Comparación y análisis de algoritmos de búsqueda y ordenamiento en Python

Alumnos: Valentín Piñeyro - Danilo Peirano
Materia: Programación I
Profesor/a: Prof. Nicolás Quirós

Fecha de Entrega: 09/06/2025

1. Introducción

Este trabajo aborda el análisis empírico y comparación de algoritmos de búsqueda y ordenamiento aplicados a una simulación de buscador de canciones. Este tema fue elegido debido a su relevancia fundamental en la programación, dado que la organización y recuperación eficiente de información son tareas centrales en la mayoría de los sistemas informáticos.

El objetivo principal es evaluar el rendimiento de distintos algoritmos como Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, y las búsquedas lineal y binaria, aplicados a listas de distinto tamaño. Se busca comprender el impacto del tamaño de datos y la elección del algoritmo en el tiempo de ejecución, así como validar los conceptos aprendidos en un contexto práctico.

2. Marco Teórico

Algoritmos de ordenamiento

- **Bubble Sort**: algoritmo de comparación que intercambia elementos adyacentes si están en el orden incorrecto. Ineficiente para listas grandes (O(n^2)).
- **Insertion Sort**: construye la lista ordenada insertando elementos uno a uno. Eficiente para listas pequeñas (O(n^2)).
- **Selection Sort**: selecciona el elemento más pequeño y lo coloca al inicio, iterativamente. También O(n^2).
- **Quick Sort**: algoritmo basado en el principio “divide y vencerás”. Selecciona un punto central y divide la lista en dos mitades (menores y mayores que el punto medio). Muy eficiente en la práctica (O(n log n) promedio).

Algoritmos de búsqueda

- **Búsqueda Lineal**: recorre secuencialmente la lista y compara cada elemento. Sencilla, pero ineficiente en listas grandes (O(n)).
- **Búsqueda Binaria**: requiere lista ordenada. Divide el espacio de búsqueda a la mitad en cada paso (O(log n)).

Implementación en Python

Se utilizan estructuras de datos tipo lista de diccionarios. La clave de ordenamiento o búsqueda puede ser "nombre", "artista" o "género".

Documentación oficial consultada:

- Python Software Foundation. (2023). *The Python Standard Library*. https://docs.python.org/3/
- Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

3. Caso Práctico: Simulación de un buscador de canciones

Descripción del problema

Se desarrolló un programa que simula un buscador de canciones, permitiendo al usuario ingresar un valor (nombre, artista o género) y buscar coincidencias mediante algoritmos de búsqueda. Además, se ordena la lista de canciones por distintos algoritmos para aplicar la búsqueda binaria y medir rendimientos.

Capturas de código relevantes

```
print("\n=== BÚSQUEDA LINEAL ===")
for campo in campos:
    inicio = time.perf_counter()
    resultado = busqueda_lineal(lista, campo, valor_busqueda)
    fin = time.perf_counter()
    tiempos_lineal.append(fin - inicio)
    mostrar_resultados(f"Lineal por {campo}", resultado)
    print(f"Tiempo búsqueda lineal: {fin - inicio:.8f} segundos")
```

```
print("\n=== BÚSQUEDA BINARIA (CON ORDENAMIENTO) ===")
for nombre_algoritmo, algoritmo in ordenamientos.items():
    print(f"\n--- {nombre_algoritmo.upper()} ---")
    for campo in campos:
        inicio_ordenamiento = time.perf_counter()
        lista_ordenada = algoritmo(lista, campo)
        fin_ordenamiento = time.perf_counter()

        inicio_busqueda = time.perf_counter()
        resultado = busqueda_binaria(lista_ordenada, campo, valor_busqueda)
        fin_busqueda = time.perf_counter()
```

Decisiones de diseño

- Se permite al usuario elegir el tamaño de la lista, ya que el programa busca testear la eficiencia de los distintos algoritmos con distintos tamaños de listas (10 a 10.000).
- El programa realiza todas las búsquedas y ordenamientos en paralelo.
- Se mide el tiempo de ordenamiento y búsqueda con `time.perf_counter()`.

Validación

Se realizaron pruebas con distintos valores y tamaños de lista. Se comprobaron coincidencias, rendimiento, y corrección de errores (ej. listas no ordenadas para búsqueda binaria).

4. Metodología Utilizada

- **Etapas:**
  - Relevamiento de algoritmos.
  - Implementación modular en Python.
  - Validación de casos de prueba.
  - Análisis de rendimiento.
- **Fuentes:** documentación oficial de Python, libros de algoritmos, foros como Stack Overflow.
- **Herramientas:**
  - VSCode como editor.
  - Librería `time` para medir ejecución.
  - `random` para generar datos.
  - `sys` para aumentar el límite de recursión
  - Control de versiones con Git.

- **Trabajo colaborativo:** Se investigó en conjunto sobre los temas a tratar y se escribió código colaborativamente a través de GitHub.

5. Resultados Obtenidos

Funcionamiento

- Las búsquedas lineales funcionaron correctamente en todos los casos.
- La búsqueda binaria requirió ordenar previamente las listas.

Rendimiento

Ejemplo para lista de 10 canciones:

Algoritmo	Tiempo Ordenamiento (seg)	Tiempo Búsqueda (seg)
Búsqueda lineal	X	0.00000544
Binaria con Bubble Sort	0.00001004	0.00000484
Binaria con Insertion Sort	0.00000837	0.00000277
Binaria con Selection Sort	0.00001002	0.00000266
Binaria con Quick Sort	0.00001418	0.00000351

Ejemplo para lista de 100 canciones:

Algoritmo	Tiempo Ordenamiento (seg)	Tiempo Búsqueda (seg)
Búsqueda lineal	X	0.00003287
Binaria con Bubble Sort	0.00075443	0.00000974
Binaria con Insertion Sort	0.00106037	0.00001168

Binaria con Selection Sort	0.00472040	0.00004810
Binaria con Quick Sort	0.00035871	0.00000774

Ejemplo para lista de 1000 canciones:

Algoritmo	**Tiempo Ordenamiento (seg)**	**Tiempo Búsqueda (seg)**
Búsqueda lineal	X	0.00022902
Binaria con Bubble Sort	0.10296386	0.00006825
Binaria con Insertion Sort	0.04587870	0.00006503
Binaria con Selection Sort	0.07977930	0.00006528
Binaria con Quick Sort	0.02957137	0.00007214

Ejemplo para lista de 10000 canciones:

Algoritmo	**Tiempo Ordenamiento (seg)**	**Tiempo Búsqueda (seg)**
Búsqueda lineal	X	0.00167025
Binaria con Bubble Sort	13.09690369	0.00003973
Binaria con Insertion Sort	5.57386582	0.00002987
Binaria con Selection Sort	7.17156810	0.00002882
Binaria con Quick Sort	1.84649113	0.00002519

Dificultades

- Necesidad de ordenar por cada campo para aplicar búsqueda binaria.
- Las diferencias de tiempo entre ordenamientos fueron muy notorias.

6. Conclusiones

Este trabajo permitió comprender de forma práctica las diferencias de rendimiento entre algoritmos de ordenamiento y búsqueda. Se destacó la eficiencia de Quick Sort para ordenar, y de la búsqueda binaria para listas ordenadas.

Se comprobó que:

- La elección del algoritmo impacta fuertemente en el tiempo total de ejecución.
- En listas pequeñas, la diferencia no es tan marcada, pero se vuelve crítica a mayor escala.

Posibles mejoras:

- Incorporar algoritmos más avanzados como Merge Sort o el uso de sorted().
- Permitir ordenamientos por múltiples criterios.
- Analizar el consumo de memoria.

7. Bibliografía

- Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Python Software Foundation. (2023). *The Python Standard Library*. https://docs.python.org/3/
- Knuth, D. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.
- Stack Overflow. (n.d.). https://stackoverflow.com/

8. Anexos

- Se anexa la documentación del código para mayor claridad.