Trabajo Práctico N°1

Problema 1

Implementar el **TAD Lista doblemente enlazada** (ListaDobleEnlazada) que permita almacenar elementos de cualquier tipo que sean comparables (por ejemplo enteros, flotantes, strings). La implementación debe respetar la siguiente **especificación lógica**:

- `esta_vacia()`
- `agregar_al_inicio(item)`
- `agregar_al_final(item)`
- `insertar(item, posicion)`
- `extraer(posicion)`
- `copiar()`
- `invertir()`
- `concatenar(Lista)`
- `__len__()`
- `__add__(Lista)`
- `__iter__()`

El inicializador __init__ debe crear una lista originalmente vacía.

Realizar una gráfica de N (cantidad de elementos) vs tiempo de ejecución para los siguientes métodos: len, copiar e invertir (verificar que los hayan implementado de la forma más eficiente posible). Explicar los resultados y deducir los órdenes de complejidad a partir de las gráficas.
Su clase ListaDobleEnlazada debe pasar el test provisto por la cátedra.

Solución

Para la implementación del TAD Lista Doblemente Enlazada se desarrollaron distintos componentes:
Nodo: clase independiente, es la unidad básica de la lista. Cada nodo tiene tres atributos principales:

- dato: almacena el valor que contendrá el nodo.
- siguiente: referencia al nodo siguiente de la lista (inicialmente *None (vacío)*).
- anterior: referencia al nodo previo dentro de la lista (inicialmente *None (vacío)*).

ListaDobleEnlazada: la estructura principal, encargada de administrar y coordinar el funcionamiento de los nodos. A partir de esta clase es posible realizar operaciones como insertar, extraer, copiar, invertir, concatenar y recorrer la lista.
Cuenta con los siguientes atributos:

- cabeza: referencia al primer nodo de la lista.
- cola: referencia al último nodo de la lista.
- tamaño: almacena la cantidad actual de elementos de la lista.

Análisis de órdenes de complejidad

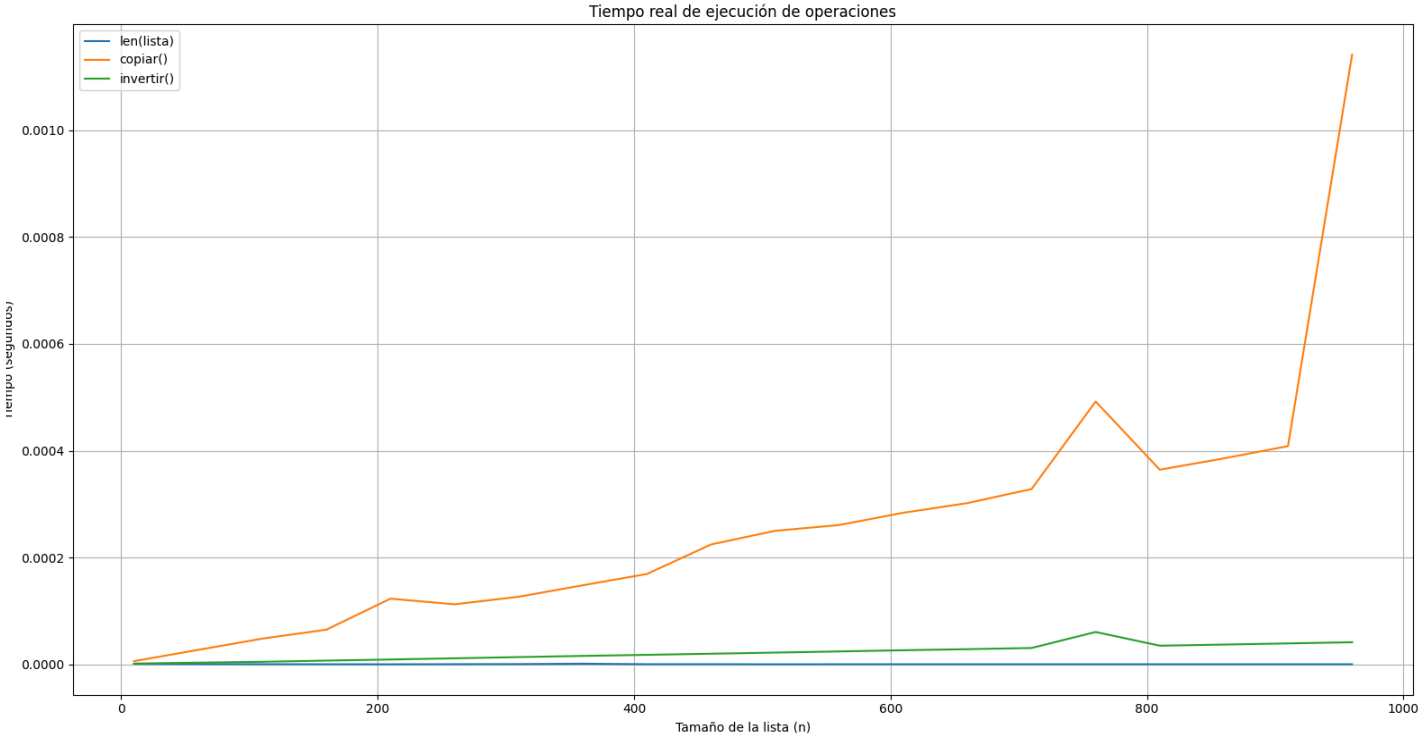
Medimos cuánto tardan algunos métodos de la Lista Doblemente Enlazada según el tamaño de la lista N. Para entender el crecimiento del tiempo usamos la notación Big-O, que nos dice cómo cambia el tiempo cuando la lista se hace más grande.

En nuestro caso vimos lo siguiente:

`len(lista)` → O(1)
Este método devuelve directamente el tamaño guardado en un atributo, sin recorrer la lista. Sin importar cuántos elementos tenga, siempre tarda lo mismo. En la gráfica se ve una línea horizontal azul. Solo devuelve un atributo.

`copiar()` → O(n)
Este método va nodo por nodo copiando los datos a una nueva lista. Si hay N elementos, se hacen N copias, así que el tiempo crece de forma lineal con el tamaño de la lista. (gráfica naranja)

`invertir()` → O(n) (gráfica verde)
También recorre todos los nodos, intercambiando punteros “anterior” y “siguiente” de cada uno, pero con menor constante que copiar, por eso crece más lento.



En la gráfica se observan algunos picos en los tiempos de copiar() e invertir(). Esto sucede porque ambos métodos recorren nodo por nodo la lista y, a medida que aumenta su tamaño, se utilizan más memoria y más ciclos de CPU. En ciertos puntos, Python realiza procesos internos como la recolección de caché o la reorganización de memoria, lo que genera demoras puntuales en la ejecución. Por eso aparecen esos saltos en la gráfica, aunque en términos generales el comportamiento siga siendo lineal y acorde a lo esperado.