Facultad de Ingeniería

Algoritmos y Estructuras de datos

Informe Trabajo Práctico N° 1

Integrantes: Obaid Aixa, Gomez Genara, Caraballo Mateo

Actividad 1

Durante esta actividad, implementamos y comparamos distintos algoritmos de ordenamiento para analizar cómo se comportan frente a listas de diferentes tamaños. Usamos cuatro métodos: Bubble Sort, Quicksort, Radix Sort y la función **sorted()** de Python. El objetivo era medir cuánto tardaba cada uno en ordenar listas con 10, 100, 200, 500, 700 y 1000 elementos generados aleatoriamente. Para registrar los tiempos usamos **time.perf_counter()**, que nos permitió tener una medición precisa del rendimiento. Cada algoritmo fue probado con las mismas listas para asegurar que la comparación sea justa.

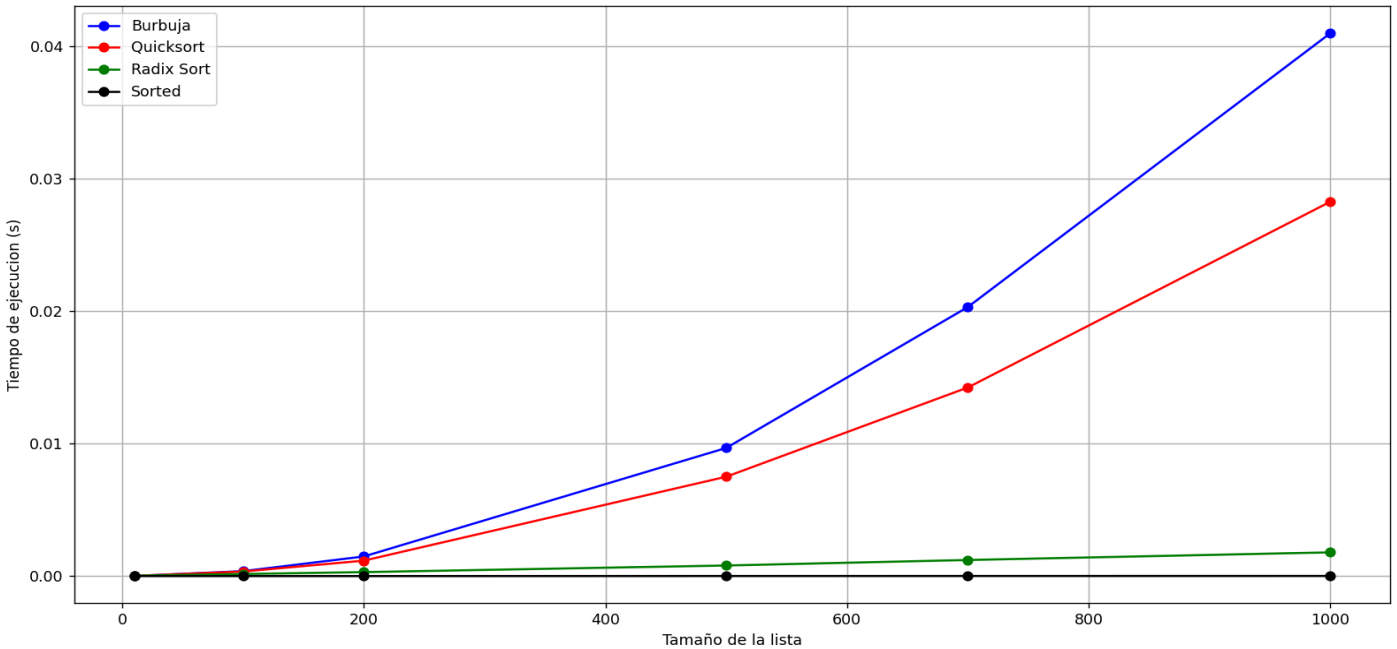
Los resultados fueron: **Bubble Sort** es el más lento, especialmente con listas más grandes, porque tiene que hacer muchas comparaciones y movimientos, lo que lo hace muy ineficiente. Este algoritmo tiene una complejidad de **$O(n^2)$**.

Quicksort, fue mucho más rápido y tiene una complejidad de **O(n log n)** en la mayoría de los casos. Aunque en el peor escenario, si se elige un mal pivote, puede ser tan lento como **$O(n^2)$**, sigue siendo bastante eficiente en general.

Radix Sort fue bueno con listas grandes de números enteros, especialmente porque tiene una complejidad de **$O(d \cdot (n+k))$**, lo que lo hace muy rápido cuando los números tienen pocos dígitos. Es ideal para ciertos tipos de datos.

La función **sorted()** de Python, fue la más rápida de todas. Utiliza **Timsort**, que es una combinación de **Merge Sort** y un algoritmo de ordenamiento por inserción adaptativo. Timsort aprovecha los patrones que pueden existir en los datos para ser más rápido en casos específicos, como listas parcialmente ordenadas, y **sorted()** no modifica la lista original, sino que devuelve una nueva lista ordenada.

Cuando comparamos los algoritmos implementados con **sorted()**, vimos que **sorted()** es generalmente más rápido que **Bubble Sort** y, en la mayoría de los casos, más eficiente que **Quicksort** debido a su adaptabilidad. Aunque **Radix Sort** puede ser superior en ciertos casos (por ejemplo, con listas de enteros con un rango fijo de dígitos), **sorted()** fue más eficiente en general para una variedad de datos.

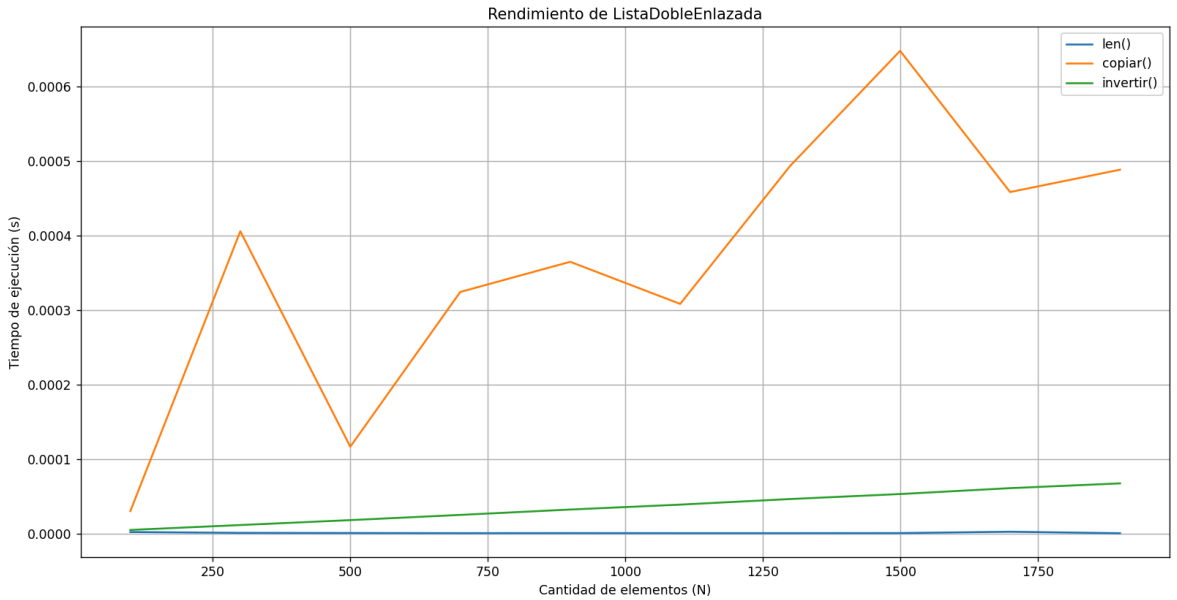


Actividad 2

Para analizar el rendimiento de los métodos **len()**, **copiar()** e **invertir()**, se ejecutaron varias pruebas midiendo los tiempos de ejecución con listas de distintos tamaños, desde 100 hasta 2000 elementos. Los tiempos fueron registrados y graficados para visualizar el comportamiento de la implementación con el aumento del tamaño de la lista.

En términos de complejidad, el método **len()** tiene un rendimiento constante, ya que solo devuelve el tamaño almacenado en el atributo privado **_tamanio**. Por otro lado, los métodos **copiar()** e **invertir()** tienen una complejidad lineal. En el caso de **copiar()**, cada elemento de la lista debe ser agregado a una nueva lista, mientras que en **invertir()**, cada nodo debe modificar sus punteros para invertir el orden de la lista.

Las gráficas muestran que tanto **copiar()** como **invertir()** tienen un tiempo de ejecución proporcional al número de elementos en la lista, lo cual confirma que ambas operaciones tienen complejidad **O(N)**. El rendimiento es eficiente para listas pequeñas, pero se hace más notorio a medida que la cantidad de elementos aumenta.



Actividad 3

Creamos la clase **Mazo** usando una Lista Doblemente Enlazada para almacenar objetos de tipo Carta y que el juego "Guerra" funcione correctamente. Esto nos permite manejar las cartas de manera eficiente.

Los métodos que implementamos en la clase son **poner_carta_arriba**, que agrega una carta al principio del mazo; **poner_carta_abajo**, que pone una carta al final del mazo; y **sacar_carta_arriba**, que extrae una carta de arriba, lanzando la excepción **DequeEmptyError** si el mazo está vacío.

También tenemos el método **esta_vacio** para saber si el mazo está vacío y **__len__**, que nos da la cantidad de cartas que quedan. Los métodos **__str__** y **__repr__** nos permiten ver el mazo de forma más amigable.

Con esto, el mazo puede hacer todo lo que necesita para el juego, manejando las cartas y asegurando que todo funcione bien durante cada turno del juego.

```python
from modules.LDE import ListaDobleEnlazada

class DequeEmptyError(Exception):
    def __init__(self, mensaje="oopps, el mazo esta vacio."):
        super().__init__(mensaje)  #se lanza la excepcion

class Mazo:
    def __init__(self):
        self._cartas = ListaDobleEnlazada()  #se crea el mazo como una lista doble enlazada

    def poner_carta_arriba(self, carta):
        """Agrega una carta al principio del mazo."""
        self._cartas.agregar_al_inicio(carta)

    def poner_carta_abajo(self, carta):
        """Agrega una carta al final del mazo."""
        self._cartas.agregar_al_final(carta)

    def sacar_carta_arriba(self, mostrar=False):
        """Saca la carta del principio del mazo. Lanza DequeEmptyError si está vacío."""
        if self.esta_vacio():
            raise DequeEmptyError()
        carta = self._cartas.extraer(0)

        if mostrar:
            carta.visible = True
        else:
            carta.visible = False
        return carta

    def esta_vacio(self):
        """Devuelve True si el mazo está vacío."""
        return len(self._cartas) == 0

    def __len__(self):
        """Devuelve la cantidad de cartas en el mazo."""
        return len(self._cartas)

    def __str__(self):
        """Devuelve una representación en cadena del mazo."""
        return str(self._cartas)

    def __repr__(self):
        return self.__str__()
```