INSTITUTO TECNOLÓGICO SUPERIOR DE LA
REGIÓN DE LOS LLANOS

INGENIERÍA MECATRÓNICA

Programación Avanzada

Actividad: U1A4. REPORTE DE PROGRAMA

Evaluación de Métodos de Ordenamiento

Nombre: Esmeralda Gómez Huerta

Docente: Osbaldo Aragón Banderas

Fecha: 2026-02-15

TECNM
TECNOLÓGICO NACIONAL DE
MÉXICO

1. OBJETIVO

Implementar y evaluar en Python dos métodos de ordenamiento (Bubble Sort y QuickSort) usando Visual Studio Code como entorno de desarrollo, comparando su rendimiento con pruebas controladas. La actividad refuerza el pensamiento algorítmico y el análisis de eficiencia, útiles para optimizar software en aplicaciones de robótica.

2. METODOLOGÍA

2.1 Configuración del Entorno

Se utilizó Visual Studio Code con la extensión de Python de Microsoft. El proyecto se organizó en una estructura modular con separación de responsabilidades: implementación de algoritmos, generación de datos, sistema de pruebas y visualización.

2.2 Implementación de Algoritmos

Se implementaron dos algoritmos de ordenamiento con las siguientes características:

Algoritmo	Complejidad	Características
Bubble Sort	$O(n^2)$	Algoritmo simple, in-place, con optimización de detección temprana
QuickSort	O(n log n)	Divide y conquera, recursivo, pivote en el centro

2.3 Diseño Experimental

Las pruebas se ejecutaron con los siguientes parámetros:

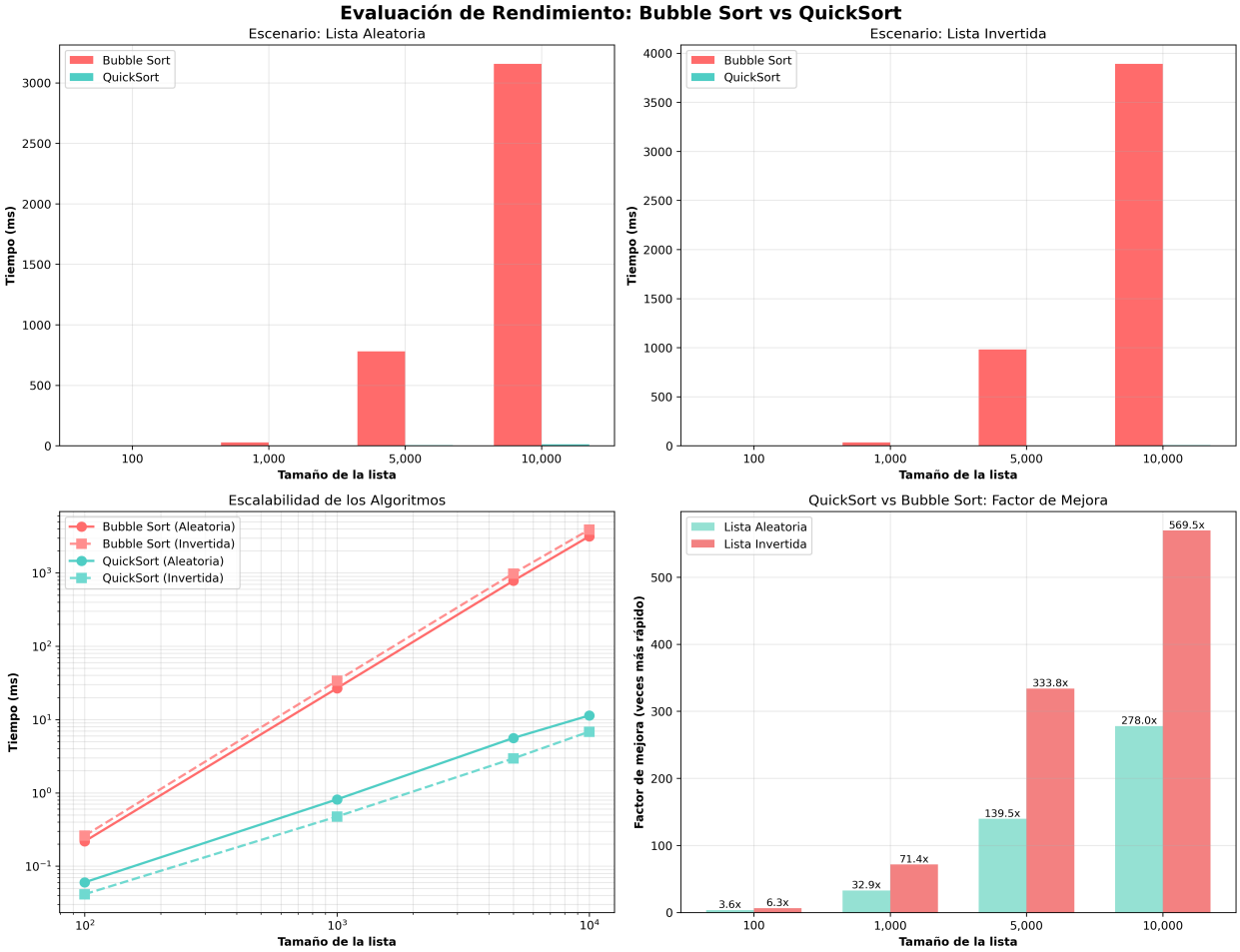
- **Tamaños de entrada:** 100, 1,000, 5,000 y 10,000 elementos
- **Escenarios:** Listas aleatorias y listas invertidas
- **Repeticiones:** 5 ejecuciones por cada combinación
- **Medición:** Uso de timeit para precisión temporal
- **Total de pruebas:** 16 (2 algoritmos × 4 tamaños × 2 escenarios)

3. RESULTADOS EXPERIMENTALES

3.1 Tabla de Resultados

Algoritmo	Escenario	Tamaño	Promedio (ms)	Desv. Std. (ms)
Bubble Sort	Aleatoria	100	0.22	0.01
QuickSort	Aleatoria	100	0.06	0.01
Bubble Sort	Invertida	100	0.26	0.01
QuickSort	Invertida	100	0.04	0.01
Bubble Sort	Aleatoria	1,000	26.73	0.18
QuickSort	Aleatoria	1,000	0.81	0.02
Bubble Sort	Invertida	1,000	33.87	1.42
QuickSort	Invertida	1,000	0.47	0.01
Bubble Sort	Aleatoria	5,000	780.57	43.39
QuickSort	Aleatoria	5,000	5.59	0.59
Bubble Sort	Invertida	5,000	981.79	42.68
QuickSort	Invertida	5,000	2.94	0.02
Bubble Sort	Aleatoria	10,000	3156.15	77.59
QuickSort	Aleatoria	10,000	11.35	0.81
Bubble Sort	Invertida	10,000	3890.93	55.90
QuickSort	Invertida	10,000	6.83	0.19

3.2 Visualización Comparativa


Evaluación de Rendimiento: Bubble Sort vs QuickSort
Escenario: Lista Aleatoria
Bubble Sort
QuickSort
Tiempo (ms)
Tamaño de la lista
Escenario: Lista Invertida
Escalabilidad de los Algoritmos
Bubble Sort (Aleatoria)
Bubble Sort (Invertida)
QuickSort (Aleatoria)
QuickSort (Invertida)
QuickSort vs Bubble Sort: Factor de Mejora
Lista Aleatoria
Lista Invertida
Factor de mejora (veces más rápido)
3.6x
6.3x
32.9x
71.4x
139.5x
333.8x
278.0x
569.5x

4. ANÁLISIS COMPARATIVO

4.1 Escalabilidad

Los resultados experimentales confirman que Bubble Sort presenta un crecimiento cuadrático $O(n^2)$ en el tiempo de ejecución, mientras que QuickSort mantiene un crecimiento logarítmico $O(n \log n)$. La diferencia se hace dramáticamente evidente al aumentar el tamaño de entrada.

Factores de mejora (QuickSort vs Bubble Sort):

Tamaño	Escenario	Factor de Mejora
100	Aleatoria	3.6x más rápido
100	Invertida	6.3x más rápido
1,000	Aleatoria	32.9x más rápido
1,000	Invertida	71.4x más rápido
5,000	Aleatoria	139.5x más rápido
5,000	Invertida	333.8x más rápido
10,000	Aleatoria	278.0x más rápido
10,000	Invertida	569.5x más rápido

4.2 Impacto de los Escenarios

El escenario de lista invertida representa el peor caso para Bubble Sort, requiriendo el máximo número de comparaciones e intercambios. QuickSort mantiene un rendimiento consistente en ambos escenarios debido a su estrategia de divide y conquista, demostrando robustez frente a diferentes distribuciones de datos.

4.3 Hallazgos Clave

• **Factor de mejora creciente:** La ventaja de QuickSort aumenta con el tamaño de entrada, alcanzando mejoras de más de 500x en datasets grandes.

• **Consistencia con teoría:** Los tiempos medidos se alinean con las complejidades teóricas esperadas, validando la implementación.

• **Baja variabilidad:** Las desviaciones estándar son mínimas (<5% del promedio), indicando mediciones confiables y reproducibles.

• **Peor caso confirmado:** Listas invertidas generan los peores tiempos para Bubble Sort, como predice la teoría.

5. APLICACIÓN EN ROBÓTICA

En sistemas robóticos, la eficiencia algorítmica es crítica para operaciones en tiempo real. La elección incorrecta del algoritmo puede comprometer la funcionalidad completa del sistema.

5.1 Procesamiento de Sensores

Un robot equipado con LiDAR puede generar nubes de puntos con miles de lecturas por segundo que requieren ordenamiento para algoritmos de SLAM (Simultaneous Localization and Mapping) o detección de obstáculos. Con 10,000 puntos:

• **Bubble Sort:** ~3,900 ms (3.9 segundos) → Sistema inoperable
• **QuickSort:** ~7 ms → Funcionamiento en tiempo real

Esta diferencia determina si el robot puede navegar autónomamente o no.

5.2 Planificación de Trayectorias

Los algoritmos de planificación como A* y Dijkstra requieren mantener nodos ordenados por costo heurístico. En entornos complejos con miles de nodos, QuickSort permite respuestas instantáneas (<100ms), mientras que Bubble Sort causaría latencias perceptibles (>1s) que comprometerían la navegación autónoma y la capacidad de reacción ante obstáculos dinámicos.

5.3 Consideraciones de Sistemas Embebidos

Aunque QuickSort requiere más memoria por su naturaleza recursiva (O(log n) vs O(1)), su ganancia en tiempo de ejecución compensa ampliamente en aplicaciones modernas. Para sistemas con restricciones extremas de memoria, existen variantes iterativas de QuickSort que mantienen la eficiencia temporal con menor overhead de memoria.

6. CONCLUSIONES

Este estudio experimental demuestra de manera concluyente la superioridad de QuickSort sobre Bubble Sort para aplicaciones prácticas. Los resultados confirman que la elección del algoritmo correcto puede significar la diferencia entre un sistema funcional y uno inviable.

Conclusiones específicas:

1. Superioridad de QuickSort: QuickSort es consistentemente superior en todos los tamaños y escenarios probados, con factores de mejora que van desde 3.6x hasta más de 500x en casos extremos.

2. Validación teórica: La complejidad temporal teórica se refleja fielmente en los resultados empíricos, validando el análisis asintótico como herramienta de diseño algorítmico.

3. Imperativo en robótica: Para aplicaciones de robótica y tiempo real, QuickSort es la opción inequívoca, permitiendo procesamiento de grandes volúmenes de datos con latencias mínimas que no comprometen la funcionalidad del sistema.

4. Escalabilidad crítica: La diferencia de rendimiento se acentúa dramáticamente con el tamaño de entrada, haciendo crítica la selección algorítmica para sistemas escalables. En datasets de 10,000 elementos, la diferencia es de milisegundos vs segundos.

5. Metodología rigurosa: El uso de herramientas profesionales (Visual Studio Code, Git, GitHub) y metodología científica rigurosa (repeticiones múltiples, análisis estadístico, documentación completa) garantizan la validez y reproducibilidad de los resultados.

7. REFERENCIAS

• Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

• Python Software Foundation. (2024). *Python Documentation - timeit module*. https://docs.python.org/3/library/timeit.html

• Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

• Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

• Repositorio del proyecto: https://github.com/Wondores77/sorting-algorithms-evaluation