Análisis de Eficiencia de Algoritmos de Ordenamiento - Grupo 5

Integrantes:

Carlos Fernando Briones Jarquín

Carlos Adrian Larios Nuñez

Roderick Uziel Caldera Tórrez

SIS0108 - Algoritmos y Estructuras de Datos - Grupo 6

Universidad Americana (UAM)

Ing. Jose Duran Garcia

18 de abril de 2025

Introducción

El análisis de la eficiencia de algoritmos de ordenamiento es un pilar fundamental en el estudio de algoritmos y estructuras de datos, ya que permite evaluar y comparar el desempeño de diferentes métodos para organizar colecciones de datos.

En el contexto de la programación en Python, un lenguaje ampliamente utilizado por su versatilidad y claridad, comprender cómo medir la eficiencia de estos algoritmos resulta esencial para optimizar aplicaciones en áreas como bases de datos, inteligencia artificial y análisis de datos.

Esta investigación, titulada Análisis de Eficiencia de Algoritmos de Ordenamiento, explora los conceptos clave para evaluar la eficiencia mediante la notación Big O, los tipos de complejidad (temporal y espacial), la definición de algoritmos de ordenamiento y los casos de análisis (mejor, peor y promedio).

Además, se examinan algoritmos representativos como Bubble Sort, Insertion Sort y Quick Sort, destacando sus características, ventajas y limitaciones. A través de un enfoque teórico respaldado por bibliografía especializada, este trabajo busca proporcionar una base sólida para entender y seleccionar algoritmos de ordenamiento según las necesidades específicas de cada problema, contribuyendo al desarrollo de soluciones computacionales más eficientes.

Medición de la Eficiencia de un Algoritmo

La eficiencia de un algoritmo se refiere a la cantidad de recursos computacionales que requiere para resolver un problema, específicamente en términos de tiempo (número de operaciones) y espacio (memoria utilizada). La herramienta principal para medir esta eficiencia es la notación Big O, que describe el comportamiento asintótico del algoritmo a medida que el tamaño de la entrada (n) crece hacia el infinito.

La notación Big O proporciona una cota superior del crecimiento de una función, enfocándose en el peor caso (escenario más desfavorable). No mide el tiempo exacto en

segundos, sino cómo el tiempo o espacio aumenta con el tamaño de la entrada. Por ejemplo, un algoritmo con $O(n^2)$ requiere un tiempo que crece cuadráticamente, mientras que uno con $O(n \log n)$ es más eficiente para grandes n.

Ejemplos de Notación Big O

Complejidad	**Descripción**	**Ejemplo**
O(1)	Tiempo constante, independiente de n	Acceso a un elemento de un arreglo
O(log n)	Tiempo logarítmico, divide el problema	Búsqueda binaria
O(n)	Tiempo lineal, crece con n	Recorrer una lista
O(n log n)	Tiempo linealitmético, eficiente	Merge Sort, Quick Sort (promedio)
$O(n^2)$	Tiempo cuadrático, crece exponencialmente	Bubble Sort, Insertion Sort (peor caso)
O(2^n)	Tiempo exponencial, muy lento	Algoritmos de fuerza bruta
O(n!)	Tiempo factorial, extremadamente lento	Permutaciones completas

(Peña, 2021)

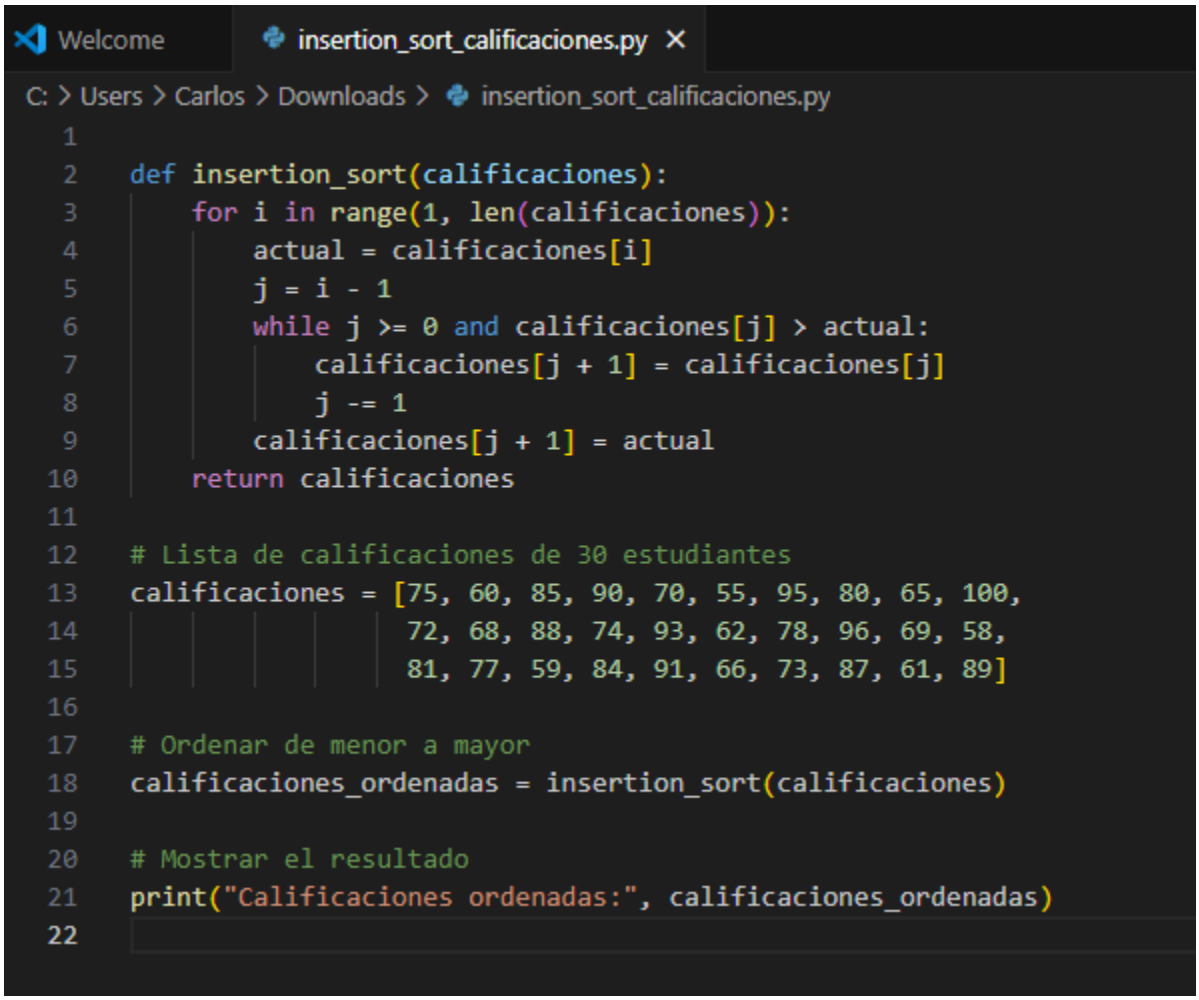
La notación Big O es universal y aplica a cualquier lenguaje de programación, incluyendo Python, lo que permite comparar algoritmos de manera independiente del hardware o software utilizado. (Fernandez, 2025)

Caso práctico

Un ejemplo práctico de uso de algoritmos de ordenación sería ordenar las calificaciones de estudiantes de un salón. Se desea ordenar de menor a mayor para determinar qué alumnos obtuvieron el mejor rendimiento.

Para este caso sería conveniente utilizar un algoritmo Insertion Sort para ordenar las calificaciones de los estudiantes, debido que es muy simple de implementar y es adaptable. En el caso de que sea un salón de clases promedio de 30 alumnos será conveniente aplicar este algoritmo. En el caso de que sea una cantidad grande de datos para ordenar, por ejemplo más de 1000, no sería apto utilizar este método de ordenación.

Un ejemplo de uso para este caso aplicado en programación sería de la siguiente manera:



Este algoritmo recorre la lista desde el segundo elemento, compara cada valor con los anteriores y lo coloca en su posición correcta desplazando los elementos mayores una posición a la derecha. Luego, se crea una lista con 30 calificaciones de estudiantes y se pasa a la función para ser ordenada. Finalmente, el programa imprime la lista de calificaciones ya ordenada, lo que permite identificar fácilmente los puntajes más bajos y más altos.

Análisis de Rendimiento

Para realizar un análisis completo de rendimiento, se evalúan la **complejidad temporal** y **complejidad espacial** del algoritmo Insertion Sort en los diferentes casos:

- **Complejidad Temporal**:
    - **Mejor Caso**: O(n), ocurre cuando la lista ya está ordenada. En este escenario, cada elemento se compara una vez y no requiere desplazamientos, lo que minimiza las operaciones. Para 30 calificaciones, el tiempo sería proporcional al número de elementos.
    - **Peor Caso**: $O(n^2)$, sucede cuando la lista está ordenada en orden inverso (de mayor a menor). Cada elemento requiere el máximo número de comparaciones y desplazamientos. Para 30 elementos, esto implica aproximadamente $30^2 = 900$ operaciones, lo que sigue siendo manejable pero ineficiente para listas más grandes.
    - **Caso Promedio**: $O(n^2)$, asumiendo una distribución aleatoria de calificaciones. El algoritmo realiza un número intermedio de comparaciones y desplazamientos, resultando en un rendimiento cuadrático promedio.
- **Complejidad Espacial**: O(1), ya que Insertion Sort es un algoritmo in-place. No utiliza memoria adicional más allá de unas pocas variables (índices y el elemento actual), lo que lo hace eficiente en términos de memoria incluso con 30 elementos.
- **Evaluación para el Caso Práctico**: Con 30 calificaciones, el algoritmo es práctico y eficiente debido a su baja complejidad espacial y la aceptable complejidad temporal para tamaños pequeños. Sin embargo, su rendimiento disminuye significativamente con listas más grandes (por ejemplo, 1000 elementos), donde el número de operaciones crecería a

1,000,000 en el peor caso, haciendo que algoritmos como Quick Sort (O(n log n)) sean más adecuados.

- **Optimización y Limitaciones**: Para mejorar el rendimiento con listas mayores, se podría combinar Insertion Sort con un algoritmo más eficiente (como en Timsort de Python) para sublistas pequeñas. Sin embargo, para este caso específico de 30 estudiantes, la simplicidad y el bajo uso de memoria justifican su uso.

Referencias

Fernandez, O. (2025, enero 14). *Big-O Para Principiantes*. Aprender BIG DATA. Retrieved junio 18, 2025, from https://aprenderbigdata.com/big-o/

Peña, M. J. (2021, junio 29). *Tipos de Algoritmos de Ordenación en Python - Másteres Online Nº 1 Empleabilidad*. Escuela Internacional de Posgrados. Retrieved junio 18, 2025, from https://eiposgrados.com/blog-python/tipos-de-algoritmos-de-ordenacion-en-python/