Problema 3

Aclaración:

En la carpeta se puede observar que hay son archivos que corresponden al ordenamiento radix sort, uno con dicho nombre, y otro llamado ordenamiento_residuos. Esto se debe a que primeramente utilizamos el archivo ordenamiento_residuos pero, al hacer la gráfica notamos algunas irregularidades. Entonces luego usamos el que está en el repositorio de ayed-codigos-teoria-para-estudiantes/04_Busqueda_y_ordenamiento_ejemplos. Por eso se puede ver también que, en el archivo de graficación hubo dos intentos. El primero (hecho por el grupo) está comentado. Y el segundo que es una mejora del primero con recomendaciones del profesor. Este último fue el utilizado para las gráficas a continuación.

Análisis anticipado del orden de complejidad

Burbuja: cada pasada compara pares de elementos y realiza potencialmente n pasadas; por tanto el ordenamiento es $O(n^2)$

Quicksort: el arreglo se divide recursivamente en dos subarreglos y luego se ordena cada uno. En el peor caso puede ser $O(n^2)$, pero con una buena elección del pivote resulta eficiente. El caso promedio es $O(n \log n)$

Radix Sort (residuos): el ordenamiento es $O(d \cdot (n + k))$, donde *d* es la cantidad de dígitos y *k* el rango de valores de cada dígito. Para números enteros de tamaño fijo, se considera prácticamente lineal en n.

Gráficas

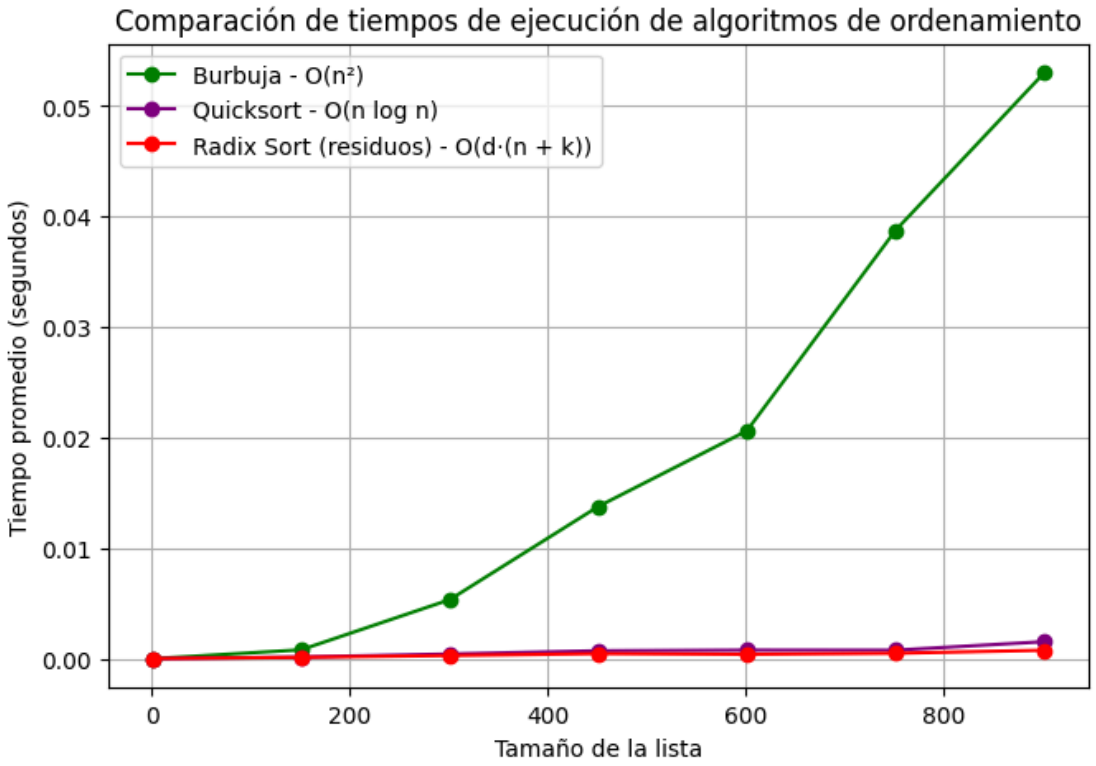

Figura 2

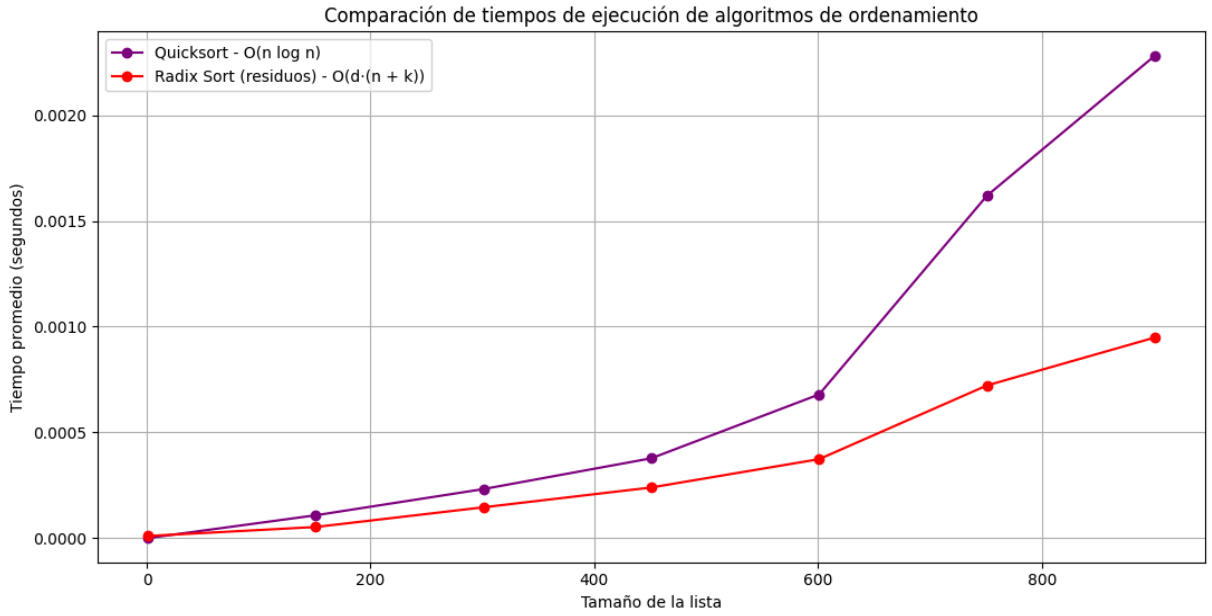

Figura 3

Los tiempos obtenidos para los tres tipos de ordenamiento se graficaron en una misma gráfica (figura 2). Para los ordenamientos quicksort y radix sort, adjuntamos otra gráfica (figura 3) distinta, sin el ordenamiento burbuja, dado que, al graficar los tres, se notaban muy similares (quicksort y radix sort) por la escala que se maneja. Entonces, en la figura 3 se analiza óptimamente la diferencia entre los dos ordenamientos ya mencionados.

Se observó que:

El ordenamiento burbuja fue el algoritmo menos eficiente: su tiempo creció rápidamente de manera cuadrática. Los picos o irregularidades que se observan pueden ser debido al funcionamiento interno propio de la computadora

El ordenamiento quicksort tuvo un desempeño mucho mejor, con tiempos que crecieron en forma aproximadamente logarítmica (en la figura 2 pareciera ser el peor caso $O(n^2)$, ya que es similar al ordenamiento burbuja en la figura 2) respecto al tamaño de la lista -aunque en la figura 2 pareciera constante-.

El ordenamiento radix sort resultó el más eficiente de los tres y prácticamente constante para los tamaños de listas usados en la prueba, dado que depende linealmente del número de dígitos y del tamaño de la lista.

Datos de `sorted()`

sorted utiliza el algoritmo Timsort, que es una combinación de *Merge Sort* e *Insertion Sort*. Su diseño híbrido le permite equilibrar la rapidez de *Insertion Sort* en listas pequeñas con la escalabilidad de *Merge Sort* en listas grandes. Por eso, Timsort no solo es usado en Python, sino también en otros lenguajes como Java (para ordenar objetos), consolidándose como uno de los algoritmos de ordenamiento más eficientes en aplicaciones prácticas.

Timsort fue diseñado para aprovechar las subsecuencias ya ordenadas dentro de una lista (llamadas *runs*). Su complejidad es $O(n \log n)$ en el peor caso, pero puede llegar a ser muy cercana a $O(n)$ cuando los datos tienen un orden parcial (es decir, ya tienen *runs* largas). En lugar de ordenar todo desde cero, identifica estas secciones preordenadas y las utiliza como bloques de construcción, lo que hace que el algoritmo sea mucho más eficiente en escenarios reales, donde los datos no siempre son completamente aleatorios.

Conclusiones

La experimentación corroboró que los algoritmos implementados cumplieron con la complejidad esperada: el ordenamiento burbuja resultó ser el más ineficiente. Quicksort y Radix Sort fueron significativamente más rápidos, con Radix Sort siendo el más eficiente para este tipo de datos.
Se realizó la comparación de cada método de ordenamiento con `sorted()` para verificar, a modo de test, que las listas estuvieran ordenadas, y todos los ordenamientos superaron la prueba