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

IIC2233 Programación Avanzada (2024-1)

Tarea 3

Entrega

- **Tarea y** `README.md`
  - **Fecha y hora oficial (sin atraso):** lunes 27 de mayo de 2024, 20:00
  - **Fecha y hora máxima (2 días de atraso):** miercoles 29 de mayo de 2024, 20:00.
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`.
    El código debe estar en la rama (*branch*) por defecto del repositorio: `main`.
  - **Pauta de corrección**: en este enlace.
  - **Bases generales de tareas (descuentos)**: 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 "`T3/`" 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

- Entender y aplicar el paradigma de programación funcional para resolver un problema.
- Manejar datos de forma eficiente utilizando herramientas de programación funcional:
  - Uso de map, lambda, filter, reduce, etc.
  - Uso de Itertools
  - Uso de generadores y funciones generadoras.
  - Uso de diccionarios, listas y conjuntos por comprensión.

Índice

1. *DCCervel*

Una vez terminada la batalla final contra Gatochico, ¡Crismiau Ruz y sus Gatos Combatientes se alzan con la victoria! Sin embargo, la DCCiudad se encuentra en ruinas: la destrucción generada por la lucha no fue leve, y sin un alcalde para liderarlos, los ciudadanos se encuentran desesperanzados. Pero en medio de la oscuridad aparece un rayo de luz: es la tortuga Peppa con su equipo de DCCervel, los cuales deciden llevar a cabo una elección democrática para hallar a un nuevo alcalde el cual pueda reconstruir la DCCiudad.

Dado que formas parte del equipo de DCCervel, deberás ayudar a la tortuga Peppa con tus habilidades de programación para analizar una serie de datos y asegurarte que las votaciones sean democráticas.

2. Flujo del programa

En *DCCervel*, deberás acceder a datos sobre las elecciones ocurridas en la DCCiudad, los cuales están almacenados en archivos de distintos tamaños. Luego deberás completar, utilizando **programación funcional**, una serie de consultas que permitirán obtener distintos tipos de información sobre el proceso electoral ocurrido.

Para ~~facilitar la corrección~~ lograr un código más ordenado, se pedirá que implementes como **mínimo**. Estas funciones y métodos ya están definidas en los archivos que te entregamos para esta evaluación. Solo debes completar estas funciones y métodos, es decir, **no debes cambiar su nombre, alterar los argumentos recibidos o cambiar lo que retornen**. Puedes crear nuevos atributos, archivos y/o funciones si estimas conveniente. También se permite crear funciones en otro módulo y que las funciones que pedimos completar únicamente llame a esa función externa. El requisito primordial es que debes mantener el formato de las funciones informadas en este enunciado.

Además, la corrección de esta tarea **será únicamente mediante el uso de *tests***, los cuales otorgarán puntaje dependiendo de cuántos tests se pasen y el tipo de estos, para cada una de las funcionalidades a implementar. Para apoyar el desarrollo de esta tarea, 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. 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.**

En el directorio de la tarea encontrarás los siguientes archivos y directorios:

- Modificar `consultas.py`: Este archivo contiene las funciones a completar señaladas en la sección Programación Funcional.
- No modificar `test_publicos/`: Este directorio contiene los distintos *tests* de la evaluación. Hay dos tipos de *tests*, de correctitud y de manejar datos. En cada consulta los últimos tres *tests* siempre serán para verificar el manejo de datos.
- No modificar `data/`: Esta carpeta contendrá una serie de archivos .csv dentro de otras carpetas. Cada subcarpeta tiene archivos de tamaño distinto, estos archivos son los necesarios para realizar las consultas. **Tu tarea debe trabajar exclusivamente con esta carpeta.**
- No modificar `todos_tests.py`: Este archivo es el que sirve para correr los *tests* de la tarea y además contiene la misma carpeta `data`.

- No modificar `test_solution.py`: Este archivo es el que contiene todos los *outputs* esperados para el manejo de datos.
- No modificar `utilidades.py`: Este archivo contiene la implementación de las *namedtuples* a utilizar.

Importante **Todos los archivos se deben abrir usando el encoding *latin-1*.**

3. Programación Funcional

Para ayudar a que las elecciones del DCCervel sean democráticas y correctas, se necesitará de tu ayuda experta para obtener información mediante la realización de diversas consultas, aplicando tus conocimientos de programación funcional.

Para esto, deberás interactuar con distintos tipos de datos, los que serán explicados con mayor detalle en Datos y completar distintas funciones pedidas en Consultas, que se realizará mediante la modificación de un código pre-existente.

3.1. Datos

Para que puedas implementar las funcionalidades, tendrás que interactuar con las siguientes *namedtuples* que se encuentran presentes en el archivo `utilidades.py`:

3.1.1. Animales

Indica la información de un animal. Posee los atributos:

Atributo	Descripción	Ejemplo
`id`	Un `int` que corresponde al identificador único del animal.	1
`nombre`	Un `str` que corresponde al nombre del animal.	Anton
`especie`	Un `str` que corresponde a la especie del animal.	Antílope saiga
`id_comuna`	Un `int` que corresponde al identificador de la comuna donde vive el animal.	239
`peso_kg`	Un `float` que corresponde al *peso* del animal	34,0
`edad`	Un `int` que corresponde a la edad del animal.	16
`fecha_nacimiento`	Un `str` correspondiente a la fecha de nacimiento del animal en el formato `YYYY/M`.	2007/2

3.1.2. Candidatos

Indica la información de los candidatos. Posee los atributos:

Atributo	Descripción	Ejemplo
`id_candidato`	Un `int` que corresponde al identificador único del candidato.	2055
`nombre`	Un `str` que corresponde al nombre del candidato.	Toccara
`id_distrito_postulacion`	Un `int` que corresponde al identificador único del distrito al que postula.	1
`especie`	Un `str` que corresponde a la especie del candidato.	Medusa

3.1.3. Distritos

Indica la información de un distrito. Posee los atributos:

Atributo	Descripción	Ejemplo
`id_distrito`	Un `int` que corresponde al identificador único del distrito.	1
`nombre`	Un `str` que corresponde al nombre del distrito.	Distrito 1
`id_comuna`	Un `int` que corresponde al identificador único de una comuna perteneciente al distrito.	1
`provincia`	Un `str` que corresponde a la provincia del distrito.	Arica
`region`	Un `str` que corresponde a la region del distrito.	Arica y Parinacota

3.1.4. Locales

Indica la información de un local de votación. Posee los atributos:

Atributo	Descripción	Ejemplo
`id_local`	Un `int` que corresponde al identificador único del local.	1
`nombre_local`	Un `str` que corresponde al nombre del local.	Local 1
`id_comuna`	Un `int` que corresponde al identificador único de la comuna donde está el local.	1
`id_votantes`	Una `list` que corresponde a una lista con todos los identificadores de los votantes del local.	[385, 626]

3.1.5. Ponderadores

Indica la equivalencia de edad que una especie posee en años humanos. Posee los atributos:

Atributo	Descripción	Ejemplo
`especie`	Un `str` que corresponde a la especie.	Perro
`ponderador`	Un `float` que indica a cuántos años humanos equivale un año de la especie.	5.5

3.1.6. Votos

Indica la información de un voto emitido por un animal. Posee los atributos:

Atributo	Descripción	Ejemplo
`id_voto`	Un `int` que corresponde al identificador único del voto.	1
`id_animal_votante`	Un `int` que corresponde identificador único del animal que emitió el voto	385
`id_local`	Un `int` que corresponde al identificador único del local en donde se emitió el voto.	1
`id_candidato`	Un `int` que corresponde a un identificador único del candidato por el que se votó.	3055

3.2. Carga de datos

Para poder trabajar las consultas de esta tarea, deberás cargar información en **generadores**, que contengan las *namedtuples* anteriores. La información de estos generadores se obtendrá a partir de archivos de extensión `.csv` que siguen el mismo formato de las *namedtuples* mencionadas en la sección anterior.

En la carpeta `data` que está dentro de `test_publicos`, podrás encontrar tres subcarpetas `s`, `m` y `l` con bases de datos de distintos tamaños: pequeños, medianos y grandes respectivamente.

Para realizar la carga de información a generadores, deberás completar la siguiente función:

Modificar
```
def cargar_datos(tipo_generador: str, tamano: str) -> Generator:
```

Esta consulta retorna un generador con la base de datos pedida, a partir del archivo del tamaño pedido.

La base de datos retornada depende de la variable `tipo_generador`, la cual puede tomar los valores (`"animales"`, `"candidatos"`, `"distritos"` o `"locales"`, `"ponderadores"`, `"votos"`), y de la variable `tamano`, que puede tomar los valores (`"s"`, `"l"`, `"m"`).

En cada caso, se deberá retornar un generador de *namedtuples*, en donde cada tupla contiene la información de una línea del archivo correspondiente al `tipo_generador` y del tamaño correspondiente a `tamano`.

En algunos *tests* que se usarán para evaluar la tarea (en particular, los que evaluan rendimiento óptimo en la tarea), esta función será usada para cargar y crear los generadores que recibirán las consultas definidas en la siguiente sección. Por lo tanto, se recomienda **completar primero esta función antes de poder completar y probar cualquier consulta**.

3.3. Consultas

A continuación se encuentran las consultas que deberás completar en esta tarea haciendo uso de programación funcional, separadas por la cantidad de generadores que necesita cada una.

3.3.1. Consultas que reciben un generador

Modificar
```
def animales_segun_edad(generador_animales: Generator,
    comparador: str, edad: int) -> Generator:
```

Recibe un **generador** con instancias de animales, una **edad** y un **comparador**, que puede ser `"<"`, `">"` o `"="`.

Retorna un **generador** con los **nombres** de los animales cuya edad sea menor, mayor, o igual que la edad recibida, según el comparador recibido. Si no hay animales que cumplan con la condición, retorna un generador vacío.

Modificar
```
def animales_que_votaron_por(generador_votos: Generator,
    id_candidato: int) -> Generator:
```

Recibe un **generador** con instancias de votos y un **id** de un candidato correspondiente a `id_candidato`.

Retorna un generador con los **id** de los animales que votaron por el candidato indicado. En caso de que ningún animal haya votado por el candidato, se debe retornar un generador vacío.

Modificar
```
def cantidad_votos_candidato(generador_votos: Generator,
    id_candidato: int) -> int:
```

Recibe un **generador** con instancias de votos y un **id** correspondiente a `id_candidato`.

Retorna un `int` que representa la cantidad de votantes que votaron por el candidato con el id entregado.

Modificar `def ciudades_distritos(generador_distritos: Generator) -> Generator:`

Recibe un **generador** con instancias de distritos.

Retorna un **generador** con los **nombres** únicos de las **provincias** a las cuales pertenecen los distritos. Los nombres de las provincias no se deben repetir, por lo que si dos distritos pertenecen a la misma provincia, el nombre de esta debe aparecer solo una vez en el generador.

Modificar
```
def especies_postulantes(generador_candidatos: Generator,
    postulantes: int) -> Generator:
```

Recibe un **generador** con instancias de candidatos y un **numero mínimo** de postulantes.

Retorna un **generador** con los nombres de especies que tengan un número de candidatos **igual o mayor** al numero de postulantes mínimo. Los nombres de las especies seleccionadas no se deben repetir. En caso de no haber especies que cumplan el requisito, se debe entregar un generador vacío.

Modificar `def pares_candidatos(generador_candidatos: Generator) -> Generator:`

Recibe un **generador** con instancias de candidatos.

Retorna un **generador** que contenga tuplas con los **nombres** que corresponden a todos los posibles pares de candidatos sin repetición[1].

Por ejemplo, si se tienen a los candidatos `A`, `B`, `C` y `D`, la función debería retornar un generador con las tuplas `(A, B)`, `(A, C)`, `(A, D)`, `(B, C)` y `(B, D)`.

Modificar
```
def votos_alcalde_en_local(generador_votos: Generator, candidato: int,
    local: int) -> Generator:
```

Recibe un **generador** con instancias de votos, el **id** de un candidato a alcalde y el **id** de un local.

Retorna un **generador** con los votos emitidos en ese local para el alcalde especificado. Para esta función no importa si el voto es válido o no. Puede ocurrir que en el local especificado no hayan votos para el candidato en cuestión, en cuyo caso debes retornar un generador vacío.

Modificar
```
def locales_mas_votos_comuna (generador_locales: Generator,
    cantidad_minima_votantes: int, id_comuna: int) -> Generator:
```

Recibe un **generador** con instancias de locales. Además, recibe una **cantidad mínima** de votantes y el **id** de una comuna.

Retorna un **generador** con el **id** de todos los locales que hayan tenido una cantidad de votantes mayor o igual a `cantidad_minima_votantes`, en la comuna indicada. Retorna un **generador** vacío en caso de no encontrar locales que cumplan con las condiciones indicadas.

Modificar `def votos_candidato_mas_votado(generador_votos: Generator) -> Generator:`

Recibe un **generador** con instancias de votos.

Retorna un **generador** con los **id** de todos los votos que emitidos para el candidato más votado. En caso de empate entre dos o más candidatos más votados, se considera como más votado al candidato con mayor id.

[1] Puede ser útil investigar funciones del módulo `itertools` para esta consulta.

3.3.2. Consultas que reciben dos generadores

Modificar
```
def animales_segun_edad_humana(generador_animales: Generator,
    generador_ponderadores: Generator, comparador: str,
    edad: int) -> Generator:
```

Recibe dos **generadores**, con instancias de animales y conversiones de edades para diferentes especies especies, además de una **edad** y un **comparador**. El comparador será `"<"`, `">"` o `"="`.

Retorna un generado con los **nombres** de los animales cuya edad en años humanos sea menor, mayor o igual que la edad recibida, según el comparador recibido. Si no hay animales que cumplan con la condición, retorna un generador vacío. Si hay nombres repetidos, deben incluir los valores repetidos.

Modificar
```
def animal_mas_viejo_edad_humana(generador_animales: Generator,
    generador_ponderadores: Generator) -> Generator:
```

Recibe dos **generadores**, con instancias de animales y conversiones de edades para diferentes especies especies.

Retorna un **generador** con los **nombres** de los animales que tengan la mayor edad de todos los animales, tras hacer la conversión a edad humana.

Modificar
```
def votos_por_especie(generador_candidatos: Generator,
    generador_votos: Generator) -> Generator
```

Recibe dos **generadores**, con instancias de candidatos y de votos.

Retorna un **generador** que entrega **tuplas** con la especie y el numero de votos totales por candidatos de esta especie, es decir, siguiendo el formato `(especie, numero_votos)`. No es necesario verificar que los votos sean válidos.

Modificar
```
def hallar_region(generador_distritos: Generator,
    generador_locales: Generator, id_animal: int) -> str:
```

Recibe dos **generadores**, con instancias de distritos y locales, y el **id** de un animal.

Retorna un `str` con el nombre de la región de votación del animal especificado. Deberás tomar en cuenta que si un distrito y un local comparten comuna, se encuentran en la misma región.

Modificar
```
def max_locales_distrito(generador_distritos: Generator,
    generador_locales: Generator) -> Generator
```

Recibe dos **generadores**, con instancias de distritos y de locales.

Retorna un **generador** que contiene los nombres de el o los distritos (en caso de empate) con la mayor cantidad de locales de votación.

Modificar
```
def votaron_por_si_mismos(generador_candidatos: Generator,
    generador_votos: Generator) -> Generator:
```

Recibe dos **generadores**, con instancias de candidatos y votos.

Retorna un **generador** con los **nombres** de los candidatos que votaron por sí mismos. Si ningún candidato votó por sí mismo, retorna un generador vacío.

Modificar
```
def ganadores_por_distrito(generador_candidatos: Generator,
    generador_votos: Generator) -> Generador:
```

Recibe dos **generadores** con instancias de candidatos y votos.

Retorna un **generador** con los candidatos ganadores del total de combinaciones candidato-candidato del mismo distrito, sin contar las combinaciones donde el candidato se agrupe consigo mismo. Por ejemplo:

Si se tiene a los candidatos `A`, `B`, `C` y `D`, de los distritos `1`, `1`, `2` y `2` respectivamente, con un total de votos 0, 1, 0 y 1 respectivamente, la función retornará un generador con `B` y `D`, dado que las parejas de candidatos distintos del mismo distrito son `(A, B)` y `(C, D)`, y los ganadores de cada pareja son `B` y `D`, con un voto cada uno.

Para esta pregunta, no es necesario revisar la validez de los votos. Si es que existe un único candidato para un distrito, este no se ha de retornar dado que se retornan a los ganadores por cada pareja del mismo distrito.

3.3.3. Consultas que reciben tres o más generadores

Modificar
```
def mismo_mes_candidato(generador_animales: Generator,
    generador_candidatos: Generator, generador_votos: Generator
    id_candidato: str) -> Generator:
```

Recibe tres **generadores**, con instancias de animales, candidatos y votos, y el **id** de un candidato.

Retorna un **generador** con los **id** de todos los animales que votaron por el candidato y nacieron el mismo mes o mismo año que este. En caso de que no haya candidatos con el id recibido o no haya votantes del candidato que hayan nacido en su mismo mes o año, retorna un generador vacío.

Modificar
```
def edad_promedio_humana_voto_comuna(generador_animales: Generator,
    generador_ponderador: Generator, generador_votos: Generator,
    id_candidato: int, id_comuna:int ) -> float:
```

Recibe tres **generadores**, con instancias de animales, conversión de edades de especies a edades humanas y votos. Además recibe dos identificadores que representan un `id_candidato` y una `id_comuna`.

Retorna un `int` la edad promedio de los animales votantes del candidato dado por `id_candidato` en su conversión edad humana, en la comuna `id_comuna`. En caso de no existir algún identificador de los anteriores en los generadores entregados, se debe retornar 0.

Modificar
```
def votos_interespecie(generador_animales: Generator,
    generador_votos: Generator, generador_candidatos: Generator,
    misma_especie: bool) -> Generator
```

Recibe tres **generadores**, con instancias de animales, instancias de candidatos e instancias de votos, y un **booleano** que indica si la búsqueda es por candidatos de la misma especie o diferente (en caso de no recibir el booleano, este será por defecto `False`).

Retorna un **generador** con instancias de animales que votaron por candidatos de su misma especie o diferente, dependiendo del valor de `misma_especie`.

Modificar
```
def porcentaje_apoyo_especie(generador_animales: Generator,
    generador_candidatos: Generator, generador_votos: Generator)
    -> Generador:
```

Recibe tres **generadores** con instancias de animales, candidatos y votos.

Retorna un **generador** de candidatos junto al porcentaje de apoyo de los animales de su misma especie (aproximado a la unidad), en tuplas siguiendo el formato `(id_candidato, porcentaje)`. El porcentaje de apoyo de los animales de la misma especie del candidato se obtiene como el total de votos de la misma especie del candidato, a favor de este candidato, partido por el total de votos emitidos por la especie del candidato. Para esta función, no es necesario revisar la validez de los votos.

Como ejemplo, si nos encontramos con un candidato `X` de especie `A`, e infinitos animales votantes en total, de los cuales 10 son de la especie `A`, y de los cuales solo 5 votan por el candidato `X`, entonces el retorno para ese candidato sería `X:50` dado que la mitad de los votantes de su especie votan por él. En el caso de que no hayan votos de la especie del candidato, el porcentaje debe ser 0.

Modificar
```
def votos_validos(generador_animales: Generator,
    generador_votos: Generator, generador_ponderadores: Generator) -> int:
```

Recibe tres **generadores** con instancias de animales, votos y conversiones de edades.

Retorna un `int` con el total de votos válidos. Para que un voto sea válido, la edad del votante, transformada en años humanos según la conversión para su especie, debe ser igual o mayor a 18 años.

Modificar
```
def cantidad_votos_especie_entre_edades(generador_animales: Generator,
    generador_votos: Generator, generador_ponderador: Generator
    especie: str, edad_minima: int, edad_maxima: int) -> str:
```

Recibe tres **generadores**, con instancias de animales, votos y conversión de edades. Además, recibe una **especie**, una **edad mínima** y una **edad máxima**.

Se debe encontrar la cantidad de votos emitidos por animales de la especie indicada, que tengan entre `edad_minima y edad_maxima` años humanos (ambos sin incluir).

Retorna un *string* indicando la cantidad total de votos encontrados, sin importar su validez. El *string* que se retorna debe tener el siguiente formato:

```
Hubo {cantidad_votos} votos emitidos por animales entre {edad_minima} y
{edad_maxima} años de la especie {especie}.
```

Modificar
```
def distrito_mas_votos_especie_bisiesto(generador_animales: Generator,
    generador_votos: Generator, generador_distritos: Generator
    especie: str) -> str:
```

Recibe tres **generadores**, con instancias de animales, votos y distritos. Además, recibe una **especie**.

Se debe encontrar el nombre del distrito en donde se hallen más votos emitidos por animales de la especie indicada y que hayan nacido en un año **bisiesto**. En caso de empate, se considerará el distrito con menor id.

Retorna un `str` indicando el distrito encontrado. El *string* que se retorna debe tener el siguiente formato:

```
El distrito {nombre_distrito} fue el que tuvo más votos emitidos por
animales de la especie {especie} nacidos en año bisiesto.
```

En caso de que no haya animal que cumpla con lo indicado se debe retorna el distrito con menor id.

Modificar
```
def votos_validos_local(generador_animales: Generator,
    generador_votos: Generator, generador_ponderador: Generator
    id_local: int) -> Generador:
```

Recibe tres **generadores**, con instancias de animales, votos y ponderadores. Además, el **id** de un local.

Retorna un **generador** con el **id** de todos los votos válidos emitidos en el local correspondiente a el id indicado. En caso de no encontrarse votos válidos en dicho local, retorna un generador vacío.

Recuerda revisar la consulta `votos_validos` donde se defino cuando un voto es válido.

Modificar
```
def votantes_validos_por_distritos(generador_animales: Generator,
    generador_distritos: Generator, generador_locales: Generator,
    generador_votos: Generator, generador_ponderadores: Generator,
```

Recibe cinco **generadores**, con instancias de animales, distritos, locales, votos y ponderadores.

Primero se debe encontrar el distrito con la mayor cantidad de votos válidos. Luego, se debe retornar un **generador** con todos los animales que pertenecen a dicho distrito. En caso de haber empate entre dos distritos, se utiliza el distrito con el menor id.

Recuerda revisar la consulta `votos_validos` donde se define cuando un voto es válido.

Importante **_Disclaimer_: Programación Funcional**

Toda consulta se debe implementar **solamente** usando programación funcional. Por lo tanto, solo se permite el uso de `for` y `while` en funciones generadoras, listas o diccionarios por compresión y dentro de `reduce`. Además, comandos que permitan crear o cambiar una estructura de dato (lista, tupla, diccionario, *set*, etc.) a otra sin aplicar comprensión (o que no sean utilizadas dentro de un `reduce`) estarán prohibidas. Algunos ejemplos de comandos prohibidos son: `list()`, `tuple()`, `dict()`, `set()`, `[]` (lista vacía), `(,)` (tupla vacía), `{}` (set vacío), `.split()`, `sorted()`, entre otros.

Cómo se indica antes, la excepción al uso de estos comandos es es si deseas utilizar `reduce` y el inicializador es una lista, *set* o diccionario vacío.

El incumplir esta condición **implicará la nota mínima (1.0) en la evaluación** dado que no se logra el objetivo de aplicar el paradigma de programación funcional para resolver un problema. Si no estás seguro de si puedes usar uno de los comandos anteriores en algún caso, puedes preguntarlo en el foro.

4. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T3/`.

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

- El enunciado.
- La carpeta `data/`
- La carpeta `test_publicos/`
- El archivo `todos_tests.py`
- El archivo `test_solution.py`
- El archivo `utilidades.py`

Recuerda **no ignorar archivos vitales de tu tarea como los que tú creas o modificas, 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.

5. README

Debido al caracter automatizado de la corrección de esta tarea, los archivos README no serán revisados manualmente por el cuerpo docente. Para esta tarea **deberás adjuntar un** `README.md` en donde solamente se indiquen las **referencias utilizadas para su desarrollo**. Además, no existirá un des-descuento por un buen README.

6. Importante: Corrección de la tarea

En el **siguiente enlace** se encuentra la distribución de puntajes. Para esta tarea, el carácter funcional del programa será el pilar de la corrección, es decir, **sólo se corrigen tareas que puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y *push* en sus repositorios, corroborando que cada *test* que les pasamos para cada consulta corra en un tiempo acotado de **20 segundos por test**, en caso contrario se asumirá un resultado incorrecto.

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 65 % de los *test* de dicho ítem, y 0 puntos para quienes no superan el 65 % de los *tests* en dicho ítem. Finalmente, todos los descuentos serán asignados automáticamente 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`**.
- Toda el código entregado debe estar contenido en la carpeta y rama (*branch*) indicadas al inicio del enunciado. Ante cualquier problema relacionado a esto, es decir, una carpte distinta a `T2` o una rama distinta a `main`, se recomienda preguntar en las ***issues* del foro**.
- 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 **único archivo markdown**, llamado `README.md`, **conciso y claro**, donde describas las referencias a código externo. El no incluir este archivo, incluir un readme vacío o el subir más de un archivo `.md`, 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)**.**