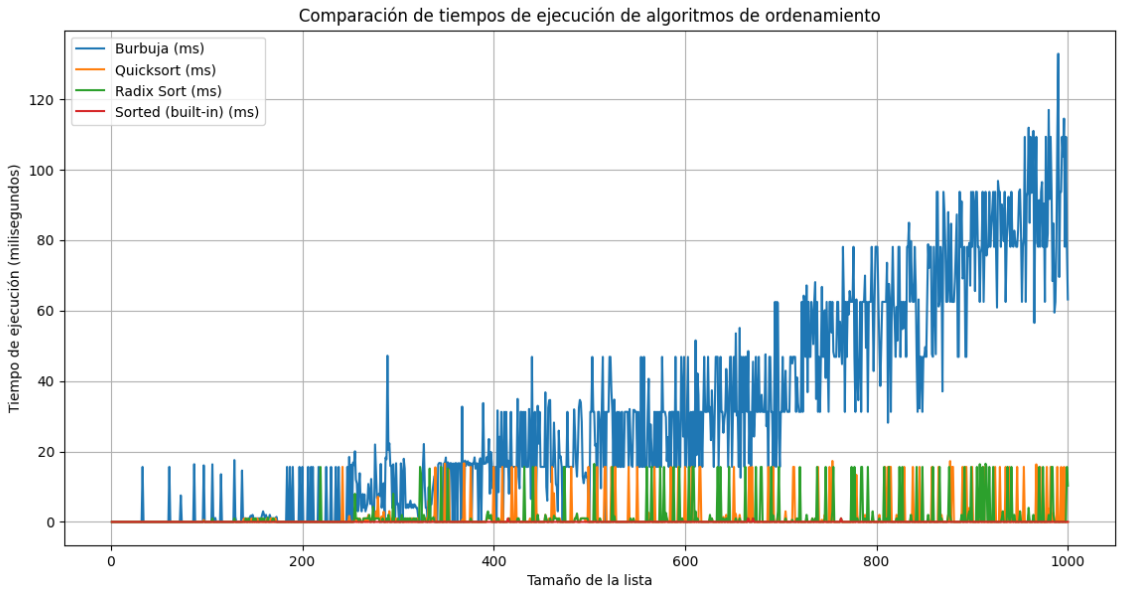
Problema 1

Gráfica de Tiempos de Ejecución



Algoritmo de Burbuja (Bubble Sort)

- **Complejidad:**
  - Peor caso y caso promedio: O(n^2)
- **Análisis a priori:**
  - El algoritmo compara e intercambia elementos adyacentes repetidamente, lo que requiere $\frac{n(n-1)}{2}$ comparaciones en el peor de los casos. Este enfoque no es eficiente para listas grandes debido a su alto número de comparaciones e intercambios.

Quicksort

- **Complejidad:**
  - Caso promedio: O($n \log n$)
  - Peor caso: O(n^2)
- **Análisis a priori:**
  - Quicksort selecciona un pivote y divide la lista en dos sublistas. En el caso promedio, divide la lista de manera equilibrada, resultando en O($n \log n$). Sin embargo, si las divisiones son desiguales (peor caso), la complejidad alcanza O(n^2).

-

Radix Sort

- **Complejidad:**
  - Peor caso: O(nk), donde k es el número de dígitos del valor más grande en la lista.
- **Análisis a priori:**
  - Radix Sort clasifica los elementos basándose en sus dígitos, utilizando Counting Sort como subrutina. Su eficiencia depende del tamaño de los números, siendo muy eficaz para ordenar grandes cantidades de números pequeños, pero menos versátil para otros tipos de datos.

Sorted (Timsort)

- **Complejidad:**
  - Mejor caso: O(n)
  - Peor caso y caso promedio: O($n \log n$)
- **Análisis a priori:**
  - Timsort es una mezcla de Merge Sort e Insertion Sort. Aprovecha las secuencias ya ordenadas ("runs") para mejorar el rendimiento en listas parcialmente ordenadas. Es estable y garantiza O($n \log n$) en el peor caso, lo que lo hace más eficiente y versátil en comparación con Quicksort y otros métodos.

Problema 2

La implementación de la lista doblemente enlazada incluye los métodos esenciales para manipular la lista, como agregar, insertar, extraer, copiar, invertir y concatenar nodos. Cada método está diseñado para mantener la integridad de la estructura de datos mientras optimiza el rendimiento de las operaciones.

`__len__`: Descripción: Retorna el tamaño actual de la lista. Esta operación es de tiempo constante, O(1). **Resultado:** Permite obtener rápidamente el número de nodos en la lista.

`__iter__`: Descripción: Itera sobre los nodos de la lista, devolviendo sus valores. La complejidad es O(n), donde n es el número de nodos. **Resultado:** Facilita la iteración a través de la lista para operaciones como impresión o procesamiento.

`agregar_al_inicio`: Descripción: Añade un nodo al inicio de la lista. Esta operación es O(1). **Resultado:** Permite insertar elementos al principio de la lista eficientemente, actualizando la cabeza y los punteros de los nodos.

`agregar_al_final`: Descripción: Añade un nodo al final de la lista. Esta operación también es O(1). **Resultado:** Permite agregar elementos al final de la lista, actualizando la cola y los punteros correspondientes.

`insertar`: Descripción: Inserta un nodo en una posición específica, requiriendo O(n) en el peor caso. **Resultado:** Permite la inserción de nodos en cualquier posición de la lista, manejando correctamente los punteros de los nodos adyacentes.

`extraer`: Descripción: Elimina un nodo en una posición específica y devuelve su valor. La complejidad es O(n) en el peor caso. **Resultado:** Elimina nodos de cualquier posición, ajustando los punteros de los nodos vecinos para mantener la integridad de la lista.

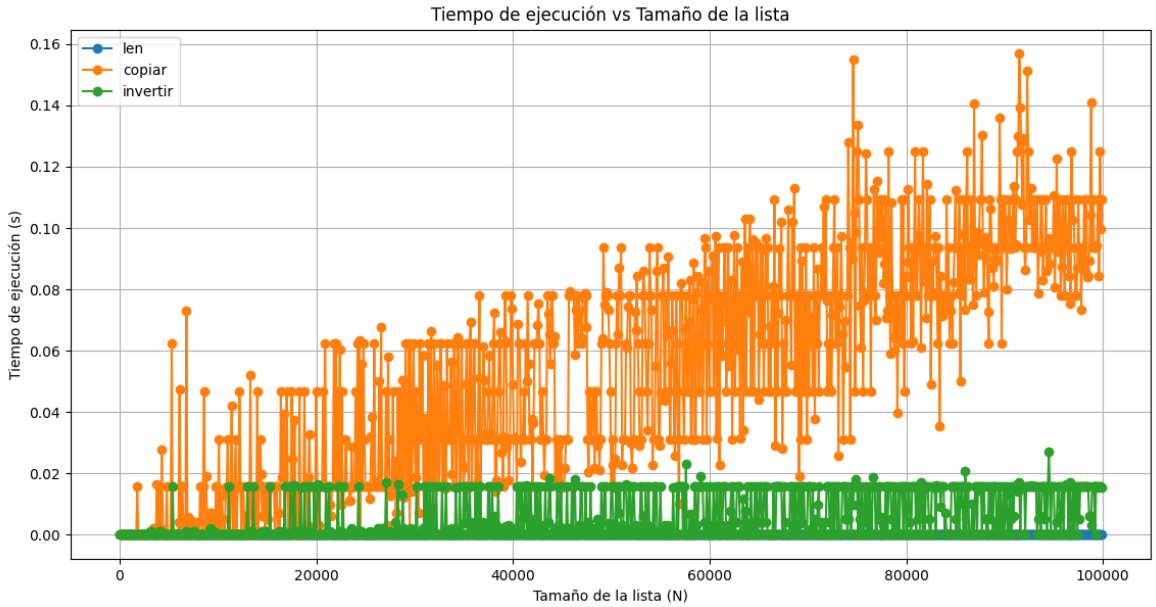
`copiar`: Descripción: Crea una copia exacta de la lista actual. La operación tiene una complejidad de O(n). **Resultado:** Genera una nueva lista con los mismos elementos, proporcionando una copia independiente de la lista original.

`invertir`: Descripción: Invierte el orden de los nodos en la lista. Esta operación es O(n). **Resultado:** Reorganiza los punteros de los nodos para invertir el orden de la lista, haciendo que la cabeza se convierta en la cola y viceversa.

`concatenar`: Descripción: Une dos listas, anexando la segunda al final de la primera. La complejidad es O(1) si la lista a concatenar está vacía, y O(m) en el peor caso, donde m es el tamaño de la lista a concatenar. **Resultado:** Fusiona dos listas en una sola, ajustando los punteros de la cola de la lista original para conectar con la cabeza de la lista añadida.

`__add__`: Descripción: Crea una nueva lista que es la unión de dos listas. La complejidad es O(n + m), donde n y m son los tamaños de las listas a unir. **Resultado:** Combina todas

las posiciones de ambas listas en una nueva, permitiendo la adición de todos los elementos de las listas involucradas.



- **`len`:** El tiempo de ejecución de la operación `len` es prácticamente constante y muy bajo, independientemente del tamaño de la lista. Esto es consistente con la complejidad teórica de O(1).
- **`copiar`:** El tiempo de ejecución de `copiar` aumenta linealmente con el tamaño de la lista. Esto indica una complejidad de O(n), ya que cada nodo debe ser copiado a la nueva lista.
- **`invertir`:** El tiempo de ejecución de `invertir` también aumenta linealmente con el tamaño de la lista. Esto se debe a que cada nodo debe tener sus punteros invertidos, y este proceso se realiza una vez por cada nodo.

Problema 3

El código del módulo `mazo`, junto con el código del juego de guerra y la clase `Carta` provistos por la cátedra, simula correctamente el juego de "Guerra". La implementación de la clase `Mazo`, basada en una lista doblemente enlazada (creada en el problema 2), permite gestionar el juego de manera eficiente.

La excepción `DequeEmptyError` permite manejar de manera segura las situaciones donde un jugador se queda sin cartas.

Métodos de la Clase `Mazo`

1. **`__len__`**
   - **Descripción**: Retorna el número de cartas en el mazo.
   - **Eficiencia**: O(1). La longitud del mazo se almacena, por lo que acceder a ella es una operación de tiempo constante.
2. **`poner_carta_arriba`**
   - **Descripción**: Agrega una carta al inicio del mazo (parte superior), simulando poner una carta en la parte superior del mazo.
   - **Eficiencia**: O(1). Esta operación se realiza en tiempo constante, ya que no es necesario recorrer la lista doblemente enlazada.
3. **`poner_carta_abajo`**
   - **Descripción**: Agrega una carta al final del mazo (parte inferior).
   - **Eficiencia**: O(1). Similar a la operación anterior, agregar una carta al final del mazo se realiza en tiempo constante.
4. **`sacar_carta_arriba`**
   - **Descripción**: Extrae una carta del inicio del mazo (parte superior) y opcionalmente la marca como visible.
   - **Eficiencia**: O(1). Extraer una carta del inicio del mazo es rápido y eficiente, y la operación está protegida contra errores lanzando la excepción `DequeEmptyError` si el mazo está vacío.
5. **`sacar_carta_abajo`**
   - **Descripción**: Extrae una carta del final del mazo (parte inferior).
   - **Eficiencia**: O(1). La extracción del final del mazo también es una operación de tiempo constante, lo que garantiza eficiencia incluso cuando se manejan muchas cartas.
6. **`__str__`**
   - **Descripción**: Proporciona una representación en cadena del contenido del mazo, mostrando todas las cartas en su orden actual.
   - **Eficiencia**: O(n), donde n es el número de cartas en el mazo. Dado que se recorre toda la lista para construir la representación en cadena, esta operación tiene una complejidad lineal.