Sistema de Análisis Automático de Complejidad Algorítmica

Juan David Ocampo González
Juan Manoel Miranda Gómez

Diciembre 5 de 2025

Docente: Luz Enith Guerrero

Índice

1. Introducción

El *Analizador de Algoritmos* es un software desarrollado por un equipo de estudiantes de ingeniería de la Universidad de Caldas, durante el curso de Análisis y Diseño de Algoritmos, dictado por la docente Luz Enith Guerrero.

Este software tiene como función analizar la eficiencia temporal y espacial de algoritmos escritos en pseudocódigo, utilizando métodos vistos en clase con el apoyo de modelos de lenguaje grandes (LLMs) para lograr un análisis más profundo y preciso.

El proyecto nos motivó a reforzar conceptos vistos en clase y a explorar tecnologías emergentes como LangGraph y Gemini para resolver este desafío de forma innovadora.

2. Análisis del Problema

Realizar el análisis de complejidad de un algoritmo requiere personas con experiencia y conocimientos avanzados en cálculo, lo que puede tomar horas o incluso días para obtener un aproximado de la eficiencia del algoritmo. Abstraer esta información para que pueda ejecutarse en código era prácticamente imposible antes, ya que no siempre es fácil extraer las ecuaciones de recurrencia o las sumatorias exactas de los algoritmos.

Nuestro *Analizador de Algoritmos* puede recibir tanto lenguaje natural como pseudocódigo, el cual puede contener arreglos, ciclos, llamados a funciones, estructuras condicionales y más. Lo que le permite hacer un analisis que tardaria demaciado tiempo, en cuestion de minutos.

2.1. Alcance del Sistema

Nuestro software tiene un alcance extenso, siendo capaz de procesar:

- **Algoritmos recursivos**: Detección automática de recursión, construcción de relaciones de recurrencia $T(n) = aT(n/b) + f(n)$, y aplicación del teorema maestro.
- **Algoritmos iterativos**: Análisis de ciclos `for`, `while`, `repeat-until`, con soporte para anidamiento múltiple.
- **Estructuras de control**: Condicionales `if-else` con análisis de mejor, peor y caso promedio.
- **Análisis línea por línea**: Cálculo de costo computacional para cada línea de código.

Su única limitación depende del LLM y su respectiva API key. Actualmente utilizamos la API gratuita de Google con los modelos `gemini-2.5-flash` y `gemini-2.5-flash-lite`, que con prompts bien diseñados e inputs organizados, son capaces de resolver los algoritmos vistos en clase con una precisión superior al 90 %.

3. Entrada de Datos y Sintaxis

La sintaxis del pseudocódigo fue definida por la docente y se estructura de la siguiente manera:

3.1. Estructuras de Control

- **FOR**: `for variableContadora ← valorInicial to limite do begin ... end`
- **WHILE**: `while (condicion) do begin ... end`
- **REPEAT**: `repeat ... until (condicion)`
- **IF**: `if (condicion) then begin ... end else begin ... end`

3.2. Reglas Generales

- **Asignación**: Símbolo ← (no se permiten asignaciones múltiples)
- **Comentarios**: Símbolo ▷ para el resto de la línea
- **Variables**: Locales al procedimiento (no globales)
- **Acceso a arreglos**: `A[i]` o `A[1..j]` para rangos
- **Tamaño de arreglo**: `length(A)`
- **Declaración de vectores locales**: Al inicio después de `begin`

3.3. Clases y Objetos

- **Definición de clase**: `NombreClase {atributo1 atributo2 ...}`
- **Declaración de objeto**: `Clase nombre_del_objeto`
- **Acceso a campos**: `objeto.campo`
- **Punteros**: Pueden tener valor `NULL`

3.4. Subrutinas

- **Definición**: `nombre_subrutina(parámetro1, ..., parámetroK) begin ... end`
- **Parámetros arreglo**: `nombre_arreglo[n..m]`
- **Parámetros objeto**: `Clase nombre_objeto`
- **Llamado**: `CALL nombre_subrutina(parámetros)`

3.5. Operadores

- **Booleanos**: `and`, `or`, `not` (con evaluación perezosa)
- **Valores booleanos**: `T` (true), `F` (false)
- **Relacionales**: $<$, $>$, $\leq$, $\geq$, $=$, $\neq$
- **Matemáticos**: $+$, $*$, $/$, $-$, `mod`, `div`, $\lceil \cdot \rceil$ (techo), $\lfloor \cdot \rfloor$ (piso)

3.6. Modalidades de Interacción

El usuario puede interactuar con el sistema de dos formas:

1. **Consola Python**: Ejecución directa mediante scripts Python.
2. **Interfaz Gráfica**: Aplicación web desarrollada en React para una experiencia más amigable.
3. **API REST**: Endpoints FastAPI para integración con otros sistemas.

Las expresiones en lenguaje natural son evaluadas por Gemini 2.5 Flash, actuando como un copiloto inteligente. Mediante ingeniería de prompts, se le proporciona la sintaxis y reglas del pseudocódigo, lo que le permite interpretar y asistir al usuario con su algoritmo de manera efectiva.

3.7. Metodología de Desarrollo

La tecnica que aplicamos fue "divide y venceras". Dividimos el problema de Analizar un algoritmo, en sub problemas.

- Recibir el lenguaje natural o código
- Hacer correcciones de sintaxis
- Generar un grafo que representa el código
- Generar una expresión matemática que resume la eficiencia del código
- Aplicar técnicas iterativas (resolver sumatorias y analizar el código) para la complejidad temporal
- Aplicar técnicas recursivas (árboles de recurrencia, teorema maestro, ecuación homogénea y cálculos de sumatorias) para su complejidad temporal
- Para el cálculo espacial miramos las variables extras que no son la entrada de datos y hacemos la sumatoria del total de ellas
- Organizar los resultados y ponerlos en funciones asintóticas apropiadas
- Análisis final y completo por un LLM

Lo mas dificil fue realizar el caso promedio. Para eso necesitamos un analisis mas profundo del algoritmo para que se pueda sacar las ecuaciones necesarias para poder resolver. Para eso utilizamos los llms, que seran de apoyo para analizar el codigo como un todo.

4. Arquitectura e Implementación del Sistema

4.1. Patrón Arquitectónico Adoptado

Nuestro analizador está organizado como una arquitectura **cliente–servidor por capas**:

- **Capa de presentación**: una pequeña interfaz web en React y, de forma alternativa, la consola de Python.
- **Capa de servicios / API**: un backend en FastAPI que expone un endpoint para enviar el pseudocódigo o la descripción en lenguaje natural.
- **Capa de lógica de negocio**: el motor de análisis implementado con Python y LangGraph, donde se hace todo el trabajo "duro" (parseo, generación de AST, análisis iterativo/recursivo, construcción de ecuaciones, etc.).
- **Capa de integración con LLMs y librerías**: aquí se hacen las llamadas a Gemini y a las librerías matemáticas (por ejemplo SymPy), siempre a través de funciones bien separadas.

Este esquema nos ayuda a separar la parte visual de la lógica de análisis y de la integración con los modelos de lenguaje.

4.2. Justificación del Diseño

Escogimos esta arquitectura por varias razones prácticas:

- **Separación de responsabilidades**: el frontend solo se encarga de mostrar y recoger información del usuario; toda la lógica complicada queda en el backend.
- **Escalabilidad**: si en el futuro queremos soportar más usuarios o más peticiones, podemos escalar el servidor sin tocar la interfaz.
- **Extensibilidad**: podemos cambiar el modelo de lenguaje, agregar nuevas técnicas de análisis o soportar nuevos tipos de algoritmos sin tener que reescribir todo.
- **Interoperabilidad**: al exponer un API REST, otros sistemas podrían usar nuestro analizador sin necesidad de conocer los detalles internos.

En resumen, intentamos que el diseño fuera sencillo, pero que nos permitiera crecer si el proyecto se sigue usando más adelante.

4.3. Diagrama de Arquitectura

El diagrama de alto nivel incluye los siguientes bloques principales:

1. **Usuario**
2. **Interfaz (Web / Consola)**
3. **Servidor FastAPI**
4. **Motor de análisis (LangGraph + Python)**
5. **Parser de pseudocódigo**
6. **Módulo de análisis iterativo**
7. **Módulo de análisis recursivo**

8. **Módulo de integración con LLM (Gemini)**
9. **Módulo matemático (SymPy y utilidades)**

Las flechas muestran el flujo básico: el usuario envía el algoritmo → la interfaz lo manda al backend → el motor analiza y pregunta al LLM cuando es necesario → se generan resultados y se devuelven al usuario en formato legible.

4.4. Componentes del Sistema

De manera sencilla, los componentes principales son:

- **Módulo de entrada**: recibe el texto (pseudocódigo o lenguaje natural) desde la interfaz o por consola.
- **Analizador léxico y sintáctico**: comprueba que el pseudocódigo cumple la sintaxis definida en clase y construye un árbol de sintaxis (AST) simplificado.
- **Evaluador semántico**: identifica si el algoritmo es iterativo, recursivo o mezcla de ambos y detecta estructuras de control importantes.
- **Módulo de deducción de complejidad iterativa**: convierte ciclos en sumatorias y las resuelve para obtener la complejidad temporal y espacial.
- **Módulo de deducción de complejidad recursiva**: extrae ecuaciones de recurrencia y aplica las técnicas vistas (teorema maestro, árboles de recurrencia, ecuaciones características, etc.).
- **Motor de interacción con el LLM**: encapsula las llamadas a Gemini y devuelve respuestas ya filtradas y con formato.
- **Interfaz de usuario**: muestra el pseudocódigo, las ecuaciones intermedias y la respuesta final (O, Ω, Θ) en un formato amigable.

4.5. Flujo de Datos y Lógica Interna

El recorrido típico de una entrada es el siguiente:

1. El usuario escribe un algoritmo en pseudocódigo o una descripción en lenguaje natural.
2. El backend detecta si el texto parece ya pseudocódigo o si primero debe pasarse por el LLM para generar el pseudocódigo equivalente.
3. El parser construye un AST y valida la sintaxis según las reglas del curso.
4. A partir del AST se decide si el algoritmo es principalmente **iterativo** o **recursivo**.
5. Si es iterativo, se generan sumatorias y se resuelven con ayuda de SymPy y reglas básicas.
6. Si es recursivo, se construye la ecuación de recurrencia y se elige el método más adecuado para resolverla (teorema maestro, iteración, árbol de recurrencia, etc.).

7. Finalmente se organiza la información: costos por línea, ecuaciones intermedias y complejidades en O, Ω y Θ, y se le pide al LLM que escriba una explicación entendible.

8. El backend devuelve un JSON con todos los resultados y la interfaz los muestra al usuario.

4.6. Manejo de Errores y Validación de Entrada

Para que el sistema no "se rompa" con entradas malas, tomamos algunas medidas:

- **Validación de sintaxis**: el parser detecta cuando el pseudocódigo no cumple las reglas (por ejemplo, falta un `end` o hay un `for` mal escrito).
- **Mensajes claros**: cuando hay errores de sintaxis, el sistema intenta indicar en qué parte del texto está el problema para que el usuario pueda corregirlo.
- **Correcciones asistidas por LLM**: en algunos casos se le pide al modelo de lenguaje que proponga una versión corregida del código manteniendo la intención del usuario.
- **Manejo de excepciones**: en el backend usamos bloques `try/except` para capturar fallos inesperados (por ejemplo, una división por cero en una ecuación) y devolver un mensaje de error controlado en lugar de que el programa se cierre.

De esta forma, aunque la entrada no sea perfecta, el usuario recibe una respuesta que le explica qué pasó.

4.7. Estructura del Código y Organización de Archivos

De forma muy resumida, la organización del proyecto es la siguiente:

- **Carpeta principal del backend (Python):**
  - `app/api.py`: define el endpoint principal donde se recibe el texto a analizar.
  - `app/agents/`: contiene el grafo de LangGraph, el estado compartido y los nodos del flujo (decisión inicial, validación, análisis iterativo, análisis recursivo, resultado, etc.).
  - `app/agents/tools/`: funciones auxiliares para resolver sumatorias y recurrencias.
  - `app/tools/ast_parser/`: gramática y parser para convertir el pseudocódigo en AST.
- **Archivos de configuración:**
  - `.env`: donde se guarda la API key de Google (no se sube al repositorio).
  - `requirements.txt` o `pyproject.toml`: dependencias del proyecto.
- **Frontend (React):**
  - Componentes para el formulario de entrada y la visualización de resultados.

Aunque hay más archivos, esta es la estructura básica que usamos para mantener el código ordenado.

4.8. Integración de LLMs

Los LLMs son utilizados para las siguientes tareas:

- **Principio de correctitud**: Análisis del código y lenguaje natural
- **Corrección y validación de sintaxis**: Verificación automática del pseudocódigo
- **Parseo de lenguaje natural a pseudocódigo**: Conversión inteligente
- **Auxiliares en la generación de ecuaciones**: Principalmente para el caso promedio
- **Resultado final**: Análisis del algoritmo en lenguaje natural

Utilizamos `gemini-2.5-flash` y `gemini-2.5-flash-lite` (gratuito). Lo implementamos con **LangGraph**, un framework para crear agentes que utilizan tanto código escrito manualmente como LLMs para mejorar los resultados.

Con una ingeniería de prompts adecuada y utilizando *tools* (herramientas auxiliares que tiene el propio LangGraph), la fiabilidad del LLM puede llegar a ser superior al 90 %.

5. Análisis de Eficiencia del Sistema

5.1. Bibliotecas y Herramientas Utilizadas

Las principales bibliotecas utilizadas son:

- Gemini 2.5 Flash
- Gemini 2.5 Flash Lite
- SymPy – Para cálculos simbólicos

Cada una tiene su propio costo computacional asociado.

5.2. Complejidad Temporal del Sistema

Podemos decir que el costo promedio para nuestra solución es $\Theta(N)$, donde N es la cantidad de líneas de código que se generaron o que se pasaron como entrada.

La ecuación de complejidad vendría a ser:

$$T(n) = C_1 \cdot N + C_2 \cdot \text{CostoLLM} + C_3 \cdot \text{CostoSympy}$$

Esta complejidad aplica para los 3 casos (mejor, peor y promedio).

5.3. Comparación de Rendimiento

- **Modelo más rápido**: Gemini 2.5 Flash Lite, que responde 3 a 4 veces más rápido que Gemini 2.5 Flash.
- **Comparación manual vs. sistema**:
  - En la comparación de análisis manual (mandar un prompt único) vs. utilizar nuestro algoritmo, se notan dos aspectos importantes:
  - El tiempo de análisis manual con prompt directo reduce mucho, pero la veracidad de la información también se reduce significativamente.
  - Nuestro algoritmo tiene complejidad $O(N)$, pero todos los procesos realizados son para abstraer la información del algoritmo y dejarlo lo más matemático posible, permitiendo que el LLM haga pequeños ajustes en la ecuación generada para los 3 casos.
- **Reducción de alucinaciones**: Como trabajamos con agentes, hay LLMs especializados para resolver problemas específicos. En comparación con un LLM normal de aplicativo, que cuanto más se pregunta, más empieza a olvidar o asumir cosas que no son ciertas (lo que reduce su precisión), nuestro sistema mantiene mayor coherencia y precisión.

6. Pruebas

Además de las figuras incluidas, probamos el sistema con varios algoritmos clásicos, tanto iterativos como recursivos. Algunos ejemplos son:

- Búsqueda lineal y búsqueda binaria.
- Ordenamiento burbuja, inserción y mergesort.
- Cálculo de factorial, Fibonacci, Torres de Hanoi.
- Problemas vistos en clase como cambio de monedas o multiplicación de matrices.

Para cada algoritmo comparamos la complejidad que entrega el analizador con la complejidad teórica esperada. En la mayoría de los casos los resultados coinciden. Los principales problemas aparecieron cuando:

- El pseudocódigo no seguía exactamente la sintaxis definida por la profesora.
- Faltaba información para el caso promedio (por ejemplo, probabilidades en un `if`).

En esos casos el sistema informa la ambigüedad y aclara que la respuesta puede ser aproximada.

Figura 1: Ejemplo de prueba 1

7. Conclusiones y Recomendaciones

7.1. Conclusiones

Implementar frameworks mas rápidos que LangGraph para optimizar tiempos de respuesta Expandir el soporte para más estructuras de datos complejas Mejorar la interfaz de usuario con visualizaciones interactivas Agregar soporte para análisis de algoritmos paralelos y distribuidos En este proyecto se aprende incluso cómo funciona el propio cerebro para hacer el análisis, ya que nos tocó abstraer absolutamente toda nuestra línea de razonamiento para poder pasarla a código. Desde la generación de un AST simplificado, ecuaciones intermediarias, y análisis completo para extraer el caso promedio, fue un ejercicio bien interesante y gratificador que nos hizo reforzar y buscar nuevos conocimientos.

El desarrollo de este sistema permitió:

- Comprender profundamente los conceptos de análisis de algoritmos
- Aplicar técnicas de ingeniería de software modernas
- Integrar inteligencia artificial de manera efectiva en la resolución de problemas complejos
- Crear una herramienta útil para estudiantes y profesionales

7.2. Recomendaciones para Mejoras Futuras

Para mejoras futuras, consideramos que podríamos:

- Utilizar otros modelos más especializados en código (como Codex o CodeLlama)
- Implementar frameworks más rápidos que LangGraph para optimizar tiempos de respuesta
- Expandir el soporte para más estructuras de datos complejas
- Mejorar la interfaz de usuario con visualizaciones interactivas
- Agregar soporte para análisis de algoritmos paralelos y distribuidos

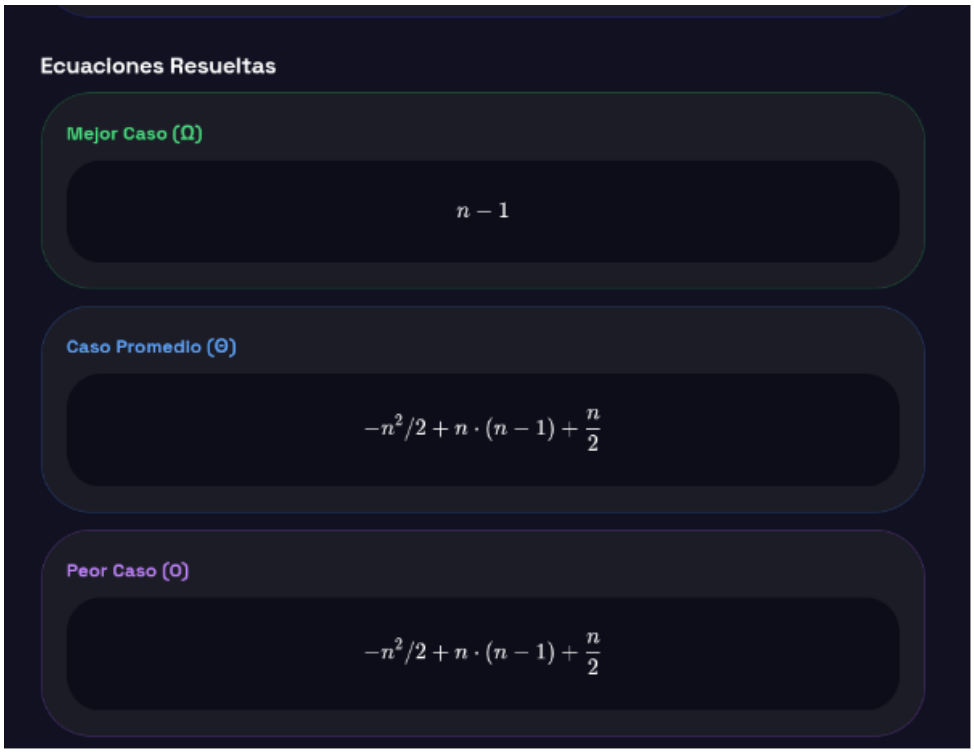

Figura 2: Ejemplo de prueba 2

8. Manual Técnico

8.1. Objetivo del Manual

Este manual está dirigido a personas que quieran **instalar, configurar o modificar** el analizador. Explica qué se necesita, cómo ejecutar el proyecto y cómo está organizado el código.

8.2. Requisitos del Sistema

- Python 3.10 o superior.
- Conexión a internet para usar la API de Gemini.
- Navegador web moderno (para la interfaz).
- Opcional: Node.js y npm si se quiere levantar el frontend en React.

8.3. Instalación y Configuración Básica

1. Clonar el repositorio del proyecto.
2. Crear y activar un entorno virtual de Python `uv venv`.
3. Instalar las dependencias con `uv sync` (o el archivo equivalente).
4. Crear un archivo `.env` con la API key de Google (`GOOGLE_API_KEY=...`) y el modelo (`GEMINI_MODEL=gemini-2.5-flash`).

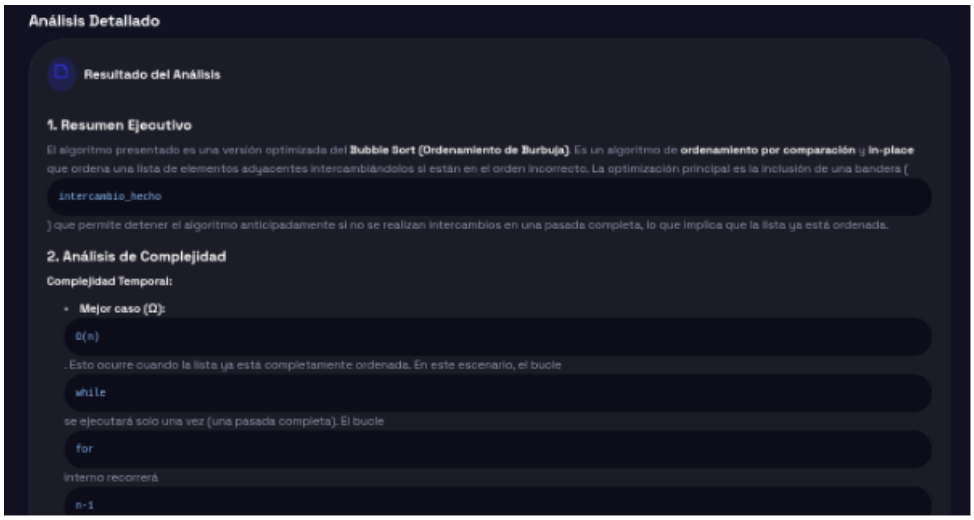

Figura 3: Ejemplo de prueba 3

5. Ejecutar el backend con `uv run main.py`.

Si se usa la interfaz en React, se entra a la carpeta del frontend, se ejecuta `npm install` y luego `npm run dev` o similar.

8.4. Ejecución del Sistema

- **Solo backend**: se puede probar el endpoint desde `/docs` (documentación automática de FastAPI) enviando el algoritmo en el campo de texto.
- **Backend + frontend**: el usuario entra a la URL del frontend, escribe el algoritmo y pulsa el botón de analizar; el frontend llama al backend por HTTP.

8.5. Estructura Básica del Código

En esta parte del manual técnico basta con referenciar la sección 4.7 del informe, donde se describe la organización de carpetas y archivos. Si el proyecto crece, aquí se pueden agregar más detalles (por ejemplo, sobre pruebas automatizadas o despliegue en la nube).

9. Manual de Usuario

9.1. Inicio Rápido

1. Abrir la aplicación web o, en su defecto, ejecutar el script de consola.
2. Elegir si se va a escribir **pseudocódigo** o una **descripción en lenguaje natural**.
3. Pegar o escribir el algoritmo en el cuadro de texto.
4. Pulsar el botón “Analizar”.

9.2. Análisis de Algoritmos en Pseudocódigo

- Escribir el algoritmo siguiendo la sintaxis que se usó en clase (palabras reservadas, `begin/end`, `for`, `while`, etc.).
- Verificar que todas las estructuras estén bien cerradas.
- El sistema devolverá:
  - La complejidad temporal en O, Ω y Θ.
  - Una idea de la complejidad espacial.
  - Una explicación en lenguaje sencillo de cómo se obtuvo el resultado.

9.3. Análisis desde Lenguaje Natural

- En este modo el usuario puede escribir algo como: "ordenar un arreglo con mergesort" o "calcular el factorial de n de forma recursiva".
- El sistema primero genera un pseudocódigo aproximado usando el LLM y luego lo analiza como en el caso anterior.
- Es posible que la explicación indique que el algoritmo fue "interpretado" a partir de la descripción.

9.4. Lectura de Resultados

En la interfaz se muestran, de forma resumida:

- Las ecuaciones clave (sumatorias o recurrencias).
- La notación asintótica final.
- Un breve comentario con el tipo de algoritmo (por ejemplo, "divide y vencerás", "resta y vencerás", etc.).

El objetivo es que el usuario no solo vea la respuesta, sino que también entienda de dónde salió.

9.5. Mensajes de Error Frecuentes

Algunos mensajes que puede mostrar el sistema:

- **"Error de sintaxis"**: el pseudocódigo no cumple la gramática esperada (por ejemplo, falta un `end`).
- **"No se pudo extraer la recurrencia"**: el algoritmo recursivo es demasiado complejo o no sigue los patrones soportados.
- **"Entrada ambigua"**: falta información para determinar el caso promedio o ciertos detalles de la lógica.

En todos estos casos el sistema intenta sugerir al usuario qué puede corregir.

10. Anexos

Para cerrar el informe, incluimos como anexos:

- **Código fuente documentado**: enlace al repositorio oficial del proyecto.
- **Lista de algoritmos de prueba**: con su complejidad esperada.
- **Capturas de pantalla de la interfaz**: mostrando ejemplos completos de uso.

Estos anexos no son obligatorios para entender el informe, pero ayudan a mostrar el alcance real del sistema y facilitan que otras personas puedan reutilizarlo o mejorarlo.

11. Codigo

- Codigo Backend
- Codigo Frontend