Trabajo Práctico 1

Algoritmos y Estructuras de Datos

Grupo 1:

- Boero, Martina
- Córdoba, Aníbal

Planteo y resolución del ejercicio 1: LDE

Planteo de la *clase nodo*, que represente cada una de sus instancias un elemento de la lista con sus respectivos enlaces a los elementos siguiente y anterior
Para implementar la Lista Doblemente Enlazada:

- Constructor: crea una lista vacía, con atributos cabeza, cola y tamaño nulos.

- Estas listas doblemente enlazadas pueden mostrar sus datos e iterarse con los métodos **str** e **iter** respectivamente.

- **Tamaño**: devuelve la cantidad de elementos de la lista. Además se sobrecarga el método **len** para poder mostrar el tamaño.

- **esta_vacia** devuelve True si el tamaño de la lista es 0, es decir, está vacía

- **agregar_al_inicio** : recibe un dato que se va a insertar en la posición 0. Contempla dos casos: si la lista esta vacía, define este elemento como cabeza y cola, y si la lista tiene elementos, define al nuevo elemento como cabeza. Se aumenta en 1 el tamaño.

- **agregar_al_final** : de manera análoga a agregar_al_inicio, aquí se recibe un elemento a agregar al final, y contempla los casos de la lista vacía (definiendo al elemento como cabeza y cola), o, si no, define al nuevo elemento como cola. Se aumenta en 1 el tamaño.

- **insertar** : permite agregar un elemento a la lista en una posición a elección. Si la posición ingresada es menor a 0 o mayor al tamaño de la lista, se levanta el error **IndexError**. Si se desea insertar en la posición 0 o al final de la lista, se usarán las fuciones agregar_al_inicio o agregar_al_final para cada caso correspondiente. La

inserción propiamente dicha, dada si se quiere insertar entre las posiciones 0 y final, se realizará avanzando iterativamente hasta el elemento anterior al de la posición deseada (temp) y cambiando las referencias anteriores y siguientes, cuidando que no se pierda ningún elemento. Tras finalizar, se aumenta en 1 el tamaño.

- **extraer**: se extrae y retorna el dato alojado en la posición indicada. Si no se indica ninguna, se utiliza el valor por defecto -1, que extrae al final de la lista. En primer lugar se evalúan las siguientes excepciones: si la lista está vacía, no se puede realizar la extracción de ningún elemento, así como sí se indica una posición menor a 0 o mayor o igual al tamaño de la lista, levantando un IndexError. Se asume que el primer índice es 0.
  Se evalúan 4 casos principales:

  - Si el tamaño de la lista es 1, la lista se quedará vacía
  - Si se quiere extraer la cabeza de la lista (desde la posición 0), se deberán reasignar las referencias y la nueva cabeza de la lista.
  - Si se desea extraer la cola (desde la posición de tamaño-1 o sin pasar parámetro, por defecto -1), se deberán reasignar las referencias y la nueva cola de la lista.
  - Para extraer cualquier otro elemento dentro de la lista, se deberá recorrer iterativamente hasta llegar a la posición anterior de la que cuyo elemento se desea extraer (temp), y se cambian las referencias anteriores y siguientes, para dejar el elemento extraído almacenado en una variable a retornar.
    El tamaño se reduce en uno.

- **copiar**: se realiza y retorna una copia de la lista. Para ello, se crea una nueva LDE, y se recorre la lista inicial, agregando al final de la nueva LDE cada dato.

- **concatenar**: modifica la lista actual para concatenarle la lista indicada a su final. Se realiza una copia de la segunda lista para evitar perderla. Para unir ambas listas, se apunta al siguiente de la cola de la primera lista como la cabeza de la segunda, así como el anterior de la cabeza de la segunda como la cola de la primera, siempre trabajando con la copia de la segunda y editando la primera. Se asigna como la nueva cola la cola de la segunda y se hace crecer el tamaño igual al tamaño de la segunda lista.

- **add**: se sobrecarga el operador '+' para concatenar dos listas sin modificarlas, devolviendo esta función una nueva lista igual a la concatenación de ambas.

- **invertir**: se invierte el orden de una lista recorriéndola e cambiando las referencias: el elemento anterior pasa a ser el siguiente y viceversa. Por último se reasignan las nuevas cabeza y cola.

- **ordenar**: análisis del algoritmo de ordenamiento implementado:

  - Utilizamos un algoritmo de ordenamiento por inserción, que tiene un orden de complejidad, a priori, de O(n^2). Dicho orden se asume a partir de los dos bucles while anidados, cada uno recorriendo n valores de la lista.

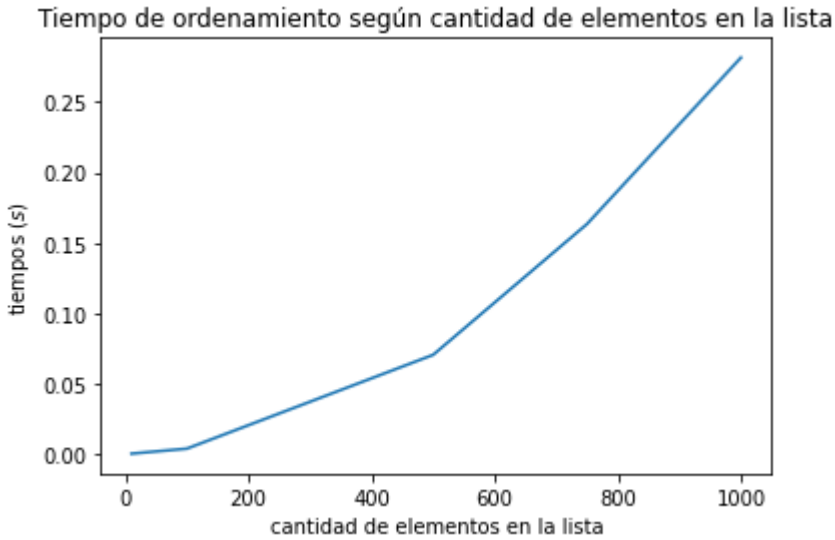
  - Adjuntamos el codigo:

```
def ordenar(self):
    """
    Ordena los elementos de la lista de "menor a mayor".
    La lista se ordena sobre sí misma con un algoritmo de insercion

    Returns
    -------
    None.

    """
    temp=self.cabeza.siguiente
    while temp:
        while (temp.anterior and temp.dato < temp.anterior.dato):
            temp.dato, temp.anterior.dato = temp.anterior.dato, temp.dato
            temp=temp.anterior
        temp=temp.siguiente
```

  - Realizamos un análisis cuantitativo, a posteriori, del orden de complejidad midiendo tiempo de ordenamiento contra cantidad de valores a ordenar, obteniendo la siguiente gráfica:



Los tests provistos por la cátedra se ejecutan sin errores.

Planteo y resolución del ejercicio 2: Juego Guerra

Se importan las implementaciones de clases Nodo y Lista Doblemente Enlazada del ejercicio anterior para ser utilizadas.
Los tests provistos por la cátedra se ejecutan sin errores.
En este caso, para simular una baraja de cartas se decidió utilizar un **TAD cola doble**, que permita agregar cartas arriba y abajo, así como remover de arriba y abajo. El remover abajo, sin embargo, no será utilizado. Para su implementación se utiliza la clase ColaDoble y se reasignan los métodos utilizados en el ejercicio anterior. De esta manera, agregarArriba() y removerArriba se encargan de poner y quitar elementos en la posición 0 (la cima del mazo), y los agregarAbajo y removerAbajo, en la posición final de la cola.
Se implementa la clase **Carta**, con atributos valor y palo y métodos que permiten comparaciones entre instancias de esta clase.
Clase **JuegoGuerra**:
- Constructor y Atributos: Se crea el mazo inicial a partir del cual se va a repartir, usando una ColaDoble. Además, cada jugador tendrá su mazo correspondiente. Para el componente de aleatoriedad, cada juego tendrá una semilla. Se declaran atributos como int turnos jugados, str ganador, y bool empate.
- **inicializar_mazo**: en primer lugar, se crea una lista de python de cartas, cada una con un valor y un palo. Utilizando la semilla, se mezcla aleatoriamente esta lista. Luego se agrega iterativamente cada carta en el mazo principal. De esta manera de tiene un mazo de ColaDoble cuyos nodos son 52 cartas mezcladas.
- **repartir**: para cada carta del mazo grande, esta se removerá y se agregará arriba en el mazo de cada jugador, intercalando una carta para cada uno.
- **jugar**: como acción principal de este juego, se retira de arriba una carta de cada mazo de jugador y se comparan. Si una carta es mayor a la otra, el jugador que sea dueño de la carta mayor agregará al final de su mazo ambas cartas. Si ambas cartas son iguales, se entra en guerra (ver método guerra). Tras esta acción, el turno se incrementa.
- **mostrar_por_consola** sigue el diseño propuesto por la cátedra para la exhibición de la mesa de juego en los turnos que no son guerra. El mostrar por consola para guerra se encuentra implementado dentro del método guerra.
- **guerra**: en primer lugar, se crea una lista de cartas sobre la mesa como ColaDoble, a la que se le van a agregar las dos cartas que entraron en guerra. Luego, se removerán seis cartas más de los mazos de los jugadores, intercalando una de cada mazo, hasta retirar tres a cada jugador. En cada iteración se chequea que los mazos posean cartas, porque si algún jugador se queda sin cartas, se declara ganador el contrincante. Las cartas retiradas a cada mazo se almacenan en la lista de cartas sobre la mesa. Una vez retiradas las 6, se vuelven a sacar 2 cartas más y se agregan a la lista. Estas dos son las que pueden volver a entrar en guerra, por lo que es importante el bucle externo que las compara en cada vuelta. Se exhibe el mazo del jugador 1 tras las extracciones de cartas. Para mostrar la lista de cartas en mesa de la manera solicitada

se utiliza una copia de la lista, de la que se extrae cada carta y muestra como su valor o como X según se encuentre en disputa o en el botín. Luego se muestra el mazo del jugador 2, dando un total de 52 cartas sobre la mesa. Por último, si no hubo un jugador que se haya quedado sin cartas durante la guerra (es decir, si no hay ganador aún), se comparan las dos últimas cartas en disputa en la lista de cartas sobre mesa, concatenando esta lista al final del mazo del jugador con la carta mayor.
- **iniciar_juego**: engloba todos los métodos anteriores. Inicializa el mazo y lo reparte, y mientras no haya ganador y la cantidad de turnos no exceda los 10000, se juegue. Si algún jugador se queda sin cartas, se declare ganador el contrincante. Si se llega al turno 10001 sin haber un ganador, se declara empate.
Los tests provistos por la cátedra se ejecutan sin errores.

Planteo y resolución del ejercicio 3: mezcla directa

En primer lugar, se utilizó el código provisto para generar el archivo con datos.
Para plantear este algoritmo de ordenamiento, es necesario que las sublistas de tamaño "clave" estén ordenadas previamente, por lo que implementamos el método **ordenar_sublistas**, que utiliza espacio en memoria en forma de lista de python para ordenar por medio del algoritmo de inserción (ver ordenar_por_inserción) las sublistas de N líneas del archivo de texto, siendo N la clave. Por último, se sobreescribe el archivo. Somos conscientes del alto orden de complejidad que tiene esta acción, por lo que recomendamos comenzar con clave 1 para evitarlo.
El método de **ordenar_por_insercion** ordena por el algoritmo de ordenamiento de inserción sublistas de tamaño clave con un orden de complejidad de O(n^2).
Habiendo realizado este paso previo, se puede llamar al método **mezcla_directa**, que hace uso de las funciones dividir y fusionar y duplicar la clave, para luego hacer llamadas recursivas hasta que el tamaño de la clave sea del número de líneas del archivo.
Para **dividir** el archivo, se abren dos archivos auxiliares en modo escritura, y el archivo principal para lectura. Por cada línea leída en el archivo principal se almacenarán en un archivo auxiliar, comenzando por el primero y al alcanzar una cantidad "clave" de líneas almacenadas en el primer auxiliar, se cambiará al segundo auxiliar, y así sucesivamente. Esta función devuelve la cantidad de datos que posee cada archivo auxiliar.
Para **fusionar** los archivos, es necesario abrir los auxiliares en modo de lectura, y el principal, en escritura. Si se consideran bloques de tamaño de la clave, se realizará la siguiente acción tantas veces como bloques existan en el primer archivo auxiliar(siempre será mayor o igual al segundo), redondeado hacia arriba. Se colocan dos punteros al comienzo de cada bloque, cuyos movimientos no pueden superar el valor de la clave. Se leerán líneas de cada archivo y se compararán, guardando el archivo principal el número menor. Ambos movimientos deben llegar al valor de la clave para asumir que se recorrieron y compararon todos los números, hecho por el cual realizamos los dos últimos bucles while. De esta manera quedarán sublistas ordenadas de tamaño $clave * 2$ en el archivo principal. Como se especificó en el

método de mezcla directa, este proceso se realizará mientras la clave sea menor al tamaño del archivo.

Realizamos tests para probar la eficacia del algoritmo, siendo el primero, **test_tamanio**, el que comprueba que el tamaño del archivo pre-ordenamiento sea igual al tamaño tras haber sido ordenado. El segundo, **test_ordenamiento**, verifica que el archivo fue ordenado, leyendo las líneas del archivo tras el ordenamiento y verificando que el valor leído sea mayor al valor leído en el ciclo anterior. Ambos tests se ejecutan sin errores.