Trabajo Práctico N°1

Integrantes: Ferrer Paira Iñaki, Spaggiari Fiamma

Respuestas

Problema 1

Lo primero que hicimos fue implementar los ordenamientos burbuja, quicksort y radix sort, los mismos se pueden encontrar en la carpeta de problema 1 en modules. Teniendo en cuenta que:

a) **Ordenamiento burbuja:** Compara pares de elementos adyacentes y los intercambia si están en orden incorrecto, lo hace hasta que la lista esté completamente ordenada (complejidad: O(n^2) en el peor de los casos)

b) **Ordenamiento quicksort:** Elige un pivote y a partir de este en la lista habrá elementos mayores al pivote o menores (complejidad: O(n^2) en el peor de los casos)

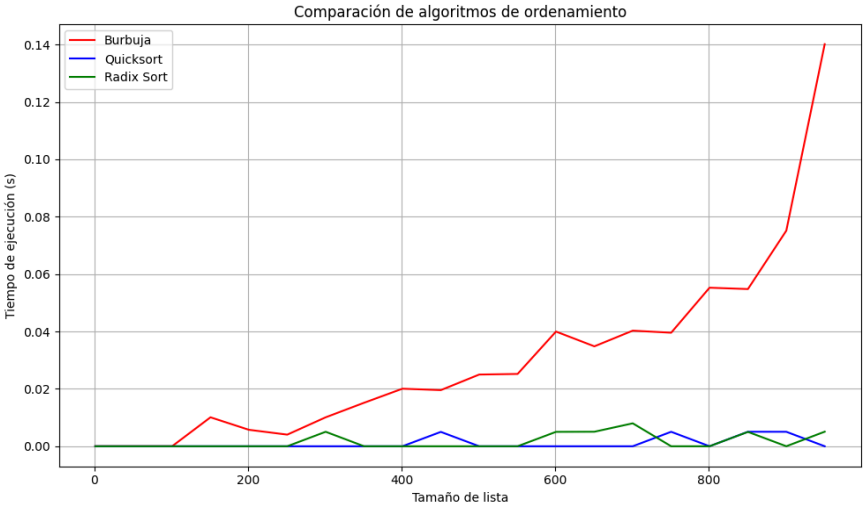
c) **Ordenamiento por residuos (radix sort):** Ordena números enteros desde el menos significativo hasta el más significativo utilizando un ordenamiento estable (complejidad: O(n*k) k → número de enteros)

Una vez implementados para corroborar que funcionan de forma correcta en la carpeta test de problema 1 nos encontramos con un código de prueba (test_problema1.py) con la función test_algoritmos_ordenamiento() generamos una lista de 500 números aleatorios de 5 cifras, creamos una lista ordenada (nuestro resultado esperado) y ejecutando cada algoritmo de ordenamiento comparamos el resultado con la lista ordenada, dependiendo de si es correcto o no se muestra un mensaje

- Si fallo algunos de los ordenamientos: mensaje de error
- Si todo funciona de manera correcta: mensaje de éxito

Una vez comprobado que todo funciona correctamente, vamos a los códigos para medir el tiempo de ejecución y graficar. (modules → tiempos.py y graficar.py)

- **tiempos.py:** mide y guarda el tiempo de ejecución de los 3 algoritmos en un archivo teniendo como prueba una lista aleatoria de números de 5 cifras.
- **graficar.py:** Los datos obtenidos en el punto anterior se grafican para comparar visualmente los tiempos de ejecución de los distintos ordenamientos. La gráfica obtenida es la siguiente:

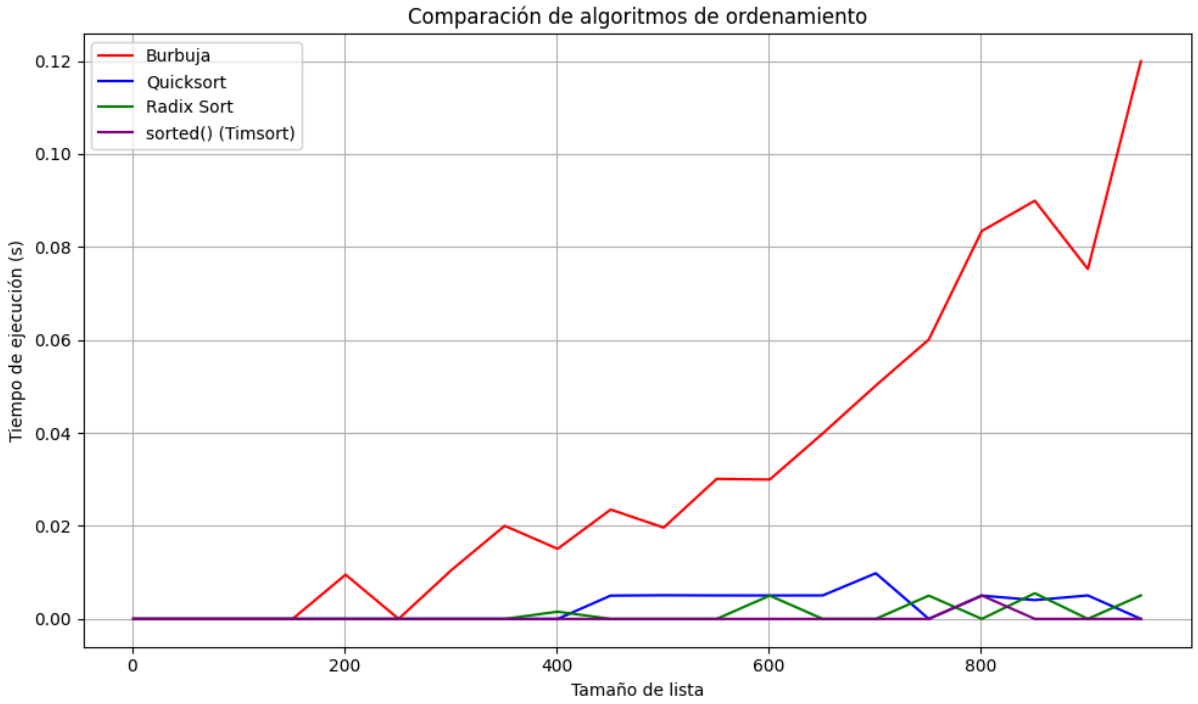


Como conclusión de las gráficas obtenidas observamos que:

a) **Ordenamiento burbuja (rojo):** Su tiempo de ejecución crece muy rápido al aumentar el tamaño, esto es debido a su complejidad es ineficiente para listas grandes.
b) **Ordenamiento quicksort (azul):** Se mantiene con bajos tiempos (eficiente) al ir incrementando el tamaño de las listas
c) **Ordenamiento por residuos (verde):** También presenta tiempos bajos y constantes como el anterior

Una diferencia entre los dos últimos es que la complejidad promedio del quicksort es O(n log n) mientras que la del radix sort es O(n k) por lo que esta última funciona mejor cuando se conocen ciertas características de los datos a trabajar (cantidad de digitos, numero de cifras, etc)

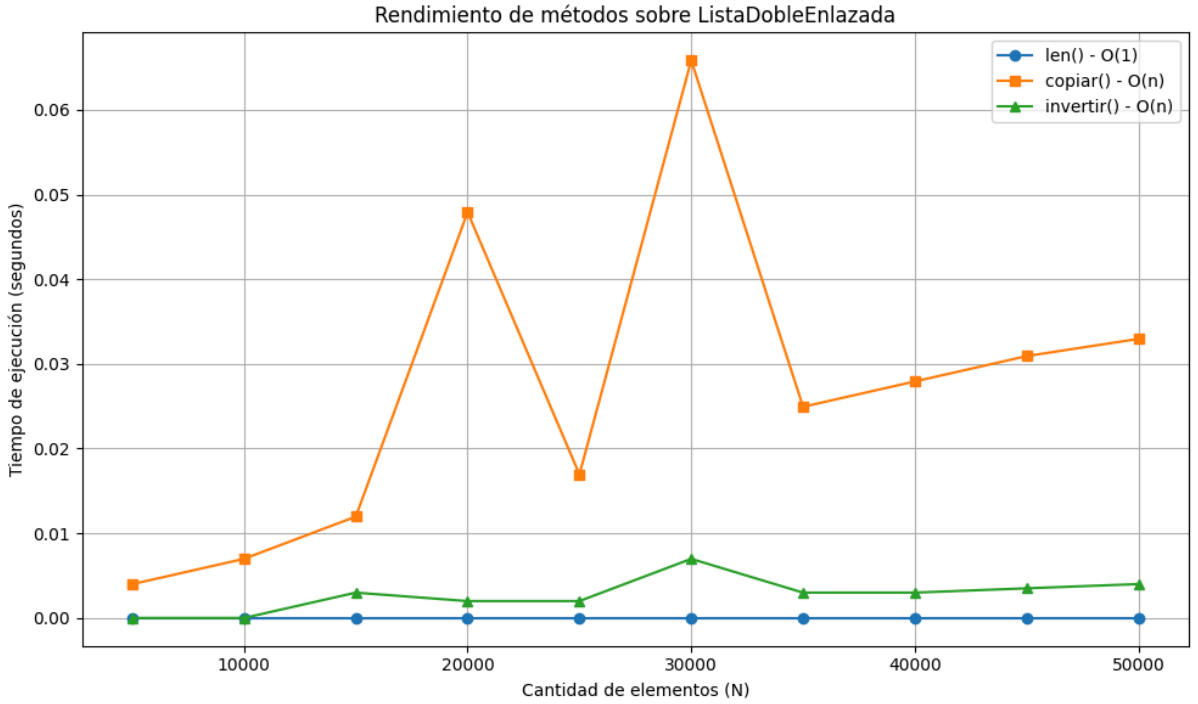
Ahora pasando a la funcion de ordenamiento integrada de python, **sorted(),** esta se usa para ordenar cualquier objeto iterable (listas, tuplas, etc) y devuelve una nueva lista ordenada sin alterar la original.
Internamiento funciona con un algoritmo llamado Timsort (complejidad O(n log n) en el peor de los casos) es muy eficiente y estable.
Para poder compararlo con nuestros algoritmos podemos modificar nuestro código y observarlo gráficamente (queda comentado):



Problema 2

Dentro de la carpeta problema_2, en modules se encuentra un código llamado lista_doble_enlazada.py en el mismo se encuentra la implementación del **TAD lista doblemente enlazada**

Luego en la misma carpeta (modules) el código llamado graficar_tiempos.py nos permite obtener la gráfica de los métodos len, copiar e invertir en base a cantidad de elementos en función del tiempo de ejecución.



Observamos gráficamente lo siguiente:

a) **len() (azul):** Su tiempo de ejecución es prácticamente cero y constante ya que self.tamanio se mantiene actualizado internamente. (complejidad O(1))
b) **copiar() (naranja):** Tiene una tendencia creciente lo cual se espera porque su complejidad es O(n) pero presenta picos irregulares debidos a procesos internos de python
c) **invertir() (verde):** Tiene una tendencia creciente como en el caso anterior ya que su complejidad es igual a O(n), pero es más eficiente que el caso de copiar()

Como resumen general de lo obtenido gráficamente, notamos que len() es más eficiente ya que se mantiene constante con un tiempo de ejecución cercano a cero para diferentes valores de N, mientras que en los demás casos nos encontramos con tiempos de ejecución más altos a medida que aumentamos la cantidad de elementos N, siendo invertir() mas eficiente que copiar().

Problema 3

Nuestra tarea consistía en incorporar la implementación de la clase Mazo, la cual debía utilizar una lista doblemente enlazada para almacenar objetos de tipo Carta y realizar las operaciones necesarias del juego. Para hacerlo, analizamos los archivos provistos por la cátedra, en particular carta.py y juego_guerra.py. A partir de ahí, fuimos identificando qué métodos debía tener la clase Mazo y cómo implementarlos para que se ajustaran al funcionamiento del juego.

Entre los métodos que incorporamos se encuentran

- agregar_carta
- sacar_carta
- esta_vacio
- _len_
- _str_

Todos ellos fueron implementados respetando el comportamiento esperado por el código del juego y los archivos de prueba.

Una vez que teníamos la clase armada, empezamos a ejecutar los tests provistos, que nos ayudaron a validar el funcionamiento. A partir de los errores o diferencias que aparecían, hicimos las correcciones necesarias en la clase Mazo hasta que todos los tests pasaron correctamente. En ningún momento modificamos los archivos entregados por la cátedra, lo que nos permitió asegurar que nuestra clase se integrara bien con el resto del proyecto.