Problema 1

En el problema 1 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.

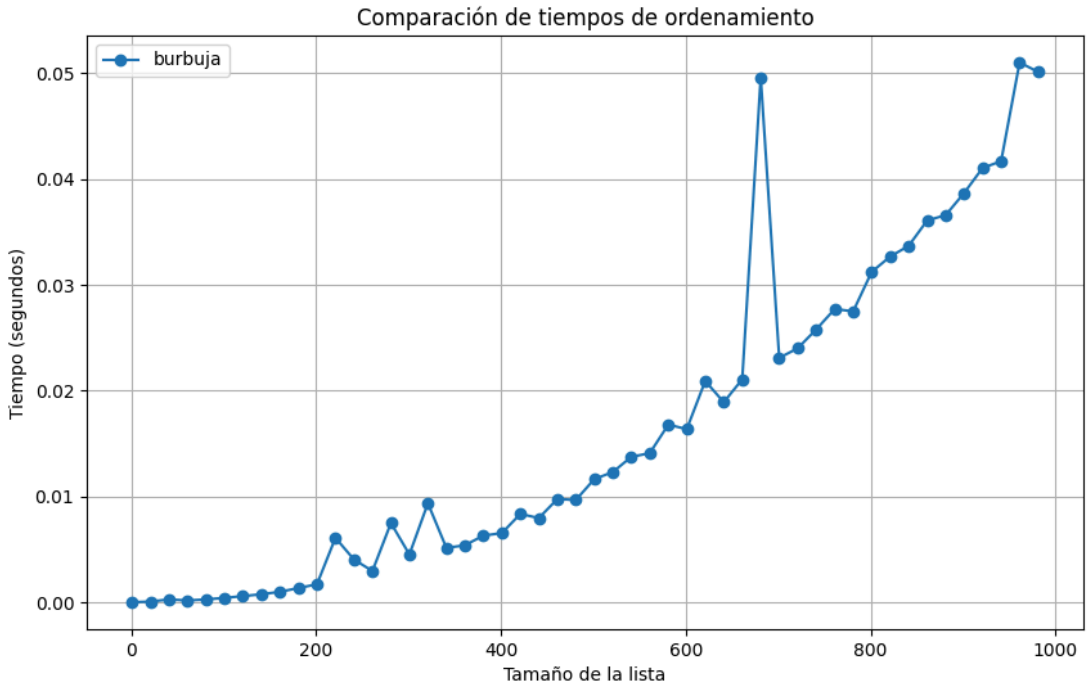

Fig. 1 Ordenamiento burbuja

El algoritmo de ordenamiento burbuja ordena una lista comparando y cambiando los elementos adyacentes si están en el orden incorrecto. Esto se repite hasta que la lista está ordenada. Aunque es fácil de implementar, el algoritmo de ordenamiento burbuja no es eficiente para listas grandes ya que su tiempo de ejecución promedio es cuadrático, $O(n^2)$.

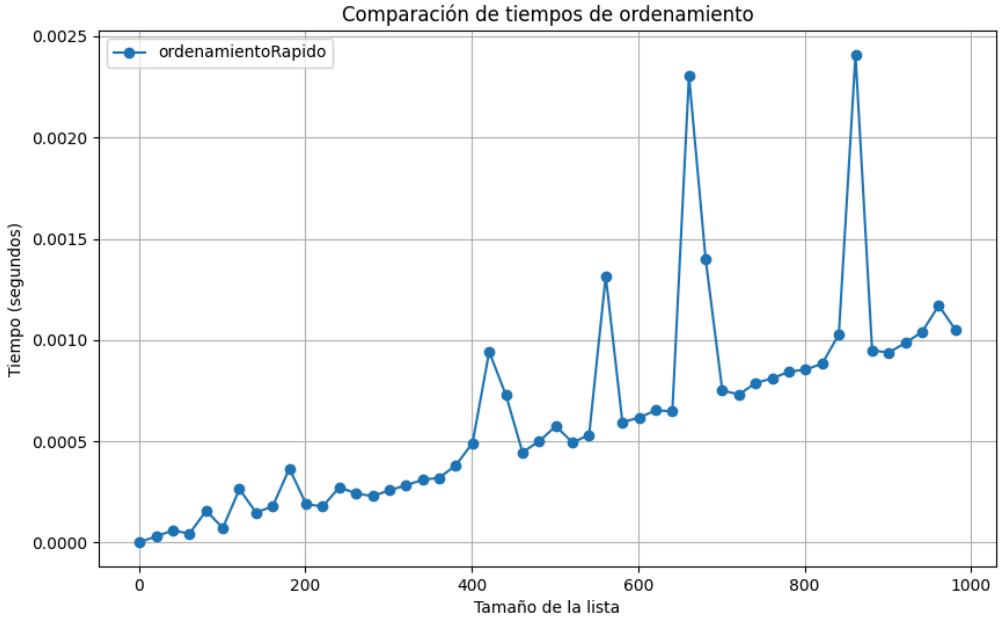

Fig. 2 Ordenamiento quicksort

Quicksort es un algoritmo de ordenamiento que funciona eligiendo un elemento llamado pivote y separando el resto de la lista en dos partes: los elementos menores al pivote y los elementos mayores. Luego aplica el mismo proceso de forma recursiva a cada parte hasta que todo queda ordenado. En promedio es muy rápido, con una complejidad de O(n log n), aunque en el peor caso puede llegar a ser $O(n^2)$ si la elección del pivote es muy mala.

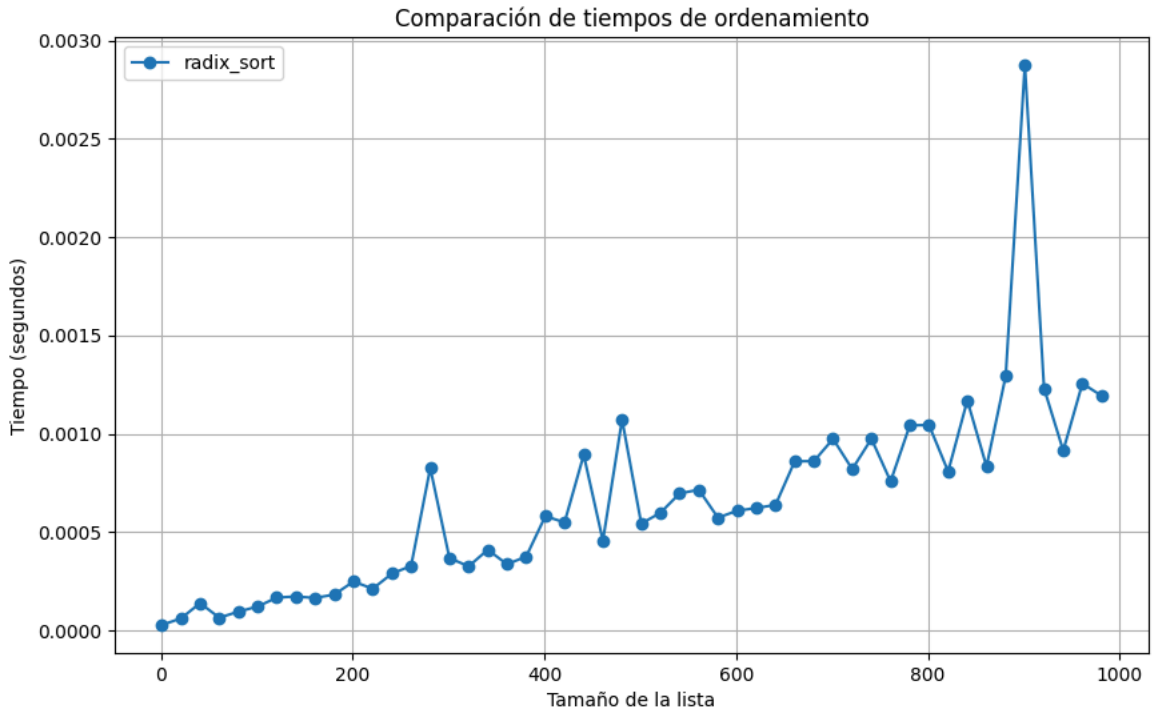

Fig. 3 Ordenamiento por residuo (radix sort)

Radix Sort es un algoritmo de ordenamiento que organiza los números procesando sus dígitos de menor a mayor, utilizando un método de orden estable en cada paso, como el conteo. Primero ordena por la cifra menos significativa, luego por las decenas, centenas, y así sucesivamente, hasta ordenar todos los dígitos. Su complejidad es O(nk), donde n es el número de elementos y k es el número de dígitos del número más grande.

En cada paso, utiliza Counting Sort para ordenar los elementos según el dígito actual. Este paso tiene una complejidad de O(n), ya que se procesan todos los elementos de la lista. Luego, repite el proceso para cada uno de los **k** dígitos. Como el algoritmo realiza **k** pasadas y en cada pasada recorre todos los **n** elementos, la complejidad total es O(nk).

La siguiente gráfica cuenta con 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.

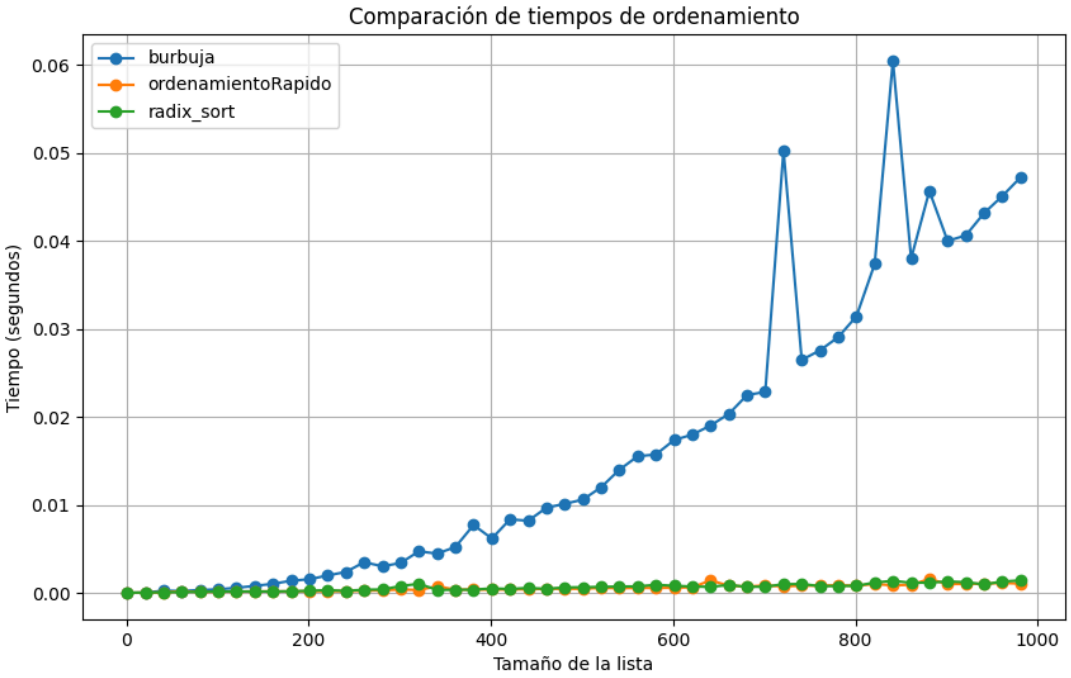

Fig. 4 Tabla comparativa entre los algoritmos utilizados

En esta comparación (Figura 4) de los 3 algoritmos, podemos observar que el tiempo requerido por el ordenamiento burbuja crece considerablemente más rápido en función del tamaño del problema que los dos restantes y podríamos suponer se cumplen las suposiciones teóricas.

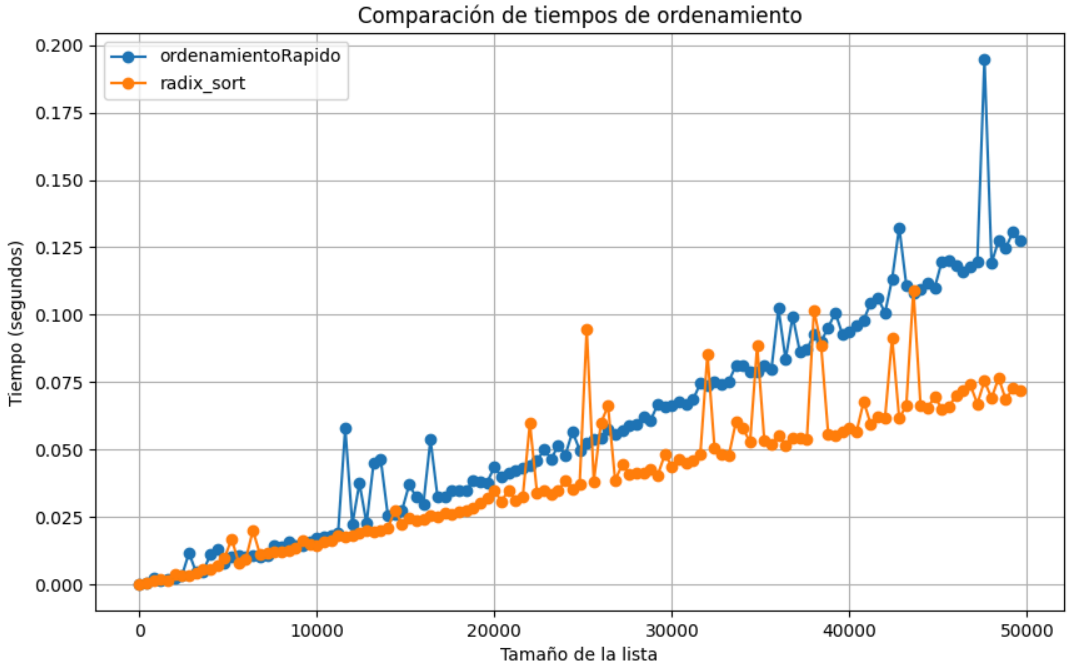

Fig. 5 Tabla comparativa entre quicksort y radixsort

Realizamos una graficación de los dos algoritmos más eficientes, extendiendo el tamaño máximo del problema 50 veces, de esta manera, pudimos notar que el tiempo requerido por el radixsort crece con menor pendiente denotando su mejor eficiencia temporal.

Función sorted

Sorted utiliza el algoritmo Timsort, que es una combinación de Merge Sort y Insertion Sort. El proceso de Timsort comienza dividiendo la lista en bloques, estos bloques o “runs” tienen un tamaño de entre 32 y 64 elementos. Estas sublistas son ordenadas utilizando Insertion Sort, un algoritmo eficiente para listas pequeñas o parcialmente ordenadas. Una vez que las sublistas están ordenadas, se procede a fusionarlas utilizando Merge Sort, un algoritmo que combina listas ordenadas de manera eficiente.

Timsort está diseñado para ser especialmente rápido con listas que ya están parcialmente ordenadas, ya que puede identificar secuencias de elementos que ya están en el orden correcto y evitarlas.

En cuanto a su complejidad, Timsort tiene un mejor rendimiento en el mejor de los casos, donde las listas están parcialmente ordenadas, alcanzando O(n). Sin embargo, en el peor de los casos, cuando la lista está completamente desordenada, la complejidad es O(n log n).