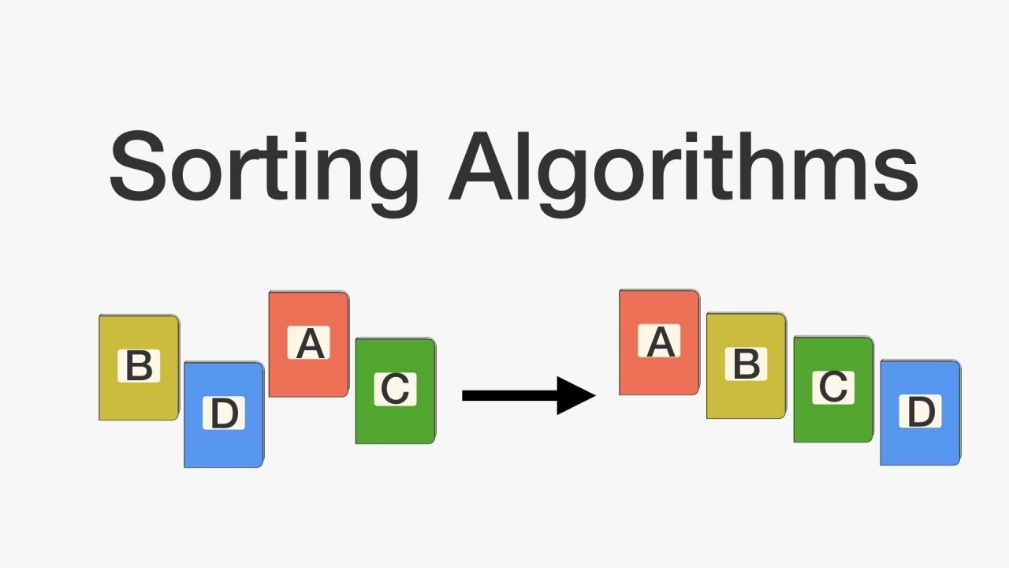
**INFORME 1: ALGORITMOS DE ORDENAMIENTO Y DE MULTIPLICACIÓN DE MATRICES**



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](#_Toc176835604)

[Descripción de Algoritmos. 4](#_Toc176835605)

[Descripción de Data Sets Algoritmos De Ordenamiento. 6](#_Toc176835606)

[Resultados Experimentales Algoritmos de Ordenamiento. 8](#_Toc176835607)

[Descripción de Data Sets Algoritmos de Multiplicación de Matrices. 15](#_Toc176835608)

[Resultados Experimentales Algoritmos de Multiplicación de Matrices. 17](#_Toc176835609)

[Conclusiones. 23](#_Toc176835610)

# Introducción.

En este informe se analizan distintos tipos de algoritmos, se entrega una breve descripción de estos y su implementación. Mediante un análisis experimental

# Descripción de Algoritmos.

* **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(n2)** que sería por ejemplo cuando la lista está en orden inverso.

* **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.**

Este algoritmo es un método eficiente, general y comparativo para ordenar listas o arrays.

La **complejidad temporal** del algoritmo para el **mejor caso, caso promedio y peor caso es la misma O (n log n)**

* **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 (n2)** que sería cuando se escoge mal el pivote.
* **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.

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 (n3).**
* **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 (n3)** igual que en el método tradicional.
* **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.  
  La **complejidad temporal** **es** O (7 log 2 n), aproximadamente **O (n2.81)**, donde **n es el tamaño de las matrices.**

# 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 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:

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

1. Texto

   Descripción generada automáticamenteDatos parcialmente ordenados, 50% de desorden de datos:

# 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.

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:

* 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.

**Gráfica de datos Aleatorios:**



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:**  
  
Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Gráfica de datos semiordenados:**



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:**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Gráfica de datos parcialmente ordenado:**



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:**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

# 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 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.

**Ejemplos de ejecución Algoritmos de Multiplicación de Matrices:**

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

**Texto

Descripción generada automáticamente**

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

Texto

Descripción generada automáticamente

1. No cuadrática aleatoria de igual dimensión:   
   Texto

   Descripción generada automáticamente

# 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 mas a detalle del tiempo de ejecución de cada algoritmo de acuerdo con el tipo y tamaño de los datos realizamos gráficas de esto, 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.

**Gráfica de Matriz Cuadrática de Igual Dimensión:**



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

Texto

Descripción generada automáticamente

* Matriz A = 24 x 24 y Matriz B = 24 x 24

Texto

Descripción generada automáticamente

* Matriz A = 32 x 32 y Matriz B = 32 x 32

Texto

Descripción generada automáticamente

* Matriz A = 40 x 40 y Matriz B = 40 x 40

Texto

Descripción generada automáticamente

**Gráfica de Matriz No Cuadrática de Distinta Dimensión:**



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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

**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:**

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

# Conclusiones.