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. 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:

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

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

Se realizaron pruebas exhaustivas con diversos algoritmos para validar la precisión del sistema:

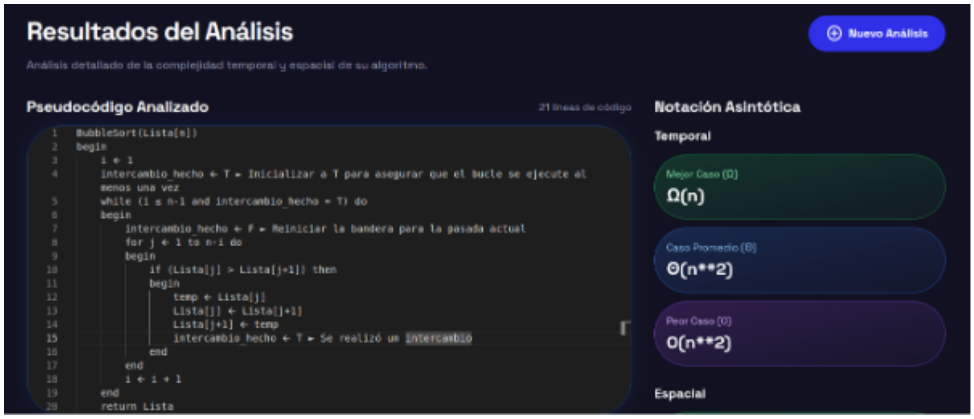

Figura 1: Ejemplo de prueba 1

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

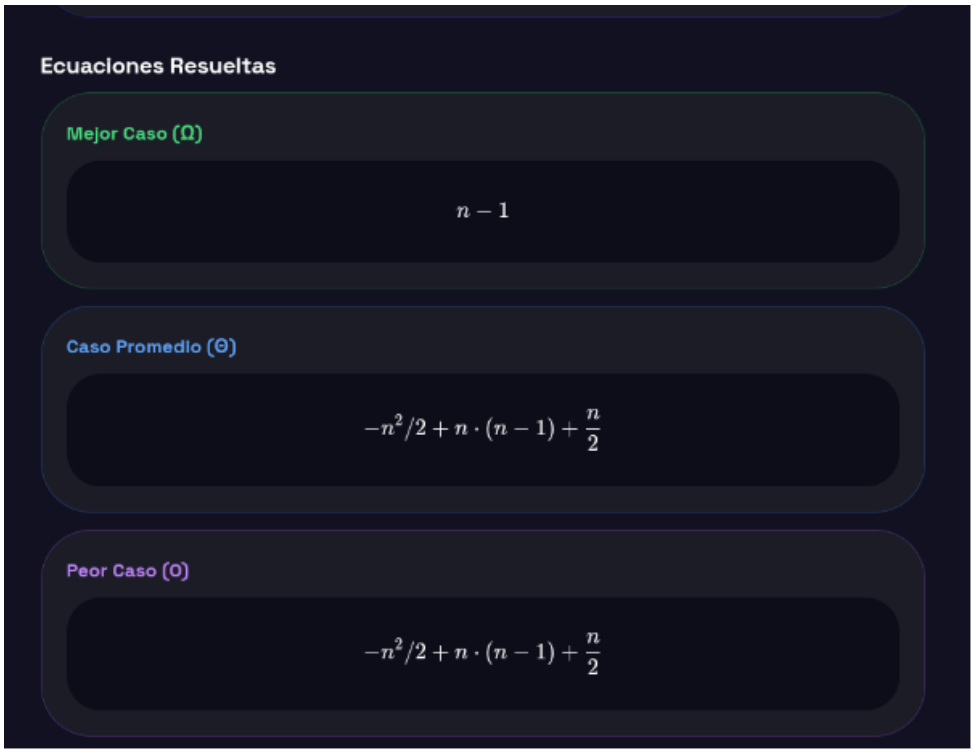

Figura 2: Ejemplo de prueba 2

- 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

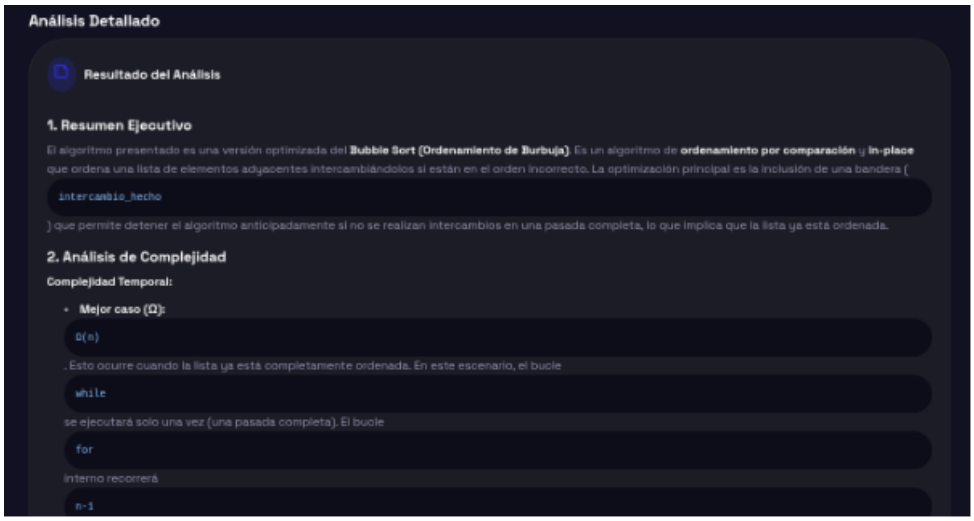

Figura 3: Ejemplo de prueba 3