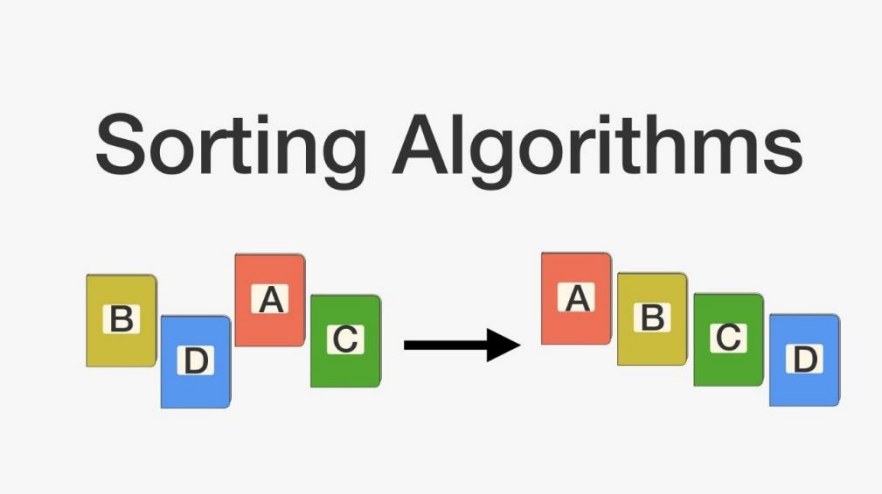
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

INFORME 1: ALGORITMOS DE ORDENAMIENTO Y DE MULTIPLICACIÓN DE MATRICES



Nombre: Javiera Jaret Layana Sepúlveda
Rol: 202073075-K
Profesor(a): Raquel Pezoa
Paralelo: 2
Fecha: 10 de septiembre del 2024
Curso: Algoritmos y Complejidad 2-2024
Departamento: Ingeniería Civil Informática

Contenido

Introducción. 3
Descripción de Algoritmos. 4
Descripción de Data Sets Algoritmos De Ordenamiento. 6
Resultados Experimentales Algoritmos de Ordenamiento. 8
Descripción de Data Sets Algoritmos de Multiplicación de Matrices. 15
Resultados Experimentales Algoritmos de Multiplicación de Matrices. 17
Conclusiones. 23
Referencias 24

Introducción.

Este informe se ocupa de la evaluación experimental de distintos tipos de algoritmos de ordenamiento y de multiplicación de matrices en el lenguaje de programación Python.

Para ello, se implementaron los siguientes algoritmos de ordenamientos:

- BubbleSort (), MergeSort () y QuickSort ()

Además, se utilizó la función de biblioteca estándar, específicamente el método sort (), que utiliza variantes eficientes como Timsort para ordenar grandes conjuntos de datos.

En cuanto a los algoritmos de multiplicación de matrices, se implementaron los siguientes:

- Algoritmo iterativo cúbico tradicional, Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos y Algoritmo de Strassen.

Todo esto nos permitió realizar comparaciones en términos de tiempo de ejecución.

Para llevar a cabo los experimentos, se generaron diversos conjuntos de datos para los algoritmos de ordenamiento, incluyendo Datos aleatorios, semiordenados y parcialmente ordenados. De igual manera, se trabajó con matrices de diferentes características, entre ellas, cuadráticas y no cuadráticas de igual y distinta dimensión.

Cada uno de estos tipos de datos permitió observar cómo los algoritmos reaccionaban ante variaciones en el tipo y tamaño de los datos.

El análisis se llevó a cabo mediante la implementación de herramientas de profiling, midiendo el tiempo de ejecución de cada algoritmo bajo diferentes escenarios. Esto permitió evaluar la eficiencia de los algoritmos, así como analizar el impacto de factores de implementación, como la preservación de la **localidad de los datos** en la memoria y la optimización de algoritmos.

Este informe también aborda los casos donde los algoritmos de multiplicación de matrices no pueden ser implementados debido al no cumplimiento de condiciones.

Por otra parte, los resultados preliminares sugieren que la elección del algoritmo correcto depende no solo del tamaño del conjunto de datos o de las matrices, sino también de las características específicas de los datos.

En las siguientes secciones del informe se detallan los resultados experimentales específicos y los gráficos que ilustran el comportamiento de los algoritmos en distintos escenarios, proporcionando un análisis comparativo detallado de sus rendimientos.

Descripción de Algoritmos.

Para poder entender a futuro de mejor manera los experimentos realizados para cada tipo de algoritmo es que en esta sección describiremos de forma general cada algoritmo, su complejidad temporal y como funciona. Además, se podrá acceder a los códigos a implementar de ordenamiento y multiplicación de matrices en el siguiente repositorio en la carpeta ***Codigos***: https://github.com/Javiera9920/Tareas-INF221/tree/main/Tarea1-INF221-JavieraLayana-202073075K

- **Bubble Sort:** Es un **algoritmo de ordenamiento simple**, recorre una lista repetidamente intercambiando los elementos adyacentes si están en el orden incorrecto, dejando el valor más alto en la última posición de la lista ordenada. Recorre **n-1** veces la lista y realiza **n*n(n-1) /2** comparaciones. Es poco útil para una gran cantidad de datos, ya que su complejidad temporal es muy alta para los casos promedios y para el peor caso.
  Este algoritmo es fácil de entender e implementar, sin embargo, al estar basado en comparaciones requiere de un operador de comparación para determinar el orden relativo de los elementos en el conjunto de datos de entrada, lo que puede ser un limitante en eficiencia para ciertos casos.
  La **complejidad temporal** del algoritmo para el **mejor caso es O(n)** cuando la lista ya está ordenada y para el **peor caso es O(n^2)** que sería por ejemplo cuando la lista está en orden inverso. (GeeksforGeeks, 2024)

- **Merge Sort:** Es **un algoritmo de ordenamiento basado en la técnica "Dividir y conquistar"**, funciona recursivamente, dividiendo el arreglo de entrada en dos mitades hasta llegar a sub-arreglos de tamaño 1.
  Se **usa la función mergeSort () para dividir el arreglo de forma recursiva** por la mitad hasta que cada sub-arreglos tenga un único elemento y ordenarla, **y la función merge** () **responsable de fusionar las dos mitades.** (GeeksforGeeks, geeksforgeeks.org, 2023)
  Este algoritmo es un método eficiente, general y comparativo para ordenar listas o arrays. (Hosting, 2024)
  La **complejidad temporal** del algoritmo para el **mejor caso, caso promedio y peor caso es la misma O (n log n).** (AlmaBetter, 2024)

- **Quick Sort:** Al igual que Merge Sort, Quick Sort también **es un algoritmo de ordenamiento basado en la técnica "Dividir y conquistar".** Este selecciona un elemento como pivote, coloca los elementos menores a su izquierda y los mayores a su derecha, y luego mediante la recursión realiza el mismo proceso a los sub-arreglos, escogiendo nuevamente pivotes y separando los elementos menores a su izquierda y mayores a su derecha. **El algoritmo utiliza la recursión y no acaba hasta tener todos los valores individualizados.** Este algoritmo es eficiente en grandes conjuntos de datos.

  La **complejidad temporal** del algoritmo para el **mejor caso es O (n log n)** que sería cuando el pivote elegido en cada paso divide el arreglo en mitades aproximadamente iguales, y para el **peor caso es O (n^2)** que sería cuando se escoge mal el pivote. (GeeksforGeeks, geeksforgeeks.org, 2024)

- **Función de biblioteca estándar:** En Python el método de ordenamiento **list.sort() ordena la lista en orden ascendente de forma predeterminada**. Este método junto a sorted () **utilizan variantes de Merge Sort o Quick Sort** dependiendo del tamaño de la lista y otros factores. En general, para casos donde el tamaño de la lista es muy grande se utiliza una variante de **Timsort**, que combina Merge Sort con Insertion Sort. (Python, 2024)

  La **complejidad temporal** del **mejor y peor caso es O (n log n)** para todas sus variaciones**.**

- **Algoritmo Iterativo Cúbico Tradicional:** Este algoritmo es la **implementación directa de la multiplicación de matrices**. Funciona mediante la **utilización de bucles anidados**. Calcula cada elemento de la matriz resultante sumando los productos de los elementos correspondientes de las filas y columnas. La **complejidad temporal** para el **mejor caso y peor caso es O (n^3).** (UChile, -)

- **Algoritmo Iterativo Cúbico Optimizado para mantener la localidad de los datos:** Este algoritmo es similar al algoritmo tradicional, pero con una optimización que consiste en que antes de la multiplicación, **se transpone la segunda matriz para mejorar la localidad de los datos en memoria**. La **complejidad temporal** para el **mejor caso y peor caso es O (n^3)** igual que en el método tradicional. (UChile, -)

- **Algoritmo de Strassen:** **Se basa en la técnica de "Dividir y conquistar"** y es muy útil para matrices grandes. Es un **método eficiente para la multiplicación de matrices** ya que reduce la cantidad de operaciones aritméticas requeridas, sin embargo, aumenta el número de sumas y restas. Divide las matrices en submatrices más pequeñas y **realiza multiplicaciones recursivas** para luego combinar los resultados. Solo se necesitan resolver 7 subproblemas. (GeeksforGeeks, geeksforgeeks.org, 2024)

  La **complejidad temporal es** O ($7^{\log_2 n}$), aproximadamente **O ($n^{2.81}$)**, donde **n es el tamaño de las matrices.** (UChile, -)

Descripción de Data Sets Algoritmos De Ordenamiento.

Los tipos de conjuntos de datos con los que se ha trabajado para algoritmos de ordenamientos han sido:

- Datos aleatorios
- Datos semiordenados
- Datos parcialmente ordenados

En primera instancia ejecutamos todos los algoritmos con los diferentes tipos de conjuntos de datos, siendo el conjunto de un tamaño de 50 datos, esto para ver que los algoritmos funcionaran adecuadamente. Para ello utilizamos el código ***dataset.py*** que se encuentra en la carpeta ***Codigos*** de nuestro ***repositorio***[1], para la generación de los archivos .txt que contienen los conjuntos de datos respectivamente.

En el código generador hemos definido para este caso en particular los siguientes limites:

- Datos aleatorios: Genera una lista de 50 números enteros aleatorios entre 1 y 1000.
- Datos semiordenados: Genera una lista de 50 números enteros ordenada entre 1 y n+1 con un porcentaje de desorden de datos del 10%.
- Datos parcialmente ordenados: Genera una lista de 50 números enteros ordenada entre 1 y n+1 con un porcentaje de desorden de datos del 50%.

Ejemplos de ejecución Algoritmos de ordenamiento:

Cantidad de datos: 50

1. Datos aleatorios:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
Datos aleatorios generados y guardados en 'aleatorios.txt'.
Datos semiordenados generados y guardados en 'semiordenados.txt'.
Matrices generadas y guardadas en 'matriz.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
[11, 25, 30, 37, 45, 53, 62, 88, 115, 121, 135, 167, 199, 214, 227, 253, 256, 263, 264, 294, 296, 311, 337, 342, 356, 417, 429, 448,
455, 458, 501, 502, 510, 548, 604, 611, 683, 716, 727, 770, 820, 847, 866, 878, 926, 937, 967, 969, 989, 998]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
[11, 25, 30, 37, 45, 53, 62, 88, 115, 121, 135, 167, 199, 214, 227, 253, 256, 263, 264, 294, 296, 311, 337, 342, 356, 417, 429, 448,
455, 458, 501, 502, 510, 548, 604, 611, 683, 716, 727, 770, 820, 847, 866, 878, 926, 937, 967, 969, 989, 998]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
[11, 25, 30, 37, 45, 53, 62, 88, 115, 121, 135, 167, 199, 214, 227, 253, 256, 263, 264, 294, 296, 311, 337, 342, 356, 417, 429, 448,
455, 458, 501, 502, 510, 548, 604, 611, 683, 716, 727, 770, 820, 847, 866, 878, 926, 937, 967, 969, 989, 998]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
[11, 25, 30, 37, 45, 53, 62, 88, 115, 121, 135, 167, 199, 214, 227, 253, 256, 263, 264, 294, 296, 311, 337, 342, 356, 417, 429, 448,
455, 458, 501, 502, 510, 548, 604, 611, 683, 716, 727, 770, 820, 847, 866, 878, 926, 937, 967, 969, 989, 998]
Tiempo de ejecución: 0.000000 segundos
```

1.Repositorio GitHub: https://github.com/Javiera9920/Tareas-INF221/tree/main/Tarea1-INF221-JavieraLayana-202073075K

2. Datos semiordenados, 10% de desorden de datos:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
Datos aleatorios generados y guardados en 'aleatorios.txt'.
Datos semiordenados generados y guardados en 'semiordenados.txt'.
Matrices generadas y guardadas en 'matriz.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
[1, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 30, 31, 32, 32, 32, 34, 35, 37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
[1, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 30, 31, 32, 32, 32, 34, 35, 37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
[1, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 30, 31, 32, 32, 32, 34, 35, 37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
[1, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 30, 31, 32, 32, 32, 34, 35, 37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
Tiempo de ejecución: 0.000000 segundos
```

3. Datos parcialmente ordenados, 50% de desorden de datos:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
Datos aleatorios generados y guardados en 'aleatorios.txt'.
Datos semiordenados generados y guardados en 'semiordenados.txt'.
Datos parcialmente ordenados generados y guardados en 'parciales.txt'.
Matrices generadas y guardadas en 'matriz.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
[1, 2, 3, 4, 4, 7, 8, 9, 10, 12, 12, 13, 13, 14, 16, 19, 20, 21, 22, 23, 23, 24, 25, 27, 28, 28,
29, 29, 30, 31, 32, 33, 33, 34, 34, 34, 35, 35, 37, 39, 39, 40, 40, 40, 41, 42, 43, 47, 49, 50]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
[1, 2, 3, 4, 4, 7, 8, 9, 10, 12, 12, 13, 13, 14, 16, 19, 20, 21, 22, 23, 23, 24, 25, 27, 28, 28,
29, 29, 30, 31, 32, 33, 33, 34, 34, 34, 35, 35, 37, 39, 39, 40, 40, 40, 41, 42, 43, 47, 49, 50]
Tiempo de ejecución: 0.001003 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
[1, 2, 3, 4, 4, 7, 8, 9, 10, 12, 12, 13, 13, 14, 16, 19, 20, 21, 22, 23, 23, 24, 25, 27, 28, 28,
29, 29, 30, 31, 32, 33, 33, 34, 34, 34, 35, 35, 37, 39, 39, 40, 40, 40, 41, 42, 43, 47, 49, 50]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
[1, 2, 3, 4, 4, 7, 8, 9, 10, 12, 12, 13, 13, 14, 16, 19, 20, 21, 22, 23, 23, 24, 25, 27, 28, 28,
29, 29, 30, 31, 32, 33, 33, 34, 34, 34, 35, 35, 37, 39, 39, 40, 40, 40, 41, 42, 43, 47, 49, 50]
Tiempo de ejecución: 0.000000 segundos
```

Resultados Experimentales Algoritmos de Ordenamiento.

Debido a la baja cantidad de datos del caso prueba, podemos notar que el tiempo de ejecución es 0, observamos que esto ocurre para conjuntos de datos muy muy pequeños.

Para poder analizar de mejor manera los distintos algoritmos, realizamos gráficas de diferencias en tiempo de ejecución para diferentes tipos de datos y tamaños de conjuntos de datos, teniendo en consideración lo mencionado con anterioridad a la hora de seleccionar el tamaño de los conjuntos para poder comparar. Estas gráficas pueden ser observadas también en la carpeta ***Gráficas*** de nuestro ***repositorio***[1].

En este punto hemos decidido separar el código generador inicial de conjuntos de datos para tratar de evitar confusiones a la hora de ejecutarlos. Para ello, hemos creado un código generador para cada tipo de datos, los cuales se mencionan a continuación y pueden ser encontrados en la carpeta ***Codigos*** de nuestro ***repositorio***[1]:

- Código generador de datos Aleatorios: **datasetA.py**
- Código generador de datos Semiordenados: **datasetSO.py**
- Código generador de datos Parcialmente ordenados: **dataset.py**

Además, en esta sección agregamos el respaldo de cada ejecución de los diferentes algoritmos para cada conjunto de datos según su tipo y tamaño.

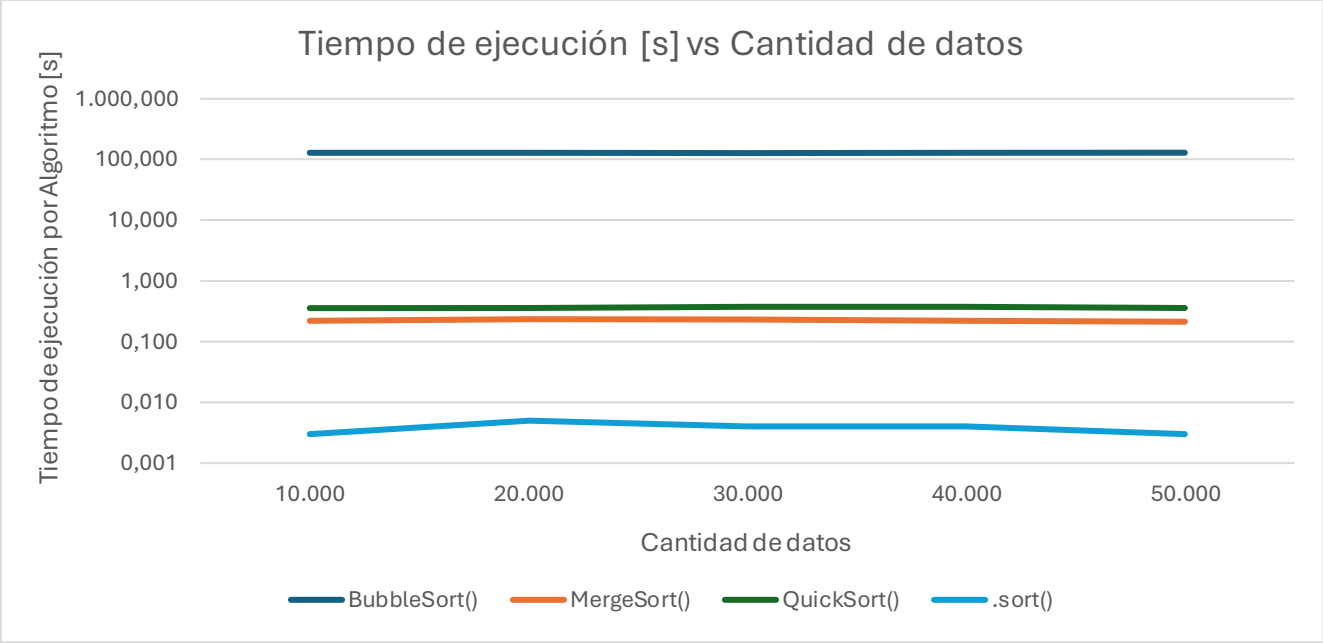
Ahora, podremos observar las diferencias entre el tiempo de ejecución de los diferentes algoritmos de ordenamiento para conjuntos de datos [s] de tamaño n (10.000, 20.000, 30.000, 40.000 y 50.000) de tipo aleatorio, semiordenado y parcialmente ordenado con los siguientes limites:

- Datos aleatorios: Genera una lista de n números enteros aleatorios entre 1 y 150.000.
- Datos semiordenados: Genera una lista de n números enteros ordenada entre 1 y n+1 con un porcentaje de desorden de datos del 10%.
- Datos parcialmente ordenados: Genera una lista de n números enteros ordenada entre 1 y n+1 con un porcentaje de desorden de datos del 50%.

También vamos a poder analizar los tiempos de ejecución y su relación con los tipos de datos y el tamaño de los conjuntos de datos.

1.Repositorio GitHub: https://github.com/Javiera9920/Tareas-INF221/tree/main/Tarea1-INF221-JavieraLayana-202073075K

Gráfica de datos Aleatorios:



	Tiempo de ejecución por Algoritmo [s]			
Cantidad de datos	BubbleSort()	MergeSort()	QuickSort()	.sort()
10.000	128,4	0,223	0,357	0,003
20.000	128,1	0,230	0,363	0,005
30.000	127,7	0,232	0,375	0,004
40.000	128,1	0,227	0,375	0,004
50.000	129,5	0,213	0,363	0,003

Tabla 1: Datos de tiempo de ejecución por algoritmo según cantidad de datos. Tipo Aleatorios

Análisis.

Podemos observar que el tiempo de ejecución para tipos de datos aleatorios es muy variante y poco estable, vemos que el tiempo es fluctuante y no sigue un patrón en relación con la cantidad de datos.

Además, el algoritmo de ordenamiento BubbleSort () tiene un tiempo de ejecución bastante alto, tanto para conjuntos de datos pequeños como para un conjunto de gran tamaño, siendo así una mala opción para implementar debido a su baja eficiencia.

Por otra parte, el algoritmo .sort() que es la función de sorting de la biblioteca de Python parece ser la mejor opción tanto para conjuntos de datos pequeños como para aquellos de gran tamaño, esto se puede deber a los algoritmos que funcionan por debajo en esta función, los cuales ya mencionamos con anterioridad

Respaldo de ejecución de datos Aleatorios:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetA.py
10000 Datos aleatorios generados y guardados en 'aleatorios.txt
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 128.392328 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.222930 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.357020 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.003013 segundos
```

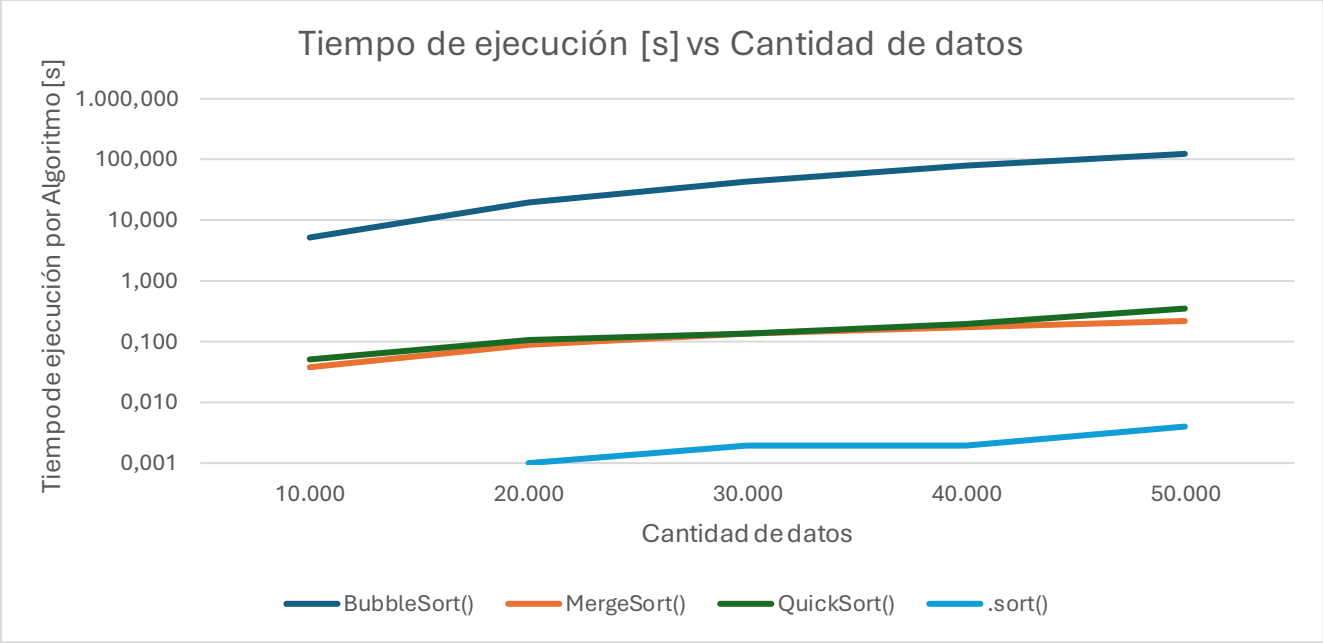
```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetA.py
20000 Datos aleatorios generados y guardados en 'aleatorios.txt'
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 128.077077 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.229591 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.363030 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.005010 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetA.py
30000 Datos aleatorios generados y guardados en 'aleatorios.txt'
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 127.697150 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.231844 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.375249 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.003991 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetA.py
40000 Datos aleatorios generados y guardados en 'aleatorios.txt'
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 128.059715 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.227299 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.375008 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.003991 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetA.py
50000 Datos aleatorios generados y guardados en 'aleatorios.txt'
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 129.474573 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.213012 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.363326 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.002983 segundos
```

Gráfica de datos semiordenados:



	Tiempo de ejecución por Algoritmo [s]			
Cantidad de datos	BubbleSort()	MergeSort()	QuickSort()	.sort()
10.000	5,193	0,038	0,051	0,000
20.000	20,07	0,089	0,104	0,001
30.000	43,13	0,134	0,135	0,002
40.000	78,09	0,170	0,194	0,002
50.000	123,9	0,219	0,351	0,004

Tabla 2: Datos de tiempo de ejecución por algoritmo según cantidad de datos. Tipo Semiordenados.

Análisis.

A diferencia de los conjuntos de datos de tipo aleatorio, podemos observar que aquí los tiempos de ejecución tienen una relación directamente proporcional con el tamaño del conjunto de datos, vemos que a medida que la cantidad de datos es mayor el tiempo de ejecución también lo es, en todos los algoritmos.

Sin embargo, al igual que en el análisis anterior, el algoritmo BubbleSort () sigue siendo el menos eficiente. Aun así, hay que destacar que el tiempo de ejecución para una menor cantidad de datos como 10.000 es mucho menor que en el punto anterior. Esto se puede deber a que al estar los datos semiordenados se facilita un poco la tarea del algoritmo.

La función de ordenamiento de la biblioteca de Python .sort() sigue demostrando ser la más eficiente para los distintos tamaños de conjuntos de datos.

Respaldo de ejecución de datos Semiordenados:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetSO.py
10000 Datos semiordenados generados y guardados en 'semiordenados.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 5.193446 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.038002 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.051003 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.000000 segundos
```

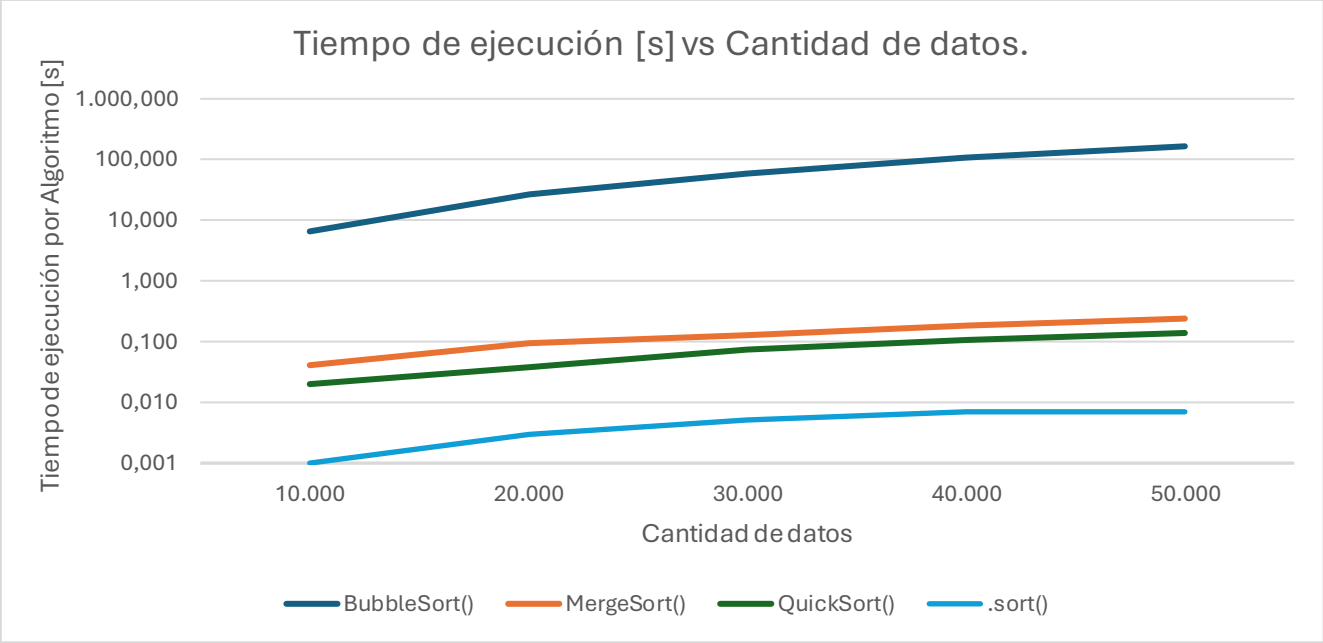
```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetSO.py
20000 Datos semiordenados generados y guardados en 'semiordenados.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 20.066150 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.088997 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.103966 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.001000 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetSO.py
30000 Datos semiordenados generados y guardados en 'semiordenados.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 43.132502 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.134030 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.134565 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.002000 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetSO.py
40000 Datos semiordenados generados y guardados en 'semiordenados.txt
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 78.085515 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.170000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.194031 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.002000 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python datasetSO.py
50000 Datos semiordenados generados y guardados en 'semiordenados.txt
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 123.921352 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.218998 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.351110 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.003994 segundos
```

Gráfica de datos parcialmente ordenado:



	Tiempo de ejecución por Algoritmo [s]			
Cantidad de datos	BubbleSort()	MergeSort()	QuickSort()	.sort()
10.000	6,551	0,041	0,020	0,001
20.000	26,13	0,096	0,038	0,003
30.000	59,75	0,130	0,074	0,005
40.000	105,7	0,183	0,106	0,007
50.000	165,1	0,240	0,139	0,007

Tabla 3: Datos de tiempo de ejecución por algoritmo según cantidad de datos. Tipo parcialmente ordenados.

Análisis.

Al igual que con los datos semiordenados, podemos observar que existe una relación directamente proporcional entre el tiempo de ejecución por algoritmos y la cantidad de datos, cuando la cantidad de datos aumenta el tiempo de ejecución también. Además, al igual que con datos aleatorios y datos semiordenados, sigue siendo BubbleSort () el algoritmo menos eficiente y la función .sort() de Python la más eficiente para todo tamaño de datos y ya sabemos también que para todo tipo de datos.
A diferencia de los conjuntos de datos semiordenados, este conjunto de datos parcialmente ordenado presenta un mayor desorden de datos, por lo que al algoritmo BubbleSort () se le complica un poco más la resolución.

Entre los algoritmos MergeSort () y QuickSort () vemos que no existe mucha diferencia en datos pequeños, cuando ya llegamos al conjunto más grande podemos observar una leve ventaja de QuickSort () por sobre MergeSort () en eficiencia, pero solo se da en este caso y no en los tipos de datos anteriores.

Respaldo de ejecución de datos:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
10000 Datos parcialmente ordenados generados y guardados en 'parciales.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 6.550645 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.040997 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.020003 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.001001 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
20000 Datos parcialmente ordenados generados y guardados en 'parciales.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 26.127071 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.096002 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.038000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.002998 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
30000 Datos parcialmente ordenados generados y guardados en 'parciales.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 59.751534 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.129992 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.074028 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.004988 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
40000 Datos parcialmente ordenados generados y guardados en 'parciales.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 105.719952 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.183033 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.106021 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.007033 segundos
```

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python dataset.py
50000 Datos parcialmente ordenados generados y guardados en 'parciales.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python bubblesort.py
Tiempo de ejecución: 165.057706 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python mergesort.py
Tiempo de ejecución: 0.239992 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python quicksort.py
Tiempo de ejecución: 0.138991 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python sort.py
Tiempo de ejecución: 0.007000 segundos
```

Descripción de Data Sets Algoritmos de Multiplicación de Matrices.

Para este caso de algoritmos de multiplicación de matrices los tipos de conjuntos de datos con los que se ha trabajado han sido:

- Matrices cuadráticas (misma cantidad de filas y columnas).
  - Igual dimensión.
  - Distinta dimensión.
- Matrices no cuadráticas.
  - igual dimensión.
  - Distinta dimensión.

En primera instancia ejecutamos los distintos algoritmos con un mismo tipo de dato y de igual tamaño, esto para comprobar que funcionen correctamente. Para ello, tomaremos el primer caso para realizar un caso prueba:

- Matrices cuadráticas de igual dimensión aleatorias de un tamaño pequeño.
- Matrices cuadráticas de distinta dimensión aleatorias de un tamaño pequeño.
- Matrices no cuadráticas aleatorias de igual dimensión.

Hemos creado un archivo llamado matrices.py que funciona como generador de matrices de números aleatorios, este código nos entrega dos matrices en diferentes archivos:

- MatrizA.txt
- MatrizB.txt

En el ***código generador de matrices***, que se puede encontrar en la carpeta ***Codigos*** de nuestro ***repositorio***[1] con el nombre ***matrices.py***, hemos definido los siguientes limites:

- Generar matriz aleatoria con números enteros aleatorios en un rango de 0 a 100.
- Generar matriz ordenada con números consecutivos, empezando desde un valor inicial que depende del índice de la fila.

Debido a que se trata de algoritmos de multiplicación de matriz y no de algoritmos de ordenamientos es que hemos considerado que el tener matrices aleatorias u ordenadas no afecta en gran medida al tiempo de ejecución, pues al tratarse de multiplicaciones el orden no es sinónimo de eficiencia. Es por ello, que en este informe solo tendremos en consideración matrices aleatorias, sin embargo, en el código quedará expresada la opción de matriz ordenada por si se desea realizar casos de prueba a futuro.

1.Repositorio GitHub: https://github.com/Javiera9920/Tareas-INF221/tree/main/Tarea1-INF221-JavieraLayana-202073075K

Ejemplos de ejecución Algoritmos de Multiplicación de Matrices:

1. Cuadrática aleatoria de igual dimensión tamaño 2 x 2:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 2 x 2 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 2 x 2 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Resultado de la multiplicación de A y B:
 [[4865 2240]
 [3056  497]]
Tiempo de ejecución: 0.000998 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Resultado de la multiplicación de A y B:
 [[4865 2240]
 [3056  497]]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Matriz C (Resultado de A * B):
 [[4865 2240]
 [3056  497]]
Tiempo de ejecución: 0.000000 segundos
```

2. Cuadrática aleatoria de distinta dimensión: A (2x2) y B (4x4)

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 2 x 2 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 4 x 4 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Las matrices no se pueden multiplicar
Resultado de la multiplicación de A y B:
 None
Tiempo de ejecución: 0.000292 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Las matrices no se pueden multiplicar
Resultado de la multiplicación de A y B:
 None
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
```

3. No cuadrática aleatoria de igual dimensión:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 3 x 3 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 3 x 3 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Resultado de la multiplicación de A y B:
 [[3580 3749 2186]
 [5841 6291 4919]
 [7289 9178 5097]]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Resultado de la multiplicación de A y B:
 [[3580 3749 2186]
 [5841 6291 4919]
 [7289 9178 5097]]
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
```

Resultados Experimentales Algoritmos de Multiplicación de Matrices.

Al igual que en el caso de algoritmos de ordenamiento, observamos que debido al tamaño pequeño de las matrices el tiempo de ejecución es en su gran mayoría 0 [s].

Para poder realizar un análisis más a detalle del tiempo de ejecución de cada algoritmo de acuerdo con el tipo y tamaño de los datos realizamos gráficas de esto, que se pueden encontrar en la carpeta ***Gráficas*** de nuestro ***repositorio***[1], teniendo nuevamente en consideración el tratar de evitar trabajar con tamaño de matrices muy pequeñas, para así poder realizar mejores comparaciones entre los algoritmos.

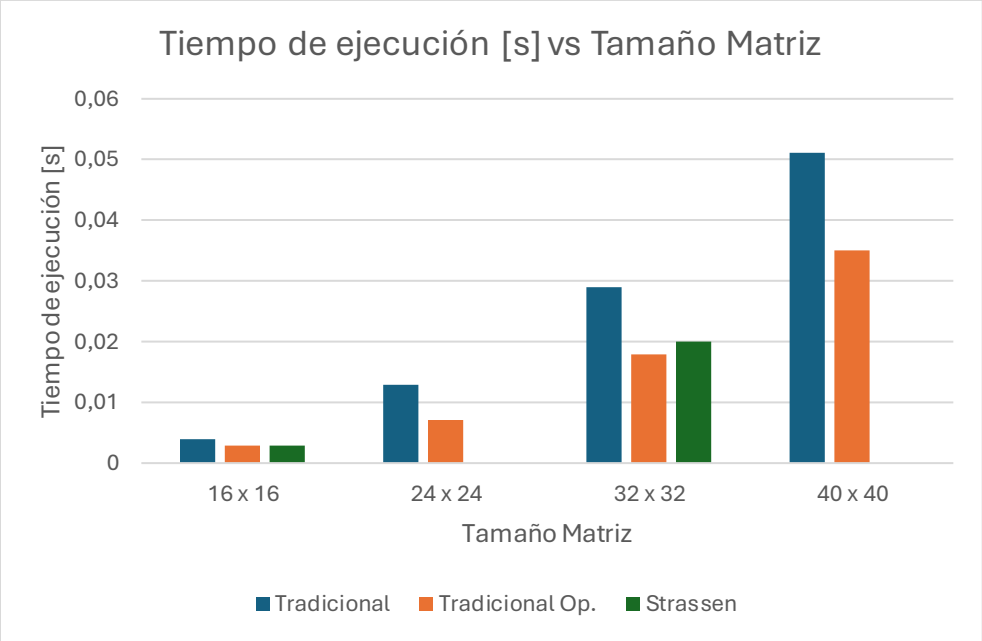
A diferencia del caso de algoritmos de ordenamiento, aquí solo utilizamos un código generador de matrices, que nos entregará nuestros dos archivos .txt con cada matriz individualizada. Las demás especificaciones del código ya se entregaron en su respectiva sección de Data Sets.

En esta sección agregaremos también el respaldo de las ejecuciones realizadas para cada tipo de conjunto de datos, su tamaño y los algoritmos de multiplicación de matrices. Hay que recordar que para efectos de este informe solo se tendrá en cuenta matrices de tipo aleatorio y no ordenadas.

Con estos resultados transformados en gráficas realizamos análisis más a detalle de acuerdo con cada ejecución con relación al tiempo de ejecución según los tipos de datos y su tamaño.

1.Repositorio GitHub: https://github.com/Javiera9920/Tareas-INF221/tree/main/Tarea1-INF221-JavieraLayana-202073075K

Gráfica de Matriz Cuadrática de Igual Dimensión:



	Tiempo de ejecución [s]		
Tamaño Matriz	Tradicional	Tradicional Op.	Strassen
16 x 16	0,004	0,003	0,003
24 x 24	0,013	0,007	
32 x 32	0,029	0,018	0,020
40 x 40	0,051	0,035	

Tabla 4: Tiempo de ejecución para cada algoritmo de multiplicación de acuerdo con el tamaño de la Matriz. Tipo Cuadrática I.D.

Análisis.

Podemos observar que existe una relación directamente proporcional entre el tamaño de la matriz y el tiempo de ejecución para cada algoritmo. Además, vemos que para que el algoritmo de Strassen presente datos debemos cumplir con las siguientes condiciones:

- La matriz es cuadrática, Potencia de 2 y de Igual dimensión.

En este caso solo se cumple en la primera ejecución y en la tercera.

También se observa que el algoritmo Tradicional Optimizado parece ser el mejor para todos los casos, ya que permite la multiplicación de matrices que no sean potencia de 2 y es eficaz en tiempo de ejecución para tamaños de matrices pequeños y grandes, con gran ventaja por sobre el algoritmo tradicional.

Respaldos de ejecución de Algoritmos:

- Matriz A = 16 x 16 y Matriz B = 16 x 16

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 16 x 16 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 16 x 16 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.004000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.002515 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Tiempo de ejecución: 0.002556 segundos
```

- Matriz A = 24 x 24 y Matriz B = 24 x 24

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 24 x 24 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 24 x 24 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.012995 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.007423 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
```

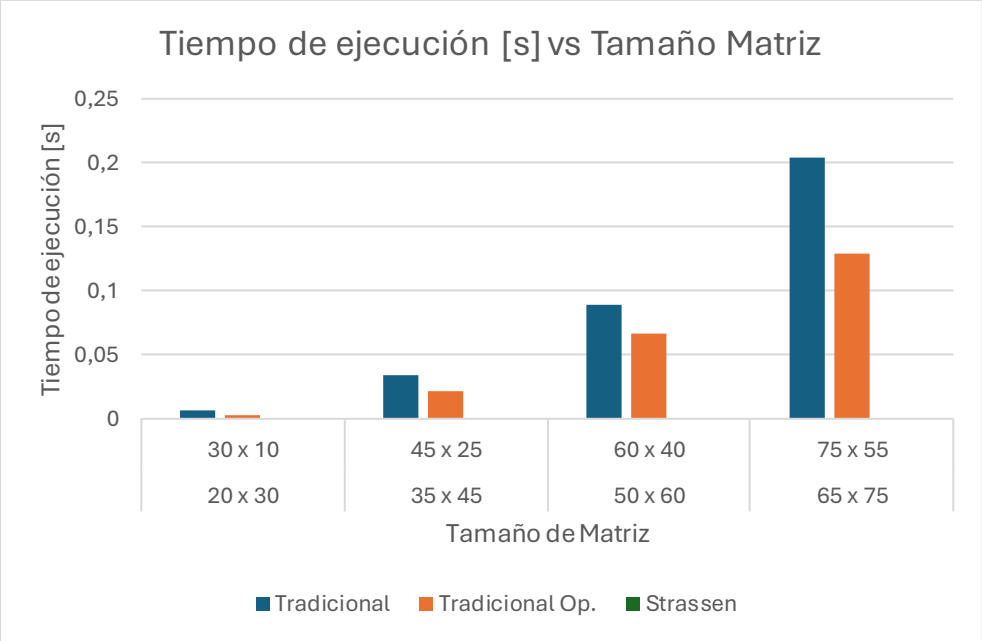
- Matriz A = 32 x 32 y Matriz B = 32 x 32

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 32 x 32 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 32 x 32 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.028977 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.017996 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Tiempo de ejecución: 0.019989 segundos
```

- Matriz A = 40 x 40 y Matriz B = 40 x 40

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 40 x 40 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 40 x 40 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.050712 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.035065 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
```

Gráfica de Matriz No Cuadrática de Distinta Dimensión:



		Tiempo de ejecución [s]		
Tamaño Matriz A	**Tamaño Matriz B**	Tradicional	Tradicional Op.	Strassen
20 x 30	30 x 10	0,006	0,003	
35 x 45	45 x 25	0,034	0,022	
50 x 60	60 x 40	0,089	0,067	
65 x 75	75 x 55	0,204	0,129	

Tabla 5: Tiempo de ejecución para cada algoritmo de multiplicación de acuerdo con el tamaño de la Matriz. Tipo No Cuadrática D.D.

Análisis.

En este caso observamos que al igual que en el caso de Matrices cuadradas de igual dimensión, el tiempo de ejecución presenta una relación directamente proporcional con el tamaño de las matrices.

Sin embargo, a diferencia del caso mencionado, en esta ocasión el algoritmo de Strassen nos entrega error en todas las ejecuciones, esto se debe a que, al ser matrices de distintas dimensiones, dejamos de cumplir con las condiciones del algoritmo de Strassen.

Vemos también que se cumple la condición de multiplicación de matrices de cantidad de columnas de la Matriz A es igual a la cantidad de filas de la Matriz B.

Agregar también que al igual que el primer caso, el algoritmo Tradicional Optimizado parece ser la mejor opción, funciona correctamente con más tipos de datos y es eficiente en tiempo para los distintos tamaños de matrices.

Respaldos de ejecución de Algoritmos:

- Matriz A = 20 x 30 y Matriz B = 30 x 10

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 20 x 30 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 30 x 10 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.005998 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.003008 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
```

- Matriz A = 35 x 45 y Matriz B = 45 x 25

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 35 x 45 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 45 x 25 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.033992 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.021995 segundos
```

- Matriz A = 50 x 60 y Matriz B = 60 x 40

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 50 x 60 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 60 x 40 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.088999 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.067000 segundos
```

- Matriz A = 65 x 75 y Matriz B = 75 x 55

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 65 x 75 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 75 x 55 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Tiempo de ejecución: 0.204463 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Tiempo de ejecución: 0.129374 segundos
```

Matriz Cuadrática de Distinta Dimensión y No cuadráticas de Igual Dimensión.

Estas matrices no pueden ser resueltas ya que no cumplen con las condiciones de multiplicación de matrices:

- Para que dos matrices puedan ser multiplicadas, la cantidad de columnas de la Matriz A deben ser igual a la cantidad de filas de la Matriz B.

Matrices Cuadráticas de Distinta Dimensión:

En este caso cumplimos la condición de que sea una matriz cuadrática ya que la Matriz A tiene igual cantidad de filas y columnas (16) y lo mismo con la Matriz B que tiene 32 filas y columnas.

Sin embargo, no cumplimos la condición mencionada más arriba, pues la cantidad de columnas de la Matriz A son 16 y la cantidad de filas de la matriz B son 32.

Matrices No Cuadráticas de Igual Dimensión:

En este caso cumplimos con la condición de que no sea una matriz cuadrática, pues observamos que la cantidad de filas y columnas son distintas, teniendo 9 Filas y 18 columnas cada matriz. Además, cumplimos con la condición de que sean de igual dimensión, pues ambas matrices tanto la Matriz A como la Matriz B tienen un tamaño de 9 x 18.

Sin embargo, igual que en el caso anterior no cumplimos con la condición ya mencionada: La cantidad de columnas de la Matriz A (18) es distinta a la cantidad de filas de la Matriz B (9).

Respaldos de ejecución de Algoritmos Matriz Cuadrática de Distinta Dimensión:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 16 x 16 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 32 x 32 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Las matrices no se pueden multiplicar
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Las matrices no se pueden multiplicar
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1>
```

Respaldos de ejecución de Algoritmos Matriz No Cuadrática de Igual Dimensión:

```
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python matrices.py
Matriz de tamaño 9 x 18 generada y guardada en 'matrizA.txt'.
Matriz de tamaño 9 x 18 generada y guardada en 'matrizB.txt'.
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicional.py
Las matrices no se pueden multiplicar
Tiempo de ejecución: 0.001005 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python tradicionalOpt.py
Las matrices no se pueden multiplicar
Tiempo de ejecución: 0.000000 segundos
PS C:\Users\JJare\Downloads\ALGOCO\Tarea1> python strassen.py
Error: Las matrices no son cuadradas o no tienen las mismas dimensiones.
```

Conclusiones.

Los resultados obtenidos en este informe confirman que la elección del algoritmo de ordenamiento y multiplicación de matrices adecuado depende tanto del tamaño como de las características de los datos.

En el caso de algoritmos de ordenamiento, Quick Sort y Merge Sort mostraron una alta eficiencia en la mayoría de los casos, al igual que la función .sort() que fue la mejor en los casos probados, mientras que Bubble Sort presentó un rendimiento significativamente inferior tanto en casos con conjuntos de datos pequeños como conjuntos de datos de gran tamaño.

Sin embargo, a pesar de que los algoritmos Quick Sort y Merge Sort demostraron rendimientos similares, Quick Sort es un algoritmo in-place, es decir, que ordena los datos sin utilizar espacio adicional, mientras que, Merge Sort es un algoritmo no-in-place. Las ventajas que puede tener Quick Sort al ser in-place es ser más eficiente en términos de uso de memoria, pero esto puede implicar mayor cantidad de operaciones sobre los datos, mientras que Merge Sort al ser no-in-place tiene un acceso más ordenado a la memoria.

El análisis realizado predijo correctamente cómo los algoritmos se comportarían bajo distintos escenarios de mejor y peor caso. Esto es importante porque ayuda a estimar con precisión cuánto tiempo tardará un algoritmo en ejecutarse en condiciones diferentes.

En el caso de la multiplicación de matrices, el algoritmo iterativo optimizado demostró ser más eficaz que el tradicional, especialmente en matrices de gran tamaño. Esto se debe a que el algoritmo iterativo optimizado preserva la localidad de los datos, accediendo a los datos de manera contigua en la memoria, aprovechando mejor la caché de la CPU. Así, reduce los tiempos de accesos a la memoria, lo que resulta en un menor tiempo de ejecución, incluso si la complejidad asintótica no cambia.

En relación con el algoritmo de Strassen pudimos observar que es bastante óptimo en eficacia, sin embargo, es muy limitado.

Finalmente, la experimentación refuerza la utilidad de las herramientas de profiling para identificar cuellos de botella y optimizar el rendimiento en situaciones prácticas, por ejemplo, en el caso de algoritmos de ordenamiento, el profiling puede mostrar que un algoritmo que debería ser eficiente teóricamente (como Quick Sort) está siendo menos eficiente debido a un mal manejo de la memoria o mala selección de pivotes. Esto sugiere posibles mejoras a la implementación. También nos permite evaluar el impacto de los datos, observando cómo distintos tamaños y tipos de datos afectan al rendimiento de los algoritmos. Por ejemplo, el análisis de tiempos de ejecución entre datos aleatorios y semiordenados mostró diferencias claras en la eficiencia de los algoritmos. (OpenIA, 2024)

Referencias

1. Repositorio GitHub: https://github.com/Javiera9920/Tareas-INF221/tree/main/Tarea1-INF221-JavieraLayana-202073075K

AlmaBetter. (12 de junio de 2024). *almabetter.com*. Recuperado el 6 de septiembre de 2024, de Complejidad de Tiempo Merge Sort(): https://www-almabetter-com.translate.goog/bytes/articles/merge-sort-time-complexity?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq&_x_tr_hist=true

GeeksforGeeks. (28 de Agosto de 2023). *geeksforgeeks.org*. Recuperado el 6 de Septiembre de 2024, de Program for Merge Sort: https://www.geeksforgeeks.org/python-program-for-merge-sort/

GeeksforGeeks. (06 de Agosto de 2024). *geeksforgeeks.org*. Recuperado el 6 de Septiembre de 2024, de Bubble Sort Algorithm: https://www.geeksforgeeks.org/bubble-sort-algorithm/

GeeksforGeeks. (1 de Septiembre de 2024). *geeksforgeeks.org*. Recuperado el 6 de septiembre de 2024, de Quick Sort: https://www.geeksforgeeks.org/quick-sort-algorithm/

GeeksforGeeks. (01 de junio de 2024). *geeksforgeeks.org*. Recuperado el 6 de septiembre de 2024, de Strassen algorithm in Python: https://www.geeksforgeeks.org/strassen-algorithm-in-python/

Hosting, S. (01 de mayo de 2024). *swhosting.com*. Recuperado el 6 de septiembre de 2024, de Introducción al algoritmo de ordenación Merge Sort: https://www.swhosting.com/es/blog/introduccion-al-algoritmo-de-ordenacion-merge-sort

OpenIA. (2024). *ChatGPT-4*. Recuperado el 8 de septiembre de 2024, de Herramientas de Profiling y Análisis Asintótico: https://chat.openai.com/

Python. (7 de septiembre de 2024). *python.org*. Recuperado el 8 de septiembre de 2024, de Sorting Techniques: https://docs.python.org/es/3/howto/sorting.html

UChile. (- de - de -). *dcc.uchile.cl*. Recuperado el 8 de septiembre de 2024, de Introducción al Análisis de Algoritmos: https://users.dcc.uchile.cl/~nbaloian/cc3001-02/Auxiliares/auxiliar2.pdf