Nombre del Proyecto:

"Diseño y Evaluación de Algoritmos y TADs".

Descripción:

En el trabajo práctico implementamos y analizamos tres algoritmos de ordenamiento (burbuja, Quicksort y radix sort), desarrollamos una estructura de datos personalizada (Lista doblemente enlazada) y simulamos el juego de cartas “Guerra” utilizando la estructura creada anteriormente.

El objetivo del trabajo es comparar los tiempos de ejecución y complejidad entre los diferentes algoritmos, así como utilizar estructuras dinámicas en contextos prácticos, en este caso un juego de cartas.

Estructura general del código:

El proyecto está dividido en tres carpetas principales: actividad_1, actividad_2 y actividad_3.

- actividad_1/: contiene implementaciones de los algoritmos de ordenamiento, scripts de medición de tiempos y generación de gráficas.
- actividad_2/: implementa la clase ListaDobleEnlazada con los métodos especificados en el enunciado y scripts de prueba y análisis de rendimiento.
- actividad_3/: contiene la implementación del juego “Guerra” basado en una clase Mazo, que utiliza la ListaDobleEnlazada. Incluye las pruebas unitarias necesarias.

Las gráficas de los resultados están disponibles en la carpeta data/.

El informe completo está disponible en la carpeta docs/.

Dependencias:

- Python 3.x
- matplotlib
- random
- time
- pytest (opcional para pruebas unitarias)
- Todas las dependencias están listadas en “deps/requirements.txt”.

Como ejecutar el proyecto:

1. Clonar o descargar este repositorio.

2. Crear y activar un entorno virtual:

    “python -m venv venv”
    “source venv/bin/activate # o .\venv\Scripts\activate en Windows”

3. Instalar las dependencias:
    “pip install -r deps/requirements.txt”

4. Ejecutar los scripts correspondientes en cada carpeta (problema1, problema2, problema3).

Conclusiones:

- Problema 1 – Ordenamiento

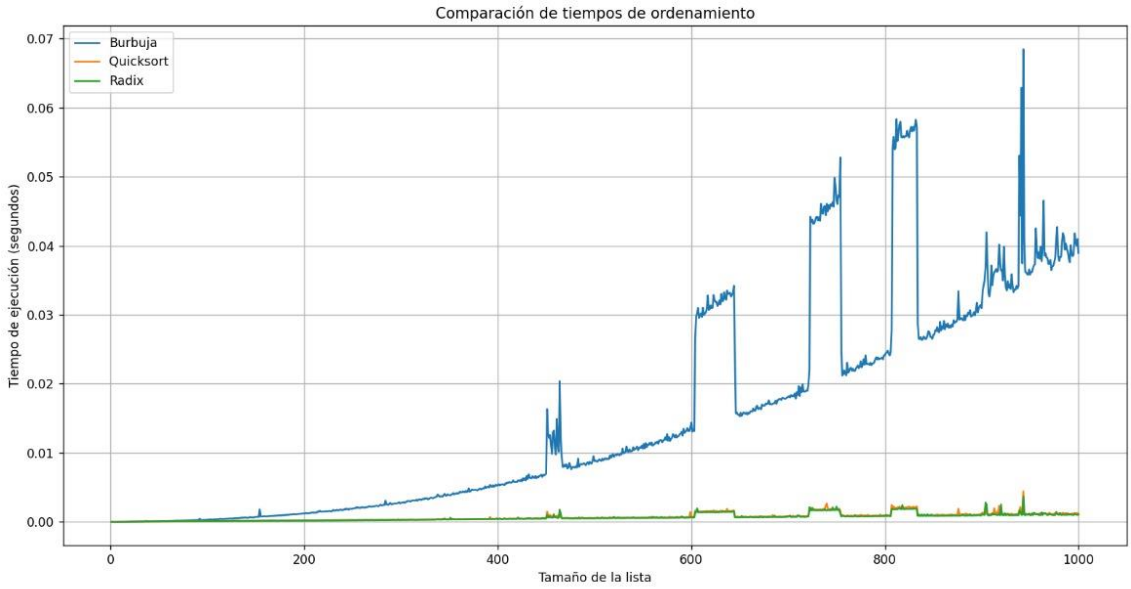
Complejidad esperada de cada algoritmo:

Bubble Sort: En el bubble sort se espera un orden de complejidad O(n) en el mejor de los casos, que sería cuando la lista esté completamente ordenada, y en el peor de los casos un orden de complejidad O(n^2) que se daría si la lista está completamente invertida.

Quicksort: En el mejor caso de un ordenamiento quicksort es O(nlogn) y se daría si el pivote divide el arreglo en dos mitades casi iguales, y en el peor de los casos O(n^2) que se da cuando el pivote es siempre el mínimo o máximo valor.

Radix Sort: Y en el caso del radix sort su orden de complejidad va a ser siempre O(n-k), ya que este no depende del orden original de los datos, sino de la cantidad de dígitos k de los números.

Análisis a partir de las gráficas de tiempos de ejecución:



Conclusiones sobre la eficiencia de los algoritmos de ordenamiento:

Radix Sort fue el algoritmo más eficiente a lo largo de todo el rango de tamaños de listas evaluado (1 a 1000 elementos). Su tiempo de ejecución se mantuvo prácticamente constante, lo cual es coherente con su eficiencia de tipo O(nk), especialmente adecuada para listas de enteros.

Quicksort mostró un rendimiento muy bueno y estable, con tiempos de ejecución bajos y con poca variación, especialmente en comparación con Burbuja. Este comportamiento se ajusta a su complejidad promedio de O(nlogn), lo que lo convierte en una excelente opción general.

Burbuja, por otro lado, fue el algoritmo menos eficiente. Su tiempo de ejecución creció rápidamente a medida que aumentaba el tamaño de la lista, presentando además mucha variabilidad. Esto es característico de su complejidad O(n2), haciéndolo inadecuado para listas grandes.

¿Cómo funciona sorted en Python y qué algoritmo utiliza?

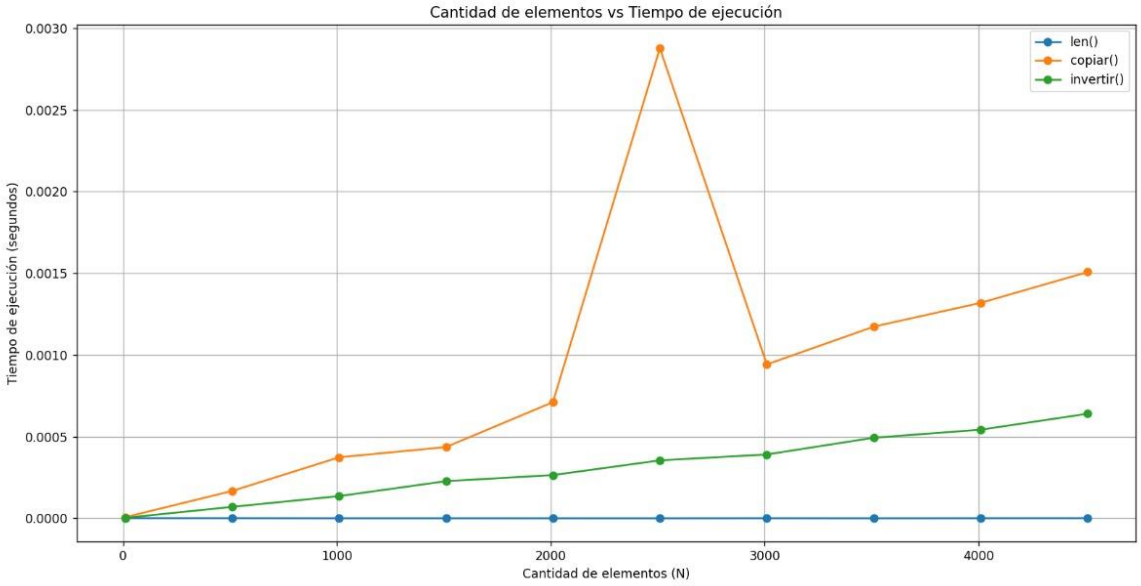
La función sorted() de Python utiliza internamente el algoritmo Timsort, una combinación optimizada de Merge Sort e Insertion Sort. Este algoritmo detecta automáticamente subsecuencias ya ordenadas (runs), las ordena con Insertion Sort si son pequeñas y luego las fusiona eficientemente con Merge Sort.

Timsort es un algoritmo estable, muy eficiente en la práctica, y su rendimiento es en el mejor caso: O(n) (si la lista ya está parcialmente ordenada); promedio y peor caso: O(n log n).

Es más rápido y confiable que algoritmos como Bubble Sort o incluso Quicksort en ciertos casos, y por eso se usa como estándar en Python.

- Problema 2 – ListaDobleEnlazada

Análisis de eficiencia de los métodos len, copiar, invertir:



Análisis de eficiencia de los métodos len, copiar, invertir:

Len(): En el método len lo que se esperaba era una eficiencia de O(1) (tiempo constante) y el comportamiento observado fueron tiempos de ejecución fueron extremadamente bajos y prácticamente constantes, independientemente del tamaño de la lista. El método simplemente retorna el valor del atributo self.tamanio, que se actualiza con cada inserción o eliminación. No se recorre la lista. En conclusión es muy eficiente y adecuado para las listas grandes.

Copiar(): La eficiencia esperada era de O(n) (tiempo lineal con respecto al tamaño de la lista) y el comportamiento observado de ejecución creció proporcionalmente al número de elementos. Se recorre nodo por nodo de la lista original, creando nuevos nodos y enlazándolos para formar una copia independiente. En conclusión es un comportamiento

esperado, es razonablemente eficiente, aunque se vuelve más costoso para listas grandes, por esa razón observamos un "pico" en la gráfica. Para listas con valores cercanos a 2500 elementos, el método toma mas tiempo en ejecutarse por lo que es uno de los peores casos.

`Invertir():` La eficiencia esperada era de O(n) (tiempo lineal) y el comportamiento observado de ejecución también lineal, similar al de `copiar()`. Se recorren todos los nodos para intercambiar sus punteros `siguiente` y `anterior`. La operación no crea nuevos nodos, solo reorganiza enlaces. En conclusión es muy eficiente para inversión in-place. Adecuada para grandes volúmenes de datos.

Conclusión sobre diseño de la estructura:

La estructura `ListaDobleEnlazada` presenta un diseño sólido y funcional, con claras ventajas como la capacidad de recorrer la lista en ambos sentidos gracias a sus referencias dobles, y una alta eficiencia en las operaciones de inserción y extracción tanto al inicio como al final de la lista, las cuales se realizan en tiempo constante. La implementación es modular, con métodos bien definidos y reutilización efectiva de nodos, lo que contribuye a su rendimiento general. Sin embargo, podrían introducirse mejoras que harían la estructura más completa, como un manejo más detallado de errores en métodos como `insertar` y `extraer`, y la inclusión de un iterador inverso (`__reversed__`). También sería útil incorporar una función para eliminar nodos por valor y agregar documentación breve en los métodos para facilitar la comprensión y el mantenimiento del código.

- Problema 3 – Juego Guerra

Evaluación del uso del TAD ListaDobleEnlazada en la clase Mazo:

La clase Mazo implementa todos los métodos necesarios para interactuar correctamente con el código provisto por la cátedra, respetando las firmas de los métodos y asegurando compatibilidad con los tests automáticos.

Para validar la implementación, se ejecutaron los tests provistos:

Test de la clase Mazo → todos los casos pasaron correctamente, confirmando que las operaciones sobre el mazo funcionan como se espera.

Test del juego Guerra → todos los casos pasaron correctamente, confirmando que la lógica del juego, usando nuestra implementación del mazo, es robusta y cumple con los requisitos.

Como resultado, se concluye que la solución es correcta.

Autores:

- Carolina Belén Pérez
- Chiara Groglio Kremer
- Ignacio Dechiara