Trabajo Integrador

Propuesta de Investigación de Programación I

Análisis de Algoritmos en Python

Alumnos: Sierra Iván, Siles Cristian
Materia: Programación I
Profesor: Quirós Nicolás
Tutor: Bianchi Neyén
Fecha de Entrega: 9 de Junio de 2025

Índice:

1- Introducción:

Se decidió elegir este tópico ya que representa un tema fundamental en el estudio de la programación y la informática. El análisis de algoritmos permite comprender cómo se comportan los algoritmos en cuanto a su eficiencia y uso de recursos, lo que lleva a tomar mejores decisiones al desarrollar software.
Además, en la programación existen siempre varias formas de resolver un mismo problema, por lo que ante varios algoritmos que den el mismo resultado, es importante tener una manera de comprarlos y así poder elegir el más adecuado según el contexto.

El análisis de algoritmos es clave tanto en la etapa formativa como en el ejercicio profesional de un desarrollador ya que nos permitirá evaluar la eficiencia temporal (tiempo de ejecución) y la eficiencia espacial (uso de memoria) al conocer la complejidad algorítmica. Con estos datos es posible anticipar cómo va a escalar un programa frente a entradas más grandes, evitando cuellos de botella y mejorando el rendimiento del software.

En el siguiente trabajo se busca alcanzar los siguientes objetivos:

- Comprender los fundamentos del análisis de algoritmos.
- Desarrollar un pensamiento crítico y eficiente para elegir estructuras de datos o algoritmos.
- Medir y comparar la eficiencia temporal y espacial de distintas soluciones.
- Utilizar herramientas y estructuras de Python para crear algoritmos clásicos y entender 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, un algoritmo con complejidad O(n) será más eficiente que uno con $O(n^2)$ para entradas grandes.

2. **Predicción de escalabilidad**:
   Ayuda a anticipar cómo se comportará un algoritmo con entradas crecientes, algo crucial en sistemas con grandes volúmenes de datos, como aplicaciones web o bases de datos.
   Ejemplo: Merge Sort (O(n log n)) escala mejor que Bubble Sort ($O(n^2)$).

3. **Detección de cuellos de botella**:
   Facilita la localización de partes del código que limitan el rendimiento, como bucles anidados o recursiones costosas, lo cual es clave para optimizar.

4. **Análisis de complejidad temporal y espacial**:
   Big O es el lenguaje formal para analizar cuánto **tiempo** y cuánta **memoria** necesita un algoritmo.
   Esto es fundamental en sistemas que deben ser rápidos y eficientes, como en aprendizaje automático, big data o sistemas embebidos.

Usos de Big O:

Big O no es solo teórica: se aplica en múltiples contextos del desarrollo de software. Algunos usos principales son:

- **Evaluación de algoritmos comunes**:
  Se analiza la eficiencia de algoritmos de búsqueda, ordenamiento, recursión y programación dinámica.

    - Búsqueda: lineal (O(n)) vs. binaria (O(log n)).
    - Ordenamiento: burbuja ($O(n^2)$) vs. combinación (O(n log n)).
    - Recursión: Fibonacci ingenuo ($O(2^n)$) vs. con programación dinámica (O(n)).
    - Programación dinámica: permite transformar algoritmos costosos en versiones más eficientes.

- **Programación competitiva**:
  En competencias con restricciones de tiempo, Big O permite elegir algoritmos que se ejecuten dentro del tiempo permitido.
  Ej.: con entradas de tamaño 10^6, algoritmos peores que O(n log n) podrían exceder el tiempo límite.

- **Diseño y optimización de sistemas reales**:
  En sistemas complejos (como aplicaciones web y distribuidas), se usa para:

    - Elegir estructuras de datos eficientes (ej.: B-trees con O(log n) para bases de datos).
    - Diseñar algoritmos de balanceo de carga eficientes.
    - Evaluar impacto de consultas a gran escala.

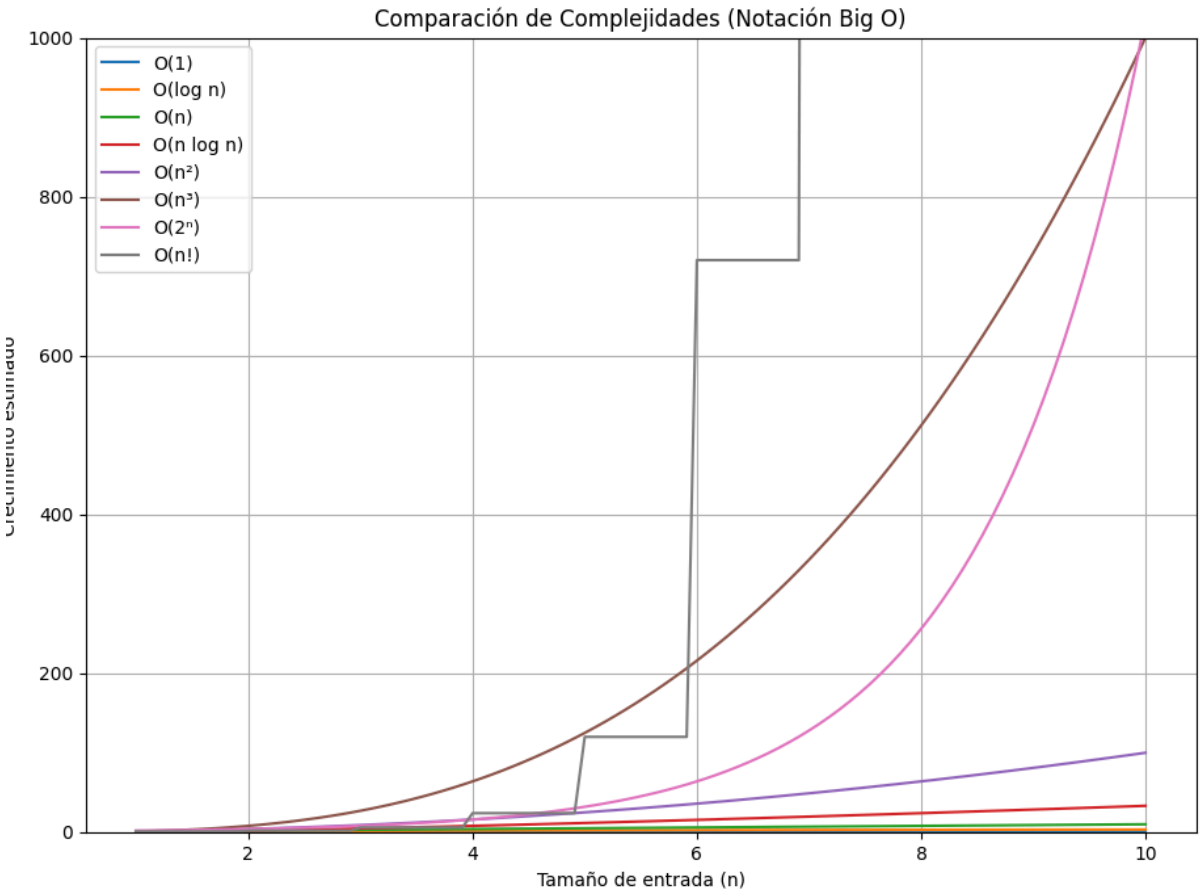
- **Optimización de código**:
  Permite detectar puntos críticos de rendimiento.
  Si un perfilador muestra un algoritmo $O(n^2)$ en una sección crítica, puede optimizarse a O(n log n) para mejorar el desempeño general.

Big O es clave para comparar algoritmos, anticipar escalabilidad y encontrar cuellos de botella. Es esencial tanto en entornos académicos como en programación competitiva y desarrollo de sistemas reales.

Notaciones Big O

Notación Big O	Nombre	Descripción
O(1)	Tiempo constante	La ejecución no depende del tamaño de la entrada.
O(log n)	Tiempo logarítmico	El tiempo crece lentamente con entradas grandes.
O(n)	Tiempo lineal	El tiempo crece en proporción directa al tamaño de entrada.
O(n log n)	Lineal logarítmico	Eficiente para algoritmos de ordenamiento óptimos.
$O(n^2)$	Tiempo cuadrático	Ineficiente para entradas grandes.
$O(n^3)$	Tiempo cúbico	Muy ineficiente, crece rápidamente.
$O(2^n)$	Tiempo exponencial	Imposible de manejar para entradas grandes.
O(n!)	Tiempo factorial	Extremadamente lento, prácticamente inutilizable para n > 10.

Gráfico de complejidades:



Generado con librería matplotlib

Estructuras de datos en Python:

Las estructuras de datos son fundamentales dentro del rendimiento de los algoritmos. Ya que el rendimiento también depende de la estructura de datos utilizada.

El uso adecuado de estructuras de datos permite optimizar el rendimiento de los algoritmos y adaptar soluciones a distintos contextos.

Específicamente en Python, las estructuras más relevantes son:

- Listas (list): Son útiles para almacenamiento dinámico, acceso O(1), pero inserción y borrado pueden ser costosos.

```
precios_frutas = {'Banana': 1200, 'Ananá': 2500, 'Melón': 3000, 'Uva':
1450}

precios_frutas["Naranja"] = 1200
precios_frutas["Manzana"] = 1500
precios_frutas["Pera"] = 2300
```

- Conjuntos(set): Permiten realizar búsquedas rápidas en O(1) en promedio, **eliminan duplicados automáticamente.**

```
listado_anios = [1993,2000,2024,1993]
conjunto_anios = set(listado_anios)
print(conjunto_anios) #{2000, 1993, 2024}
```

- Diccionarios(dict): Son estructuras que se rigen por clave-valor, lo cual las hace muy eficientes para búsquedas.

```
reloj: dict = {
    "hora": 18,
    "minuto": 34,
    "segundo": 22
}

print(f"{reloj["hora"]}:{reloj["minuto"]}:{reloj["segundo"]}")
#18:34:22
```

- **Pilas y colas:** Se pueden implementar con list o collections.deque, siendo ideales para problemas que requieren control de orden de ejecución.

Ejemplo de Cola:

```python
class Cola:
    def __init__(self) -> None:
        self.elementos = deque()

    def agregar_cliente(self, elemento):
        self.elementos.append(elemento)

    def atender_cliente(self):
        self.elementos.popleft()

    def mostrar_siguiente_cliente(self):
        return self.elementos[0] if self.elementos else "La lista esta vacia"
```

Ejemplo de Pila:

```python
from collections import deque

class Pila:
    def __init__(self, claves:dict = { "(":")", "[":"]", "{":"}" }) -> None:
        self.elementos = deque()
        self.claves = claves

    def esta_vacia(self):
        return len(self.elementos) == 0

    def apilar(self, elemento):
        self.elementos.append(elemento)

    def desapilar(self):
        return self.elementos.pop() if not self.esta_vacia() else
"La pila está vacía"

    def ver_tope(self):
        return self.elementos[-1] if not self.esta_vacia() else
"La pila está vacía"
```

Implementación:
Python brinda sintaxis clara y herramientas integradas para escribir y analizar algoritmos. Por ejemplo, el módulo *time* permite medir el tiempo de ejecución, y *sys* puede utilizarse para medir el uso de memoria.

Ejemplo de medición del tiempo de ejecución:

```python
import time

inicio = time.time()
resultado = sum(range(1000000))
fin = time.time()

print("Tiempo de ejecución:", fin - inicio, "segundos")
```

Finalizando:
El análisis de algoritmos tiene un papel esencial en el diseño de software eficiente. Permite anticipar problemas de rendimiento y elegir la mejor solución entre múltiples alternativas. En campos como la inteligencia artificial, el análisis de datos, la seguridad informática o la programación de sistemas, el uso de algoritmos eficientes puede marcar la diferencia entre una aplicación funcional y una inutilizable.

3. Caso Práctico:
Se desea encontrar un número en una lista de enteros, por lo cual se implementaron dos algoritmos de búsqueda:

- **Búsqueda lineal**: recorre la lista elemento por elemento hasta encontrar el valor.
  Secuencia:
  1. Se empieza en el primer elemento de la lista.
  2. Se compara con el objetivo.
  3. Si son iguales, devuelve la posición (índice).
  4. Si no, pasa al siguiente elemento.
  5. Si se llega al final sin encontrarlo, se devuelve -1 (no encontrado).

Complejidad temporal:
Peor caso: O(n) → si el elemento está al final o no está.
Mejor caso: O(1) → si el elemento está al principio.

Ventajas:
- No necesita que la lista esté ordenada.
- Fácil de implementar.

Desventajas:
- Poco eficiente para listas grandes.

- Mucho más lento que otros algoritmos como la búsqueda binaria, si la lista está ordenada.

- **Búsqueda binaria**: busca dividiendo la lista ordenada a la mitad en cada paso.
  Secuencia:
    1. Se toma el elemento del medio de la lista.
    2. Si el elemento es igual al objetivo, lo devuelve.
    3. Si el objetivo es menor, busca en la mitad izquierda.
    4. Si el objetivo es mayor, busca en la mitad derecha.
    5. Repite el proceso hasta encontrarlo o agotar la lista.
    6. El objetivo es comparar su rendimiento y entender en qué casos conviene usar cada uno.

Requisito:
La lista debe estar ordenada previamente, si no, el algoritmo no funciona correctamente.

Complejidad temporal:
Mejor caso: O(1) → si el objetivo está justo en el medio.
Peor caso: O(log n) → se reduce a la mitad en cada iteración.

Ventajas:
- Mucho más rápida que la búsqueda lineal en listas grandes.
- Escala muy bien con el tamaño de la lista.

Desventajas:
- Solo funciona si la lista está ordenada.
- Un poco más compleja de implementar que la búsqueda lineal.

Ejemplo:

```python
import time
import random

# Algoritmo de búsqueda lineal (O(n))
def busqueda_lineal(lista, objetivo):
    for i, valor in enumerate(lista):
        if valor == objetivo:
            return i
    return -1

# Algoritmo de búsqueda binaria (O(log n))
def busqueda_binaria(lista, objetivo):
    izquierda = 0
    derecha = len(lista) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# Generar lista ordenada
n = 100_000
lista = sorted(random.sample(range(n * 2), n))  # Sin repetidos
objetivo = lista[-1]  # Elegimos el último número para forzar el peor caso

# Medir tiempos
inicio = time.time()
resultado_lineal = busqueda_lineal(lista, objetivo)
tiempo_lineal = time.time() - inicio

inicio = time.time()
resultado_binaria = busqueda_binaria(lista, objetivo)
tiempo_binaria = time.time() - inicio

# Resultados
print(f"Búsqueda lineal: índice {resultado_lineal}, tiempo: {tiempo_lineal:.6f}
 seg")
print(f"Búsqueda binaria: índice {resultado_binaria}, tiempo: {tiempo_binaria:.6f}
 seg")
```

Captura de la terminal:

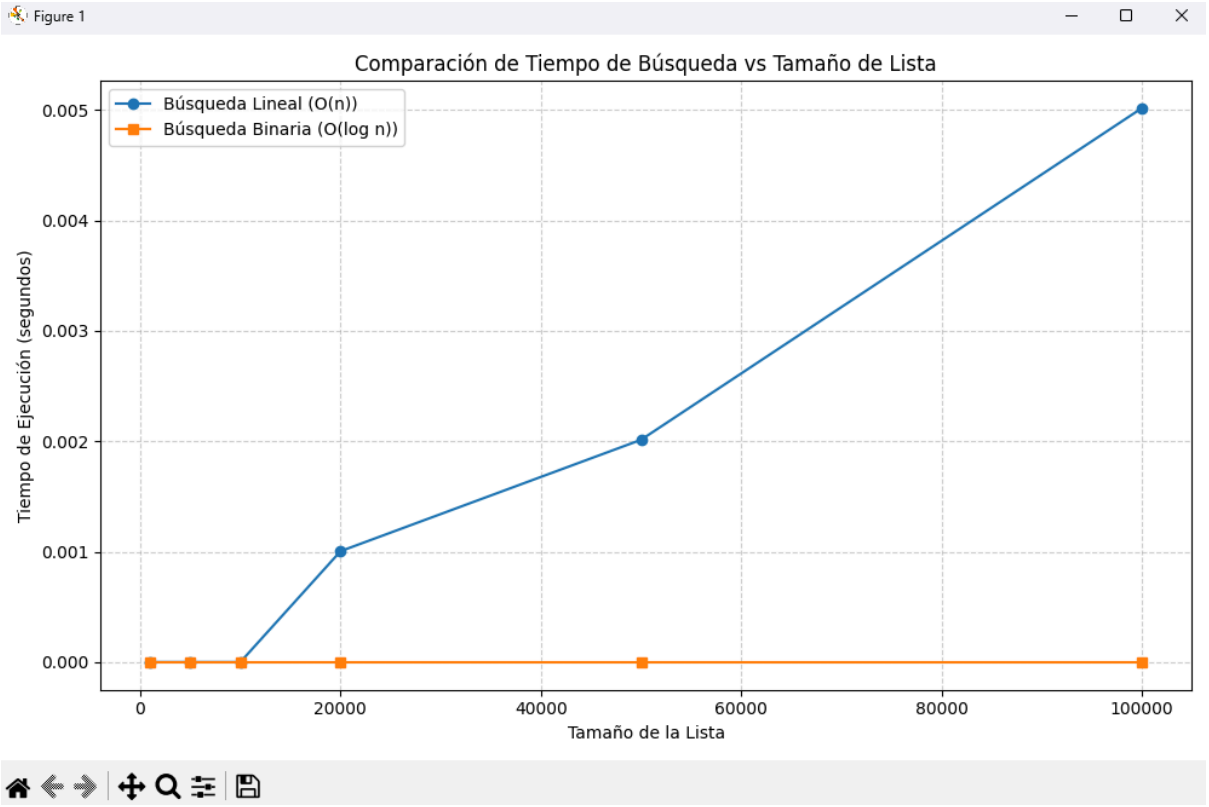
```
UTN-TUPaD-P1 on  main [✗?] took 2h33m1s
❯ & C:/Users/sierr/AppData/Local/Programs/Python/Pytho
Búsqueda lineal: índice 99999, tiempo: 0.004203 seg
Búsqueda binaria: índice 99999, tiempo: 0.000000 seg
```

Ejemplo 2 con generación de gráfico:

```python
import time
import random
import matplotlib.pyplot as plt

# Algoritmos
def busqueda_lineal(lista, objetivo):
    for i, valor in enumerate(lista):
        if valor == objetivo:
            return i
    return -1

def busqueda_binaria(lista, objetivo):
    izquierda = 0
    derecha = len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# Rangos de tamaño de listas para comparar
tamaños = [1_000, 5_000, 10_000, 20_000, 50_000, 100_000]
tiempos_lineal = []
tiempos_binaria = []

# Medición de tiempos por tamaño
for n in tamaños:
    lista = sorted(random.sample(range(n * 2), n))
    objetivo = lista[-1]  # peor caso

    inicio = time.time()
    busqueda_lineal(lista, objetivo)
    tiempos_lineal.append(time.time() - inicio)

    inicio = time.time()
    busqueda_binaria(lista, objetivo)
    tiempos_binaria.append(time.time() - inicio)

# 📈 Gráfico de líneas
plt.figure(figsize=(10, 6))
plt.plot(tamaños, tiempos_lineal, label='Búsqueda Lineal (O(n))', marker='o')
plt.plot(tamaños, tiempos_binaria, label='Búsqueda Binaria (O(log n))', marker=
's')
plt.title('Comparación de Tiempo de Búsqueda vs Tamaño de Lista')
plt.xlabel('Tamaño de la Lista')
plt.ylabel('Tiempo de Ejecución (segundos)')
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()
```

En base a nuestro código original, realizamos algunas modificaciones para incorporar la generación de gráficos con la librería *matplotlib.* Se evaluarán listas de 5 tamaños distintos que generan un gráfico lineal.

Captura del gráfico generado:



Decisiones de diseño:

- Se elige una lista ordenada para que ambos métodos sean aplicables.
- Se selecciona el último valor como objetivo para simular el peor caso en la búsqueda lineal.
- La búsqueda lineal es útil en listas pequeñas o no ordenadas.
- La búsqueda binaria es óptima cuando la lista está ordenada y grande.

Validación del funcionamiento:

Ambos métodos devuelven el mismo índice, lo que valida su funcionamiento correcto. Sin embargo, se evidencia que la búsqueda binaria tiene un tiempo de ejecución mucho menor, lo que demuestra la ventaja de su menor complejidad algorítmica.

4. Metodología Utilizada:

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

- **Investigación previa:**
  Se consultaron fuentes teóricas sobre algoritmos de búsqueda, su complejidad algorítmica y eficiencia temporal.

- **Diseño y prueba del código:**
  Se implementaron dos algoritmos: búsqueda lineal y búsqueda binaria. La lista utilizada para las pruebas fue generada aleatoriamente pero ordenada, requisito indispensable para que la búsqueda binaria funcione correctamente. Para forzar el peor caso en la búsqueda lineal, se eligió el último elemento como objetivo.

- **Herramientas utilizadas:**
  El desarrollo se realizó en Python 3.12, utilizando el IDE **Visual Studio Code**. Se emplearon las librerías estándar `random` y `time` para generación de datos y medición de rendimiento. Y se utilizó la librería *matplotlib* para generación de gráficos. El código fue versionado localmente con Git y probado en diferentes ejecuciones para asegurar su estabilidad.

- **Trabajo colaborativo:**
  El trabajo fue realizado de forma grupal, organizando las tareas en etapas: investigación, codificación, pruebas y documentación.

5. Resultados Obtenidos:

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

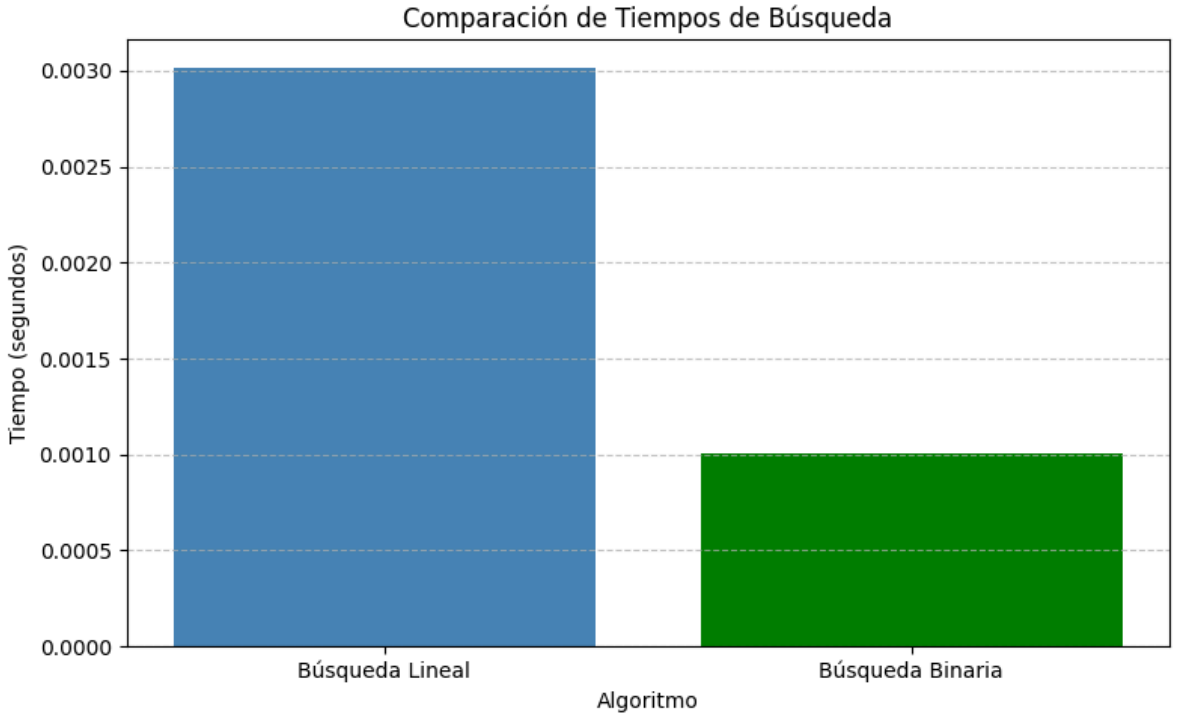
- **Casos de prueba:**

Se generó una lista de 100.000 elementos únicos y ordenados para evaluar ambos algoritmos con el mismo conjunto de datos. El elemento buscado fue el último de la lista para simular el peor caso en la búsqueda lineal. También se realizó una prueba con 5 listas con distintos tamaños (1000, 5000, 10000, 20000, 50000, 100000) para ver en detalle las variaciones de rendimiento.

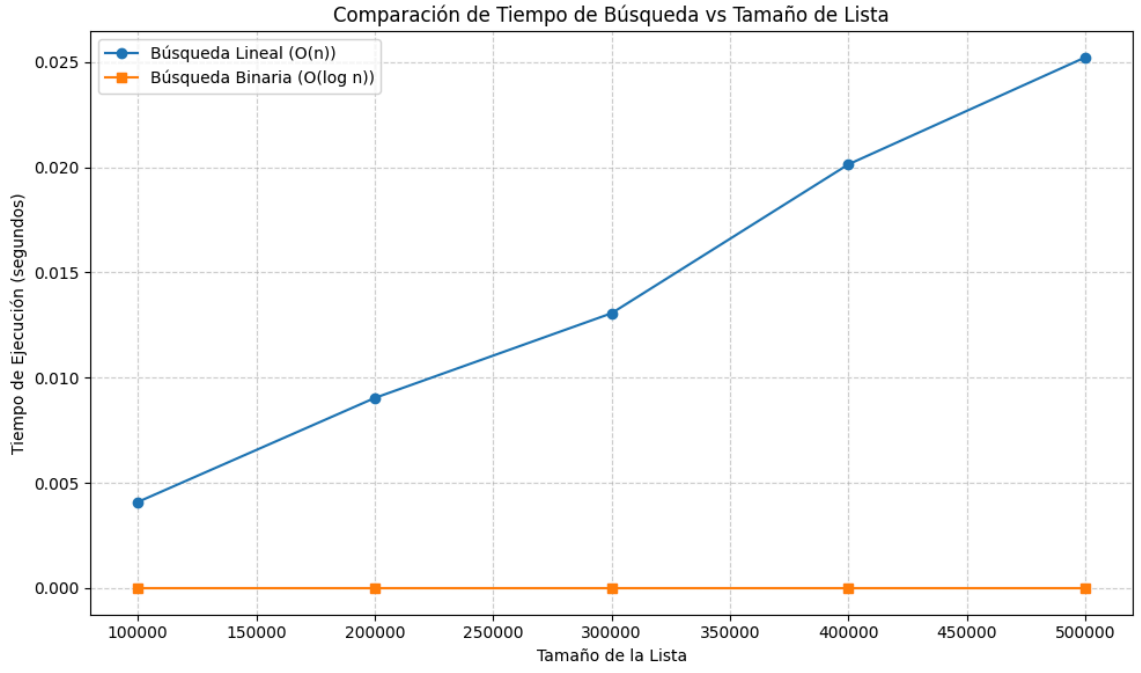
- **Evaluación de rendimiento:**

Se midieron los tiempos de ejecución utilizando la librería *time*.

El siguiente gráfico compara el rendimiento de los algoritmos con una única lista de 100.000 elementos.



El siguiente gráfico compara el rendimiento de los algoritmos con 5 listas de los siguientes tamaños: [100_000, 200_000, 300_000, 400_000, 500_000].



Esto demuestra que la búsqueda binaria es significativamente más eficiente, especialmente en listas grandes, validando su complejidad logarítmica frente al comportamiento lineal del otro método.

- **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:
Ejemplo práctico:
A través de este trabajo se pudo:

- Comprender en profundidad cómo dos algoritmos que resuelven el mismo problema pueden tener rendimientos muy distintos dependiendo de su complejidad algorítmica.

- Se evidenció la importancia del análisis de algoritmos al momento de tomar decisiones en programación, especialmente para tareas que involucran grandes volúmenes de datos.

- Se reforzaron conocimientos sobre estructuras de datos, medición de tiempos y pruebas comparativas.

- Como posibles mejoras futuras, se propone automatizar múltiples pruebas con distintos tamaños de entrada y graficar los resultados para visualizar las curvas de rendimiento.

- Entre las dificultades resueltas, se destaca la necesidad de garantizar que la lista esté ordenada para el correcto funcionamiento de la búsqueda binaria, lo cual se resolvió usando *sorted()* sobre la lista generada aleatoriamente.

Teóricas:

- **Pruebas y perfiles:**
  Aunque la notación Big O proporciona un marco teórico, es fundamental realizar pruebas reales para evaluar el rendimiento.
  Los desarrolladores deben:

  - Ejecutar los algoritmos con distintos tamaños de entrada.

  - Medir el tiempo de ejecución y uso de memoria en entornos concretos. Esto permite detectar diferencias entre lo esperado teóricamente y el comportamiento en producción.

- **Mejora iterativa:**
  El desarrollo de algoritmos no es estático:

  - Las aplicaciones evolucionan.

  - Es necesario ajustar y optimizar algoritmos de forma continua según las métricas obtenidas. Este ciclo de mejora garantiza que la solución se mantenga eficiente a lo largo del tiempo.

- **El contexto es clave:**
  La elección del algoritmo óptimo depende:

  - Del problema específico.

  - De las características de los datos.

  - Del entorno de ejecución. Comprender el contexto permite tomar decisiones informadas que impactan directamente en el rendimiento real.

Comprender la notación Big O es crucial para el análisis de algoritmos y la eficiencia en la informática.
Sin embargo, no debería ser la única consideración sino ser tomada como parte de un análisis más amplio que combine teoría con pruebas empíricas y consideraciones de ingeniería.

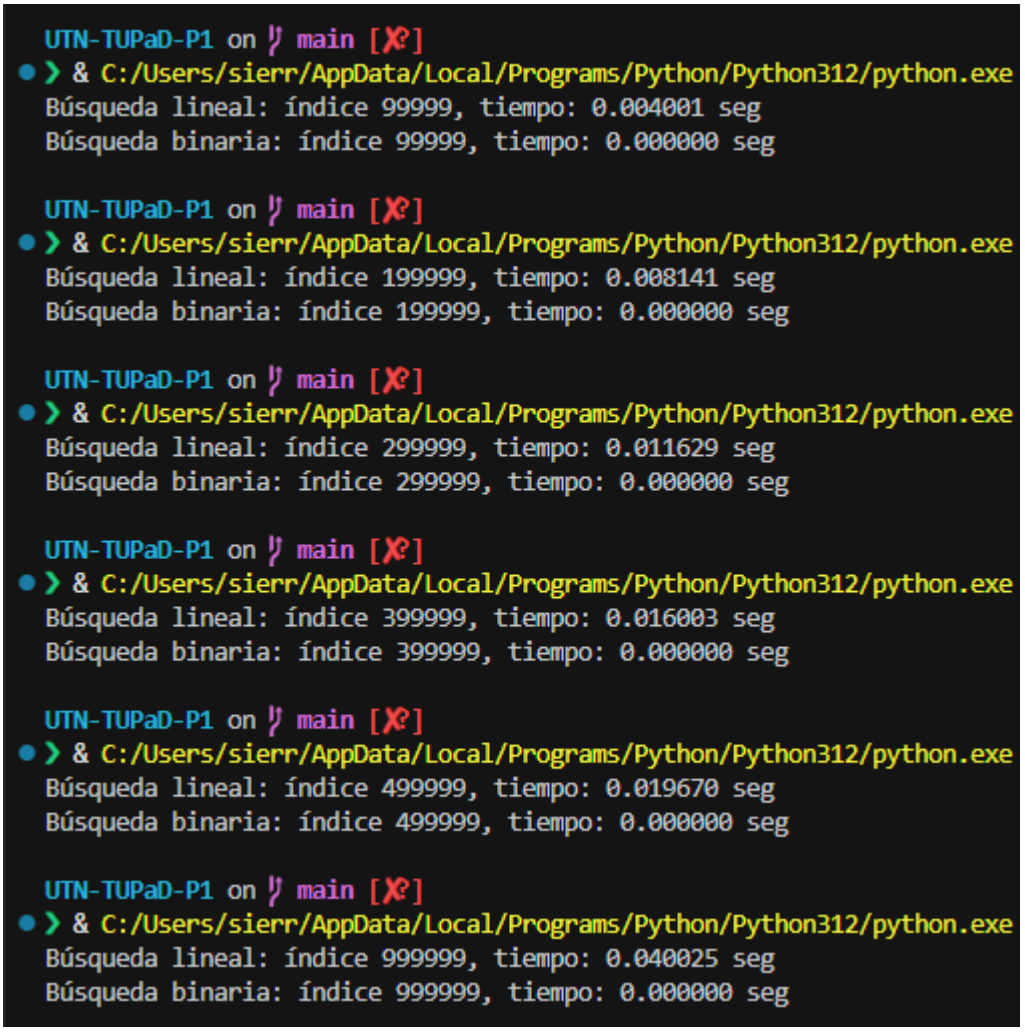
7. Bibliografía:

- 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

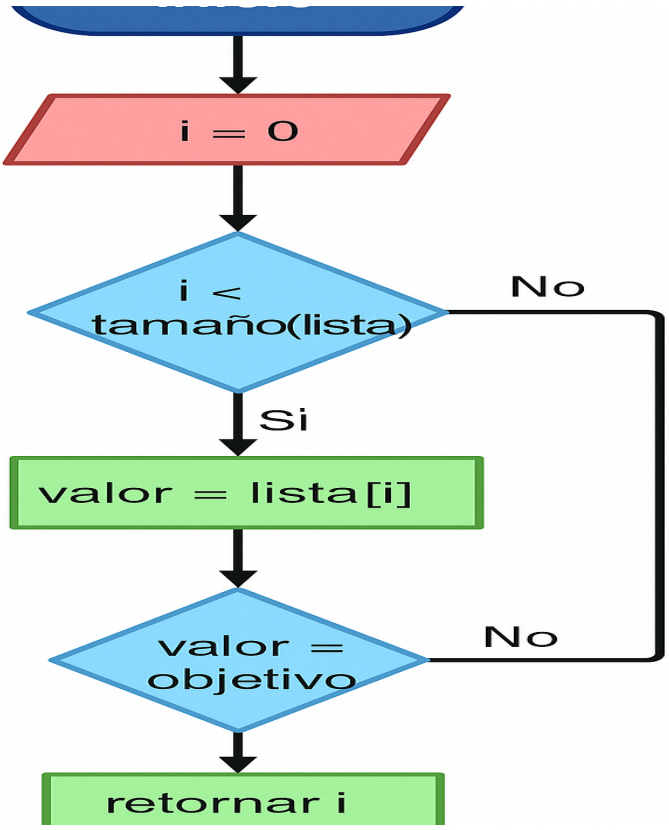
8. Anexos:
A continuación se incluye material complementario que respalda y enriquece el desarrollo del trabajo práctico:

Código fuente completo:
El código utilizado para el caso práctico está disponible como archivo adjunto (codigo_busqueda.py, codigo_busqueda_grafico_lineas.py, codigo_busqueda_grafico_barras.py y grafico_bigO.py) y se encuentra debidamente comentado e identado.

Comparativa de rendimiento:
Se incluye una captura de la terminal, donde se fue ampliando el tamaño de la lista ordenada, con el fin de visualizar cómo escalan los tiempos de ejecución de ambos algoritmos conforme aumenta el tamaño de la entrada.



Diagramas de flujo:
Se pueden agregar diagramas que expliquen el funcionamiento de los algoritmos implementados, facilitando la comprensión visual del proceso.



Código para generar el gráfico de barras:

```python
# 📊 Gráfico de barras comparativo
algoritmos = ['Búsqueda Lineal', 'Búsqueda Binaria']
tiempos = [tiempo_lineal, tiempo_binaria]

plt.figure(figsize=(8, 5))
plt.bar(algoritmos, tiempos, color=['steelblue', 'green'])
plt.title('Comparación de Tiempos de Búsqueda')
plt.ylabel('Tiempo (segundos)')
plt.xlabel('Algoritmo')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

Código para generar el gráfico de tipos de notaciones Big O:

```python
import matplotlib.pyplot as plt
import numpy as np
import math

# Rango de entrada
n = np.linspace(1, 10, 100)

# Definición de funciones de complejidad
O_1 = np.ones_like(n)
O_log_n = np.log2(n)
O_n = n
O_n_log_n = n * np.log2(n)
O_n2 = n ** 2
O_n3 = n ** 3
O_2n = 2 ** n
O_fact_n = [math.factorial(int(i)) if i <= 10 else np.nan for i in n]
# Evitamos overflow

# Crear el gráfico
plt.figure(figsize=(12, 8))
plt.plot(n, O_1, label="O(1)")
plt.plot(n, O_log_n, label="O(log n)")
plt.plot(n, O_n, label="O(n)")
plt.plot(n, O_n_log_n, label="O(n log n)")
plt.plot(n, O_n2, label="O(n²)")
plt.plot(n, O_n3, label="O(n³)")
plt.plot(n, O_2n, label="O(2ⁿ)")
plt.plot(n, O_fact_n, label="O(n!)")

# Ajustes estéticos
plt.ylim(0, 1000)  # Limitar eje Y para que todo sea visible
plt.title("Comparación de Complejidades (Notación Big O)")
plt.xlabel("Tamaño de entrada (n)")
plt.ylabel("Crecimiento estimado")
plt.legend()
plt.grid(True)
plt.tight_layout()

# Mostrar gráfico
plt.show()
```