



Trabajo Práctico N° 1

ALUMNOS:

Biasoli, Ana Inés

Taborda, Veronica

Análisis de ordenamiento Burbuja

El ordenamiento Burbuja (o *Bubble Sort*) es un algoritmo de ordenamiento sencillo que compara pares de elementos adyacentes en una lista y los intercambia si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista queda completamente ordenada. Su nombre proviene del modo en que los elementos más grandes "flotan" hacia el final de la lista, como burbujas en el agua. Aunque es fácil de implementar, no es eficiente para listas grandes, ya que su complejidad en el peor caso es $O(n^2)$.

```
import random
import matplotlib.pyplot as plt
import time

def OrdenamientoBurbuja(lista1):
    for extremo in range(len(lista1)-1,0,-1):
        for i in range(extremo):
            if lista1[i]>lista1[i+1]:
                temp = lista1[i]
                lista1[i] = lista1[i+1]
                lista1[i+1] = temp
    return lista1

lista1 = [random.randint(10000,99999) for _ in range(500)]
#print("Lista 1:", lista1)
listaordenadaburb = OrdenamientoBurbuja(lista1)
#print ("lista ordenada:",listaordenadaburb)

#if __name__=="__main__":
#    #lista1 = [random.randint(10000,99999) for _ in range(500)]
#    #print("Lista 1:", lista1)
#    #listaordenadaburb = OrdenamientoBurbuja(lista1)
#    #print ("lista ordenada:",listaordenadaburb)

contador=0
ordenada = True
while contador < len(listaordenadaburb) - 1:
    if listaordenadaburb[contador] > listaordenadaburb[contador + 1]:
        ordenada = False
        break
    contador += 1
if ordenada:
    print("La lista está correctamente ordenada.")
else:
    print("La lista NO está ordenada.")
```

Este programa genera una lista de 500 números aleatorios entre 10.000 y 99.999, la ordena utilizando el algoritmo de burbuja (implementado en la función `OrdenamientoBurbuja`) y luego verifica si la lista quedó correctamente ordenada de menor a mayor. El algoritmo recorre repetidamente la lista comparando pares de elementos adyacentes e intercambiándolos si están en el orden incorrecto, lo que hace que los valores más grandes "floten" hacia el final. Después de ordenar, se recorre la lista con un bucle `while` para comprobar que cada elemento sea menor o igual al siguiente; si se encuentra un par desordenado, se marca como incorrecto y se detiene la verificación. Finalmente, el programa imprime un mensaje indicando si la lista está correctamente ordenada.

Análisis de ordenamiento por residuos

El ordenamiento por residuos (también conocido como ordenamiento por resto o ordenamiento modular) es una técnica que organiza los elementos de una lista basándose en el residuo que deja cada uno al ser dividido por un número fijo (el *módulo*). Por ejemplo, si usamos el módulo 3, agrupamos los elementos según si su residuo es 0, 1 o 2. Este método no es un algoritmo de ordenamiento clásico como Burbuja o Quicksort, pero puede utilizarse en situaciones específicas, como para clasificar datos en grupos según ciertas propiedades numéricas. Es útil cuando se busca una agrupación previa o una forma particular de ordenar basada en divisiones.

```
def ordenamiento_porconteo(lista, exp):
    n = len(lista)
    lista_datos_ordenados = [0] * n    #esta lista va a contener los
    datos ordenados
    conteo = [0] * 10    #esta lista es la que va a contar la cantidad
    de veces que aparece cada dígito en la lista original

    # Contar cuántos números tienen cada dígito en la posición 'exp'
    for i in range(n):
        indice = (lista[i] // exp) % 10
        conteo[indice] += 1

    # Acumular los conteos para saber las posiciones correctas
    for i in range(1, 10):
        conteo[i] += conteo[i - 1]

    # Construir la lista ordenada según el dígito actual
    for i in reversed(range(n)):
        indice = (lista[i] // exp) % 10
        lista_datos_ordenados[conteo[indice] - 1] = lista[i]
        conteo[indice] -= 1

    return lista_datos_ordenados
```

```
# Función principal: Ordenamiento Radix
def ordenamiento_radix(lista):
    maximo = max(lista)      # Encontrar el número más grande
    exp = 1                  # Comenzar por las unidades

    # Repetir para cada dígito (unidades, decenas, centenas, etc.)
    while maximo // exp > 0:
        lista = ordenamiento_porconteo(lista, exp)
        exp *= 10

    return lista
```

Este programa implementa el algoritmo de ordenamiento Radix Sort, que ordena una lista de números enteros comparando sus dígitos de menor a mayor (unidades, decenas, centenas, etc.). La función principal `ordenamiento_radix` comienza identificando el número más grande de la lista para determinar cuántas cifras tiene, y luego aplica repetidamente el ordenamiento por conteo (`ordenamiento_porconteo`) en cada posición decimal. Esta función auxiliar cuenta cuántas veces aparece cada dígito en la posición actual (`exp`), acumula esos conteos para determinar las posiciones correctas de los elementos, y finalmente construye una nueva lista ordenada según ese dígito. El proceso se repite aumentando la potencia de 10 hasta haber ordenado por todas las cifras del número más grande. Así, al finalizar, la lista queda completamente ordenada de menor a mayor.

Análisis de ordenamiento Quicksort

El **ordenamiento Quicksort** es un algoritmo eficiente de ordenamiento que utiliza el enfoque de **divide y vencerás**. Consiste en seleccionar un elemento llamado *pivote* y dividir la lista en dos sublistas: una con los elementos menores al pivote y otra con los mayores. Luego, se aplica recursivamente el mismo procedimiento a cada sublista. Finalmente, se combinan las sublistas ordenadas con el pivote en el medio.

```
import random
import time
import matplotlib.pyplot as plt

# Implementación de quicksort
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivote = arr[len(arr) // 2]
        menores = [x for x in arr if x < pivote]
        iguales = [x for x in arr if x == pivote]
        mayores = [x for x in arr if x > pivote]
        return quicksort(menores) + iguales + quicksort(mayores)
```

```

# Medición de tiempos para listas de tamaño 1 a 1000
tamanios = list(range(1, 1001))
tiempos_quick = []

for n in tamanios:
    lista = [random.randint(10000, 99999) for _ in range(n)]
    inicio = time.time()
    quicksort(lista)
    tiempos_quick.append(time.time() - inicio)

# Graficar los resultados
plt.figure(figsize=(10, 6))
plt.plot(tamanios, tiempos_quick, label='Quicksort', color='blue')
plt.xlabel('Tamaño de la lista')
plt.ylabel('Tiempo de ejecución (segundos)')
plt.title('Tiempo de ejecución de Quicksort (listas con números de 5 dígitos)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```

Se implementa algoritmo quicksort. Este es un algoritmo de ordenamiento que divide una lista de datos en sublistas más pequeñas, para luego ordenarlas individualmente.

```

def quicksort(arr):
    if len(arr) <= 1:           (si la lista tiene de 0a 1 elemento, ya está ordenada)
        return arr
    else:
        pivote = arr[len(arr) // 2]           (elige como pivote el elemento del medio)
        menores = [x for x in arr if x < pivote]
        iguales = [x for x in arr if x == pivote]
        mayores = [x for x in arr if x > pivote]
        return quicksort(menores) + iguales + quicksort(mayores)

```

Ordena recursivamente menores y mayores y une todo en un nuevo orden

Sobre la medición del tiempo en la ejecución

tamanios: genera una lista de tamaños de 1 a 1000 (ej: [1, 2, ..., 1000]).

tiempos_quick: lista vacía donde se guardarán los tiempos de ejecución.

```

for n in tamanios:
    lista = [random.randint(10000, 99999) for _ in range(n)]

```

```

inicio = time.time()
quicksort(lista)
tiempos_quick.append(time.time() - inicio)

```

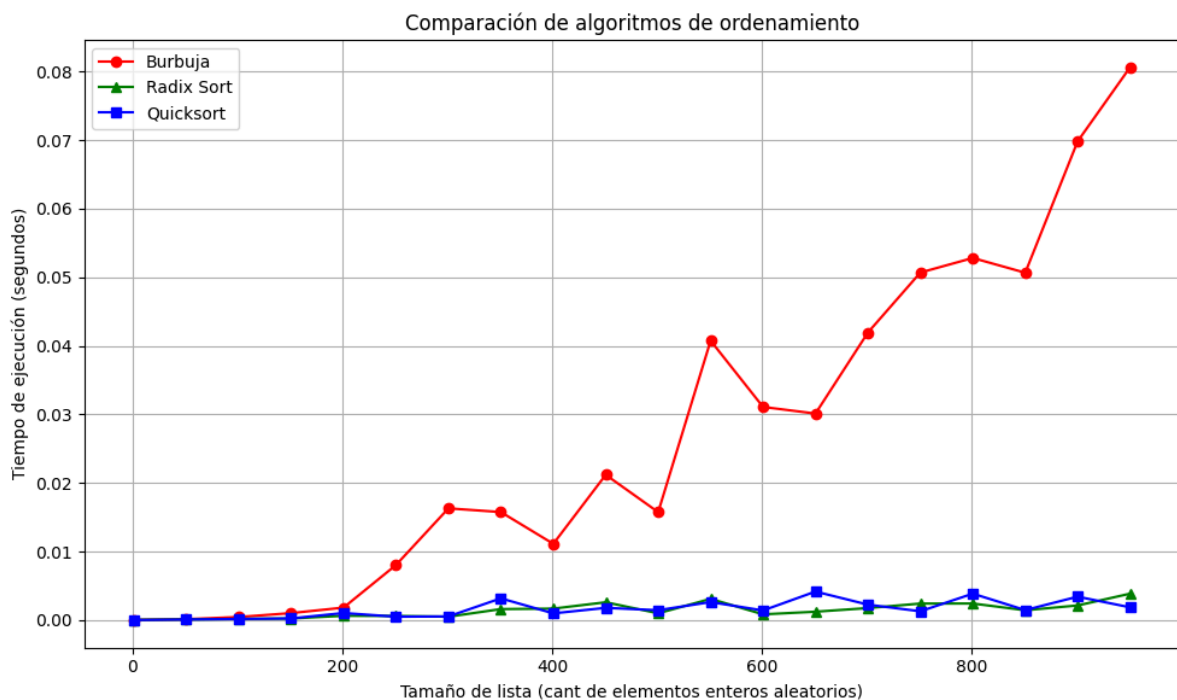
Se genera una lista con n números aleatorios de 5 dígitos

`inicio=time.time()`: guarda el tiempo justo antes de ordenar
`quicksort (lista)` ordena la lista
`tiempos_quick.append(...)`: calcula cuánto tiempo tardó y lo guarda en la lista `tiempos_quick`

Se graficó el tiempo de ejecución en función del tamaño de la lista. En la imagen puede observarse que, aunque hay cierta variabilidad, el tiempo de ejecución tiende a crecer de forma **logarítmico-lineal**, lo que es esperable para este tipo de algoritmo.

Para la misma se crea una figura de tamaño 10x6 pulgadas, se grafica el tiempo (`tiempos_quick`) en función del tamaño de la lista (`tamanios`). Finalmente, se ponen etiquetas a los ejes y se muestra la leyenda, se activa la cuadrícula del gráfico, se ajustan los márgenes y se muestra la gráfica en la pantalla.

En la siguiente imagen mostramos la gráfica que cuenta con los tres tipos de ordenamiento, los cuales varían por cada ejecución ya que son listas aleatorias.



En cada algoritmo, el orden de complejidad de O son:

- Bubble Sort: $O(n^2)$: este algoritmo compara un par de elementos adyacentes y los intercambia si están en orden incorrecto, lo que implica n^2 comparaciones en el peor de los casos. En la gráfica podemos ver como la curva roja crece de forma cuadrática, concordando con orden de complejidad O . Podemos ver que para listas de 0 a 200 elementos tiene un comportamiento similar a algoritmos como QuickSort

o RadixSort, pero siendo mayores a 200 elementos el tiempo de funcionamiento crece de forma considerable, lo que demuestra su ineficiencia en estos casos.

- Radix Sort: $O(nk)$: en este caso, k es el número de dígitos del número más grande de la lista. Este algoritmo ordena los números por dígitos: primero por las unidades, luego decenas, centenas, etc. Por cada dígito se hace una pasada completa sobre la lista, usando un algoritmo estable como Counting Sort (que es $O(n)$). Si hay k dígitos, se realizan k pasadas de $O(n)$ cada una, dando un total de $O(nk)$. Es por esto que este algoritmo es eficiente para enteros y cuando k es pequeño y constante. En la gráfica podemos observar que la curva es casi plana o crece muy poco, lo cual es consistente con un crecimiento lineal; y se ve que es muy eficiente para los tamaños de listas grandes que utilizamos.
- Quick Sort: tiene una complejidad teórica promedio de $O(n \cdot \log(n))$: este algoritmo divide la lista en 2 partes y las ordena recursivamente; eligiendo un buen pivote el número de comparaciones es cercano a $n \cdot \log(n)$, en el caso de no elegir un buen pivote el número de comparaciones es n^2 , siendo en ese caso la complejidad teórica tipo $O(n^2)$. En la gráfica podemos observar un crecimiento lento, lo que indica que a mayor cantidad de elementos este método tarda más tiempo en ordenar la lista, pero no tanto como el ordenamiento Burbuja, lo que demuestra que es más eficiente que este último pero un que Radix Sort en cuanto a tiempo es el más eficiente de todos.

La función *sorted* usa el algoritmo **Timsort**, que es una combinación de Merge Sort y de Insertion Sort. Es eficiente en datos reales, sobre todo si ya están parcialmente ordenados. Su complejidad en el mejor de los casos, que es cuando la lista se encuentra casi ordenada, es $O(n)$ y en el peor de los casos es $O(n \cdot \log(n))$.

La función *sorted* detecta patrones ya ordenados y los aprovecha con el fin de ahorrar trabajo; divide la lista en subsecuencias ordenadas y luego las fusiona eficientemente; mantiene el orden de los elementos iguales y está optimizada en C, lo que hace que funcione mucho más rápido.

Haciendo un cuadro comparativo podemos comparar complejidad, estabilidad y rendimiento de cada uno de los algoritmos.

Algoritmo	Complejidad	Estabilidad	Rendimiento
Bubble Sort	$O(n^2)$	Sí	Malo
Radix Sort	$O(nk)$	A veces	Bueno en el caso de enteros
Quick Sort	$O(n \cdot \log(n))$	No	Muy bueno
Sorted	$O(n)$ o $O(n \cdot \log(n))$	Sí	Excelente

PROBLEMA 2:

Se debe implementar un TAD **ListaDobleEnlazada** para almacenar elementos comparables. La lista debe incluir métodos para verificar si está vacía, obtener su longitud, agregar o insertar elementos, extraer elementos (con complejidad $O(1)$ en los extremos), copiar, invertir, concatenar y sumar listas. La clase debe ser eficiente en tiempo y memoria, evitando el uso de estructuras auxiliares como listas de Python. Además, se debe graficar el tiempo de ejecución de los métodos **len**, **copiar** e **invertir** en función de la cantidad de elementos y deducir su complejidad. La implementación debe pasar los tests dados por la cátedra.

Un nodo doblemente enlazado es una estructura que forma parte de una **lista doblemente enlazada**. Cada nodo guarda:

1. **Un dato** (el valor que queremos almacenar, como un número, texto, etc.).
2. **Un puntero (referencia) al nodo anterior**.
3. **Un puntero (referencia) al nodo siguiente**.

Esto le permite a la lista recorrer elementos **hacia adelante y hacia atrás**, a diferencia de una lista simplemente enlazada, que solo puede avanzar.

Módulo nodo

En principio, creamos la clase nodo

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato          # El valor almacenado (int, string, float, etc.)
        self.siguiente = None    # Referencia al nodo anterior
        self.anterior = None     # Referencia al nodo siguiente
```

Luego creamos una clase denominada ListaDobleEnlazada

```
from Nodo import Nodo

class ListaDoblementeEnlazada:
    def __init__(self):
        self.cabeza = None #referencia al primer nodo de la lista (None
al principio, porque la lista está vacía).
        self cola = None   #referencia al último nodo de la lista
(también None inicialmente).
        self.tamano = 0    #contador de elementos en la lista (inicia
en 0)
        def lista_vacia(self):
```



```

        return self.tamano == 0

    def __len__(self):
        return self.tamano # Devolvemos el tamaño actual de la lista

```

Se crea un nuevo nodo con el dato que se quiere insertar. Este nodo es una instancia de la clase **Nodo**.

```

def agregar__al__inicio(self, dato):
    nuevo_nodo = Nodo(dato)
    if self.cabeza is None:
        self.cabeza = nuevo_nodo
        self cola = nuevo_nodo #Si la lista está vacía (cabeza es
None), significa que este será el único nodo de la lista. Por lo tanto,
tanto la cabeza como la cola apuntan al mismo nodo.

    else:
        nuevo_nodo.siguiente = self.cabeza
        self.cabeza.anterior = nuevo_nodo
        self.cabeza = nuevo_nodo
        self.tamano += 1

```

Si la lista **ya tiene elementos**, entonces:

- El nuevo nodo apunta hacia el antiguo primer nodo
(nuevo_nodo.siguiente = self.cabeza).
- El antiguo primer nodo apunta hacia atrás al nuevo
(self.cabeza.anterior = nuevo_nodo).
- Luego, se actualiza la cabeza de la lista para que sea el nuevo
nodo.

Finalmente se incrementa el contador de elementos de la lista

```

def agregar__al__final(self, dato):
    nuevo_nodo = Nodo(dato)
    if self.lista_vacia():
        self.cabeza = nuevo_nodo
        self.cola = nuevo_nodo
    else:

```

```

        nuevo_nodo.anterior = self cola
        self.cola.siguiete = nuevo_nodo
        self.cola = nuevo_nodo
        self.tamano += 1

    def insertar(self, dato, posicion): #inserta un nodo en una posicion
especifica
        if posicion < 0 or posicion > self.tamano:
            raise Exception("Posición inválida") #controla que posicion
este dentro de los limites validos
        if posicion == 0:
            self.agregar__al__inicio(dato)
            return

        elif posicion == self.tamano:
            self.agregar__al__final(dato)
            return

        nuevo_nodo=Node(dato)
        nodo_actual = self.cabeza
        for _ in range(posicion - 1): # Recorremos hasta la posición
anterior
            nodo_actual = nodo_actual.siguiete

        # Ajuste de punteros para insertar en medio
        nuevo_nodo.siguiete = nodo_actual.siguiete
        nuevo_nodo.anterior = nodo_actual

        if nodo_actual.siguiete: # Si existe un nodo después, lo
enlazamos correctamente
            nodo_actual.siguiete.anterior = nuevo_nodo
            nodo_actual.siguiete = nuevo_nodo
            self.tamano += 1 # Aumentamos el tamaño de la lista

    def __iter__(self):

        self._nodo_actual = self.cabeza # Iniciamos desde la cabeza
        return self

    def __next__(self):
        if self._nodo_actual is None:

```

```

        raise StopIteration # Detener iteración cuando ya no hay
nodos

    dato = self._nodo_actual.dato # Devolver el dato, no el nodo
    self._nodo_actual = self._nodo_actual.siguiente # Avanzar al
siguiente nodo
    return dato

def __add__(self, otra_lista):
    if not isinstance(otra_lista, ListaDoblementeEnlazada):
        raise TypeError("Se requiere otra instancia de
ListaDoblementeEnlazada")
    nueva_lista = ListaDoblementeEnlazada()
    for lista in (self, otra_lista):
        nodo = lista.cabeza
        while nodo:
            nueva_lista.agregar__al__final(nodo.dato)
            nodo = nodo.siguiente
    return nueva_lista

def copiar(self):
    lista_copiada = ListaDoblementeEnlazada()
    nodo_actual = self.cabeza
    while nodo_actual:
        lista_copiada.agregar__al__final(nodo_actual.dato)
        nodo_actual = nodo_actual.siguiente
    return lista_copiada

def concatenar(self, otra_lista):
    if otra_lista.lista_vacia():
        return

    # Si la lista original está vacía, simplemente asignamos el
primer nodo de la otra lista
    if self.lista_vacia():
        self.cabeza = otra_lista.cabeza
        self cola = otra_lista.cola
    else:
        # Si ambas listas tienen elementos, conectamos el último
nodo de la lista original
        # al primer nodo de la otra lista

```

```

        self cola.siguiente = otra_lista.cabeza
        otra_lista.cabeza.anterior = self.cola
        self.cola = otra_lista.cola # Actualizamos el último nodo
de la lista original
        self.tamano += otra_lista.tamano # Actualizamos el tamaño
total de la lista

# Método para extraer un elemento de una posición dada
def extraer(self, posicion):
    if self.lista_vacia():
        raise Exception("La lista está vacía")

    if posicion < 0 or posicion >= self.tamano:
        raise Exception("Posición fuera de rango")
    if posicion == -1:
        posicion = self.tamano - 1

    # Si la posición es 0 (extraer el primer nodo)
    if posicion == 0:
        dato_extraido = self.cabeza.dato
        if self.cabeza.siguiente:
            self.cabeza = self.cabeza.siguiente
            self.cabeza.anterior = None
        else:
            self.cabeza = None
            self.cola = None
        self.tamano -= 1
        return dato_extraido

    # Si la posición es la última (extraer el último nodo)
    if posicion == self.tamano - 1:
        dato_extraido = self.cola.dato
        if self.cola.anterior:
            self.cola = self.cola.anterior
            self.cola.siguiente = None
        else:
            self.cabeza = None
            self.cola = None
        self.tamano -= 1
        return dato_extraido

    # Extraer un nodo intermedio
    actual = self.cabeza

```

```

        for i in range(posicion):
            actual = actual.siguiente

        dato_extraido = actual.dato
        actual.anterior.siguiente = actual.siguiente
        actual.siguiente.anterior = actual.anterior
        self.tamano -= 1
        return dato_extraido

    def invertir(self):
        if self.cabeza is None or self.cabeza == self cola:
            return

        actual = self.cabeza
        while actual:
            temp = actual.siguiente
            actual.siguiente = actual.anterior
            actual.anterior = temp
            actual = temp # avanzar al siguiente nodo (el original
siguiente)

        self.cabeza, self.col = self.col, self.cabeza

```

Las funciones utilizadas y otorgadas por la cátedra son las siguientes:

Método	¿Qué hace?
---------------	-------------------

<code>esta_vacia()</code>	Dice si está vacía.
---------------------------	---------------------

<code>__len__()</code>	Devuelve cuántos elementos hay (como <code>len(lista)</code>).
------------------------	---

<code>agregar_al_inicio(item)</code>	Agrega un dato al principio .
--------------------------------------	--------------------------------------

<code>agregar_al_final(item)</code>	Agrega un dato al final .
-------------------------------------	----------------------------------

`insertar(item, posicion)` Inserta un dato en una posición dada.

`extraer(posicion)` Elimina y devuelve el dato en una posición dada.

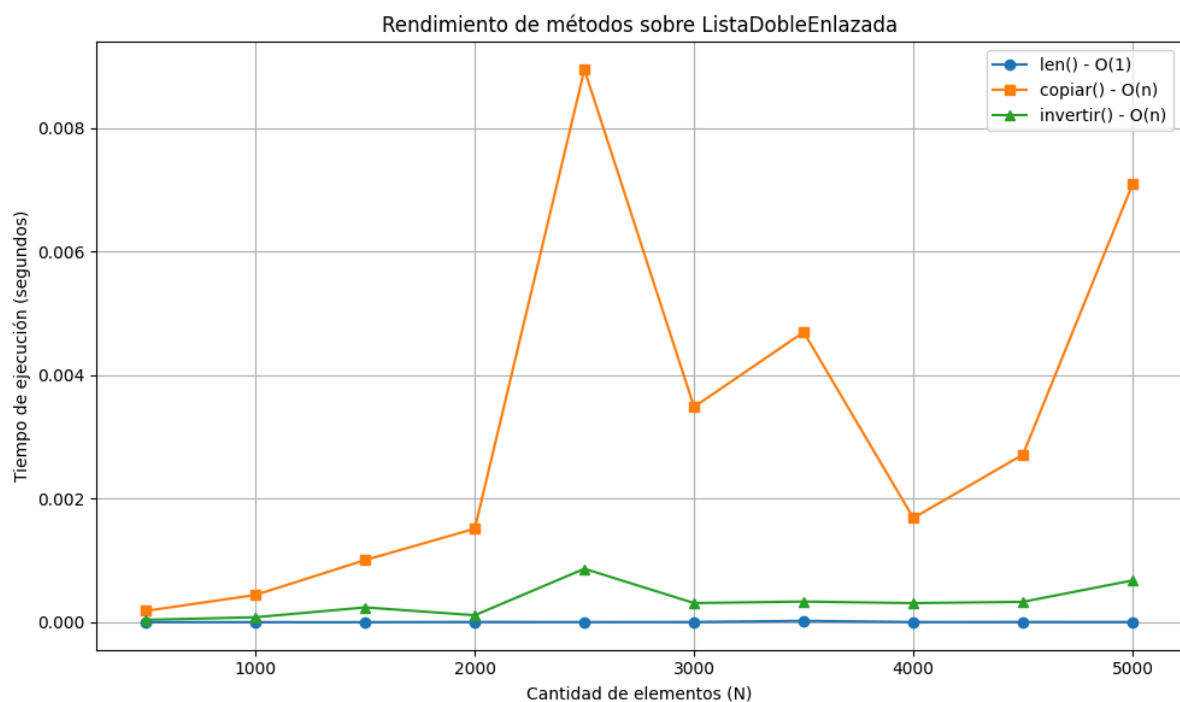
`copiar()` Devuelve una **copia nueva** de la lista.

`invertir()` Invierte el orden de los elementos.

`concatenar(otra_lista)` Junta otra lista al final de esta.

`__add__(otra_lista)` Lo mismo que concatenar, pero usando `+`.

La gráfica del problema 2 en una de sus compilaciones es la siguiente:



Observamos que:

- la curva $len() - O(1)$ es casi plana, cercana a cero. Esto podríamos interpretarlo como que el tiempo de ejecución no varía de manera significativa con el tamaño de la lista, lo que indica que este método es constante en el tiempo, lo que es típico en casos de que la lista mantenga un contador de elementos actualizado, sin necesidad de recorrerla para contarlos.
- la curva del método $copiar() - O(n)$ aumenta de manera irregular pero general con N, y muestra picos y caídas que podrían estar relacionadas al sistema; lo que se interpretaría como una complejidad lineal $O(n)$, lo que es consistente con tener que recorrer todos los nodos para copiarlos.
- la curva del método $invertir() - O(n)$ también aumenta con N pero de una manera menos “ruidosa” que copiar, por lo que podemos decir que sigue la tendencia general de crecimiento con N. Por lo que podemos concluir que este metodo tambien parece tener complejidad lineal $O(n)$, porque necesita recorrer la lista para invertir punteros

PROBLEMA 3

El juego de cartas "Guerra" es un juego por turnos entre dos jugadores que buscan quedarse con todas las cartas. Cada jugador comienza con 26 cartas tomadas de un mazo común de 52. En cada turno, ambos revelan la primera carta de su mazo; quien tenga la más alta se lleva ambas. Si hay empate, se produce una "guerra": colocan tres cartas boca abajo y una más boca arriba para desempatar; el ganador se lleva todas. El proceso se repite hasta que un jugador gana todas las cartas o el otro se queda sin cartas.

Se proporciona el código del juego, pero falta implementar la clase `Mazo`, que debe usar una lista doblemente enlazada para almacenar cartas y soportar ciertas operaciones. También debe lanzarse la excepción `DequeEmptyError` al intentar extraer una carta de un mazo vacío. Los tests del juego y de la clase deben pasar con una implementación correcta.

```
"""
Simulación del juego de cartas Guerra en un solo archivo.
"""

import random

# =====
# Clase Carta
# =====

class Carta:
    valores = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
    valor_numerico = {v: i for i, v in enumerate(valores, start=2)}
```

```

def __init__(self, valor, palo):
    self.valor = valor
    self.palo = palo

def __str__(self):
    return f"{self.valor}{self.palo}"

def __gt__(self, otra):
    return Carta.valor_numerico[self.valor] >
Carta.valor_numerico[otra.valor]

def __lt__(self, otra):
    return Carta.valor_numerico[self.valor] <
Carta.valor_numerico[otra.valor]

def __eq__(self, otra):
    return Carta.valor_numerico[self.valor] ==
Carta.valor_numerico[otra.valor]

# =====
# Excepción personalizada
# =====

class DequeEmptyError(Exception):
    pass

# =====
# Clase Nodo para la Lista Doblemente Enlazada
# =====

class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.anterior = None
        self.siguiente = None

# =====
# Clase Lista Doblemente Enlazada
# =====

class ListaDoblementeEnlazada:
    def __init__(self):
        self.primeros = None

```



```

        self.ultimo = None
        self._tamanio = 0

    def insertar_inicio(self, dato):
        nuevo = Nodo(dato)
        if not self.primeros:
            self.primeros = self.ultimo = nuevo
        else:
            nuevo.siguiente = self.primeros
            self.primeros.anterior = nuevo
            self.primeros = nuevo
        self._tamanio += 1

    def insertar_final(self, dato):
        nuevo = Nodo(dato)
        if not self.ultimo:
            self.primeros = self.ultimo = nuevo
        else:
            nuevo.anterior = self.ultimo
            self.ultimo.siguiente = nuevo
            self.ultimo = nuevo
        self._tamanio += 1

    def eliminar_inicio(self):
        if not self.primeros:
            raise DequeEmptyError("La lista está vacía")
        dato = self.primeros.dato
        self.primeros = self.primeros.siguiente
        if self.primeros:
            self.primeros.anterior = None
        else:
            self.ultimo = None
        self._tamanio -= 1
        return dato

    def __len__(self):
        return self._tamanio

    def __str__(self):
        elementos = []
        actual = self.primeros
        while actual:
            elementos.append(str(actual.dato))

```

```

        actual = actual.siguiente
    return ' '.join(elementos)

# =====
# Clase Mazo
# =====

class Mazo:
    def __init__(self):
        self.cartas = ListaDoblementeEnlazada()

    def poner_carta_arriba(self, carta):
        self.cartas.insertar_inicio(carta)

    def poner_carta_abajo(self, carta):
        self.cartas.insertar_final(carta)

    def sacar_carta_arriba(self, mostrar=False):
        if len(self.cartas) == 0:
            raise DequeEmptyError("El mazo está vacío")
        carta = self.cartas.eliminar_inicio()
        if mostrar:
            print(f"    saca {carta}")
        return carta

    def __len__(self):
        return len(self.cartas)

    def __str__(self):
        return str(self.cartas)

# =====
# Clase JuegoGuerra
# =====

N_TURNOS = 10000

class JuegoGuerra:

    valores = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
    palos = ['♠', '♥', '♦', '♣']

    def __init__(self, random_seed = 0):

```

```

        self._mazo_inicial = Mazo()
        self.mazo_1 = Mazo()
        self.mazo_2 = Mazo()
        self._guerra = False
        self._ganador = ''
        self.empate = False
        self._turno = 0
        self._cartas_en_la_mesa = []
        self._seed = random_seed

    @property
    def turnos_jugados(self):
        if self.empate:
            return N_TURNOS
        return self._turno + 1

    @property
    def ganador(self):
        return self._ganador

    def armar_mazo_inicial(self):
        random.seed(self._seed)
        cartas = [Carta(valor, palo) for valor in JuegoGuerra.valores
                  for palo in JuegoGuerra.palos]
        random.shuffle(cartas)
        for carta in cartas:
            self._mazo_inicial.poner_carta_arriba(carta)
        return self._mazo_inicial

    def repartir_cartas(self):
        while len(self._mazo_inicial):
            carta_1 = self._mazo_inicial.sacar_carta_arriba()
            self.mazo_1.poner_carta_arriba(carta_1)
            carta_2 = self._mazo_inicial.sacar_carta_arriba()
            self.mazo_2.poner_carta_arriba(carta_2)
        return self.mazo_1, self.mazo_2

    def iniciar_juego(self, ver_partida=True):
        self.armar_mazo_inicial()
        self.repartir_cartas()
        self._cartas_en_la_mesa = []
        self._turno = 0

```

[illegible]

```

        self._cartas_en_la_mesa = []
        self._guerra = False
        if len(self.mazo_1):
            self._turno += 1
        else:
            self._guerra = True
            if ver_partida:
                print('**** Guerra!! ****')

    finally:
        if self._turno == N_TURNOS:
            self.empate = True
            if ver_partida:
                print('***** Empate *****')

if self._turno != N_TURNOS and not self._ganador:
    if len(self.mazo_1):
        self._ganador = 'jugador 1'
    else:
        self._ganador = 'jugador 2'
    if ver_partida:
        print(f'***** {self._ganador} gana la partida *****')

def mostrar_juego(self):
    print(f"Turno: {self._turno+1}")
    print('jugador 1:')
    print(self.mazo_1)
    print()
    print('          ', end='')
    for carta in self._cartas_en_la_mesa:
        print(carta, end=' ')
    print('\n')
    print('jugador 2:')
    print(self.mazo_2)
    print()
    print('-----')
    if self._ganador:
        print(f'***** {self._ganador} gana la partida *****')

# =====
# Ejecutar el juego
# =====

```

```

if __name__ == "__main__":
    n = random.randint(0, 1000)
    juego = JuegoGuerra(random_seed=n)
    juego.iniciar_juego()
    print(f"Semilla usada: {n}")

```

Aqui hacemos la clase mazo

```

import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))
from LDE import ListaDoblementeEnlazada # LDE.py contiene la clase
ListaDobleEnlazada
from carta import Carta # Importa la clase Carta

#from modules.LDE import ListaDobleEnlazada # Importa la clase
ListaDobleEnlazada
import random

class DequeEmptyError(Exception):
    pass

# =====
# Clase Mazo
# =====

class Mazo:
    def __init__(self):
        self.cartas = ListaDoblementeEnlazada() #Crea un mazo que
internamente usa una lista doblemente enlazada para guardar las cartas

    def poner_carta_arriba(self, carta):
        self.cartas.insertar_inicio(carta)

    def poner_carta_abajo(self, carta):
        self.cartas.insertar_final(carta)

```

Permite agregar una carta al **inicio** o al **final** del mazo, simulando que la carta se coloca arriba o abajo del mazo.

```
def sacar_carta_arriba(self, mostrar=False):
    if len(self.cartas) == 0:
        raise DequeEmptyError("El mazo está vacío")
    carta = self.cartas.eliminar_inicio()
    if mostrar:
        print(f"saca {carta}")
    return carta
```

Si el mazo está vacío, lanza un error. Si no, elimina y retorna la carta de arriba. Si **mostrar=True**, imprime qué carta se sacó.

```
def __len__(self):
    return len(self.cartas)

def __str__(self):
    return str(self.cartas)
```

__len__: permite usar **len(mazo)** para obtener la cantidad de cartas.

__str__: devuelve una representación en texto del mazo.

[illegible]

Como podemos observar, al compilar el código se ejecuta correctamente pero no muestra los valores de los números sino que muestra puras X. Probamos haciendo algunas modificaciones en la clase Carta que fue proporcionada por la cátedra pero no pudimos solucionarlo igualmente.