Actividad 1:

1. ¿Cómo funcionan sorted() en Python?

La función sorted() en Python usa un algoritmo llamado **Timsort**, que es una combinación optimizada de **Merge Sort** e **Insertion Sort**. Este algoritmo está diseñado para ser eficiente en la práctica, con una excelente complejidad tanto en listas grandes como en listas parcialmente ordenadas.

Características de sorted():

- **Complejidad temporal**:
  - **Mejor caso**: O(n), cuando la lista ya está ordenada o casi ordenada.
  - **Caso promedio y peor caso**: O(n log n), típico de algoritmos basados en comparaciones, como Merge Sort.
- **Complejidad espacial**: O(n), ya que Timsort no es in-place (necesita memoria adicional para realizar el merge).
- **Funciona bien con listas heterogéneas**: Además de números, puede ordenar cualquier tipo de datos que implemente comparaciones (<, >, ==).
- **Optimizado para listas parcialmente ordenadas**: Si una lista tiene segmentos ya ordenados ("runs"), Timsort lo aprovecha para reducir el número de operaciones.

Ventajas:

- **Robustez y eficiencia**: Timsort es rápido tanto en listas completamente desordenadas como en listas parcialmente ordenadas, lo que lo hace más flexible que otros algoritmos.
- **Estabilidad**: Mantiene el orden relativo de los elementos que son iguales, lo que es importante en algunos contextos.

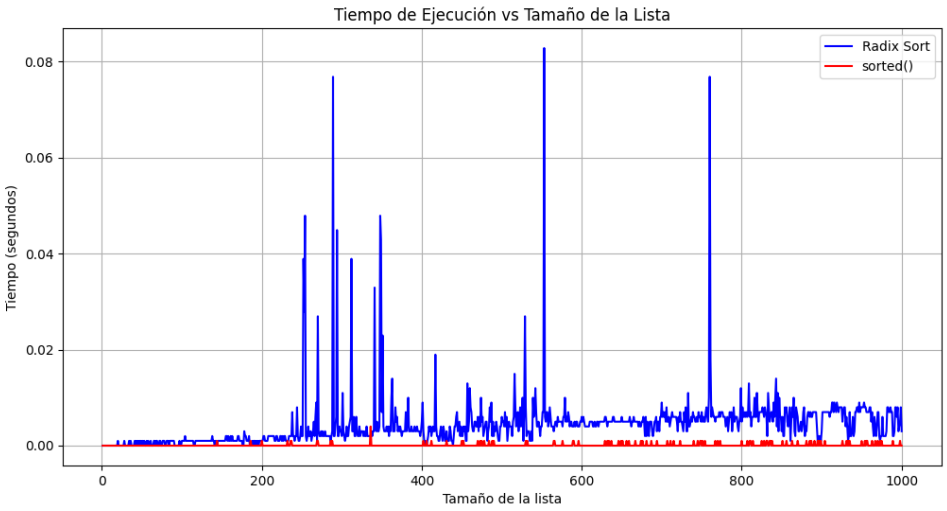
2. Comparación de los algoritmos de la Actividad 1

Ahora comparemos los algoritmos de **Actividad 1** (Ordenamiento Burbuja, Quicksort y Radix Sort) con sorted() en términos de eficiencia y aplicación práctica.

a. Ordenamiento Burbuja

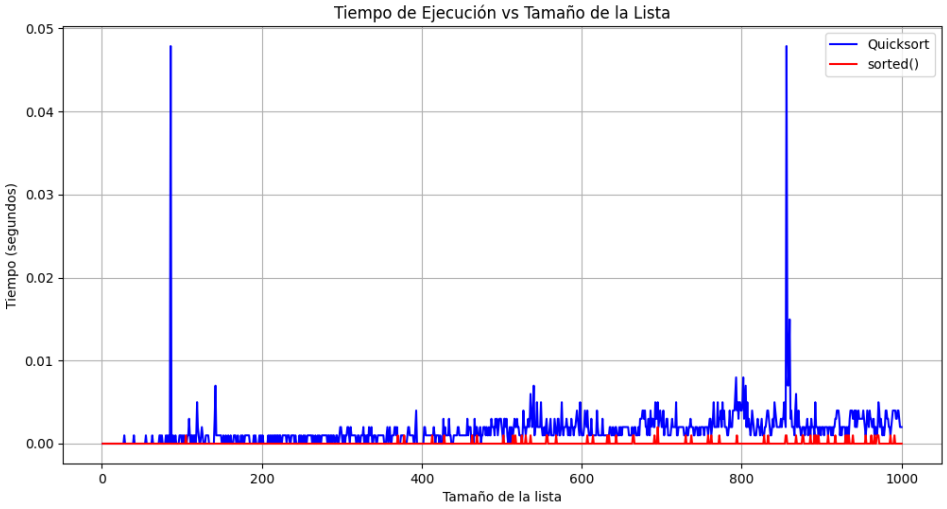
- **Complejidad temporal**:
  - **Peor caso y caso promedio**: $O(n^2)$
  - **Mejor caso**: O(n), si la lista ya está ordenada.
- **Comparación con sorted()**:
  - Burbuja es extremadamente ineficiente para listas grandes, ya que realiza muchas comparaciones e intercambios innecesarios.
  - sorted() es mucho más rápido con O(n log n) en el peor caso.
  - **Conclusión**: Burbuja es solo útil para listas pequeñas o para aprendizaje, mientras que sorted() es mucho más práctico en cualquier

escenario.



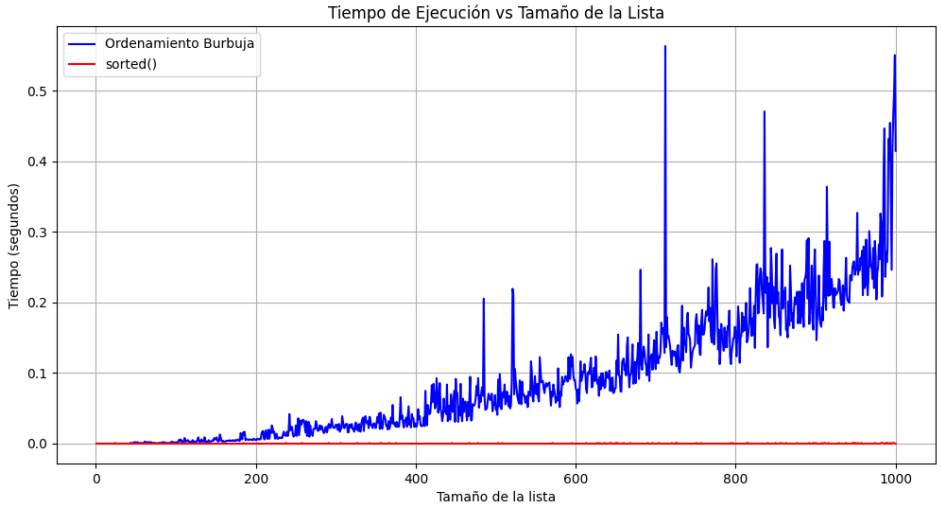
b. Quicksort

- **Complejidad temporal**:
  - **Mejor caso y caso promedio**: O(n log n), como sorted().
  - **Peor caso**: O(n²), si la lista está desbalanceada (por ejemplo, si ya está ordenada y se elige un mal pivote).
- **Comparación con sorted()**:
  - Quicksort es eficiente y rápido en la mayoría de los casos, pero tiene un peor caso de O(n²), aunque esto puede minimizarse con optimizaciones en la elección del pivote.
  - sorted() (Timsort) nunca cae en el caso O(n²), siempre mantiene O(n log n), lo que lo hace más predecible y robusto.
  - **Conclusión**: Quicksort es eficiente, pero sorted() ofrece mayor estabilidad y predictibilidad.



c. Radix Sort

- **Complejidad temporal**:
  - **Mejor caso y caso promedio**: O(n * k), donde k es el número de dígitos en el número más grande.
  - **Peor caso**: O(n * k), donde **k** puede aumentar en función de los datos, pero generalmente se considera un algoritmo lineal.
- **Comparación con sorted()**:
  - Radix Sort es muy eficiente cuando los elementos son enteros y los valores tienen un número limitado de dígitos, como en listas de números de 5 dígitos.
  - sorted() (Timsort) es más flexible, ya que puede ordenar cualquier tipo de dato y es más general, mientras que Radix Sort solo funciona con enteros.
  - **Conclusión**: Radix Sort es excelente para ordenar grandes volúmenes de enteros con un número limitado de dígitos, pero sorted() es más versátil y adecuado para datos heterogéneos.



3. Resumen Comparativo

Algoritmo	Complejidad Promedio	Mejor Caso	Peor Caso	Espacio Adicional	Observaciones
Burbuja	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Ineficiente para listas grandes.
Quicksort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	Rápido, pero el peor caso puede ser lento.

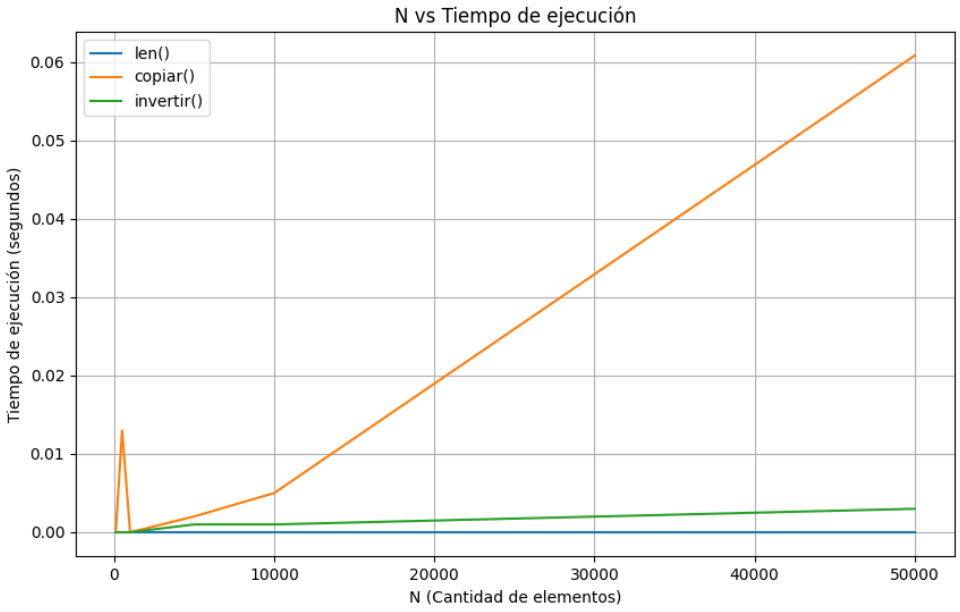
Radix Sort	O(n * k)	O(n * k)	O(n * k)	O(n)	Muy eficiente para números enteros con pocos dígitos.
sorted() (Timsort)	O(n log n)	O(n)	O(n log n)	O(n)	Eficiente, robusto y general para diferentes tipos de datos.

Conclusión Final

- **sorted()** es generalmente la mejor opción cuando se trata de listas heterogéneas o no se conoce de antemano la naturaleza de los datos, ya que es eficiente, flexible y estable.
- **Quicksort** y **Radix Sort** pueden ser alternativas eficaces en escenarios específicos, como cuando se busca velocidad en listas grandes de enteros (Radix Sort) o cuando se sabe que la entrada es favorable para Quicksort.
- **Burbuja**, por su parte, es más un algoritmo educativo y es prácticamente ineficiente para listas grandes.

Actividad 2:

Analisis de complejidad con N cantidad de datos para las funciones implementadas



La gráfica muestra el tiempo de ejecución en función de la cantidad de elementos N para los métodos len(), copiar(), e invertir() de la lista doblemente enlazada. A continuación, explico los resultados y deduzco los órdenes de complejidad:

1. **len()**: El tiempo de ejecución se mantiene constante (línea prácticamente horizontal). Esto confirma que su complejidad es **O(1)**, ya que el método simplemente devuelve el valor del atributo tamanio sin realizar ningún recorrido en la lista.

   Explicación de la complejidad:

   Operación constante: El método simplemente devuelve el valor de self.tamanio, que se mantiene actualizado en cada operación de inserción o eliminación de elementos. Esta operación no requiere recorrer la lista ni realizar cálculos adicionales.

   Dado que acceder a un valor almacenado en una variable es una operación de tiempo constante, la complejidad es **O(1)**.

2. **copiar()**: El tiempo de ejecución aumenta linealmente con NNN, lo que sugiere una complejidad **O(N)**. El método recorre todos los nodos de la lista para copiar cada uno, lo que es coherente con este comportamiento.

   Explicación de la complejidad:

- **Recorrido de la lista**: El bucle while actual: recorre todos los nodos de la lista original, lo que implica que el tiempo de ejecución depende directamente del número de nodos N.
- **Agregar al final (agregar_al_final)**: Agregar un nuevo nodo al final de una lista doblemente enlazada es una operación de tiempo constante **O(1)**, ya que se actualizan los punteros de la cola en cada inserción sin necesidad de recorrer la lista.

Cada nodo de la lista original se copia y se agrega al final de la nueva lista, lo que implica un trabajo proporcional al número total de nodos. Como cada inserción es **O(1)** y hay N nodos, la complejidad total es **O(N)**.

3. **invertir()**: Al igual que copiar(), el tiempo de ejecución de invertir(), aunque en menor medida, crece linealmente con N, indicando una complejidad **O(N)**. Esto es esperable, ya que el método invierte los punteros de cada nodo, recorriendo toda la lista una vez.

   Explicación de la complejidad:

- **Recorrido completo de la lista**: El bucle while actual: itera a través de todos los nodos de la lista, realizando dos operaciones principales en cada nodo: intercambiar los punteros siguiente y anterior, y luego moverse al siguiente nodo. Esto requiere visitar cada nodo una vez.
- **Intercambio de punteros**: El intercambio de punteros es una operación de tiempo constante **O(1)**.
- El bucle realiza una cantidad de trabajo proporcional al número de nodos en la lista, es decir, si hay N nodos, se ejecuta N veces. Por lo tanto, la complejidad es **O(N)**, donde N es la cantidad de nodos en la lista.

Actividad 3:

El juego de cartas "Guerra" es un juego de azar en el que dos jugadores compiten por ganar todas las cartas de un mazo. El juego incluye las siguientes mecánicas:

- Reparto inicial del mazo de 52 cartas, dividiendo 26 cartas para cada jugador.
- Cada jugador voltea una carta en cada turno, y quien tenga la carta más alta gana las cartas jugadas.
- Si ambos jugadores sacan cartas del mismo valor, se produce una "guerra", en la que ambos jugadores colocan varias cartas boca abajo y una última carta boca arriba para decidir el ganador.
- El jugador que logre ganar todas las cartas del oponente será el vencedor.

Solución Propuesta

La implementación del juego se basa en la clase `Mazo`, que hace uso de una lista doblemente enlazada para almacenar y manipular las cartas. La **lista doblemente enlazada** proporciona una estructura para extraer cartas de la parte superior o inferior del mazo, así como para añadir cartas después de cada turno.

Estructura de la Solución

- **Clase `Mazo`**: Representa el mazo de cartas de cada jugador. Los métodos principales implementados permiten agregar y extraer cartas, tanto desde el inicio como desde el final del mazo.
- **Clase `ListaDobleEnlazada`**: Implementa la lista doblemente enlazada. Esta estructura se usa para almacenar los objetos de tipo `Carta`, facilitando las operaciones de inserción y extracción de cartas con una complejidad eficiente.
- **Clase `Carta`**: Representa una carta del mazo con un valor numérico. El valor determina cuál carta es más alta, sin importar el palo.

Funcionalidades de la Clase `Mazo`

La clase `Mazo` implementa los siguientes métodos clave:

- **`poner_carta_arriba(carta)`**: Añade una carta al inicio del mazo (parte superior).
- **`poner_carta_abajo(carta)`**: Añade una carta al final del mazo (parte inferior).
- **`sacar_carta_arriba(mostrar=False)`**: Extrae la carta en la parte superior del mazo. Si se indica, la carta se muestra.
- **`sacar_carta_abajo()`**: Extrae la carta en la parte inferior del mazo.
- **`esta_vacio()`**: Verifica si el mazo está vacío.
- **`tamanio()`**: Devuelve el tamaño del mazo (número de cartas).

La implementación de la clase `Mazo` utilizando una lista doblemente enlazada permite manejar el mazo de cartas de manera eficiente, cumpliendo con los requisitos del juego "Guerra". El uso de esta estructura de datos garantiza un acceso rápido a las cartas en los extremos del mazo, lo que es esencial para el funcionamiento del juego. Además, el manejo de excepciones asegura que no se produzcan errores al interactuar con un mazo vacío.

La correcta implementación de la clase `Mazo` permite que los tests de la cátedra para el juego de "Guerra" y la clase `Mazo` se ejecuten sin problemas, validando el correcto funcionamiento de las operaciones básicas de inserción, extracción y manipulación de las cartas.