Morales, Moreira Yanz, Zampedri
AyED 1C2025

Facultad de Ingeniería UNER

Trabajo Práctico N°1

Informe Problema 1

Implementamos los algoritmos de ordenamiento Burbuja/Bubble Sort, QuickSort y RadixSort dentro de la carpeta módulos.

Generación y Validación de Datos

Para asegurarnos de que todos los algoritmos funcionaran correctamente, generamos listas de 500 números aleatorios de cinco dígitos (usando la función randint() del módulo random). Comparamos los resultados de nuestros algoritmos con la función sorted() de Python, que sabemos que ordena bien, donde si los resultados coincidían, significaba que el algoritmo funcionaba correctamente.
También escribimos pruebas unitarias usando unittest (ver carpeta tests) así, si hacíamos cambios en el código, podíamos volver a correr los tests y verificar que todo siga funcionando bien.

Medición y comparación de tiempos

Una vez que tuvimos todo funcionando, medimos cuánto tardaba cada algoritmo en ordenar listas de distintos tamaños. Para eso usamos la función time.perf_counter(), que nos permite medir el tiempo exacto que tarda cada ejecución. Probamos con listas de tamaño 1, 10, 100, 200, 500, 700 y 1000. Guardamos esos tiempos en listas para después graficarlos; así pudimos ver cómo se comportan los algoritmos a medida que aumenta el tamaño de la lista.

Tamaño de la lista	Tiempo de ejecución (bubble sort)	Tiempo de ejecución (quick sort)	Tiempo de ejecución (radix sort)
1	0.000003	0.000004	0.000026
10	0.000013	0.000026	0.000037
100	0.000921	0.000138	0.000075
200	0.002418	0.000248	0.000182
500	0.014152	0.000857	0.000321
700	0.033929	0.001093	0.000587
1000	0.067402	0.001797	0.000602

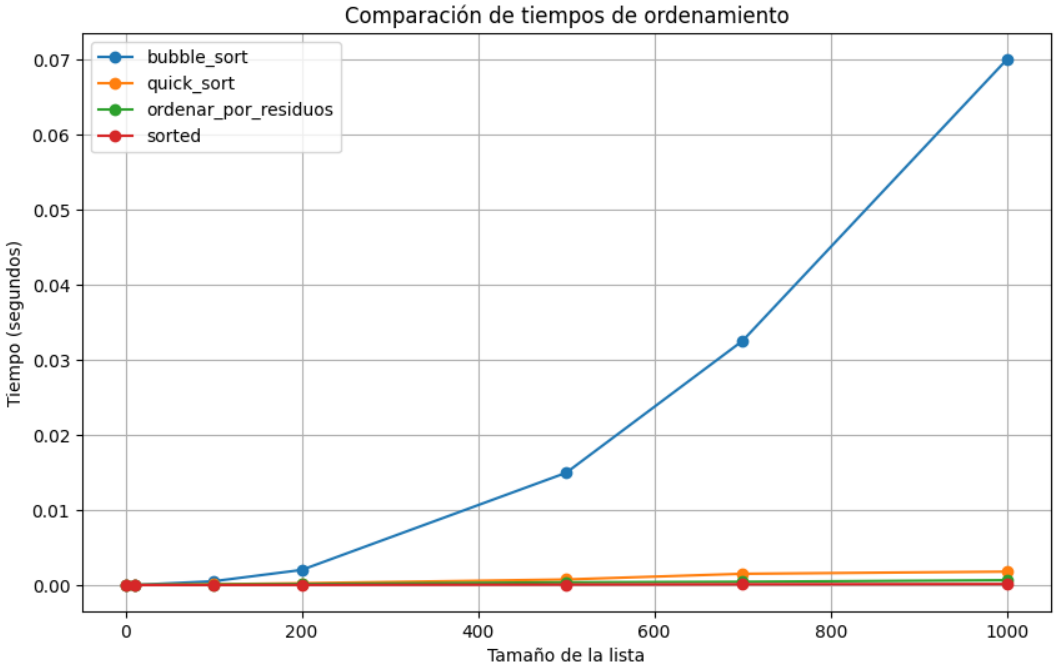
Notamos diferencias muy marcadas entre los algoritmos:

- **Bubble Sort:** Claramente el más lento. A medida que el tamaño de la lista aumentaba, el tiempo crecía de forma muy rápida. Esto era esperable porque su complejidad es **O(n²)**. En listas pequeñas funcionaba aceptablemente, pero con listas de más de 500 elementos ya era notoriamente ineficiente.

- **Quicksort**: Mucho mejor. Los tiempos crecían de forma “más suave”, hasta con listas grandes. Como tiene una complejidad promedio de **O(n log n)**, esperábamos este rendimiento. Fue mucho más rápido que bubble sort en casi todos los casos.
- **Radix Sort**: También tuvo un buen rendimiento, fue el más rápido en algunos casos, especialmente con listas más grandes. Al no hacer comparaciones y estar orientado a enteros, tuvo muy buenos tiempos. Su complejidad es lineal en la mayoría de los casos: **O(n·k)**, donde *k* es la cantidad de dígitos.

Gráfica de tiempos

Graficamos los tiempos usando matplotlib. Pusimos los tamaños de las listas en el eje X y los tiempos que tardó cada algoritmo en el eje Y. Para cada algoritmo usamos un color distinto y una etiqueta, y mostramos todo en una misma figura para poder comparar fácilmente su rendimiento. Gracias a estas gráficas pudimos ver de forma clara cuáles algoritmos escalan mejor con listas más grandes y cuáles se vuelven lentos rápidamente.



Por otro lado, comparamos nuestros algoritmos con la función built-in sorted() de Python. Investigamos y encontramos que internamente utiliza un algoritmo llamado Timsort, que es una mezcla de merge sort y insertion sort, es un algoritmo optimizado para datos reales y tiene una complejidad de O(n log n) en el peor caso, pero aprovecha si la lista ya tiene partes ordenadas. Con lo que, al compararlo con los otros métodos, sorted() fue claramente el más eficiente y confiable.