Universidad Técnica Federico Santa María
Algoritmos y Complejidad

Alejandro Vergara Araneda
202273568-6

Algoritmos Dividir y Conquistar

En el presente informe se investigo sobre diversos algoritmos de ordenamiento de vectores y multiplicación de matrices con distintos dataset con el propósito de analizar sus complejidades experimentalmente y relacionarlas con la teoría, los algoritmos utilizados son ; selection sort, mergesort, quicksort, sort brindado por C++, y para matrices, el algoritmo iterativo cubico tradicional, el algoritmo iterativo cubico transpuesto, y el algoritmo de Strassen. De esta investigación se espera que los algoritmos mas rápidos sean mergesort y quicksort para ordenamiento y Strassen para multiplicación de matrices.

Todos los algoritmos estarán el siguiente repositorio de GitHub, donde también estarán los Excel con los valores medidos y sus respectivos gráficos, y también estarán los programas utilizados para generar los dataset, estos son datasetgenerator.cpp, el de vectores y el de matrices están en su respectiva carpeta.

Para la toma de tiempos se considero utilizar “time” al momento de correr los códigos para medir los tiempos, ya que se puede ver el tiempo que utilizo el programa y también el tiempo de las llamadas al sistema, específicamente da el tiempo de **sys**, que son las llamadas al sistema, el tiempo de **user**, que es el tiempo que realmente utilizado el algoritmo y el tiempo **real** que es la suma de los últimos dos.

Para las pruebas se utilizaron nData datos de un valor entre 0 y 1000000 para vectores y 0 y 999 para matrices

Para compilar los algoritmos se realizo en “Ubuntu 18.04.6”, en donde hay que posicionarse en el respectivo directorio del algoritmo a compilar y utilizar

```
g++ -o out -Wall archivo.cpp
```

y para correr el programa, viendo los tiempos se utilizo el comando

```
time ./out
```

Algoritmos de Ordenamiento

Los Algoritmos de ordenamiento están en la carpeta Algoritmos_de_ordenamiento, aquí se puede encontrar las carpetas que llevan a cada algoritmo y dos archivos cpp, uno es el que genera el dataset y el otro es common, este ultimo es el programa que guarda, el tamaño del vector, y posee las funciones generar_arr() que lee el dataset y lo guarda en un vector, y también generar_txt, que genera el archivo TXT de salida con el vector ordenar, este archivo se creo para que las acciones fuera de los vectores de ordenamiento tarden un tiempo muy similar.

Alejandro Vergara Araneda
202273568-6

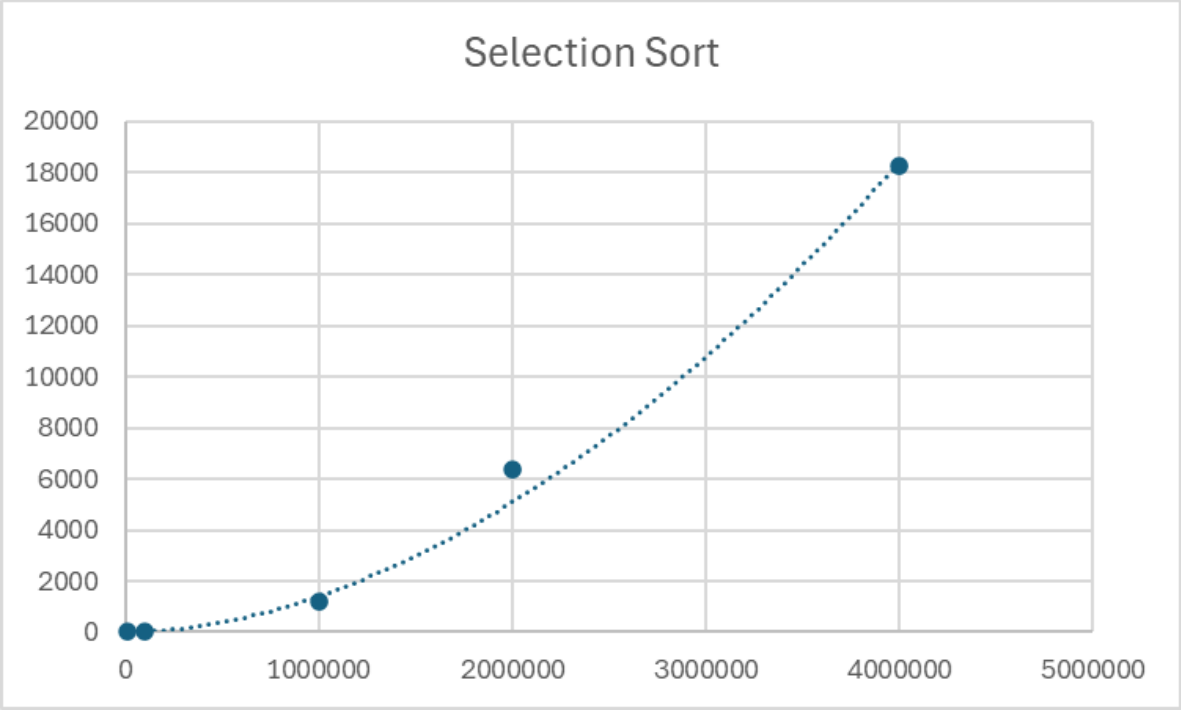
1- Selection Sort

Este algoritmo recorre todo el vector buscando un elemento menor al elemento en el índice i, si lo encuentra, los intercambia y continua con i+1 hasta recorrer todo el vector, esto se realiza con un ciclo anidado. Este algoritmo es bastante ineficiente ya que tiene una complejidad de **O(n^2)**, en el mejor y peor caso ya que aunque el vector este ordenado, igual tendrá que recorrerlo entero en los dos for anidados. Este es un algoritmo in-place y estable.

Los resultados obtenidos fueron los siguientes:

	Tiempo (Segundos)		
n	real	user	sys
4000000	18282.245	18239.953	2.031
1000000	1222.089	1219.141	0.156
100000	16.427	16.016	0.094
10000	0.340	0.313	0.016

De esta tabla se puede apreciar que para 4 millones de datos, el algoritmo se demorará mas de 5 horas, es importante tener en cuenta esto ya que en su grafico personal no se aprecia la magnitud de lo que tarda;



Alejandro Vergara Araneda
202273568-6

2- Mergesort

Este es un algoritmo de dividir y conquistar, se utiliza la recursión para dividir el vector, en 2, luego realiza una copia para cada uno de los subvectores, y los va recorriendo al mismo tiempo, guardando el menor valor en la posición del vector original. Este es un algoritmo estable.

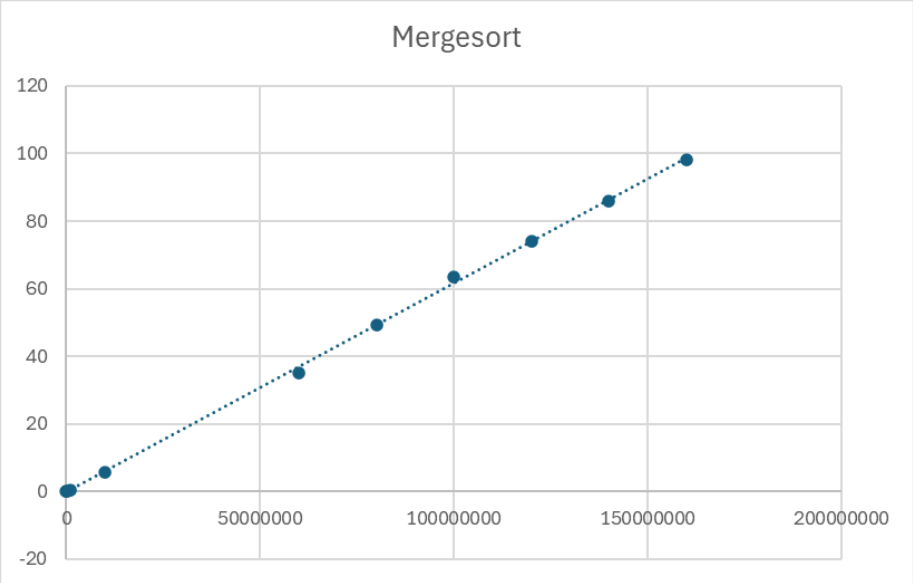
Para este algoritmo tendremos un tiempo de ejecución del tipo:

$$T(n) = 2T\left(n/2\right) + O(n)$$

Lo cual indica que según el teorema maestro tendrá una complejidad de **O(n log(n))**, como este algoritmo siempre divide a la mitad, en el mejor y peor caso también será **O(n log(n))**.

Los resultados obtenidos fueron los siguientes:

	Tiempo(s)		
n	**real**	**user**	**sys**
160000000	**102.599**	**98.188**	**3.641**
140000000	**89.414**	**85.797**	**3.188**
120000000	**77.983**	**74.078**	**2.797**
100000000	**66.539**	**63.375**	**2.656**
80000000	**51.418**	**49.359**	**1.688**
60000000	**36.980**	**35.266**	**1.219**
10000000	**6.111**	**5.859**	**0.219**
1000000	**0.546**	**0.484**	**0.047**
100000	**0.119**	**0.094**	**0.016**
10000	**0.039**	**0.016**	**0.031**



Se presencia una gran mejora en comparación con selection sort ya que con mergesort se obtiene un poco mas de 100 segundos para 160 millones de datos

Alejandro Vergara Araneda
202273568-6

3- Quicksort

Este también es un algoritmo de dividir y conquistar, en este se elige un pivote, y se revisan todos los valores, los que sean menores quedaran a la izquierda y los mayores a la derecha, de esta manera nos aseguramos que el pivote quedara en su posición real, luego se hace una llamada recursiva para los menores y para los mayores. Este es un algoritmo in-place.

Para este algoritmo tendremos un tiempo de ejecución del tipo:

$$T(n) = 2T\left(n/2\right) + O(n)$$

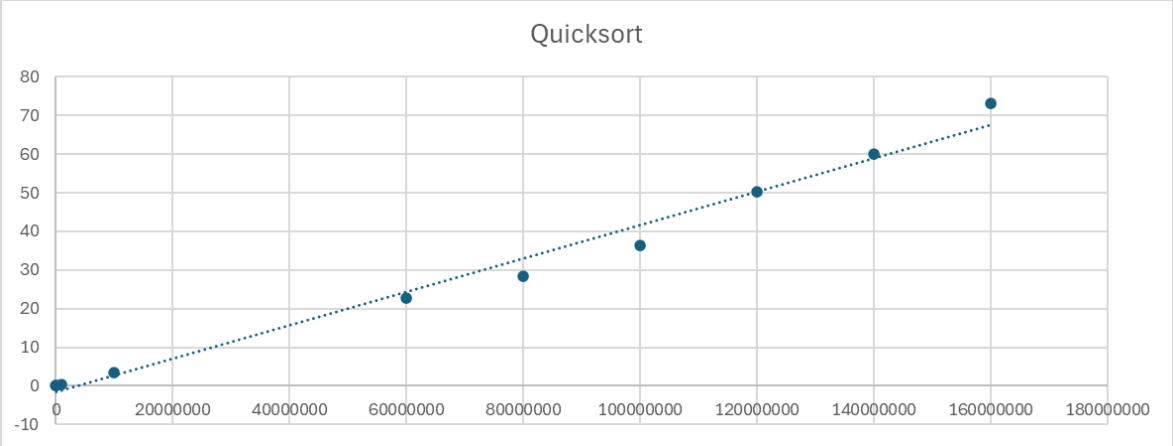
Lo cual indica que según el teorema maestro tendrá una complejidad de **O(n log(n))**, sin embargo este es el caso promedio, si se escoge mal el pivote tendremos que ir recorriendo todo el vector como lo haríamos en Selection Sort dejándonos en el peor caso

$$T(n) = 2T(n) + O(n)$$

Lo cual nos deriva a una complejidad en el peor caso de **O(n^2)**, el mejor caso es cuando el vector se va dividiendo justo en la mitad, este caso también quedaría como **O(n log(n))**.

Los resultados obtenidos fueron los siguientes:

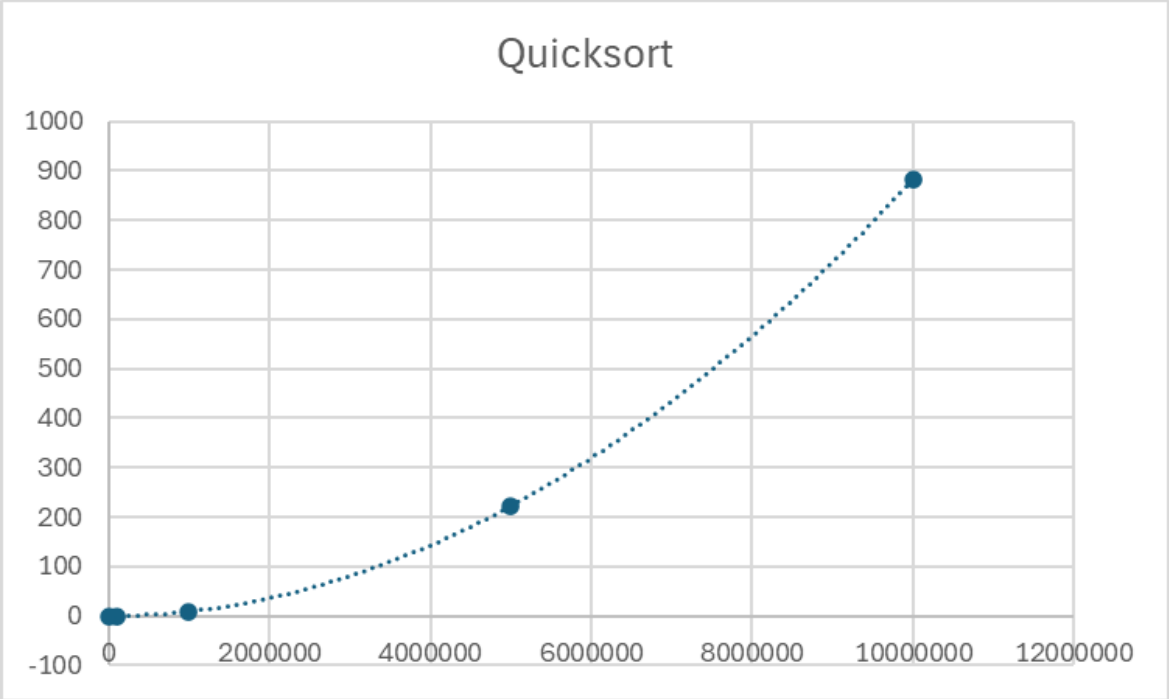
	Tiempo		
n	real	user	sys
160000000	75.856	73.188	2.016
140000000	62.736	59.922	2.359
120000000	51.924	50.156	1.578
100000000	37.889	36.328	1.328
80000000	30.045	28.438	1.344
60000000	24.905	22.719	0.984
10000000	3.679	3.422	0.203
1000000	0.477	0.422	0.047
100000	0.135	0.078	0.031
10000	0.025	0.000	0.000



Alejandro Vergara Araneda
202273568-6

Para este algoritmo también se decidió probar con datos con menor dispersión, que se utiliza siempre en el dataset es de valores entre 0 y 1000000, y se probo con valores entre 0 y 100, (también se probo con otros algoritmos pero no había un cambio relevante por ende se utilizo el por defecto) dejando unos resultados expresados por la siguiente tabla:

	Tiempo		
n	real	user	sys
10000000	841.887	882.75	0.781
5000000	223.909	222.391	0.188
1000000	8.781	8.609	0.063
100000	0.289	0.188	0.016
10000	0.028	0.000	0.016

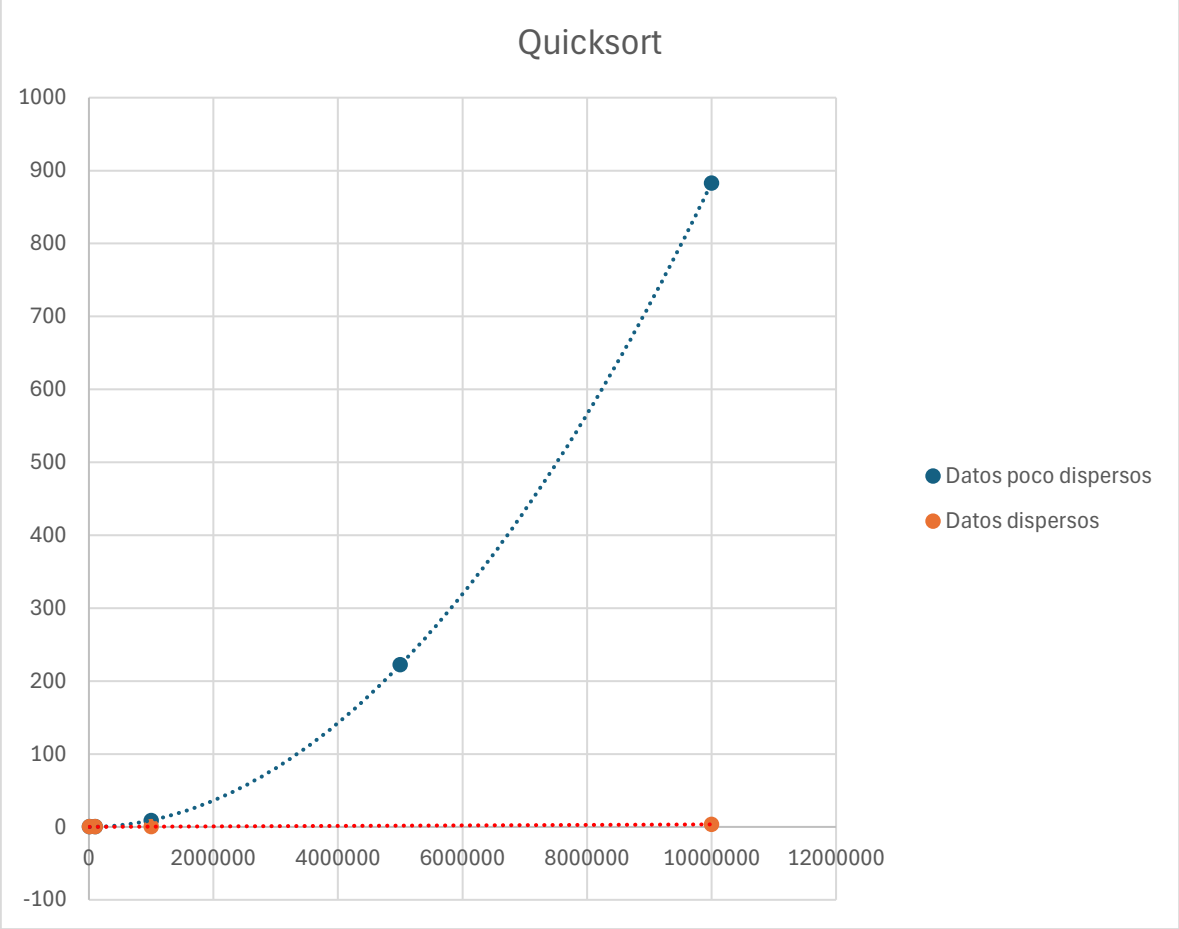


De los datos se puede ver claramente que crece de forma exponencial, y por ende tarda mas, que si tuviera los datos mas dispersos, esto pasa porque al haber menos variedad de datos habrán muchos repetidos, lo que quiere decir que el pivote no será único, y será un caso degenerado, es decir donde el pivote genera subvectores muy desbalanceados.

Universidad Técnica Federico Santa María
Algoritmos y Complejidad

Alejandro Vergara Araneda
202273568-6

El siguiente grafico compara el algoritmo de quicksort para datos dispersos y poco dispersos:



En este grafico se puede presenciar los datos dispersos con color anaranjado y los datos poco dispersos con color azul, aquí se puede evidenciar el gran crecimiento de los datos poco dispersos por acercarse al peor caso, crece tanto que en el caso común no se puede siquiera distinguir el crecimiento.

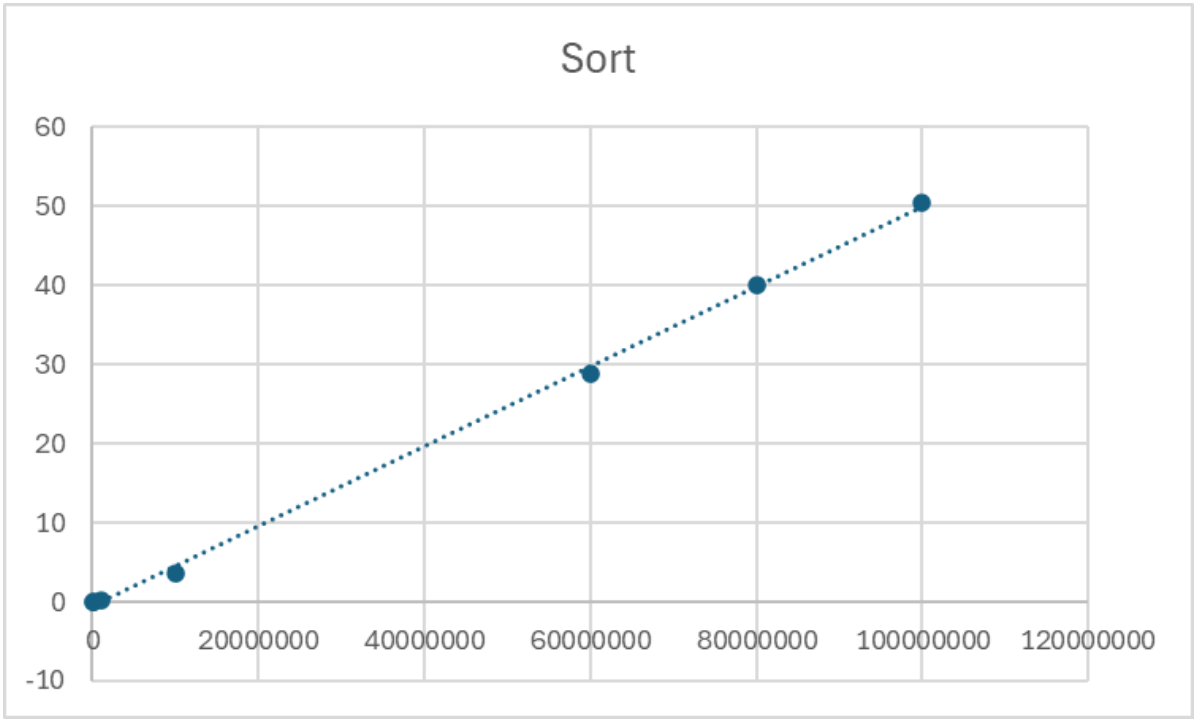
Alejandro Vergara Araneda
202273568-6

4- Sort

Como algoritmo por defecto de C++ se utilizo sort de la librería algorithm que realiza un quicksort, por ende sus complejidades en el mejor caso y en el caso promedio será de **O(n log(n))**, y en el peor caso será de **O(n^2)**.

Los resultados obtenidos fueron los siguientes:

	Tiempo		
n	**real**	**user**	**sys**
100000000	**52.472**	**50.359**	**1.625**
80000000	**42.268**	**39.984**	**1.313**
60000000	**30.365**	**28.813**	**0.938**
10000000	**4.058**	**3.641**	**0.281**
1000000	**0.419**	**0.281**	**0.016**
100000	**0.079**	**0.063**	**0.016**
10000	**0.037**	**0.016**	**0.000**



Se puede apreciar que es ligeramente mas lento que Quicksort, pero ligeramente mas rápido que mergesort

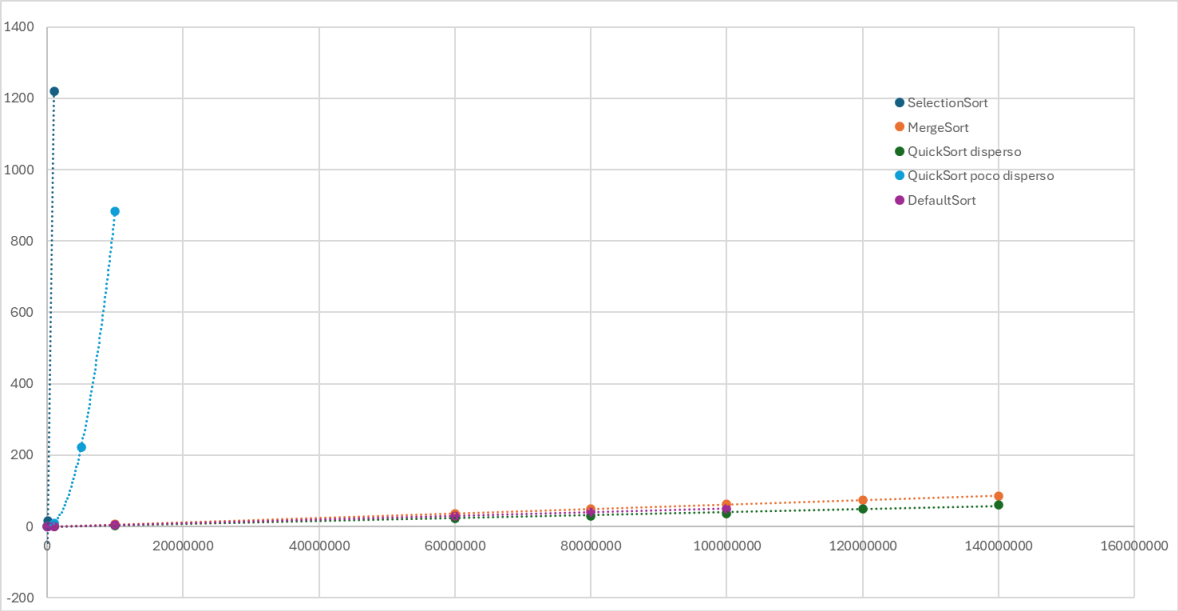
Alejandro Vergara Araneda
202273568-6

Conclusiones de algoritmos de ordenamiento

Analizando a fondo los datos obtenidos se concluye que claramente hay algunos algoritmos mejores que otros, por ejemplo selection sort, seria muy poco recomendado dado que se tarda demasiado, y en general quicksort fue mas rápido que mergesort pero depende de la distribución de los datos a ordenar, si los valores están bien distribuidos seria mejor quicksort, en caso contrario mergesort.

Para resumir se realizo el siguiente grafico que compara todos los algoritmos ya presentados, se eliminaron algunos puntos para intentar que sea un poco mas agradable a la vista.

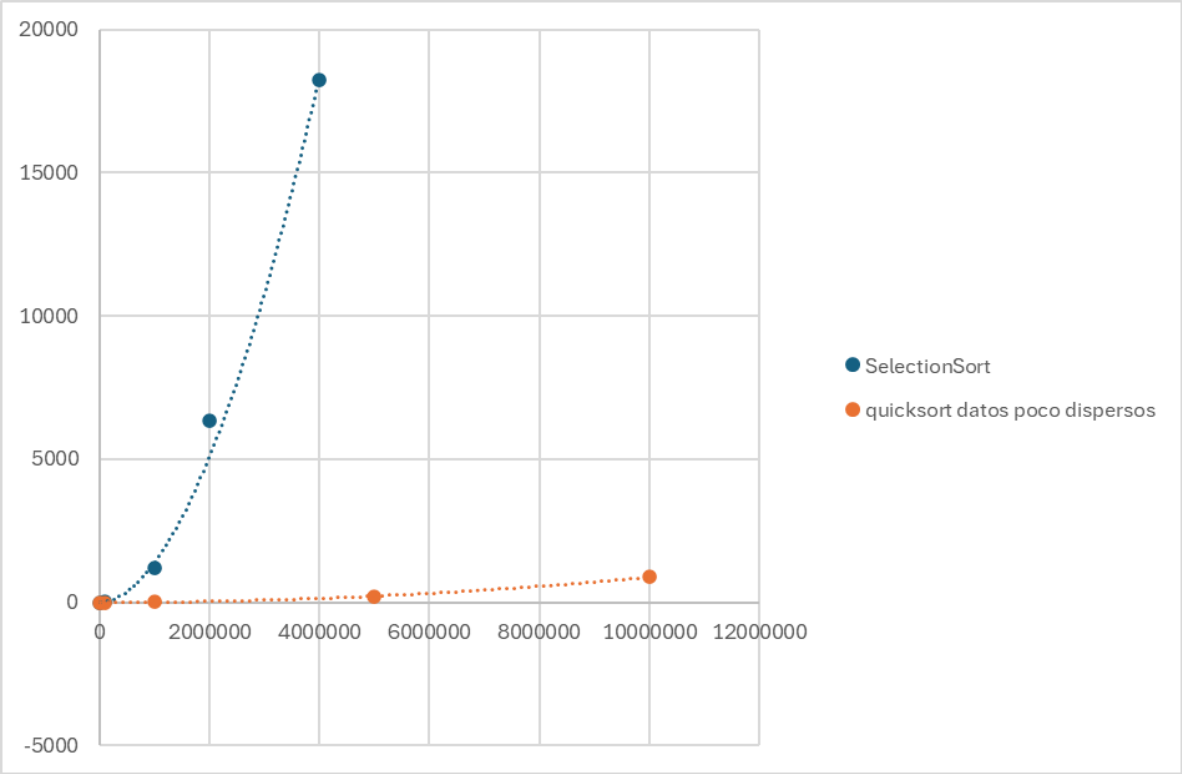
Recordar que todos los gráficos están presentes en el repositorio de GitHub



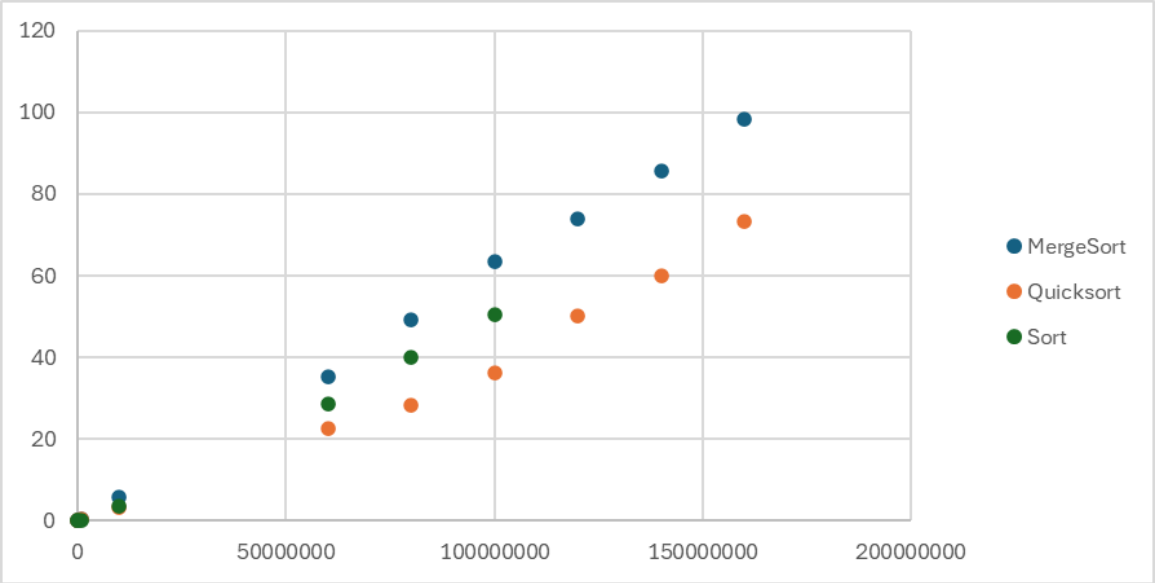
Se presencia una gran diferencia entre los algoritmos, donde selectionsort (representado con el azul) crece muy rápido, seguido de Quicksort con valores cercanos entre si (representado por celeste), y luego los mas eficaces son quicksort para valores dispersos entre si (verde), sort por defecto(morado), y mergesort (naranjo).

Alejandro Vergara Araneda
202273568-6

Se opto por separar los gráficos para que los resultados sean mas analizables;



En este grafico se observa que aunque selectionsort este relativamente cercano a su peor caso, aun así no se compara con Selection sort con respecto a cual de los dos tarda mas



En este ultimo grafico se aprecia mejor cuanta diferencia hay entre los 3 mejores algoritmos de ordenamiento considerados

Universidad Técnica Federico Santa María
Algoritmos y Complejidad

Alejandro Vergara Araneda
202273568-6

Algoritmos de multiplicación de matrices

Los algoritmos de multiplicación de matrices están en su respectiva carpeta Multiplicacion_de_matrices, donde al igual que algoritmos de ordenamiento tendrá un archivo common.cpp con las funciones básicas y también datasetgenerator.cpp que genera el dataset para una matriz cuadrada de nData x nData y lo guarda en un archivo txt.

1- Algoritmo Tradicional

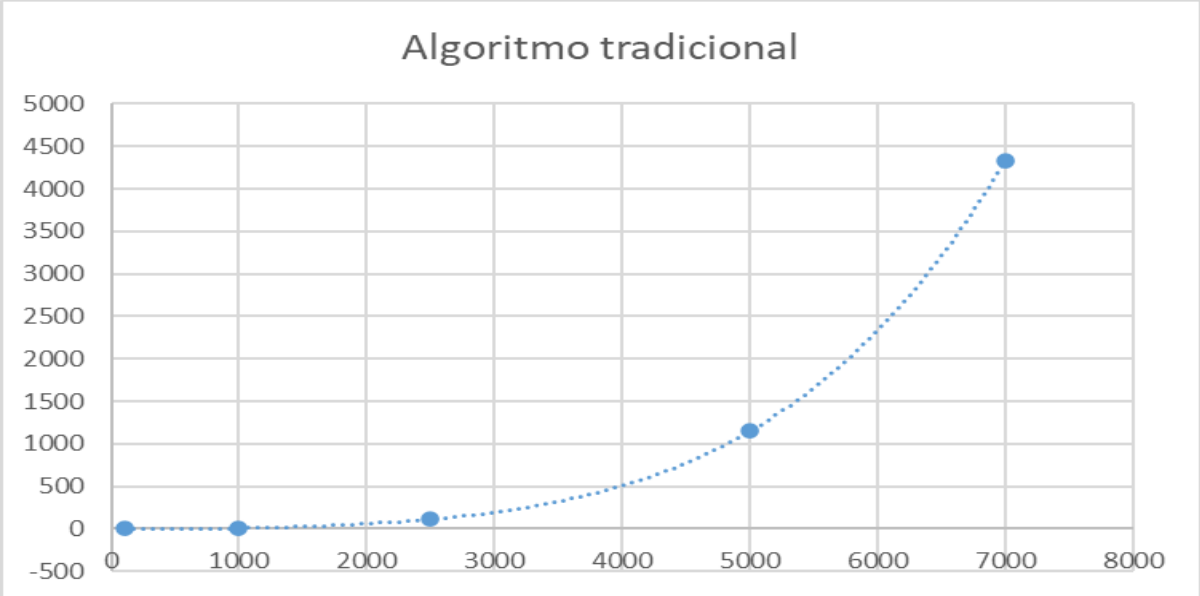
Este algoritmo se basa en ir multiplicando la matriz en base a la formula;

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] * B[k][j]$$

Para la cual necesitaremos 2 ciclos for anidados, para i, j, k, por ende tendremos una complejidad de **O(n^3)**.

Los resultados obtenidos fueron los siguientes:

	Tiempo		
n	real	user	sys
7000	4390.690	4335.813	5.344
5000	1161.872	1146.188	1.422
2500	113.413	112.375	0.203
1000	6.735	6.609	0.031
100	0.052	0.031	0.000



Alejandro Vergara Araneda
202273568-6

2- Algoritmo con matriz transpuesta

Este algoritmo sigue la misma idea del anterior, sin embargo primero transpone la segunda matriz con el propósito de aprovechar el como trabaja el computador, dado que las matrices son un gran arreglo de las filas consecutivas, por ende el procesador puede precargar bloques de datos en la cache haciendo que el acceso a los datos sea mas rápido.

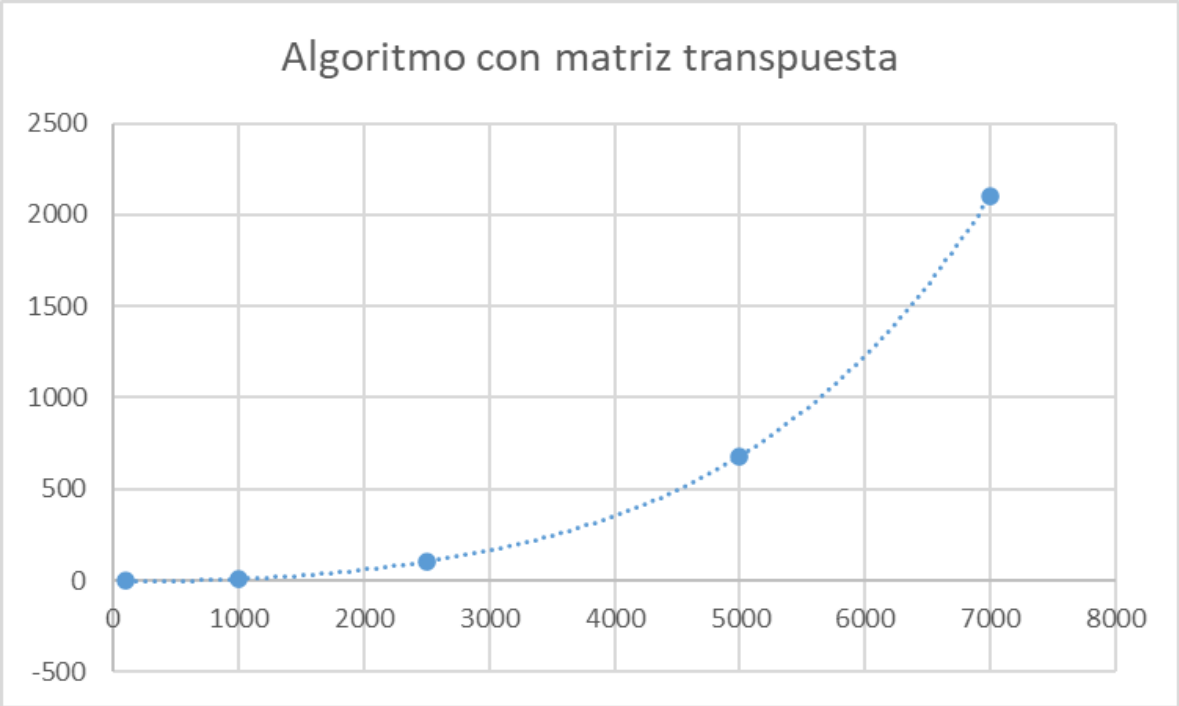
Para este caso como la matriz B ahora esta transpuesta la formula cambia a;

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] * B[j][k]$$

Aun así estos cambios no sirven para mejorar la complejidad del problema **O(n^3)**.

Los resultados obtenidos fueron los siguientes:

	Tiempo		
n	real	user	sys
7000	2225.168	2105.328	4.831
5000	685.221	681.313	1.281
2500	105.878	104.109	0.234
1000	7.590	7.422	0.078
100	0.056	0.031	0.031



Se observa que para pocos datos es mejor el algoritmo tradicional, pero conforme van aumentando, el Algoritmo con la matriz transpuesta es mucho mejor

Alejandro Vergara Araneda
202273568-6

3- Algoritmo de Strassen

El algoritmo utilizado fue creado por lucasletum y esta en el siguiente enlace, se modifico un poco pero se intento ser lo mas fiel posible a lo que se publico en esa pagina. Este algoritmo es parte de dividir y conquistar, que lo que hace es dividir las 2 matrices en 4, formando 8 submatrices, luego se calculan 7 ecuaciones con esas 8 particiones que luego nos servirán para calcular los resultados de la multiplicación sumando y restando matrices.

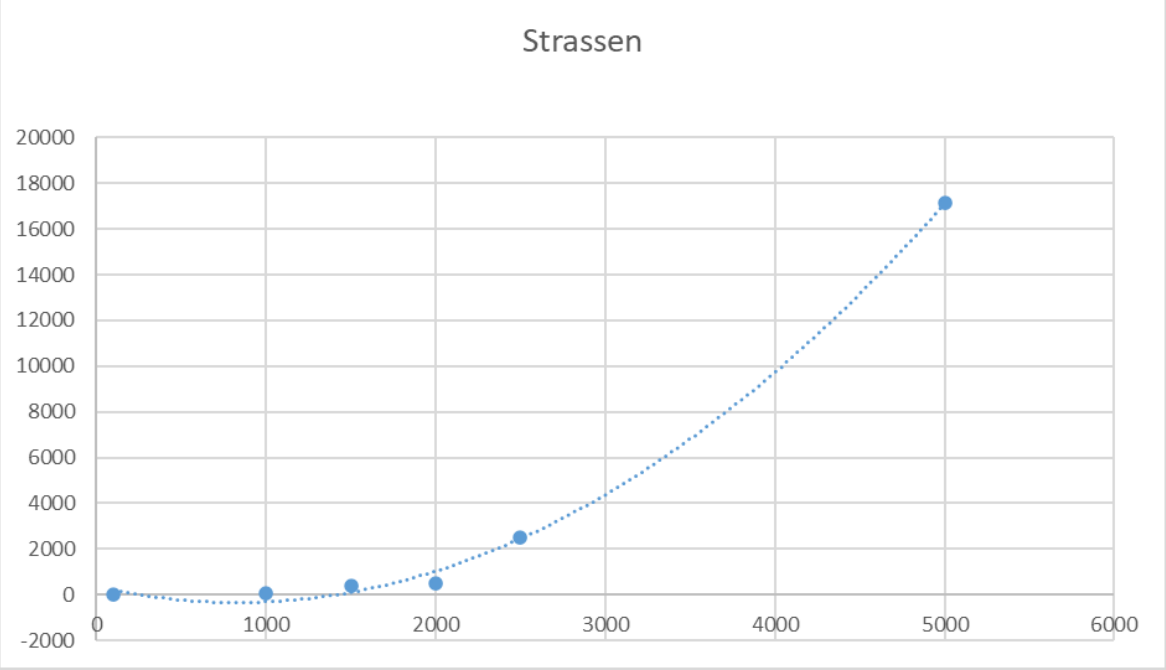
Para este algoritmo tendremos un tiempo de ejecución del tipo:

$$T(n) = 7 * T(n/2) + O(n^2)$$

Lo cual según el teorema maestro se dicta una complejidad $\mathbf{O}(\mathbf{n}^{\log_2(7)})$ o mejor expresado como **O($n^{2,8}$)**.

Los resultados obtenidos fueron los siguientes:

	Tiempo		
n	real	user	sys
5000	17172.216	17137.438	3.938
2500	2529.006	2518.109	0.641
2000	490.469	486.453	0.531
1500	381.112	376.922	0.281
1000	59.799	59.375	0.063
100	0.324	0.328	0.016



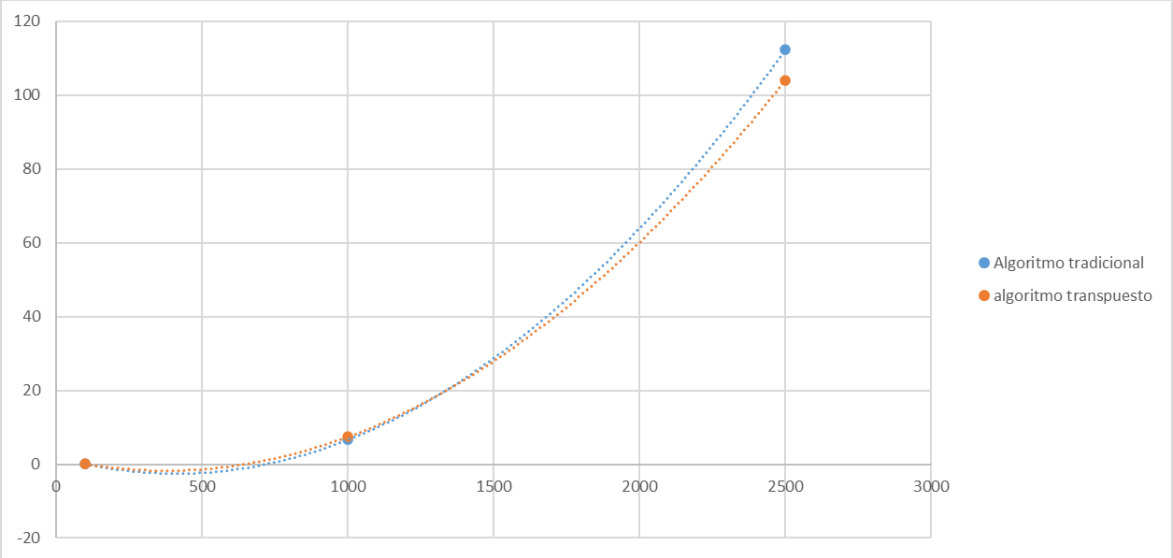
Se observa que este algoritmo se tarda mas que los anteriores, a pesar de tener menor complejidad, esto es debido a que las constantes ocultas son muy grandes ya que se tienen varios for anidados tanto para multiply_matrix y add_matrix.

Universidad Técnica Federico Santa María
Algoritmos y Complejidad

Alejandro Vergara Araneda
202273568-6

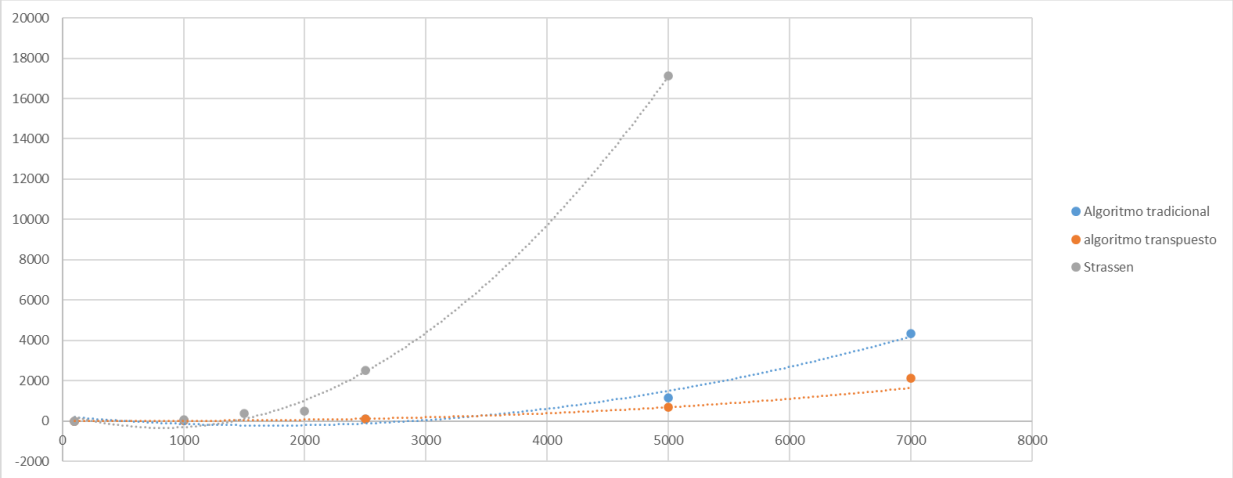
Conclusiones algoritmos de multiplicación de matrices

Analizando a fondo los datos obtenidos se concluye que hay casos en donde la complejidad no satisface toda la información necesaria para decidir si un algoritmo es mejor que otro o no, esto ya que el algoritmo de Strassen debería ser el mas rápido, sin embargo para que este cambio se note se necesitara un n demasiado grande, por ende si alguien quiere trabajar con pocos datos, es muy probable que Strassen funcione peor que el algoritmo tradicional o el algoritmo transpuesto, pasa algo parecido con el algoritmo transpuesto, ya que si lo comparamos con el tradicional, los dos tienen la misma complejidad, sin embargo para valores de n pequeños (n < 1000) funcionara mejor el tradicional, esta diferencia se puede apreciar en el siguiente grafico con n<2500:



Se observa que en n = 1000 el algoritmo transpuesto tarda ligeramente mas que el tradicional, pero para n = 2500 comienza a tardar mas el algoritmo tradicional, dejando como mas rápido al transpuesto.

Alejandro Vergara Araneda
202273568-6



De este otro grafico podemos apreciar como el algoritmo de Strassen se dispara mientras los otros se mantienen con un tiempo menor, como ya fue explicado, esto pasa porque el algoritmo de Strassen posee una constante muy grande la cual hace que el programa tarde mucho, se necesitara un n muy grande para poder apreciar que el algoritmo de Strassen tarde menos que los otros.

Conclusión general

Se concluye que para elegir un algoritmo eficiente no solo se necesita saber la complejidad, sino mas importante es conocer el problema y saber a que tipo de datos estará expuesto, con el propósito de decidir en base a eso, ya que como se vio con quicksort la diversidad de datos es importante para que trabaje bien, además de que si ya esta ordenado, este algoritmo tardara muy parecido a selection sort, o también como se analizo en los algoritmos de multiplicación de matrices, donde Strassen, el de menor complejidad fue el que mas tardo, y esto solo por la longitud de la matriz, en resumen **no porque un algoritmo tenga menor complejidad significa que será mejor, todo depende de los datos a trabajar**.