Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

TRABAJO INTEGRADOR PROGRAMACIÓN I

Título: “Análisis de la eficiencia algorítmica en Python: Búsqueda lineal y búsqueda binaria”

Alumnos: GUDIÑO, Gabriel

HERBEL, José Ignacio

Materia: Programación I

Docente Titular: Prof. Ariel Enferrel

Fecha de entrega: 09/06/2025

ÍNDICE

1) Introducción

El análisis de algoritmos es una herramienta fundamental para evaluar el rendimiento de distintas soluciones computacionales. En particular, cuando se trata de buscar elementos dentro de una lista, la elección del algoritmo adecuado puede marcar una gran diferencia en términos de eficiencia temporal y uso de recursos.

Buscar eficientemente es una de las tareas más comunes y críticas en programación. Desde sistemas de bases de datos hasta algoritmos en motores de búsqueda, la forma en que localizamos información impacta directamente en la velocidad general del sistema. La elección entre una búsqueda secuencial y una búsqueda binaria dependerá no solo del tamaño de los datos, sino también de si están o no ordenados.

Este trabajo práctico compara dos algoritmos de búsqueda ampliamente conocidos: búsqueda lineal y búsqueda binaria. Ambos son estudiados desde el punto de vista teórico y empírico. Se analiza su comportamiento frente al crecimiento del tamaño de entrada, se implementan en Python, y se miden sus tiempos de ejecución en distintos tamaños de datos ordenados para obtener una comparación clara de su eficiencia relativa. dos algoritmos de búsqueda ampliamente conocidos: búsqueda lineal y búsqueda binaria. Ambos son estudiados desde el punto de vista teórico y empírico utilizando implementaciones en Python y midiendo sus tiempos de ejecución en distintos tamaños de datos.

2) Marco Teórico

Fundamento de búsqueda secuencial y binaria

En el capítulo 6 de The Art of Computer Programming, Vol. 3: Sorting and Searching (Knuth, 1998), se introducen las principales estrategias de búsqueda mediante comparación de claves. Knuth distingue dos enfoques fundamentales: la búsqueda secuencial, adecuada cuando no existe un orden en los datos, y la búsqueda binaria, que aprovecha una relación de orden para optimizar la búsqueda.

Algoritmo 1: Búsqueda Lineal

Según Knuth en *The Art of Computer Programming* (1998, cap. 6.1), la búsqueda lineal (o secuencial) “es el método más obvio y a menudo el único posible cuando no hay otra estructura que explotar”. Este algoritmo compara directamente cada elemento con el valor deseado, uno por uno, sin requerir ningún orden previo. Es simple de implementar, pero su rendimiento decrece a medida que crece la cantidad de datos.

Pseudocódigo:

Supongamos una lista de tamaño n. El pseudocódigo recorre cada elemento de la lista y compara con el objetivo:

```
para i desde 0 hasta n-1:
   si lista[i] == valor:
      retornar i
retornar -1
```

Analicemos cuántas operaciones elementales se ejecutan en el peor caso:

- Una comparación de índice (*i < n*) por iteración
- Una comparación de igualdad (*lista[i] == valor*) por iteración
- Un incremento de i por iteración
- Una instrucción final (*retornar -1*) si no se encuentra el valor

Cada iteración realiza 3 operaciones, y se repiten n veces:

$T(n) = 3n + 1$

Esta es la función de tiempo exacta en el peor caso. Ahora bien, cuando analizamos el crecimiento de esta función para valores grandes de n, descartamos constantes y términos menores:

O(T(n)) = O(3n + 1) = O(n)

Por lo tanto, la búsqueda lineal tiene una complejidad temporal lineal. Ya que cada paso ejecuta una comparación constante y se repite “n” veces. La función asintótica es:

O(n)

Ventajas:

- Sencillez.
- No requiere orden, aunque puede aplicarse también sobre listas ordenadas.

Desventajas:

- Ineficiente para listas grandes.

Algoritmo 2: Búsqueda Binaria

Tal como describe Knuth en The Art of Computer Programming (1998, cap. 6.2.1), la búsqueda binaria es una técnica eficiente para listas ordenadas, ya que permite reducir el espacio de búsqueda a la mitad en cada paso. “Después de comparar el argumento K con una clave Ki, el dominio de búsqueda se reduce drásticamente”, señala. Este comportamiento hace que el algoritmo sea especialmente útil en contextos donde se requiere acceder a grandes volúmenes de datos de manera eficiente.

Análisis del pseudocódigo:

Supongamos una lista de tamaño n:

inicio = 0
fin = n - 1
mientras inicio <= fin:

```
medio = (inicio + fin) // 2
si lista[medio] == valor:
    retornar medio
sino si lista[medio] < valor:
    inicio = medio + 1
sino:
    fin = medio - 1
```

En el análisis completo, debemos considerar también las operaciones iniciales antes del bucle:

- Asignación *inicio = 0*
- Cálculo *fin = n - 1* (una resta y una asignación)

Estas operaciones iniciales representan un costo constante y no afectan la complejidad asintótica del algoritmo. Luego, en cada iteración del bucle se realizan:

- Una comparación *inicio <= fin*
- Un cálculo de índice medio
- Una comparación de igualdad
- Una comparación para decidir si ir a la izquierda o derecha

Cada iteración realiza un número constante de operaciones. El número total de iteraciones corresponde a la cantidad de veces que se puede dividir el rango de búsqueda a la mitad hasta alcanzar 1:

$$T(n) = c \cdot \log_2(n)$$

Al eliminar la constante para obtener la complejidad asintótica:

$$O(T(n)) = O(\log n)$$

Por tanto, la búsqueda binaria es logarítmica en tiempo en el peor caso.

Ventajas:

- Muy rápido en listas grandes.

- Consistente en eficiencia.

Desventajas:

- Requiere orden previo.

Comparación de funciones T(n)

A partir del análisis realizado:

- Búsqueda Lineal: $T(n) = n$
- Búsqueda Binaria: $T(n) = log_2(n)$

La función n crece mucho más rápido que $log_2(n)$, lo que significa que la búsqueda binaria es asintóticamente más eficiente. En términos prácticos, cuando el tamaño de la lista es grande, la diferencia en tiempo de ejecución entre ambos algoritmos es notable. Aunque la búsqueda lineal puede aplicarse a listas sin orden, en este caso se analiza su rendimiento sobre listas ordenadas, la búsqueda binaria es preferible para grandes volúmenes de datos ordenados.

3) Caso Práctico

Para garantizar la validez del análisis, se ha trabajado exclusivamente con listas ordenadas. Esta condición es imprescindible para aplicar correctamente el algoritmo de búsqueda binaria. Si bien la búsqueda lineal no requiere orden previo, se ha mantenido el mismo conjunto de datos para poder comparar ambos algoritmos bajo igualdad de condiciones.

En este apartado se realiza un análisis empírico de ambos algoritmos utilizando implementaciones en Python. Se evalúan los tiempos de ejecución para distintos tamaños de entrada y se comparan los resultados medidos con el comportamiento teórico previamente analizado.

Además, se incluye un gráfico comparativo generado automáticamente que ilustra el tiempo de ejecución de cada algoritmo a medida que aumenta el tamaño de la lista, facilitando la visualización de las diferencias de eficiencia entre ambos enfoques.En este apartado se realiza un análisis empírico de ambos algoritmos utilizando implementaciones en Python. Se evalúan los

tiempos de ejecución para distintos tamaños de entrada y se comparan los resultados medidos con el comportamiento teórico previamente analizado.

- Análisis teórico de la complejidad de cada algoritmo.
- Implementación en Python.
- Generación de listas ordenadas de distintos tamaños.
- Medición del tiempo de ejecución utilizando el módulo time.
- Comparación de resultados empíricos.

Código (búsqueda_algoritmos.py)

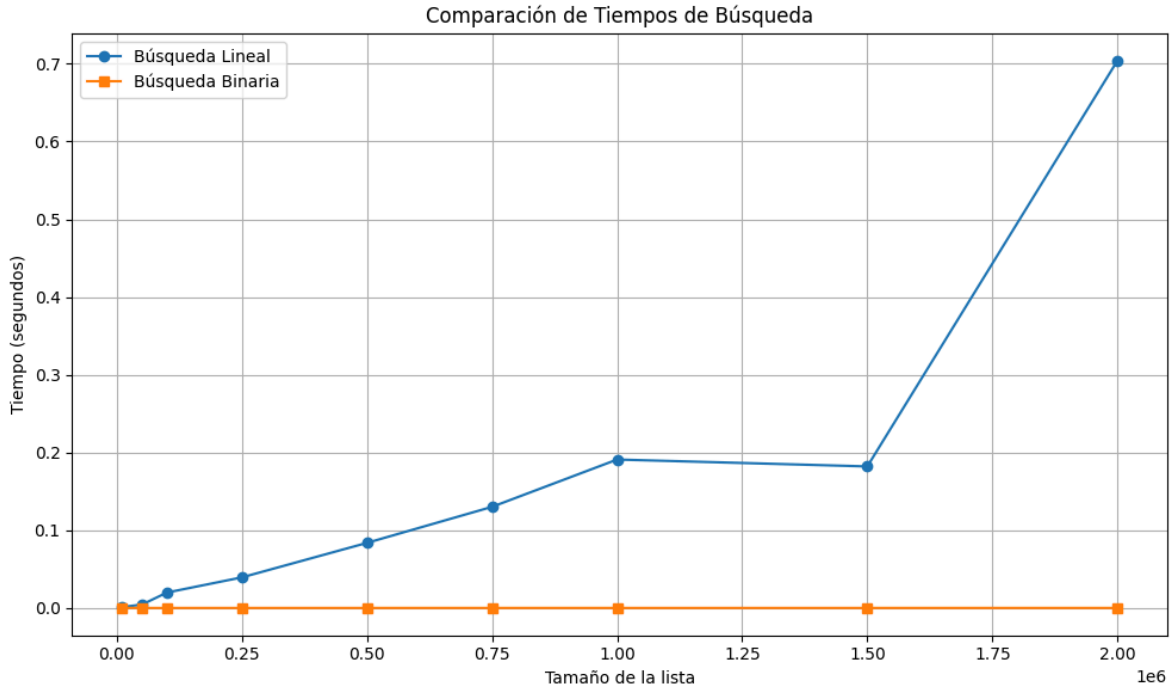
```
import time
import random
import matplotlib.pyplot as plt

def busqueda_lineal(arr, objetivo):
    for i, val in enumerate(arr):
        if val == objetivo:
            return i
    return "No encontrado"

def busqueda_binaria(arr, objetivo):
    inicio, fin = 0, len(arr) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if arr[medio] == objetivo:
            return medio
        elif arr[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
```

```
    return "No encontrado"

def medir_tiempo(funcion, arr, objetivo):
    inicio = time.time()
    resultado = funcion(arr, objetivo)
    fin = time.time()
    return resultado, fin - inicio

if __name__ == "__main__":
    tamanos = [10000, 50000, 100000, 250000, 500000, 750000, 1000000, 1500000, 2000000]
    tiempos_lineal = []
    tiempos_binaria = []

    for n in tamanos:
        lista = sorted(random.sample(range(n * 10), n))
        objetivo = lista[-1]  # elemento al final para peor caso en búsqueda lineal

        _, tiempo_lineal = medir_tiempo(busqueda_lineal, lista, objetivo)
        _, tiempo_binaria = medir_tiempo(busqueda_binaria, lista, objetivo)

        tiempos_lineal.append(tiempo_lineal)
        tiempos_binaria.append(tiempo_binaria)

        print(f"n={n} | Lineal: {tiempo_lineal:.6f}s | Binaria: {tiempo_binaria:.6f}s")

    # Gráfico comparativo
    plt.figure(figsize=(10, 6))
    plt.plot(tamanos, tiempos_lineal, marker='o', label='Búsqueda Lineal')
    plt.plot(tamanos, tiempos_binaria, marker='s', label='Búsqueda Binaria')
    plt.title('Comparación de Tiempos de Búsqueda')
```

```
plt.xlabel('Tamaño de la lista')
plt.ylabel('Tiempo (segundos)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Resultados Obtenidos

n	Lineal	binaria
10000	0.001226s	0.000007s
50000	0.004259s	0.000007s
100000	0.019810s	0.000012s
250000	0.039351s	0.000010s
500000	0.083740s	0.000013s
750000	0.130099s	0.000019s
1000000	0.190950s	0.000010s
1500000	0.182071s	0.000014s
2000000	0.704118s	0.000020s



4) Metodología Utilizada

Para garantizar la validez del análisis, se ha trabajado exclusivamente con listas ordenadas. Esta condición es imprescindible para aplicar correctamente el algoritmo de búsqueda binaria. Si bien la búsqueda lineal no requiere orden previo, se ha mantenido el mismo conjunto de datos para poder comparar ambos algoritmos bajo igualdad de condiciones.

- Análisis teórico de la complejidad de cada algoritmo.
- Implementación en Python.
- Generación de listas ordenadas de distintos tamaños.
- Medición del tiempo de ejecución utilizando la librería `time`.
- Representación gráfica de los tiempos utilizando la librería `matplotlib`.
- Comparación de resultados empíricos.

5) Conclusiones

Ambos algoritmos cumplen su función de encontrar elementos dentro de una lista. Sin embargo, la diferencia en eficiencia es notable cuando el volumen de datos aumenta. La búsqueda binaria demuestra una escalabilidad excelente gracias a su complejidad logarítmica, mientras que la búsqueda lineal se vuelve ineficiente en tamaños grandes.

Este trabajo evidencia la importancia de comprender la complejidad algorítmica al elegir una solución, especialmente en aplicaciones que manipulan grandes volúmenes de datos.

Desde una perspectiva teórica, la función T(n) asociada a cada algoritmo justifica los resultados observados en la práctica. El comportamiento lineal de la búsqueda secuencial implica un aumento proporcional del tiempo a medida que crece la entrada, mientras que la búsqueda binaria crece muy lentamente incluso con millones de elementos.

Además, el gráfico empírico refuerza visualmente esta diferencia: mientras que la línea correspondiente a la búsqueda lineal crece de forma constante, la curva de la búsqueda binaria permanece casi horizontal, indicando una estabilidad en su rendimiento.

En conclusión, si se cuenta con listas ordenadas, la búsqueda binaria debe ser el método preferido en la mayoría de los casos debido a su gran eficiencia. La búsqueda lineal, por otro lado, puede seguir siendo útil en listas pequeñas o cuando no se garantiza un orden previo.

6) Anexos

Captura de resultados de la ejecución

```
n=10000 | Lineal: 0.001226s | Binaria: 0.000007s
n=50000 | Lineal: 0.004259s | Binaria: 0.000007s
n=100000 | Lineal: 0.019810s | Binaria: 0.000012s
n=250000 | Lineal: 0.039351s | Binaria: 0.000010s
n=500000 | Lineal: 0.083740s | Binaria: 0.000013s
n=750000 | Lineal: 0.130099s | Binaria: 0.000019s
n=1000000 | Lineal: 0.190950s | Binaria: 0.000010s
n=1500000 | Lineal: 0.182071s | Binaria: 0.000014s
n=2000000 | Lineal: 0.704118s | Binaria: 0.000020s
```

Repositorio de Github

https://github.com/ggudinodata/integrador_prog1.git

Video explicativo

Enlace en archivo readme.md

7) Bibliografía

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.