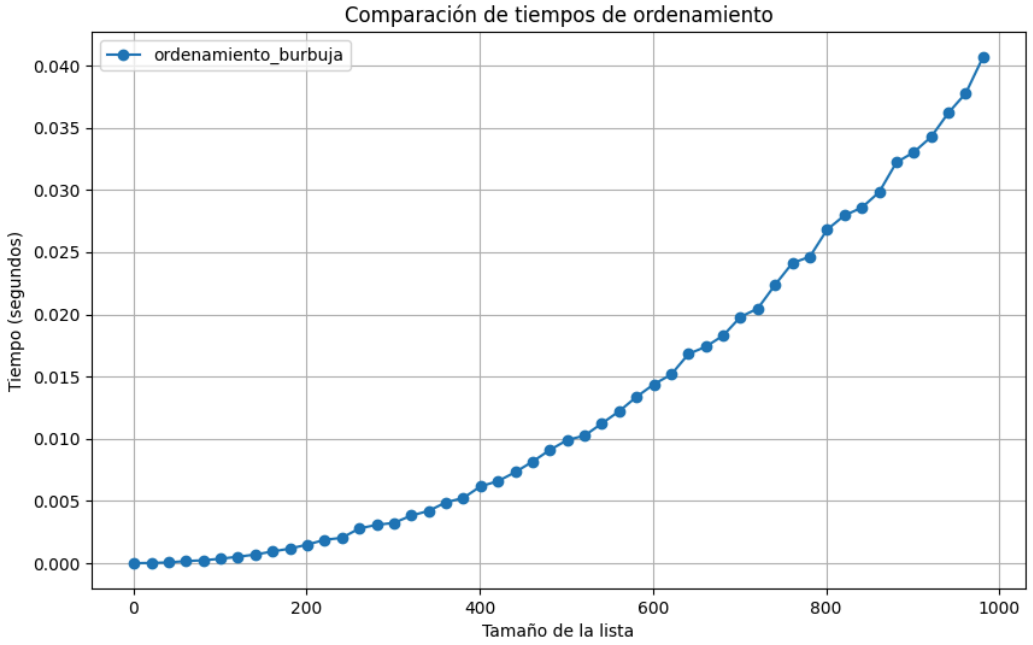
TP N°1

Algoritmos y Estructuras de Datos
Deleon, Iván; Gerbaudo, Sabina; Rogau, Victoria.

Problema 3

En el problema N°3 se realizaron los siguientes algoritmos de búsqueda; ordenamiento burbuja, ordenamiento quicksort y ordenamiento por residuos (radix sort).
A continuación se muestran las gráficas de tiempos de cada método analizado y su funcionamiento.

Ordenamiento burbuja



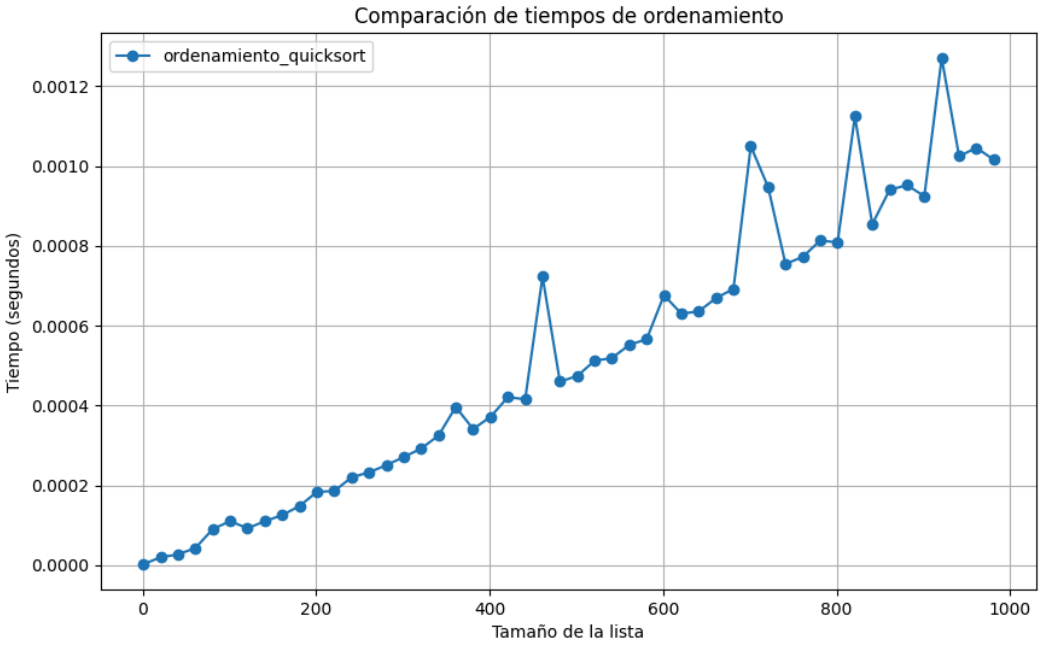
Gráfica obtenida del ordenamiento burbuja

El algoritmo de burbuja compara de manera repetida pares de elementos adyacentes en la lista y los intercambia si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista queda ordenada.

En el código, el ciclo externo controla el número de pasadas y el ciclo interno compara cada elemento con el siguiente. La versión optimizada incluye una bandera (intercambiado) que detiene el algoritmo si en una pasada no hubo cambios, indicando que la lista ya está ordenada.

Es un método sencillo pero ineficiente en listas grandes, con una complejidad de $O(n^2)$.

Ordenamiento quicksort



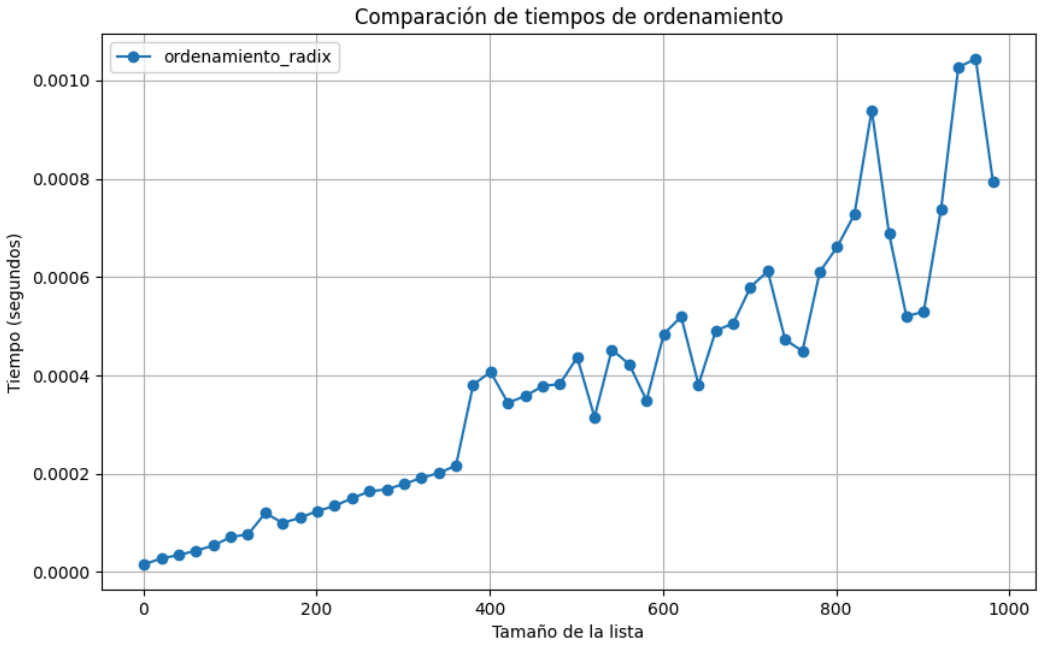
Gráfica obtenida del ordenamiento quicksort

El quicksort es un algoritmo de divide y vencerás. Primero selecciona un elemento llamado pivote y reorganiza la lista de manera que todos los menores al pivote queden a su izquierda y los mayores a su derecha (esto se hace en la función particion). Luego, se aplica recursivamente el mismo proceso a cada sublista.

Este método no necesita listas auxiliares y suele ser muy rápido en la práctica, con una complejidad promedio de O(n log n), aunque en el peor de los casos puede llegar a $O(n^2)$.

Es importante aclarar que en la gráfica obtenida se denotan ciertos picos, cuando la forma ideal de la misma debería ser una recta (función lineal), esto es debido a procesos internos que realiza la computadora (cada computadora en la que se ejecutó el código arrojó una gráfica distinta, intentando comportarse como una recta).

Ordenamiento por residuo (radix sort)



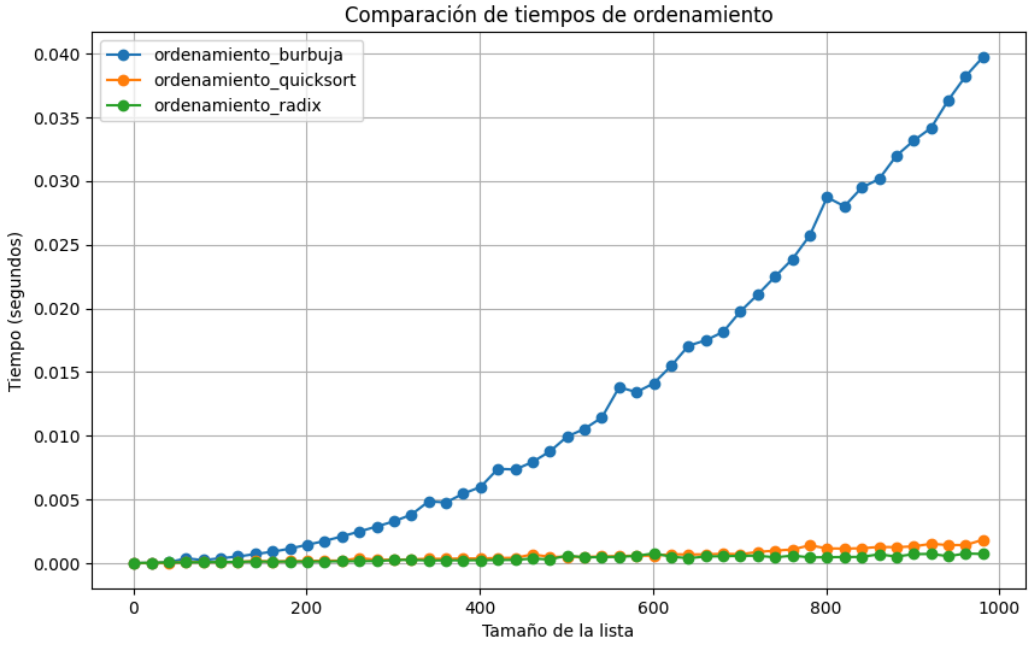
El radix sort ordena los números basándose en sus dígitos, procesándolos desde el menos significativo al más significativo. En cada pasada, distribuye los números en "cajones" según el valor de un dígito específico y luego reconstruye la lista.

Este proceso se repite hasta que se han recorrido todos los dígitos del número más grande.

Es un algoritmo no comparativo, muy eficiente para ordenar números, con complejidad O(n·k) (donde *k* es el número de dígitos máximos).

En este caso, al igual que el anterior, los picos de la gráfica se deben a procesos internos de la computadora. También debería verse como una recta, con diferente pendiente de la anterior.

La siguiente gráfica es una comparación de todos los algoritmos empleados, en dicha gráfica se hace evidente la diferencia de tiempo entre los distintos métodos, aunque esto no es necesariamente una descripción directa de eficiencia, pues no se está comparando la memoria utilizada, aspecto fundamental para el análisis de eficiencias, dado que cada computadora puede variar en la utilización de la memoria para este trabajo.



La gráfica muestra cómo el ordenamiento burbuja se vuelve mucho más lento a medida que aumenta el tamaño de la lista, mientras que quicksort y radix sort mantienen tiempos casi constantes. Esto indica que, aunque todos ordenan, algunos algoritmos son mucho más rápidos y eficientes que otros.

Función sorted

La función incorporada sorted() permite ordenar listas, tuplas, cadenas u otros iterables en Python. Devuelve una nueva lista ordenada sin modificar la original.

- Por defecto ordena de menor a mayor.

- Se puede usar el parámetro reverse=True para ordenar en orden descendente.

- Con el parámetro key se define una función de criterio de ordenación (por ejemplo, ordenar palabras por longitud).
  Internamente, sorted() utiliza el algoritmo Timsort, una mezcla de *merge sort* e *insertion sort*, optimizado para datos reales. Su complejidad es O(n log n) en promedio.