



Informe del Trabajo Práctico N°1

Aplicaciones de Tipos Abstractos de Datos (TADs)

Materia: Algoritmos y Estructuras de Datos - FIUNER

Integrantes:

Flores, Valentina

Aguilar Gonzales, Andrea Fernanda

Lell, Camila Luciana

Fecha de entrega: 2 de mayo del 2025

ACTIVIDAD 1

Planteamiento inicial

Para desarrollar este trabajo práctico, decidimos organizar el proyecto en diferentes módulos de Python, separando la lógica de los algoritmos de ordenamiento de la medición de tiempos y de la generación de gráficas. Esto nos permitió trabajar de manera modular, ordenada y reutilizable. Cada algoritmo fue implementado en su propia función dentro de un archivo llamado modules, mientras que los tiempos de ejecución y su grafica fueron medidos desde un archivo main.

También nos sirvió para probar los algoritmos por separado y entender bien cómo funcionaban. La separación en módulos es algo que aprendimos que ayuda un montón en proyectos más grandes o cuando se trabaja en grupo.

Algoritmos Implementados

Bubble Sort: Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Se repite el proceso hasta que la lista esté ordenada.

- **Código:** Se recorre la lista varias veces y se usa un indicador para cortar si ya está ordenada. Es fácil de entender pero no muy eficiente.
- **Complejidad:**
 - Mejor caso: $O(n)$
 - Peor y promedio: $O(n^2)$
- Cuando la lista está desordenada, compara todos los elementos muchas veces, lo que hace que tarde mucho más.

Quicksort: Algoritmo de división y conquista que selecciona un pivote, particiona la lista en dos sublistas y ordena recursivamente.

- **Código:** Se elige un pivote (en nuestro caso usamos el del medio), se separan los elementos y se usa recursión para ordenar cada grupo.
- **Complejidad:**
 - Mejor y promedio: $O(n \log n)$
 - Peor caso: $O(n^2)$
- Si el pivote no divide bien la lista, se vuelve más lento, pero normalmente anda muy bien.

Radix Sort: Algoritmo que ordena los elementos procesando dígito a dígito, usando un algoritmo estable como conteo para cada dígito.

- **Código:** Detectamos cuántos dígitos tiene el número más grande, y en cada pasada se agrupan los números por el dígito que toque, usando "baldes" o listas.
- **Complejidad:**
 - General: $O(k \cdot n)$, donde k es la cantidad de dígitos del número más largo.
- Es muy rápido cuando los números tienen una cantidad fija de dígitos.

Comparación con sorted()

El método `sorted()` de Python está super optimizado. Usa un algoritmo llamado Timsort, que mezcla ideas de Merge Sort e Insertion Sort. Es muy eficiente sobre todo si la lista ya está parcialmente ordenada.

- **Complejidad:**
 - Mejor caso: $O(n)$
 - Promedio y peor caso: $O(n \log n)$
- `sorted()` es bastante más rápido que nuestros algoritmos, pero igual sirve para comparar.

Análisis a priori de la complejidad

El análisis a priori se refiere a predecir el comportamiento teórico de los algoritmos sin necesidad de ejecutarlos. Para eso, nos basamos en cómo están estructurados los algoritmos y cuántas operaciones hacen dependiendo del tamaño de la lista.

Por ejemplo:

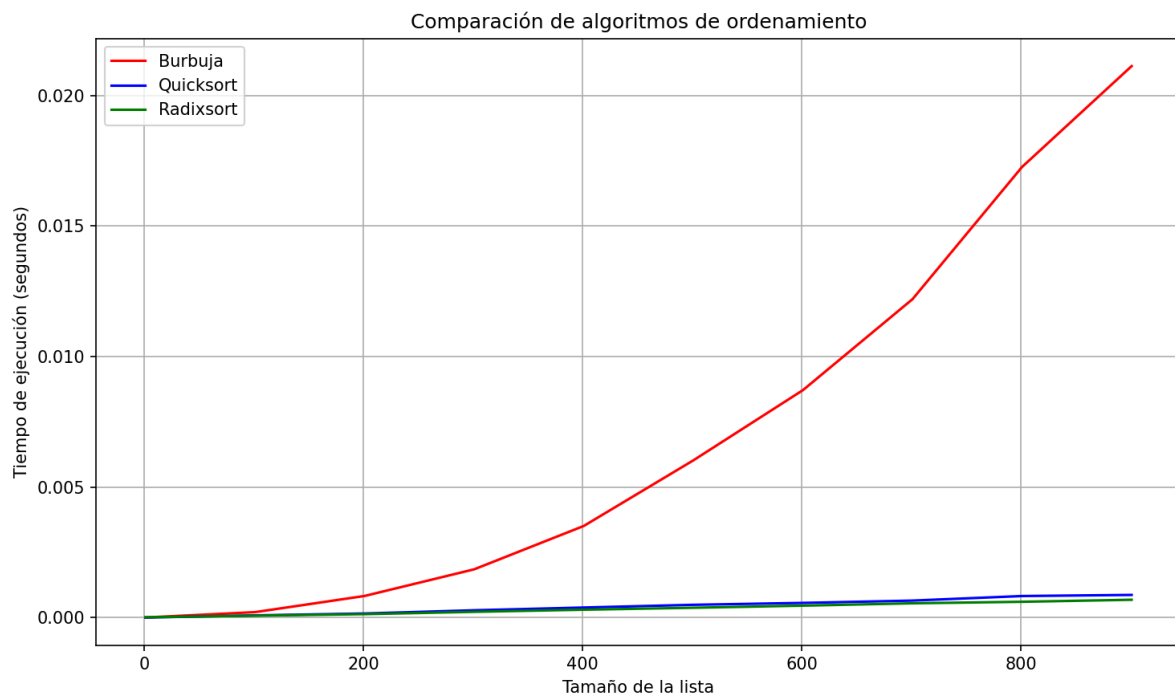
- En Bubble Sort, hay dos bucles anidados que recorren la lista, por eso el tiempo crece cuadráticamente ($O(n^2)$) en la mayoría de los casos.
- En Quicksort, al dividir la lista en partes más pequeñas y ordenar cada una, se obtiene en promedio un comportamiento de $O(n \log n)$. Esto depende de cómo se elige el pivote.
- En Radix Sort, como se recorre la lista una vez por cada dígito del número más grande, se dice que su complejidad es $O(k \cdot n)$, donde k es una constante si los números son de largo fijo.

Este análisis nos ayuda a saber qué algoritmo conviene usar según el tipo de datos que tengamos, incluso antes de probarlos con ejemplos concretos.

Resultados Experimentales

Se realizaron mediciones de tiempo para listas aleatorias de 5 dígitos con tamaños desde 1 hasta 1000 elementos. Los resultados se graficaron, mostrando el comportamiento de cada algoritmo.

Gráfica de tiempos:



Observaciones:

- Bubble Sort: el tiempo de ejecución crece cuadráticamente, siendo el más ineficiente.
- Quicksort: muestra muy buen rendimiento promedio.
- Radix Sort: mantiene un tiempo de ejecución bajo y bastante lineal.
- sorted(): resulta el más rápido en la mayoría de los casos.

Conclusiones

- Bubble Sort sirve para entender cómo funciona un algoritmo de ordenamiento, pero no es bueno para listas grandes.
- Quicksort es una gran opción en general, aunque hay que elegir bien el pivote.

- Radix Sort es ideal cuando se trata de números con la misma cantidad de dígitos.
- La función `sorted()` de Python es muy eficiente y está pensada para cubrir la mayoría de los casos con buen rendimiento.

En general, esta actividad nos ayudó a entender mejor cómo funcionan los algoritmos de ordenamiento tanto teóricamente como en la práctica. Además, practicar con mediciones reales nos permitió ver con claridad las diferencias de rendimiento entre ellos.