TP N°1:

Se trabajó en tres proyectos: algoritmos de ordenamiento, siendo estos burbuja, quicksort y radix sort y se analizó la función sorted() de Python, que combina eficiencia y estabilidad en diversos casos prácticos (Proyecto 1); estructura de datos dinámicas (TAD), en específico la Lista Doblemente Enlazada (Proyecto 2); y la simulación de un juego mediante programación centrada a objetos, donde se aplicaron los conceptos anteriores para implementar la lógica de un juego de cartas ("Guerra"), utilizando la clase Mazo basada en listas doblemente enlazadas. Esta parte integró tanto la correcta estructuración de clases como la interacción entre objetos y estructuras complejas.

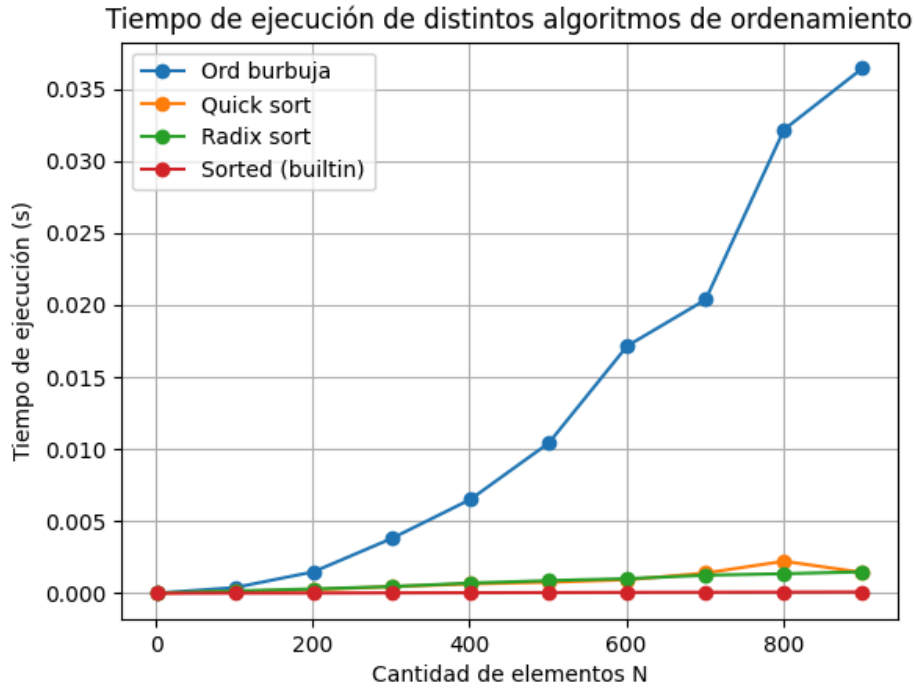
Proyecto 1: Algoritmos de Ordenamiento y Comparativa de Tiempos

En este proyecto se implementaron tres algoritmos de ordenamiento : ordenamiento burbuja, Quicksort y Radix Sort. El objetivo principal fue evaluar el desempeño de cada uno al ordenar listas de números de cinco dígitos generados aleatoriamente. Además, se compararon estos algoritmos con la función sorted(), una herramienta built-in altamente optimizada.
Finalmente, se graficaron y analizaron los tiempos de ejecución de cada uno de los algoritmos para entender su eficiencia.

Para realizar las pruebas se generaron listas con un tamaño variable entre 1 y 1000, compuestas por números de 5 dígitos aleatorios, utilizando el módulo random, y se midieron los tiempos de ejecución para cada algoritmo usando el módulo time.
Al implementar los algoritmos se pudo observar que el ordenamiento burbuja realiza múltiples pasadas comparando y permutando elementos adyacentes, es simple pero ineficiente para listas grandes, su complejidad es de O(n^2) debido a que examina cada combinación posible de pares en cada iteración ; Quicksort se basa en el enfoque de divide y vencerás, es de complejidad O(n log n), eficiente en promedio pero su desempeño puede degradarse en el peor caso; y Radix Sort es de complejidad O(d*(n+k)), ordena los números procesando dígito a dígito, lo que lo hace efectivo para datos numéricos.

Se realizó una comparación de estos algoritmos de ordenamiento con la función sorted(), la misma es una herramienta para ordenar elementos que utiliza Timsort, un algoritmo híbrido que combina Merge Sort e Insertion Sort, lo que le permite ser rápido en listas parcialmente ordenadas y mantener una complejidad de O(n log n) en promedio, consta de un ordenamiento estable, funciona con cualquier iterable, devuelve una nueva lista ordenada sin alterar el iterable original, y admite parámetros como key para personalizar el criterio de ordenamiento y reverse para ordenar de mayor a menor.

Análisis del gráfico:



El gráfico compara los tiempos de ejecución de los cuatro algoritmos de ordenamiento en función del número de elementos en la lista.
Ordenamiento burbuja: se representa con una línea azul, muestra un crecimiento abrupto en el tiempo de ejecución a medida que aumenta el número de elementos, lo que significa que es ineficiente en listas grandes.
Quicksort: está representado con una línea naranja, mantiene tiempos bajos y estables , es eficiente en la mayoría de los casos excepto en el peor escenario (cuando la lista está casi ordenada y el pivote es mal elegido).
Radix Sort: representado con una línea verde, muestra un rendimiento con tiempos relativamente constantes.
Sorted: la línea roja refleja la eficacia de este algoritmo ya que es la más baja en todo el gráfico,confirmando su rapidez y optimización.

Proyecto 2: TAD

Se ha implementado el TAD ListaDobleEnlazada que contiene las funciones solicitadas. La lista se compone de nodos enlazados bidireccionalmente, con referencias explícitas al nodo anterior y al siguiente. Los métodos respetan la complejidad O grande solicitada, en los casos que no aclaraba buscamos la O más conveniente.

Cada nodo tiene self.dato, self.anterior, self.siguiente y las funciones necesarias de asignación y obtención de datos actual, anterior y siguiente. La clase principal usa referencias a los extremos: self.cabeza, self.cola; un contador: self.tamanio; y su posicionamiento: self.actual, luego tiene las funciones:

agregar_al_inicio(dato): Crea un nuevo nodo. Si la lista está vacía, el nuevo nodo es también el final , si no actualiza las referencias para que el nuevo nodo apunte al anterior primero, y viceversa. Complejidad: O(1)

agregar_al_final(dato): Similar a agregar_al_inicio, pero se enlaza al final. Complejidad: O(1)

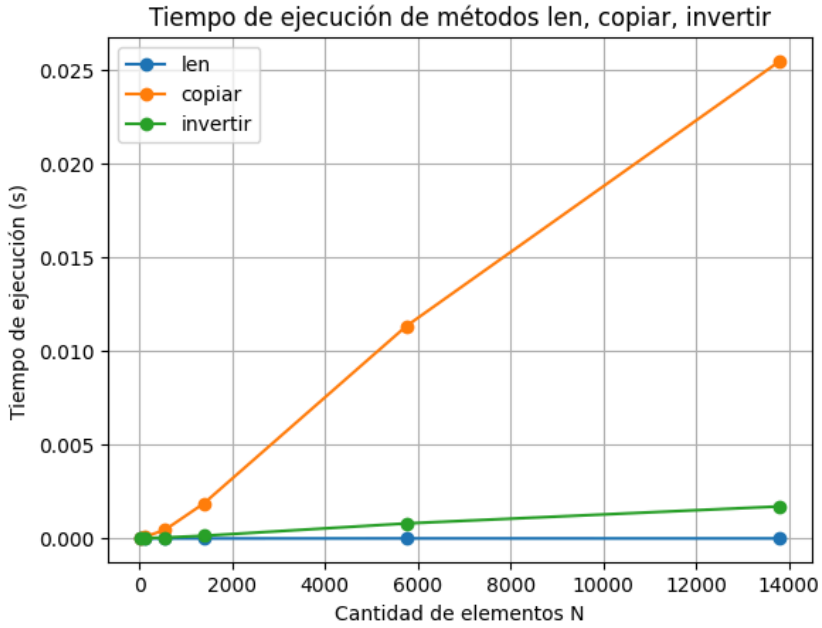
extraer_primero() / extraer_ultimo(): Quita el nodo del principio o final (cabeza/cola). Devuelve su valor y ajusta las referencias de la nueva cabeza o cola. Complejidad: O(1)

copiar(): Crea una nueva lista, para eso recorre la lista actual y agrega cada valor al final de la nueva lista. Complejidad: O(n), más costosa en tiempo por creación de nodos y enlaces.

invertir(): Recorre la lista y en cada nodo intercambia siguiente y anterior. Al final, intercambia cabeza y cola. Complejidad: O(n)

__len__(): Devuelve un contador de tamaño, actualizado en cada operación de inserción/extracción. Complejidad: O(1)

Gráfica 2: Comparación del tiempo de ejecución entre los métodos **len** (complejidad O(1)), **copiar** (O(n), recorre la lista y clona nodo por nodo) e **invertir** (O(n), intercambia punteros anterior y siguiente en cada nodo).



len(): Su crecimiento de tiempo es constante sin importar el tamaño de la lista. Se confirma la **complejidad O(1)**.
copiar() **e** **invertir()**: Presentan un crecimiento lineal del tiempo con respecto a la cantidad de nodos. Se confirma la **complejidad O(n)**. La difenercia de tiempo de ejecución se puede deber a la cantidad de operaciones internas y la memoria requerida (copiar crea nuevos nodos, invertir solamente acomoda las referencias en los nodos ya existentes, sin generar nuevos objetos).

Proyecto 3: Implementación de un Juego de Cartas con Estructuras Enlazadas

En este proyecto se nos pidió implementar el juego de cartas "Guerra", en el cual dos jugadores compiten por quedarse con todas las cartas del mazo. El objetivo del proyecto radica en la implementación de la clase **Mazo**, basada en una lista doblemente enlazada (ListaDobleEnlazada del **Proyecto 2**), que permite una manipulación eficiente de las cartas desde ambos extremos.
El flujo del juego se basa en una secuencia de rondas, donde ambos jugadores revelan sus cartas, y el jugador con la carta más alta gana ambas cartas. Si ambas cartas tienen el mismo valor, se inicia una "guerra", donde cada jugador pone cartas adicionales en juego para resolver el empate.
Para implementar esto, la clase **Mazo** debe permitir operaciones específicas como:

- Repartir cartas: Distribuir las cartas entre los jugadores.
- Voltear cartas: Los jugadores deben ser capaces de extraer cartas del mazo en un orden específico (delante o detrás del mazo), lo que debe estar optimizado para una lista doblemente enlazada.
- Lógica de guerra: Si los jugadores empatan, deben añadir cartas a la pila de guerra, y la implementación de esta lógica de desempate es crucial.

De allí se implementan las operaciones necesarias para manipular las cartas del mazo donde los métodos principales incluyen:

- **__init__**: Inicializa el mazo como una lista doblemente enlazada

- **poner_carta_arriba** y **poner_carta_abajo**: Insertan cartas al inicio y al final del mazo, respectivamente.

- **sacar_carta_arriba**: Extrae una carta del inicio, lanzando una excepción DequeEmptyError si el mazo está vacío.

- **__len__**: Devuelve la cantidad de cartas en el mazo.

- **__iter__**: Hace que el mazo sea iterable.

- **__str__**: Devuelve una representación en cadena del mazo, mostrando 10 cartas por línea.

Observaciones:
El diseño aprovecha las ventajas de las listas doblemente enlazadas para insertar y extraer cartas de forma eficiente en ambos extremos del mazo.
La excepción **DequeEmptyError** asegura que se maneje adecuadamente el intento de sacar cartas de un mazo vacío.
El método **mostrar** dentro de **sacar_carta_arriba** sugiere que las cartas tienen un atributo **visible**, lo cual podría ser relevante para la lógica de mostrar cartas durante el juego.