Trabajo Práctico N°1

Alumno: Pucheta Luciano Valentin

Materia: Algoritmo y Estructura de Datos

Docentes: Javier Eduardo Diaz Zamboni, Jordán F. Insfrán y Bruno M. Breggia

Ejercicio 1: Lista Doblemente Enlazada.

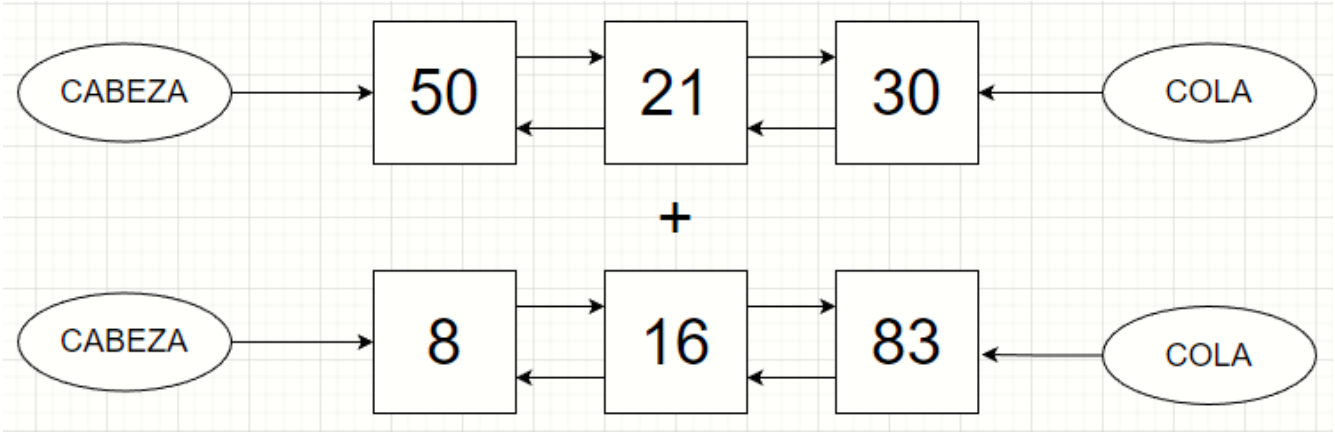
A continuación explico algunos métodos de la clase y su orden de complejidad.

Atributos del init :

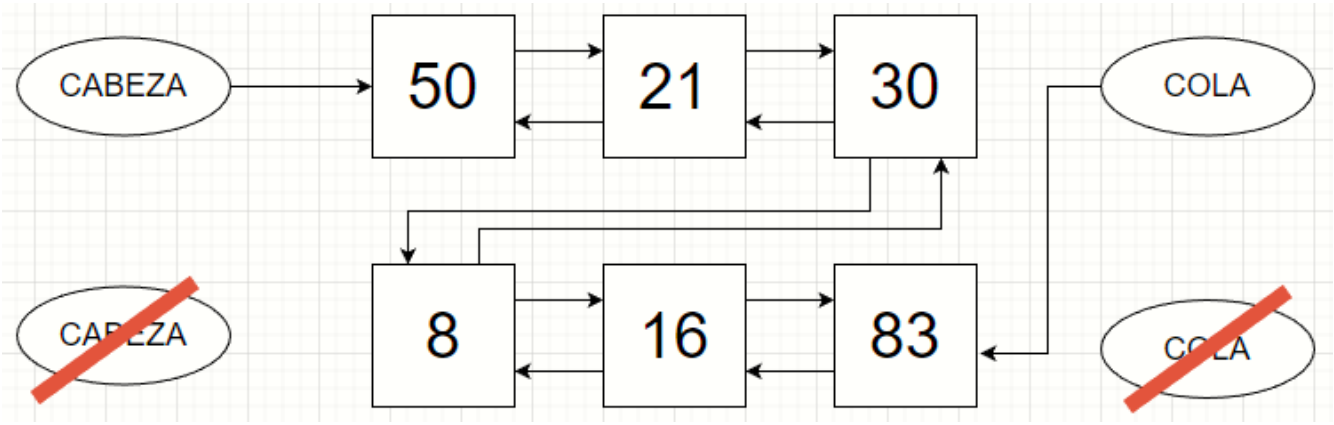
- Tamaño: registra el numero de nodos dentro de la lista
- Cabeza: registra el primer nodo de la lista
- Cola: registra el ultimo nodo en la lista

Método de concatenación O(1):

Primeramente, se crean 2 copias, una de la lista original, y otra de la que se planea concatenar a la primera, esto a fin de no modificar las listas originales en el proceso.



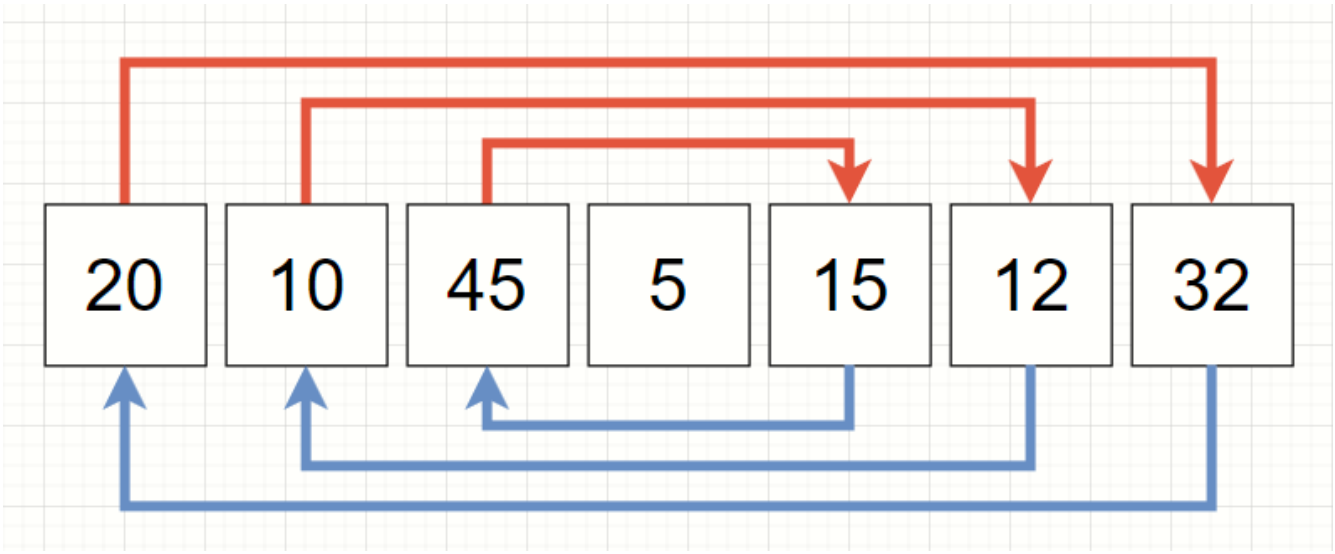
La lista resultante, se almacena en la variable asignada a la copia de la lista original, por lo que prescindimos tanto de la cola como la cabeza de la segunda lista, y enlazamos la cola de la primera con la cabeza de la segunda, seguido de esto, la nueva cola de la primera lista pasa a ser la misma que la cola de la segunda lista.



Dado que el tamaño de la lista no afecta en el algoritmo, el orden de complejidad es constante, es decir O(1)

Método invertir O(n/2):

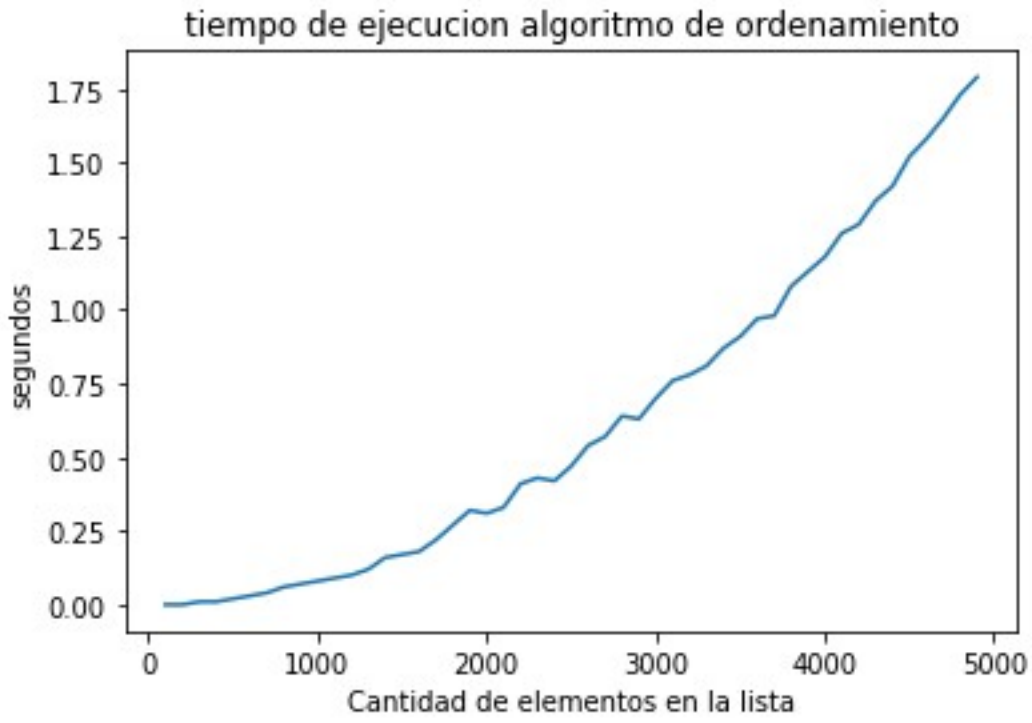
El método para invertir recorre la lista por ambos extremos, e intercambia los valores de ambos lados hasta llegar al centro, dado que el numero de intercambios o ciclos, es la mitad de la longitud de la lista (por ejemplo, para invertir una lista de 100 elementos se requieren 50 iteraciones) el orden de complejidad es O(n/2) o solamente O(n).



En este caso, el 5 queda en su misma posición.

Algoritmo de ordenamiento O(n^2):

La clase Lista Doblemente enlazada se ordena por medio de el algoritmo de ordenamiento por inserción. El gráfico a continuación muestra los tiempos de ejecución del algoritmo de 100 hasta 5000 elementos, con un incremento de 100 entre cada medición (el código con el que se generó el gráfico se encuentra en el módulo de “main.py” del repositorio).



Análisis del algoritmo:

```
def ordenar(self):
    """ Ordena la lista utilizando el algoritmo de ordenamiento insercion"""


    #si la lista esta vacia entonces ya esta ordenada
    if self.esta_vacia():                                          # O(1)
        return


    nodo_actual =self.cabeza.siguiente                             # O(1)
    while nodo_actual is not None:                                 # O(n)
        dato = nodo_actual.dato                                    # O(n)
        nodo_comparar = nodo_actual.anterior                       # O(n)

        while nodo_comparar is not None and nodo_comparar.dato > dato:      # O(n**2)
            nodo_comparar.siguiente.dato = nodo_comparar.dato       # O(n**2)
            nodo_comparar = nodo_comparar.anterior                 # O(n**2)

        if nodo_comparar is None:                                  # O(n)
            self.cabeza.dato = dato                                # O(n)
        else:
            nodo_comparar.siguiente.dato = dato                    # O(n)

        nodo_actual = nodo_actual.siguiente                        # O(n)
```

Figura 1: Linea 145 del módulo "ListaDobleEnlazada.py"

Todo lo que esta por fuera del primer bucle while, se ejecuta una sola vez, por lo que su tiempo de ejecución es constante O(1), por dentro del primer bucle cada linea se ejecuta N veces O(n), y cada linea dentro del segundo bucle se ejecuta N veces también, pero al ser un bucle anidado

ese while también se ejecuta N veces, por lo que cada linea dentro de el se ejecuta N*N o bien $O(n^2)$.

La formula general para este caso quedaría de la siguiente manera:

$$3\,n^2 + 7n + 2$$

siendo que $3\,n^2$ es el miembro mas representativo, el tiempo de ejecución del algoritmo es **$O(n^2)$.**

Ejercicio 2: Juego De Guerra.

El módulo “guerra.py” contiene 3 clases:

- **Carta:** Modela, como su nombre lo indica, cada carta del juego, posee los atributos palo, valor, y boca arriba, ademas de métodos para mostrar las cartas y compararlas.
- **Cola Doble:** es una estructura de cola doble que se usa para modelar el mazo de cartas de cada jugador.
- **Juego de Guerra:** contiene los mazos de los jugadores, registra la cantidad de turnos y posee todos los métodos propios del juego como el duelo o la guerra. Ademas tiene 3 atributos que se utilizan como auxiliares para mostrar el progreso del juego por consola.

Método duelo O(1):

Este método es el primero y el que mas se ejecuta, se ocupa tomar una carta del mazo de cada jugador, compararlas, repartirlas al ganador, corroborar que cada jugador tiene cartas suficientes para continuar y llamar al método guerra si ocurriese un empate.

Para comparar las cartas se utiliza el atributo “valor” de la clase carta, salvo en los casos donde no hay un valor numérico almacenado en dicho atributo. En dado caso, los puntos para comparar se calculan mediante la siguiente función:

```
def puntos (valor1):
    """Calcula los puntos a una carta para su comparacion"""

    if valor1 in ['A','J','Q','K']:
        if valor1 =='A':
            valor1=14
        elif valor1== 'J':
            valor1=11
        elif valor1== 'Q':
            valor1=12
        elif valor1=='K':
            valor1=13

    else:
        valor1=int(valor1)

    return valor1
```

Figura 2: Linea 8 del módulo "guerra.py"

Si el ganador del juego se define en un duelo, la función retorna “1” o ”2” dependiendo del ganador, o bien retorna un string vacío si el juego continua.

Método Guerra:

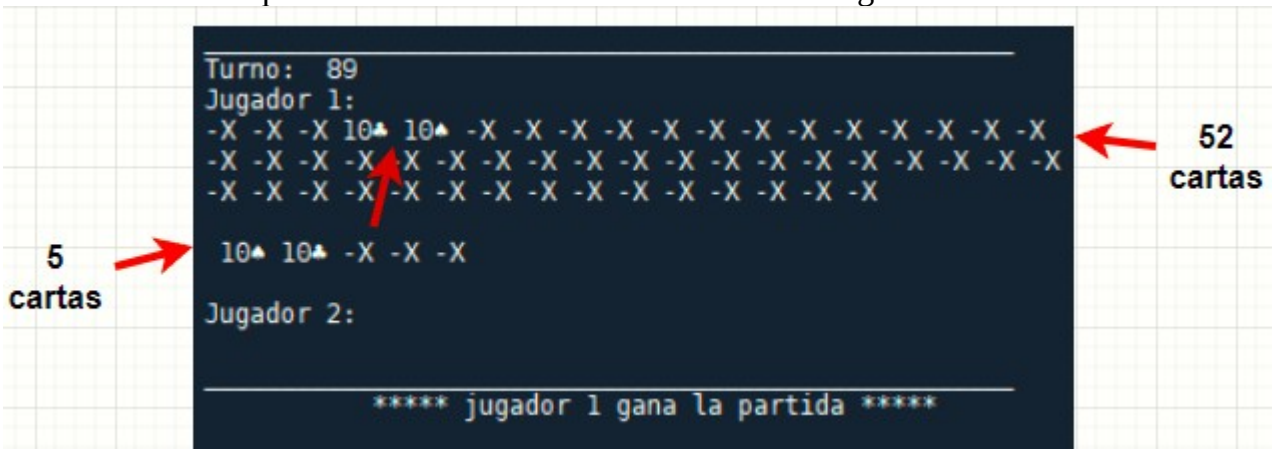
El método “guerra” es llamado por el método “duelo” cuando ocurre un empate al realizar la comparación. Se encarga de ejecutar la mecánica de guerra, corroborando que ambos jugadores tengan cartas suficientes para seguir y repartir las cartas disputadas al ganador.
Si llega a darse el caso de que al sacar las ultimas cartas para comparar, ocurriese otro empate, se llamará recursivamente al método guerra, hasta tener un ganador.

Problemas sin resolver:

Testing: Si bien modifiqué el juego de guerra para que pueda recibir una random seed al iniciarse, las partidas no se deciden en el turno que deberían. Intenté cambiar el método por el cual se mezcla el mazo antes de repartir las cartas, cambia el resultado con una misma semilla, y dado que el enunciado no especificaba cómo mezclar y repartir las cartas al inicio no estoy seguro si los fallos son por errores en el código, si no estoy mezclando correctamente las cartas, o si se están dando ambos casos.

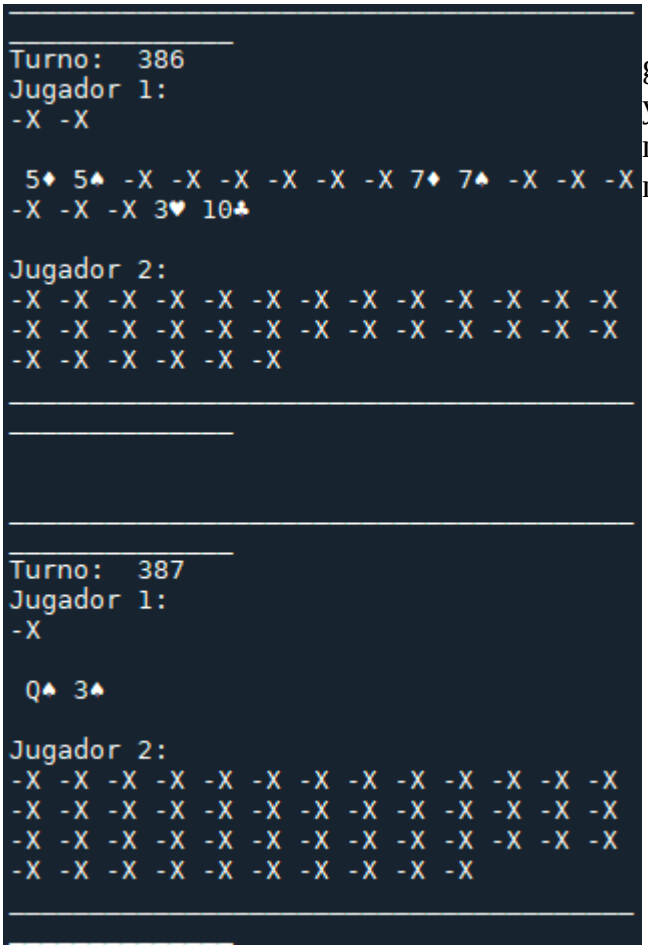
Guerra doble: Hay 2 problemas que ocurren cuando se da el llamado recursivo al método guerra:

- **Cartas extras:** Existe un condicional que comprueba después de cada turno que el número de cartas totales sea correcto, y lanza una excepción de no ser así, por lo que internamente está funcionando bien. El problema se da al mostrar por consola un turno donde ocurre una guerra doble, donde el total de cartas mostradas es superior a 52.
- **Cartas boca arriba:** Al mostrar cada duelo, las cartas de los jugadores que no se están comparando deberían estar boca abajo, pero al darse una guerra doble hay cartas que se muestran boca arriba en el mazo del ganador de esa ronda.



En la imagen se ve que el jugador 1 ganó la guerra cuando el jugador 2 se quedó sin cartas para continuar, pero hay 2 cartas boca arriba en su mazo y el total de cartas representadas en pantalla es de 57 cartas.

Sin embargo, esto último al parecer solo se da si durante la segunda guerra uno de los jugadores se quedó sin cartas, ya que aparentemente si ambos tenían cartas suficientes la guerra se muestra correctamente.



En este ejemplo, en el turno 386 se dio una guerra doble y pudo completarse hasta el final, ya en el turno 387 no hay cartas boca arriba en el mazo de ningún jugador, y en ambos turnos el numero total de cartas en pantalla es de 52.

Ejercicio 3: Ordenar archivo de texto.

Ordenamiento previo O(n):

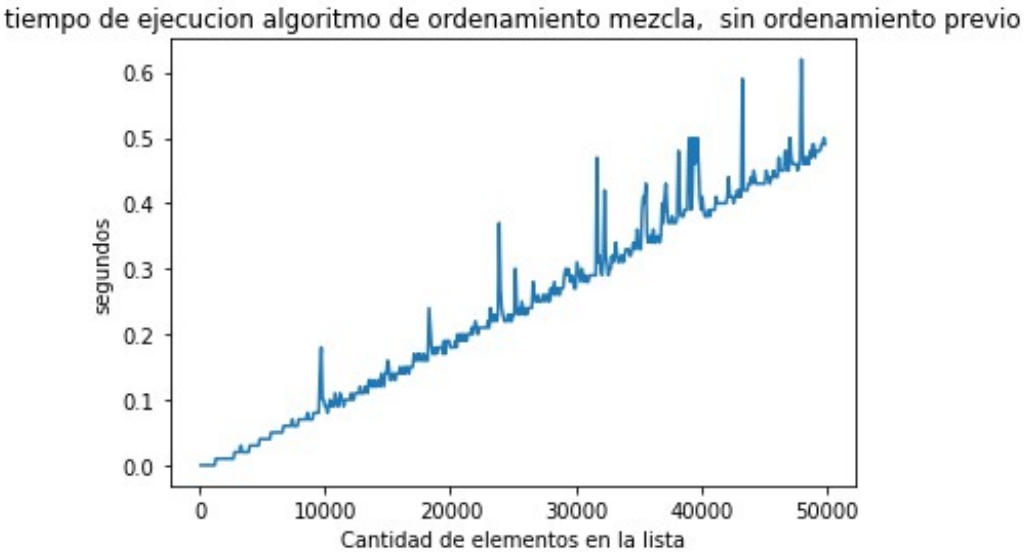
Es una función que recibe una lista y un entero B, para seguidamente ordenar cada bloque de tamaño B dentro de la lista.

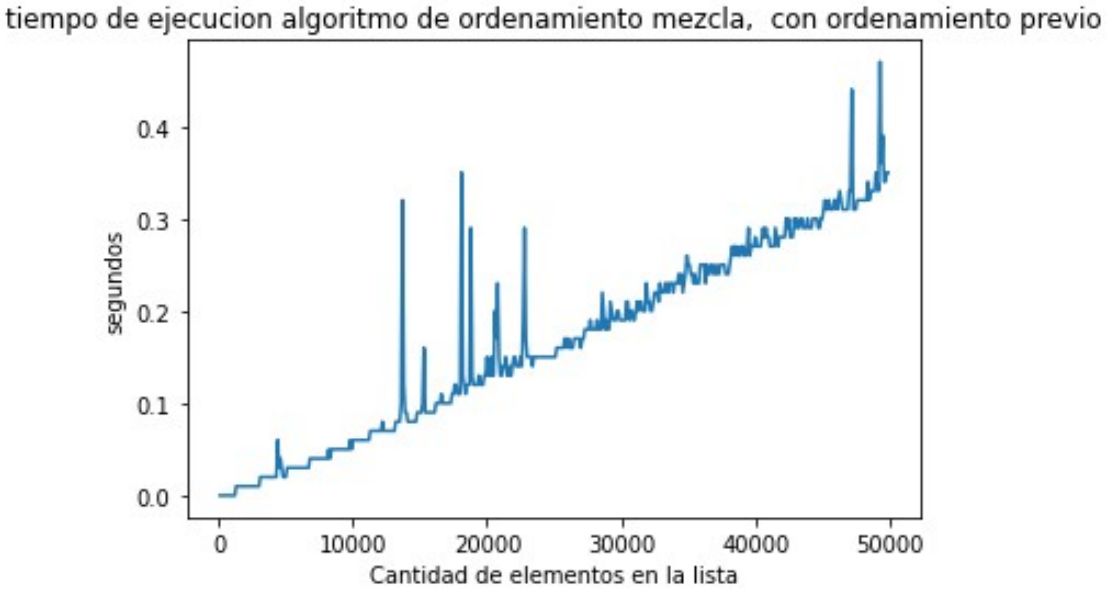
```python
def ordenamiento_previo(lista, b):
    """ordena individualmente segmentos de longitud B dentro de una lista"""

    #creo un bucle para recorrer la lista en saltos de tamaño B
    for i in range(0, len(lista), b):
        #selecciono el segmento a ordenar
        segmento = lista[i:i+b]

        #si el segmento esta completo, lo ordenamos
        if len(segmento) == b:
            segmento.sort()

        #reemplazo los valores en la lista original con los del segmento ordenado
        lista[i:i+b] = segmento
```

Figura 3: Linea 51 del módulo "ordenamiento_texto.py"

Ordenamiento merge sort O(n Log n):

Recibe una lista y la ordena por medio del algoritmo de mezcla binaria. A continuación, se muestra en los gráficos los tiempos de ejecucion para el algoritmo de ordenamiento con y sin ejecutar la funcion “ordenamiento previo”, valor B para la funcion anterior es de 100, las mediciones se hicieron de 100 a 50.000 con saltos de 100.





Si omitimos los picos mas altos de la gráfica, sin el ordenamiento previo el tiempo máximo estuvo bordeando los 0.5s mientras que con el ordenamiento previo el tiempo máximo está entre 0.3s y 0.4s, por lo que el ordenamiento previo parece mejorar ligeramente el rendimiento.

El código para generar los gráficos está en el módulo “main.py” del repositorio, el código para ordenar el archivo de texto que indicaba el enunciado está en el módulo “ordenamiento_texto.py”.

Testing para ordenamiento de texto:

El módulo de prueba para el ejercicio 3 se llama “testing_ejercicio3.py”, esta compuesto por 4 pruebas unitarias y estructurado de la siguiente manera:

- **Setup:** Se inicializa una lista de 300 elementos aleatorios entre 2^{20} y 2^{23} a la cual se le concatena una lista con 2 valores fuera de ese rango para saber de antemano el valor máximo y mínimo de la lista.
- **Test ordenamiento previo:** ejecuta el ordenamiento previo sobre la lista y luego la recorre por segmentos corroborando que cada segmento se encuentre ordenado.
- **Test mayor-menor:** ejecuta el algoritmo de ordenamiento por mezcla y comprueba que el valor mínimo este al inicio de la lista y el valor máximo se encuentre al final.
- **Test ordenamiento mezcla:** ordena la lista utilizando ordenamiento por mezcla, y recorre cada elemento de la lista para asegurarse que esté efectivamente ordenada.
- **Test tamaño del archivo:** Genera un archivo de texto con la lista desordenada del setup y luego genera otro archivo de texto con la lista ya ordenada, para seguidamente comparar que el tamaño de ambos archivos sea el mismo.