Trabajo Integrador
Programación I
Análisis de Algoritmos en Python

Alumnos: Sarrú Franco, Ronderos Gonzalo
Materia: Programación I
Profesor: Quirós Nicolás
Tutor: Quirós Nicolás
Fecha de Entrega: 9 de Junio de 2025

Índice:

Contenido

1. Introducción

La elección de este tema responde a su relevancia central en el ámbito de la programación y la informática. Analizar algoritmos permite entender su rendimiento en términos de eficiencia y consumo de recursos, facilitando decisiones informadas durante el desarrollo de software.

Dado que un mismo problema puede resolverse de múltiples maneras, surge la necesidad de comparar diferentes algoritmos que producen resultados idénticos. Este proceso de evaluación es esencial para seleccionar la solución más adecuada según las demandas específicas de cada proyecto.

El estudio de la eficiencia algorítmica, tanto temporal (tiempo de ejecución) como espacial (uso de memoria), resulta fundamental. Conocer la complejidad de un algoritmo permite prever su comportamiento ante conjuntos de datos más grandes, optimizando el rendimiento y previniendo limitaciones técnicas.

Este trabajo tiene como propósito alcanzar los siguientes objetivos:

- Profundizar en los principios del análisis de algoritmos.
- Fomentar un enfoque crítico para seleccionar algoritmos y estructuras de datos óptimas.
- Evaluar y comparar el desempeño temporal y espacial de diversas soluciones.
- Implementar algoritmos clásicos en Python, utilizando herramientas y estructuras para analizar su comportamiento.

2. Marco Teórico:

Algoritmo:
Un algoritmo es una secuencia finita de pasos bien definidos que resuelven un problema o ejecutan una tarea. En programación, los algoritmos son implementaciones computacionales que transforman una entrada en una salida, siguiendo una lógica ordenada.

Análisis de algoritmos:

El análisis de algoritmos es el proceso de estudiar el rendimiento de un algoritmo, principalmente su tiempo de ejecución y uso de memoria ya que el objetivo del análisis es predecir el comportamiento del algoritmo antes de su ejecución, comparando su desempeño teórico con otros algoritmos que resuelven el mismo problema.
La complejidad algorítmica indica cómo varía el uso de los recursos **(tiempo y memoria)**, según el tamaño de la entrada, el cual es representado comúnmente por la letra *n*. Se utiliza la notación Big O para expresar la cantidad de operaciones necesarias en **el peor de los casos.**

Importancia de Big O:

La notación Big O es esencial en informática porque permite **evaluar y comparar la eficiencia de los algoritmos**. Estas son las razones principales:

1. **Comparación de rendimiento**:
   Permite determinar qué algoritmo funciona mejor al aumentar el tamaño de la entrada.
   Por ejemplo, si tomamos como parámetro la complejidad temporal (cómo aumenta el tiempo de ejecución de un algoritmo a medida que crece el tamaño de la entrada (denotado como n)), un algoritmo con complejidad O(n) será más eficiente que uno con $O(n^2)$ para entradas grandes.

   Para **n = 100**:

O(n): ~100 operaciones.

O(n^2): ~10,000 operaciones.

2. **Predicción de Escalabilidad**:
   Ayuda a anticipar el comportamiento de un algoritmo con entradas más grandes, esencial en sistemas que manejan grandes volúmenes de datos, como aplicaciones web o bases de datos. Por ejemplo, **Merge Sort** (O(n log n)) escala mejor que **Bubble Sort** (O(n^2)).

3. **Detección de cuellos de botella**:
   Facilita la detección de secciones de código que limitan el rendimiento, como bucles anidados o recursiones ineficientes, permitiendo optimizaciones específicas.

4. **Análisis de complejidad temporal y espacial**:
   Big O ofrece un marco formal para medir el tiempo y la memoria que requiere un algoritmo, crucial en aplicaciones que demandan alta velocidad y eficiencia, como aprendizaje automático o sistemas embebidos.

Aplicaciones Prácticas de Big O:

La notación **Big O** trasciende la teoría y tiene usos concretos en el desarrollo de software:

- **Análisis de Algoritmos Comunes**: Se utiliza para evaluar la eficiencia de algoritmos de búsqueda, ordenamiento, recursión y programación dinámica. Ejemplos incluyen:
  - Búsqueda: Lineal (O(n)) frente a binaria (O(log n)).
  - Ordenamiento: Burbuja (O(n^2)) frente a combinación (O(n log n)).
  - Recursión: Fibonacci recursivo (O(2^n)) frente a programación dinámica (O(n)).
  - Programación dinámica: Convierte algoritmos costosos en soluciones más rápidas.
- **Programación Competitiva**: En concursos con límites de tiempo estrictos, **Big O** ayuda a seleccionar algoritmos que se ejecuten dentro de los plazos. Por ejemplo, para entradas de tamaño 10^6, algoritmos con complejidad superior a O(n log n) suelen ser inviables.
- **Diseño de Sistemas Reales**: En sistemas complejos, como aplicaciones web o distribuidas, **Big O** se usa para:
  - Seleccionar estructuras de datos óptimas (ej., B-trees con O(log n) para bases de datos).
  - Diseñar algoritmos eficientes para balanceo de carga.
  - Evaluar el impacto de consultas a gran escala.
- **Optimización de Código**: Permite identificar áreas críticas de rendimiento. Por ejemplo, si un perfilador detecta un algoritmo O(n^2) en una sección clave, optimizarlo a O(n log n) puede mejorar significativamente el desempeño general.

La notación **Big O** es esencial para comparar algoritmos, prever su escalabilidad y optimizar sistemas, siendo una herramienta clave tanto en entornos académicos como en programación competitiva y desarrollo profesional.

3. Caso Práctico

Para ilustrar el análisis de algoritmos, se plantea el problema de buscar un número en una lista de enteros. Se implementaron dos algoritmos de búsqueda: **búsqueda lineal** y **búsqueda binaria**, cuyos rendimientos se comparan para determinar cuál es más adecuado según el contexto.

Búsqueda Lineal

La búsqueda lineal examina cada elemento de la lista de forma secuencial hasta localizar el valor objetivo o determinar que no está presente.

Procedimiento:

- Comienza en el primer elemento.
- Compara el elemento con el objetivo.
- Si coinciden, retorna su índice.
- Si no, avanza al siguiente elemento.
- Si se recorre toda la lista sin éxito, retorna -1 (no encontrado).

Complejidad Temporal:

- **Peor caso**: O(n) – el elemento está al final o no existe.
- **Mejor caso**: O(1) – el elemento está al inicio.

Ventajas:

- No requiere que la lista esté ordenada.
- Implementación sencilla y directa.

Desventajas:

- Ineficiente para listas extensas.
- Más lento que la búsqueda binaria cuando la lista está ordenada.

Búsqueda Binaria

La búsqueda binaria opera en una lista ordenada, dividiendo el espacio de búsqueda a la mitad en cada iteración para localizar el objetivo de forma eficiente.

Procedimiento:

- Selecciona el elemento central de la lista.
- Si coincide con el objetivo, retorna su índice.
- Si el objetivo es menor, busca en la mitad izquierda.
- Si es mayor, busca en la mitad derecha.
- Repite hasta encontrar el objetivo o agotar las opciones.

Requisito:

- La lista debe estar ordenada previamente; de lo contrario, el algoritmo falla.

Complejidad Temporal:

- **Mejor caso**: O(1) – el objetivo está en el centro.
- **Peor caso**: O(log n) – el espacio se reduce a la mitad en cada paso.

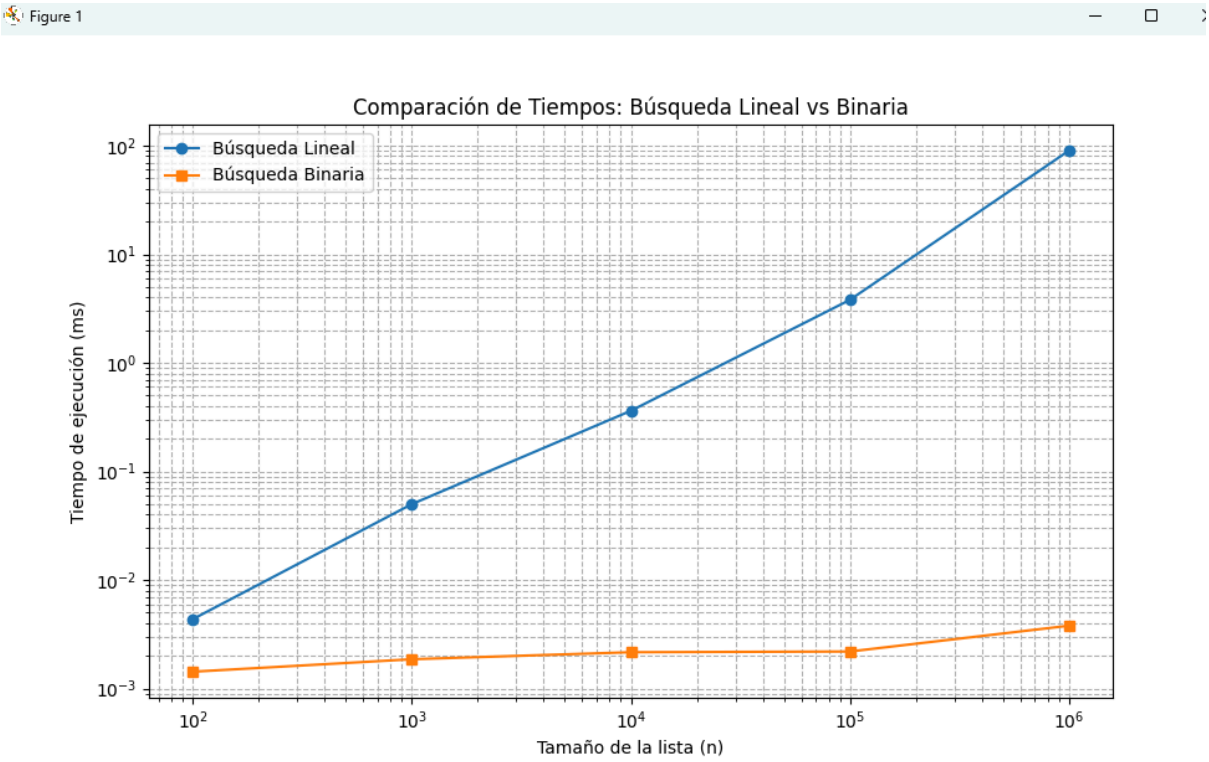
Ventajas:

- Mucho más rápida que la búsqueda lineal en listas grandes.
- Escala eficientemente con el tamaño de la lista.

Desventajas:

- Requiere una lista ordenada.
- Ligeramente más compleja de implementar.

Ejemplo:



4. Metodología Utilizada:

Para el desarrollo de este trabajo se siguieron los siguientes pasos:

Investigación previa: Se investigaron fuentes teóricas relevantes sobre algoritmos de búsqueda, analizando su complejidad algorítmica, eficiencia temporal y casos de uso prácticos.

Diseño y prueba del código: Se implementaron los algoritmos de búsqueda lineal y búsqueda binaria. Las pruebas se realizaron con listas generadas aleatoriamente y ordenadas, condición necesaria para el correcto funcionamiento de la búsqueda binaria.

Herramientas utilizadas: El desarrollo se llevó a cabo en Python 3.12, utilizando el entorno de desarrollo Visual Studio Code. Se emplearon las librerías estándar random para la generación de datos y time para medir el rendimiento. La librería matplotlib se utilizó para generar gráficos comparativos.

Trabajo colaborativo: El proyecto se desarrolló en equipo, organizando las tareas en cuatro etapas: investigación, implementación, validación de resultados y documentación.

5. Resultados Obtenidos

Con el desarrollo del caso práctico se lograron los siguientes resultados:

- **Casos de prueba:**

Se generó una lista ordenada de 1.000.000 elementos aleatorios para comparar el rendimiento de los algoritmos de búsqueda lineal y binaria con un mismo conjunto de datos. El elemento buscado fue el último de la lista para simular el peor caso en la búsqueda lineal. Además, se evaluaron cinco listas de diferentes tamaños (100, 1.000, 10.000, 100.000, 1.000.000) para analizar las variaciones en el tiempo de ejecución. Los resultados muestran que la búsqueda binaria es significativamente más rápida que la lineal, especialmente en listas grandes, confirmando su eficiencia logarítmica.

- **Errores corregidos:**

Se ajustó la generación de la lista para evitar elementos duplicados, lo que podría alterar el comportamiento de búsqueda binaria. Además, se verificó la correcta medición del tiempo y se descartaron valores atípicos en algunas pruebas.

6. Conclusiones

A través de este trabajo se pudo:

- Demostrar la diferencia significativa en rendimiento entre la búsqueda lineal y binaria, destacando la eficiencia logarítmica de esta última en listas ordenadas de gran tamaño.
- Confirmar la relevancia del análisis de algoritmos para optimizar soluciones en escenarios con grandes volúmenes de datos.
- Fortalecer habilidades en la implementación de algoritmos, medición de tiempos y visualización de resultados mediante gráficos.

7. Bibliografía

- Python Software Foundation (2024). *The Python Language Reference*. https://docs.python.org/3/
- Cómo analizar tus algoritmos (en Ingeniería Informática)(BettaTech) (2025). https://www.youtube.com/watch?v=IZgOEC0NIbw
- Think Python: An Introduction to Software Design - Autor: Allen B. Downey - (2002)
- Matplotlib: Visualization with Python - (2025) - https://matplotlib.org/
- Análisis de algoritmos (2025) - https://es.wikipedia.org/wiki/An%C3%A1lisis_de_algoritmos
- Introducción a la Complejidad Computacional (2025) - http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap01.htm

8. Anexos

Código fuente completo:
El código utilizado para el caso práctico está disponible como archivo adjunto (Prueba análisis de algoritmo-Ronderos-Sarru.py) y se encuentra debidamente comentado e indentado.