Trabajo Práctico 1

Algoritmos y Estructuras de datos

Integrantes: Cavallotti Guadalupe, Brochero Máximo, Navarrete Caterina

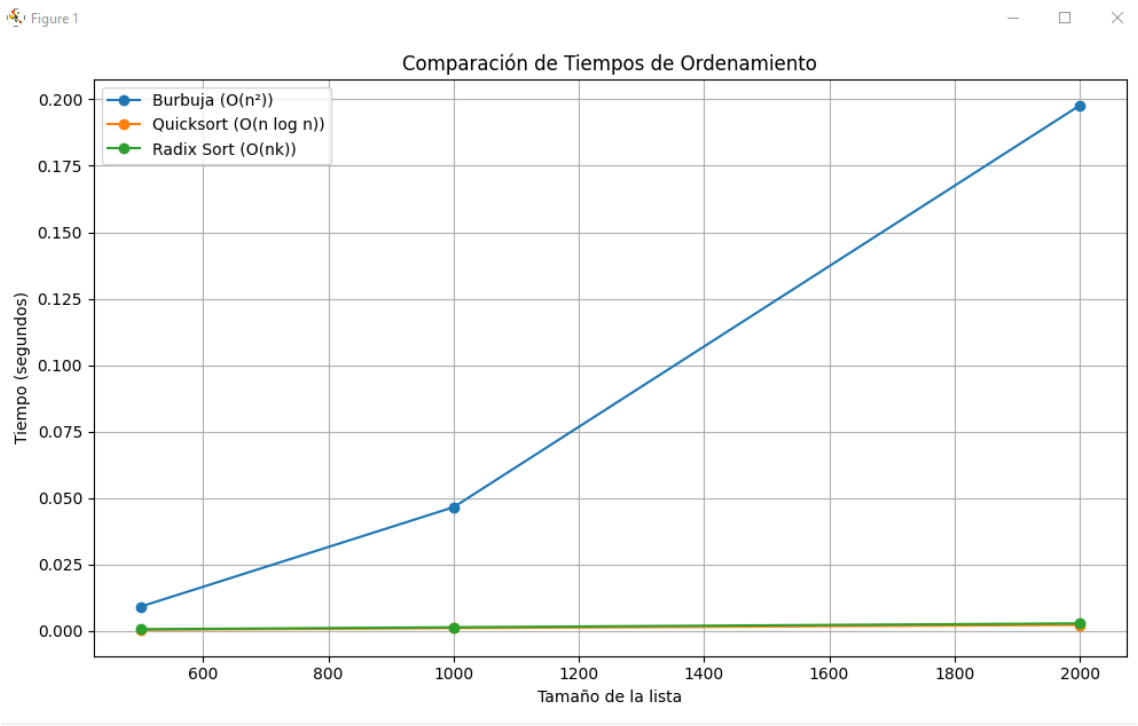
Problema 1: Análisis de Complejidad y Resultados de los Algoritmos de Ordenamiento

En esta parte del trabajo, implementamos los algoritmos de ordenamiento Burbuja, Quicksort y Radix Sort, y luego medimos cuánto tardaban en ordenar listas de distintos tamaños. La idea fue ver en la práctica cómo se comparan entre ellos y si los resultados coincidían con lo que habíamos estudiado sobre su complejidad.

Análisis Teórico de la Complejidad:

- **Ordenamiento Burbuja:**
    - Lógica simple: se compara cada elemento con el de al lado y se intercambian si están al revés.
    - Esto se repite varias veces hasta que todo queda ordenado.
    - El problema es que, en el peor de los casos, hay que hacer muchísimas comparaciones e intercambios.
    - En concreto, el número de operaciones crece como el cuadrado del número de elementos (n).
    - Por eso decimos que tiene una complejidad de O(n)^2.
- **Quicksort:**
    - Elige un elemento (pivote) y organiza la lista para que los menores queden a un lado y los mayores al otro.
    - Luego, aplica el mismo proceso a cada lado.
    - La clave está en cómo se elige el pivote. Si se elige bien, el algoritmo es muy rápido.
    - En el mejor y caso promedio, su complejidad es O(n log n).
- **Radix Sort:**
    - Radix Sort es diferente. No compara directamente los elementos.
    - Ordena los números dígito por dígito.
    - Es muy eficiente para enteros con una cantidad de dígitos más o menos fija.
    - Su complejidad es O(nk), donde k es la cantidad de dígitos.
    - Si k es pequeño, Radix Sort puede ser casi lineal (O(n)).

Resultados Experimentales (Análisis de la Gráfica):



- **Diferencia:** La gráfica "Comparación de Tiempos de Ordenamiento" muestra una diferencia importante entre los algoritmos.
- **El ordenamiento Burbuja se dispara:** La línea del ordenamiento Burbuja (azul) se va para arriba muy rápido. Se nota clarísimo el O(n)^2. A partir de los 1000 elementos, el tiempo de ejecución se dispara.
- **Quicksort y Radix Sort, mucho mejor:** Las líneas de Quicksort (naranja) y Radix Sort (verde) están casi pegadas al eje x. Esto significa que tardan mucho menos en ordenar, incluso con listas grandes.
- **Radix Sort es constante (casi):** La línea de Radix Sort es casi horizontal, lo que sugiere que su tiempo de ejecución no se ve tan afectado por el tamaño de la lista, al menos en este rango. Esto coincide con su complejidad de O(nk) y que k sea pequeño.
- **Confirmación de la teoría:** En general, los resultados de la gráfica confirman lo que esperábamos de la teoría. Los algoritmos con mejor complejidad (Quicksort y Radix Sort) son mucho más rápidos en la práctica, especialmente para listas grandes.

Comparación con sorted() de Python:

- Investigando un poco, descubrimos que sorted() usa un algoritmo llamado Timsort.
- Timsort es una mezcla de Merge Sort e Insertion Sort, optimizado para datos del mundo real.
- Es muy eficiente (O(n log n) en el peor caso) y se adapta bien a diferentes tipos de listas.
- Probablemente, por eso es tan rápido y en general, es la mejor opción para ordenar en Python.

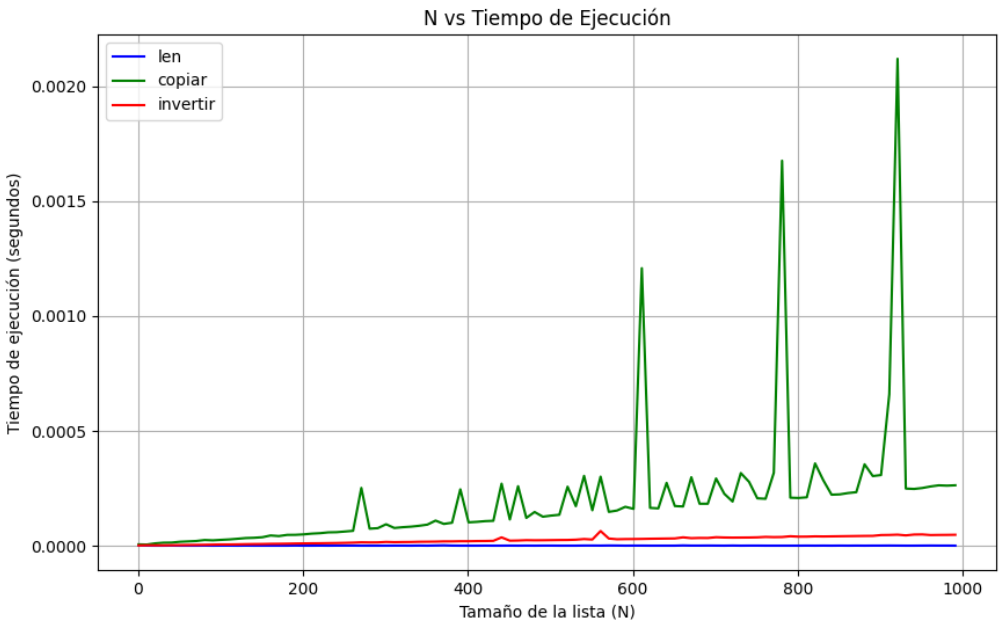
Conclusiones:

- La complejidad de un algoritmo realmente importa, especialmente para grandes cantidades de datos.
- Los resultados experimentales tienden a confirmar el análisis teórico, aunque puede haber pequeñas variaciones por la implementación y el hardware.
- Es importante elegir el algoritmo adecuado para cada situación. A veces, un algoritmo simple (como el de la burbuja) puede ser suficiente para listas pequeñas, pero para listas grandes, los algoritmos más eficientes (como Quicksort o Radix Sort) son cruciales.
- Python ya nos da herramientas muy buenas (como sorted()), pero es fundamental entender cómo funcionan los algoritmos por dentro.

Problema 2: Implementación y Análisis del TAD Lista Doblemente Enlazada

En este ejercicio, implementamos el Tipo Abstracto de Datos (TAD) Lista Doblemente Enlazada. Este TAD nos permite almacenar una secuencia de elementos, donde cada elemento tiene una referencia tanto al elemento anterior como al siguiente. Esto facilita la navegación en ambas direcciones de la lista.

Gráfica experimental:



Análisis de Complejidad y Resultados (Basado en la Gráfica):

Para evaluar la eficiencia de nuestra implementación, medimos el tiempo de ejecución de las operaciones __len__, copiar e invertir para listas de diferentes tamaños.

- **__len__:**
    - Esta operación simplemente devuelve el valor del atributo tamanio.
    - Como se observa en la gráfica (línea azul), el tiempo de ejecución de __len__ se mantiene prácticamente constante, independientemente del tamaño de la lista.
    - Esto confirma que su complejidad es **O(1)** (tiempo constante).
- **copiar:**
    - La operación copiar crea una nueva lista y recorre la lista original para agregar cada elemento a la nueva lista.
    - La gráfica (línea verde) muestra una tendencia lineal, aunque con algunas fluctuaciones.
    - Esto indica que el tiempo de ejecución aumenta proporcionalmente al tamaño de la lista, lo que concuerda con la complejidad **O(n)** que esperábamos.
    - Las fluctuaciones podrían deberse a factores como la sobrecarga del sistema o la asignación de memoria.
- **invertir:**
    - La operación invertir modifica los punteros anterior y siguiente de cada nodo para invertir el orden de la lista.
    - Al igual que copiar, la gráfica (línea roja) muestra una tendencia lineal, aunque con un tiempo de ejecución significativamente menor.
    - Esto también confirma la complejidad **O(n)**, ya que se recorre la lista una vez.
    - La menor duración se explica porque solo se modifican punteros, sin crear nuevos nodos como en copiar.

Conclusiones:

- Los resultados experimentales obtenidos a través de las mediciones de tiempo y la visualización en la gráfica respaldan el análisis teórico de la complejidad de las operaciones implementadas.
- La operación __len__ presenta un rendimiento óptimo con una complejidad constante, mientras que copiar e invertir muestran un comportamiento lineal, como se esperaba.
- Es importante destacar que, si bien la complejidad asintótica nos da una idea general del rendimiento, factores como la implementación específica, el hardware y las fluctuaciones del sistema pueden influir en los tiempos de ejecución reales