Trabajo Práctico N°1 - ***Documentación de las Actividades***

Integrantes:

- Agustin Facundo Gaitan
- Candelaria Macchi Rojas

Asignatura: Algoritmos y estructuras de datos.

Fecha de Entrega: Lunes 18 de septiembre.

Introducción
El presente documento busca informar acerca de los resultados obtenidos durante la realización del primer trabajo práctico de la asignatura algoritmos y estructuras de datos, para ello, primero repasamos cuáles fueron los objetivos a realizar.

Problema 1:
La implementación de una lista doblemente enlazada
Implementar el TAD Lista doblemente enlazada utilizando la estructura de datos (ED) de nodos doblemente enlazados para almacenar elementos de cualquier tipo. Utilizar un TAD Nodo para gestionar la estructura de datos interna. La lista debe permitir las siguientes operaciones:

- Crear una lista vacía (implementar inicializador).
- Copiar la lista (implementar método “copiar”) para generar una copia profunda. La operación debe tener orden de complejidad O(n).
- Agregar un elemento en cualquier posición de la lista. La posición debe ser válida.
- Eliminar un elemento en cualquier posición de la lista. La posición debe ser válida. Para la eliminación en los extremos de la lista, la operación debe tener orden de complejidad O(1).
- Invertir el orden de los elementos de la lista.
- Iterar sobre la lista.
- Ordenar de “menor a mayor” los elementos de la lista.
- Concatenar dos listas con el operador ‘+’ (suma).

Su clase “ListaDobleEnlazada” debe pasar el programa de testing provisto por la cátedra.
Aclaraciones:

- 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.
- El algoritmo de ordenamiento del método “ordenar” de la Lista debe tener la eficiencia del algoritmo de ordenamiento por inserción, o mejor.

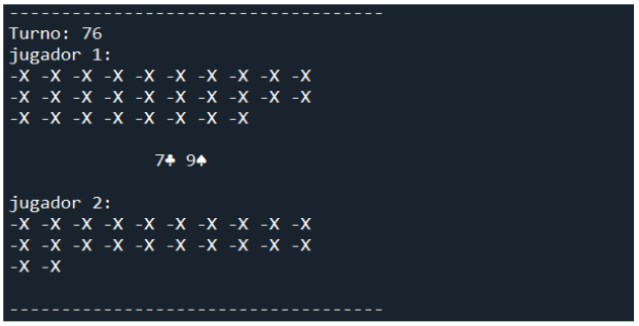
Recuerde respetar la siguiente terminología:

```
esta_vacia(): Devuelve True si la lista está vacía.
tamanio(): Devuelve el número de ítems de la lista.
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
```

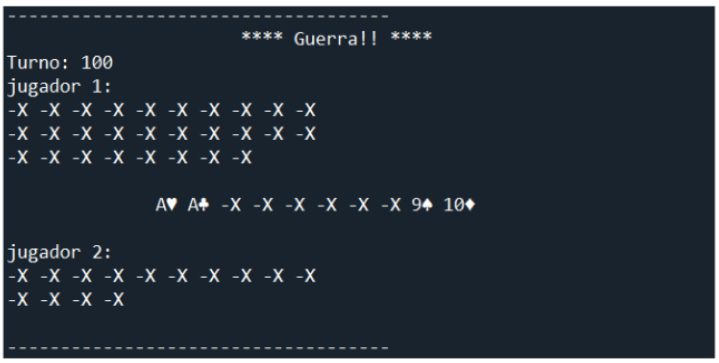
se pasa como argumento, el ítem debe añadirse al final de la lista. "posicion" es un entero que indica la posición en la lista donde se va a insertar el nuevo elemento.
extraer(posicion): elimina y devuelve el ítem en "posición". Si no se indica el parámetro posición, se elimina y devuelve el último elemento de la lista.
copiar(): Realiza una copia de la lista elemento a elemento y devuelve la copia.
invertir(): Invierte el orden de los elementos de la lista.
ordenar(): Ordena los elementos de la lista de "menor a mayor".
concatenar(Lista): Recibe una lista como argumento y retorna la lista actual con la lista pasada como parámetro concatenada al final de la primera. Esta operación también debe ser posible utilizando el operador de suma '+'. Aclaración: No se deben modificar las listas.

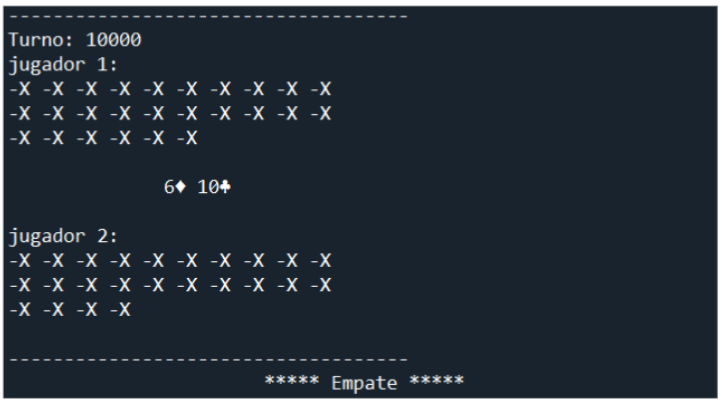
Problema 2:
El juego de cartas, Guerra

Se desea simular este juego (para 2 jugadores) para predecir en qué turno termina y cual jugador gana la partida ('jugador 1' o 'jugador 2') dado un estado inicial de los mazos de cartas de cada jugador (determinado por una semilla generadora de números aleatorios). La simulación debe mostrar la evolución de la partida por consola de la siguiente manera:



En cada turno debe mostrar las cartas de ambos jugadores. Las cartas en los mazos se deben mostrar 'boca abajo', las cartas en la mesa se deben mostrar 'boca arriba' o 'boca abajo', según corresponda.
Se debe mostrar un mensaje en cada turno donde se entra en Guerra, las cartas que se agregan en la mesa deben mostrarse con sus estados correspondientes 'boca abajo' y 'boca arriba'.





Si hubo ganador, mostrar mediante un mensaje al final de la partida.

Implemente una clase ‘JuegoGuerra’ que simule el juego de cartas, su clase debe pasar el programa de testing provisto por la cátedra. Seleccione la estructura de datos adecuada para representar los mazos de los jugadores (pila, cola, cola doble, etc.) y utilice la lista doblemente enlazada del problema anterior para implementar dicha estructura. Implemente las clases adicionales que considere necesarias para representar el juego.

Utilice las siguientes listas para representar los valores y palos de la baraja:
valores = ['2','3','4','5','6','7','8','9','10','J','Q','K','A']
palos = ['♠', '♥', '♦', '♣']
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:
El archivo de texto
Escribir un programa que genere un archivo de texto con un gran número de líneas tal que el archivo tenga un tamaño mayor o igual a 100 megabytes. En cada línea del archivo debe haber un único número entero. Los números deben estar desordenados en este archivo. Ejemplo: con números de 20 cifras y unas 5 millones de líneas se logra un archivo de unos 100 megabytes. Puede utilizarse el código en el siguiente enlace como punto de partida. Luego, codifique el algoritmo de ordenamiento externo "mezcla directa" tomando bloques de B claves en cada lectura. B es un número entero positivo menor al número de datos a ordenar que representa la cantidad de claves por bloque leído. Típicamente B es un valor tal

que el número de claves por bloque leído cabe cómodamente en memoria principal. Los bloques de B claves se ordenan luego de cada lectura de bloque, o se ordenan, antes de iniciar el proceso de mezcla directa, esto es opcional.

Ejemplo de un archivo donde N es 8 y B es 3. Notar que, en este caso, cada bloque ha sido ordenado previo al inicio del proceso de ordenamiento externo, y que el último bloque queda incompleto:

4	58	90	15	22	30	7	9

Posteriormente, aplique el algoritmo indicado al archivo previamente creado para ordenarlo de menor a mayor de manera tal que la primera línea del archivo resultante contenga el menor número entero y la última el mayor.
Finalmente, escribir una prueba que verifique el funcionamiento del algoritmo; esto es, que el archivo resultante posea el mismo tamaño (en bytes) que el archivo original y que los datos en su interior están realmente ordenados de menor a mayor luego de aplicar el algoritmo.

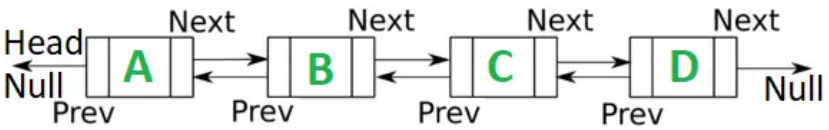
Desarrollo de las actividades.

Problema 1:

Inicialmente, vamos a definir lo que será nuestra **lista doblemente enlazada o LDE**, es conveniente para resolver este primer paso partir de algo que ya conocemos, y eso es **una lista simplemente enlazada**, la cual es una secuencia de datos (los cuales no están necesariamente contiguos en memoria) que son albergados por lo que se llama **nodo**, un nodo es consciente de la existencia del nodo siguiente, y , por lo tanto, en donde se ubica exactamente, de forma abstracta decimos que existe un **enlace** que los relaciona.

Lista simplemente enlazada.

Extrapolando, una lista doblemente enlazada va a ser una en la que no solo tengamos ese único enlace, sino que además vamos a contar con un segundo que apunte a la izquierda del nodo subsiguiente, es decir:



Un nodo es consciente de la existencia del nodo que le procede y el que le antecede, por lo tanto, nosotros definimos nuestra clase LDE de la siguiente manera:

Atributos:

- **Cabeza de la lista. (referencia al primer nodo)**
- **Cola de la lista. (referencia al último nodo)**
- **tamaño de la lista (inicialmente cero)**

```
class ListaDobleEnlazada:

    def __init__(self):
        self.cabeza = None
        self.cola = None
        self.tamanio = 0
```

Pero tambien hacemos uso de una clase “Nodo”, que van a ser los que componen la LDE, la definimos de la siguiente manera:

Atributos:

- **Ítem que posee el nodo**
- **Referencia al nodo siguiente**
- **Referencia al nodo anterior**

```
class Nodo:
    def __init__(self, dato):

        self.dato = dato
        self.siguiente = None
        self.anterior = None
```

Ahora, procederemos a dar una breve explicación de cada una de las tareas que se nos está pidiendo implementar y algunas que agregamos nosotros:

def esta_vacia(self): Este método chequea si la referencia al primer nodo, que es la “cabeza”, existe, de ser así, arroja un valor booleano informando V o F.

def tamanio(self): Cada vez que nosotros agregamos o quitamos algún ítem de la lista, modificamos su tamaño en el método correspondiente, por lo tanto el valor del tamaño se va actualizando y podemos consultarlo llamando al método. Cabe mencionar que la variable tamaño nosotros la tenemos como un atributo de la lista.

def agregar_al_inicio(self,item): Inicialmente, contemplamos el caso en que la lista está vacía, llamamos al método esta_vacia e insertamos el ítem en la cabeza de la misma en caso de que arroje el valor true.
Caso contrario, la idea es crear un nuevo nodo que va a ser la nueva cabeza, simplemente creamos el nodo y una referencia a él, es decir hacemos que la que era antes la cabeza apunte ahora al nuevo nodo que va a estar a su izquierda. Y finalmente como el tamaño de la lista aumentó en uno, incrementamos la variable tamaño.

def agregar_al_final(self,item): Agregar al final es muy parecido, también tenemos el control que verifica si la lista está vacía, de ser así ahora la cola de la lista va a ser el único nodo (podríamos haber puesto la cabeza también porque la cabeza coincide con la cola pero, a efectos de comprensión del código dijimos que sera la cola). Por otro lado si la lista no está vacía entonces procedemos como antes, solo que ahora la referencia al nuevo nodo va a estar en la cola y no en la cabeza.

def insertar(self,item,posicion): Para este método, vamos a recibir lo que es el ítem a insertar y la posición donde lo queremos insertar, entonces lo que queremos hacer es crear un nuevo nodo que contenga ese ítem. Si la lista está vacía, no importa la posición que el usuario ingrese, siempre vamos a colocar el nuevo item al inicio, en la cabeza, por lo que en ese caso simplemente llamamos a agregar_al_inicio y le pasamos el ítem, para insertar en la cabeza. Si en cambio se da que la posición en donde quiero insertar el nuevo nodo es la última, es decir, si la posición coincide con el tamaño de la lista +1, entonces llamamos a agregar_al_final e insertamos el nuevo nodo al final, estos casos son los más simples.
Ahora si la posición a insertar está en el medio por ejemplo, lo primero que debemos hacer es movernos a la posición donde lo queremos colocar, luego modificar los enlaces aledaños y así crear referencias a nuestro nuevo nodo, para movernos utilizamos un ciclo for que va hasta posición -1, una vez allí, efectuamos los cambios necesarios para insertar nuestro nuevo nodo.

def extraer (self,posicion): Este método se utiliza para eliminar y devolver un elemento de la lista en una posición específica. Si no se proporciona una posición, se elimina el último elemento de la lista.

- Si la posición es None, se elimina el último elemento de la lista y se devuelve su valor.
- Si la posición es un número válido que representa una posición en la lista, se elimina el elemento en esa posición y se devuelve su valor.
- Si la posición es -1, se asume que se desea eliminar y devolver el último elemento.

extraer_primero(self): Este método se utiliza para eliminar y devolver el primer elemento de la lista. Si la lista está vacía, se genera un error.

extraer_ultimo(self): Este método se utiliza para eliminar y devolver el último elemento de la lista. Si la lista está vacía, se genera un error.

def copiar(self): Lo que hacemos en este caso es crear una nueva lista, una lista doblemente enlazada, y decimos que una variable llamada actual va a ser la cabeza, entonces, mientras actual sea distinto de None, es decir mientras no estemos en el final de la lista, vamos a agregar al final de la nueva lista, luego, actualizamos el valor de actual para irnos moviendo en nuestra lista digital, y vamos repitiendo hasta crear toda la copia.

def invertir(self): Inicialmente nos copiamos en un nodo auxiliar la cabeza de la lista “cabeza_aux”, el cambio que hacemos después es intercambiar la cabeza por la cola, y luego para irnos moviendo hacemos un ciclo while, donde decimos que sí

mientras el siguiente de la cabeza (ahora actualizada), es distinto del valor de cabeza aux, (básicamente estamos diciendo que, “mientras no nos encontremos con la cola de la lista invertida”), el siguiente a la cabeza va a ser el anterior de la cola invertida, y luego le resto uno a self.cola.anterior, para que así la próxima vez que haga la asignación, self.cabeza.siguiente ahora va a ser el anterior a self.cola.anterior y así sucesivamente, al final, lo que obtenemos es la lista invertida.

def ordenar(self): Este método implementa el algoritmo de ordenación por inserción para ordenar la lista en orden ascendente. Compara cada elemento con los elementos anteriores y los reorganiza en su lugar adecuado.

def concatenar (self,Lista): Este método se utiliza para concatenar otra lista doblemente enlazada (otra_lista) al final de la lista actual. Ambas listas se fusionan en una sola. El método concatenar toma otra lista como argumento y agrega todos sus elementos al final de la lista actual. La lista original (otra_lista) no se modifica en el proceso.

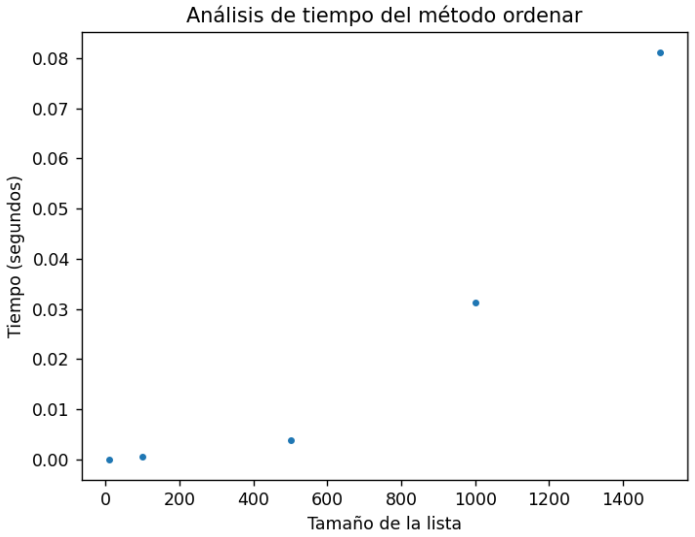
__iter__(self): Permite iterar sobre los elementos de la lista.

__add__(self, otra_lista): Permite sumar dos listas doblemente enlazadas, creando una nueva lista que es la concatenación de ambas.

__str__(self): Devuelve una representación en cadena de la lista para facilitar la visualización.

__len__(self): Devuelve el tamaño de la lista (número de elementos).

Adjuntamos la gráfica de eficiencia del algoritmo de ordenamiento por inserción, podemos comprobar que, efectivamente, por la morfología de la gráfica, este algoritmo posee un orden de complejidad O(n^2)



Problema 2:

Para resolver este problema, fuimos planteando distintas clases para modelar el juego de cartas, principalmente, tenemos 3, las cuales son:

- **Mazo**

Para representar el mazo, utilizamos un TAD que es la cola doble, llegamos a la conclusión de que es la mejor estrategia a seguir debido a que cuando nosotros agregamos/quitamos elementos del mazo durante el juego, lo podemos hacer de tanto del frente del mazo como del final del mazo, sin embargo al mismo tiempo ignoramos el hecho de que podemos quitar elementos del frente del mazo porque durante el juego nunca se da el caso, por eso los métodos de la clase mazo solo incluyen operaciones que son:

- Agregar/Quitar del final = poner_arriba/sacar_arriba.
- Agregar al frente. = poner_abajo

También tenemos un método que determinar si el mazo no tiene cartas porque así sabremos cuando un jugador ganó la partida, y uno para conocer el tamaño del mazo en un momento determinado de la partida.
__str__(self): Este método devuelve una representación en cadena del mazo. Utiliza una comprensión de lista para crear una cadena que contiene todas las cartas en el mazo.

- **Carta**

Una carta que pertenece al mazo tiene como atributos:

1) Palo
2) Número
3) Además de estos dos atributos, la clase Carta tiene un tercer atributo llamado dato, que es una cadena que combina el valor y el palo de la carta para representar la carta completa. Esto se hace en el constructor __init__ de la clase

La clase también implementa un método __str__ que devuelve una representación en cadena de la carta.

- **JuegoGuerra**

Creación del Mazo: Se crea una instancia de la clase JuegoGuerra.
Se crea un mazo de cartas estándar (52 cartas) y se mezcla utilizando random.shuffle.Se divide el mazo en dos partes iguales, una para cada jugador. Cada jugador recibe 26 cartas.

Método jugar_turno: En cada turno, se comparan las cartas en la parte superior del mazo de cada jugador. Se muestra el estado actual de los mazos de ambos jugadores, ocultando el número de cartas en cada mazo con "X". Se muestra la carta de cada jugador que se está comparando. Se compara el valor de las cartas (basado en la posición en la lista valores). El jugador con la carta de mayor valor gana las dos cartas y las coloca en la parte inferior de su mazo. Si las cartas tienen el mismo valor, se inicia una "Guerra".

Método jugar_guerra: En una "Guerra", se toman tres cartas de cada jugador y se comparan las terceras cartas de ambos. Las cartas de "Guerra" se muestran, al igual que en el turno regular. El jugador con la carta de mayor valor gana todas las cartas de la "Guerra" y las coloca en la parte inferior de su mazo.

Método jugar: El juego continúa en un bucle mientras el número de turnos sea menor a 10000. Se llama al método jugar_turno en cada iteración del bucle.
Si en algún momento un jugador se queda sin cartas en su mazo, se muestra un mensaje indicando al ganador y se termina el juego.
Si se alcanzan los 10000 turnos sin un ganador, se declara un empate.

Muestra del Resultado:
Al final del juego, se muestra el resultado (ganador o empate).

Problema 3:

Para este problema, vamos a tener que modificar el programa python provisto por la cátedra para que este genere archivos .txt que no superen los 100MB, de manera que cada línea del archivo posea solamente un número entero (vamos a hacer que todos los demás números en las líneas sean flotantes), luego, es necesario implementar el algoritmo “mezcla directa” para poder ordenar este archivo de menor a mayor, teniendo en cuenta para ordenarlas el único número entero que aparece en cada línea, de forma que la primer línea tiene el entero más pequeño, la segunda línea el mayor que le sigue a ese, y así sucesivamente. Una vez logrado este ordenamiento, se nos pide codificar un programa testing para verificar si el tamaño del archivo generado luego del ordenamiento coincide con el tamaño del archivo original, y si efectivamente nuestro algoritmo de ordenamiento funciona. A continuación, daremos una breve explicación de las soluciones:

generador_de_datos_100MB: Se importan varios módulos necesarios para el funcionamiento del código, incluyendo **randint** y **uniform** del módulo random para generar números aleatorios, **os** para interactuar con el sistema operativo y **random** para permutar líneas de números aleatorios. Primero, en orden de una mejor visualización del archivo para saber cuando comienza una línea o termina otra, limitamos la cantidad de dígitos en los números. Definimos el tamaño del bloque, que representa la cantidad de valores que se escribirán en cada bloque y el tamaño máximo del archivo (100 MB), además N es la cantidad total de valores que se escribirán en el archivo y F la cantidad de líneas. Luego de generar el archivo con estas características, se obtiene el tamaño actual del archivo “nombre” usando os.path.getsize() y se actualiza tam_actual, si el tamaño actual del archivo excede tam_max, se abre el archivo en modo de lectura y escritura binaria ('rb+') y se reduce su tamaño al valor máximo permitido (tam_max) utilizando archivo.truncate(), Finalmente, se llama a la función crear_archivo_de_datos con el nombre de archivo 'datos.txt'.

mezcla_directa: Esta función toma dos argumentos: nombre (el nombre del archivo de entrada que se va a ordenar) y B (el tamaño máximo de cada bloque durante la ordenación). Se define el nombre del archivo de salida ordenado como 'ordenado.txt'.
Se llama a la función dividir_archivo(nombre, B) para dividir el archivo de entrada en bloques de tamaño máximo B.
Luego, cada bloque se ordena utilizando el método sort de Python, utilizando una función lambda para especificar que la clave de ordenación es el número entero obtenido de cada línea mediante la función obtener_numero_entero.
Después de ordenar los bloques, se utiliza la función merge_blocks para combinarlos en un solo archivo ordenado. El archivo ordenado se escribe en 'ordenado.txt'.

- **dividir_archivo(nombre, B):** Toma dos argumentos, nombre (el nombre del archivo a dividir) y B (el tamaño máximo de cada bloque). Inicialmente, se crea una lista vacía llamada bloques para almacenar los bloques resultantes. A continuación, se abre el archivo de entrada en modo lectura ('r') y se procesa línea por línea.

Las líneas se agrupan en un bloque actual hasta que su tamaño alcance B. Cuando eso sucede, se agrega una copia del bloque actual a la lista de bloques y se limpia el bloque para comenzar uno nuevo. Si al finalizar el archivo todavía hay un bloque actual no vacío, se agrega a la lista de bloques. Finalmente, la función devuelve la lista de bloques resultante.

- **obtener_numero_entero(línea):** Esta función toma una línea del archivo como argumento y se encarga de extraer el primer número entero que encuentre en esa línea. Divide la línea en palabras usando split() y luego itera sobre las palabras. Si una palabra puede convertirse en un número entero (no genera una excepción ValueError), se devuelve ese número entero.Si ninguna palabra puede convertirse en un número entero, se devuelve 0.
- **merge_blocks(bloques):** Esta función toma una lista de bloques ordenados como entrada. Inicializa un heap (montículo) llamado heap para realizar la fusión y una lista vacía llamada merged para almacenar las líneas fusionadas en orden.
  Luego, recorre cada bloque y toma la primera línea de cada uno. Estos elementos se almacenan en el montículo heap junto con su número entero asociado (obtenido con obtener_numero_entero). A continuación, se utiliza heapq.heapify(heap) para convertir el montículo en un montículo mínimo. En el bucle principal, se extraen los elementos más pequeños del montículo y se añaden a la lista merged. Luego, se toma la siguiente línea del bloque correspondiente y se vuelve a agregar al montículo con su número entero asociado. Este proceso continúa hasta que el montículo esté vacío, lo que significa que todas las líneas han sido fusionadas y ordenadas correctamente. La función devuelve la lista de líneas fusionadas y ordenadas.

test_mezcla_directa: La función **verificar_ordenamiento** toma dos argumentos: archivo_original, que es el nombre del archivo original que se quiere verificar, y archivo_ordenado, que es el nombre del archivo que se supone que contiene los datos ordenados. Obtenemos el tamaño en bytes del archivo original y del archivo ordenado utilizando la función os.path.getsize(). Esto nos permite verificar si ambos archivos tienen el mismo tamaño. Si el tamaño es diferente, significa que el proceso de ordenamiento no se realizó correctamente. La función **obtener_numero_entero** se utiliza para extraer un número entero de una línea dada. Divide la línea en palabras, intenta convertir cada palabra en un entero y devuelve el primer entero que encuentra. Si no se encuentra ningún entero, devuelve 0. Iteramos a través de las líneas ordenadas en un bucle for. En cada iteración, llamamos a la función obtener_numero_entero(lineas_ordenadas[i]) para obtener el número entero de la línea actual y de la siguiente línea. Comparamos numero_actual con numero_siguiente. Si numero_actual es mayor que numero_siguiente, significa que el archivo no está ordenado correctamente y generamos un mensaje de error. Después de completar la iteración, si no se encontraron errores de ordenamiento, imprimimos un mensaje indicando que el archivo está ordenado correctamente.