ANÁLISIS DE PROYECTO 1:

En este proyecto se implementan tres algoritmo de ordenamiento (Bubble sort, quicksort, y radix sort) con el objetivo de analizar su funcionamiento y eficiencia a la hora de ordenar listas aleatorias, además los resultados fueron comparados con la función **sorted()** de python. A continuación analizaremos cada algoritmo por separado

Bubble Sort
Este es un algoritmo de ordenamiento simple que compara pares adyacentes de elementos y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista queda ordenada. Tiene la desventaja de ser poco eficiente a medida que el tamaño de la lista aumenta (En comparación con otros algoritmos). En todos los casos, tanto en el mejor (Lista ya ordenada) como en el peor tiene una complejidad de $O(n^2)$

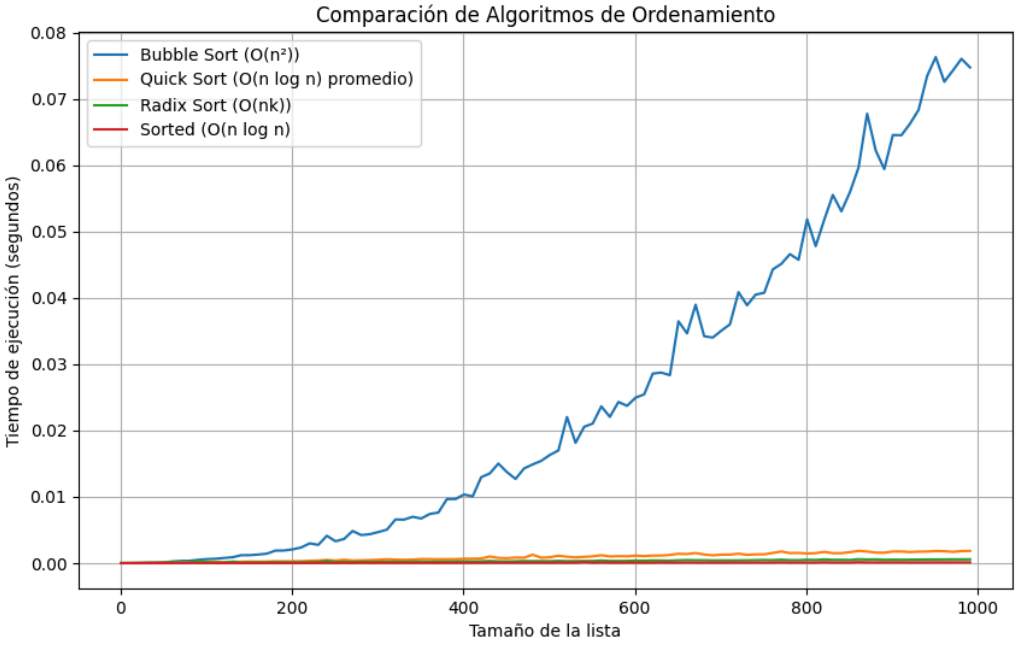
Quicksort
Este es un algoritmo que selecciona un pivote y divide la lista en dos sublistas: los elementos menores que el pivote y los mayores. Luego aplica recursivamente el algoritmo sobre las sublistas hasta que esta queda ordenada. Tiene la desventaja de que al ser recursivo puede consumir mucha memoria para listas grandes. En el mejor caso y caso promedio tiene una complejidad de $O\ (nlog(n))$ pero en el peor de los casos (Lista ya ordenada o pivote más elegido) tiene una complejidad de $O(n^2)$

Radixsort
Este es un algoritmo no comparativo que ordena números por dígitos, procesándolos del menos significativo al más significativo (en este caso, cinco iteraciones para números de cinco cifras). Son muy eficientes para listas cuyos números tienen una longitud fija. Este posee una complejidad de $O(Kn)$ Siendo K el número de dígitos por número

Sorted
La función sorted() en Python implementa el algoritmo Timsort, una combinación híbrida entre Merge Sort y Insertion Sort, desarrollado específicamente para obtener buen rendimiento en listas reales, que suelen estar parcialmente ordenadas. Tiene una complejidad en casos promedios de $O\ (nlog(n))$ y en el mejor de los casos con una lista casi ordenada tiene una complejidad de $O\ (n)$

Resultados
Para analizar los resultados de los distintos algoritmos utilizamos la librería **time** de python, generamos listas aleatorias (**random**) de diferentes tamaños, y medimos el tiempo de inicio y de finalización del algoritmo luego procedemos a graficar

mediante la librería **matplotlib.pyplot** además calculamos el tiempo que le tomaria a la función **Sorted()** de python para compararlo con nuestros algoritmos



Bubble Sort mostró un crecimiento cuadrático del tiempo, acorde con su complejidad teórica $O(n^2)$.

Quicksort tuvo un rendimiento considerablemente mejor que Bubble Sort, confirmando su comportamiento $O(n \log n)$ en el promedio.

Radix Sort fue el algoritmo más rápido en todos los tamaños, con un comportamiento lineal, como predice su complejidad $O(k \cdot n)$.

ANÁLISIS DE PROYECTO 2:

En este ejercicio se implementó una estructura de datos dinámica del tipo Lista Doblemente Enlazada, que permite almacenar elementos comparables (Con su anterior y posterior) de cualquier tipo. Se desarrollaron operaciones básicas como inserción, extracción, copia, rotación y concatenación, junto con el uso de excepciones para un manejo robusto de errores.

Funcionamiento del codigo (Clases):

Definimos dos clases:

Nodo: contiene un dato (numero en este caso) y las referencias con el nodo anterior y siguiente

ListaDobleEnlazada: Gestiona los nodos y las diferentes operaciones

Metodos:

- esta_vacia(): Si len==0 devuelve true
- __len()__ : Devuelve la longitud de la lista
- agregar_al_inicio(item): insertar en la cabeza, teniendo en cuenta las relaciones de los nodos
- agregar_al_final(item): insertar en la cola, teniendo en cuenta las relaciones de los nodos
- insertar(item,posicion): inserta un dato en una posicion en especifico
- extraer(posicion): extrae un dato de una posicion en especifico
- copiar(): crea una copia elemento a elemento
- invertir(): invierte la lista en lugar, sin listas auxiliares
- concatenar(Lista): añade otra lista al final (Concatena)
- __add__(Lista): permite usar el operador **“+”** para sumar dos listas (reutiliza concatenar y copiar).

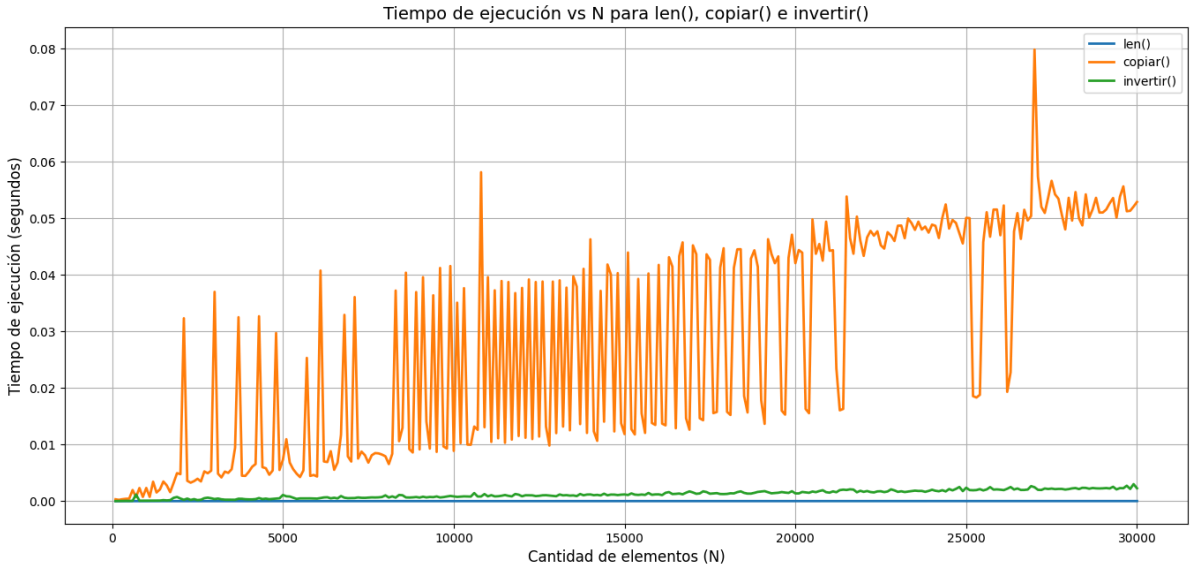
Analizamos la complejidad de tres de estos metodos y los comparamos su eficacia con listas de hasta 10000 elementos

Resultados:

len(): el tiempo de ejecución es constante, confirmando su complejidad $\mathcal{O}(1)$.

copiar(): el tiempo crece linealmente con el tamaño, confirmando $\mathcal{O}(n)$, aunque observamos irregularidades en la grafica podemos observar una clara tendencia.

invertir(): también mostró crecimiento lineal, como se esperaba.



La estructura desarrollada es eficiente, pasa todos los tests de unidad y cumple con los requisitos del enunciado. Las operaciones fueron cuidadosamente implementadas para respetar la eficiencia esperada en cada caso, y el análisis empírico respalda las estimaciones teóricas.

ANÁLISIS DE PROYECTO 3:

En este ejercicio se implementó un juego de cartas “Guerra” mediante programación orientada a objetos. La lógica principal del juego fue provista por la cátedra, y se nos encargó implementar correctamente la clase **`Mazo`**, utilizando como base una lista doblemente enlazada desarrollada previamente (Proyecto 2). Esta clase **`Mazo`** gestiona las operaciones de apilar, desapilar y manipular cartas durante el transcurso del juego.

Implementación

Para representar un mazo, se utilizó una estructura de doble cola, implementada manualmente mediante una lista doblemente enlazada. Esta decisión fue clave para garantizar que las operaciones de inserción y extracción tanto desde el inicio como desde el final tuvieran complejidad constante $O(1)$, como lo exige el enunciado.

Se definió una excepción personalizada para manejar intentos de extracción de cartas desde un mazo vacío.

El mazo fue utilizado por la clase **`JuegoGuerra`**, que coordina la lógica de turnos, comparación de cartas, manejo de empates (“guerra”) y determinación del ganador.

Para representar las cartas, se utilizó la clase **`Carta`**, con métodos especiales para determinar el valor numérico y comparar dos cartas de manera adecuada.

Resultados:

Una vez implementada correctamente la clase **`Mazo`**, se corroboró que el juego funciona según lo esperado. Se ejecutaron partidas completas entre dos jugadores simulados, y se observó que:

- Las reglas del juego se respetan adecuadamente, incluyendo el manejo de empates (guerra).
- Se detectan correctamente los escenarios de victoria y empate tras un número máximo de turnos (**`N_TURNOS`**).
- No se detectaron errores al utilizar mazos vacíos, gracias al manejo adecuado de excepciones.

Se realizaron múltiples ejecuciones con distintas semillas aleatorias (**`random_seed`**) para asegurar el correcto reparto y funcionamiento estocástico del juego.

TEST

Para validar el correcto funcionamiento de las clases **`Mazo`** y **`JuegoGuerra`**, se implementaron pruebas unitarias utilizando el módulo **`unittest`**.

En **`test_mazo.py`**, se realizaron los siguientes tests:

- **`test_poner_sacar_arriba`**: verifica que al colocar cartas en la parte superior del mazo y luego extraerlas, se respete el orden LIFO esperado.
- **`test_poner_abajo`**: valida que al insertar cartas por la parte inferior del mazo, estas se conserven en el orden de inserción al ser retiradas por la parte superior.

En **`test_juego_guerra.py`**, se realizaron partidas automáticas con semillas fijas para comprobar resultados reproducibles. Se testearon tres escenarios:

- **`test_resulta_gana_jugador1`**: se ejecutaron partidas donde el jugador 1 gana con distinta cantidad de turnos (137, 638, 1383).

- **`test_resulta_gana_jugador2`**: se simularon partidas ganadas por el jugador 2 (145, 1112, 1373).

- **`test_resulta_empate`**: se verificó que con ciertas semillas (547 y 296) el juego termina en empate al alcanzar el máximo de turnos.

Todos los tests se ejecutaron exitosamente, lo que confirma que tanto las reglas del juego como las estructuras de datos implementadas responden correctamente a los requisitos del enunciado.

Conclusión:

Este ejercicio permitió consolidar conceptos clave de estructuras de datos dinámicas, encapsulamiento, manejo de errores mediante excepciones, y simulación de eventos. El uso de una lista doblemente enlazada fue esencial para garantizar eficiencia y respetar las restricciones de memoria del enunciado, evitando el uso de estructuras estándar como **`list`** o **`deque`** de Python.

Además, el diseño modular y reutilizable de las clases permitió una integración fluida entre los componentes: **`Carta`**, **`Mazo`** y **`JuegoGuerra`**. Finalmente, la implementación pasó exitosamente los tests provistos por la cátedra, lo que confirma la validez de la solución propuesta.