Trabajo Práctico N°1

Problema 3

Implementar en python los siguientes algoritmos de ordenamiento:

a) Ordenamiento burbuja
b) Ordenamiento quicksort
c) Ordenamiento por residuos (radix sort)

Corroborar que funcionen correctamente con listas de números aleatorios de cinco dígitos generados aleatoriamente (mínimamente de 500 números en adelante).

Medir los tiempos de ejecución de tales métodos con listas de tamaño entre 1 y 1000. Graficar en una misma figura los tiempos obtenidos. ¿Cuál es el orden de complejidad O de cada algoritmo? ¿Cómo lo justifica con un análisis a priori?

Comparar ahora con la función built-in de python **sorted**. ¿Cómo funciona sorted? Investigar y explicar brevemente.

ENTREGABLE DEL INFORME 1: Comparación de algoritmos de ordenamiento

En este trabajo se implementaron tres algoritmos de ordenamiento: Burbuja, Quicksort y Radix Sort, y se compararon sus tiempos de ejecución con la función incorporada sorted() de Python (basada en el algoritmo Timsort).

Análisis experimental

El objetivo fue ordenar listas de números de cinco dígitos generados aleatoriamente, para lo cual generamos una función que genere 750. Luego verificamos que los algoritmos funcionan correctamente, midiendo sus tiempos de ejecución (también con la utilización de una función) para tamaños de listas entre 1 y 1000 elementos. Posteriormente se graficó todos los algoritmos en un mismo gráfico.

Breve explicación de cada ordenamiento:

Ordenamiento burbuja:

- Realiza múltiples pasadas a lo largo de una lista, comparando e intercambiando los ítems adyacentes si no están en orden. En cada pasada a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado. En esencia, cada ítem “burbujea” hasta el lugar al que pertenece. Este proceso se repite hasta que la lista queda ordenada.

Ordenamiento quicksort:

- Dividir y conquistar. Selecciona un pivote, en este caso es el primer elemento y particiona la lista en dos sublistas: elementos menores al pivote a la izquierda y mayores a la derecha. El procedimiento se aplica recursivamente a las sublistas hasta obtener la lista ordenada.

Ordenamiento radix sort:

- Funciona examinando los números dígito por dígito, comenzando por el dígito menos significativo (unidades). Los números se distribuyen en "cajitas" (cubetas) según el valor de dicho dígito. Las cajitas se recolectan en orden, y el proceso se repite con el siguiente dígito (decenas, centenas, etc.), hasta que todos los dígitos han sido procesados.

Ordenamiento sorted:

- Es la función de ordenamiento incorporada en Python. Devuelve una nueva lista ordenada a partir de cualquier iterable de entrada, sin modificar el original. Es un algoritmo híbrido que combina Merge Sort e Insertion Sort, conocido por su rendimiento, estabilidad y robustez.

Orden de complejidad de cada algoritmo a priori:

- Ordenamiento burbuja: $O(n^2)$. Recorre la lista comparando e intercambiando elementos adyacentes. Muy ineficiente en listas grandes.
- Quicksort: En el mejor de los casos, la lista ya está ordenada, no se realizan cambios la complejidad es $O(n \log n)$, pero el peor de los casos, cada comparación causará un intercambio $O(n^2)$.
- Radix sort: $O(nk)$ siendo k la cantidad de dígitos. Para enteros de tamaño fijo, se aproxima a $O(n)$
- Sorted: combina Merge Sort e Insertion Sort, $O(n)$ en el mejor de los casos, y en el peor de los casos o en el promedio $O(n \log n)$.

Resultados obtenidos (ver Figura A):

- Burbuja: el más lento, con crecimiento cuadrático evidente.
- Quicksort y Radix Sort: mucho más rápidos, con comportamiento casi lineal en este rango de tamaños.
- sorted(): el más eficiente de todos, gracias a las optimizaciones de Timsort

Conclusión

Los resultados coinciden con el análisis teórico: algoritmos cuadráticos como Burbuja se vuelven inviables al crecer el tamaño de la entrada, mientras que algoritmos O(nlogn) o casi lineales como Quicksort, Radix y Timsort mantienen tiempos muy bajos. En Python, la mejor opción práctica siempre es sorted(), por ser optimizado, estable y robusto.

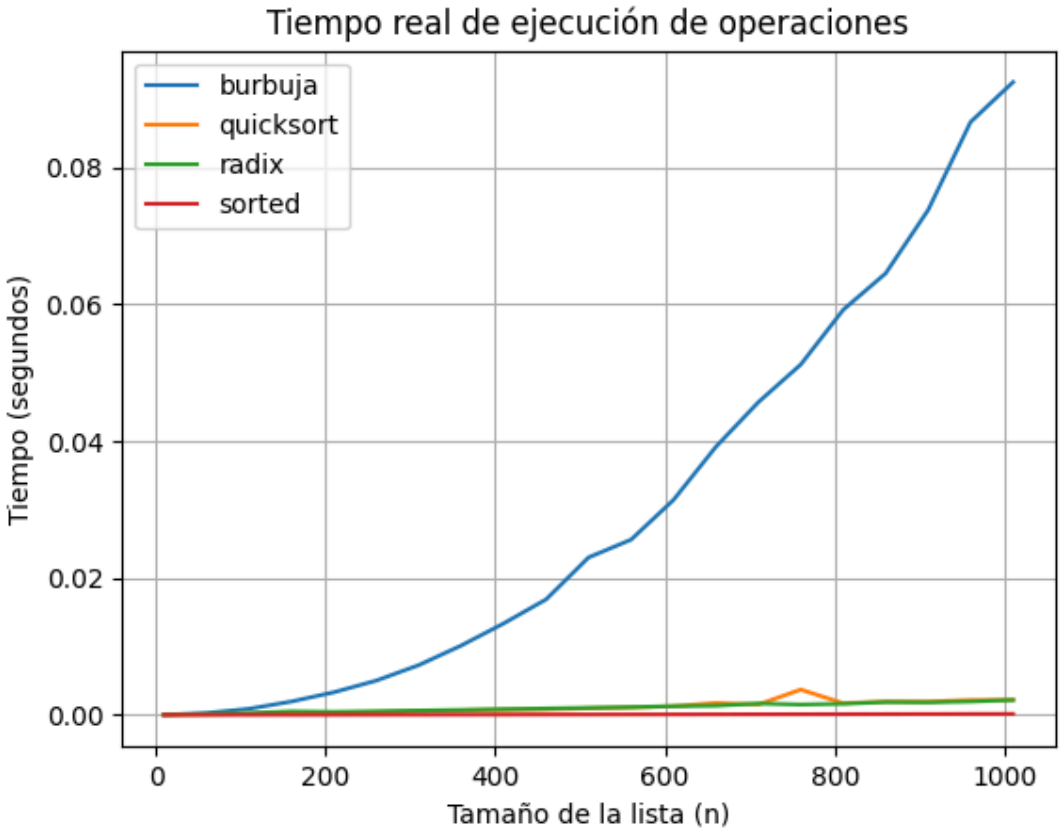

Figura A: Gráfica del tiempo de ejecución de todos los ordenamientos, para listas de tamaño entre 1 y 1000