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

IIC2233 Programación Avanzada (2024-1)

Tarea 4

Entrega

- **Tarea y** `README.md`
  - **Fecha y hora oficial (sin atraso):** martes 18 de junio de 2024, 20:00
  - **Fecha y hora máxima (2 días de atraso):** viernes 21 de junio de 2024, 20:00.
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T4/`.
    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 "`T4/`" 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

- Utilizar conceptos de interfaces y `PyQt6` para implementar una aplicación gráfica e interactiva.
- Traspasar decisiones de diseño y modelación en base a un documento de requisitos.
- Diseñar e implementar una arquitectura cliente-servidor. Además, en el cliente se debe entender y aplicar los conceptos de *back-end* y *front-end*.
- Aplicar conocimientos de *threading* en interfaces (`Qtimer` y/o `QThread`).
- Aplicar conocimientos de señales.
- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores (*networking*).
- Aplicar conocimientos de manejo de *bytes* para seguir un protocolo de comunicación preestablecido.

Índice

1. *DCCome Lechuga*

Luego de las emocionantes votaciones democráticas en **DCCiudad**, un sorprendente resultado abatió a todos los ciudadados... ¡**Pepa** salió electa alcaldesa! Sin embargo esto, su mandato no sería fácil, pues tendría que reconstruir desde las ruinas a **DCCiudad**. De esta forma, comienza un proceso de **cosecha y tala** con tal de poder volver una vez más las tierras de cultivo fértiles.

Ayuda a **Pepa** en este proceso de tratamiento de suelos con un toque artístico, permitiéndole tanto generar comida para sus ciudadanos y, así mismo, deleitarlos con la vista de una nueva y renovada **DCCiudad**.

Figura 1: Logo de *DCCome Lechuga*.

2. Flujo del programa

DCCome Lechuga es un juego que tiene como objetivo principal resolver distintos *puzzles*, de tal manera que el jugador deberá guiar a **Pepa** por el mapa con el propósito de resolverlos y así encontrar la imagen oculta. Para cumplir esta misión, deberás crear una interfaz gráfica que permita controlar el movimiento de **Pepa** mediante las teclas **WASD**, y utilizar la tecla **G** para interactuar con el mapa. Además, el usuario deberá interactuar con ciertos objetos del juego mediante el uso de *clicks*.

El programa debe contar tanto con una interfaz gráfica como con un servidor funcional que permita verificar la correctitud de las soluciones. **Lo primero que debe ejecutarse en la tarea es el servidor, seguido del cliente.** El jugador debe ser una instancia de la clase del cliente, la cual está encargada de la interfaz gráfica del jugador y será el único medio de comunicación con el servidor del programa.

El jugador interactuará con el programa a través de dos ventanas gráficas: Ventana de Inicio y Ventana de Juego. Al iniciar el programa, aparecerá la Ventana de Inicio, la cual le permite al usuario ingresar un **nombre de usuario**, seleccionar un *puzzle* existente y revisar el "Salón de la Fama", que exhibe los mejores puntajes registrados en el cliente. Una vez ingresado un nombre de usuario y *puzzle* válido, se abrirá la Ventana de Juego en la cual el usuario puede resolver *puzzle* seleccionado, y revisar el tiempo restante que tiene para resolverlo.

Para corroborar que el usuario resolvió correctamente el *puzzle* en la Ventana de Juego, deberá enviar su solución a un servidor y este validará si la respuesta es la esperada o no. En caso de que la solución sea correcta, se registrará el puntaje en un archivo `puntaje.txt`; en caso contrario, se permitirá seguir jugando siempre que tenga tiempo de juego disponible. Si el tiempo de juego se acaba, se debe volver inmediatamente a la Ventana de Inicio. Finalmente, si el usuario sale del programa antes de resolver exitosamente el *puzzle*, no se guarda la información de la partida.

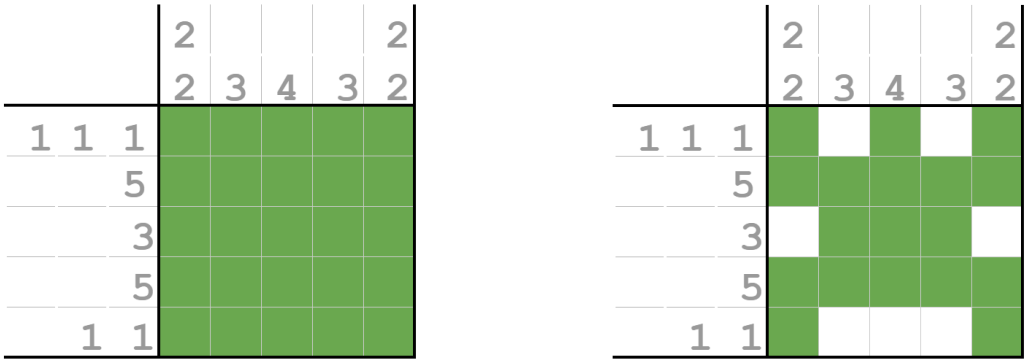
3. Mecánicas de Juego

3.1. *Puzzle*

Al iniciar el *puzzle*, se mostrará en la ventana un tablero de dimensión $n \times n$. Estas dimensiones son variables y hacen referencia a la dificultad de cada *puzzle*. Esto se verá en detalle en la parte de Puntaje. El objetivo del juego es, a partir de un conjunto de números, disponer de las casillas como llenas o vacías para así revelar una imagen oculta.

Los conjuntos de números se encontrarán en los bordes superior e izquierdo del tablero, y representan la cantidad de celdas consecutivas que contendrán lechugas, y por ende llenas. En caso de tener 2 o más números en una misma fila o columna, indica que debe existir al menos un espacio entre cada conjunto de celdas con lechugas. En caso de que no haya lechugas en una fila o columnas, el número correspondiente a dicha fila/columna contendrá únicamente un guión (-).

El *puzzle* parte inicialmente lleno de lechugas en su totalidad, es decir con todas las casillas llenas. El usuario deberá mover a **Pepa** para coma las lechugas en caso de querer dejar una casilla en blanco.



(a) *Puzzle* inicial (b) *Puzzle* resuelto

Figura 2: Ejemplo de un *puzzle* de tortuga.

Como se ve en el ejemplo anterior:

- La tercera fila únicamente contiene el número 3, lo que indica que dicha fila solo tiene un único conjunto de 3 casillas consecutivas con lechugas. Notar que el conjunto de casillas no debe estar necesariamente pegado directamente a alguno de los bordes del tablero.
- La quinta fila tiene dos números 1, señalando que debe en esa fila debe haber 1 celda con lechuga, luego **al menos** un espacio de separación y luego otra celda con 1 lechuga.

3.2. Modo de Juego

En el modo de juego el usuario debe ser capaz de mover a **Pepa** en cuatro direcciones (arriba, abajo, izquierda, derecha) según las especificaciones en Teclado, además de poder hacer que **Pepa** coma una lechuga o haga *poop* dependiendo de si se quiere llenar o vaciar una celda.

Para la resolución del *puzzle* se contará con `TIEMPO_JUEGO` para completarlo, no obstante será posible aumentar este tiempo gracias las **sandías**. Las sandías aparecerán en secciones aleatorias de la ventana, las cuales al presionarlas con un *click* sumarán un tiempo `TIEMPO_ADICIONAL` a la cuenta regresiva, permitiendo de esta forma tener tiempo extra para al resolución del *puzzle*. Así mismo, debe ser posible **pausar** el juego mediante un botón de pausa que bloquee la vista e interacción del tablero.

Cuando el tablero esté resuelto, el usuario deberá apretar el botón de validación para así verificar el resultado en el servidor. En caso de que la solución del *puzzle* sea correcta, se le debe informar al usuario y posteriormente volver a la Ventana de Inicio. En caso que la solución sea incorrecta, se le debe informar al usuario y continuar con la resolución del *puzzle*, así como con la cuenta regresiva.

Finalmente, en caso de no lograr terminar en el tiempo disponible, el juego termina **indicando que se perdió por falta de tiempo,** para luego volver a la Ventana de Inicio.

3.3. Puntaje

Al iniciar un *puzzle* empieza la cuenta regresiva determinada por `TIEMPO_JUEGO`, la cual debe estar en segundos. Este tiempo podrá verse aumentado si se hace *click* en una de las sandías que aparecen en el mapa. **Esto no afecta el tiempo que el usuario lleva utilizado en el *puzzle*.**

En caso de solucionar el *puzzle* exitosamente, se calculará un puntaje en función del tiempo utilizado para completar el *puzzle*, en segundos (`int`); el tiempo restante de la cuenta regresiva, en segundos (`int`); y las dimensiones del tablero (`int`). La formula para calcular el puntaje es la siguiente y el resultado de este **siempre se debe redondear al segundo decimal**:

$$puntaje = \frac{tiempo\_restante \times n \times n \times \texttt{CONSTANTE}}{tiempo\_utilizado}$$

Cada vez que se solucione un *puzzle* se deberá guardar el nombre de usuario y puntaje obtenido en un archivo llamado `puntaje.txt`. Cabe destacar que en este archivo debe existir una línea por cada partida jugada, **permitiendo repeticiones para distintas jugadas del mismo usuario**.

3.4. Fin del juego

Al terminar el juego, se volverá a la Ventana de Inicio, donde se debe mostrar el "Salón de la Fama" con los actualizados.

En caso de que el usuario desee resolver otro puzzle, deberá volver a ingresar un nombre de usuario y seleccionar el *puzzle* a resolver.

4. Entidades

4.1. Pepa

Pepa es el único personaje de *DCCome Lechuga*, el cual puede ser controlado por el usuario mediante el uso del teclado. El movimiento de este personaje está limitado por las casillas del tablero, permitiendo que **Pepa** únicamente se pueda desplazar hacia arriba, abajo, izquierda y derecha, dentro de la dimensión $n \times n$ del tablero. Cabe destacar que los movimientos de **Pepa siempre** estarán delimitados por el *puzzle*, esto quiere decir, no puede salirse del tablero.

El desplazamiento de este personaje es **discreto**, pero la animación del movimiento es **continua**. Es decir, el personaje se mueve únicamente de una casilla a otra y jamás quedará en medio de dos casillas; sin embargo, el movimiento debe verse de forma fluida y continua. Para esto, el movimiento entre casillas debe mostrarse como una animación correspondiente a la dirección en que se está moviendo, con una cantidad de **4 *sprites***, los cuales deben intercalarse entre sí para animar el movimiento entre casillas, de manera similar a este ejemplo. En caso de no estar en movimiento, se debe mostrar un *sprite* estático.

Adicionalmente, **Pepa** tiene la capacidad de comer lechugas y de hacer *poop* cuando está situada en una casilla del *puzzle*. Si la casilla posee una *sprite* de lechuga -es decir se encuentra llena-, **Pepa** se la comerá

y dejará la casilla en blanco. En cambio, si la casilla está en blanco, **Pepa** hará *poop*, lo cual implica la aparición de una *sprite* de popó, y al cabo de `TIEMPO_TRANSICION` segundo(s) la popó se volverá lechuga instantáneamente, es decir, no se requiere animar esta transición.

4.2. Sandía

Durante una partida, el usuario podrá capturar sandías que aparecerán en una posición aleatoria dentro de los márgenes interiores de la ventana. Estas aparecerán cada `TIEMPO_APARICION` segundos y su captura resultará en adicionar `TIEMPO_ADICIONAL` segundos al tiempo restante del *puzzle*. Su aparición dentro de la ventana es temporal, por ende, esta desaparecerá cuando el usuario haga ***click*** sobre ella, o pasen `TIEMPO_DURACION` segundos desde su aparición en la ventana, siendo una acción opcional para el usuario.

5. Interfaz Gráfica

5.1. Modelación del programa

Se evaluará, entre otros, los siguientes aspectos:

- Correcta **modularización** del programa, lo que incluye una adecuada separación entre ***back-end*** y ***front-end***, con un **diseño cohesivo** y de **bajo acoplamiento**. Así mismo, la separación **cliente** y **servidor** debe estar definida.
- Correcto uso de **señales** entre ***back-end*** y ***front-end***, además ***threading*** para modelar todas las interacciones en la interfaz.
- Presentación de la información y funcionalidades pedidas (puntajes o *cheatcodes*, por ejemplo) a través de la **interfaz gráfica**. Es decir, **no se evaluarán** *ítems* que solo puedan ser comprobados mediante la terminal o por código, a menos que la pauta lo explicite.
- Generación de las ventanas mediante código programado por el estudiante. Es decir, no se permite la creación de ventanas con el apoyo de herramientas como QtDesigner, entre otros.

5.2. Ventanas

Para la correcta implementación de *DCCome Lechuga*, se espera la creación de al menos **dos ventanas** que serán mencionadas a continuación, junto con los **elementos mínimos** que se solicitan en cada una.

Los ejemplos de ventanas expuestos en esta sección son simplemente para que te hagas una idea de cómo se deberían ver y no es necesario que tu tarea se vea exactamente igual ni que presente la misma composición.

5.2.1. Ventana de Inicio

Es la primera ventana que se debe mostrar cuando el usuario ejecute el programa, la cual deberá contener como mínimo:

(A) El logo de *DCCome Lechuga*.

(B) Una **línea de texto** editable para ingresar el **nombre de usuario**.

(C) Un **menú desplegable** para seleccionar el *puzzle* a resolver.

(D) Una sección para el "**Salon de la Fama**", que muestre el *ranking* de los mejores **puntajes** históricos guardados en el cliente. Este deberá ser una sección de tipo ***scroll***.

(E) Un **botón** para comenzar la partida.

(F) Un **botón** para salir del programa. Este botón debe ser independiente del botón cerrar que presentan las ventanas por defecto.

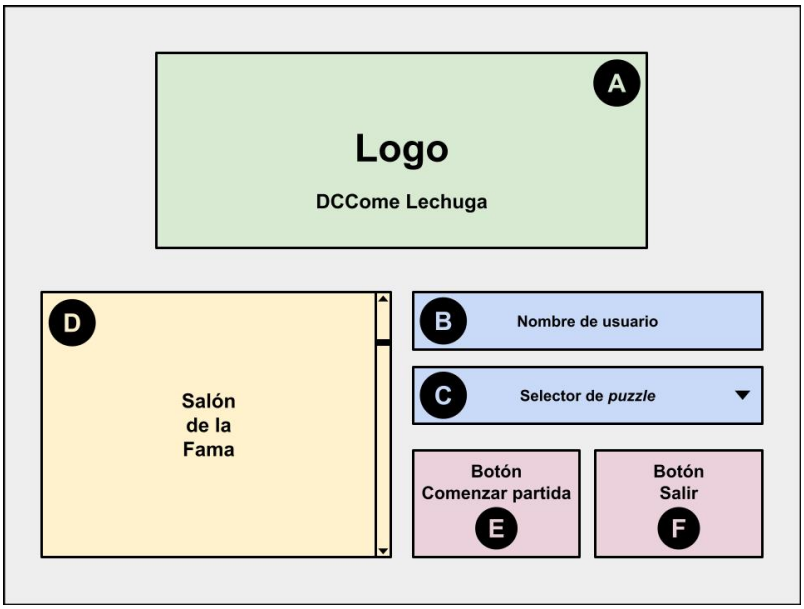


Figura 3: Esquema de ejemplo para Ventana de Inicio.

Cuando el usuario seleccione el mapa de *puzzle* (C) se deberá mostrar el nombre de **todos los archivos y sólo los archivos** que se encuentren en la carpeta `base_puzzles`. Luego al presionar el botón para comenzar la partida (E), se deberá verificar antes que el nombre de usuario ingresado en (B) sea **alfanumérico**, contenga **al menos una mayúscula y un número, que no sea vacío**. Todas las verificaciones las deberá llevar a cabo el **cliente**.

En caso de no cumplir con alguna condición, **se deberá notificar al cliente** mediante un mensaje de error o *pop up* en la interfaz, mencionando el motivo del error. Si cumple todos los requerimientos para el nombre de usuario, se cerrará la ventana y automáticamente se abrirá la Ventana de Juego. Por último, si se selecciona la opción de salir (F), se deberá cerrar la ventana de inicio y cerrar el programa.

El "**Salón de la Fama**"(D) deberá mostrar todos los puntajes de las personas que han jugado *DCCome Lechuga*, el en formato `nombre_usuario - puntaje_obtenido`. Los nombres de usuario y los puntajes deberán ser ordenados de manera descendente, es decir, el primer puntaje será el más alto y el último puntaje será el más bajo.

5.2.2. Ventana de Juego

En esta ventana se desarrolla el flujo del juego y debe estar compuesta por dos secciones principales: **estadísticas del juego** y el ***puzzle*** (A).

La primera sección deberá mostrar la **cuenta regresiva** del juego (B), además de tres botones: uno para en **comprobar** la solución del usuario (C), otro para **pausar** el juego (D), y el último para **salir** del programa (E).

La segunda sección se compone del *puzzle* del juego, donde se llevarán a cabo todas las mecánicas del juego, estando presentes los distintos elementos y entidades del juego. Acá se deberá mostrar el *puzzle* en el centro, en la parte superior de este la cantidad de casillas por columna que deben tener una lechuga, y a la izquierda del *puzzle* deben aparecer la cantidad de casillas por fila que se espera, tengan lechugas.

Siempre que el usuario posea tiempo podrá seguir jugando; de lo contrario, se deberá mostrar un mensaje o *pop-up* en la ventana indicando el término del juego y volviendo a la ventana inicial.

Una vez que el usuario finalice el juego, se deberá mostrar un mensaje o *pop-up* señalando su victoria o derrota y el puntaje final.

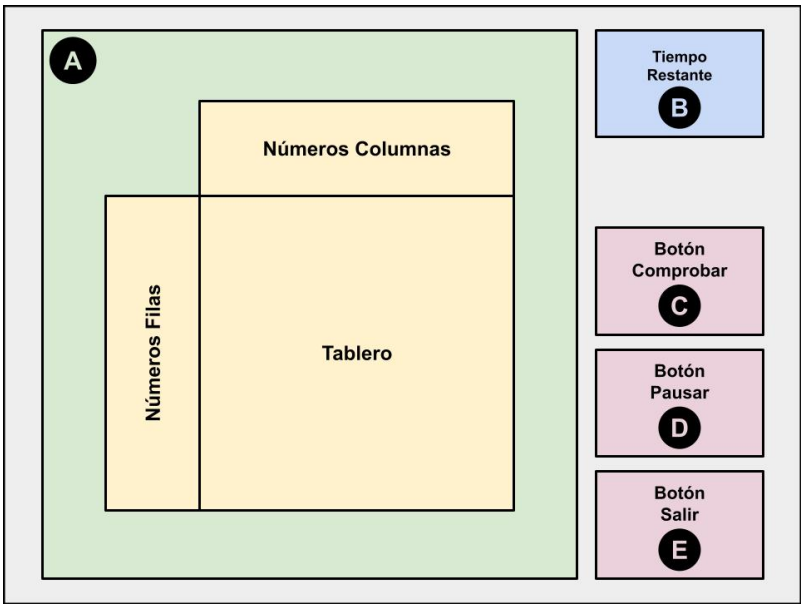


Figura 4: Esquema de ejemplo para Ventana de Juego.

6. Interacción

6.1. Teclado

El movimiento de **Pepa** es gatillado por el uso de las teclas `WASD` donde la tecla `W` es moverse hacia arriba, `A` hacia la izquierda, `S` hacia abajo y `D` hacia la derecha. Para marcar y desmarcar casillas se usara la tecla `G`.

6.2. *Click*

En caso de aparecer una sandía en alguna parte de la pantalla, el usuario debe poder hacer *click* en ella para ganar sus beneficios.

6.3. Botones

Se deberán mostrar 3 botones en pantalla: uno para pausar el tiempo, otro para validar la respuesta, y un último para **salir del programa**.

Al presionar el botón de pausa, tanto la cuenta regresiva, como el tiempo de las sandías, y el movimiento y animación de **Pepa** deben detenerse. Además, se debe ocultar el tablero y los números con el objetivo de impedir que el usuario siga resolviendo el *puzzle*. El juego debe reanudarse de nuevo si es que se presiona una vez más el botón de pausa.

Al presionar el botón de validar respuesta, se confirmará la solución con el servidor, donde en caso de ser correcta, se calcula el puntaje y en caso de ser incorrecta se le avisa al usuario y se sigue jugando.

6.4. *Cheatcodes*

Para esta tarea, y con el fin de ~~facilitar la corrección~~ mejorar la jugabilidad, se deberá poder ingresar *cheatcodes* que llevarán a cabo ciertas acciones. Estos *cheatcodes* se deberán activar cuando el usuario presionar de forma simultanea las teclas indicadas, o presionar de manera consecutiva la combinación de teclas. Queda a tu criterio cuál de las dos formas implementar.

Los *cheatcodes* a implementar son los siguientes:

- `I + N + F`: esta combinación da tiempo infinito para completar el *puzzle*. En este caso el puntaje del *puzzle* es por defecto `PUNTAJE_INF`.
- `M + U + T + E`: esta combinación desactiva el uso de todos los sonidos **durante la partida actual**, perdiendo su efecto si se termina la partida y se empieza otra.

6.5. Sonido

Al momento de que **Pepa** coma una lechuga, se deberá reproducir `comer.wav`; por lo contrario, cuando haga *poop*, se deberá reproducir `poop.wav`.

En caso de que el usuario haga *click* sobre una sandía, se deberá reproducir `obtener_sandia.wav`. Si el usuario resuelve exitosamente un *puzzle*, se deberá reproducir `juego_ganado.wav` y en caso de perder, se deberá reproducir `juego_perdido.wav`.

Por último, a lo largo de todo el juego se deberá reproducir una música de fondo, esta puede ser `musica_1.wav` o `musica_2.wav`. Queda a criterio del estudiante qué canción utilizar, mientras haya música de fondo de forma constante.

7. *Networking*

Para comprobar si un *puzzle* de *DCCome Lechuga* fue resuelto correctamente, tendrás que enviar la solución del usuario a un servidor utilizando todos tus conocimientos de *networking*. Deberás desarrollar una arquitectura `cliente` - `servidor` con el modelo **TCP/IP** haciendo uso del módulo **socket**.

Tu misión será implementar dos programas separados, uno para el **`servidor`** y otro para el **`cliente`**. De los mencionados, **siempre** se deberá ejecutar primero el servidor y este quedará escuchando para que se puedan conectar uno o más clientes. Debes tener en consideración que la comunicación es siempre entre cliente y servidor.

Para efectos de esta tarea y su corrección, se considerará el manejo de un usuario a la vez, pero el servidor deben ser capaz de manejar distintos usuarios consecutivos.

7.1. Arquitectura cliente-servidor

Las siguientes consideraciones **deben ser cumplidas al pie de la letra**. Lo que no esté especificado aquí puedes implementarlo según tu criterio, siempre y cuando se cumpla con lo solicitado y no contradiga nada de lo indicado. Si algo no está especificado o si no queda completamente claro, puedes **preguntar aquí**.

7.1.1. Separación funcional

El cliente y el servidor deben estar separados. Esto implica que deben estar en directorios diferentes e independientes, uno llamado **`cliente`** y otro llamado **`servidor`**. Cada directorio debe contar con los archivos y módulos necesarios para su correcta ejecución, asociados a los recursos que les correspondan, además de un archivo principal **`main.py`**, el cual inicializa cada una de estas entidades funcionales. La estructura que debes seguir se indica en el siguiente diagrama:

```
T4
├── cliente
│   ├── main.py
│   ├── backend
│   │   └── ...
│   └── frontend
│       └── ...
├── servidor
│   ├── main.py
│   └── ...
├── .gitignore
└── README.md
```

Si bien las carpetas asociadas al `cliente` y al `servidor` se ubican en el mismo directorio (`T4/`), la ejecución del `cliente` **no debe depender de archivos en la carpeta del `servidor`**, y la ejecución del `servidor` **no debe depender de archivos y/o recursos en la carpeta del `cliente`**. Esto significa que debes tratarlos como si estuvieran ejecutándose en computadores diferentes. La Figura 5 muestra una representación esperada para los distintos componentes del programa:

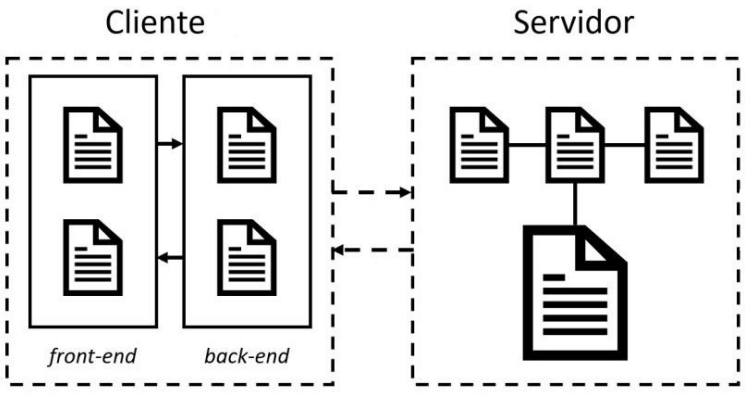


Figura 5: Separación cliente-servidor y *front-end*/*back-end*.

Cabe destacar que **solo el cliente tendrá una interfaz gráfica**. Por lo tanto, todo cliente debe contar con una separación entre *back-end* y *front-end*, mientras que la comunicación entre el `cliente` y el `servidor` debe realizarse mediante *sockets*. Ten en cuenta que la separación deberá ser de carácter **funcional** según lo indicado en Roles.

7.1.2. Conexión

El `servidor` contará con un archivo de formato `JSON`, ubicado en la carpeta del `servidor`. Este archivo debe contener el *host* para instanciar un *socket*. El archivo debe llevar el siguiente formato:

```
{
    "host": "direccion ip",
    ...
}
```

Por otra parte, el `cliente` deberá conectarse al *socket* abierto por el `servidor` haciendo uso de los datos encontrados en el archivo `JSON` de la carpeta del `cliente`. La selección del puerto del *socket*, tanto para

el `cliente` como para el `servidor`, se deberá hacer por medio de **argumento de consola**, pasando el puerto como argumento al ejecutar el archivo `.py`. Por ejemplo, si se quiere elegir el puerto 8000, se deberá ejecutar el archivo en consola de la siguiente forma: `python main.py 8000`.

Es importante recalcar que el `cliente` y el `servidor` **no deben usar el mismo archivo** `JSON` para obtener los parámetros.

7.1.3. Método de codificación

Cuando se establece la conexión entre el **cliente** y el **servidor**, deberán encargarse de intercambiar información constantemente entre sí. Por ejemplo, el **cliente** le comunica al **servidor** la solución del *puzzle* y este le responderá si el resultado corresponde al esperado. Como no queremos que un jugador intente *hackear* el mensaje para alterarlo, debemos asegurarnos de codificar el contenido antes de enviarlo, de tal forma que si alguien lo intercepta no pueda descifrarlo. Deberás codificar los mensajes enviados entre el cliente y el servidor según la siguiente estructura:

- Los primeros 4 *bytes* indican el **largo del mensaje**, los cuales deben ser enviados en el formato *big endian*[1].
- A continuación, debes enviar el contenido del mensaje. Este debe separarse en *chunks* de 25 *bytes*, los cuales deben ser precedidos por otros 3 *bytes* que indiquen el número de bloque enviado partiendo desde el cero y codificados en *big endian*.
- Si para el último bloque no alcanza el mensaje para completar los 25 *bytes*, deberás rellenar los espacios restantes del bloque con *bytes* ceros (`b'\x00'`).

Finalmente, el mensaje completo que se envía se verá de la siguiente forma:

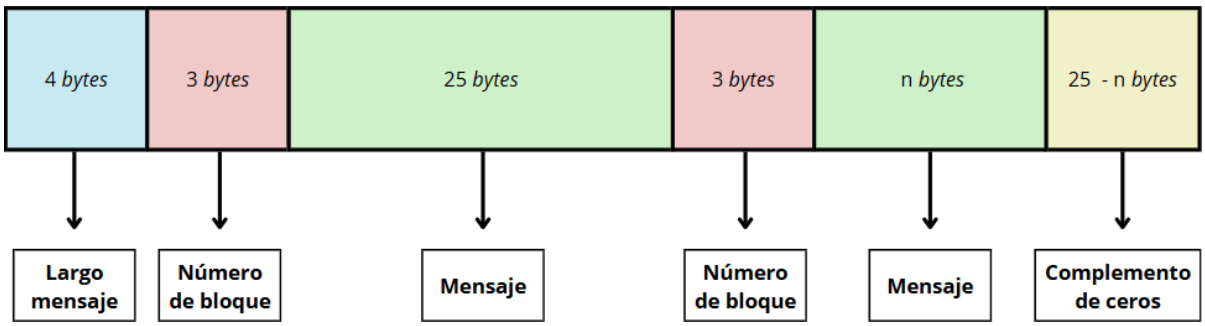


Figura 6: Ejemplo de un *bytearray* codificado.

7.1.4. Desconexión repentina

En caso de que el servidor o algún cliente se desconecte, ya sea por error o a la fuerza, tu programa debe reaccionar para resolverlo:

- Si es el **`servidor`** quien se desconecta, cada cliente conectado debe mostrar un mensaje en la ventana (ya sea como texto plano o *pop-up*) explicando la situación, antes de cerrar el programa.
- Si es un **`cliente`** quien se desconecta, se descarta su conexión y se muestra en consola un mensaje que indica lo anterior.

[1] El *endianness* es el orden en el que se guardan los *bytes* en un respectivo espacio de memoria. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberás proporcionar el *endianness* que quieras usar, además de la cantidad de *bytes* que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

7.2. Roles

A continuación, se detallan las funcionalidades que deben ser manejadas por el `servidor` y las que deben ser manejadas por el `cliente`:

7.2.1. Servidor

- **Procesar y validar** la solución del *puzzle* completado por el usuario. Cuando el usuario envíe la información del *puzzle* completado con su solución e identificador del *puzzle*, el servidor comprobará su validez para luego enviarle una respuesta al cliente según si esta es válida o inválida.

7.2.2. Cliente

Todos los clientes cuentan con una interfaz gráfica que le permitirá interactuar con el programa y enviar mensajes al servidor. Maneja las siguientes acciones:

- **Enviar la información de los *puzzles*** completados por el usuario hacia el servidor.
- **Recibir e interpretar** las respuestas y actualizaciones que envía el servidor.
- **Actualizar** la interfaz gráfica de acuerdo a las respuestas que recibe del servidor.

8. Archivos

Para el desarrollo de la tarea, deberás hacer uso de los siguientes archivos entregados en la carpeta `assets/` de `cliente/`:

- `sprites/`: Corresponden a los elementos visuales.
- `sonidos/`: Corresponde a los sonidos utilizados en la interacción del personaje, la música de fondo, y efectos al ganar o perder algún *puzzle*.
- `base_puzzles/`: Corresponde a los *puzzles* que deberán ser mostrados durante el juego.

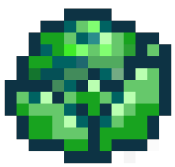
Y además los siguientes archivos en la carpeta `assets/` de `servidor/`:

- `solucion_puzzles/`: Corresponde a la solución de los *puzzles* jugables.

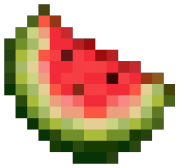
8.1. `sprites/`

Esta carpeta contiene todas las diferentes imágenes en formato `.png` que se ocuparán para tu tarea, estos corresponden a:

- Las entidades ó elementos con los que interactúa el personaje.



(a) Lechuga



(b) Sandía

(c) *Poop*

Figura 7: Elementos interactivos.

- Las imágenes de movimiento para el personaje.

Figura 8: Caminata de **Pepa** hacia abajo.

8.2. `sonidos/`

Esta carpeta contiene los efectos de sonido que deben implementarse en el juego. Se encuentran:

- `comer.wav`: Este deberá reproducirse cada vez que **Pepa** coma una lechuga.
- `poop.wav`: Este deberá reproducirse cuando se marque una casilla vacía, o sea, cuando **Pepa** haga *poop*.
- `obtener_sandia.wav`: Este deberá reproducirse cada vez que el usuario haga *click* encima de una sandía.
- `juego_ganado.wav`: Este deberá reproducirse cuando el usuario complete correctamente un *puzzle*.
- `juego_perdido.wav`: Este deberá reproducirse cuando el usuario no logre completar un *puzzle*.
- `musica_1.wav` y `musica_2.wav`: Es la música de fondo que deberá sonar mientras el juego esté abierto.

8.3. *Puzzles*

Para toda representación del estado de las casillas, un `1` indica una casilla que debe estar llena, y un `0` indica una casilla que debe permanecer en blanco, como se muestra a continuación:

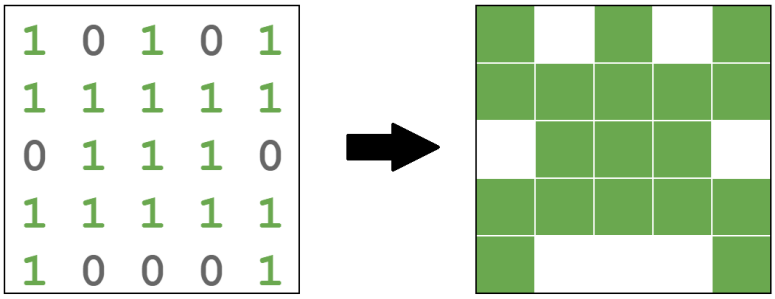


Figura 9: Ejemplo representación de los *puzzles*.

8.3.1. `base_puzzles/`

Esta carpeta contiene la información necesaria en cada archivo para resolver los distintos *puzzles*. Cada archivo `.txt` almacena en la primera línea representa los números correspondientes a las columnas, y en la segunda línea representa los números correspondientes a las filas. Dentro de una misma línea, cada columna o fila estará separada por el carácter `";"`; y en caso de que una misma columna o fila contenga varios números, estos estarán separados por el carácter `","`. **En caso de que una fila o columna no requiera de ninguna casilla llena, no aparecerá un cero (`0`), sino que un guión (`-`).**

A partir de lo descrito anteriormente, el archivo base del *puzzle* mostrado en el Figura 2: Ejemplo de un *puzzle* de tortuga. quedaría de la siguiente forma:

```
2,2;3;4;3;2,2
1,1,1;5;3;5;1,1
```

A partir de la información de estos archivos, se deberá calcular las dimensiones de los tableros.

8.3.2. `solucion_puzzles/`

Esta carpeta contiene la solución a los distintos *puzzles* que se deben poder jugar en tu aplicación. Cada archivo `.txt` tiene exactamente el mismo nombre del *puzzle* del que es solución, pero su contenido sólo son las `n` líneas solucionadas del *puzzle*. Como se mencionó anteriormente, las celdas que contengan un `1` indican casillas que deben estar llenas, mientras que las casillas con `0` indican celdas en blanco.

De esta manera, el archivo solución del *puzzle* mostrado en la Figura 2: Ejemplo de un *puzzle* de tortuga. quedaría de la siguiente forma:

```
10101
11111
01110
11111
10001
```

9. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T4/`.

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

- El enunciado.
- La carpeta `cliente/assets/`
- La carpeta `servidor/assets/`
- El archivo `BASE.md` (no confundir con el archivo **obligatorio** `README.md`)

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.

10. Importante: Corrección de la tarea

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 y que la funcionalidad esté correctamente integrada en el programa. En color **azul** se señalará cada ítem a evaluar el correcto uso de señales para la comunicación *front-end*/*back-end*. Todo aquel que no esté pintado de amarillo o azul, significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea.

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. Finalmente, todos los descuentos serán asignados manualmente por el cuerpo docente respetando lo expuesto en **el documento de bases generales**.

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**.

11. 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 los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y 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)**.**