Trabajo Práctico N°1

Integrantes: Gabirondo Lucas y Filchtinsky Tomas

Problema 1

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.

RESPUESTAS:

En esta primera actividad se implementaron 3 (tres) algoritmos de ordenamiento los cuales voy a enumerar y describir brevemente:

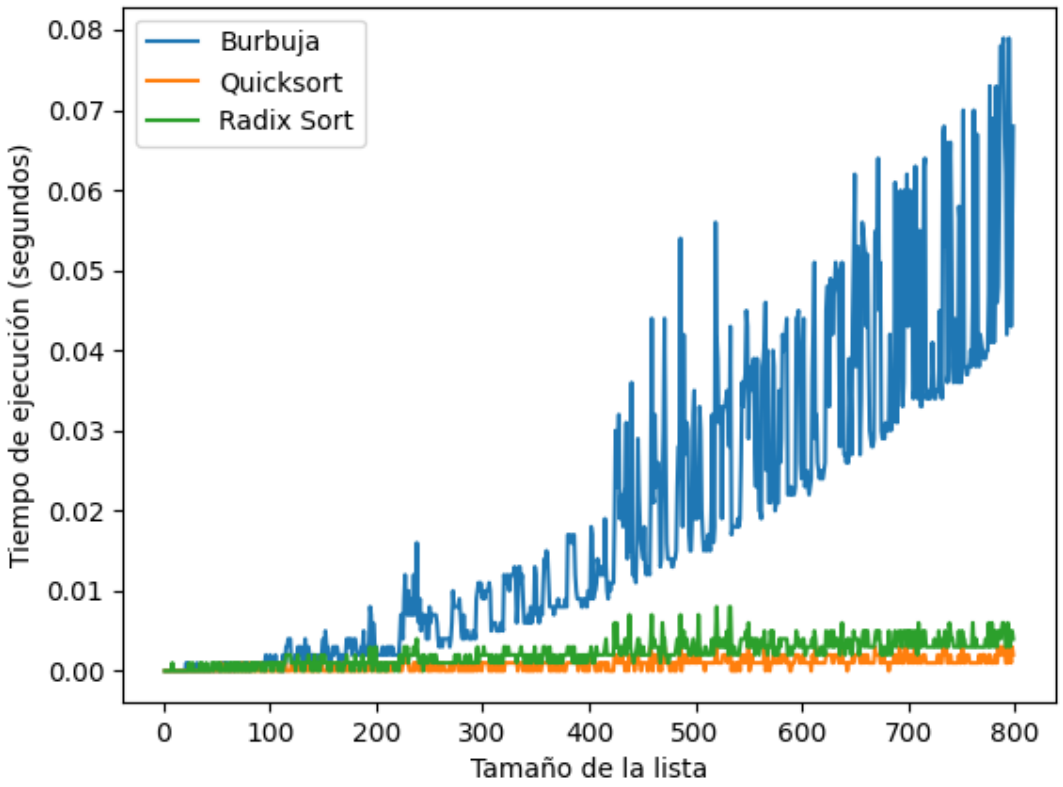
a) ***Ordenamiento Burbuja:*** En el algoritmo de ordenamiento burbuja, en cada iteración se recorre la lista realizando comparaciones entre elementos adyacentes, efectuando n−1 comparaciones por pasada, y este proceso se repite hasta que no se necesiten más intercambios. En el peor de los casos, cuando la lista está completamente desordenada, se requieren n−1 pasadas completas, lo que da lugar a un total de n(n−1)/2 comparaciones, resultando en una complejidad de O(n2). Aunque en el mejor de los casos, cuando la lista ya está ordenada, el algoritmo solo necesita una pasada para comprobar que todo está en orden, logrando una complejidad de O(n) su desempeño promedio y en el peor caso sigue siendo O(n2).

b) ***Ordenamiento Quicksort:*** En un análisis a priori del algoritmo Quicksort, la complejidad temporal promedio es O(n log n) ya que en cada nivel de recursión el algoritmo divide la lista en dos sub-listas aproximadamente iguales y realiza una operación de partición lineal O(n). Como la altura del árbol de recursión es O(log n), el tiempo total es O(n log n). En el peor de los casos, si las particiones están desequilibradas (por ejemplo, cuando el pivote es siempre el mayor o menor

elemento), el árbol de recursión alcanza una altura de O(n), lo que resulta en una complejidad de O(n2). Sin embargo, dado que el pivote generalmente se elige de manera que las particiones sean más equilibradas, la complejidad promedio es O(n log n).

c) ***Ordenamiento Radixsort:*** En un análisis a priori del algoritmo Radix Sort, su complejidad temporal es O(d·(n+k)), donde **n** es el número de elementos, **d** es el número de dígitos (o caracteres) en los datos, y **k** es el rango de los valores posibles por dígito (como 0-9 en números decimales). En cada pasada, el algoritmo realiza una ordenación estable en función de un dígito particular, lo que toma O(n+k). Como se repite este proceso **d** veces (una por cada dígito), el tiempo total es O(d·(n+k)). Este comportamiento lo hace muy eficiente para conjuntos de datos con estructura fija, como enteros o cadenas de longitud limitada, ya que **d** y **k** son constantes en muchos casos prácticos, resultando en un desempeño cercano a O(n).

A continuación, adjunto grafico de los tiempos de ejecución de tales métodos con listas de tamaño entre 1 y 800 para un mejor análisis y visualización:



Sorted(): La función **sorted()** de Python es una función **built-in** que utiliza una implementación altamente optimizada del algoritmo **Timsort**, que es una combinación de **Merge Sort** e **Insertion Sort**.
Entonces ¿Cómo funciona Sorted()?:

Para eso debemos explicar que es el **Timsort** este fue diseñado específicamente para casos de la vida real donde los datos a ordenar a menudo tienen partes ya ordenadas. Combina lo mejor de Merge Sort y Insertion Sort para manejar grandes conjuntos de datos de manera eficiente en donde el **Insertion Sort** se usa para ordenar sublistas pequeñas, ya que es rápido para listas pequeñas y casi ordenadas y por otro lado el **Merge Sort** se usa para combinar las sublistas de manera eficiente.

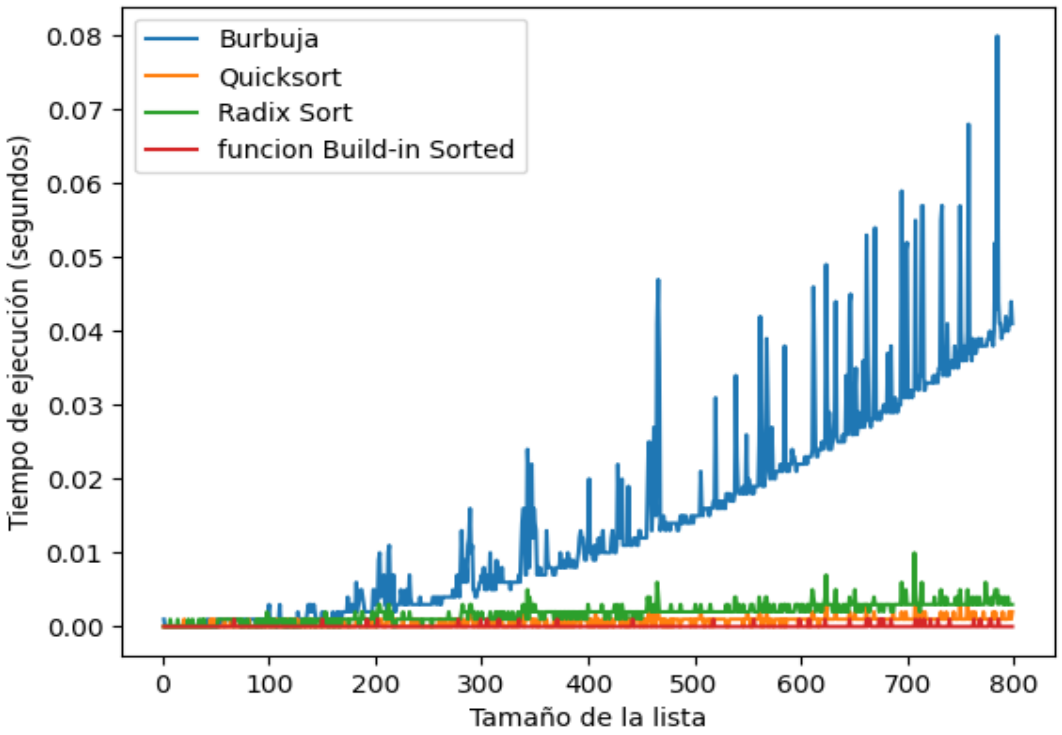
Complejidad:

Complejidad promedio y en el peor caso: O(n log n).

Mejor caso: O(n) cuando la lista está casi ordenada, ya que Timsort aprovecha esta estructura para minimizar el trabajo.

Comparación con otros algoritmos:

- **Timsort vs Quicksort**: Aunque ambos tienen una complejidad promedio de O(n log n), Timsort es más robusto en el peor caso (O(n log n) frente a O(n2) de Quicksort, y es más eficiente cuando los datos ya están parcialmente ordenados.
- **Timsort vs Radix Sort**: Radix Sort puede ser más rápido que Timsort en conjuntos de datos con enteros o cadenas de longitud fija, con una complejidad O(n) bajo ciertas condiciones. Sin embargo, Radix Sort no es general y requiere datos con estructura específica, mientras que Timsort es versátil y funciona con cualquier tipo de dato que se pueda comparar.
- **Timsort vs Bubble Sort**: Timsort es mucho más eficiente que Bubble Sort, que tiene una complejidad de O(n2) en el peor y promedio caso, mientras que Timsort garantiza O(n log n).



Problema 2

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.
__len__(): 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. 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.
__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.

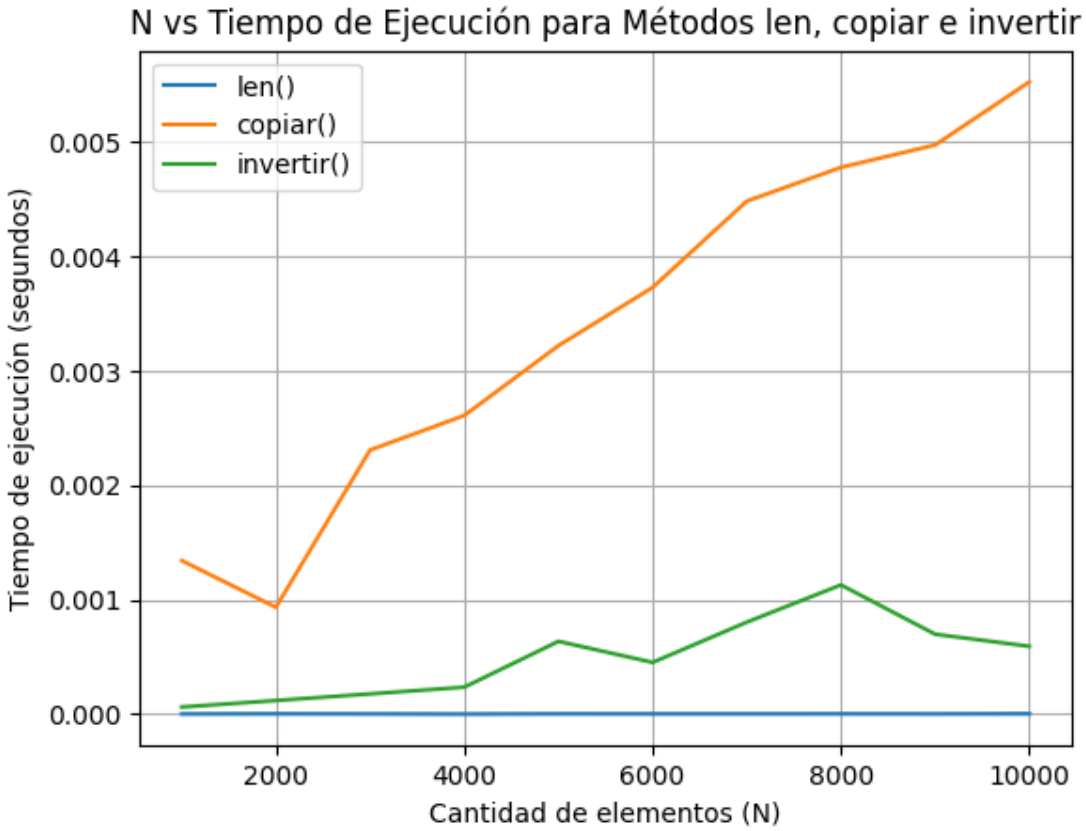
El inicializador __init__ debe crear una lista originalmente vacía.

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.

RESPUESTAS:



En la siguiente grafica podemos observar el desempeño Tiempo vs Espacio de los siguientes métodos de la LDE:

El **método len()**: Este retorna un entero expresando el número de elementos que posee la LDE, este está optimizado para que se tenga el mejor desempeño al no tener que recorrer o iterar la lista y simplemente buscando el atributo self.tamanio el cual almacena un elemento tipo int(entero) que indica el numero de elementos que posea la LDE. Este método posee una complejidad de O(1) ya que solo retorna un atributo.

```
def __len__(self):
    """Devuelve el número de ítems de la lista."""
    return self.tamanio
```

El **método copiar()**: Este método comienza creando una lista extra la cual será nuestra copia para luego a través de un ciclo recorrer la lista que queremos copiar, en cada iteración toma el dato de la lista vieja y por medio de la función agregar_al_final(item) vamos creando los nuevos nodos que integraran las lista nueva. Esta función al tener solo un ciclo y unas pocas operaciones de asignación es de Orden O(n).

```python
def copiar(self):
    """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(n2)."""
    nueva_lista = ListaDobleEnlazada()
    nodo_actual = self.cabeza

    while nodo_actual is not None:
        nueva_lista.agregar_al_final(nodo_actual.dato)
        nodo_actual = nodo_actual.siguiente

    return nueva_lista
```

El **método invertir()**: Este método funciona de manera tal que comienza asignando a una variable la referencia a la cabeza de la LDE, luego se ejecuta un ciclo while que se encargara de recorrer los nodos cambiando e invirtiendo sus referencias para que finalmente al terminar de iterar sobre la lista sus nodos esten invertidos, finalmente se cambias las referencias de la cola y cabeza y se retorna la misma lista ya invertida. Este método tiene un orden de complejidad temporal de O(n) y Espacial de O(1).

```python
def invertir(self):
    """Invierte el orden de los elementos de la lista."""
    actual_nodo = self.cabeza
    while actual_nodo is not None:
        temp = actual_nodo.siguiente
        actual_nodo.siguiente = actual_nodo.anterior
        actual_nodo.anterior = temp
        actual_nodo = temp

    self.cabeza,self.cola = self.cola,self.cabeza
    return self
```

Problema 3

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.