INSTITUTO TECNOLÓGICO SUPERIOR DE LA
REGIÓN DE LOS LLANOS

INGENIERÍA MECATRÓNICA

Programación Avanzada

Actividad: REPORTE DE PROGRAMA

Evaluación de Métodos de Ordenamiento.

Nombre: Bryan Emmanuel Barboza Carrillo

Docente: Osbaldo Aragón Banderas

Fecha: 2026-02-15

TECNM
TECNOLÓGICO NACIONAL DE
MÉXICO

Índice

1. Introducción

Existen diversos algoritmos de ordenamiento, cada uno con características particulares en términos de complejidad temporal y comportamiento ante distintos tipos de entrada. Entre los más conocidos se encuentran **Bubble Sort**, un algoritmo sencillo basado en comparaciones sucesivas e intercambios adyacentes, y **Quicksort**, un método más sofisticado que emplea una estrategia de partición y divide el problema en subproblemas más pequeños. Aunque ambos cumplen la misma función, su eficiencia difiere considerablemente cuando el tamaño de los datos aumenta.

Desde el punto de vista teórico, Bubble Sort presenta una complejidad temporal cuadrática $O(n^2)$, lo que implica que su tiempo de ejecución crece rápidamente conforme aumenta el número de elementos. Por otro lado, Quicksort posee una complejidad promedio $O(n \log n)$, lo que lo convierte en una alternativa mucho más eficiente para conjuntos de datos grandes. Sin embargo, más allá del análisis teórico, resulta fundamental validar experimentalmente estas diferencias mediante mediciones controladas.

En el caso de un robot seguidor de línea entrenado mediante técnicas de inteligencia artificial, la eficiencia algorítmica adquiere una importancia crítica. Este tipo de sistema requiere procesar continuamente datos provenientes de sensores para ajustar la velocidad de los motores y optimizar la trayectoria en tiempo real.

El presente trabajo tiene como objetivo implementar y evaluar experimentalmente los algoritmos Bubble Sort y Quicksort en el lenguaje de programación Python, utilizando Visual Studio Code como entorno de desarrollo. A través de pruebas controladas con distintos tamaños de entrada y escenarios de datos, se comparará su rendimiento, se analizará su escalabilidad y se relacionarán los resultados obtenidos con la complejidad teórica esperada, destacando su impacto en aplicaciones prácticas como la robótica.

2. Desarrollo

Entorno de desarrollo

El desarrollo del presente trabajo se realizó utilizando Visual Studio Code como entorno de programación, configurado con la extensión oficial de Python de Microsoft. Se empleó Python como lenguaje de implementación debido a su claridad sintáctica, facilidad para realizar pruebas experimentales y disponibilidad de herramientas estándar para medición de rendimiento.

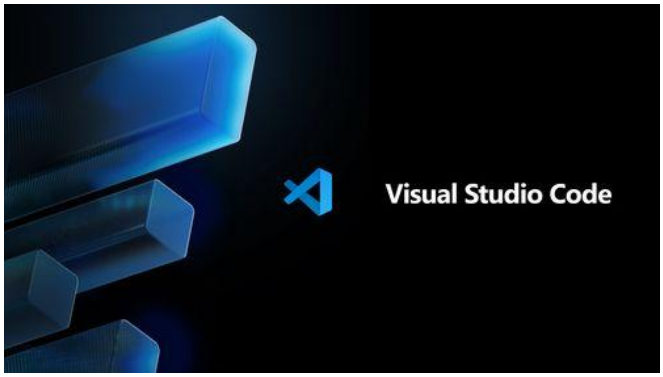

***Figura 1**. Entorno Visual Studio Code*

Las pruebas fueron ejecutadas en un entorno local bajo sistema operativo Windows. Para garantizar una ejecución controlada y organizada del proyecto, se utilizó un entorno virtual (venv), permitiendo aislar las dependencias del sistema. No se requirieron librerías externas, ya que las mediciones se realizaron utilizando módulos estándar de Python como timeit, statistics, random y csv.

El proyecto fue estructurado en carpetas separadas para el código fuente y los resultados experimentales, siguiendo buenas prácticas de organización profesional. Además, se utilizó Git para el control de versiones y GitHub como repositorio público para documentar y evidenciar el desarrollo del trabajo.

Implementación de los algoritmos

Bubble Sort

El algoritmo Bubble Sort fue implementado como una función que recibe una lista de números y devuelve una nueva lista ordenada. Este método se basa en comparar pares de elementos adyacentes e intercambiarlos si se encuentran en el orden incorrecto. El proceso se repite múltiples veces hasta que no se requieren más intercambios.

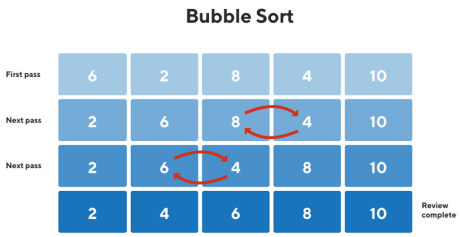

Figura 1.1 *Proceso ilustrativo del algoritmo Bubble Sort*

En cada iteración externa, el elemento de mayor valor se desplaza progresivamente hacia el final de la lista, reduciendo el rango de comparación en las siguientes pasadas. Para mejorar ligeramente su eficiencia, se incorporó una condición de optimización que detiene el algoritmo si durante una pasada completa no se realizan intercambios, lo que indica que la lista ya se encuentra ordenada.

Este algoritmo presenta una complejidad temporal promedio y en el peor caso de $O(n^2)$, ya que realiza comparaciones anidadas que dependen directamente del tamaño de la list

Quicksort

El algoritmo Quicksort fue implementado en su versión iterativa. Este método se basa en una estrategia de división y conquista, donde se selecciona un elemento como pivote y se reorganiza la lista de tal manera que los elementos menores queden a la izquierda y los mayores a la derecha. Posteriormente, el proceso se repite sobre los subarreglos resultantes.

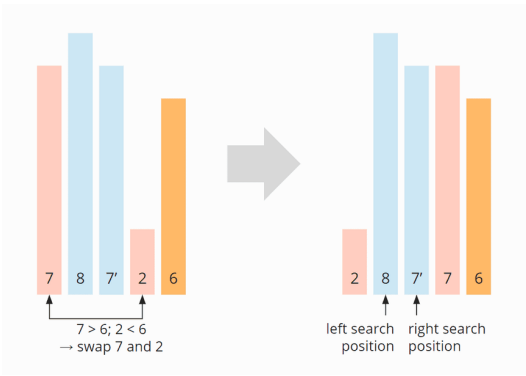

Figura 1.2 *Proceso ilustrativo de partición en el algoritmo Quicksort*

En lugar de utilizar una implementación recursiva tradicional, se optó por una versión iterativa mediante el uso de una pila explícita. Esta decisión se tomó para evitar posibles problemas asociados con el límite de recursión en Python cuando se trabaja con listas de tamaño considerable. Además, se utilizó un pivote aleatorio con el fin de reducir la probabilidad de que el algoritmo caiga repetidamente en su peor caso teórico.

En promedio, Quicksort presenta una complejidad temporal O(n log n), lo que lo convierte en una alternativa significativamente más eficiente que Bubble Sort para listas de gran tamaño. No obstante, en el peor escenario posible puede alcanzar una complejidad $O(n^2)$, aunque esta situación es poco frecuente cuando se emplea selección aleatoria del pivote.

Diseño experimental

Con el objetivo de evaluar el rendimiento de los algoritmos implementados, se diseñó un conjunto de pruebas controladas utilizando el módulo **timeit** de Python, el cual permite medir con precisión el tiempo de ejecución de fragmentos de código.

Se evaluaron cuatro tamaños de entrada representativos: 100, 1 000, 5 000 y 10 000 elementos. Estos valores fueron seleccionados para observar el comportamiento de los algoritmos tanto en conjuntos pequeños como en volúmenes de datos considerablemente mayores.

Para cada tamaño se consideraron dos escenarios distintos:

- Lista aleatoria, generada mediante números distribuidos sin orden específico.
- Lista invertida, organizada inicialmente en orden descendente, representando un escenario cercano al peor caso para algunos algoritmos.

Cada combinación de tamaño, escenario y algoritmo fue ejecutada cinco veces con el fin de reducir la variabilidad experimental y obtener resultados más confiables. A partir de estas repeticiones se calcularon dos métricas principales:

- Tiempo promedio de ejecución.
- Desviación estándar.

Los resultados fueron almacenados en un archivo CSV para su posterior análisis y presentación en formato tabular.

Resultados

Los tiempos de ejecución obtenidos para los algoritmos Bubble Sort y Quicksort se presentan en la Tabla 1. Las mediciones corresponden al promedio de cinco ejecuciones por combinación de tamaño de entrada y escenario de datos. Los tiempos están expresados en segundos.

Tabla 1

Tiempos promedio de ejecución de Bubble Sort y Quicksort en distintos tamaños y escenarios

Tamaño	Escenario	Algoritmo	Repeticiones	Promedio (s)	Desv. Estándar (s)
100	Aleatoria	Bubble Sort	5	0.000386	0.000007
100	Aleatoria	Quicksort	5	0.000116	0.000014
100	Invertida	Bubble Sort	5	0.000509	0.000012
100	Invertida	Quicksort	5	0.000107	0.000005
1000	Aleatoria	Bubble Sort	5	0.044731	0.000912
1000	Aleatoria	Quicksort	5	0.001433	0.000051
1000	Invertida	Bubble Sort	5	0.061903	0.001497
1000	Invertida	Quicksort	5	0.0017	0.000392
5000	Aleatoria	Bubble Sort	5	1.40892	0.092756
5000	Aleatoria	Quicksort	5	0.011045	0.00316
5000	Invertida	Bubble Sort	5	1.870214	0.075518
5000	Invertida	Quicksort	5	0.008811	0.000974
10000	Aleatoria	Bubble Sort	5	5.86692	0.398592
10000	Aleatoria	Quicksort	5	0.020033	0.002034
10000	Invertida	Bubble Sort	5	8.04852	0.547974
10000	Invertida	Quicksort	5	0.0265	0.005524

Discusión

1. ¿Cuál algoritmo escala mejor y por qué?

El algoritmo que mostró mejor escalabilidad fue Quicksort. A medida que el tamaño de entrada aumentó, el crecimiento del tiempo de ejecución de Quicksort fue considerablemente menor en comparación con Bubble Sort. En el caso de 10 000 elementos con lista invertida, Quicksort fue aproximadamente 300 veces más rápido que Bubble Sort. Esta diferencia se debe a que Quicksort divide el problema en subproblemas más pequeños, reduciendo el número total de comparaciones necesarias.

2. Relación con la complejidad esperada ($O(n^2)$ vs $O(n \log n)$)

Los resultados experimentales concuerdan con la teoría de complejidad temporal. Bubble Sort presenta una complejidad $O(n^2)$, lo que implica que el número de operaciones crece proporcionalmente al cuadrado del tamaño de la lista. Esto se refleja en el incremento drástico del tiempo al pasar de 1 000 a 10 000 elementos.

Por otro lado, Quicksort posee una complejidad promedio $O(n \log n)$, lo que permite que su tiempo crezca de manera mucho más controlada conforme aumenta el tamaño de entrada. El comportamiento observado en los datos experimentales confirma esta diferencia teórica.

3. Impacto práctico en sistemas robóticos con IA

En un robot seguidor de línea entrenado mediante inteligencia artificial, el tiempo de procesamiento es un factor crítico. El sistema debe analizar datos de sensores, evaluar estados y ajustar la velocidad de los motores en intervalos muy cortos. Si el procesamiento de datos incluye algoritmos con complejidad elevada como $O(n^2)$, el tiempo de respuesta podría incrementarse significativamente, afectando la estabilidad del control y la precisión del seguimiento.

3. Conclusión

El presente trabajo permitió implementar y evaluar experimentalmente dos algoritmos clásicos de ordenamiento: Bubble Sort y Quicksort. A través de pruebas controladas utilizando distintos tamaños de entrada y escenarios de datos, fue posible analizar su comportamiento práctico y compararlo con la complejidad teórica esperada.

Los resultados obtenidos confirmaron que Bubble Sort presenta un crecimiento cuadrático $O(n^2)$, lo que provoca incrementos significativos en el tiempo de ejecución conforme aumenta el número de elementos. En contraste, Quicksort demostró un comportamiento promedio $O(n \log n)$, manteniendo tiempos considerablemente menores incluso en listas de gran tamaño. En el escenario más exigente evaluado (10 000 elementos invertidos), Quicksort fue aproximadamente 300 veces más rápido que Bubble Sort, evidenciando una diferencia sustancial en escalabilidad.

Este análisis experimental refuerza la importancia de comprender la complejidad algorítmica al diseñar sistemas computacionales. En aplicaciones como robots seguidores de línea entrenados con inteligencia artificial, donde el procesamiento de datos y la toma de decisiones deben realizarse en tiempo real, la selección de algoritmos eficientes resulta fundamental para garantizar estabilidad, precisión y capacidad de respuesta.

En conclusión, aunque Bubble Sort es útil con fines educativos por su simplicidad, Quicksort representa una alternativa significativamente más adecuada para aplicaciones reales que requieren eficiencia y escalabilidad.