Informe Trabajo Práctico N°1

Problema 1: Lista doblemente Enlazada

Enunciado:

El objetivo es implementar un **TAD Lista doblemente enlazada** capaz de almacenar elementos comparables (enteros, flotantes, strings). Realizar gráficas de N (cantidad de elementos) vs tiempo de ejecución para ciertos métodos.
Debe cumplir con los siguientes métodos:

- esta_vacia()
- agregar_al_inicio(dato)
- agregar_al_final(dato):
- insertar(item, posición)
- extraer(posición)
- copiar()
- invertir()
- __len__()
- __add__(Lista)
- __iter__()

Solución:

Para la creación de la Lista Doblemente Enlazada se utilizaron distintos componentes que trabajan de manera conjunta para dar forma a la estructura:

Nodo: definido como una clase interna, representa la unidad básica de la lista. Cada nodo cuenta con tres atributos principales:

- dato: almacena el valor que contendrá el nodo.
- siguiente: referencia al nodo que continúa en la lista (inicialmente vacío).
- anterior: referencia al nodo previo dentro de la lista (inicialmente vacío).

ListaDoblementeEnlazada: constituye la estructura principal, encargada de gestionar y coordinar el funcionamiento de los nodos. A partir de esta clase es posible realizar las operaciones antes mencionadas.
Cuenta con los siguientes atributos:

- cabeza: indica el primer nodo de la lista
- cola: indica el último nodo de la lista.
- tamaño: indica el tamaño actual de la lista.

Resultados:

Al ejecutar los tests propuestos por la cátedra, se detectaron algunos errores en ciertas operaciones. Estos errores fueron corregidos, permitiendo que los tests se ejecutaran correctamente y comprobando que todas las operaciones funcionarán según lo esperado.

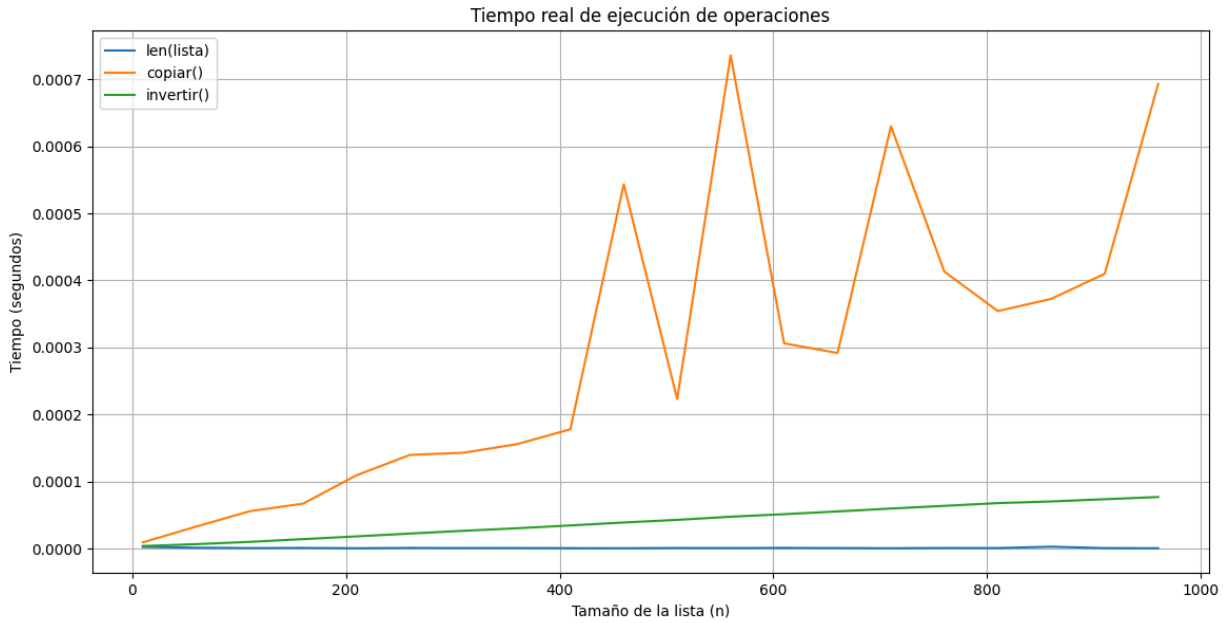
Análisis de complejidad:

Se midió el tiempo que tarda un método en ejecutarse en función del tamaño de la lista N (cantidad de elementos).
Utilizamos la notación Big-O, describe cómo crece el tiempo de ejecución de un algoritmo cuando aumenta el tamaño de los datos N.
En nuestro caso realizamos el análisis sobre:
__len__ → O(1).
Retorna directamente el valor del atributo tamaño sin recorrer la lista, no importa la cantidad de elementos que tenga la lista, siempre tarda el mismo tiempo.
La gráfica tendría que ser una línea horizontal (constante).

copiar() → O(n).
Recorre todos los nodos uno por uno y crea un nuevo nodo para cada elemento agregándolo a una nueva lista.
Si la lista tiene N elementos, la operación se realiza N veces, el tiempo de ejecución crece proporcionalmente al tamaño de la lista (crecimiento lineal).

invertir() → O(n).
Se recorre la lista completamente y se intercambian los punteros (anterior y siguientes) de cada Nodo.
Se hace una operación por cada nodo, por eso también la gráfica tiene un crecimiento lineal.

Gráficas comparativas del orden de complejidad:



La línea azul len(lista) se mantiene constante a lo largo de todos los tamaños de lista, confirmando que la operación tiene complejidad O(1).
La línea naranja copiar() crece aproximadamente de manera lineal con N. Esto indica que cada elemento de la lista se recorre una vez para generar la copia, lo que confirma la complejidad O(n).
La línea verde invertir() también muestra crecimiento lineal con N, aunque con menor tiempo que copiar(). Esto se puede deber al hecho de que invertir solo intercambia punteros en cada nodo sin crear nodos nuevos, confirmando O(n) pero con menor costo de operación por nodo.
Las pequeñas variaciones o picos que se observan en las curvas se deben a la fluctuación normal de los tiempos de ejecución en Python y al comportamiento del sistema operativo, pero en general, la tendencia de las gráficas refleja correctamente el comportamiento esperado de cada operación.