red debe existir como un archivo dentro de la carpeta "`data/`". En caso que no se cumpla esta restricción, se debe indicar, en consola, que la red no existe. Más información en la sección Archivos.
- El nombre de la estación debe existir en la red cargada. En caso que no se cumpla esta restricción, se debe indicar, en consola, que la estación no existe.

En caso de levantar uno de los mensajes de validación, el programa debe finalizar. En caso contrario, es decir, se cumplan todas las validaciones, se debe abrir el Menú de Acciones con sus opciones correspondientes.

3.2.1. Menú de Acciones

Este menú permitirá al usuario poder interactuar con la red de metro elegida. Este menú debe indicar el nombre de la red y estación, utilizando la información entregada al momento de ejecutar el programa. Además, debe incluir cuatro (4) opciones: **mostrar red, encontrar ciclo más corto, asegurar ruta y salir del programa**. Este menú debe ser a prueba de errores ante todo dato ingresado por el usuario.

En esta parte de la tarea, se evaluará principalmente que (1) el menú sea aprueba de errores, (2) incluya las cuatro opciones solicitadas, (3) que cada opción del menú llame, mediante código, a la función o método correspondiente, y (4) que una vez ejecutada una opción distinta a "salir", se vuelva al menú para escoger otra opción.

En esta parte no se evaluará si la opción seleccionada realmente hace lo esperado, eso será evaluado en la Parte 1 mediante *tests*. En esta parte se espera un correcto manejo de *inputs* y llamar, en el código, a las funciones o métodos que correspondan según la opción elegida.

A continuación, se detalla la función que debe cumplir cada una de las cuatro acciones que podrá realizar el usuario sobre la ciudad escogida:

1. **Mostrar red**: Imprime la red en la consola. Se debe hacer uso de la función entregada en el archivo `dcciudad.pyc`.
2. **Encontrar ciclo más corto**: Se encarga de utilizar el método `ciclo_mas_corto` en donde el parámetro `estacion` será el nombre de la estación indicada al momento de cargar el programa. Finalmente, se deberá imprimir en consola el resultado retornado por dicho método.
3. **Asegurar ruta**: Se encarga de utilizar el método `asegurar_ruta` en donde el parámetro `inicio` será el nombre de la estación indicada al momento de cargar el programa. Para los parámetros de `destino` y `p_intermedias`, deberás utilizar la función `input()` para solicitar, al usuario, que ingrese ambos datos. Puedes asumir que el usuario ingresará correctamente los datos, es decir, entregará el nombre de una estación existente para el parámetro de `destino` y un número entero positivo para el parámetro de `p_intermedias`. El orden y formato para pedir ambos datos quedan a criterio del programador mientras se utilice la función `input()`. Finalmente, se deberá imprimir en consola el resultado retornado por dicho método.
4. **Salir del programa**: Termina la ejecución del programa.

A continuación se entrega un ejemplo de cómo se podría ver este menú tras haber ejecutado: `python3 main.py Tokyo Shibuya`.

```
¡Se cargó la red Tokyo.txt!
La estación elegida es: Shibuya

*** Menú de Acciones ***

[1] Mostrar red
[2] Encontrar ciclo más corto
[3] Asegurar ruta
[4] Salir del programa

Indique su opción (1, 2, 3 o 4):
```

Figura 6: Ejemplo de Menú de Acciones

4. Archivos

4.1. Código inicial

Para esta tarea, se te hará entrega de diversos archivos que deberás completar con funcionalidades. Puedes crear más archivos si lo estimas conveniente.

- Modificar `red.py`: Aquí encontrarás la definición básica de la clase `RedMetro` que debes completar.
- Crear `main.py`: Este será el archivo que será ejecutado para levantar el menú por consola.
- No modificar `dcciudad.pyc`: Este archivo contiene las funciones `imprimir_red`, `elevar_matriz` y `alcanzable` que deberás ocupar en esta tarea. Más información de esta archivo en la sección de `dcciudad.pyc`.
- Modificar `ejemplo.py`: Este archivo hará uso de las 3 funciones en `dcciudad.pyc` para tener un ejemplo de cómo utilizar las funciones. Puedes modificar este archivo para experimentar más con `dcciudad.pyc`.

- No modificar `data/`: Esta carpeta contendrá una serie de archivos `.txt` que corresponden a diferentes redes de metro a cargar.
- No modificar `tests_publicos/`: Esta carpeta contendrá una serie de archivos `.py` que corresponden a diferentes *tests* para apoyar el desarrollo de la Parte 1. Puedes utilizar los archivos entregados para ir viendo si lo desarrollado hasta el momento cumple con lo esperado en esta evaluación. **Importante**: los *tests* entregados en esta carpeta no serán los mismos que se utilizarán para la corrección de la evaluación.
- No modificar `todos_tests.py`: Este archivo se encarga de ejecutar todos los *tests* contenidos en la carpeta `tests_publicos`.

4.2. `data/**.txt`

Para poder entender el formato de las redes de metro, se te facilitará múltiples archivos que contendrán información sobre las diferentes redes de metro a cargar.

Dentro de la carpeta "`data/`" encontrarás múltiples archivos `.txt` cuyo nombre es igual a la red que deseamos cargar. El contenido de cada archivo estará dado por diversas líneas de texto, donde cada línea tendrá el siguiente formato:

- La primera línea será un número entero positivo que representa la cantidad de estaciones que tiene la red. Llamaremos a este número `N`.
- Las siguientes `N` líneas del archivo serán *strings* que representan el nombre de cada estación.
- Luego vendrán `NxN` elementos separados por una coma (`","`), donde por cada `N` elementos se interpretará como una fila de la red que representa los túneles que salen desde una estación. Cada $celda_{ij}$ puede tomar el valor de `1` o `0`. Donde `1` indica la existencia de un túnel directo desde la estación de la $fila_i$ hacia la estación de la $columna_j$, y `0` representa que no existe dicho túnel.

A continuación se presenta un ejemplo del archivo "`ciudad_T.txt`" y su representación de la red de metro:

```
4
A
B
C
D
0,1,1,0,1,0,0,1,0,0,0,1,1,0,0,0
```

	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	0	0	0	1
D	1	0	0	0

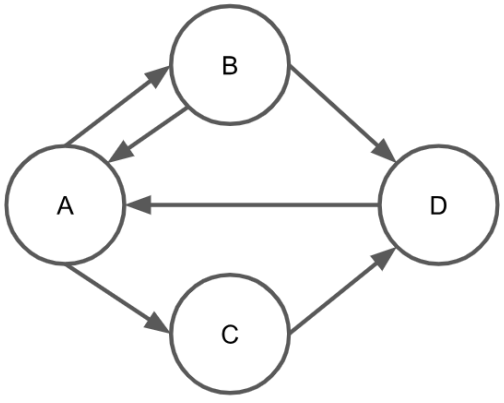

Figura 7: Representación de la red de metro de `"ciudad_T"` en forma de matriz y otra visual.

4.3. `dcciudad.pyc`

Para facilitar tu trabajo y evaluar el uso de módulos, se te hará entrega del módulo `dcciudad.pyc` que contiene diferentes funciones que deberás utilizar en el desarrollo de la tarea. Este archivo se encuentra compilado y funciona únicamente con la versión de Python del curso: 3.11.X con X mayor a 7. Solo es necesario realizar `import dcciudad` desde otro archivo `.py` para utilizar sus funciones. Además, se provee de un archivo: `ejemplo.py` donde se muestra cómo importar y utilizar `dcciudad.pyc`.

Las funciones que contiene `dcciudad.pyc` son 3:

- `imprimir_red(red: list, estaciones: list) -> None:`
  Esta función recibe dos parámetros, la red que corresponde a una lista de listas con `1` y `0`, y una lista con el nombre de cada estación de metro. Luego, se encargará de imprimir en consola la red de metro.
- `elevar_matriz(red: list, exponente: int) -> list:`
  Esta función recibe dos parámetros, la red que corresponde a una lista de listas con `1` y `0`, y un número entero que indica por cuanto elevar la matriz. Este número debe ser un número entero igual o superior a 2 para funcionar correctamente. Esta función retornará una lista de listas con la matriz elevada al número indicado en `exponente`.
- `alcanzable(red: list, inicio: int, destino: int) -> bool:`
  Esta función recibe tres parámetros, la red que corresponde a una lista de listas con `1` y `0`, la posición de la estación de `inicio` en la red, y la posición de la estación de `destino` en la red. Ambos números deben ser enteros entre 0 y `N-1`, donde `N` es la cantidad de estaciones. Esta función retornará `True` si es que existe una ruta que permita llegar desde la estación `inicio` hasta la estación `destino`, o retornará `False` en caso de no existir dicha ruta.

Será tu deber importar **correctamente** este archivo y hacer uso de sus funciones para el desarrollo de la tarea. No debes modificar su contenido.

5. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T1/`.

Los elementos que no debes subir y **debes ignorar mediante** `.gitignore` para esta tarea son:

- El enunciado.
- Los archivos que no debes modificar: `dcciudad.pyc` y `todos_tests.py`
- La carpeta `data/`
- La carpeta `tests_publicos/`.

Recuerda **no ignorar archivos vitales de tu tarea como los que tú debes modificar, o tu tarea no podrá ser revisada**.

Es importante que hagan un correcto uso del archivo `.gitignore`, es decir, los archivos **deben** no subirse al repositorio debido al uso correcto del archivo `.gitignore` y no debido a otros medios.

6. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del programa 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, corroborando que cada archivo de *tests* que les pasamos **corra en un tiempo acotado de 5 minutos**, en caso contrario se asumirá un resultado incorrecto.

En el **siguiente enlace** se encuentra la distribución de puntajes. En esta señalará con color **amarillo** cada ítem que será evaluado a nivel funcional y de código, es decir, aparte de que funcione, se revisará que el código esté bien confeccionado. 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 o con los *tests*.

Importante: Todo ítem corregido por el cuerpo docente será evaluado únicamente de forma ternaria: cumple totalmente el ítem, cumple parcialmente o no cumple con lo mínimo esperado. Del mismo modo, todo ítem corregido automáticamente será evaluado de forma ternaria: puntaje completo si pasa todos los *tests* de dicho ítem, medio punto para quienes pasan más del 70 % de los *test* de dicho ítem, y 0 puntos para quienes no superan el 70 % de los *tests* en dicho ítem. Finalmente, todos los descuentos serán asignados manualmente por el cuerpo docente.

Para terminar, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar al ayudante de Bienestar de tu sección. El correo está en el **siguiente enlace**.

7. 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 3.11.X con X mayor o igual a 7.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py` que estén correctamente ordenados por carpeta. **No se revisará archivos en otra extensión como `.ipynb`**.
- 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. El no incluir este archivo o bien incluirlo pero que se encuentre vacío conllevará un **descuento** en tu nota.
- Esta tarea se debe desarrollar **exclusivamente** con los contenidos liberados al momento de publicar el enunciado. No se permitirá utilizar contenidos que se vean posterior a la publicación de esta evaluación.
- Se encuentra estrictamente prohibido citar código que haya sido publicado **después de la liberación del enunciado**. En otras palabras, solo se permite citar contenido que ya exista previo a la publicación del enunciado. Además, se encuentra estrictamente prohibido el uso de herramientas generadoras de código para el apoyo de la evaluación.
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro que sí sea especificado por enunciado.

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