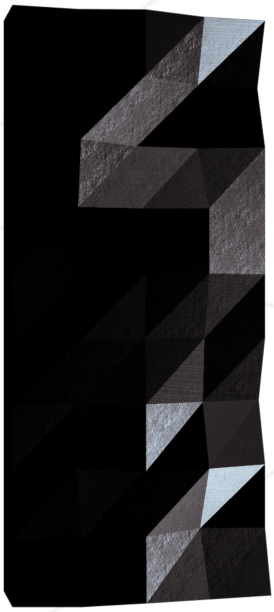


A-Maze-ing

Este es el camino

Resumen: ¡Crea tu propio generador de laberintos y muestra su resultado!

Versión: 1.0

Índice general

Capítulo I

Preámbulo

Los laberintos han fascinado a los humanos durante miles de años. Desde el legendario **Laberinto de Knossos** en la mitología griega — construido por Dédalo para aprisionar al Minotauro — hasta los modernos libros de acertijos y videojuegos, los laberintos siempre han simbolizado el misterio, el desafío y el diseño ingenioso. En informática, la generación de laberintos es más que solo diversión: es una aplicación práctica de algoritmos, aleatoriedad y teoría de grafos. Algunos algoritmos famosos utilizados para la generación de laberintos —como los de *Prim*, *Kruskal* o el *recursive backtracker* — también se utilizan en problemas del mundo real como el diseño de redes o la generación de contenido procedimental. Curiosamente, los laberintos perfectos (con un único camino entre dos puntos) están directamente relacionados con los *árboles de expansión* en la teoría de grafos. Construir un laberinto, especialmente uno que puedas visualizar y compartir, es una excelente manera de explorar cómo las computadoras pueden crear estructura a partir del caos, y divertirse un poco mientras lo hacen.

"Un laberinto no es un lugar para perderse, sino un camino para encontrarse."

— Anónimo

Capítulo II

Instrucciones sobre la IA

Contexto

Durante tu proceso de aprendizaje, la IA puede ayudarte con muchas tareas diferentes. Tómate el tiempo necesario para explorar las diversas capacidades de las herramientas de IA y cómo pueden apoyarte con tu trabajo. Sin embargo, siempre debes abordarlas con precaución y evaluar de forma crítica los resultados. Ya sea código, documentación, ideas o explicaciones técnicas, nunca podrás saber con total certeza si tu pregunta está bien formulada o si el contenido generado es el adecuado. Las personas que te rodean son tu recurso más valioso para ayudarte a evitar errores y puntos ciegos.

Mensaje principal:

- Utiliza la IA para reducir las tareas repetitivas o tediosas.
- Desarrolla habilidades de prompting, ya sea para programacion o para otros temas, que beneficiarán tu futura carrera.
- Aprende cómo funcionan los sistemas de IA para anticipar de forma eficiente y evitar los riesgos comunes, sesgos y problemas éticos.
- Sigue trabajando con tus compañeros para desarrollar tanto habilidades técnicas como habilidades transversales.
- Utiliza únicamente contenido generado por IA que entiendas completamente y del cual puedas responsabilizarte.

Reglas para estudiantes:

- Debes tomarte el tiempo necesario para explorar las herramientas de IA y comprender cómo funcionan, para poder utilizarlas de manera ética y reducir los sesgos potenciales.
- Debes reflexionar sobre tu problema antes de dar instrucciones a la IA. Esto te ayuda a escribir preguntas, instrucciones o conjuntos de datos más claros, detalladas y relevantes utilizando un vocabulario preciso.

- Debes desarrollar el hábito de revisar, cuestionar y probar sistemáticamente cualquier contenido generado por la IA.
- Debes buscar siempre la revisión de otras personas, no te limites a confiar en tu propia validación.

Resultados de esta etapa:

- Desarrollar habilidades de prompting tanto generales como de ámbito especifico.
- Aumentar tu productividad con un uso eficaz de las herramientas de IA.
- Seguir fortaleciendo el pensamiento computacional, la resolución de problemas, la adaptabilidad y la colaboración.

Comentarios y ejemplos:

- Ten en cuenta que la IA puede no tener la respuesta correcta porque esa respuesta no esté ni siquiera en Internet. Además, si te da soluciones incorrectas, intenta no insistir y busca ayuda entre las personas que te rodean. Vas a ahorrarte tiempo y vas a sumar en compresión.
- Vas a enfretarte con frecuencia a situaciones (como exámenes o evaluaciones) donde debes demostrar una comprensión real. Prepárate, sigue construyendo tanto tus habilidades técnicas como transversales.
- Explicar tu razonamiento y debatir con otras personas suele revelar lagunas en tu comprensión de un concepto. Prioriza el aprendizaje entre pares.
- Lo normal es que la herramienta de IA que utilices no conozca tu contexto específico (a menos que se lo indiques), así que te dará respuestas genéricas. Si buscas información más adecuada y más precisa en relación a tu entorno cercano, confía en el resto de estudiantes.
- Donde la IA tiende a generar la respuesta más probable, el resto de estudiantes puede proporcionar perspectivas alternativas y matices valiosos. Confía en la comunidad de 42 como un punto de control de calidad.

✓ Buenas prácticas:

> Le pregunto a la IA: "¿Cómo pruebo una función de ordenación?"Me da algunas ideas. Las pruebo y reviso los resultados con otra persona. Refinamos el enfoque de manera conjunta.

✗ Mala práctica:

> Le pido a la IA que escriba una función completa, la copio y la pego en mi proyecto. Durante la evaluación entre pares, no puedo explicar qué hace ni por qué. Pierdo credibilidad. Suspendo mi proyecto.

✓ Buenas prácticas:

Utilizo la IA para ayudarme a diseñar un parser. Luego, reviso la lógica con otra persona. Encontramos dos errores y lo reescribimos juntos: mejor, más limpio y comprendiendo al 100

✗ Mala práctica:

Dejo que Copilot genere mi código para una parte clave de mi proyecto. Compila, pero no puedo explicar cómo maneja los pipes. Durante la evaluación, no puedo justificarlo y suspendo mi proyecto.

Capítulo III

Instrucciones comunes

III.1. Reglas generales

- El proyecto debe estar escrito en **Python 3.10 o una versión posterior**.
- El proyecto debe adherirse al estándar de programación **flake8**.
- Las funciones deben manejar excepciones de forma adecuada para evitar fallos. Se pueden usar bloques `try-except` para gestionar posibles errores. Hay que priorizar gestores de contexto para recursos como archivos o conexiones para asegurar su depuración automática. Si tu programa falla debido a excepciones no manejadas durante la revisión, se considerará no funcional.
- Todos los recursos (por ejemplo, manejadores de archivos, conexiones de red) deben gestionarse correctamente para evitar fugas. Se deberán usar gestores de contexto siempre que sea posible para su manejo automático.
- El código debe incluir anotaciones de tipo para los parámetros de las funciones, tipos de retorno y variables cuando corresponda (usando el módulo `typing`). Utiliza `mypy` para la comprobación estática de tipos. Todas las funciones deben pasar mypy sin errores.
- Debe incluir docstrings en funciones y clases siguiendo PEP 257 (por ejemplo, estilo Google o NumPy) para documentar propósito, parámetros y valores de retorno.

III.2. Makefile

Se deberá incluir un `Makefile` en el proyecto para automatizar tareas comunes. Debe contener las siguientes reglas (el lint obligatorio implica los flags especificados; se recomienda encarecidamente probar `-strict` para una verificación más rigurosa):

- **install**: Instalar las dependencias del proyecto usando `pip`, `uv`, `pipx`, u otro gestor de paquetes a elección.
- **run**: Ejecutar el script principal del proyecto (por ejemplo, mediante el intérprete de Python).

- **debug**: Ejecutar el script principal en modo depuración usando el depurador integrado de Python (por ejemplo, pdb).
- **clean**: Eliminar archivos temporales o cachés (por ejemplo, ___pycache___, .mypy_cache) para mantener limpio el entorno del proyecto.
- **lint**: Ejecutar los comandos `flake8 .` y `mypy . --warn-return-any --warn-unused-ignores --ignore-missing-imports --disallow-untyped-defs --check-untyped-defs`
- **lint-strict** (opcional): Ejecutar los comandos `flake8 .` y `mypy . --strict`

III.3. Directrices adicionales

- Es recomendable crear programas de prueba para verificar la funcionalidad del proyecto (no se entregan ni se califican). Se pueden utilizar frameworks como `pytest` o `unittest` para pruebas unitarias, cubriendo casos límite.
- Se debe incluir un archivo `.gitignore` para excluir artefactos de Python.
- Se recomienda usar entornos virtuales (por ejemplo, venv o conda) para aislar dependencias durante el desarrollo.

Si se aplican requisitos adicionales específicos del proyecto, se indicarán inmediatamente debajo de esta sección.

Capítulo IV

Instrucciones

IV.1. Resumen

Se deberá implementar un generador de laberintos en Python que reciba un archivo de configuración, genere un laberinto — eventualmente perfecto (con un único camino entre la entrada y la salida)— y lo escribe en un archivo usando una representación hexadecimal de las paredes. También se proporcionará una representación visual del laberinto y se organizará el código para que la lógica de generación pueda reutilizarse más adelante.

IV.2. Uso

El programa debe ejecutarse con el siguiente comando:

```
python3 a_maze_ing.py config.txt
```

- `a_maze_ing.py` es el archivo principal del programa. Es obligatorio usar este nombre.
- `config.txt` es el único argumento. Es un archivo de texto plano que define las opciones de generación del laberinto. Se puede usar un nombre de archivo diferente.

El programa debe manejar todos los errores de manera elegante: configuración inválida, archivo no encontrado, sintaxis incorrecta, parámetros de laberinto imposibles, etc. Nunca debe fallar inesperadamente y siempre debe proporcionar un mensaje de error claro al usuario.

IV.3. Formato del archivo de configuración

El archivo de configuración debe contener un par 'KEY=VALUE' por línea.

Las líneas que comienzan con `#` son comentarios y deben ser ignoradas.
Las siguientes claves son **obligatorias**:

Clave	Descripción	Ejemplo
`WIDTH`	Ancho del laberinto (número de celdas)	`WIDTH=20`
`HEIGHT`	Alto del laberinto	`HEIGHT=15`
`ENTRY`	Coordenadas de entrada (x,y)	`ENTRY=0,0`
`EXIT`	Coordenadas de salida (x,y)	`EXIT=19,14`
`OUTPUT_FILE`	Nombre del archivo de salida	`OUTPUT_FILE=maze.txt`
`PERFECT`	¿Es el laberinto perfecto?	`PERFECT=True`

Se pueden añadir claves adicionales (por ejemplo: semilla, algoritmo, modo de visualización) si son útiles.

Debe estar disponible un archivo de configuración por defecto en el repositorio Git.

IV.4. Requisitos del Laberinto

- El laberinto debe ser generado aleatoriamente, pero es requisito indispensable poder reproducirlo mediante una semilla.
- Cada celda del laberinto debe tener entre 0 y 4 paredes, en cada punto cardinal (Norte, Este, Sur, Oeste).
- El laberinto debe ser válido, lo que significa:
  - La entrada y la salida existen y son diferentes, dentro de los límites del laberinto.
  - La estructura asegura conectividad total y ninguna celda aislada (excepto el patrón '42', ver abajo).
  - Como la entrada y la salida son celdas específicas, debe haber paredes en los bordes externos.
  - Los datos generados deben ser coherentes: cada celda vecina debe tener la misma pared si la hay. Por ejemplo, está prohibido tener una primera celda con una pared en el lado este, y la segunda celda detrás de esa pared sin una pared en el lado oeste.
- El laberinto no puede tener grandes espacios abiertos. Los pasillos no pueden ser más anchos de 2 celdas, una zona abierta de 3x3 está prohibida.
- Cuando se represente visualmente (ver abajo), el laberinto debe contener un "42"visible dibujado por varias celdas completamente cerradas.
- Si la bandera `PERFECT` está activada, el laberinto debe contener **exactamente un camino** entre la entrada y la salida (es decir, debe ser un laberinto perfecto).

El patrón "42"puede omitirse en el caso de que el tamaño del laberinto no lo permita (es decir, demasiado pequeño). Debe imprimirse un mensaje de error en la consola en ese caso.

IV.5. Formato del Archivo de Salida

El laberinto debe escribirse en el archivo de salida usando un dígito hexadecimal por celda, donde cada dígito codifica qué paredes están cerradas:

Bit	Dirección
0 (LSB)	Norte
1	Este
2	Sur
3	Oeste

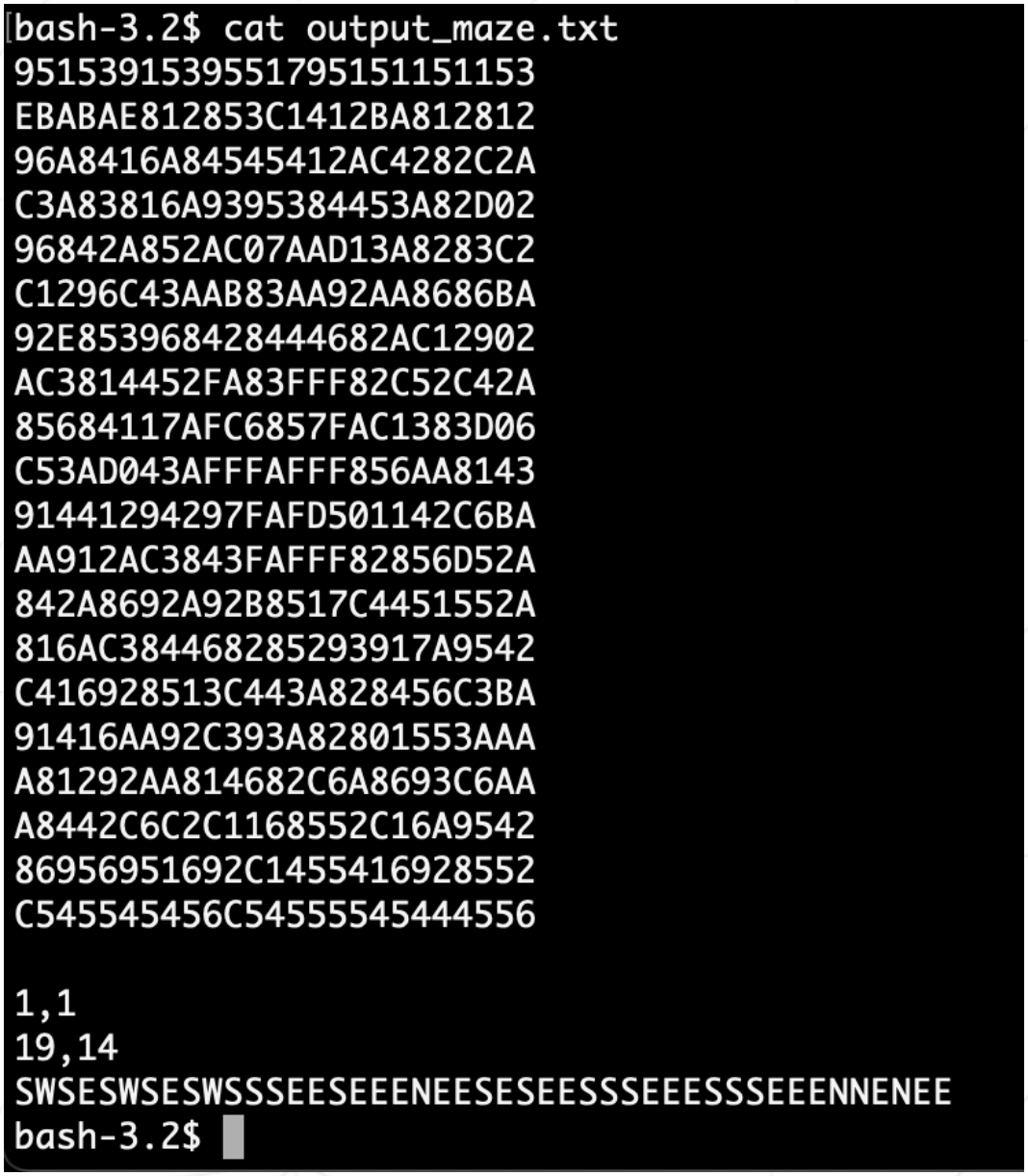
- Una pared cerrada establece el bit a `1`, abierta significa `0`.
  Ejemplo: Un `3` (binario `0011`) significa que las paredes están abiertas al **sur** y **oeste**.
  En cambio, `A` (binario `1010`) significa que las paredes **este** y **oeste** están cerradas.

- Las celdas se almacenan fila por fila, una fila por línea.

- Después de una línea vacía, los siguientes 3 elementos siguientes se insertan en el archivo de salida en 3 líneas:
  - las coordenadas de entrada, las coordenadas de salida, y el camino válido más corto desde la entrada hasta la salida, usando las cuatro letras `N` , `E` , `S` , `W` .

- Todas las líneas terminan con un `\n` .

En conjunto con su archivo de configuración específico, este archivo de salida podría ser probado automáticamente por una Moulinette. Además, se proporcionará un script de validación con este documento (subject) para controlar que el archivo de salida contenga datos coherentes.



Ejemplo de archivo de salida

Capítulo V

Representación visual

El programa debe proporcionar una forma de mostrar el laberinto visualmente, usando:

- Renderizado ASCII en terminal, o
- Una visualización gráfica usando la librería MiniLibX (MLX).

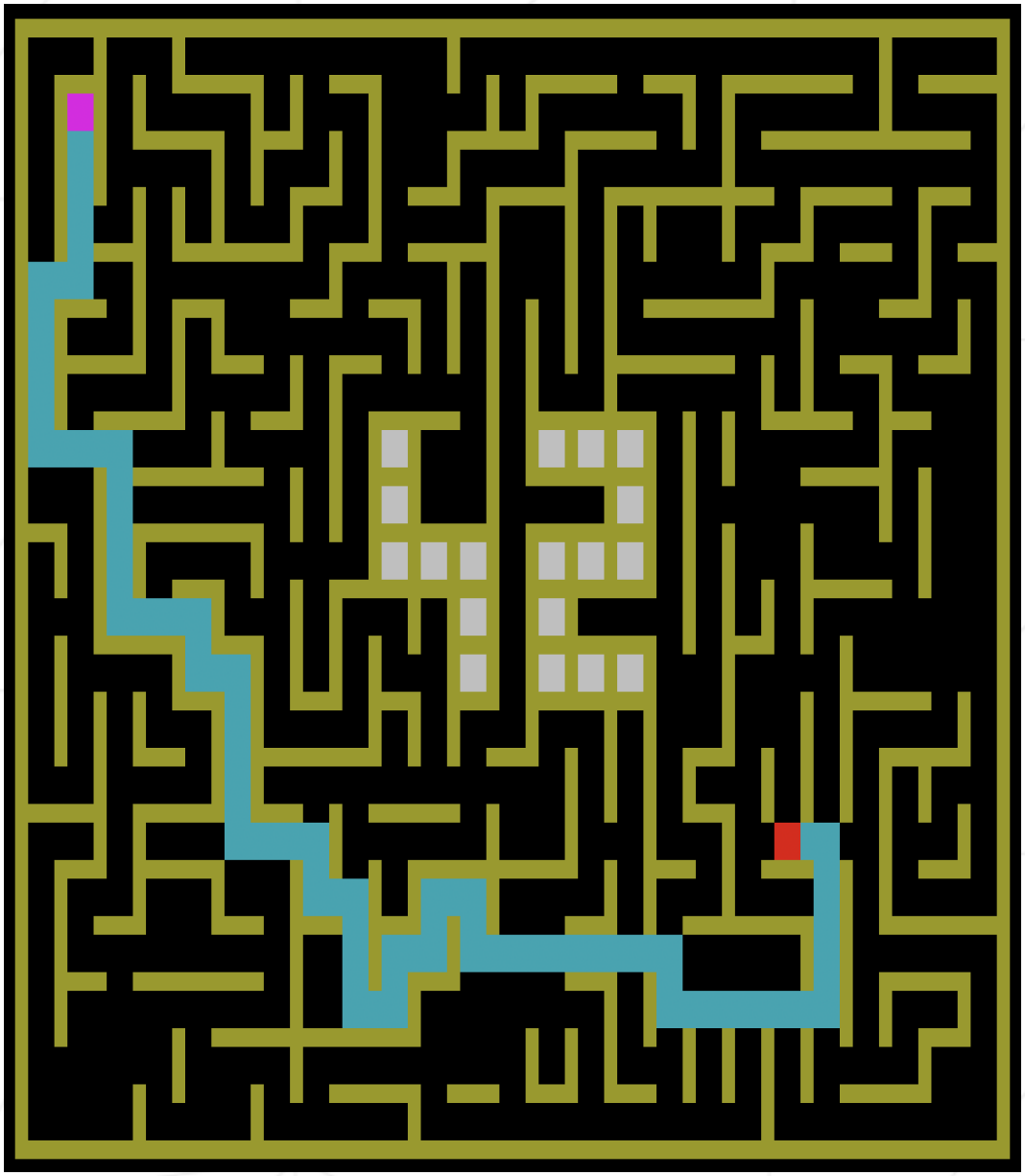
La visualización debe mostrar claramente las paredes, la entrada, la salida y el camino de la solución.

El programa debe tener, también disponibles interacciones para el usuario, al menos para las siguientes tareas:

- Regenerar un nuevo laberinto y mostrarlo.
- Mostrar/Ocultar un camino válido más corto desde la entrada hasta la salida.
- Cambiar los colores de las paredes del laberinto.
- Opcional: establecer colores específicos para mostrar el patrón "42".

Se pueden añadir interacciones de usuario adicionales.

Renderizado por defecto del laberinto en terminal



Laberinto diferente, camino más corto y colores de pared

Capítulo VI

Requisitos de reusabilidad del código

Se deberá implementar la generación del laberinto como una clase única (ej., 'Maze-Generator') dentro de un módulo independiente que pueda ser importado en un proyecto futuro.

También se tiene que proporcionar una breve documentación que describa cómo:

- Instanciar y usar el generador, con al menos un ejemplo básico.
- Pasar parámetros personalizados (ej., tamaño, semilla).
- Acceder a la estructura generada y acceder, al menos, a una solución.

`El módulo generador de laberintos permite el acceso a la estructura del laberinto, pero no es necesariamente el mismo formato que el archivo de salida.`

Todo este módulo reutilizable (código y documentación) debe estar disponible en un único archivo adecuado para una instalación posterior mediante `pip`. Este paquete debe llamarse `mazegen-*` y el archivo debe estar ubicado en la raíz del repositorio git. Se permiten las extensiones `.tar.gz` y `.whl`, tal como las genera la construcción estándar de un paquete Python. Ejemplo de un nombre de archivo completo: `mazegen-1.0.0-py3-none-any.whl` .

Se deberá proporcionar en el repositorio Git todos los elementos necesarios para construir el paquete. Durante la evaluación se solicitará que, en un entorno virtual o equivalente, se instalen las herramientas necesarias y se construya el paquete de nuevo desde las fuentes.

El archivo principal `README.md` (no parte del módulo reutilizable) también debe contener esta breve documentación.

Capítulo VII

Requisitos del Readme

Debe incluirse un archivo `README.md` en la raíz del repositorio Git. Su propósito es permitir que cualquier persona que no esté familiarizada con el proyecto (pares, personal, responsables de selección, etc.) pueda entender rápidamente de qué trata el proyecto, cómo ejecutarlo y dónde encontrar más información sobre el tema.
El `README.md` debe incluir, como mínimo:

- La primera línea debe estar en cursiva y decir: *Este proyecto ha sido creado como parte del currículo de 42 por <login1>[, <login2>[, <login3>[...]]].*
- Una sección de "**Descripción**"que presente claramente el proyecto, incluyendo su objetivo y una breve visión general.
- Una sección de "**Instrucciones**"que contenga cualquier información relevante sobre compilación, instalación y/o ejecución.
- Una sección de "**Recursos**"que enumere referencias clásicas relacionadas con el tema (documentación, artículos, tutoriales, etc.), así como una descripción del uso de IA, especificando para qué tareas y en qué partes del proyecto se ha utilizado.

➠ **Podrían requerirse secciones adicionales dependiendo del proyecto** (por ejemplo, ejemplos de uso, lista de características, decisiones técnicas, etc.).

Cualquier contenido extra requerida se listará explícitamente a continuación.

- La estructura completa y el formato de su archivo de configuración.
- El algoritmo de generación de laberintos que eligieron.
- Por qué eligieron este algoritmo.
- Qué parte de su código es reutilizable y cómo.
- Su equipo y gestión del proyecto con:
    - Los roles de cada miembro del equipo.

- Su planificación prevista y cómo evolucionó hasta el final.
- Qué funcionó bien y qué podría mejorarse.
- ¿Han utilizado alguna herramienta específica? ¿Cuáles?

Si se han implementado características avanzadas (múltiples algoritmos, opciones de visualización), deben describirse también en el archivo `README.md`.

Capítulo VIII

Bonus

Es posible añadir varios bonus al proyecto. Aquí hay posibles ejemplos:

- Soporte para múltiples algoritmos de generación de laberintos.
- Añadir animación durante la generación del laberinto.

Capítulo IX

Entrega y evaluación

Entrega el trabajo en un repositorio `Git`, como de costumbre. Solo el trabajo dentro del repositorio será evaluado durante la defensa. Se recomienda verificar que los nombres de los archivos sean los correctos.

Durante la evaluación, es posible que se solicite una ligera **modificación del proyecto**. Esto puede consistir en ajustar ligeramente el comportamiento, modificar unas cuantas líneas de código o incorporar una característica fácil de implementar.

Puede que este paso **no sea necesario en todos los proyectos**, pero hay que tenerlo en cuenta si así se especifica en la hoja de evaluación.

Este paso sirve para verificar la comprensión real de una parte específica del proyecto. La modificación se puede realizar en cualquier entorno de desarrollo que se elija (por ejemplo, la configuración habitual), y debería ser factible en unos pocos minutos, a menos que se defina un plazo específico como parte de la evaluación.
Por ejemplo, se puede pedir hacer una pequeña actualización en una función o *script*, modificar lo que se vería en pantalla o ajustar una estructura de datos para almacenar nueva información, etc.

Los detalles (alcance, objetivo, etc.) se especificarán cada **hoja de evaluación** y pueden variar de una evaluación a otra para el mismo proyecto.