Análisis de Algoritmos y Estructuras de Datos
CI0116 – Grupo 4

Andrew Umaña Quirós – B37091
Brandon Alfaro Saborio - C4C251

Tarea 1

Promedios para Bubble Sort

Promedios de tiempos de ejecución para las diferentes implementaciones:

Table 1: Tiempos de ejecución de Bubble Sort (en segundos)

Implementación	Hilera	Ejec. 1	Ejec. 2	Ejec. 3	Ejec. 4	Ejec. 5	Promedio
Personal	10	0.000006	0.000005	0.000010	0.000057	0.000009	0.000017
	100	0.000309	0.000292	0.000287	0.000292	0.000286	0.000293
	1000	0.036207	0.036390	0.036572	0.035554	0.034371	0.035819
ChatGPT	10	0.000006	0.000004	0.000003	0.000004	0.000004	0.000004
	100	0.000283	0.000305	0.000304	0.000309	0.000299	0.000300
	1000	0.034625	0.033988	0.033303	0.033687	0.033940	0.033909
Gemini	10	0.000005	0.000004	0.000004	0.000004	0.000004	0.000004
	100	0.000289	0.000290	0.000295	0.000286	0.000291	0.000290
	1000	0.034763	0.036377	0.034498	0.034275	0.034858	0.034954

En la tabla 1 se denotan los tiempos obtenidos al correr los 3 programas en el mismo hardware, estos están divididos por número de ejecución y por el programa utilizado; el programa hecho por nosotros, un programa/algoritmo de bubble sort optimizado por ChatGPT–5 y un programa/algoritmo de bubble sort optimizado por Gemini.

Para la versión optimizada por ChatGPT-5:

Optimización 1: Uso de un flag de intercambio (swapped)
Cambio: Añade una variable swapped que se reinicia en cada pasada.

Si en una iteración se completa y no hay intercambios, el algoritmo se detiene inmediatamente.

Mejora: Permite finalizar el ordenamiento antes de completar todas las pasadas en listas que ya están ordenadas o casi ordenadas, reduciendo el tiempo de ejecución. El algoritmo clásico siempre recorre todas las pasadas, incluso si ya estaba ordenado.

Optimización 2: Reducción del rango de comparación en cada pasada

Cambio: En cada iteración externa, se reduce el rango de comparación usando un loop for j in range (0, n - i - 1), ya que el último elemento de la iteración anterior ya está en su lugar.

Mejora: Se eliminan las comparaciones con elementos que ya están ordenados ya que no se necesitan, optimizando la eficiencia sin alterar la lógica base del algoritmo Bubble Sort.

Para la versión optimizada por Gemini:

Optimización 1: Uso de un bucle e implementación de un flag (hubo_intercambio):.

Cambio: En lugar de un bucle for externo que siempre ejecuta n pasadas, se usa un bucle while que solo se ejecuta si la bandera hubo_intercambio es True.

Mejora: Esto permite que el algoritmo se detenga tan pronto como la lista esté ordenada. En el caso de una lista ya ordenada o casi ordenada, el algoritmo solo hará una o dos pasadas y saldrá, lo que reduce drásticamente el número de comparaciones. El algoritmo original siempre hace n pasadas, sin importar si la lista ya está ordenada.

Optimización 2:
Reducción del Rango de Búsqueda (n -= 1):

Cambio: Después de cada iteración completa del loop interno, el elemento más grande "burbujea" hasta el final de la sublista no ordenada.
Al final de la pasada, este elemento está en su lugar correcto.

Mejora: En cada iteración del bucle while, se decrementa el tamaño de la lista a recorrer (n -= 1). Esto evita comparaciones innecesarias con los elementos que ya están en su lugar. Esta optimización estaba presente de manera similar en el código básico, pero en esta versión, se gestiona explícitamente y se combina con el bucle while para una mayor eficiencia.