INSTITUTO TECNOLÓGICO SUPERIOR DE LA
REGIÓN DE LOS LLANOS

INGENIERÍA MECATRÓNICA

Ciberfisca

Actividad: U1A4. EJERCICIO DE APLICACIÓN.

Nombre: Angel Daniel Cisneros Perez

Docente: Osbaldo Aragón Banderas

Fecha: 2026-02-16

TECNM
TECNOLÓGICO NACIONAL DE
MÉXICO

Índice

1) Introducción

En esta actividad se implementaron y evaluaron dos métodos de ordenamiento en Python: Burbuja (Bubble Sort) y Quicksort. El objetivo fue comparar su rendimiento bajo pruebas controladas utilizando diferentes tamaños de entrada y dos escenarios (lista aleatoria e invertida), con el fin de relacionar resultados empíricos con la complejidad esperada y discutir su impacto práctico en software orientado a robótica.

2) Desarrollo del proyecto

2.1 Estructura y entorno de desarrollo (VS Code)

El proyecto se desarrolló en Visual Studio Code utilizando la extensión oficial de Python. Se organizó el repositorio separando el código fuente en la carpeta src y la documentación en docs, facilitando la mantenibilidad y la publicación posterior en GitHub.

- src/algoritmos.py: implementaciones de Bubble Sort y Quicksort.
- src/main.py: generación de datos, ejecución de pruebas con timeit y reporte en consola.
- docs/: carpeta destinada al reporte final en PDF y evidencias.
- README.md: instrucciones de ejecución, estructura y resumen de resultados.

2.2 Implementación de algoritmos

Bubble Sort se implementó con una optimización de parada temprana: si en una pasada no ocurren intercambios, la lista ya está ordenada. Esto mejora casos favorables (casi ordenados), aunque su complejidad sigue siendo $O(n^2)$.

Quicksort se implementó de forma recursiva por legibilidad y por su enfoque divide y vencerás. Se eligió el primer elemento como pivote (decisión simple), lo cual es importante porque puede degradar el rendimiento en listas invertidas (caso desfavorable).

2.3 Generación de datos de prueba

Se generaron listas de tamaños 100, 1000, 5000 y 10000 en dos escenarios: (1) aleatoria (valores enteros pseudoaleatorios) y (2) invertida (orden estrictamente descendente), para observar tanto comportamiento típico como un caso más exigente.

3) Metodología de pruebas de rendimiento

Las mediciones se realizaron con la librería timeit, ejecutando 5 repeticiones por cada combinación de tamaño, escenario y algoritmo. Para cada conjunto se calculó el promedio y la desviación estándar, proporcionando una estimación más estable del rendimiento y su variabilidad.

- Herramienta de medición: timeit.repeat (number=1, repeat=5).
- Métrica reportada: tiempo promedio (s) y desviación estándar.
- Control de comparabilidad: se mide cada algoritmo sobre el mismo conjunto base por tamaño y escenario.

4) Resultados

La Tabla 1 resume los tiempos promedio y la desviación estándar obtenidos en las pruebas. Se observa una diferencia de escalabilidad marcada entre ambos algoritmos conforme crece el tamaño de entrada.

Tamaño	Escenario	Algoritmo	Tiempo promedio (s)	Desviación estándar
100	aleatoria	Burbuja	0.000431	0.000019
100	aleatoria	Quicksort	0.000150	0.000030
100	invertida	Burbuja	0.000497	0.000055
100	invertida	Quicksort	0.000379	0.000027
1000	aleatoria	Burbuja	0.045403	0.007548
1000	aleatoria	Quicksort	0.001008	0.000059
1000	invertida	Burbuja	0.055246	0.021060
1000	invertida	Quicksort	0.038626	0.002550
5000	aleatoria	Burbuja	1.357675	0.051665
5000	aleatoria	Quicksort	0.009379	0.000274
5000	invertida	Burbuja	1.708948	0.106137
5000	invertida	Quicksort	0.888121	0.018553
10000	aleatoria	Burbuja	5.509475	0.212112
10000	aleatoria	Quicksort	0.019214	0.002519
10000	invertida	Burbuja	6.653699	0.166408
10000	invertida	Quicksort	3.699714	0.126259

Tabla 1. Tiempos de ejecución para Bubble Sort y Quicksort bajo escenarios aleatorio e invertido (5 repeticiones).

5) Análisis comparativo

5.1 Escalabilidad y complejidad

Bubble Sort presenta crecimiento cuadrático $O(n^2)$, por lo que al aumentar el tamaño de entrada su tiempo se incrementa rápidamente. En cambio, Quicksort suele comportarse como O(n log n) en promedio, manteniendo tiempos bajos incluso para 10,000 elementos en el escenario aleatorio.

5.2 Caso desfavorable de Quicksort por selección de pivote

En la lista invertida, Quicksort se degrada de forma notable porque el pivote fijo (primer elemento) induce particiones muy desbalanceadas. Esto aproxima el comportamiento al peor caso $O(n^2)$, lo cual explica el incremento fuerte del tiempo para 10,000 elementos invertidos.

5.3 Conclusión del análisis

Empíricamente, Quicksort domina en escenarios típicos; sin embargo, su implementación debe considerar estrategias de pivote (por ejemplo, pivote aleatorio o mediana de tres) para evitar degradaciones en entradas estructuradas.

6) Impacto práctico en robótica

En robótica, el tiempo de cómputo afecta directamente la capacidad de respuesta del sistema: control, navegación y procesamiento de sensores requieren decisiones rápidas. Un algoritmo $O(n^2)$ puede volverse inviable si el tamaño de datos crece o si el ordenamiento se ejecuta repetidamente dentro de un lazo de control. Por ello, elegir algoritmos con mejor escalabilidad reduce latencia y permite mayor frecuencia de actualización en tareas críticas.

- Menor latencia → mejor toma de decisiones en tiempo real.
- Procesamiento más rápido → mayor frecuencia de muestreo/actualización.
- Mayor robustez → evitar peores casos con pivotes mejorados en Quicksort.

7) Evidencias

7.1 Estructura del proyecto en VS Code

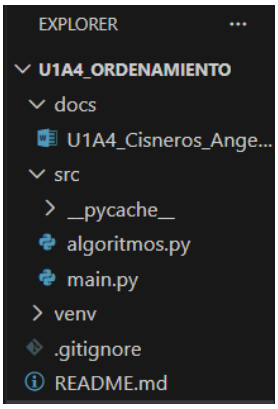

Fig 1. Estructura del proyecto U1A4 en Visual Studio Code. Se observa la organización en carpetas src/ y docs/, así como los archivos algoritmos.py, main.py, README.md y .gitignore.

7.2 Ejecución del programa y pruebas de rendimiento

```
PS C:\Users\Angel Daniel\OneDrive\Documentos\ProgramacionAvanzada\U1A4_Ordenamiento> python src/main.py
Iniciando pruebas de rendimiento... (Esto puede tardar unos minutos)
Tamaño     | Escenario  | Algoritmo    | Tiempo Promedio (s)  | Desviación Estándar
--------------------------------------------------------------------------------
100        | aleatoria  | Burbuja      | 0.000431 s           | 0.000019
100        | aleatoria  | Quicksort    | 0.000150 s           | 0.000030
100        | invertida  | Burbuja      | 0.000497 s           | 0.000055
100        | invertida  | Quicksort    | 0.000379 s           | 0.000027
1000       | aleatoria  | Burbuja      | 0.045403 s           | 0.007548
1000       | aleatoria  | Quicksort    | 0.001008 s           | 0.000059
1000       | invertida  | Burbuja      | 0.055246 s           | 0.021060
1000       | invertida  | Quicksort    | 0.038626 s           | 0.002550
5000       | aleatoria  | Burbuja      | 1.357675 s           | 0.051665
5000       | aleatoria  | Quicksort    | 0.009379 s           | 0.000274
5000       | invertida  | Burbuja      | 1.708948 s           | 0.106137
5000       | invertida  | Quicksort    | 0.888121 s           | 0.018553
10000      | aleatoria  | Burbuja      | 5.509475 s           | 0.212112
10000      | aleatoria  | Quicksort    | 0.019214 s           | 0.002519
```

Fig 2. Evidencia de ejecución del script principal (python src/main.py) mostrando tamaños evaluados, escenarios, tiempos promedio y desviación estándar.

8) Publicación en GitHub

Para completar la evidencia profesional, el repositorio público debe incluir el código fuente organizado, el reporte final en PDF dentro de docs y un README con instrucciones claras de ejecución y resumen de resultados.

- Repositorio público con estructura limpia (src/, docs/, README.md, .gitignore).
- Reporte final exportado a PDF dentro de docs/.
- Evidencias (capturas/tablas) incluidas en docs/ o enlazadas desde README.
- Instrucciones de ejecución: python src/main.py

9) Conclusión técnica

La evaluación experimental permitió contrastar empíricamente el comportamiento de Bubble Sort y Quicksort bajo distintos tamaños y escenarios de entrada. Los resultados confirman que la complejidad teórica se refleja claramente en la práctica: mientras Bubble Sort exhibe crecimiento cuadrático $O(n^2)$, Quicksort mantiene un comportamiento cercano a $O(n \log n)$ en escenarios promedio, logrando tiempos significativamente menores conforme aumenta el tamaño de los datos.

Sin embargo, también se evidenció que la selección de pivote influye de manera crítica en el rendimiento de Quicksort. En listas invertidas, el uso del primer elemento como pivote genera particiones desbalanceadas, aproximando el algoritmo a su peor caso $O(n^2)$. Este comportamiento resalta la importancia de decisiones de diseño en la implementación, más allá de la complejidad teórica del algoritmo.

Desde una perspectiva aplicada a la robótica y sistemas embebidos, la eficiencia algorítmica impacta directamente en la capacidad de respuesta del sistema. Procesamientos ineficientes pueden incrementar la latencia en la toma de decisiones, afectar la frecuencia de actualización de datos y comprometer la estabilidad del control. Por tanto, seleccionar estructuras y algoritmos adecuados no es únicamente una cuestión académica, sino un factor determinante en el desempeño real del software.

En conclusión, Quicksort resulta ampliamente superior para conjuntos de datos medianos y grandes en condiciones promedio, pero su implementación debe considerar estrategias robustas de selección de pivote para evitar degradaciones. La práctica refuerza la importancia del análisis empírico como complemento indispensable del análisis teórico de complejidad.

10) Anexo: fragmentos de código relevantes

10.1 Bubble Sort (fragmento)

```
def bubble_sort(lista):
    n = len(lista)
    lista_ordenada = lista.copy()
    for i in range(n):
        intercambio = False
        for j in range(0, n - i - 1):
            if lista_ordenada[j] > lista_ordenada[j + 1]:
                lista_ordenada[j], lista_ordenada[j + 1] = lista_ordenada[j +
1], lista_ordenada[j]
                intercambio = True
        if not intercambio:
            break
    return lista_ordenada
```

10.2 Quicksort (fragmento)

```
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)
```