Universidad Nacional de Entre Ríos
Facultad de Ingeniería

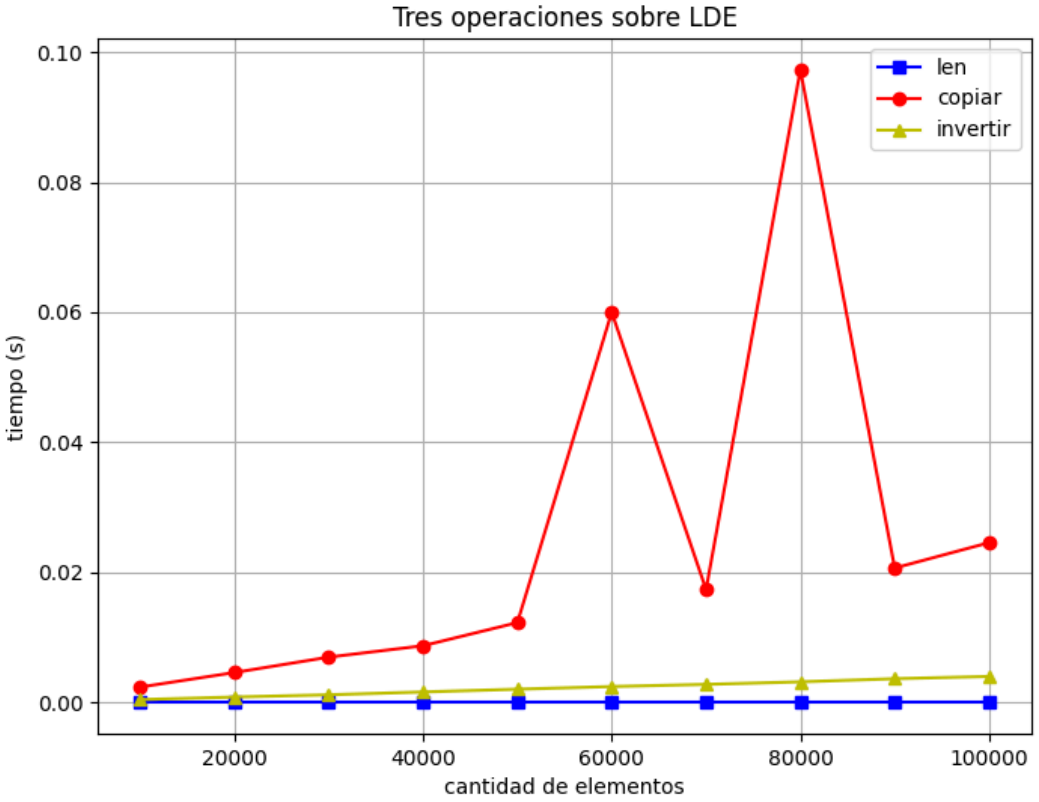
Algoritmos y estructuras de datos

Informe general del Trabajo Práctico N°1
“Aplicaciones de los tipos abstractos de datos”

Alumnos:

- Tarabini Melina
- Rodríguez Esteban

Problema 1:



En la lista doblemente enlazada, la operación `len` tiene complejidad **O(1)** porque solo consulta un contador interno. En cambio, **invertir** y **copiar** son lineales: invertir es más rápido ya que sólo intercambia punteros, mientras que copiar requiere crear nuevos nodos, lo que lo hace más costoso.

La estructura se basa en **nodos** con un valor y dos referencias (anterior y siguiente). La clase `ListaDobleEnlazada` gestiona estos nodos mediante tres atributos:

- **__ cabeza** : referencia al primer nodo (su atributo anterior siempre es None).
- **__cola**: referencia al último nodo (su atributo siguiente siempre es None).

- **__cantidad_elementos**: contador que lleva la cantidad de nodos, inicializado en cero.

Entre las funciones implementadas se encuentran:

- **__str__(self)**: devuelve una cadena con los elementos separados por " - ". Si no hay nodos, retorna "lista sin elementos".

- **esta_vacia(self)**: indica si la lista está vacía.

- **__len__(self)**: permite utilizar len(lista) para conocer el total de nodos.

- **agregar_al_inicio(self, item)**: inserta un nodo al inicio. Si estaba vacía, el nodo se vuelve cabeza y cola a la vez.

- **agregar_al_final(self, item)**: incorpora un nodo al final, con manejo especial si la lista aún no tiene elementos.

- **insertar(self, item, posicion=-1)**: inserta en cualquier posición válida. En 0 agrega al inicio, en negativo o al final coloca el nodo al final, y en posiciones intermedias ajusta los punteros vecinos.

- **extraer(self, posicion=-1)**: elimina un nodo y devuelve su valor. Si no se pasa posición, quita el último. Contempla los casos de cabeza y cola y actualiza referencias. Incluye validaciones de errores.

- **copiar(self)**: genera una lista nueva con los mismos valores, preservando el orden. Se crean nodos nuevos con insertar.

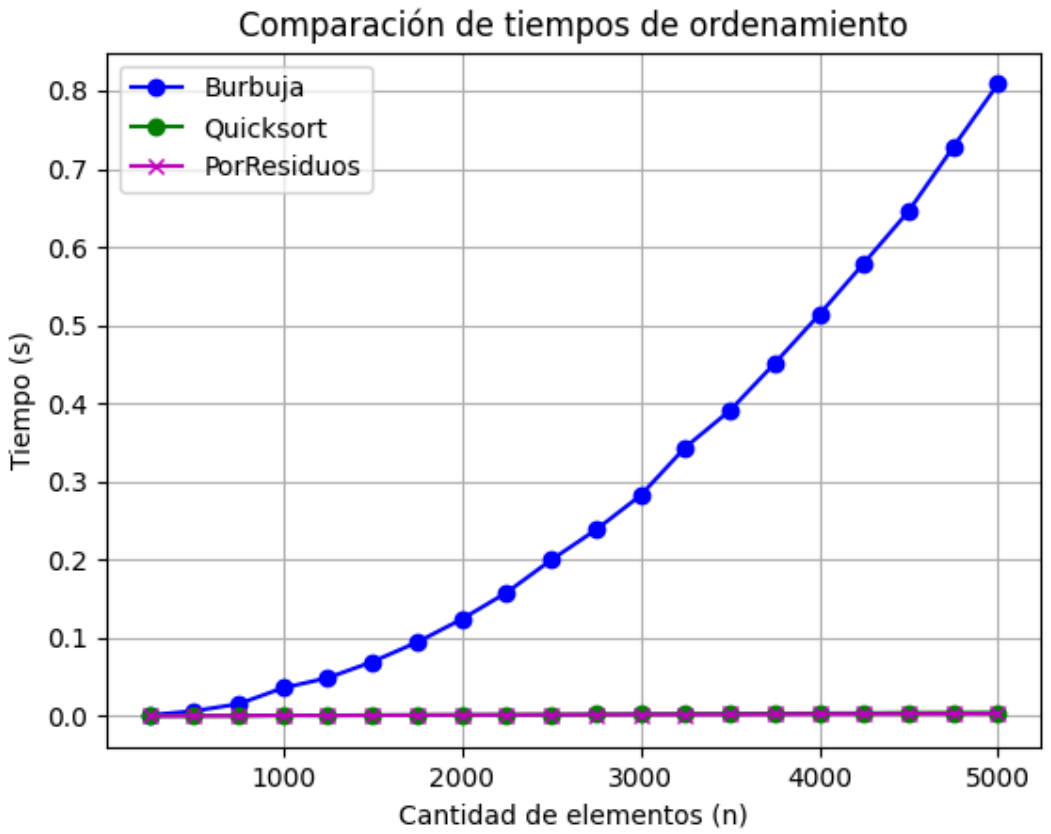
- **invertir(self)**: invierte el orden de los nodos intercambiando sus punteros y luego actualiza cabeza y cola.

- **concatenar(self, p_lista)**: agrega al final de la lista todos los elementos de otra lista, sin modificar la original.

- **__add__(self, p_lista)**: redefine el operador +, creando una lista nueva que combina primero la actual y luego la pasada como parámetro, usando internamente concatenar.

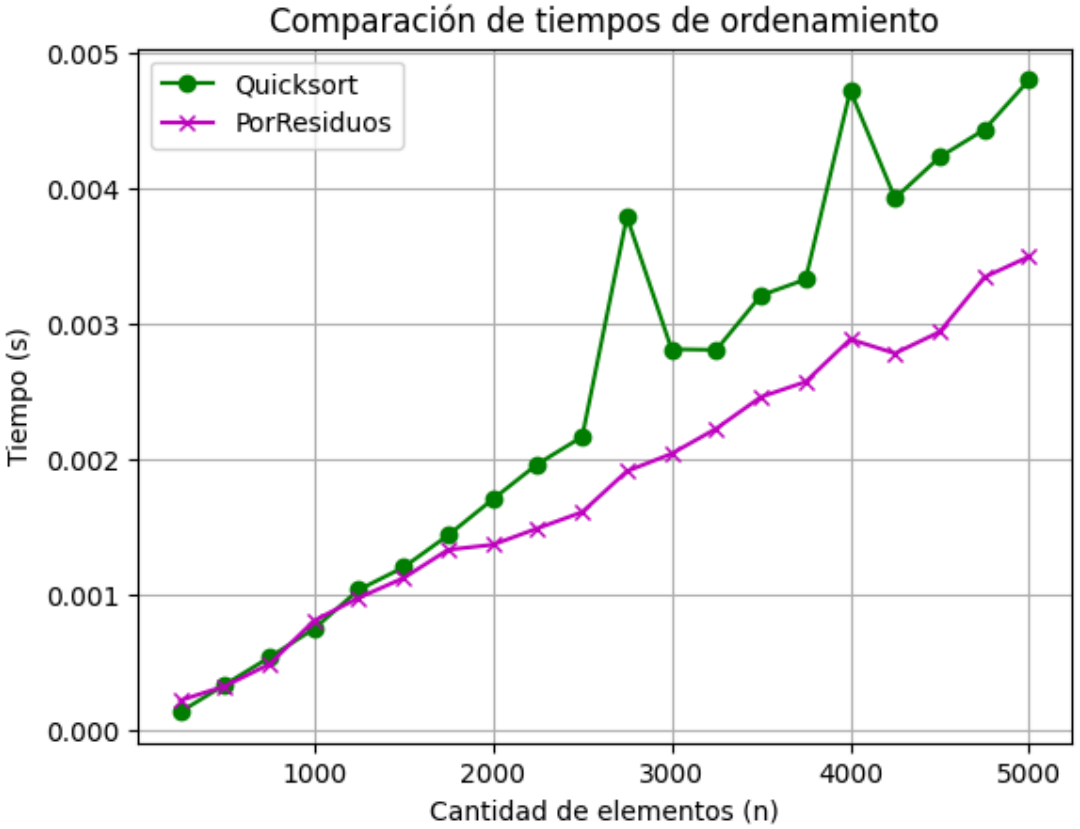
Problema 2:
Para la resolución de este problema se consideró al código brindado por la cátedra y posteriormente implementamos a la clase mazo, la cual representa una baraja de objetos Carta mediante una lista doblemente enlazada -que se implementó en el problema anterior. La estructura de lista doblemente enlazada permite una manipulación eficiente de las cartas, facilitando operaciones como agregar, quitar y recorrer las cartas tanto desde el principio como desde el final del mazo. Como resultado, el código informará si el ganador fue el jugador 1, jugador 2 o bien si hubo algún empate. La lógica para determinar el ganador se basa en la comparación de las cartas jugadas por cada jugador en cada ronda.

Problema 3:



Descripción:
En este gráfico se comparan los tiempos de ejecución de los tres algoritmos de ordenamiento implementados. Se observa claramente que el método Burbuja presenta un crecimiento cuadrático, aumentando el tiempo de forma muy pronunciada a medida que crece el tamaño de la lista. En contraste, Quicksort y el método Por Residuos (Radix Sort) mantienen tiempos

prácticamente constantes en la escala del gráfico, evidenciando una mayor eficiencia.



Descripción:
En este gráfico se amplía la comparación entre **Quicksort** y **Por Residuos**, dejando afuera al Burbuja para poder apreciar mejor las diferencias. Se ve que ambos algoritmos crecen con el tamaño de la lista, pero el **Radix Sort** tiende a ser más eficiente y presenta un crecimiento más suave en el tiempo de ejecución.

Ordenamiento Burbuja Complejidad: O(n^2):
● Compara pares de elementos adyacentes y los intercambia si están en orden incorrecto.
● En el peor caso, requiere recorrer el arreglo n veces y en cada pasada compara hasta n elementos.
● En el gráfico se observa que el tiempo de ejecución de Burbuja crece de manera no lineal y rápidamente a medida que n aumenta, lo cual concuerda con una complejidad cuadrática.

Quicksort:
Complejidad promedio: O(n log (n))
Peor caso: si la lista ya está ordenada o semi-ordenada, el algoritmo puede presentar una complejidad O(n^2), debido al desbalance que quedaría en las listas.
● Quicksort utiliza el enfoque de “divide y vencerás”.
● Divide el arreglo en dos subarreglos y luego ordena recursivamente.
● En el gráfico, la línea de Quicksort es prácticamente plana, lo que indica que su tiempo de ejecución es muy bajo en comparación con la de burbuja, y crece mucho más lentamente, como se espera de un algoritmo eficiente.

Por residuos:
Complejidad estimada:O(n·k), donde k es el número de dígitos analizados.
● Aunque no es un algoritmo de ordenamiento típico, puede estar clasificando elementos con base en una operación rápida (como el módulo).
● En el gráfico, su tiempo también se mantiene constante y muy bajo, incluso más bajo que Quicksort, lo cual sugiere que se trata de un algoritmo lineal o casi constante en este rango de valores.

La función sorted() de Python se utiliza para ordenar elementos de cualquier iterable (como listas, tuplas, diccionarios, etc.) y devuelve una nueva lista ordenada, sin modificar el iterable original. Esta divide una lista en sublistas de tamaño óptimo y las ordena con un algoritmo de inserción, para luego fusionarlas, lo que logra un orden de complejidad de O(n log n) donde n es el número de elementos en el iterable que se está ordenando.