ACTIVIDAD 3

Se implementaron en Python los tres algoritmos de ordenamiento solicitados:

Ordenamiento Burbuja (Burbuji): Es un algoritmo de comparación directa. La implementación utiliza dos bucles anidados. El bucle exterior (pasada) define el segmento de la lista que aún no está ordenado, y el bucle interior (i) realiza comparaciones de elementos adyacentes Si los elementos no están en el orden correcto, se intercambian. Este mecanismo asegura que, en cada pasada completa del bucle exterior, el elemento de mayor valor se desplaza progresivamente hasta su posición final correcta.

Ordenamiento Quicksort (ordenamientoRapido): Es un algoritmo recursivo basado en la técnica de partición. La implementación selecciona un elemento como pivote (en este caso, el primero de la sub-lista]). La función particion reorganiza la lista de forma que todos los elementos con valores menores al pivote se sitúan a un lado de este, y los elementos con valores mayores se sitúan al otro. Posteriormente, el algoritmo se aplica de manera recursiva a las dos sub-listas resultantes.

Ordenamiento por Residuos (radix_sort): Es un algoritmo de ordenamiento no comparativo que opera procesando los números en base a sus dígitos. Utiliza counting_sort (ordenamiento por conteo) como subrutina estable. El algoritmo principal (radix_sort) itera sobre las posiciones de los dígitos y utiliza counting_sort para ordenar la lista completa basándose en el valor de ese dígito en particular

ANÁLISIS DE COMPLEJIDAD

Ordenamiento Burbuja:

- Complejidad: O(n^2)
- Justificación: El algoritmo consiste en dos bucles anidados. El bucle externo se ejecuta $n - 1$ veces. El bucle interno se ejecuta n-1 veces en la primera pasada, $n - 2$ en la segunda, y así sucesivamente. El número total de comparaciones es la suma de $\frac{n(n-1)}{2}$ lo cual se simplifica a O(n^2). El tiempo de ejecución crece cuadráticamente con el número de elementos.

Ordenamiento Quicksort:

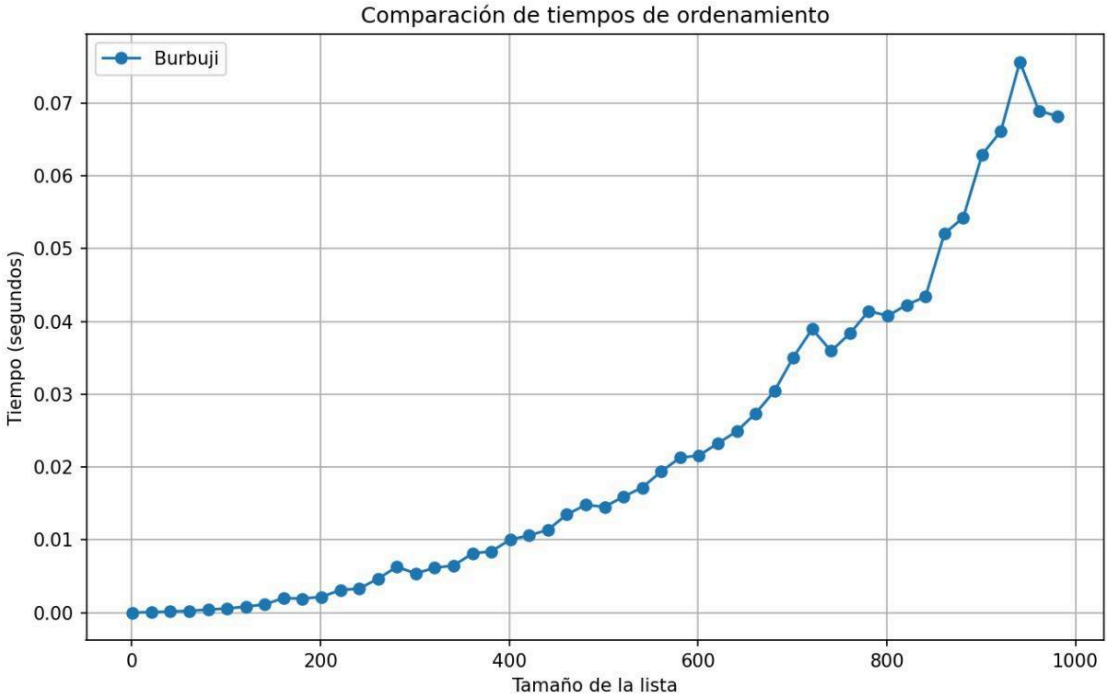
- Complejidad (Caso Promedio): O($n \log n$)
- Justificación: En el caso promedio, la función particion (que tiene complejidad O(n) divide la lista en dos mitades aproximadamente iguales. Esto da lugar a una profundidad de recursión de log n. En cada nivel de la recursión, se procesan n elementos en total. Por lo tanto, la complejidad total es $n \log n$
- Complejidad (Peor Caso): O(n^2)
- Justificación: El peor caso ocurre cuando el pivote es siempre el elemento más pequeño o más grande, generando particiones desbalanceadas (una de 0 elementos y otra de n-1. Dado que la implementación usa el primer elemento como pivote, este

peor caso se presentará si la lista ya está ordenada o inversamente ordenada. La profundidad de la recursión se vuelve n, resultando en una complejidad $O(n^2)$

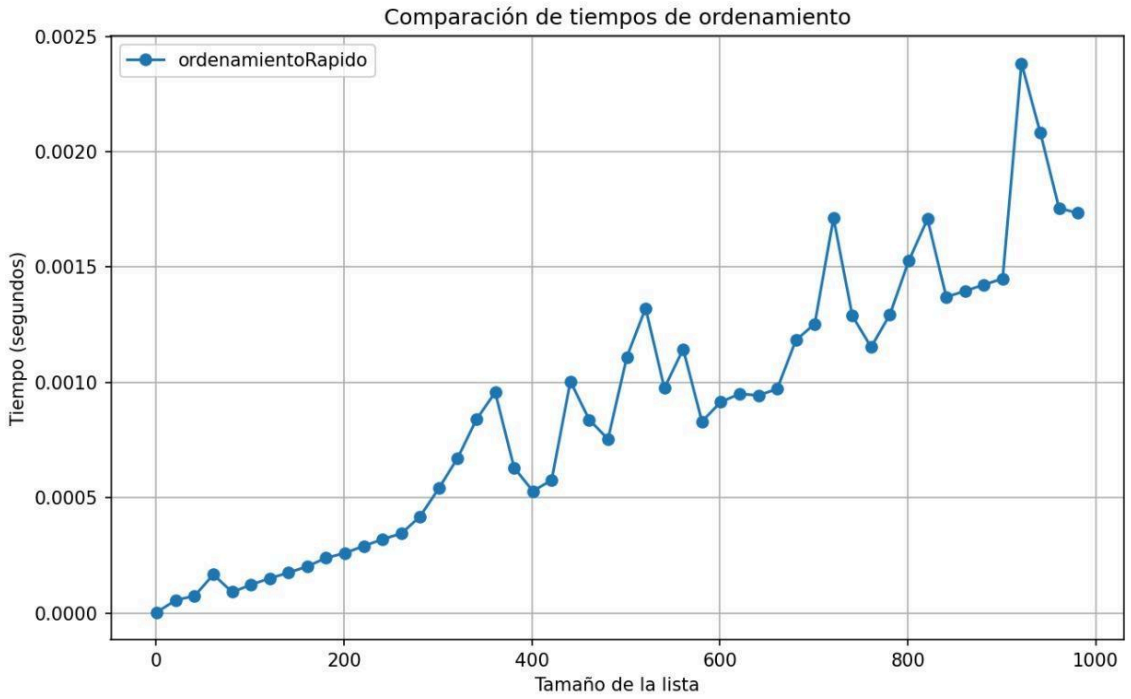
Ordenamiento por Residuos (Radix Sort):

- Complejidad: O(k (n + b)).
- Justificación: El tiempo de ejecución depende de tres factores: n (número de elementos), k (número de dígitos del número máximo) y b (la base, en este caso 10, por los dígitos 0-9).
- El algoritmo radix_sort llama a counting_sort k veces.
- La subrutina counting_sort tiene una complejidad de O(n + b), ya que debe iterar n veces para construir el conteo += 1 y n veces para construir la salida, además de b veces para sumar los conteos.
- La complejidad total es O(k(n + b)). Para este problema, k (dígitos) y b (base 10) son constantes, por lo que la complejidad se puede simplificar a O(n) (Lineal).

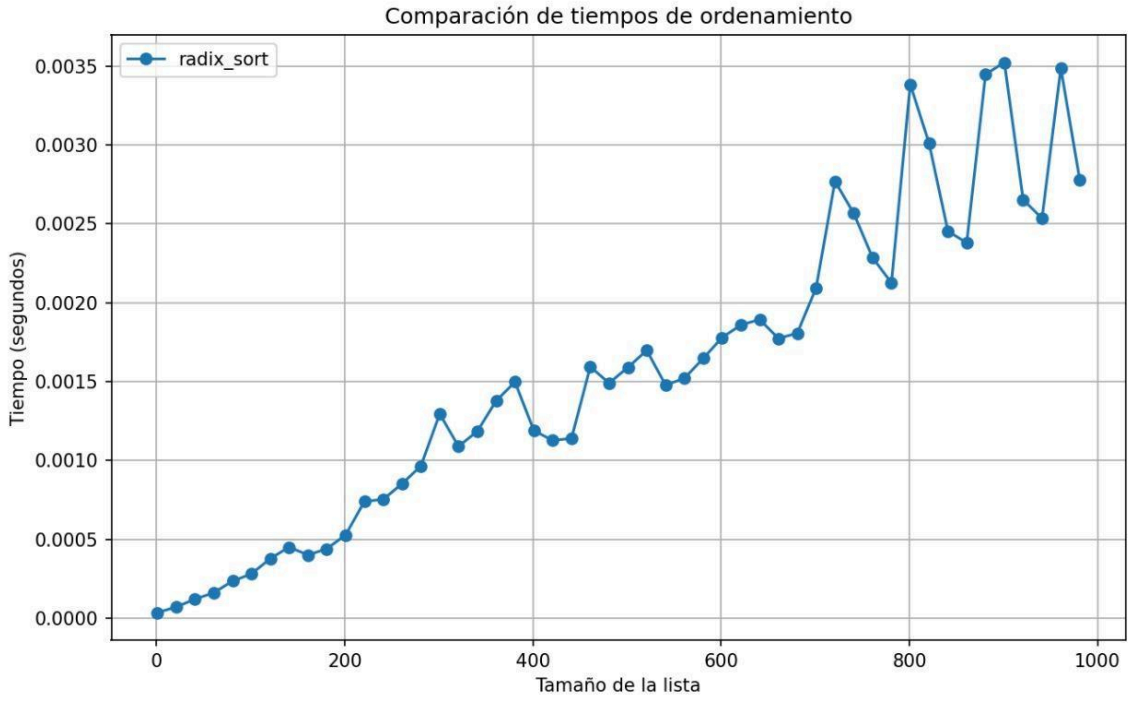
Burbuja: Se espera ver una curva cuadrática (parabólica) clara. Será el algoritmo más lento por un amplio margen, y su tiempo crecerá drásticamente a medida que N aumenta.



Quicksort: Se espera ver una curva log-lineal (n log n), que a simple vista parecerá una línea casi recta, pero con una ligera curvatura hacia arriba. Será uno de los más rápidos.



Radix Sort: Se espera ver una línea recta (O(n)), demostrando su escalabilidad lineal. Es probable que sea el algoritmo más rápido en este escenario, dado que k es una constante pequeña



Tanto en el ordenamiento por residuos como en el ordenamiento rápido podemos notar unos picos en la funciones esto puede deberse a ruido dentro del sistema. Lo que afecta al rendimiento computacional causando los picos antes descriptos.

La función sorted() utiliza un algoritmo de ordenamiento llamado Timsort.

Es un algoritmo híbrido: Timsort no es un solo algoritmo, sino que combina inteligentemente dos:

Merge Sort (Ordenamiento por Mezcla): Se usa para la estrategia general, garantizando una complejidad de peor caso de O(n log n).

Insertion Sort (Ordenamiento por Inserción): Se usa para ordenar tramos pequeños de la lista (llamados "runs"). Insertion Sort es extremadamente rápido para listas pequeñas o casi ordenadas.

Timsort está diseñado para ser muy rápido con los tipos de datos que se encuentran comúnmente. Primero, hace una pasada rápida por la lista para encontrar "corridas" (runs), que son subsecuencias que ya están ordenadas (ascendente o descendentemente). Luego, ordena eficientemente esas corridas usando Insertion Sort y las fusiona usando Merge Sort.

Una razón clave de su velocidad es que Timsort no está escrito en Python. Está implementado directamente en el lenguaje C, por lo que se ejecuta como código compilado nativo, mucho más rápido que el código Python interpretado.

Su complejidad es O(n log n) en el caso promedio y, a diferencia de nuestro Quicksort, también en el peor de los caso