Trabajo Integrador de Programación: Algoritmos de Búsqueda Binaria y Ordenamiento en Python.

Alumno: Maximo Perrotta

1. Introducción:

Se decidió abordar el estudio conjunto de algoritmos de búsqueda binaria y ordenamiento, dos pilares esenciales en el mundo de la informática, dado que la eficiencia de búsqueda muchas veces depende directamente del estado previo (ordenado o no) de los datos.

La motivación principal radica en comprender cómo elegir el algoritmo adecuado según el problema planteado y el tamaño del conjunto de datos, y en visualizar de forma práctica la diferencia que existe entre algoritmos sencillos pero poco eficientes (como Bubble Sort) frente a otros ligeramente más optimizados (como Insertion Sort), destacando el rol de estos procesos en la búsqueda binaria.

2. Marco Teórico:

Algoritmos de Búsqueda

- **Búsqueda Lineal:** recorre cada elemento de la lista secuencialmente hasta encontrar el objetivo o llegar al final. Tiene

- una complejidad de O(n), lo que significa que su tiempo de ejecución crece proporcionalmente al tamaño de la lista. Es simple, no requiere orden previo, pero se vuelve ineficiente con listas grandes.

- **Búsqueda Binaria:** sólo funciona si la lista está ordenada. Divide el rango de búsqueda a la mitad en cada iteración, descartando la mitad no útil. Su complejidad es O(log n), por lo que el tiempo crece muy lentamente aunque la lista aumente de tamaño. Es ampliamente utilizado en sistemas donde se requiere alta eficiencia, como en motores de búsqueda o en bases de datos indexadas.

Algoritmos de Ordenamiento

Ordenar datos es una operación previa indispensable para muchos algoritmos de búsqueda y análisis. El tiempo que demore este proceso puede condicionar el rendimiento global de cualquier aplicación.

- **Bubble Sort:** es un método intuitivo que compara pares adyacentes e intercambia sus posiciones si están desordenados, repitiendo el proceso múltiples veces hasta que la lista queda ordenada. Su desventaja principal es la complejidad O(n^2), lo que lo vuelve poco práctico para listas grandes, pero muy didáctico para aprender lógica de comparación e intercambio.

- **Insertion Sort:** construye gradualmente la lista ordenada, insertando cada elemento en su posición correcta. Tiene el mismo orden de complejidad (O(n^2)), pero tiende a hacer menos comparaciones e intercambios cuando la lista ya está parcialmente ordenada, por lo que puede superar a Bubble Sort en casos concretos.

Análisis del rendimiento

En algoritmos, se utiliza el concepto de complejidad temporal (Big O) para expresar cómo escala el tiempo de ejecución en función del tamaño de entrada (n). Esto no mide tiempo en segundos, sino el número relativo de operaciones.

Por ejemplo:

- Bubble e Insertion tienen O(n^2), su tiempo crece rápidamente al duplicar el tamaño de la lista.

- Búsqueda Binaria tiene O(log n), su tiempo crece muy lentamente aunque la lista crezca mucho.

Esto es crucial en la programación real: un programa que maneje miles o millones de datos debe planificarse eligiendo cuidadosamente los algoritmos, o puede volverse impracticable.

3. Caso Práctico:

Descripción del desarrollo

Como caso práctico, se diseñó un programa modular en Python, estructurado con funciones separadas para cada algoritmo:

- bubble_sort(lista): ordena la lista usando el método Bubble.
- insertion_sort(lista): ordena la lista con Insertion Sort.
- busqueda_binaria(lista, elemento): busca un elemento en la lista ordenada.
- medir_tiempo(func, lista, *args): mide el tiempo exacto que tarda cualquier función.

Además, se utilizó el módulo random para generar listas aleatorias y time para obtener mediciones precisas en segundos.

Procedimiento seguido

El programa realiza las siguientes etapas:

1. Generación de una lista pequeña de 10 enteros aleatorios entre 1 y 100.
2. Se muestra la lista original sin ordenar, evidenciando el punto de partida desorganizado.
3. Se aplica Bubble Sort y se muestra la lista ordenada, junto con el tiempo que demoró.
4. Luego se aplica Insertion Sort sobre una copia de la misma lista original, comparando resultados y tiempos.
5. Con la lista ya ordenada, se elige un elemento conocido y se lo busca usando Búsqueda Binaria, mostrando la posición encontrada y el tiempo consumido.
6. Para cerrar, se repite el experimento con una lista grande de 5000 elementos aleatorios entre 1 y 10000, observando cómo aumentan los tiempos de ordenamiento en comparación con la lista pequeña.

Ejemplo ilustrativo

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Función Bubble_sort.

```
/PrimerCuatrimestre/Programacion/NuevaEntrega
$ C:/Users/Maxgard/AppData/Local/Microsoft/WindowsApps/python3.13.exe d:/FacuUTN/PrimerCuatrimestre/Programacion
/NuevaEntrega/busqueda_y_ordenamiento.py

*** Pruebas con listas pequeñas ***

Lista original: [21, 20, 75, 90, 35, 45, 40, 9, 10, 95]
Ordenada con Bubble: [9, 10, 20, 21, 35, 40, 45, 75, 90, 95]
Tiempo Bubble Sort: 0.000012 segundos

Ordenada con Insertion: [9, 10, 20, 21, 35, 40, 45, 75, 90, 95]
Tiempo Insertion Sort: 0.000010 segundos

Buscando 90 en la lista ordenada -> posición 8
Tiempo Búsqueda Binaria: 0.000004 segundos


*** Pruebas con listas grandes ***

Tiempo Bubble Sort con lista grande: 1.525069 segundos
Tiempo Insertion Sort con lista grande: 0.738146 segundos
```

Salida típica del script.

Código y recursos

Todo el código fuente fue subido a un repositorio público en GitHub, acompañado de un readme con instrucciones y reflexiones. También se grabó un video mostrando la ejecución en vivo, demostrando los algoritmos en acción con listas pequeñas y grandes.

4. Metodología Utilizada:

Para el desarrollo del caso práctico se siguieron los siguientes pasos metodológicos:

- **Investigación teórica:** recopilando definiciones, ventajas y desventajas de cada algoritmo, así como ejemplos de complejidad temporal.
- **Diseño modular:** planificando cada algoritmo como una función autónoma en Python para favorecer claridad, reutilización y mantenimiento.

- **Implementación incremental:** primero se codificaron los algoritmos con listas pequeñas, verificando su correcto funcionamiento, para luego escalar a listas grandes.

- **Medición precisa:** empleando la librería time para obtener datos objetivos sobre el rendimiento.

- **Documentación:** agregando comentarios al código para que sea autoexplicativo.

- **Pruebas variadas:** comparando listas pequeñas y grandes, observando cómo escalan los tiempos y fundamentando así la elección de algoritmos según el contexto.

5. Resultados obtenidos:

Los resultados demostraron claramente cómo:

- Bubble Sort e Insertion Sort son viables para listas pequeñas, con tiempos casi imperceptibles.
- Al incrementar el tamaño a 5000 elementos, ambos algoritmos muestran un crecimiento pronunciado en el tiempo de ejecución, aunque Insertion Sort logra un leve mejor desempeño.
- La Búsqueda Binaria, en cambio, mantiene tiempos despreciables independientemente del tamaño, siempre que los datos estén previamente ordenados.

Esto permite concluir que el verdadero cuello de botella en la eficiencia no radica en la búsqueda binaria, sino en el proceso previo de ordenamiento. Por ello, en contextos profesionales se suelen emplear algoritmos mucho más eficientes como QuickSort o MergeSort.

6. Conclusiones:

Gracias a este trabajo se pudo:

- Visualizar de manera práctica cómo la eficiencia algorítmica impacta en los tiempos reales de ejecución, reforzando la importancia de analizar la complejidad.

- Confirmar que no existe un algoritmo universal, sino que la elección depende del volumen de datos y del problema concreto.

- Aprender a modularizar y medir tiempos en Python, habilidades esenciales en el desarrollo de software de calidad.

Finalmente, el trabajo integró teoría y práctica, afianzando conceptos clave de algoritmos, estructuras de datos y análisis de rendimiento.

7.Bibliografía:

- Cormen, T. et al. *Introduction to Algorithms*. MIT Press.
- Python Docs. *time module*. https://docs.python.org/3/library/time.html
- GeeksforGeeks. *Bubble Sort, Insertion Sort, Binary Search*. https://www.geeksforgeeks.org/
- Visualgo. *Data Structures and Algorithms*. https://visualgo.net/en
- Documentación oficial Python: https://docs.python.org/3/

8.Anexos:

Link Repositorio:
https://github.com/MaximoPerrotta/IntegralBusquedaYOrdenamiento

Link video YouTube:
https://youtu.be/yWMhN3oHJSE