Trabajo Práctico N°1

Objetivos

- Aplicar los conceptos que resulten necesarios para la implementación de algoritmos de ordenamiento.
- Implementar tipos abstractos de datos (TAD).
- Analizar y determinar la complejidad de los algoritmos implementados.
- Utilizar manejo de excepciones para validación de datos y operaciones en los TADs.
- Uso de pruebas unitarias para corroborar el correcto funcionamiento de los códigos implementados.

Consignas

Problema 1

Implementar el **TAD Lista doblemente enlazada** (ListaDobleEnlazada) que permita almacenar elementos de cualquier tipo que sean comparables (por ejemplo enteros, flotantes, strings). La implementación debe respetar la siguiente **especificación lógica**:

- **esta_vacia()**: Devuelve **True** si la lista está vacía.
- **agregar_al_inicio(item)**: Agrega un nuevo ítem al inicio de la lista.
- **agregar_al_final(item)**: Agrega un nuevo ítem al final de la lista.
- **insertar(item, posicion)**: Agrega un nuevo ítem a la lista en "posicion". Si la posición no se pasa como argumento, el ítem debe añadirse al final de la lista. "posicion" es un entero que indica la posición en la lista donde se va a insertar el nuevo elemento. Si se quiere insertar en una posición inválida, que se arroje la debida excepción.
- **extraer(posicion)**: elimina y devuelve el ítem en "posición". Si no se indica el parámetro posición, se elimina y devuelve el último elemento de la lista. La complejidad de extraer elementos de los extremos de la lista debe ser O(1). Si se quiere extraer de una posición indebida, que se arroje la debida excepción.
- **copiar()**: Realiza una copia de la lista elemento a elemento y devuelve la copia. Verificar que el orden de complejidad de este método sea O(n) y no $O(n^2)$.
- **invertir()**: Invierte el orden de los elementos de la lista.
- **concatenar(Lista)**: Recibe una lista como argumento y retorna la lista actual con la lista pasada como parámetro concatenada al final de la primera.
- **__len__()**: Devuelve el número de ítems de la lista.
- **__add__(Lista)**: El resultado de "sumar" dos listas debería ser una nueva lista con los elementos de la primera lista y los de la segunda. Aprovechar el método concatenar para evitar repetir código.
- **__iter__()**: permite que la lista sea recorrida con un ciclo for

El inicializador __init__ debe crear una lista originalmente vacía.

Realizar una gráfica de N (cantidad de elementos) vs tiempo de ejecución para los siguientes métodos: len, copiar e invertir (verificar que los hayan implementado de la forma más eficiente posible). Explicar los resultados y deducir los órdenes de complejidad a partir de las gráficas.

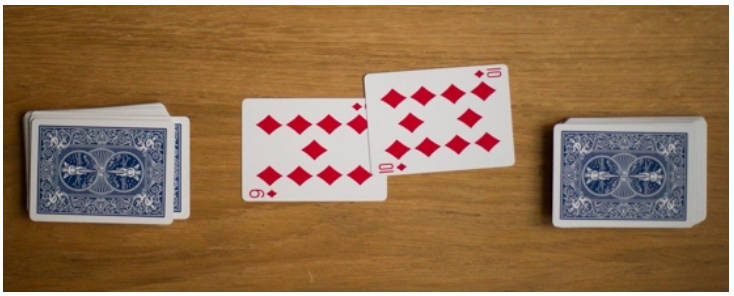
Su clase ListaDobleEnlazada debe pasar el test provisto por la cátedra.

Aclaración: No utilice almacenamiento adicional innecesario ni funciones de la biblioteca estándar de python o de terceros en la implementación de los métodos. La implementación debe ser eficiente en relación al uso de la memoria de la computadora. Ejemplo: no se puede copiar el contenido de la Lista doblemente enlazada a una lista de Python, y viceversa, para implementar las operaciones del TAD.

Problema 2

El juego de cartas **"Guerra"** es un juego de azar donde el objetivo es ganar todas las cartas. El juego consiste en las siguientes etapas:

- Inicialmente, un mazo de cartas común (52 cartas) se reparte entre 2 jugadores de forma que ambos jugadores tengan 26 cartas cada uno (un mazo de 26 cartas por jugador). Los jugadores no pueden ver sus cartas ni las del oponente.
- El juego se realiza por turnos: en cada turno, ambos jugadores deben colocar sus cartas boca abajo sobre la mesa. El jugador 1 voltea la primera carta del mazo en el centro de la mesa. El jugador 2 hace lo mismo con la primera carta de su mazo.



- El jugador con la carta más alta gana el turno y se queda con ambas cartas para añadirlas al final de su mazo en el mismo orden en el que fueron reveladas. El orden de las cartas de menor a mayor es: 2,3,4,5,6,7,8,9,10,J,Q,K,A (no se tiene en cuenta el palo de la baraja).
- Ambos jugadores vuelven a voltear la siguiente carta en su mazo y se repite el proceso. Esto continúa hasta que uno de los jugadores gana todas las cartas.
- Si al voltear sus cartas ambos jugadores tienen el mismo valor numérico, entonces hay guerra: cada uno pone tres cartas boca abajo (que hace de botín de guerra) y voltean otra carta más cada uno para desempatar. El ganador de la ronda se lleva

todas las cartas puestas en juego. Si hay empate nuevamente, se repite el proceso. Si alguno se queda sin cartas en el proceso, pierde.

Se les provee un código con el algoritmo del juego guerra, pero falta incorporar la implementación de la clase Mazo, el cual debe hacer uso de una lista doble enlazada (realizada en el problema anterior) para almacenar objetos de tipo Carta y realizar las operaciones que se le solicitan. Observar cuáles son los métodos que debe tener la clase Mazo e implementarlos correctamente. En caso de querer extraer una carta de un mazo vacío, se deberá lanzar la excepción *DequeEmptyError* (debe estar definido en el mismo archivo que la clase Mazo).

Si la clase Mazo está correctamente implementada, el test para la clase mazo y el test para el juego guerra deberán ejecutarse sin problemas.

Todos los códigos provistos por la cátedra para el presente ejercicio pueden encontrarse en un único repositorio en el siguiente enlace.

Aclaración:

- **No modifique el programa de prueba de ninguna manera, la firma de los métodos de su clase deben corresponderse con los del test para poder pasarlo.**

Problema 3

Implementar en python los siguientes algoritmos de ordenamiento:

a) Ordenamiento burbuja
b) Ordenamiento quicksort
c) Ordenamiento por residuos (radix sort)

Corroborar que funcionen correctamente con listas de números aleatorios de cinco dígitos generados aleatoriamente (mínimamente de 500 números en adelante).

Medir los tiempos de ejecución de tales métodos con listas de tamaño entre 1 y 1000. Graficar en una misma figura los tiempos obtenidos. ¿Cuál es el orden de complejidad O de cada algoritmo? ¿Cómo lo justifica con un análisis a priori?

Comparar ahora con la función built-in de python **sorted**. ¿Cómo funciona sorted? Investigar y explicar brevemente.

Entregables

El proyecto deberá trabajarse en un repositorio remoto en GitHub. Pueden hacer commits y push hasta la fecha límite (fijarse en el campus). El proyecto deberá tener una carpeta por ejercicio, y en cada una de éstas debe contener el código de resolución, los tests y demás material específico de cada inciso. Separar el código de cada ejercicio en módulos y una aplicación principal.

El proyecto también deberá contener un informe en pdf con una breve explicación de la solución y resultados de cada uno de los ejercicios (conclusiones o gráficas según lo solicite el enunciado). Puede agregarse algún pseudocódigo si se considera necesario para explicar las soluciones entregadas. En el caso del ejercicio 1 y 3, debe contener los análisis de complejidad que solicita el enunciado, con las gráficas correspondientes.