Informe Trabajo Práctico N°1

Problema 3: Algoritmos de ordenamientos

Enunciado:

Implementar burbuja, quicksort y ordenamiento por residuo, corroborar que funcionen correctamente con listas de números aleatorios de cinco dígitos generados aleatoriamente, medir tiempos de ejecución para listas aleatorias (N=1 a 1000), graficar resultados y comparar con `sorted` de Python.

Solución:

Burbuja (Bubble Sort):
Compara los ítems adyacentes e intercambia si no están en orden.
Quicksort:
Selecciona un pivote y divide la lista en dos partes: los elementos menores y los mayores que dicho pivote. Luego aplica el mismo procedimiento recursivamente a cada sublista.
Ordenamiento por residuo:
Utiliza el algoritmo counting sort. Ordena los números procesando cada dígito, comenzando por el " menos significativo" avanzando hacia el "dígito más significativo".
Sorted (Timsort):
Algoritmo de Python que combina mergesort con insertion sort y aprovecha secuencias ya ordenadas dentro de la lista para optimizar el rendimiento.

Análisis de complejidad:

Se midió el tiempo de ejecución aplicando los diferentes métodos de ordenamiento sobre listas aleatorias. Utilizamos la notación **Big-O** para describir cómo crece el tiempo de ejecución en relación al tamaño de la lista N.
En nuestro caso realizamos el análisis sobre:
Burbuja
Si la lista tiene N elementos, el bucle externo realiza N pasadas y, en cada pasada, compara hasta N pares de elementos.
Esto da aproximadamente N×N=N2 comparaciones, por lo que la complejidad es **O(n²).**
En el **peor caso** y en el **promedio**, la complejidad se mantiene.
En el **mejor caso**, cuando la lista ya está ordenada, gracias a la optimización que implementamos en el código, la ejecución termina después de una sola pasada → **O(n).**

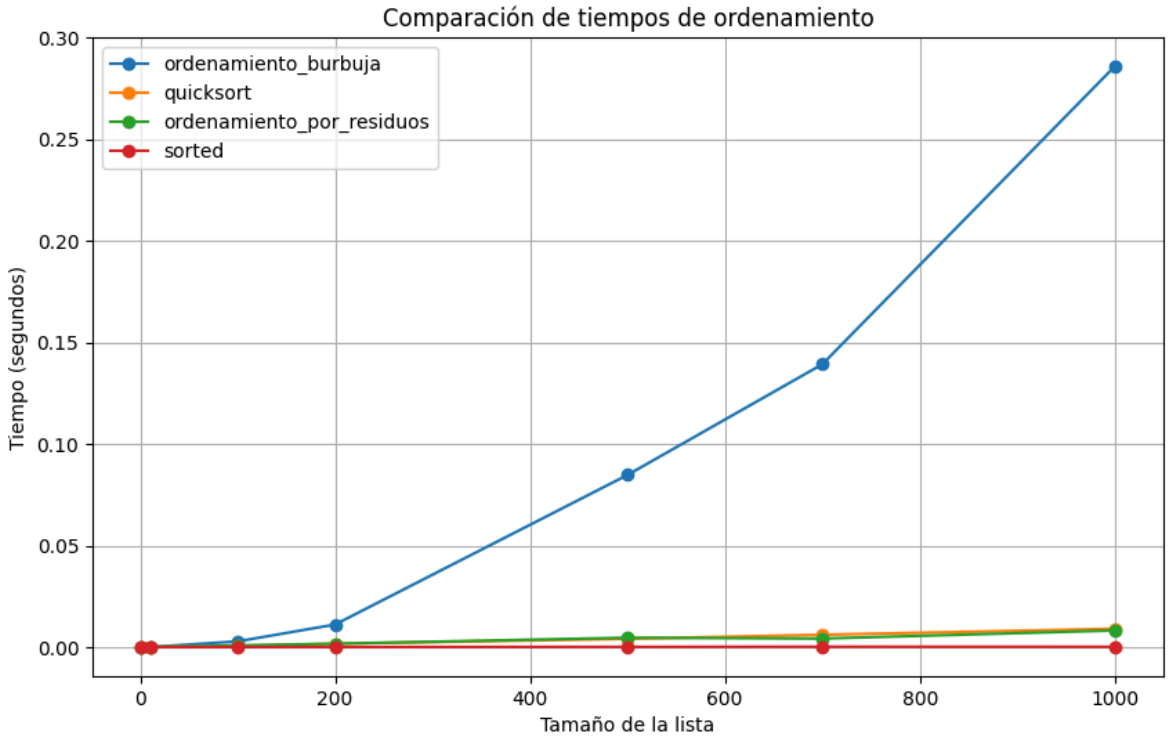
Quicksort:
Caso promedio: si el pivote divide la lista de manera equilibrada, se forman aproximadamente log2(N) niveles de recursión y en cada nivel se comparan todos los

elementos (N comparaciones por nivel). → **O(n log n)**.
Peor caso: si el pivote es siempre el mayor o menor elemento, las sublistas quedan muy desbalanceadas, y cada nivel recorre casi toda la lista → **O(n²)**.

Ordenamiento por residuo (Radix Sort):
Número de **operaciones por dígito**: cada vez que se llama a counting-sort, se recorren los N elementos → **O(n).**
Número de dígitos: si el número más grande tiene k dígitos, se necesitan k pasadas → **O(k·n)**.
Caso promedio y **peor caso**: para enteros positivos, siempre se requieren k pasadas sobre toda la lista → **O(k·n)**.
Limitación: al comprobar si el código ordenaba las listas nos dimos cuenta, que de la forma que está implementado, tiene problemas a la hora de ordenar números negativos, por lo que habría que adaptarlo si la lista contiene números menores a 0.

Sorted:
Caso promedio y **peor caso**: Python divide la lista en sublistas ordenadas (“runs”) y las combina eficientemente, realiza aproximadamente Nlog N comparaciones y movimientos → **O(n log n)**
Mejor caso: si la lista ya está parcialmente ordenada, se detectan los “runs” y reduce el número de comparaciones, llegando a → **O(n)**
Limitaciones: Sorted siempre devuelve una nueva lista, por lo que se realiza un recorrido adicional de todos los elementos para copiar → **O(n)**

Gráficas comparativas de los tiempos de ejecución



La línea azul ordenamiento_buruja muestra un crecimiento cuadrático con el tamaño de la lista, confirmando que la operación tiene complejidad **$O(n^2)$.**
La línea naranja quicksort muestra un crecimiento mucho más lento y casi lineal-logarítmico, lo que coincide con la complejidad promedio de **$O(n \log n)$**.
La línea verde ordenamiento_por_residuo, muestra un crecimiento casi plano y muy lento, similar al de quicksort, pero con un tiempo de ejecución aún menor. Esto se debe a su complejidad de **$O(k \cdot n)$**
La línea roja sorted se mantiene prácticamente constante a lo largo de todos los tamaños de lista, dando la impresión de que la operación tiene complejidad **$O(1)$**. Sin embargo, su complejidad teórica sigue siendo **$O(n \log n)$**
Es el algoritmo más rápido de los que analizamos. Su eficiencia se basa en la combinación de mergesort e insertion sort, permitiéndole aprovechar las secuencias ya ordenadas en la lista.
Las gráficas validan el comportamiento teórico esperado.