**¿Cuál es el orden de complejidad O de cada algoritmo?**

**1. Bubble Sort**

* **Orden de complejidad:** O(n2)
* **Justificación:** en el código de bubble\_sort, se utilizan **dos bucles anidados**: el externo se ejecuta n−1 veces y el interno hasta i veces en cada iteración, lo que da lugar a una cantidad de comparaciones que crece cuadráticamente con el tamaño de la lista. No hay ninguna condición que permita salir antes si la lista ya está ordenada, por lo que siempre se hacen las mismas comparaciones. Esto lo convierte en un algoritmo de orden O(n2), independientemente del caso. Su eficiencia es baja para listas grandes.

**2. Quick Sort**

* **Orden de complejidad:** O(n log n)
* **Justificación:** el algoritmo divide recursivamente la lista en dos sublistas en torno a un **pivote**, y ordena cada parte. En cada nivel de recursión, el trabajo hecho es proporcional a **n** (comparaciones para particionar). Como en promedio el pivote divide la lista en mitades, la profundidad de la recursión será de orden **n log n**, dando una complejidad total de **O(n log n)**.

**3. Radix Sort**

* **Orden de complejidad:** O(n⋅k)
* **Justificación:** se aplica una especie de ordenamiento por conteo (Counting Sort) para cada dígito del número, de derecha a izquierda. La cantidad de dígitos (k) es **constante y pequeña** si se trata de números de tamaño fijo (por ejemplo, de 5 dígitos). Por lo tanto, el algoritmo hace **k** pasadas por la lista de longitud **n**, y cada pasada toma tiempo lineal **O(n)**, resultando en una complejidad total de **O(n⋅k)**.

**Comparación con la función built-in de python sorted. ¿Cómo funciona la misma?**

La función sorted() de Python devuelve una nueva lista ordenada a partir de cualquier iterable sin modificar el original. Internamente, sorted utiliza un algoritmo llamado **Timsort**, que es una combinación de **Merge Sort** y **Insertion Sort**. Este algoritmo fue diseñado para ser eficiente con datos reales y adaptarse al orden parcial de los elementos.

Su complejidad en el **peor caso es O(n log n)**, al igual que en el **caso promedio**, pero puede llegar a ser **O(n)** en el mejor de los casos cuando la lista ya está parcialmente ordenada.

En la práctica, sorted suele superar a la mayoría de los algoritmos de ordenamiento implementados manualmente, debido a que está altamente optimizado a nivel de bajo nivel en el intérprete de Python (escrito en C). **Por eso, se recomienda usar sorted para tareas reales de ordenamiento cuando no se necesita implementar el algoritmo uno mismo.**