TP N° 1: Problema 2.

Alumnas: Nuñez Juliana, Morello Milagros.

Facultad de Ingeniería.

Universidad Nacional de Entre Ríos.

Lic. en Bioinformática.

Algoritmos y Estructuras de datos.

Desarrollo:

En este documento se procederá a realizar una breve explicación y análisis de resultados del Problema 2, del trabajo práctico n°1.

Para esta parte del trabajo se nos pidió la implementación de un tipo de estructura de datos (TAD) de listas doblemente enlazadas (mediante una clase), las cuales debían contar con los siguientes métodos:

> **esta_vacia()**: Devuelve **True** si la lista está vacía.
> **__len__()**: Devuelve el número de ítems de la lista.
> **agregar_al_inicio(item)**: Agrega un nuevo ítem al inicio de la lista.
> **agregar_al_final(item)**: Agrega un nuevo ítem al final de la lista.
> **insertar(item, posicion)**: Agrega un nuevo ítem a la lista en "posicion". Si la posición no se pasa como argumento, el ítem debe añadirse al final de la lista. "posicion" es un entero que indica la posición en la lista donde se va a insertar el nuevo elemento. Si se quiere insertar en una posición inválida, que se arroje la debida excepción.
> **extraer(posicion)**: elimina y devuelve el ítem en "posición". Si no se indica el parámetro posición, se elimina y devuelve el último elemento de la lista. La complejidad de extraer elementos de los extremos de la lista debe ser O(1). Si se quiere extraer de una posición indebida, que se arroje la debida excepción.
> **copiar()**: Realiza una copia de la lista elemento a elemento y devuelve la copia. Verificar que el orden de complejidad de este método sea O(n) y no $O(n^2)$.
> **invertir()**: Invierte el orden de los elementos de la lista.
> **concatenar(Lista)**: Recibe una lista como argumento y retorna la lista actual con la lista pasada como parámetro concatenada al final de la primera.
> **__add__(Lista)**: El resultado de “sumar” dos listas debería ser una nueva lista con los elementos de la primera lista y los de la segunda. Aprovechar el método concatenar para evitar repetir código.

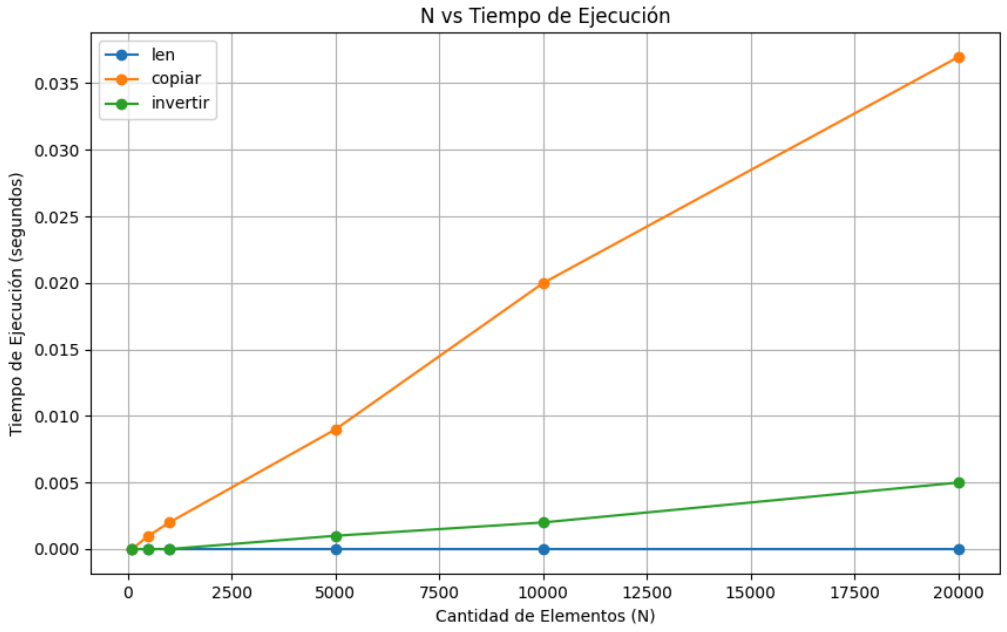
Una vez la misma fue implementada, debía adecase a los tests proporcionados por la cátedra, lo cual nos llevó varios problemas (principalmente con el método copiar(), a que él mismo nos devolvió un objeto none en lugar de no del tipo de dato que nosotras implementamos en la clase)

También se realizó una medición del tiempo de ejecución que tenían copiar(), __len__() e invertir() con el siguiente código (muy similar al problema 1):

```python
import time
from lista_doble_enlazada import ListaDobleEnlazada

def medir_tiempo(func, *args):
    inicio = time.time()
    func(*args)
    fin = time.time()
    return fin - inicio
```

```python
# Crear listas con diferentes tamaños
tamanios = [100, 500, 1000, 5000, 10000, 20000]
tiempos_len = []
tiempos_copiar = []
tiempos_invertir = []

for n in tamanios:
    lde = ListaDobleEnlazada()
    for i in range(n):
        lde.agregar_al_final(i)

    # Medir tiempo para len
    tiempo_len = medir_tiempo(len, lde)
    tiempos_len.append(tiempo_len)

    # Medir tiempo para copiar
    tiempo_copiar = medir_tiempo(lde.copiar)
    tiempos_copiar.append(tiempo_copiar)

    # Medir tiempo para invertir
    tiempo_invertir = medir_tiempo(lde.invertir)
    tiempos_invertir.append(tiempo_invertir)
```

Y con esos datos se realizó la siguiente gráfica (utilizando matplotlib):



Conclusiones y resultados:

A partir de esta gráfica se pidió inferir la eficiencia y el orden de complejidad de cada método, analizando el comportamiento de las curvas:

- **__len__():** su curva está muy cerca del eje x y es prácticamente horizontal. Esto demuestra que el tiempo de ejecución de esta operación no depende del número de elementos y es constante, osea, calcular la longitud de una lista, arreglo u otra estructura de datos toma el mismo tiempo independientemente de su tamaño. **Complejidad: O(1).** Dado que su tiempo de ejecución no aumenta significativamente con el aumento del tamaño de la entrada, las operaciones O(constante) son las más eficientes.
- **copiar():** muestra un aumento lineal. Copiar una estructura de datos requiere cada uno de sus componentes y el proceso es directamente proporcional al tamaño de la estructura, por lo que, el tiempo de ejecución aumenta proporcionalmente con el número de elementos. **Complejidad es O(n).** Las operaciones con orden de complejidad lineal son consideradas eficientes en muchos casos, pero, cuando se duplica el tamaño de la entrada, su tiempo de ejecución se duplica.
- **invertir():** La curva de invertir también muestra un crecimiento lineal, como la de copiar. Esto se debe a que, en general, invertir una estructura de datos también requiere examinar al menos una vez cada uno de sus componentes. **Complejidad es O(n).** Investir tiene un orden de complejidad lineal, al igual que la operación de copiar.