# ACTIVIDAD 1

## EXPLICACIÓN DE LA TAD UTILIZADA:

Se implementó el Tipo Abstracto de Datos (TAD) ListaDobleEnlazada utilizando una clase Nodo.

- La clase Nodo almacena el dato y dos referencias: anterior y siguiente.
- La clase ListaDobleEnlazada gestiona la estructura principal, manteniendo referencias a cabeza (el primer nodo), cola (el último nodo) y tamanio (la cantidad de elementos).

La implementación cumple con la especificación solicitada:

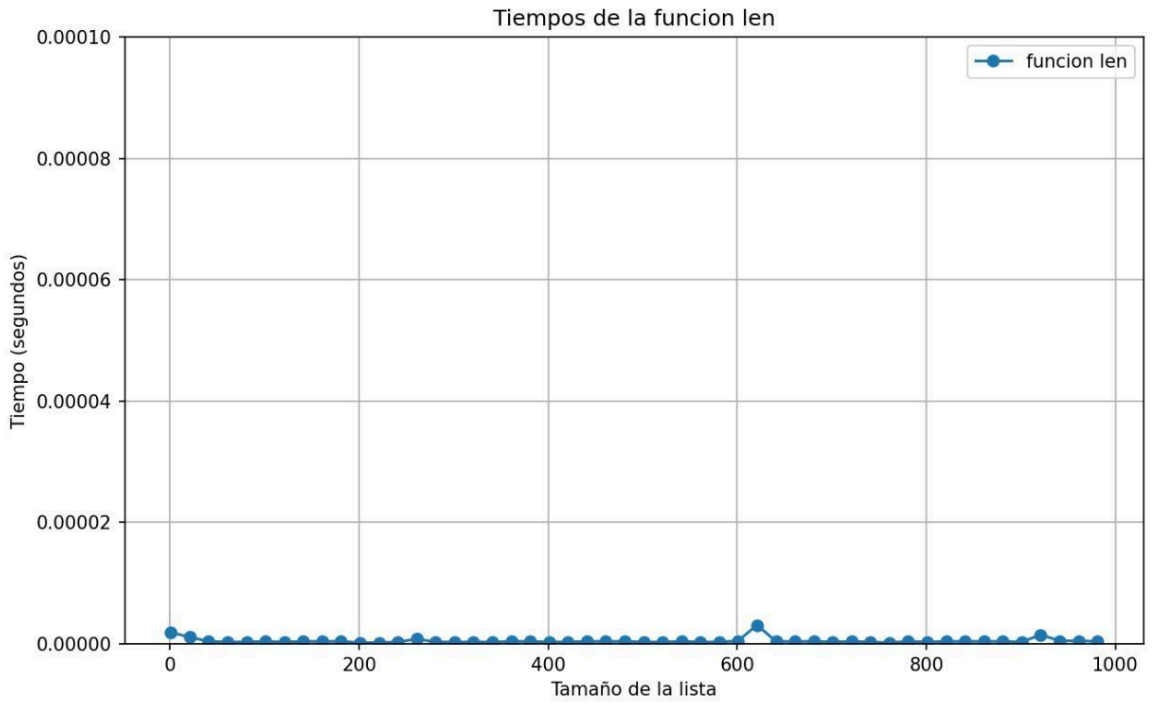
- **__init__ y esta_vacia**: Crean una lista vacía y verifican su estado
- **agregar_al_inicio y agregar_al_final**: Añaden elementos en los extremos de la lista actualizando cabeza y cola respectivamente
- **insertar(item, posicion)**: Permite agregar un elemento en cualquier punto. Si la posición no se especifica, el código le asigna al final (self.tamanio)
- **extraer(posicion)**: Elimina y devuelve un ítem. Si la posición es None, se asigna self.tamanio - 1 para extraer el último elemento. La implementación de los casos posicion == 0 (cabeza) y posición == self.tamanio - 1 (cola) accede directamente a los punteros cabeza y cola, sin bucles, lo que garantiza una complejidad O(1)
- **copiar()**: Crea y devuelve una nueva ListaDobleEnlazada con los mismos elementos
- **invertir()**: Modifica la lista actual invirtiendo el orden de sus elementos.
- **concatenar(lista)**: se usa para unir dos listas
- **__len__**: Devuelve el valor de self.tamanio.
- **__add__**: Devuelve una nueva lista que es el resultado de la concatenación, utilizando los métodos copiar y concatenar para evitar la duplicación de código
- **__iter__ y __next__**: Permiten que la lista sea recorrida con un ciclo for.

La implementación no utiliza listas de Python como estructura de almacenamiento principal, respetando la consigna propuesta por la cátedra.

## ANÁLISIS DE COMPLEJIDAD:

Mediante el uso de las gráficas (N vs tiempo) obtenidas vamos a realizar el análisis de complejidad para los métodos len, copiar e invertir:

*Gráfica función len*




Complejidad Deducida: O(1) (Tiempo constante).

La complejidad del método __len__ es O(1) porque la implementación del TAD ListaDobleEnlazada no calcula el tamaño de la lista al momento de la consulta, sino que mantiene un registro constante del mismo.

Esto se logra por el atributo interno, self.tamanio, que guarda el número de elementos de la lista en todo momento.

Inicialización: El contador self.tamanio se inicializa en 0 durante la creación de la lista (en el método (__init__)

Actualización Constante: Cada vez que un método de modificación altera la lista, el contador se actualiza.

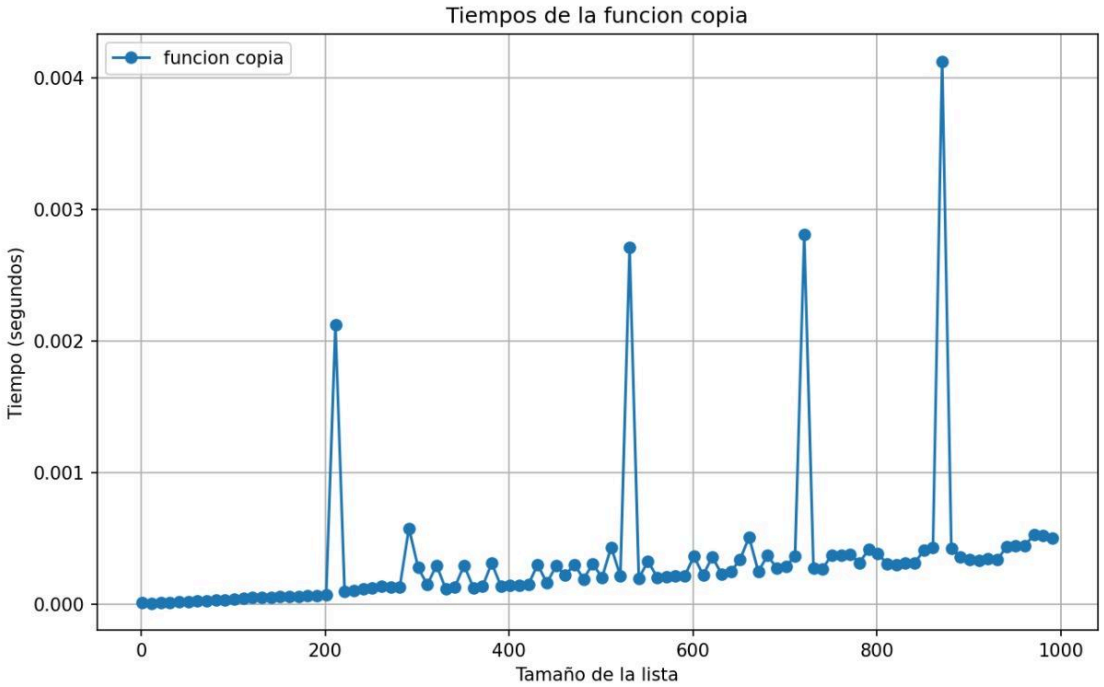
Las operaciones de inserción (agregar_al_inicio, agregar_al_final, insertar) incrementan el contador (self.tamanio += 1).

Las operaciones de extracción (extraer) decrementan el contador (self.tamanio -= 1).

Estas actualizaciones de incremento/decremento son operaciones de tiempo constante O(1)

Como resultado, cuando se invoca el método __len__ su única responsabilidad es retornar el valor ya almacenado en el atributo self.tamanio. Esta operación consiste en un único acceso a memoria, cuyo tiempo de ejecución es constante y totalmente independiente del número de elementos (N) que contenga la lista.

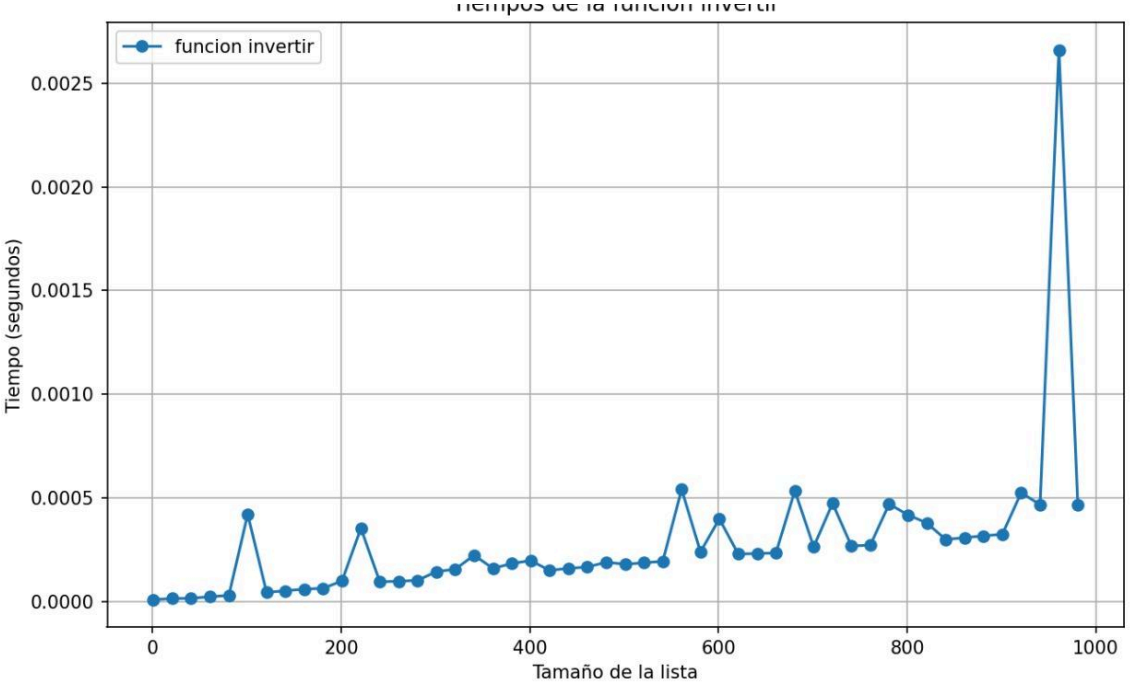
*Gráfica función copia*




Complejidad Deducida: O(n).

Justificación: El método copiar itera una vez sobre la lista original. El bucle for i in range(self.tamanio) se ejecuta n veces. Dentro del bucle, se llama a nueva_lista.agregar_al_final(), que es una operación O(1). Por lo tanto, el tiempo total es nO(1) = O(n). El tiempo de ejecución crece linealmente con el número de elementos, lo cual cumple el requisito de ser O(n) y no O($n^2$)

*Gráfica función invertir*




Complejidad Deducida:O(n).

Justificación: La implementación de invertir crea una nueva lista vacía. Luego, recorre la lista original completa una vez, lo cual toma n pasos. En cada paso, ejecuta nueva_lista.agregar_al_inicio(pivote.dato). Esta operación es O(1) (porque la nueva lista mantiene una referencia a cabeza). El tiempo total para llenar la nueva lista es nO(1) = O(n). Finalmente, se reasignan las referencias de la lista original, lo cual es O(1). La complejidad dominante es O(n).