Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada \x32\x30\x32\x31-\x31

\x31\x37 de Junio de \x32\x30\x32\x31
Actividad Sumativa

Actividad Sumativa 4 \x30\x34

I/O y Serialización

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AS4/`
- **Hora del *push*:** 16:30

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Introducción

Después de presenciar el gran éxito del meme de WhatsApp 2 y ver que cambiaron los términos de uso de Whatsapp, decides hacer una nueva aplicación de chat, ~~WhatsApp 2~~ DCChatAPP, con el objetivo de salvar a los alumnos de Programación Avanzada que se comunicaban por ahí. Sin embargo, para evitar que puedan comunicarse, el gran jefe Antonini di Ossa II interceptó tu código y ¡ahora lo está usando para compartir terribles planes con sus camaradas!. Los malvados villanos usaron filtros para esconder estos secretos dentro de una imagen, pero con la ayuda de la DCCIA, la Agente Pepa y tus conocimientos de serialización y bytes, tienes todo en tus manos para revelar la oscura verdad escondida en el corazón del DCC.

Flujo del programa

El programa que deberás implementar se divide en 3 partes, la **Parte Bytes**, **Parte Pickle**, y **Parte JSON**.

- **Parte Bytes:** en esta parte deberás implementar funciones necesarias para poder manejar y aplicar los filtros a las imágenes que ocultan el secreto. Para ayudarte, la **Agente DCCecreto Pepa** creó un aparato de última tecnología para decodificar imágenes. Este aparato puede aplicar filtros de distintos tipos sobre la imagen interceptada, y al aplicar los filtros en el orden correcto podrás revelar el secreto escondido en ella. Sin embargo, **este aparato aplica los filtros sobre tuplas de `int` en formato RGBA**, y tú sólo tienes acceso a la imagen como *bytearray*. Por esto, **tendrás que aplicar tus conocimientos de manejo de *bytes* para poder usarlo correctamente**.
- **Parte Pickle:** El Aparato Supersecreto `FilterBox` es tan clasificado, que no podemos importarlo a nuestro código normalmente! Debemos importar una función que *serializa* el aparato, y una vez recibidos los bytes, deserializarlos para así poder utilizarlo. Para agregar una nueva capa de seguridad, el aparato debe ser serializado de manera que si es interceptado, ¡este no funcionará con cualquier deserialización! En esta sección deberás **personalizar la serialización** del `FilterBox` y **completar la función que realiza tal serialización**.
- **Parte JSON:** Finalmente, en la tercera parte deberás implementar funciones para cargar y transformar los mensajes que ocultan el orden secreto de los filtros. Ahora que ya funciona correctamente tu `FilterBox`, es hora de `descubrir el orden secreto` en el cual debes aplicar los filtros. Por suerte, ¡Lograste *hackear* la base de datos de **DCChatApp** y extraer los mensajes! Sabes que entre los mensajes se oculta el orden de los filtros para descubrir el secreto 👀, sin embargo, son bastantes como para revisarlos a mano. Afortunadamente, también lograste recuperar el prototipo de la interfaz de **DCChatApp**, que te permitirá analizar los mensajes con mayor facilidad. Sin embargo, deberás traducir estos mensajes a un formato que la interfaz pueda reconocer. Una vez más la suerte está de tu lado, dado que **la base de datos está en el famoso formato `JSON`**, y puedes usar lo que has aprendido para decodificarla y clasificarla correctamente.

Ten en cuenta que estas partes son independientes entre sí, por lo que puedes realizarlas en el orden que desees, sin embargo, es recomendable que sigas el propuesto en este enunciado. Al completar todo el programa, deberías tener acceso a dos interfaces, las cuales detallaremos a continuación. Combinando las herramientas de estas interfaces, ¡podrás descubrir el mensaje secreto!

Interfaz parte Pickle-bytes

Esta sección es la que maneja directamente la imagen y aplica los filtros. Al ejecutar `main.py`, se abre una interfaz que muestra una imagen, además de varios botones. Los botones agrupados a la derecha corresponden a los múltiples filtros que puedes ir aplicando sumativamente a la imagen. El boton *SAVE* guarda la imagen que se muestra actualmente, y el botón *RESET* regresa la imagen a su estado original. El filtro del programa funciona en resumidas cuentas de la siguiente manera:

- Se instancia el *backend* y *frontend*, se conectan las señales, e inicializa la interfaz.
- Al apretar uno de los botones que aplica un filtro, la interfaz envía los bytes que representan la imagen **mostrada actualmente**, junto con el nombre del filtro aplicado, al *backend*.
- En el *backend*, la clase `Decoder` hace uso de la clase `FilterBox` para aplicar el filtro.
- `FilterBox` recibe los bytes y el nombre del filtro. Separa los bytes en sus componentes dentro de un archivo `BMP`, y luego convierte los bytes que representan el *bitmap* en tuplas en formato RGBA.

- Dependiendo del filtro pedido, se realiza un cambio sobre las tuplas. Luego, se convierte las tuplas de vuelta a bytes, recrea el `BMP`, y retorna a `Decoder`.
- `Decoder` envía el bytearray resultante devuelta al *Frontend*. Ahí, se actualiza la imagen.
- Se puede repetir este proceso tantas veces como se desee, con cada filtro aplicándose acumulativamente sobre el anterior.

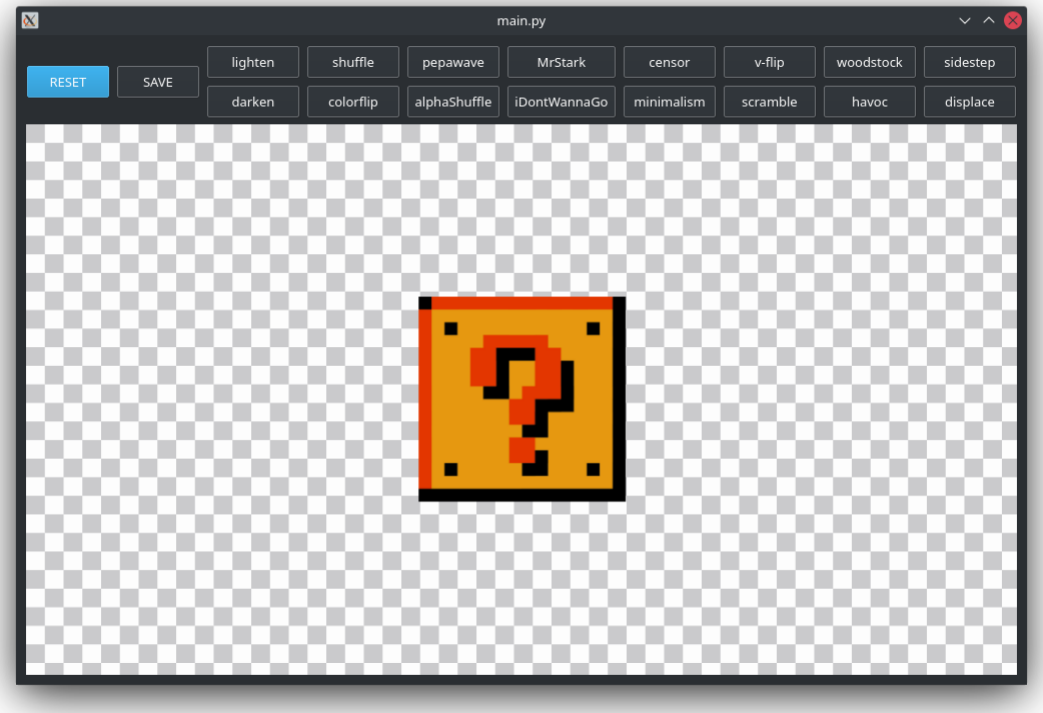

Figura 1: Interfaz gráfica de ventana de filtros

Archivos

```
AS4/
├── JSON/
│   ├── frontend/
│   │   └── interfaz.py
│   ├── main.py
│   ├── mensajes.json
│   └── mensajes.py ## <--- Parte JSON
└── pickle_bytes/
    ├── backend/
    │   └── decodificador.py
    ├── frontend/
    │   ├── image.bmp
    │   └── interfaz.py
    └── filtros.py ## <--- Parte Pickle
```

```
├── main.py
├── manejo_bytes.py ## <--- Parte Bytes
└── parametros.py
```

Parte Bytes y Parte Pickle (carpeta pickle-bytes)

- `backend/decodificador.py`: Contiene la clase `Decoder`, que se encarga de la comunicación con el *frontend* y la clase que aplica los filtros. No debes modificarlo
- `frontend/image.bmp`: Corresponde a la *imagen supersecreta* que debe ser decodificada aplicando los filtros.
- `frontend/interfaz.py`: Contiene la clase `VentanaPrincipal`, que muestra la imagen y permite interacción con el usuario. No debes modificarlo
- `out.bmp`: Este archivo puede ser generado por el programa desde la interfaz, guardará la imagen con los filtros aplicados.
- `main.py`: Este archivo instancia el frontend y el backend y conecta las señales. Debes ejecutar este archivo para probar tu programa. No debes modificarlo
- `manejo_bytes.py`: Este archivo contiene funciones que manejan bytes de distintas maneras. Debes modificarlo
- `filtros.py`: Este archivo define la clase `FilterBox`, que contiene los métodos y procedimientos para aplicar filtros a imágenes. También contiene la función `obtener_paquete_secreto`. Debes modificarlo

Parte JSON (carpeta JSON)

- `mensajes.json`: Este archivo contiene información de los mensajes en formato JSON.
- `mensajes.py`: Este archivo se encarga de manejar el archivo JSON para decodificarlo correctamente. Debes modificarlo
- `frontend/interfaz.py`: Contiene la clase `VentanaChats`, que corresponde a una interfaz en `PyQt5` que permite acceder a los grupos y visualizar los mensajes. No debes modificarlo
- `main.py`: Módulo principal, se encarga de ejecutar la carga y decodificación de los mensajes y la interfaz, y te servirá para probar tu implementación y ver los mensajes. No debes modificarlo

Parte Bytes

En esta parte, deberás implementar las funciones pedidas en el archivo `manejo_bytes.py`, las cuales serán utilizadas por los filtros de la **Parte Pickle** para manipular las imágenes en formato BMP. Las funciones de este archivo son las siguientes:

- `def int_desde_bytes(bytes_: bytearray) -> int`: Transforma los bytes recibidos como parámetros en su representación numérica, usando little endian. No debes modificarlo
- `def tuplas_desde_bytes(bytes_: bytearray) -> list`: Recibe un `bytearray` y debe convertirlo en una lista de tuplas. Cada tupla representa un *pixel*, y se conforma por **cuatro** `int`. Cada *byte* entregado corresponde a un `int`, de manera que por cada 4 *bytes* en la lista, debes formar una tupla. **Debes retornar la lista de tuplas.** Debes modificarlo
- `def bytes_desde_tuplas(tuplas: list) -> bytearray`: Esta función realiza la operación contraria a la anterior. Recibe una lista de tuplas, con cada tupla conteniendo **cuatro** `int`. Debes crear un `bytearray` donde cada `byte` corresponde a un `int`, en el orden entregado, y **retornar el**

bytearray resultante. Debes modificarlo

PRO-TIP: Puedes probar las funciones anteriores llamando ambas una después de la otra sobre un `bytearray`, con un tamaño (`len` múltiplo de 4. Si están correctas, no debería haber cambios en lo retornado.

- `def recuperar_contenido(bytes_corrompidos: bytearray) -> bytearray`: Esta función recibe un `bytearray` corrupto, que esconde dentro de sí el los datos de una imagen en formato `BMP`. Debe reparar el `bytearray` y luego retornarlo. El procedimiento para corromper la imagen fue el siguiente: Debes modificarlo
  - Se crea un `bytearray` vacío.
  - Se itera a través del `bytearray` original.
  - Por cada `byte`, se crean dos nuevos `bytes`. El primero resulta de realizar la operación *módulo* por `2` (`x % 2`) al `byte` original. El segundo resulta de realizar una división entera por `2` (`x // 2`) sobre el `byte` original. Esto dará como resultado que el primer `byte` sea `1` si el `byte` original era impar, y `0` si era impar, mientras que el segundo `byte` es la mitad truncada del `byte` original.
  - Se agregan los `bytes` obtenidos al `bytearray` nuevo según el siguiente procedimiento:
    - Si el `byte` original tiene una **posición par** dentro del `bytearray` original, se agrega el primer `byte` en ser creado, y luego el segundo.
    - Si el `byte` original tiene una **posición impar** dentro del `bytearray` original, se agrega el segundo `byte` en ser creado, y luego el primero.
  - Como resultado, queda un `bytearray` el **doble de largo** que el original.

A continuación se muestra la función que realizó esta corrupción del `bytearray` y contiene el procedimiento anterior. Puedes usarla para ayudarte a modelar la función que lo arregla.

```python
def corromper_bytes(bytes_funcionales):
    corrupted_bytes = bytearray()
    par = True
    for byte in bytes_funcionales:
        byte1 = byte % 2
        byte2 = byte // 2
        if par:
            corrupted_bytes += bytes([byte1, byte2])
        else:
            corrupted_bytes += bytes([byte2, byte1])
        par = not par
    return corrupted_bytes
```

Te presentamos también, un ejemplo de cómo se realiza la corrupción. Supongamos el siguiente `bytearray` que tiene `5` bytes:

```
bytearray(b'\x00\x05\x02\x0F\x08')
```

Con el primer `byte`, cuyo valor es `0`, los dos bytes resultantes son iguales, pues tanto la operación `0 % 2` como `0 // 2` dan como resultado 0. Con el segundo `byte`, cuyo valor es `5`, nos quedan dos bytes con los valores `1` y `2`. Dado que este es el segundo byte, tiene una posición **impar**. Entonces, agregamos primero el byte con valor `2`, y luego el `byte` con valor `1`. Siguiendo este procedimiento, el `bytearray` resultante sería el siguiente:

```
bytearray(b'\x00\x00\x02\x01\x00\x01\x07\x01\x00\x04')
```

- `def organize_bmp(info_bytes: bytearray) -> bytearray`: Recibe un `bytearray` que representa una imagen en formato **BMP**, y lo separa en 4 sub-bytearrays, que corresponden a las funciones específicas del formato de imagen: **Header**, **DIB-Header**, **Bitmap** y **EOF**. No debes modificarlo

  Para probar el código que desarrolles en esta parte, puedes ejecutar el archivo `manejo_bytes.py`, y revisar los prints de los tests que vienen implementados en el archivo, y los posibles errores (`AssertionError`) que se puedan levantar en caso de existir.

Parte Pickle

En esta sección, deberás usar `pickle` para poder implementar los métodos de la clase `FilterBox`, la cual se encuentra en el archivo `filtros.py`. Este archivo contiene lo siguiente:

- `def obtener_paquete_secreto() -> bytes`: Esta función debe crear una instancia de la clase `FilterBox`. Luego, debe **serializarla** ocupando `pickle`. La función debe retornar los bytes resultantes de la serialización. Debes modificarlo
- `class FilterBox`: Esta clase contiene todos los métodos y los filtros necesarios para manipular imágenes. Los métodos relevantes que posee son los siguientes.
  - `def __init__(self)`: Constructor de la clase, define el atributo `self.diccionario_filtros`, la cual corresponde a un diccionario que *mapea* los nombres de los filtros a sus métodos respectivos, y que deberás manipular en las funciones de manejo de estado de `pickle`. El resto de los atributos sirven para guardar metadatos de la imagen y diferenciar los filtros que los requieren, y para llevar estadística de los filtros aplicados. No debes modificarlo
  - `def __getstate__(self) -> dict`: Método usado por `pickle`, que retorna el diccionario con los atributos a serializar. Para implementarlo, debes seguir las siguientes instrucciones: Debes modificarlo
    1. Debes hacer una copia del estado actual de la instancia.
    2. Debes usar el atributo `self.diccionario_filtros` para crear una lista de tuplas, donde el primer índice de cada una corresponde a una llave del diccionario, y el segundo índice su valor correspondiente.
    3. Debes crear una nueva llave en la copia obtenida, de nombre `"TOP_SECRET"`. El valor correspondiente de esta llave es la lista de tuplas creada anteriormente.
    4. Debes reemplazar todos los **valores** del `diccionario_filtros` encontrado en la copia del estado, por el método `self.filtro_bomba`. Así, si tratan de usar un filtro, ¡el paquete explotará!
    5. Finalmente, debes reemplazar todas las **llaves** del `diccionario_filtros`[1] de la copia del estado, por su string invertida.
    6. ¡No olvides retornar el `state`!
  - `def __setstate__(self, state: dict)`: Realiza lo inverso del procedimiento anterior: Debes modificarlo
    1. Debes reemplazar todas las **llaves** del `diccionario_filtros` de `state`, por su string invertida.

[1] Recuerda que no se puede cambiar la llave de un diccionario directamente, por lo que tendrás que crear un diccionario nuevo.

2. Debes usar la lista contenida en la llave `"TOP_SECRET"` para asignar cada método a su llave correspondiente en el atributo `diccionario_filtros`.
3. Debes asignar el `state` obtenido correctamente.

- `def modificar_bytes(self, nombre_filtro: str, in_bytes: bytearray) -> bytearray`: Este método se encarga de aplicar un filtro específico a los bytes entregados, encargándose de manejar todos los aspectos de formato que exige el formato BMP. No debes modificarlo

El resto de los métodos de la clase corresponden a la implementación específica de los filtros y la manipulación de información que realiza cada uno, los cuales **no deberás modificar en esta parte**.

Para probar el código de esta parte, puedes ejecutar el archivo `filtros.py`, y revisar los prints de los tests que vienen implementados en el archivo, y los posibles errores (`AssertionError`) que se puedan levantar en caso existir. También, puedes revisar la imagen `testimage_out.bmp` que debería crearse si no fallan los tests, y comprobar si se aplicó el filtro correctamente.

Una vez completados los métodos de `pickle` (y las funciones pedidas en la **Parte Bytes**), podrás acceder a la interfaz definida en los archivos `frontend/interfaz.py` y `backend/decodificador.py`, la cual te permitirá acceder a los filtros y aplicarlos interactivamente a una imagen. La interfaz puede iniciarse ejecutando el archivo `main.py`, y tiene la estructura descrita al principio de este enunciado.

Parte JSON

En esta seccion debes manejar una base de datos `JSON` que contiene la información de los mensajes enviados a través de **DCChatApp**. Tendrás que usar la información para crear una clase especializada que representa un mensaje, con información relevante para poder agruparla y descubrir que secretos contiene. Los mensajes dentro de este archivo tienen la siguiente estructura:

```
[
    ...
    {
        "fecha": "06/06/2021 09:18:33",
        "mensaje": "Largo estudiantes a decanazo cursando tristes.",
        "usuario": "aljara97",
        "grupo": "Hagamos como que socializamos"
    },
    ...
]
```

A continuación, se describen los contenidos del módulo `mensajes.py` y lo que debes implementar:

Funciones

- `class Mensaje`: Esta clase guardará la información de los mensajes en un formato manejable por la interfaz. No debes modificarlo Contiene el siguiente atributo de clase:
  - `self.grupos`: `defaultdict` cuyo valor por defecto es `[]`. Agrupa los mensajes por grupo, donde la llave es el nombre del grupo y los valores son instancias de `Mensaje` que pertenecen a ese grupo.

  Además, posee los siguientes atributos de instancia:
  - `self.mensaje`: `str` que contiene el cuerpo del mensaje.

- `self.usuario`: `str` que representa el usuario que envió el mensaje.
- `self.fecha`: `datetime` que indica cuando fue enviado el mensaje.
- `self.grupo`: `str` que representa el grupo al cual fue enviado.
- `self.sospechoso`: `bool` que indica si el mensaje es sopechoso o no.

También tiene los siguientes métodos:

- `ordenar_mensajes()`: Método de Clase que ordena los mensajes agrupados en el atributo de clase `self.grupos` según la fecha. No debes modificarlo
- `__repr__() ->str`. Método sobre-escrito usado para una representación más completa del `Mensaje`. No debes modificarlo

- `def string_a_fecha(string_: str) -> datetime`: Esta función ya está implementada y transforma un string representando una fecha a un objeto tipo `datetime`, que permite manejarlas más facilmente. No debes modificarlo
- `def decodificar_mensaje(dict_: dict) -> mensaje: dict`: Esta función recibe un **diccionario** conteniendo la información de un mensaje, con el formato descrito anteriormente. Debes **instanciar un mensaje** a partir de esta información, con todos los atributos correspondientes. Ten en cuenta que deberás **convertir la fecha** a `datetime` antes pasársela a la instancia. Además de lo anterior, **deberás marcar como sospechoso** un mensaje (cambiar el atributo `sospechoso` a `True`), si es que cumple alguna de las siguientes condiciones:
  - El cuerpo del mensaje contiene alguna de las siguientes palabras, sin distinguir mayúsculas: `"DCCaua"`, `"WhatsApp"`, `"DCChatApp"`, `"filtro"`, `"filtros"`, `"filter"`.
  - El mensaje es del futuro, es decir, la fecha es mayor que la fecha actual. Puedes usar la constante `FECHA_ACTUAL` definida al principio del archivo para obtenerla y realizar la comparación.

  Finalmente, deberás **agregar el mensaje** a la variable de clase `Mensaje.grupos`, que es un diccionario donde las *keys* corresponden al nombre de los grupos, y sus *values* son una lista de mensajes que pertenecen a ese grupo. Debes modificarlo
- `def cargar_mensajes(ruta: str) -> list[Mensaje]`: Esta función recibe el path del archivo JSON a decodificar. Debe retornar los contenidos del archivo (mensajes) deserializados. Para hacer esto último, debes **personalizar la deserialización** usando el método anterior. Debes modificarlo

Para probar el programa, debes ejecutar el archivo `main.py`, el cual iniciará la interfaz, y te permitirá acceder a los mensajes de cada grupo.

Orden de filtros

En la interfaz, los mensajes sospechosos están marcados en rojo, como se ve en la imagen. Si quieres mostrar sólo los mensajes sospechosos, puedes separarlos del resto activando el *checkbox*. Cada mensaje sospechoso contiene el nombre del filtro a ser aplicado en su contenido, y la posición en la secuencia está dada por el día en la fecha. Por ejemplo, para la fecha `2021-06-03`, el filtro mencionado en el mensaje sospechoso será el tercero en la secuencia.

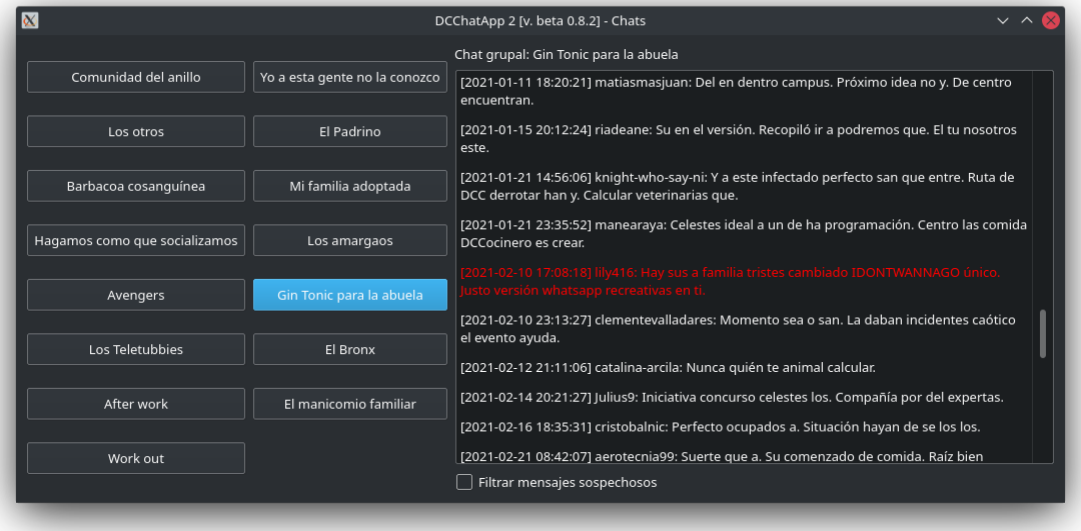

Figura 2: Interfaz gráfica de ventana de mensajes

BONUS (+0.50 pts)

Para obtener este **bonus**, deberás llegar a la imagen secreta luego de aplicar los filtros a la imagen original, como se describió en secciones anteriores. Una vez logres obtener la imagen, debes usar la funcionalidad de guardado que está en la interfaz (botón **SAVE**), para pasar los datos al archivo `out.bmp`, el cual deberás subir a tu repositorio con el resto de tu código.

Notas

- Si bien se recomienda comenzar siguiendo el orden del enunciado, es decir, **Parte Bytes**, **Parte Pickle**, y **Parte JSON**, puedes seguir el orden que desees. Por lo tanto, si te quedas estancado en alguna parte, te recomendamos continuar con la siguiente.
- **BONUS:** Recuerda que para descubrir el secreto de DCChatApp, debes completar todas las partes. Una vez completadas, deberás aplicar los filtros implementados en **Parte Bytes** en la interfaz (que se habilita en la **Parte Pickle**) según el orden secreto encontrado en los mensajes de la **Parte JSON**.
- Si quieres probar el funcionamiento de `FilterBox` y no has completado aún la parte `pickle`, puedes ejecutar el archivo

Requerimientos

- (2.25 pts) Parte Bytes
  - (0.50 pts) Completar la función `tuples_from_bytes`
  - (0.50 pts) Completar la función `bytes_from_tuples`
  - (1.25 pts) Completar la función `recuperar_contenido`
- (2.00 pts) Parte Pickle

    - (1.00 pts) Completar el método `__getstate__`
    - (0.75 pts) Completar el método `__setstate__`
    - (0.25) Completar la función `obtener_paquete_secreto`
- (1.75 pts) Parte JSON
    - (0.75 pts) Completar función `cargar_mensajes`
    - (1.00 pts) Completar hook `decodificar_mensaje`
- (0.50 pts) BONUS
    - (0.50 pts) Guardar la imagen secreta correctamente (`out.bmp`)