Algoritmos Dividir y Conquistar

Daniel Quispe
Rol: 202273529-5 Paralelo: 200

7 de septiembre de 2024

1. Introducción

En este trabajo, veremos distintas implementaciones de algoritmos de ordenamiento, que tal como su nombre indica, a través de una serie de elementos (en el caso de este trabajo, numericos) estos puedan entregar los arreglos ordenados, y multiplicación de matrices, que lo que busca es ver que tan eficientes son los algortimos que permiten este tipo de operaciones. Y de esta manera, mostrar como el tiempo que los programas tardan en ejecutar pueden reducirse dependiendo de como nosotros programemos la resolución de dichos problemas.

Siguiendo el lema de "Dividir y conquistar" se han puesto a prueba los algoritmos de bubblesort; el cual ordena recorriendo todos los datos comparando el primero con el que le sigue, mergesort; el cual subdivide el arreglo de datos en dos recorriendolo recursivamente, quicksort; este algoritmo agarra un pivote con el cual va ordenando los datos a su derecha o izquierda según el valor del pivote y std::sort(); el cual es el algoritmo entregado por la libreria estandar, para los casos de ordenamientos. Mientras que para las multiplicaciones se han puesto a prueba el algoritmo iterativo cubicos tradicional; el cual multiplica las matrices fila por columna, iterativo cubicos optimizado; este a traves de un reordenamientos de datos multiplica las matrices filas con filas, y Strassen; el cual a traves de una serie de operaciones logra reducir en una llama recursiva la multiplicación.

Para poder visualizar el tiempo que estos algoritmos tardan en ser ejecutados, se someteran a distintos datasets generados a través de codigo con un cierto formato especifico para pasarlas como input y de esta manera comprobar el tiempo que estas tardan. Dado que mientras más eficiente sea la forma de resolver los problemas, menor será el tiempo, se espera que aquellas funciones con recursividad logren mejorar el tiempo, pero dado que esto también depende de la cantidad de datos que se tengan, se verá que no siempre se logra ver con los casos de prueba utilizados.

2. Algoritmos

Para probar la eficiencia de algoritmos de ordenamiento y multiplicación de matrices se han usado los siguientes algoritmos, cada uno al final de cada explicación se encuentra el enlace directo a cada uno de los codigos con los cuales se probaron los distintos datasets utilizados. Además, las fuentes de donde estos codigos fueron obtenidos se dispondrá en la ultima sección de este informe.

Así tambien, algunos algoritmos utilizan las mismas funciones de la libreria estandar, por lo que para no reiterar tanta información, se documentara en esta sección.

- swap(): Esta funcion intercambia 2 elementos que se le pasa por parametro, la complejidad de esta función es de $O(1)$.
- vector: Esta es una estructura que tiene distintas funciones, las usadas son:
  - size(): Tiene una complejidad de $O(1)$ y no usa espacio auxiliar.
  - clear(): Esta función limpia todo el vector, tiene espacio auxiliar constante, es decir $O(1)$ y su complejidad es de $O(n)$.

Adicionalmente, en el caso de los codigos de multiplicación de matrices, se creo un codigo aparte que contiene distintas funciones que no contribuyen directamente con el algoritmo, dejandolas apartadas para no aglomerar funciones y tenerlas más modularizadas. El codigo se encuentra aqui

2.1. Algoritmos de ordenamiento

2.1.1. BubbleSort

El bubblesort es un algoritmo que ordenamiento el cual funciona revisando los dos primeros elementos y comparandolos entre si, si el segundo elemento es menor que el primero, estos se intercambian. Este proceso se repite constantemente hasta revisar todos los datos, y luego se repite pero desde la segunda posición y así sucesivamente.

Dado que este algoritmo no utiliza espacio auxiliar, este es constante, es decir $O(1)$, y la función de la libreria estandar que utiliza es la función 'swap' que se menciono anteriormente. Dada la implementación del bubblesort, en el mejor de los casos, este tiene una complejidad de $O(n^2)$, y para el peor de los casos este igual es $O(n^2)$, esto se debe a que al ordenar los datos, obligatoriamente se debe revisar todos los datos dos veces.

El algoritmo de Bubblesort utilizado se encuentra en este enlace.

2.1.2. MergeSort

Siguiendo el pensamiento de "dividir y conquitar", el algoritmo de mergersort separa el arreglo en dos, esto de manera recursiva hasta que solo quede un elemento en el arreglo, a partir de acá se devuelve y comienza a ordenar los elementos una vez se revisa tanto el lado izquierdo como el derecho.

Tal como se menciono, el algoritmo comienza a ordenar cuando el lado izquierdo y derecho son devueltos por la recursión, para ello, se crean dos arreglos auxiliares, uno para el lado izquierdo y el otro para el lado derecho, de esta manera se pueden comparar los elementos y ordenarlos. Dicho proceso de auxiliar, tiene un coste de $O(n)$ dado que tiene que pasar los "n"datos a los arreglos auxiliares. Además, dada la recursión que este algoritmo genera en el mejor de los casos este es $O(n \log n)$, pero tambien en el peor de los casos este es $O(n \log n)$.

El algoritmo de Mergesort utilizado se encuentra en este enlace.

2.1.3. QuickSort

Similar al Mergesort, el algoritmo quicksort divide el arreglo en dos, pero a diferencia del algoritmo mencionado anteriormente, este los divide según un pivote, este pivote puede ser un elemento arbitrario del arreglo a ordenar o un elemento predeterminado, esto según la implementación utilizada en el codigo. De esta manera, particiona el arreglo en dos y los comienza a ordenar en base al pivote, dejando los elementos menores a este a la izquierda, y los mayores a la derecha, esto de manera recursiva hasta llegar a tener un solo elemento en el arreglo, cuando ambos lados estan ordenados, este comienza a devolverse y de esta forma obteniendo el arreglo ordenado.

A diferencia del algoritmo mergesort, el espacio auxiliar usado por este algoritmo es de $O(\log n)$ y en el peor de los casos es de $O(n)$. Además la unica función que ocupa de la libreria estandar es 'swap'.

Dado un arreglo de elementos completamente desordenados, este algoritmo tiene una complejidad de $O(n \log n)$, dado que para esta función este es su mejor caso, mientras que su peor caso viene dado cuando el pivote tiene el elemento más pequeño o más grande del arreglo, en dicho caso, dado que particionaria el arreglo de manera que quedara solo un elemento fuera del ordenamiento, su complejidad será de $O(n^2)$.

El algoritmo de Quicksort utilizado se encuentra en este enlace. Tal como se dijo anteriormente, el pivote puede ser escogido dependiendo de la programación del algoritmo, en este caso, el pivote siempre será el ultimo elemento del arreglo.

2.1.4. std::sort() de la libreria estandar

La función "std::sort()"de la libreria estandar 3 funciones sort por detras (Quicksort, Heapsort e Insertion Sort), siendo esta una función de ordenamiento híbrido. Es el algoritmo más rápido dado que usa varias funciones de ordenamiento a la vez. Esta función recibe 2 parametros obligatorios, el primero entrega desde donde se quiere comenzar a ordenar, y el segundo donde termina de ordenar. Adicionalmente, existe un tercer parametro para ordenar de manera descendente.

Esta función tiene $O(1)$ de espacio auxiliar, dado que no tiene. Ademas, la complejidad en el mejor de los casos es de $O(n \log n)$ y en el peor de los también es de $O(n \log n)$. La implementación de esta función se puede apreciar en este programa.

2.2. Algoritmos de Multiplicación de Matrices

2.2.1. Algoritmo iterativo cúbico tradicional

Este algoritmo se basa en la multiplicación de matrices basica que se enseña, donde se suma la multiplicación de los elementos de las filas de la primera matriz con los elementos de las columnas de la segunda matriz. A continuación se muestra un ejemplo de una multiplicación de matrices de 2x2 con otra de 2x2.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix} \tag{1}$$

De esta manera, para resolver este problema, se recorren las filas de la primera matriz multiplicandola con la columna de la segunda matriz a traves de tres ciclos for anidados. Dado que para resolver la multiplicación, obligatoriamente tiene que recorrer la matriz completa, sin importar el caso, este tendrá complejidad $O(n^3)$.

Su implementación usada en esta evaluación esta a continuación. Iterativo cuadratico.

2.2.2. Algoritmo iterativo cúbico optimizado

Este algoritmo sigue la misma logica que el algoritmo iterativo cúbico tradicional, pero a diferencia de este, se transpone la segunda matriz usada para que al momento de obtener los datos, el programa tenga más cerca la información a utilizar para la multiplicación. Como se puede apreciar a continuación:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} e & f \\ g & h \end{bmatrix}^{\top} = \begin{bmatrix} e & g \\ f & h \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix} \tag{4}$$

Como se aprecia, la multiplicación de matrices se efectua multiplicando filas por filas, esto hará más eficiente la busqueda de datos.

Al igual que el algoritmo anterior, la complejidad de este algoritmo sigue siendo de $O(n^3)$ para cualquier caso, pero este algoritmo es más rapido que el mencionado anteriormente dado que los datos que se necesitan al recorren la matriz estan más cerca, haciendo que para el sistema sea más facil su acceso y busqueda, ahorrando tiempo.

Su implementación usada en esta evaluación esta a continuación. Iterativo cuadratico optimizado.

2.2.3. algoritmo de Strassen

Este algoritmo consiste en multiplicar las matrices con las siguientes formulas:

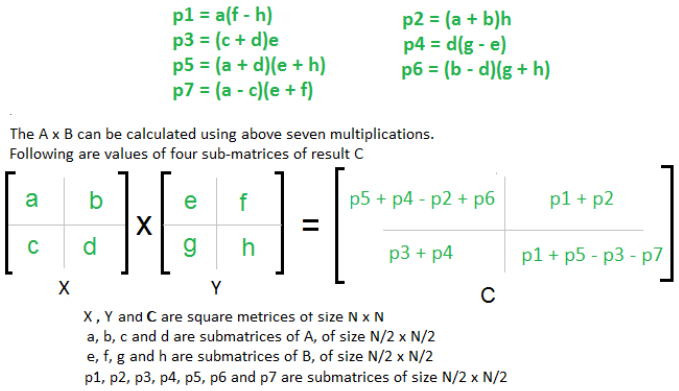

Figura 1: Algoritmo de Strassen

De esta manera, la cantidad de datos a evaluar se reduce de ocho a siete, logrando una reducción de tiempo. Para ello, a la hora de programar la función esta reducira de ocho a siete llamadas recursivas, y de esta manera logrando una reducción de tiempo de $O(n^3)$ a $O(n^{\log_2 7})$. Pero dado que dada la función que esta genera, la reducción de tiempo se verá reflejado con tamaños muy grandes de datos procesados, por lo que para esta experiencia, en los casos de prueba no se podrá visualizar la mejora de rendimiento que esta tiene. El codigo que se implemento para Strassen contiene las funciones de vectores "size()" y "clear()", sus complejidades se mencionan anteriormente.

La implemetación utilizada se puede apreciar en el siguiente enlace. Algoritmo de Strassen.

3. Datasets

Para la creación de ambos datasets se implementaron funciones que generan los datos, cada uno de los codigos se entregará en su respectiva sección.

3.1. Datasets Ordenamiento

El dataset utilizado para los algoritmos de ordenamiento se encontraran en esta carpeta. Como se puede apreciar, se poseen 4 archivos con extensión ".txt", se posee dos archivos de datos completamente aleatorios, uno con datos parcialmente ordenados y otro con datos inversamente ordenados. Estos archivos continen los principales casos que se tienen a la hora de ordenar elementos, así como también una gran variación de tamaños en sus respectivos casos, y de esta manera tener una gran variedad de datos para comprobar el comportamiento de cada algoritmo y como es su eficiencia en dichos casos.

Cada archivo sigue la siguiente estructura:

- La primera linea contiene un numero entero N que es una potencia de 10 ($10^1 \leq N \leq 10^7$) el cual indica el tamaño del arreglo.
- La segunda linea contiene N numeros enteros K ($0 \leq K < 100000$).

Además, cada codigo tiene en su función main un ciclo que recorre el archivo hasta que este se haya recorrido completamente. adicionalmente, cada archivo tiene de manera creciente la cantidad de elemento siendo la primera toma de datos de 10 y la ultima 10^7 (en el caso del Bubblesort, la toma de datos se detiene con 10^6 datos, ya que la ejecución de este tarda más de una hora).

El archivo se pasa por consola a cada algoritmo, mientras que la salida las genera creando un archivo ".txt" donde se encuentran los datos ordenados.

El codigo de generación de datos se encuentra aqui.

3.2. Datasets Multiplicación de Matrices

El datasets utilizado para la multiplicación de matrices se encuentra en esta carpeta. En esta carpeta se encuentran dos archivos ".txt" los cuales tienen los datos utilizados. El archivo "Matrices.txt" contiene matrices cuadradas, con esta se podrá observar de mejor manera la eficiencia de cada algoritmo de multiplicación, mientras que el archivo "MatricesR.txt" contiene varias matrices con distintos tamaños, estas fueron implementadas para observar la diferencia de eficiencia entre el algoritmo iterativo cubico tradicional con el optimizado.

Matrices.txt sigue el siguiente formato:

- La primera linea contiene 3 entero de tamaño N, siendo N una potencia de 2. Los 3 enteros son iguales ($2^4 \leq N \leq 2^{11}$).
- Los siguientes datos estan conformados por $N * N$ numero K ($0 \leq K < 10$) que corresponden a la primera matriz.
- Los ultimos datos estan conformados por $N * N$ numero K ($0 \leq K < 10$) que corresponden a la segunda matriz.

Mientras que, MatricesR.txt sigue el siguiente formato:

- La primera linea contiene 3 entero (N, M, R), los 3 elementos pueden ser o no iguales. ($1 \leq N, M, R \leq 3000$)
- Los siguientes datos estan conformados por $N * M$ numero K ($0 \leq K < 10$) que corresponden a la primera matriz.
- Los ultimos datos estan conformados por $M * R$ numero K ($0 \leq K < 10$) que corresponden a la segunda matriz.

Cada algoritmo en su función main posee un ciclo que permite leer todos los datos que se encuentren en el archivo. Los archivos se pasan por consola para la ejecución del programa, y las salidas son mediante la creación de un archivo ".txt" que caracteriza cual algoritmo fue el que se ejecuto.

El codigo de generación de datos se encuentra aqui.

4. Resultados

En esta sección se mostraran los datos obtenidos a través de los casos de prueba realizados por cada algoritmo. Dado que algunos de los algoritmos ante una gran cantidad de datos tardaban mucho en ser ejecutados, se puso como limite de tiempo una hora y media, por lo que los "(−)" en las tablas significan que sobrepasaron el limite de tiempo establecido.

Además resaltar que los graficos utilizados fueron acotados en los ejes para mostrar con mejor claridad el comportamiento que tuvieron los algoritmos.

4.1. Resultados Algoritmos de Ordenamiento

Aqui se pueden ver los tiempos que tardaron cada algoritmo ante el dataset de datos parcialmente ordenados.

tiempo (en segundos) de datos parcialmente ordenados				
Cantidad de datos	Bubblesort	Mergesort	quicksort	std::sort()
10^1	0.00000	0.00000	0.00000	0.00000
10^2	0.00001	0.00001	0.00004	0.00000
10^3	0.00094	0.00008	0.00423	0.00004
10^4	0.09476	0.00095	0.42194	0.00052
10^5	9.51766	0.01061	43.65962	0.01109
10^6	−	0.12146	−	0.06239
10^7	−	1.46921	−	0.83163

A continuación se muestra graficamente los tiempos de los algoritmos.

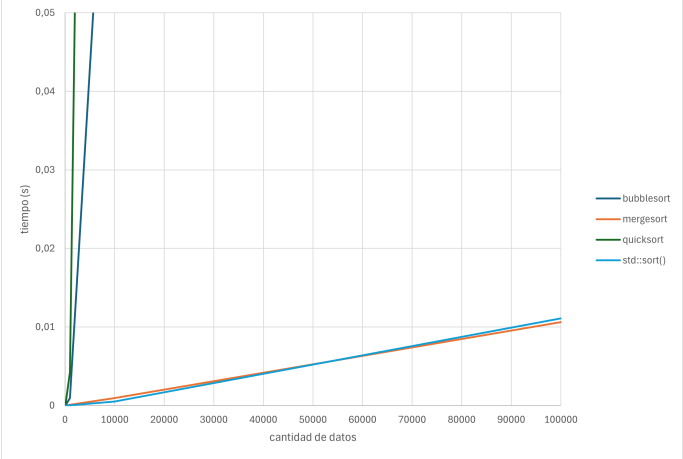

Figura 2: Grafico del tiempo vs la cantidad de datos

Tal como se ve, los algoritmos que lograron ejecutar todos los casos sin excederse del limite de tiempo establecido fueron los algoritmos de mergesort y el sort de la libreria estandar, siendo el más rapido el std::sort() y el más lento el quicksort.

En el siguiente casos se presenta ante datos inversamente ordenados.

tiempo (en segundos) de datos inversamente ordenados				
Cantidad de datos	Bubblesort	Mergesort	quicksort	std::sort()
10^1	0.00000	0.00000	0.00000	0.00000
10^2	0.00004	0.00001	0.00003	0.00000
10^3	0.00415	0.00008	0.00262	0.00004
10^4	0.42718	0.00095	0.24740	0.00045
10^5	42.00425	0.01111	19.16094	0.00536
10^6	–	0.12991	–	0.06723
10^7	–	1.41819	–	0.78907

En este caso, tanto std::sort() como mergesort siguen liderando en velocidad de ejecución, mientras que bubblesort es ahora más lento en este caso de prueba. A continuación se muestra graficamente los tiempos de los algoritmos.

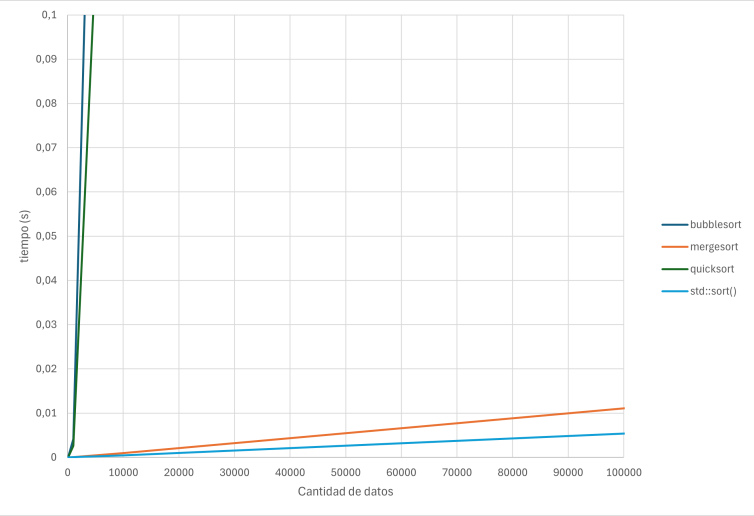

Figura 3: Grafico del tiempo vs la cantidad de datos

Llendo hacia los casos donde los datos son completamente aleatorios, se tiene lo siguiente.

tiempo (en segundos) de datos aleatorios 1				
Cantidad de datos	Bubblesort	Mergesort	quicksort	std::sort()
10^1	0.00000	0.00000	0.00000	0.00000
10^2	0.00003	0.00001	0.00001	0.00001
10^3	0.00319	0.00012	0.00012	0.00010
10^4	0.37945	0.00152	0.00154	0.00128
10^5	39.85485	0.01873	0.01926	0.01574
10^6	3959.88109	0.20300	0.24498	0.17694
10^7	–	2.24091	6.55425	1.90280

Como se puede ver, a diferencia de los resultados anteriores, quicksort ahora mejora su tiempo, esto debido a que mientras más dispersos esten sus datos, la eficiencia de este mejorara, mientras que bubblesort sigue igual de lento sin importar que haya cambiado la distribución de dato.

Aqui se podrá corroborar que lo dicho anteriormente realmente pasa así y no es solo una coincidencia:

tiempo (en segundos) de datos aleatorios 2				
Cantidad de datos	Bubblesort	Mergesort	quicksort	std::sort()
10^1	0.00000	0.00000	0.00000	0.00000
10^2	0.00004	0.00002	0.00001	0.00001
10^3	0.00306	0.00012	0.00011	0.00010
10^4	0.37785	0.00149	0.00148	0.00129
10^5	39.26512	0.02082	0.01833	0.01537
10^6	4004.69991	0.20206	0.25366	0.17675
10^7	–	2.22912	6.64509	1.92778

Y con estos datos recopilados, se puede apreciar que solo en el caso de quicksort la eficiencia mejora dependiendo la distribución de los datos, mientras que los otros algoritmos se mantienen invariantes. Esto se ve mucho más claramente en el bubblesort, teniendo que esperar aproximadamente una hora para el ordenamiento de datos. A continuación se muestra graficamente ambos casos aleatorios

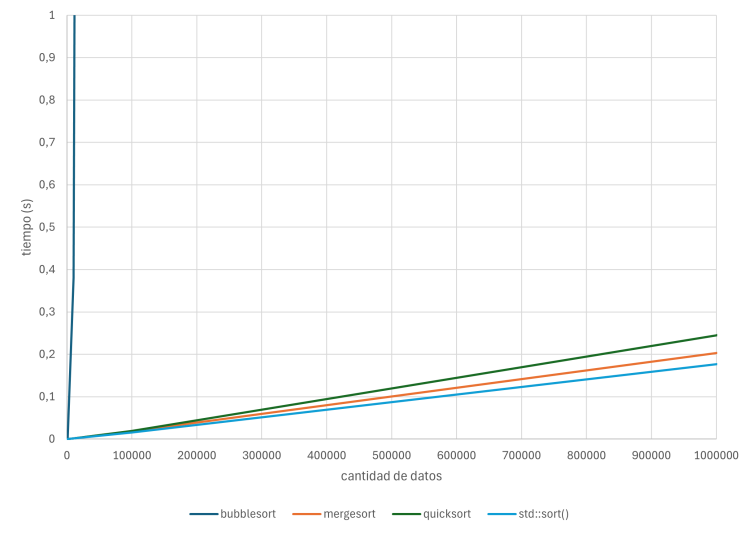

Figura 4: Grafico del tiempo vs la cantidad de datos (Aleatorio 1)

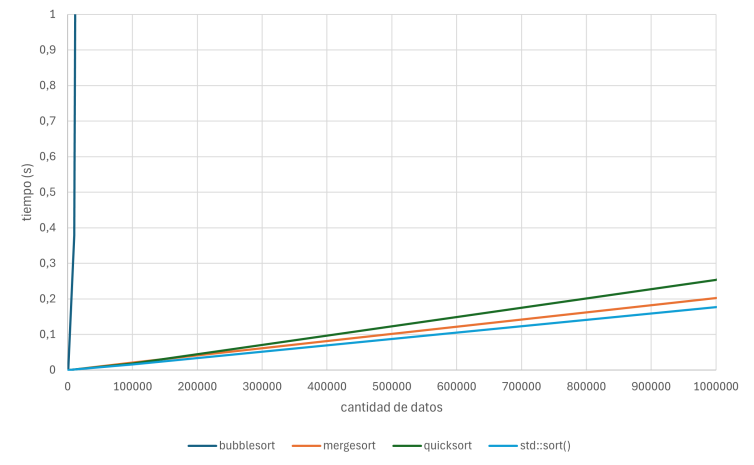

Figura 5: Grafico del tiempo vs la cantidad de datos (Aleatorio 2)

4.2. Resultados Algoritmos de Multiplicación de Matrices

A continuación se veran los tiempos de los distintos algoritmos al ser sometidos al datasets de matrices con distintas dimesiones.

tiempo (en segundos) de matrices con distintas dimensiones		
Multipliación	iterativo cubico tradicional	iterativo cubico optimizado
5x5 × 5x1	0.00000	0.00000
12x7 × 7x3	0.00000	0.00000
37x9 × 9x4	0.00002	0.00002
2x43 × 43x11	0.00001	0.00001
45x7 × 7x30	0.00011	0.00011
3x996 × 996x1130	0.04723	0.03681
209x2254 × 2254x1003	8.19339	5.33794

Como se puede observar, el algoritmo de Strassen no participa dado que este algoritmo opera bajo matrices cuadradas. De los datos obtenidos, se puede apreciar que el algoritmo iterativo cubico tiene unos tiempos más bajos a medida que crece el tamaño de las matrices. A continuación se muestra graficamente los tiempos de los algoritmos.

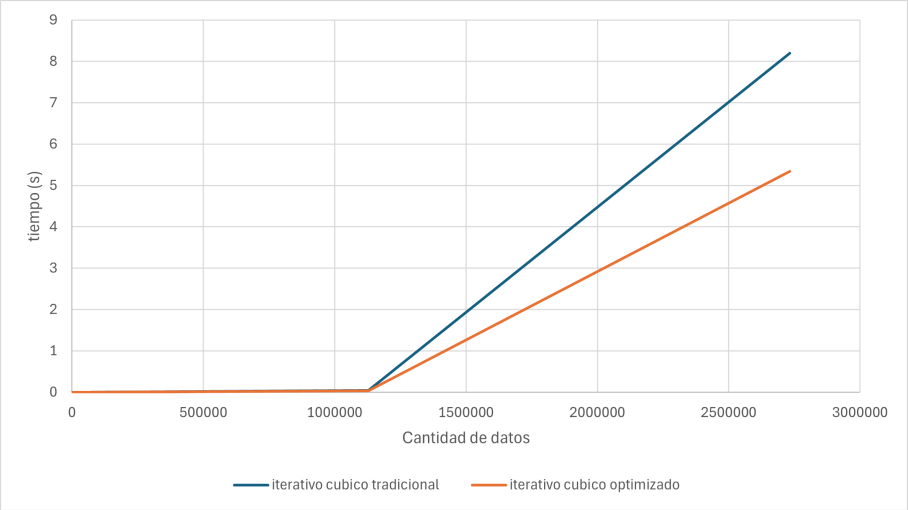

Figura 6: Grafico del tiempo vs la cantidad de datos

Y ahora, se muestra el tiempo de ejecución de cada algoritmo ante matrices cuadradas.

tiempo (en segundos) de matrices cuadradas			
Multiplicación	iterativo cubico tradicional	iterativo cubico optimizado	Strassen
16x16 × 16x16	0.00005	0.00005	0.00839
32x32 × 32x32	0.00037	0.00036	0.05562
64x64 × 64x64	0.00285	0.00288	0.39927
128x128 × 128x128	0.02271	0.02692	2.77594
256x256 × 256x256	0.18326	0.18218	19.26139
512x512 × 512x512	1.61289	1.46757	135.02700
1024x1024 × 1024x1024	14.79767	12.21480	947.39903
2048x2048 × 2048x2048	151.85970	95.07030	–

En esta muesta se puede ver uno de los casos más particulares del trabajo realizado, ya que a pesar de que el algoritmo de Strassen tiene una complejidad que acota más bajo que a la de los algoritmos tradicionales. El impacto que este tiene no se puede ver con una cantidad tan pequeña de datos, para poder observar la disminución de complejidad que este algoritmo tiene se necesita un poder de computo mucho más grande, asi como también más tiempo para ver su impacto. A continuación se muestra graficamente los tiempos de los algoritmos.

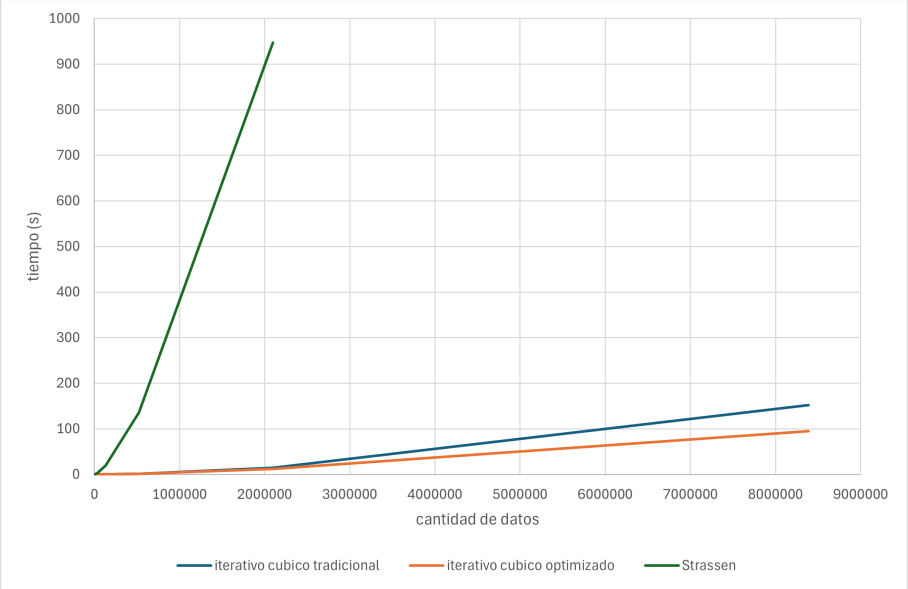

Figura 7: Grafico del tiempo vs la cantidad de datos

5. Conclusión

A trves del trabajo realizado, se pudo apreciar como la manera de programar distintas soluciones para un mismo problema puede mejorar el rendimiento y de esta manera lograr reducir el tiempo que un programa tarda en ejecutarse.

Como se pudo apreciar en los algoritmos de ordenamientos, siendo el bubblesort el algoritmo más caro de ejecutar, teniendo una complejidad de $O(n^2)$ para todos sus casos, este comenzaba a demorarse cada vez más, siendo ineficiente a gran escala de elementos, al punto de no lograr resolver en el tiempo estipulado casos los cuales, algoritmos como mergesort, quicksort y std::sort(), si lograron resolver. Pero también se puede ver como un algoritmo inplace mejora el tiempo de ejecución de un programa, siendo este el caso del std::sort(), el cual a través de su modelo hibrido el cual no posee espacio auxiliar y teniendo una complejidad algoritmica de $O(n \log n)$, logra tener mejores tiempos que el algoritmo mergesort que es un algoritmo no-inplace dado que ocupa un espacio auxiliar, siendo este $O(n)$ y con complejidad para todos sus casos de $O(n \log n)$ como se menciono en su debido momento. El algoritmo quicksort a pesar de ser uno de los más rapido debido a que este no posee espacio auxiliar y en su caso promedio su complejidad es de $O(n \log n)$ siendo este un algoritmo inplace, obtuvo peores tiempos que el mergesort, esto se debe a que a pesar de que quicksort sea un algoritmo inplace, la cota que tenía la muestra de datos era tal que era más complicado para el algoritmo correr de manera más eficiente, aunque a pesar de esto, a una escala mayor de datos, el algoritmo quicksort logra mejorar en tiempos al mergesort gracias a que este no necesita dicha memoria auxiliar.

En el caso de las matrices, en los primer caso de prueba, se pudo apreciar una mejora en el tiempo del algoritmo de multiplicación, a pesar que el algoritmo iterativo cubico tradicional y optimizado tengan la misma complejidad de $O(n^3)$, la mejora de tiempo que tiene el optmizado se debe a la cercania de los datos que esta tenía, demostrando que es un factor importante a la hora de programar tener un eficiente conocimiento de alojamiento de datos y como se comportan estos. Para el segundo caso, donde entra a participar el algoritmo de Strassen, se puede apreciar como no se obtuvieron los tiempos esperados, esto ya que a pesar de que el algoritmo de Strassen tuviera una complejidad menor, siendo esta de $O(n^{\log_2 7})$, los algoritmos cubicos siguen siendo más eficiente para la cantidad de datos a los que estos programas fueron cometidos, esto en particular se debe a que el algoritmo de Strassen tiene una mayor eficiencia a una gran cantidad de datos, a los cuales, para este trabajo no se pudieron llegar.

Como conclusión, se puede apreciar en base a este trabajo, como las formas de programar pueden afectar al rendimiento de resolusión de un problema, teniendo en cuenta como se distribuyen los espacios auxiliares y como a través de simplificación de operaciones o manipulación de alojamiento de

datos estos tienen un gran impacto.

6. Anexo

- Se uso de referencia el algoritmo de bubblesort de Geeks for Geeks. Bubblesort
- Se uso de referencia el algoritmo de mergesort de Geeks for Geeks. Mergesort
- Se uso de referencia el algoritmo de quicksort de Geeks for Geeks. Quicksort
- Se uso de referencia el algoritmo de strassen de Geeks for Geeks. Algoritmo de Strassen