

Algoritmos de Búsqueda y Ordenamiento en Python

Alumnas:

Daiana Gamarra daiana.gamarra.cp@gmail.com

Comisión: 14

Lucía Chamorro lucychamorro@gmail.com

Comisión: 11

Materia: Programación I

Profesor/a: Ariel Enferrel

Fecha de Entrega: 20/06/2025

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

1. Introducción

El presente trabajo se enfoca en la investigación, desarrollo y análisis de algoritmos de búsqueda y ordenamiento en el lenguaje de programación Python. Estos algoritmos constituyen herramientas fundamentales en la programación, ya que permiten organizar y localizar información de manera eficiente.

La elección de este tema se debe a su aplicabilidad en problemas reales y a la importancia de comprender su funcionamiento interno para poder seleccionar adecuadamente cuál usar en función del contexto. Este trabajo se propone implementar diferentes algoritmos, medir su rendimiento y reflexionar sobre su utilidad práctica.

2. Marco Teórico

Algoritmos de Ordenamiento:

- Bubble Sort: algoritmo simple que compara y ordena elementos adyacentes. Ineficiente para listas grandes ($O(n^2)$).
- Insertion Sort: eficiente en listas pequeñas o casi ordenadas. Complejidad promedio $O(n^2)$.
- Merge Sort: método divide y vencerás, garantiza $O(n \log n)$ incluso en el peor caso.
- Quick Sort: muy eficiente en promedio ($O(n \log n)$), aunque puede llegar a $O(n^2)$ si no se elige bien el pivote.

Algoritmos de Búsqueda:

- Búsqueda Lineal: recorre la lista elemento por elemento. $O(n)$.
- Búsqueda Binaria: requiere lista ordenada. Divide por la mitad cada vez. Complejidad $O(\log n)$.

3. Caso Práctico

Se diseñó un programa en Python que genera una lista de 10.000 números aleatorios y luego aplica distintos algoritmos de ordenamiento (Bubble Sort, Merge Sort, Quick Sort) y búsqueda (Lineal y Binaria). Se mide el tiempo de ejecución de cada uno.

El código fuente se encuentra organizado en módulos con funciones claramente comentadas y estructuras reutilizables.

4. Metodología Utilizada

1. Se investigaron conceptos teóricos en libros, documentación oficial y cursos online.
2. Se diseñó el programa modularmente.
3. Se utilizaron listas generadas aleatoriamente con ``random``.
4. Se hicieron pruebas repetidas para validar resultados.
5. Se subió el código a GitHub para documentación y versionado.

5. Resultados Obtenidos

- Quick Sort resultó ser el más rápido, incluso con grandes volúmenes de datos.
- La búsqueda binaria fue mucho más rápida que la búsqueda lineal en listas ordenadas.
- Se identificaron diferencias notables en los tiempos de ejecución según el algoritmo.

6. Conclusiones

- El trabajo permitió comprender la importancia de la eficiencia algorítmica.
- Se reforzaron conceptos de programación estructurada y uso de módulos.
- La experiencia práctica fue fundamental para interiorizar la teoría.
- Como mejora futura, se propone implementar árboles binarios de búsqueda para ampliar el análisis.

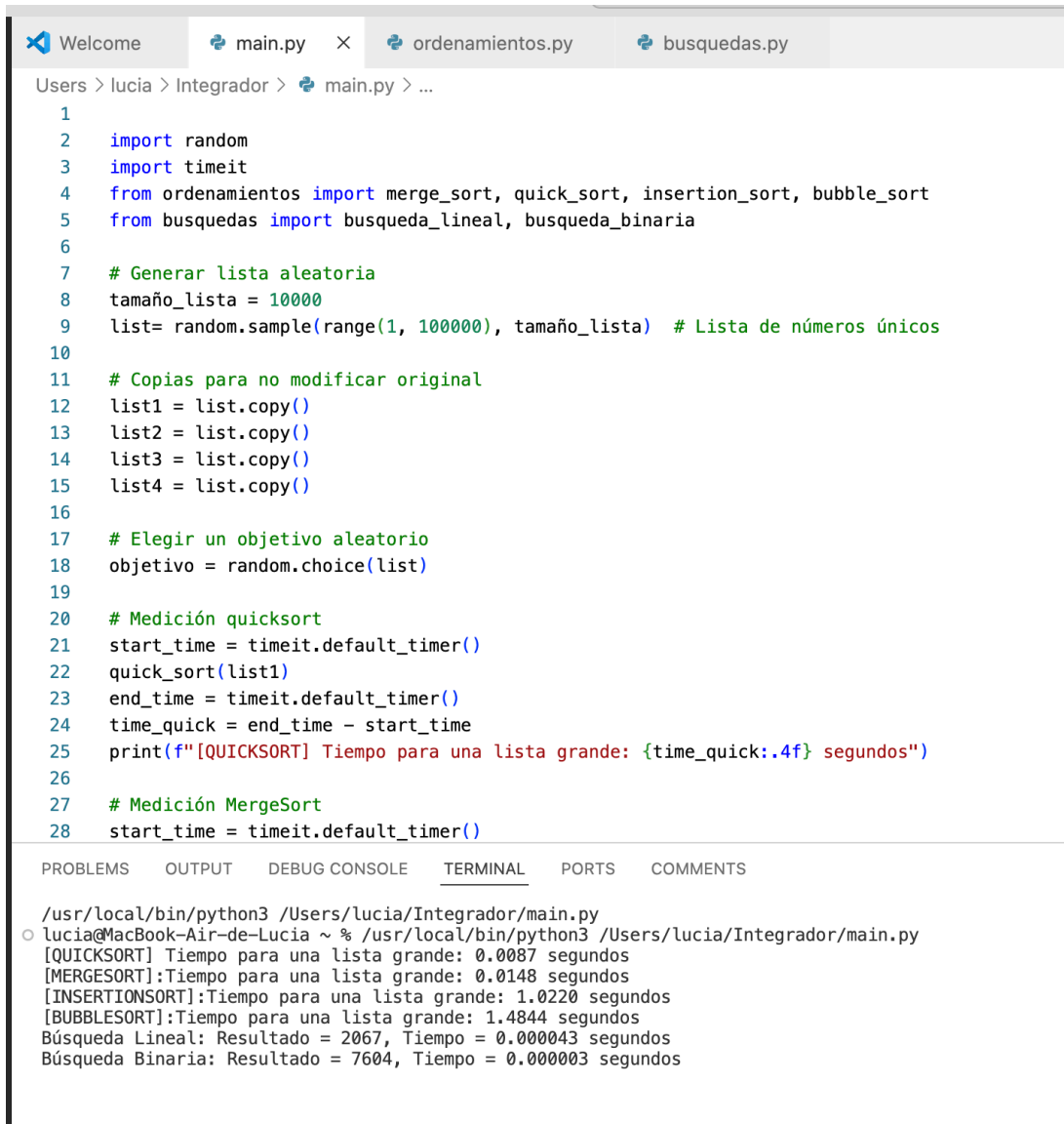
7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. <https://docs.python.org/3/>
- Cormen, T. H. et al. (2009). Introduction to Algorithms. MIT Press.
- Sweigart, A. (2019). Automate the Boring Stuff with Python. No Starch Press.

8. Anexos

- Capturas del programa ejecutándose.
- Enlace al repositorio Git:
- Enlace al video explicativo:
- Código completo disponible en la carpeta adjunta al trabajo.

Capturas de pantalla



The screenshot shows an IDE with four tabs: 'Welcome', 'main.py', 'ordenamientos.py', and 'busquedas.py'. The 'main.py' tab is active, displaying a Python script. The script generates a random list of 10,000 unique numbers, creates four copies, and then measures the execution time for quicksort, mergesort, insertion sort, and bubble sort. It also performs a linear search and a binary search on the original list. The terminal output at the bottom shows the execution results for each algorithm and search method.

```
1
2 import random
3 import timeit
4 from ordenamientos import merge_sort, quick_sort, insertion_sort, bubble_sort
5 from busquedas import busqueda_lineal, busqueda_binaria
6
7 # Generar lista aleatoria
8 tamaño_lista = 10000
9 list= random.sample(range(1, 100000), tamaño_lista) # Lista de números únicos
10
11 # Copias para no modificar original
12 list1 = list.copy()
13 list2 = list.copy()
14 list3 = list.copy()
15 list4 = list.copy()
16
17 # Elegir un objetivo aleatorio
18 objetivo = random.choice(list)
19
20 # Medición quicksort
21 start_time = timeit.default_timer()
22 quick_sort(list1)
23 end_time = timeit.default_timer()
24 time_quick = end_time - start_time
25 print(f"[QUICKSORT] Tiempo para una lista grande: {time_quick:.4f} segundos")
26
27 # Medición MergeSort
28 start_time = timeit.default_timer()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
/usr/local/bin/python3 /Users/lucia/Integrador/main.py
○ lucia@MacBook-Air-de-Lucia ~ % /usr/local/bin/python3 /Users/lucia/Integrador/main.py
[QUICKSORT] Tiempo para una lista grande: 0.0087 segundos
[MERGESORT]:Tiempo para una lista grande: 0.0148 segundos
[INSERTIONSORT]:Tiempo para una lista grande: 1.0220 segundos
[BUBBLESORT]:Tiempo para una lista grande: 1.4844 segundos
Búsqueda Lineal: Resultado = 2067, Tiempo = 0.000043 segundos
Búsqueda Binaria: Resultado = 7604, Tiempo = 0.000003 segundos
```

