Universidad Nacional de Entre Ríos
Facultad de Ingeniería

Carrera: Bioingeniería

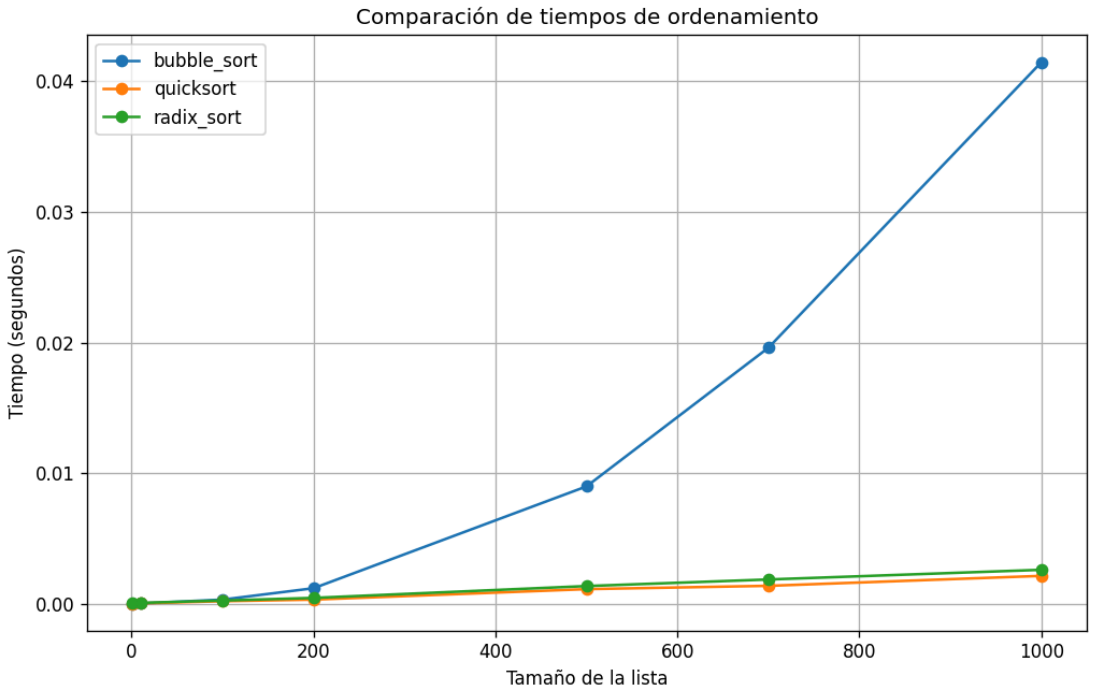
Materia: Algoritmos y Estructuras de Datos

Estudiantes: Arener Rocio, Cardoso Josefina Belén y Segovia Lucas

Trabajo Práctico N°1

Fecha de entrega: 01/05/2025

Proyecto 1
Tiempos de ejecución de tales métodos con listas de tamaño entre 1 y 1000.



Los órdenes de complejidad a priori correspondientes a cada algoritmo son:

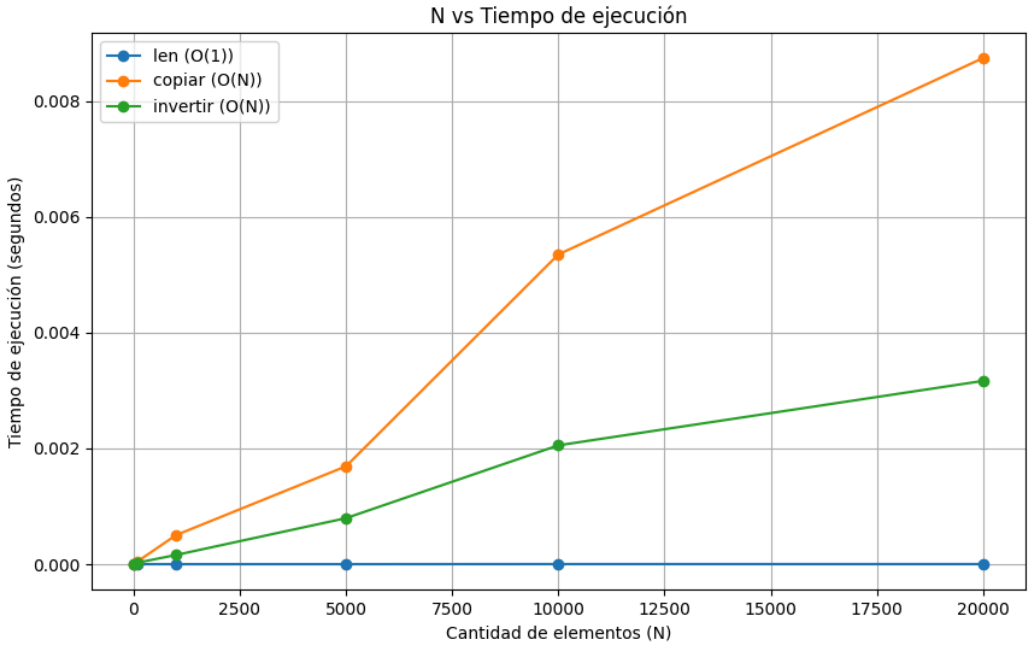
a) Ordenamiento burbuja = $O(n^2)$
b) Ordenamiento quicksort = $O(n^2)$
c) Ordenamiento radix sort = $O(n)$

Los algoritmos de ordenamiento de burbuja y quicksort son n^2 debido a que son algoritmos cuadráticos con bucles anidados, donde el tiempo de ejecución crece exponencialmente con el tamaño de la entrada; por otra parte el algoritmo de ordenamiento radix sort tiene orden de complejidad o(n) debido a que es un algoritmo lineal donde el tiempo de ejecución crece proporcionalmente al tamaño de la entrada.

Dos de los tres algoritmos de ordenamiento (bubble y radix) se comportan según lo esperado en el análisis a priori; quicksort, por otro lado, se comporta como un O(n log(n)), ya que este es el orden de complejidad promedio ($\Theta(n \log(n))$).

La función sorted() en Python es una función built-in que ordena cualquier iterable (listas, tuplas, etc.) y devuelve una nueva lista ordenada (sin modificar el original).
El algoritmo de ordenamiento que utiliza la función sorted() es Timsort, un algoritmo híbrido que combina MergeSort (para ordenamiento general, estable y eficiente en grandes conjuntos de datos) e Insertion Sort (para pequeñas sublistas, ya que es rápido en tamaños reducidos).

Proyecto 2
Tiempos de ejecución de tales métodos con listas de tamaño entre 1 y 1000.



__len__: Este método (__len__) simplemente devuelve el atributo _tamanio, lo cual es una operación de tiempo constante O(1). Es eficiente.
copiar: Este método recorre todos los nodos de la lista y los copia en una nueva lista. Su complejidad es O(N), ya que depende de la cantidad de elementos.
invertir: Este método recorre todos los nodos de la lista y ajusta los punteros siguiente y anterior. Su complejidad también es O(N).

__len__: Es la forma más eficiente de implementar este método, ya que no recorre la lista para contar los elementos, sino que utiliza un contador que se mantiene actualizado.
copiar: Esta es la forma más eficiente de copiar una lista doblemente enlazada, ya que no hay forma de evitar recorrer todos los nodos para duplicarlos.
invertir: Es la forma más eficiente de invertir una lista doblemente enlazada, ya que cada nodo debe ser procesado para ajustar sus punteros.

Proyecto 3
Se implementa la clase Mazo, la cual hace uso de la lista doble enlazada para almacenar objetos de tipo Carta, utiliza las funciones poner_carta_arriba, sacar_carta_arriba y poner_carta_arriba, las cuales utilizan las funciones agregar_al_final, extraer, agregar_al_inicio y __len__ definidas en el proyecto 2. En caso de querer extraer una carta de un mazo vacío, se lanza la excepción DequeEmptyError (definida en el mismo archivo que la clase Mazo.) que utiliza una Exception para verificar si el mazo está vacío, si no es así, se continúa el flujo normal del programa.