UNER

FACULTAD DE INGENIERÍA

BIOINGENIERÍA

Algoritmos y Estructuras de Datos

TP N°1

Aplicaciones de TADs

GRUPO:7

INTEGRANTES:

-Rios, Rodrigo Ezequiel
-Schönfeldt, Andres

COMISIÓN:2

Problema 1

Análisis de Complejidad y Funcionamiento de Algoritmos

1. Ordenes de complejidad O de cada algoritmo

- **Ordenamiento de burbuja**:O(n^2) en promedio y peor caso.
  Usa dos bucles anidados (for i in range(n) y for j in range(0, n-i-1)), lo que genera comparaciones proporcionales a n(n-1)/2.

- **Quick Sort**: O(n log n) en promedio, O(n²) en peor caso.
  Divide el arreglo recursivamente (log n niveles) y en cada nivel realiza particiones O(n). El peor caso ocurre con pivotes desbalanceados, pero es raro en implementaciones como esta (pivote central).

- **Residuos o Radix Sort**: O(dn) donde d = dígitos (5 en este código).
  Realiza (d) pasadas (una por dígito). En cada pasada usa Ordenamiento

por cuentas (Counting sort) (O(n + k), k=10 para dígitos 0-9), resultando en O(5(n + 10)) ≈ O(n).

- Python sorted():O(n log n) garantizado.
  Implementa Timsort, un algoritmo híbrido que combina Merge Sort(Ordenamiento por mezcla) de Ordenamiento por inserción (Insertion sort), optimizado para datos del mundo real.

2. Análisis a priori basado en el código

Burbuja

```
for i in range(n):  # O(n)
   for j in range(0, n-i-1):  # O(n) decreciente
      if arr[j] > arr[j+1]:  # O(1)
         swap  # O(1)
```

Total: O(n) × O(n) = O(n²).

Quick Sort

```
pivot = arr[len(arr)//2]  # O(1)
left/middle/right = [...]  # O(n) por partición
return quick_sort(left) + ...  # Llamadas recursivas log n veces
```

Total: O(n) × O(log n) = O(n log n) en promedio.

Radix Sort (radix_sort() + counting_sort())

```
while max_num//exp > 0:  # O(d) (d=5)
   counting_sort()  # O(n + 10) por pasada
```

Total: O(5(n + 10)) ≈ O(n).

3. Sorted()

Python utiliza Timsort, un algoritmo:

1. *Híbrido: Combina Merge Sort e Insertion Sort.*
2. *Adaptativo: Detecta secuencias ordenadas ("runs") para minimizar operaciones.*
3. *Estable: Mantiene el orden relativo de elementos iguales.*

4. *Optimizado: Funciona especialmente bien con datos parcialmente ordenados o con patrones reales (ej. timestamps).*

Características clave:

- *Tamaño mínimo de run: 32-64 elementos.*
- *Usa Insertion Sort para runs pequeños.*
- *Fusiona runs de forma eficiente (gestión de memoria inteligente).*
- *Complejidad espacial: O(n).*

Gráfica esperada

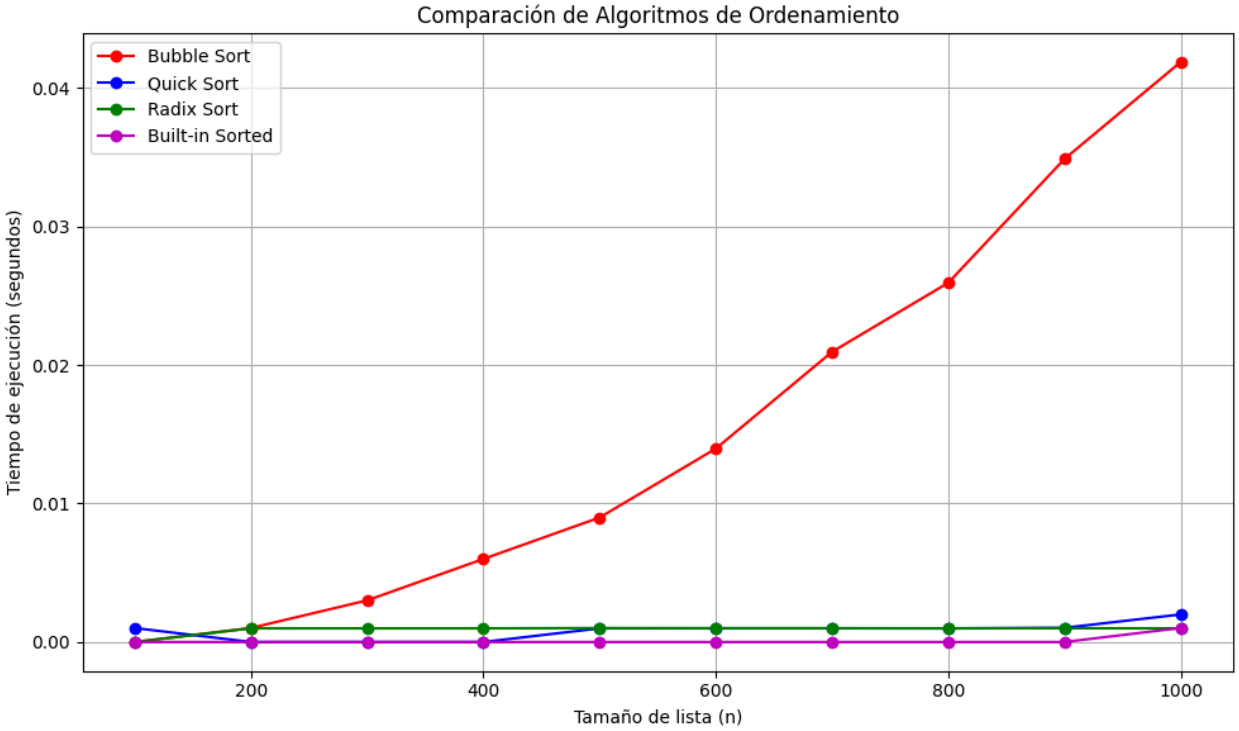
En la gráfica generada por el código:

- Bubble Sort (rojo) mostrará crecimiento cuadrático.
- Quick Sort (azul) y sorted() (morado) tendrán comportamiento cercano a lineal-logarítmico.
- Radix Sort (verde) destacará con crecimiento casi lineal, siendo el más eficiente para números de 5 dígitos.

CONCLUSIÓN

El código implementado permite comparar la eficiencia de varios algoritmos de ordenamiento midiendo sus tiempos de ejecución con listas de diferentes tamaños y visualizando los resultados en una gráfica. Esto ayuda a entender cómo se comporta cada algoritmo según la cantidad de datos.
Burbuja (línea roja) es el menos eficiente, mostrando un aumento rápido y cuadrático ($O(n^2)$) en su tiempo de ejecución. Esto lo hace útil sólo para listas pequeñas o como ejemplo educativo, ya que su rendimiento se deteriora notablemente con más elementos.
Quicksort (línea azul) ofrece un mejor equilibrio, con un comportamiento promedio cercano a $O(n \log n)$. Aunque en casos extremos (pivotes desbalanceados) puede degradarse a $O(n^2)$, en la práctica suele ser rápido y confiable para listas grandes, gracias a su enfoque recursivo de "dividir y conquistar".
Radix Sort (línea verde) es el más eficiente en este escenario, con un tiempo casi lineal ($O(n)$), ya que evita comparaciones directas y ordena dígito por dígito. Sin embargo, su aplicación está limitada a datos numéricos con dígitos predefinidos, como los números de 5 cifras usados en el código.
La función `sorted()` de Python (línea morada), que utiliza Timsort, combina lo mejor de Merge Sort e Insertion Sort. Es altamente eficiente ($O(n \log n)$) y adaptativo, detectando patrones en los datos (como secuencias parcialmente ordenadas) para optimizar el proceso. Esto lo hace ideal para datos del mundo real, donde es común encontrar información con cierto orden preexistente.

La elección del algoritmo depende del contexto: Quicksort o sorted() son ideales para listas genéricas grandes, Radix Sort es imbatible con números de dígitos fijos, y Burbuja sirve principalmente para aprendizaje. La gráfica generada refleja claramente estas diferencias, destacando la importancia de seleccionar el método adecuado según el tipo y tamaño de los datos.



Problema 2

Analisis de complejidad O de las funciones len, copia e invertir:

- __len__(): *retorna el valor de una variable (size) por lo que es O(1)*

```
def __len__(self):      #Retorna el valor de Size, variable relacionada al tamaño de la lista
    return self.size
```

-copia(): *crea una nueva lista copiando los n datos de la lista actual por lo que su complejidad es O(n)*

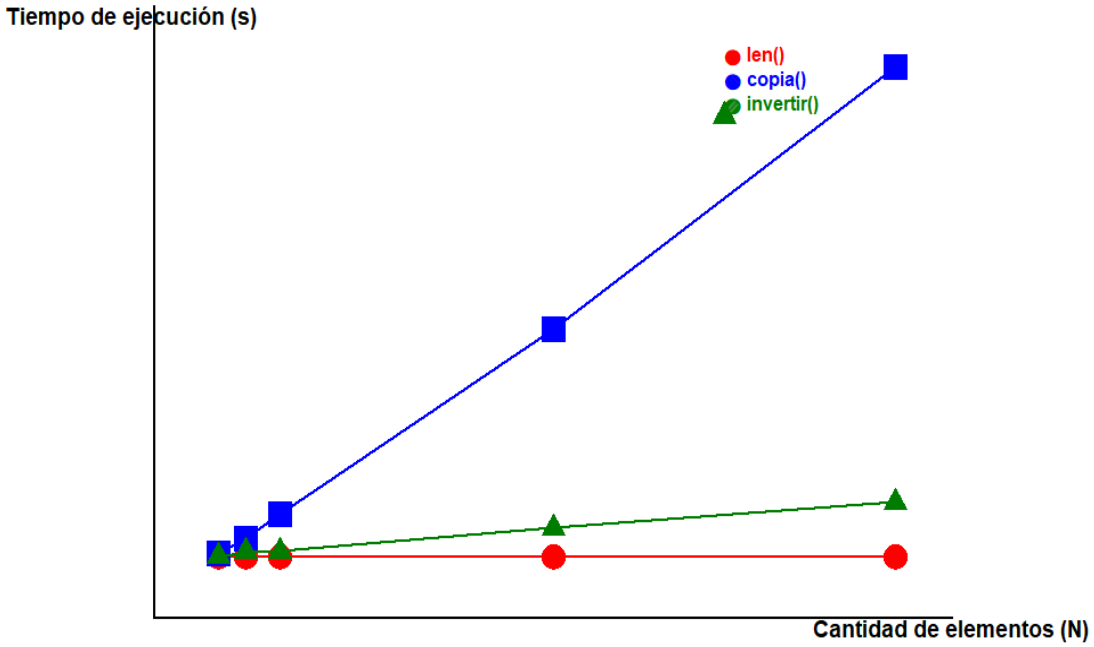
```python
def copia(self):    #Crea y retorna una copia de la lista doble enlazada
    copia = ListaDobleEnlazada()
    actual = self.primero
    while actual:
        copia.agregar_al_final(actual.valor)
        actual = actual.siguiente
    return copia
```

-Invertir(): *para poder invertir el orden de la lista se debe recorrer y modificar cada punto de los n datos de la lista por lo que también es de complejidad O(n)*

```python
def invertir(self):   #invierte el orden de la lista
    actual = self.primero
    self.primero, self.ultimo = self.ultimo, self.primero

    while actual:
        actual.siguiente, actual.anterior = actual.anterior, actual.siguiente
        actual = actual.anterior
```

<u>Gráfica de tiempo de ejecución en función de cantidad de elementos N</u>



En la gráfica podemos observar que la funcion len en efecto es O(1) ya que es constante. Por otro lado copia e invertir son funciones lineales O(n), la diferencia es el tiempo de ejecución de cada una.

El código fue testeado por un código suministrado por la cátedra (Problema_2/tests/test_Actividad2.py) y completó con éxito todas las secciones de prueba.

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.008s

OK
PS C:\Users\riosf\OneDrive\Escritorio\Algoritmo\Rodri\AyED2025c1-Schonfeldt-Rios-main\TrabajoPractico_1\Problema_2>
```

Problema 2 (Conclusión):

Se logró realizar un código que satisfaga las condiciones del trabajo práctico, pasando las pruebas suministradas por la cátedra. Analizamos la complejidad de las funciones __len__(), copia() e invertir() haciendo uso de turtle para realizar una gráfica del tiempo de

ejecución en segundos con respecto a la cantidad de elementos de la lista y llegamos a la conclusión que __len__() es una función constante por ende tiene una complejidad de O(1), por otro lado copia() e invertir() son lineales por ende complejidad O(n). La diferencia entre copia() e invertir() es la pendiente de sus rectas, lo que implica que copia() requiere más tiempo de ejecución que invertir() a medida que aumentan la cantidad de elementos de la lista.

Problema 3 (CONCLUSIÓN)

El código presentado simula el juego de cartas "Guerra", donde dos jugadores se enfrentan utilizando una baraja estándar. El objetivo es que un jugador gane todas las cartas. El juego se divide en varias etapas clave:

1. Inicialización: Se crean dos jugadores, cada uno con un mazo de 26 cartas. Los jugadores no pueden ver sus cartas.
2. Desarrollo del juego: En cada turno, los jugadores revelan la carta superior de su mazo. El jugador con la carta de mayor valor gana el turno y añade ambas cartas al final de su mazo.
3. Guerra: Si ambos jugadores revelan cartas del mismo valor, se declara una "guerra". En este caso, cada jugador pone tres cartas boca abajo y luego revela otra carta. El ganador de la guerra se lleva todas las cartas en juego. Este proceso se repite si hay empate de nuevo.
4. Finalización: El juego continúa hasta que un jugador gana todas las cartas, o si se alcanza un número máximo de turnos, resultando en un empate.

El código usa la librería **turtle** para proporcionar una representación visual del juego, mostrando las cartas, los mazos de los jugadores y otra información relevante. La librería random se utiliza para mezclar las cartas al principio del juego y para simular el azar inherente al juego. El objetivo principal del código es simular el juego de "Guerra" y proporcionar una interfaz visual interactiva.

¡Jugador 2 gana!
Jugador 1 Ganadas: 8
Jugador 2 Ganadas: 10
Turnos Jugados: 21

Jugador 1
Cartas: 20
Ganadas: 8

Jugador 2
Cartas: 30
Ganadas: 9

Turno: 21