Tarea 1
Algoritmos Dividir y Conquistar

Benjamín Camus Valdés
202173072-9

Septiembre 2024

1 Introducción

En este trabajo se presentará un análisis comparativo de algoritmos basados en el paradigma de "dividir y conquistar", específicamente enfocados en problemas de ordenamiento y multiplicación de matrices. Se implementaron algoritmos como Bubble Sort, Selection Sort, Merge Sort, Quick Sort, y se utilizó la función sort de la biblioteca estándar de C++ para el análisis de ordenamiento. Para la multiplicación de matrices, se consideraron el algoritmo tradicional cúbico, un algoritmo cúbico optimizado, y el algoritmo de Strassen, destacando sus diferencias en eficiencia y comportamiento en distintos casos.
El principal objetivo de este trabajo es evaluar el rendimiento de estos algoritmos en diversos escenarios de entrada y comprarlos con su complejidad teórica para ver que tan bien nos ayuda la notación asintótica a estimar el comportamiento de los algoritmos. Los datos de entrada fueron generados a partir de datasets diseñados específicamente para probar los algoritmos bajo diferentes niveles de complejidad, incluyendo datos aleatorios, parcialmente ordenados y completamente ordenados. Para la multiplicación de matrices, se evaluaron tanto matrices cuadradas como no cuadradas.
De manera preliminar, los resultados indican que los algoritmos de ordenamiento basados en dividir y conquistar, como Merge Sort y Quick Sort ofrecen mejoras significativas en términos de complejidad asintótica respecto a métodos más simples como Bubble Sort o selection sort, y esto comprobando los análisis de complejidad.
Mientras que para los algoritmos de multiplicación de matrices, en general el método que mejor se comporta es el iterativo cúbico optimizado usando la matriz transpuesta para la segunda matriz, esto debido principalmente a la localidad de los datos y como se acceden a los valores de las matrices en el cache de la CPU. El algoritmo de Strassen cuando llega hasta la recesión base = 1, es el que peor rendimiento tiene, debido a que tantas llamadas recursivas generan un overhead excesivo en la CPU, incluso el el algoritmo cúbico tradicional para multiplicación de matrices tiene mejor rendimiento. Aunque el método de Strassen también fue probado para distintos niveles base de recursión, solo se ve la mejora para matrices de tamaño muy grandes, logra comportarse mejor en tiempo que el cúbico tradicional (solo para strassen con hojas, matrices cuadradas y para tamaños grandes de las matrices), pero solo en algunos intervalos logra ganarle al método de multiplicación de matrices optimizado, es por esto que la implementación del método de Strassen introduce desafíos adicionales relacionados con el uso de la memoria y la sobrecarga recursiva, que deben ser considerados en la práctica, debido a que no está bien optimizado para la multiplicación de matrices en computadoras modernas.

Información del hardware en el que se realizaron las pruebas:
Procesador: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
Tarjeta gráfica: NVIDIA GeForce GTX 1650 Ti
RAM instalada: 16,0 GB (15,4 GB utilizable)
Tipo de sistema Sistema operativo de 64 bits, procesador x64
Sistema operativo: Windows 11, pruebas realizadas y programadas en WSL Ubuntu 22.04.4 LTS

Link del repositorio de GitHub: https://github.com/kamus1/tarea1-algorithms

2 Descripción de los algoritmos a ser comparados

Algoritmos de ordenamiento:

1. **Bubble Sort**
   Código: https://github.com/kamus1/tarea1-algorithms/blob/main/sorting/bubble_sort.cpp

   - Mejor Caso: $O(n)$
   - Caso Promedio: $O(n^2)$
   - Peor Caso: $O(n^2)$

   Bubble Sort es un algoritmo de ordenamiento simple del tipo cuadrático. Compara elementos adyacentes en la lista. Si están en el orden incorrecto, los intercambia. Repite este proceso, pasando por la lista varias veces. En cada pasada, el elemento más grande se "burbujea" hacia el final. El algoritmo termina cuando no se necesitan más intercambios.
   El mejor caso ocurre cuando la lista o array ya está ordenado, entonces Bubble Sort solo tiene que recorrer el arreglo de tamaño n una vez $O(n)$.
   El peor caso ocurre cuando los números de la lista están desordenados completamente y el algoritmo debe hacer el máximo número de comparaciones e intercambios, es $O(n^2)$ principalmente porque tiene 2 bucles for anidados en el código, un bucle externo que recorre toda la lista n veces, y un bucle interno que compara pares adyacentes y hace intercambios si es necesario.
   Y el caso promedio ocurre con listas parcialmente ordenadas o desordenadas aleatoriamente. Aunque puede variar, en promedio el rendimiento se acerca más al peor caso que al mejor, por lo tanto también es $O(n^2)$.
   Podemos verlo también con el numero de comparaciones que va realizando, en la primera iteración del bucle externo, el bucle interno realiza $(n-1)$ comparaciones, en la segunda iteración $O(n-2)$, en la siguiente $O(n-3)$ y así sucesivamente, entonces el numero total de comparaciones es una sumatoria, la cual lleva a la formula $\frac{n(n-1)}{2}$, al aplicarle la notación Big O nos da $O(n^2)$.

2. **Selection Sort**
   Código: https://github.com/kamus1/tarea1-algorithms/blob/main/sorting/selection_sort.cpp

   - Mejor Caso: $O(n^2)$
   - Caso Promedio: $O(n^2)$
   - Peor Caso: $O(n^2)$

   Selection Sort es un algoritmo de ordenamiento simple, también del tipo cuadrático. Lo que hace es seleccionar el elemento más pequeño en cada iteración y lo coloca en su posición correcta. Comienza desde el inicio de la lista y busca el elemento mínimo de todo el arreglo. Intercambia este elemento mínimo con el primer elemento de la lista, luego, pasa al siguiente elemento y repite el proceso en el resto de la lista. Este algoritmo continúa seleccionando y posicionando los elementos hasta que toda la lista está ordenada. El mejor caso, caso promedio y peor caso tienen la misma complejidad $O(n^2)$ porque, independientemente del orden de los elementos, siempre realiza el mismo número de comparaciones. Este algoritmo es simple de entender e implementar, pero no es eficiente para listas grandes debido a su complejidad cuadrática.
   Selection Sort también cuenta con 2 bucles for anidados, el bucle externo itera n veces (1 por cada posición del arreglo) y dentro del bucle interno el algoritmo busca el mínimo recorriendo la parte desordenada. En la primera iteración, revisa n elementos, en la segunda $(n-1)$, en la siguiente $(n-2)$ y así sucesivamente. Lo que al hacer la suma total también nos da $\frac{n(n-1)}{2}$ que al aplicarle la notación Big O es $O(n^2)$.

3. **Merge Sort**
   Código: https://github.com/kamus1/tarea1-algorithms/blob/main/sorting/merge_sort.cpp

   - Mejor Caso: $O(n \log n)$
   - Caso Promedio: $O(n \log n)$
   - Peor Caso: $O(n \log n)$

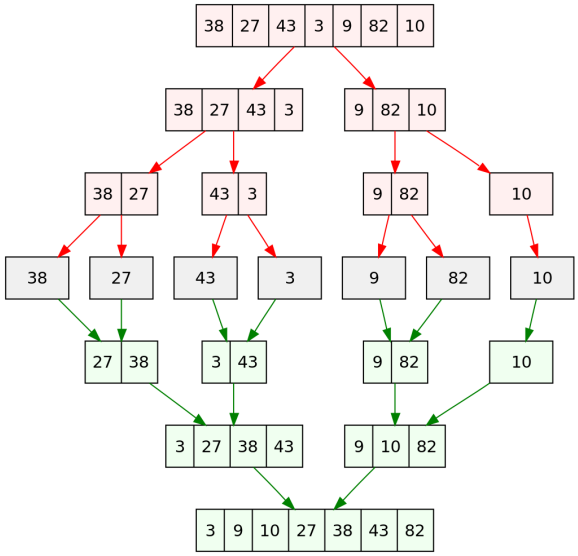


Figure 1: Merge Sort

Merge Sort es un algoritmo de ordenamiento eficiente y estable que utiliza el **enfoque de dividir y conquistar**. Divide repetidamente la lista en mitades (función merge_sort) hasta que cada sublista contiene un solo elemento. Luego, combina (función merge) las sublistas de forma ordenada para crear listas más grandes hasta que se reconstruya la lista completa ordenada. El proceso de división es recursivo, y la fusión asegura que los elementos estén en el orden correcto al combinar las sublistas. Merge Sort es consistente en su complejidad $O(n \log n)$ en los casos mejor, peor y promedio debido a que siempre divide la lista en mitades y fusiona ordenadamente, sin importar el orden inicial de los elementos. Aunque es más eficiente que los algoritmos cuadráticos como Bubble Sort o Selection Sort, requiere espacio adicional de memoria proporcional al tamaño de la lista debido a las operaciones de fusión.
Debido a que utiliza recursión podemos analizar su complejidad con el teorema maestro, donde la función recursiva sería $T(n) = 2T(\frac{n}{2}) + O(n)$, donde:

- $T(n)$: Es el Tiempo de ejecución para un arreglo de tamaño n.
- $2T\left(\frac{n}{2}\right)$: Divide el problema en dos subproblemas de tamaño $\frac{n}{2}$ y los resuelve recursivamente. Esto lo haría la función merge_sort dentro de mi código.
- $O(n)$: Es el tiempo para combinar las dos mitades, que toma tiempo lineal en el tamaño del arreglo, esto correspondería a la función merge que solo junta las mitades ordenadamente.

El Teorema Maestro se usa para resolver recurrencias de la forma:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Donde:

- a: Es el número de subproblemas en los que se divide el problema o la cantidad de llamadas recursivas (en este caso, $a = 2$).
- b: Es el factor de reducción del tamaño de la entrada (en este caso, $b = 2$).
- d: Es el exponente que describe el costo de combinar las soluciones (en este caso, $d = 1$, ya que la combinación toma tiempo $O(n)$).

Y los casos son:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Para el caso de Merge Sort tenemos que: $a = 2$, $b = 2$, $d = 1$

$a = b = 2$ Por lo tanto caemos en el primer caso, la complejidad es:

$$T(n) = O(n^d \log n) = O(n \log n)$$

Por lo tanto, el tiempo total es $O(n \log n)$, lo que muestra que Merge Sort es más rápido que algoritmos de ordenamiento como el selection sort o Bubble sort, que tienen una complejidad de $O(n^2)$.

4. **Quick Sort** (con pivot como el último elemento)
   Código: https://github.com/kamus1/tarea1-algorithms/blob/main/sorting/quicksort.cpp

   - Mejor Caso: $O(n \log n)$
   - Caso Promedio: $O(n \log n)$
   - Peor Caso: $O(n^2)$

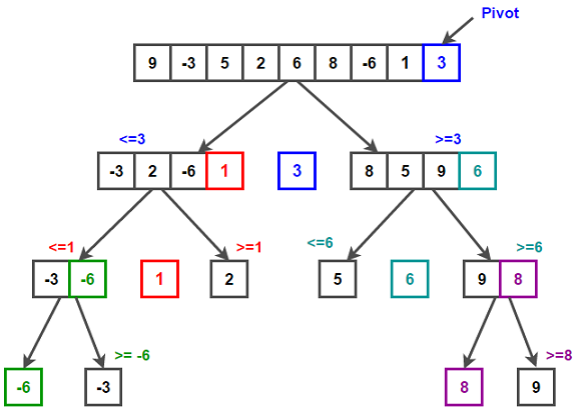


Figure 2: Quick Sort

Quick Sort es un algoritmo de ordenamiento eficiente que también utiliza el **enfoque de dividir y conquistar**. Se selecciona un pivote (para el caso de mi implementación, el último elemento de la lista) y reorganiza los elementos de tal manera que los elementos menores al pivote queden a la izquierda y los mayores a la derecha. Después, aplica el mismo proceso recursivamente a las sublistas izquierda y derecha hasta que toda la lista esté ordenada. El mejor caso ocurre cuando el pivote divide la lista en dos partes casi iguales en cada paso, lo que resulta en un rendimiento de $O(n \log n)$. El caso promedio también es $O(n \log n)$, ya que las divisiones tienden a ser razonablemente equilibradas para una lista desordenada al azar. El peor caso ocurre cuando la lista está ordenada o casi ordenada (ascendente o descendente), lo que provoca que una de las particiones esté vacía, y el algoritmo tenga que realizar $n - 1$ particiones, resultando en una complejidad de $O(n^2)$. Lo cual más adelante al ver las tablas con los tiempos de ordenamiento o los gráficos se podrá evidenciar el peor caso.

Análisis del Teorema Maestro para Quick Sort

Podemos analizar la complejidad de Quick Sort usando el Teorema Maestro. La relación de recurrencia para Quick Sort es:

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

donde:

- $T(n)$: Tiempo de ejecución para un arreglo de tamaño n.

- $T\left(\frac{n}{2}\right)$: Divide el problema en dos subproblemas de tamaño aproximadamente $\frac{n}{2}$, que ocurre en el mejor y promedio caso.
- $O(n)$: Tiempo para reorganizar los elementos en torno al pivote (partición).

El Teorema Maestro se usa para resolver recurrencias de la forma:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Donde:

- a: Número de subproblemas (para el mejor y caso promedio, asumimos $a = 2$ porque se hacen 2 llamadas recursivas al llamar a quicksort.
- b: Factor de reducción del tamaño del problema (en este caso, $b = 2$).
- d: Exponente que describe el costo de combinar las soluciones (en este caso, $d = 1$ ya que la partición toma tiempo lineal, $O(n)$).

Para Quick Sort, tenemos que: $a = 2$, $b = 2$, $d = 1$, lo que también corresponde al primer caso donde $a = b^d$

Por lo tanto:

Cuando $d = \log_b a$, la complejidad $T(n)$ es:

$$T(n) = O(n^d \log n) = O(n \log n)$$

Análisis del Peor Caso

El peor caso de Quick Sort ocurre cuando el pivote divide la lista de manera desigual, por ejemplo, cuando el arreglo ya está ordenado o inversamente ordenado, y el pivote es el elemento más grande o más pequeño. Esto resulta en la siguiente recurrencia:

$$T(n) = T(n-1) + n - 1$$

o bien:

$$T(n) = T(n-1) + O(n)$$

Sin embargo para esta recurrencia no se puede aplicar el Teorema Maestro, debido a que este solo se aplica para recurrencias del tipo $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Pero podemos podemos hacer una expansión de la función recursiva para encontrar una solución.

Expandiendo la recurrencia, obtenemos:

$$T(n) = T(n-1) + (n-1).$$

$$T(n) = T(n-2) + (n-2) + (n-1),$$

$$T(n) = T(n-3) + (n-3) + (n-2) + (n-1),$$

entonces generalizando con un reemplazo de tipo k nos queda:

$$T(n) = T(n-k) + (n-k) + ... + (n-3) + (n-2) + (n-1)$$

Y terminará cuando el arreglo se disminuye a tamaño 1, es decir $n - k = 1$ o que $k = n - 1$, si lo reemplazamos en la ecuación anterior nos queda:

$$T(n) = T(1) + 1 + 2 + 3 + ... + (n-3) + (n-2) + (n-1)$$

La cual puede ser expresada como una sumatoria donde:

$$T(n) = \frac{n(n-1)}{2}$$

Le podemos aplicar notación Big O, por lo tanto, el tiempo total se puede aproximar como:

$$T(n) = O\left(\frac{n(n-1)}{2}\right) = O(n^2).$$

Esto lleva a la complejidad $T(n) = O(n^2)$, que es la razón por la que Quick Sort es ineficiente en su peor caso. Podemos también pensar en el árbol de recurrencias, en donde en cada paso solo vamos disminuyendo en 1 el tamaño total N, hasta llegar a 1 (lo que corresponde a la sumatoria anterior), y debido a que ya está ordenando, las recursiones solo van creciendo hacia un lado del árbol.

Por lo tanto, en el mejor y caso promedio, Quick Sort tiene una complejidad de $O(n \log n)$, comparable con Merge Sort. Sin embargo, su peor caso es $O(n^2)$, lo que lo hace menos predecible en comparación con Merge Sort, que siempre es $O(n \log n)$.

5. **Función Sort de C++**
Código: https://github.com/kamus1/tarea1-algorithms/blob/main/sorting/cpp_fun_sort.cpp

   - Mejor Caso: $O(n \log n)$
   - Caso Promedio: $O(n \log n)$
   - Peor Caso: $O(n \log n)$

   La función `sort` de C++ es una implementación altamente optimizada de un algoritmo de ordenamiento híbrido que combina Quick Sort, Heap Sort, e Insertion Sort, ajustándose dinámicamente según las características de la entrada. Esta implementación se conoce como Introsort.

   El algoritmo comienza usando Quick Sort, lo que permite un rendimiento rápido en la mayoría de los casos. Si la recursión de Quick Sort se vuelve demasiado profunda (lo que indica que el pivote no está dividiendo la lista de manera efectiva), el algoritmo cambia a Heap Sort, que garantiza un rendimiento de $O(n \log n)$ en el peor caso. Para listas pequeñas, se utiliza Insertion Sort, que es más eficiente para listas casi ordenadas debido a su baja sobrecarga.

   Algoritmos que operan por debajo:

   - **Quick Sort**: Utilizado en la fase inicial del ordenamiento. Selecciona un pivote y reorganiza los elementos para que los menores queden a la izquierda y los mayores a la derecha, aplicándose recursivamente a las sublistas.
     - Mejor Caso: $O(n \log n)$
     - Caso Promedio: $O(n \log n)$
     - Peor Caso: $O(n^2)$

     Si Quick Sort detecta una recursión profunda que indica un mal pivoteo (como listas casi ordenadas), se cambia a Heap Sort.
   - **Heap Sort**: Convierte el array en un heap, luego extrae el elemento máximo y lo coloca en su posición correcta. Garantiza un tiempo de ejecución de $O(n \log n)$ incluso en el peor caso.
     - Mejor Caso: $O(n \log n)$
     - Caso Promedio: $O(n \log n)$
     - Peor Caso: $O(n \log n)$
   - **Insertion Sort**: Se utiliza cuando el tamaño de la lista es pequeño. Es eficiente para listas casi ordenadas debido a su bajo costo computacional.
     - Mejor Caso: $O(n)$ (cuando la lista ya está ordenada)
     - Caso Promedio: $O(n^2)$
     - Peor Caso: $O(n^2)$

   La combinación de estos algoritmos permite que `sort` mantenga una complejidad de $O(n \log n)$ en todos los casos, asegurando un rendimiento eficiente y estable sin importar la distribución inicial de los datos.

Algoritmos de multiplicación de matrices:

1. **Algoritmo iterativo cúbico tradicional**
   Código: https://github.com/kamus1/tarea1-algorithms/blob/main/matrix_multiplication/cubic_traditional.cpp

   - Mejor Caso: $O(n^3)$
   - Caso Promedio: $O(n^3)$
   - Peor Caso: $O(n^3)$

```cpp
vector<vector<int>> multiplicar_matrices(
    const vector<vector<int>>& aMatrix,
    const vector<vector<int>>& bMatrix,
    int filas_A,
    int columnas_A,
    int columnas_B
    ) {
    vector<vector<int>> producto(filas_A, vector<int>(columnas_B, 0));

    for (int row = 0; row < filas_A; ++row) {
        for (int col = 0; col < columnas_B; ++col) {
            for (int valor = 0; valor < columnas_A; ++valor) {
                producto[row][col] += aMatrix[row][valor] * bMatrix[valor][col];
            }
        }
    }

    return producto;
}
```

Listing 1: Algoritmo iterativo cúbico tradicional

El algoritmo iterativo cúbico tradicional multiplica dos matrices utilizando tres bucles anidados, calculando directamente el producto de las entradas. Su complejidad es siempre $O(n^3)$, sin importar el orden o los valores de las matrices.

2. **Algoritmo iterativo cúbico optimizado** (transponiendo la segunda matriz)
   Código: https://github.com/kamus1/tarea1-algorithms/blob/main/matrix_multiplication/cubic_optimizated.cpp
   Código contando el tiempo de transposición de la matriz: https://github.com/kamus1/tarea1-algorithms/blob/main/matrix_multiplication/cubic_optimizated_T.cpp

   - Mejor Caso: $O(n^3)$
   - Caso Promedio: $O(n^3)$
   - Peor Caso: $O(n^3)$

```cpp
vector<vector<int>> multiplicar_matrices_optimizado(
    const vector<vector<int>>& aMatrix,
    const vector<vector<int>>& bMatrix_T,
    int filas_A,
    int columnas_A,
    int columnas_B
    ) {

    //la matriz de producto con dimensiones n x m
    //[fila_a][columna_a] * [fila_b][columna_b] --> resultante de la multiplicacion es [fila_a][columna_b]
    vector<vector<int>> producto(filas_A, vector<int>(columnas_B, 0));

    for (int row = 0; row < filas_A; ++row) {
        for (int col = 0; col < columnas_B; ++col) {
            for (int valor = 0; valor < columnas_A; ++valor) {
                producto[row][col] += aMatrix[row][valor] * bMatrix_T[col][valor]; // solo este ultimo cambia, en el normal es [valor][col]
            }
        }
    }

    return producto;
}
```

Listing 2: Algoritmo iterativo cúbico optimizado

Este algoritmo optimiza el acceso a la memoria transponiendo la segunda matriz antes de la multiplicación. Aunque mejora la eficiencia de acceso a la memoria, su complejidad sigue siendo $O(n^3)$ en todos los casos, ya que la estructura de bucles y el número de multiplicaciones realizadas no cambian. Sin embargo en la practica este algoritmo es mas rápido debido a que mejora el uso de la jerarquía de memoria, específicamente las cachés. En el algoritmo tradicional, los accesos a las columnas de la segunda matriz son no secuenciales, lo que lleva a una mala localización espacial y mayores tiempos de acceso a la memoria. Al transponer la segunda matriz, los accesos se vuelven secuenciales, favoreciendo la carga eficiente de datos en la caché. Además, este enfoque minimiza la latencia de acceso a memoria, lo que puede ser importante en matrices grandes, sin embargo hay que tener en cuenta el tiempo para la transposición de la matriz, para mis análisis en el código cubic_optimizated.cpp a la función ya se le entrega la matriz transpuesta y los test realizados no consideran este tiempo adicional, ya que el enfoque para leer y guardar las matrices desde los .txt debería ser otro, pero para simplificar los códigos, las matrices se leen normalmente de los .txt y antes de pasarle la matriz B a la función. De todas formas a medio de comparación se considera el código cubic_optimizated_T.cpp el cual si toma en cuenta el tiempo de transponer la matriz en las mediciones.

3. **Algoritmo de Strassen**
Código Algoritmo de Strassen recursión base = 1: https://github.com/kamus1/tarea1-algorithms/blob/main/matrix_multiplication/strassen_algorithm_g4g.cpp
Código Algoritmo de Strassen considerando tamaño de hojas (o niveles de recursión): https://github.com/kamus1/tarea1-algorithms/blob/main/matrix_multiplication/strassen_algorithm.cpp
Código de Algoritmo de Strassen para matrices no cuadradas: https://github.com/kamus1/tarea1-algorithms/blob/main/matrix_multiplication/strassen_algorithm_no_square_matrix.cpp

- Mejor Caso: $O(n^{2.81})$
- Caso Promedio: $O(n^{2.81})$
- Peor Caso: $O(n^{2.81})$

El algoritmo de Strassen utiliza un enfoque de divide y vencerás para reducir la complejidad a aproximadamente $O(n^{2.81})$ en todos los casos. Este rendimiento se mantiene constante independientemente del mejor, promedio o peor caso debido a la estructura recursiva del algoritmo y la forma en que reduce las operaciones de multiplicación mediante combinaciones específicas de sumas y restas de submatrices.
Para realizar el método de Strassen se realizan los siguientes pasos:

1. División de las matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Las matrices A y B, ambas de tamaño $n \times n$, se dividen en cuatro submatrices de igual tamaño, es decir, $\frac{n}{2} \times \frac{n}{2}$. Si n no es una potencia de 2, se pueden rellenar las matrices con ceros hasta que lo sean. (sin embargo puede no resultar óptimo en la practica).

2. Cálculo de 7 productos intermedios:
Considero que este paso es uno de los importantes en el método de multiplicación de Strassen, ya que se calculan los 7 productos intermedios de las submatrices, que luego se mezclan para poder obtener la matriz resultante C.

$$P_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22}) \cdot B_{11}$$

$$P_3 = A_{11} \cdot (B_{12} - B_{22})$$

$$P_4 = A_{22} \cdot (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12}) \cdot B_{22}$$

$$P_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

3. Construcción de las submatrices de la matriz resultante C:
Usando los productos intermedios, se forman las submatrices de la matriz C.

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 - P_2 + P_3 + P_6$$

4. Combinar las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ en la matriz resultante C:
Finalmente reagrupamos las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ calculadas anteriormente utilizando los productos intermedios P_i, en la matriz C, la cual es el resultado de multiplicar $A \times B$.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

El algoritmo de Strassen se basa en el concepto de recursión para realizar las multiplicaciones de matrices en niveles sucesivos. A medida que las matrices originales A y B se dividen en submatrices más pequeñas, el mismo proceso de cálculo de los productos intermedios P_i se aplica recursivamente a estas submatrices hasta que las submatrices se reducen a un tamaño base donde la multiplicación directa es más eficiente (generalmente matrices 1×1 o matrices lo suficientemente pequeñas utilizando un tamaño de "hojas" como veremos más adelante).

La idea clave es que, en lugar de multiplicar directamente las matrices grandes, la multiplicación de submatrices más pequeñas se realiza recursivamente, y los resultados se combinan utilizando las fórmulas de Strassen para formar la matriz resultante C. Este enfoque reduce el número de multiplicaciones requeridas en comparación con la multiplicación de matrices convencional.

Análisis de Complejidad con el Teorema Maestro

El análisis de la complejidad del algoritmo de Strassen se puede realizar utilizando el Teorema Maestro, como hicimos anteriormente con los algoritmos recursivos de ordenamiento, recordando la forma de algoritmos recursivos:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

donde:

- a: Que era el número de subproblemas en los que se divide el problema original o la cantidad de llamadas recursivas.
- b: El factor por el cual se reduce el tamaño del problema en cada nivel de recursión.
- $f(n)$: Coste de combinar las soluciones de los subproblemas, ahora usamos f(n) cabe mencionar que puede existir una constante c por la cual se multiplica el costo de combinar las operaciones, aunque luego con notación Big O ya no lo tomamos en cuenta.

En el caso del algoritmo de Strassen, la recurrencia se puede expresar como:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + c \cdot n^2$$

donde:

- $a = 7$: Hay 7 productos intermedios que corresponden a los 7 subproblemas recursivos P_i.
- $b = 2$: Cada subproblema reduce el tamaño de las matrices a la mitad. $\frac{n}{2}$
- $f(n) = c \cdot n^2$: El costo de las operaciones de suma y resta de matrices (combinación de resultados) es $O(n^2)$.

Lo que nos lleva a:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

Aplicación del Teorema Maestro: Ahora, aplicamos el Teorema Maestro. Para ello, necesitamos calcular $\log_b a$ (viene de que se compara $a = b^d$ para ver en cual caso estamos):

$$\log_b a = \log_2 7 \approx 2.81$$

Comparamos $f(n) = O(n^2)$ (d=2) con $n^{\log_b a} = O(n^{2.81})$:
Los casos pero visto como logaritmos:

- Si $f(n) = O(n^{\log_b a})$ $(a = b^d)$, entonces $T(n) = O(n^{\log_b a} \log n)$.
- Si $f(n) = \Omega(n^d)$ con $d > \log_b a$, entonces $T(n) = O(f(n))$.
- Si $f(n) = O(n^d)$ con $d < \log_b a$, entonces $T(n) = O(n^{\log_b a})$.

Dado que $f(n) = O(n^2)$ y $n^2 < n^{2.81}$, estamos en el **Caso 3** Teorema Maestro (o bien lo vemos como $7 > 2^2$ $a > b^d$). del Por lo tanto, el término dominante es $n^{\log_2 7}$, lo que nos lleva a:

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

La complejidad del algoritmo de Strassen es $O(n^{2.81})$, lo que representa una mejora sobre el algoritmo de multiplicación de matrices estándar $O(n^3)$, al menos de manera teórica

Explicación de códigos realizados:
El código del algoritmo de Strassen que tengo más explicado, comentado, y el que dio mejores resultados tiene el nombre *strassen_algorithm.cpp*, la implementación de este código proviene de la pagina de Wikipedia Algoritmo de Strassen (a pesar de que el articulo lleva como título Algoritmo de Schönhage-Strassen, el código mostrado corresponde al de Strassen al igual que su explicación, el Algoritmo de Schönhage-Strassen es más complejo como podemos comprobar en la versión inglesa de la página de Wikipedia Algoritmo de Schönhage-Strassen), en este código se utiliza el concepto de "Hojas" o leafs, que significa que a partir de un cierto tamaño de las matrices, en vez de utilizar el algoritmo de Strassen se utilizará el algoritmo cubico tradicional (algoritmo ikj que según el orden de los iteradores es la mejor) para multiplicar las matrices. Por ejemplo si mi tamaño de hoja (leafSize) es igual a 16, las matrices de tamaño 16 o menos serán multiplicadas por el algoritmo ikj. Esto debido que las llamadas recursivas hasta el mínimo nivel genera un overhead excesivo en computadores modernas además de que las operaciones a nivel de caché y la optimización de los compiladores modernos favorecen el uso de algoritmos iterativos para tamaños pequeños de matrices. También se implementa en este código que cuanto el tamaño de hojas sea igual a 1, se realiza un multiplicación normal del único elemento que hay en ambas matrices, la multiplicación es del tipo $C[0][0] += A[0][0] \times B[0][0]$, que sería el caso base del algoritmo de Strassen.

El uso del algoritmo cúbico tradicional (ikj) en matrices pequeñas es eficiente debido a la localidad de datos, lo cual reduce el número de "cache misses" y mejora el rendimiento. A medida que las matrices se hacen más grandes, el algoritmo de Strassen toma el control, aprovechando su menor complejidad asintótica para reducir el número de multiplicaciones requeridas y, por ende, acelerar el cálculo global. Este enfoque permite combinar las ventajas de ambos métodos: el algoritmo cúbico para matrices pequeñas y Strassen para matrices más grandes.

En mi implementación del código hay una variable global llamada "leafsize" en la cual se puede modificar el tamaño de las hojas, para los test realizados compare el redimiendo del algoritmo de Strassen para hojas de tamaño 1×1 (caso base),16×16, 32×32, 64×64 y 128×128, para probar el comportamiento del algoritmo en mi computador. Sin embargo, este valor puede variar según la arquitectura del procesador, el tamaño de la caché y otros factores específicos del hardware. Según investigué ajustar el tamaño de la hoja es una técnica común en implementaciones de multiplicación de matrices que buscan obtener un balance óptimo entre las llamadas recursivas y la eficiencia de la multiplicación directa en pequeñas dimensiones.

Pero a forma de comparación tengo otro código de nombre *strassen_algorithm_g4g.cpp* este archivo contiene la implementación del código de la página de GeeksForGeeks donde muestran como implementar el algoritmo de Strassen, este es el código de Strassen con recursión nivel 1, que es el caso base cuando las matrices tienen un único elemento y se hace la multiplicacón $C[0][0] += A[0][0] \times B[0][0]$, este código llega hasta el mínimo nivel de las submatrices, lo cual

genera un overhead excesivo y no tan buen uso de la memoria Caché. Cabe mencionar que este código no está tan comentado como el anterior, ya que me decidí en mejorar el código de *strassen_algorithm.cpp* y explicarlo de mejor manera, sin embargo lo único en que varían es el caso en donde parar de usar el algoritmo de Strassen, en donde a *strassen_algorithm.cpp* se le puede indicar el tamaño de la hoja ("leaf" una variable global) a usar, si en este código leaf = 1, va a realizar el mismo procedimiento que *strassen_algorithm_g4g.cpp*, sin embargo este ultimo no está tan optimizado en algunas cosas del código como llamados a referencia en las funciones, entre otras cosas que lo ralentizan. Como se podrá ver más adelante en los resultados experimentales, aunque ambos lleguen a recursión nivel 1, tiene mejor redimiendo *strassen_algorithm.cpp* en S_1.

Y el otro código que tengo de nombre **strassen_algorithm_no_square_matrix.cpp** es el **strassen_algorithm.cpp** pero modificado para que pueda multiplicar matrices que no sean cuadradas, sin embargo al hacer esto, el redimiendo del algoritmo de Strassen disminuye, por esta razón es un código aparte, además para no mezclar las cosas, pero era necesario para poder hacer las comparaciones con matrices que no sean cuadradas.

3 Descripción de los datasets

En este trabajo se han generado varios conjuntos de datos con características específicas, cada uno diseñado para evaluar el rendimiento de algoritmos en diferentes escenarios. A continuación, se describen los datasets utilizados:

Datasets para algoritmos de ordenamiento
El archivo para generar los dataset se encuentra en */sorting/datasets/generate_datasets.cpp* , este código genera los siguientes archivos .txt, con números separados por espacios, con un rango [0, 600,000] (para cada número individual).

- **Dataset de números aleatorios (*random_dataset.txt*):** Este dataset contiene números enteros generados de manera completamente aleatoria. El propósito de este conjunto es simular un escenario sin ningún tipo de orden.
- **Dataset semi ordenado (*semisorted_dataset.txt*):** En este caso, se generan números aleatorios, pero la primera mitad (50%) están ordenados en orden ascendente, mientras que la segunda mitad es completamente aleatoria. Este dataset simula un escenario parcialmente ordenado desde el inicio.
- **Dataset completamente ordenado (*sorted_random_dataset.txt*):** Este dataset se compone de números generados aleatoriamente, pero ordenados en su totalidad de menor a mayor. Sirve para evaluar el comportamiento del algoritmo con datos completamente ordenados, por ejemplo para probar el peor caso de Quick Sort.
- **Dataset parcialmente ordenado (*partially_sorted_dataset.txt*):** La forma como consideré este dataset es la siguiente: contiene números aleatorios, donde el 70% (aunque este porcentaje es modificable en el código de generación de datasets) de los números ubicados alrededor del centro del conjunto de números están ordenados, mientras que el resto sigue siendo aleatorio. Este dataset permite evaluar algoritmos en situaciones en las que los datos están parcialmente ordenados en zonas específicas.

- **Dataset de números únicos (*unique_random_dataset.txt*):** Este conjunto contiene números únicos generados aleatoriamente. Para asegurar la unicidad, los números se seleccionan de un rango de 600,000 posibles valores y luego se mezclan para evitar que estén ordenados. Este dataset es útil para probar algoritmos en escenarios donde no existen duplicados en los datos, y para comprobar experimentalmente como se comportan los algoritmos de ordenamiento ante esta situación.

  La longitud de cada uno de estos datasets (tamaño de N), fueron probado con valores entre [10, 25.000]. Sin embargo, para los algoritmos de ordenamiento más rápido fueron probado 2 tamaños adicionales 50.000 y 100.000. Pero esto se verá de manera mas detallada en la sección de resultados.

Datasets para algoritmos de multiplicación de matrices El archivo para generar dataset se encuentra en */matrix_multiplication/datasets/generate_matrix.cpp*, este código se encarga de generar 2 matrices, en el cual se les puede indicar el tamaño $N \times M$ de cada matriz, así como el rango de los números en el cual se encuentra cada número individual de la matriz, los test realizados para este caso fueron con números de rango [1,1000], los archivos que genera son .txt donde los números están separados por espacios, y cada linea representa una fila de la matriz, los .txt son los siguientes:

- **Dataset de la matriz 1 (*matriz_1.txt*):** Corresponde a los datos generados de la matriz 1 de tamaño $N \times M$.
- **Dataset de la matriz 2 (*matriz_2.txt*):** Corresponde a los datos generados de la matriz 2 de tamaño $N \times M$.

Los tamaños probados para las matrices cuadradas van entre un rango de [4,6000] para el tamaño de N (la matriz es $N \times N$), pero también hay otro rango de intervalos mas detallado entre los valores [4, 1000] ya que quería probar como era el comportamiento en este rango de matrices pequeñas de manera más detallada, pero esto se explicará mas adelante en la sección de resultados. Tipo de multiplicación es $[N][M] \times [N][N]$

Para las matrices no cuadradas de tipo $N \times M$ fueron probadas con un rango de valores de N ente [1,3000], donde M es la mitad de N, $M = \frac{N}{2}$, y la multiplicación de las matrices es del tipo $[N][M] \times [M][M]$, de esta manera la matriz resultante no es cuadrada y es de tipo $[N][M]$.

Archivos usados para la exportación de los datos luego de aplicar los algoritmos
Algoritmos de ordenamiento:
Estos resultados contienen los números ordenados separados por espacios, usando cada uno de los datasets. Dentro de estos .txt se indica el tiempo de ejecuación, así como el nombre de cual dataset provienen. Ejemplo.

- s_bubble_sort.txt: Datos ordenados (sorted) luego de usar Bubble Sort.
- s_selection_sort.txt: Datos ordenados luego de usar Selection Sort.
- s_merge_sort.txt: Datos ordenados luego de usar Merge Sort.
- s_quick_sort.txt: Datos ordenados luego de usar Quick Sort.
- s_cpp_fun_sort.txt: Datos ordenados luego de usar la función Sort de la librería estándar de C++.

Algoritmos de Multiplicación de Matrices:

- res_cubic_traditional.txt: Resultado de la multiplicación de matrices del algoritmo tradicional cúbico.
- res_cubic_op.txt: Resultado de la multiplicación de matrices del algoritmo cúbico optimizado, sin considerar el tiempo de transponer la matriz B.
- res_cubic_op_T.txt: Resultado de la multiplicación de matrices del algoritmo cúbico optimizado, considerando el tiempo de transponer la matriz B.
- res_strassen_g4g.txt: Resultado de la multiplicación de matrices del algoritmo de Strassen.
- res_strassen.txt: Resultado de la multiplicación de matrices del algoritmo de Strassen que usa hojas.
- res_strassen_no_square.txt: Resultado de la multiplicación de matrices del algoritmo de Strassen adaptado a multiplicar matrices no cuadradas.

4 Resultados experimentales

Resultados de los algoritmos de ordenamiento
Para los algoritmos de ordenamiento, primero realicé una comparación individual de cada uno para ver como se comportaban y variaban sus tiempos respecto a los distintos datasets. Luego se presentarán gráficos para una comparación general entre los algoritmos probados.
Para la parte de los algoritmos individuales se tiene que los $Time_i$ son el tiempo medido en milisegundos para los distintos datasets probados, donde:

Time1 = Dataset de números aleatorios.
Time2 = Dataset parcialmente ordenado.
Time3 = Dataset semiordenado.
Time4 = Dataset de números únicos.
Time5 = Dataset completamente ordenado.

Y para los gráficos estos tiempos se presentan en una forma abreviada como T_i (Ejemplo: Time1 = T1, Time2=T2,...etc).

Table 1: Tiempos de Ejecución de Bubble Sort

N	**Time1 [ms]**	**Time2 [ms]**	**Time3 [ms]**	**Time4 [ms]**	**Time5 [ms]**
10	0,000671	0,000601	0,000571	0,000611	0,00033
50	0,009718	0,006712	0,008967	0,024115	0,003828
100	0,038622	0,027602	0,035155	0,040035	0,014447
200	0,204072	0,096942	0,127809	0,14992	0,055915
500	0,951117	0,611162	0,782331	0,977746	0,34288
1000	4,19906	2,48174	3,3689	4,02408	1,41433
5000	106,711	66,2952	87,3684	106,21	34,4856
10000	429,06	274,499	356,943	427,534	138,373
15000	964,318	615,131	848,535	983,738	311,592
20000	1737,9	1099,27	1446,62	1727,2	651,252
25000	2697,09	1689,77	2251,54	2697,84	861,785

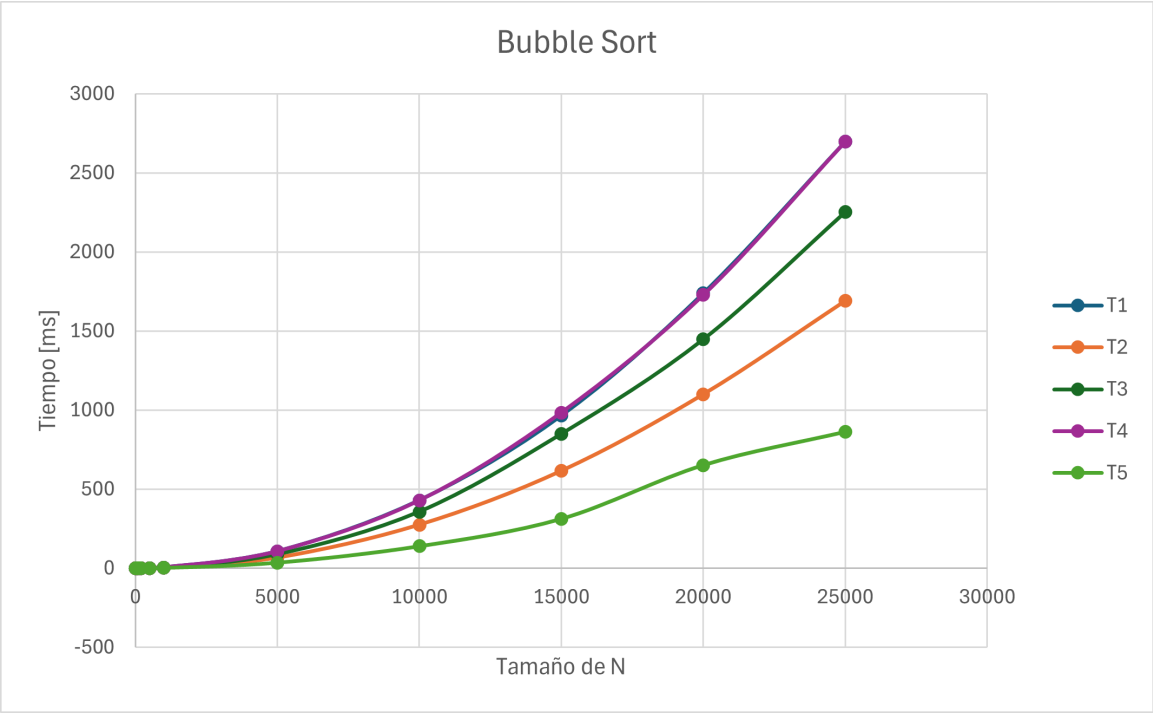


Figure 3: Bubble Sort

Para Bubble Sort podemos notar que mientras más ordenado esté el arreglo o listado de números, mejor rendimiento tiene, esto se debe principalmente a que debe hacer menos swaps de números cuando esté ya están ordenados en su posición, y debe dar menos vueltas para ordenar el listado, y como podemos ver en el gráfico, el mejor tiempo se da en **T5** que es cuando el listado de números ya está ordenado.

Table 2: Tiempos de Ejecución de Selection Sort

N	Time1 [ms]	Time2 [ms]	Time3 [ms]	Time4 [ms]	Time5 [ms]
10	0,000782	0,000671	0,000582	0,000792	0,000491
50	0,007283	0,006752	0,007153	0,007013	0,005841
100	0,024175	0,022913	0,023865	0,023925	0,020889
200	0,086561	0,08522	0,086242	0,086531	0,178654
500	0,59916	0,501857	0,501116	0,508079	0,520453
1000	2,01239	1,94965	1,98853	1,96336	1,9391
5000	48,8976	48,6369	48,8536	49,096	48,5924
10000	191,695	191,676	191,832	193,094	190,923
15000	434,006	434,645	435,751	439,259	439,17
20000	823,119	795,704	777,302	773,528	775,51
25000	1219,16	1202,6	1211,54	1211,78	1197,00

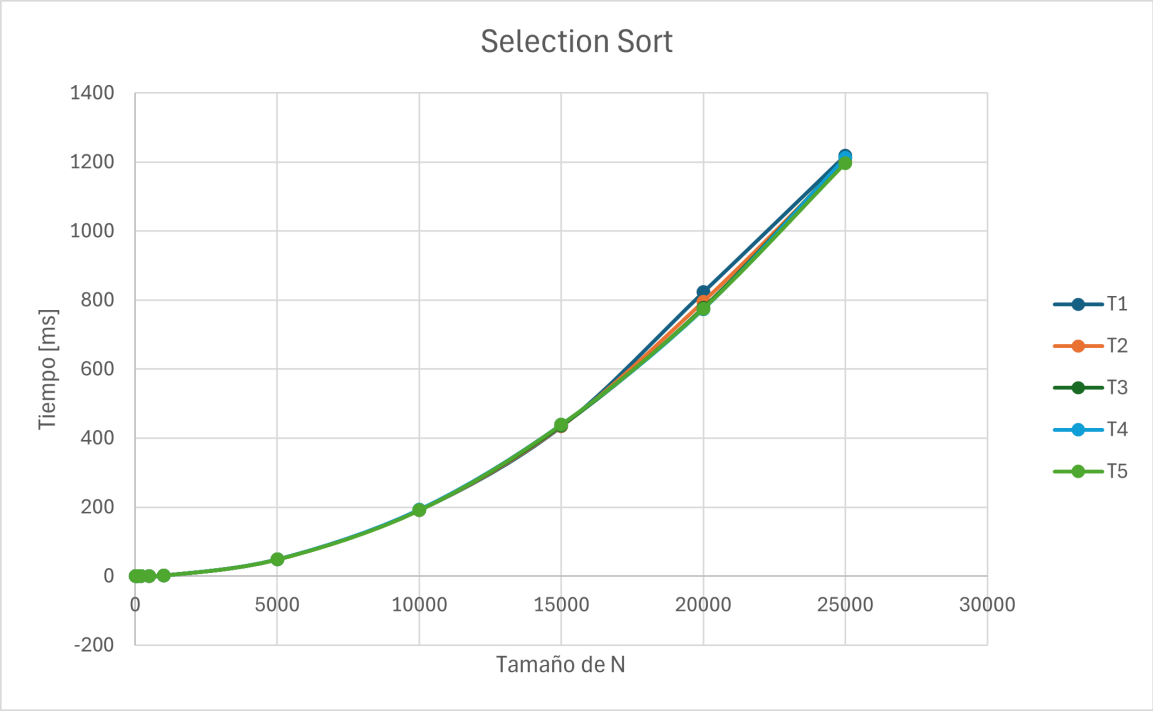


Figure 4: Selection Sort

Para Selection Sort los tiempos de ejecución entre los distintos datasets son muy similares, esto debido a que el algoritmo Selection Sort no se ve tan afectado según el ordenen de los números ya que se ejecuta de la misma manera.

Table 3: Tiempos de Ejecución de Merge Sort

N	Time1 [ms]	Time2 [ms]	Time3 [ms]	Time4 [ms]	Time5 [ms]
10	0,003877	0,003006	0,003427	0,003396	0,003186
50	0,014207	0,013075	0,012894	0,013495	0,011972
100	0,033522	0,059221	0,05885	0,061214	0,054872
200	0,05927	0,054091	0,055073	0,057948	0,051166
500	0,153387	0,136365	0,156562	0,149559	0,12833
1000	0,364601	0,290693	0,290112	0,317834	0,263302
5000	1,72665	1,55628	1,59059	1,75859	1,42702
10000	3,6247	3,22131	3,41977	3,80597	3,08867
15000	5,53867	4,82157	5,05117	5,56796	4,46729
20000	7,51931	6,68451	6,87233	7,55492	6,12969
25000	9,63574	8,49268	8,91572	9,5998	7,9637
50000	20,6834	17,185	18,0369	19,6859	16,4243
100000	41,1181	35,798	38,0591	41,5428	33,8636

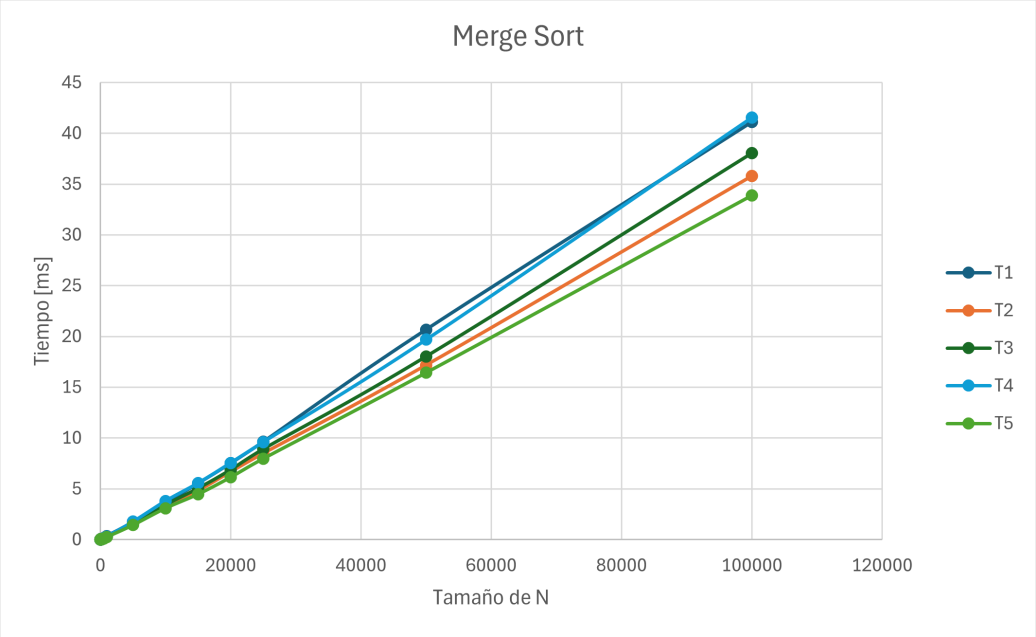


Figure 5: Merge Sort

En el gráfico podemos notar que merge sort es más rápido con datos ordenados (T5), aunque en teoría siempre realiza el mismo número de divisiones recursivas, lo que puede ocurrir es que en la fase de combinación (merge) se vuelva más eficiente. Al combinar subarreglos ya ordenados, se hacen menos comparaciones y movimientos, aprovechando mejor la memoria y el caché de la CPU, lo que puede reducir el tiempo de ejecución. Sin embargo, es mucho mas rápido que los algoritmos cuadráticos como Bubble Sort y Selection Sort.

Table 4: Tiempos de Ejecución de Quick Sort

N	**Time1 [ms]**	**Time2 [ms]**	**Time3 [ms]**	**Time4 [ms]**	**Time5 [ms]**
10	0,000481	0,000551	0,000521	0,000511	0,000731
50	0,003316	0,002705	0,002965	0,003126	0,011462
100	0,007514	0,006662	0,007343	0,007043	0,041237
200	0,014848	0,016561	0,014327	0,015218	0,161832
500	0,047409	0,04292	0,04302	0,044954	0,908407
1000	0,100879	0,101189	0,099395	0,102942	2,79295
5000	0,645218	0,736769	0,590916	0,617616	102,21
10000	1,39298	1,36401	1,35924	1,39287	402,243
15000	2,20708	2,16471	2,40776	2,11788	895,969
20000	2,99435	3,8931	3,02733	3,06652	1467,88
25000	3,74176	4,51369	3,7521	3,95561	2175,52
50000	7,99522	8,51695	8,29666	8,05895	10751,6
100000	17,4721	17,7807	17,33	17,271	40350,4

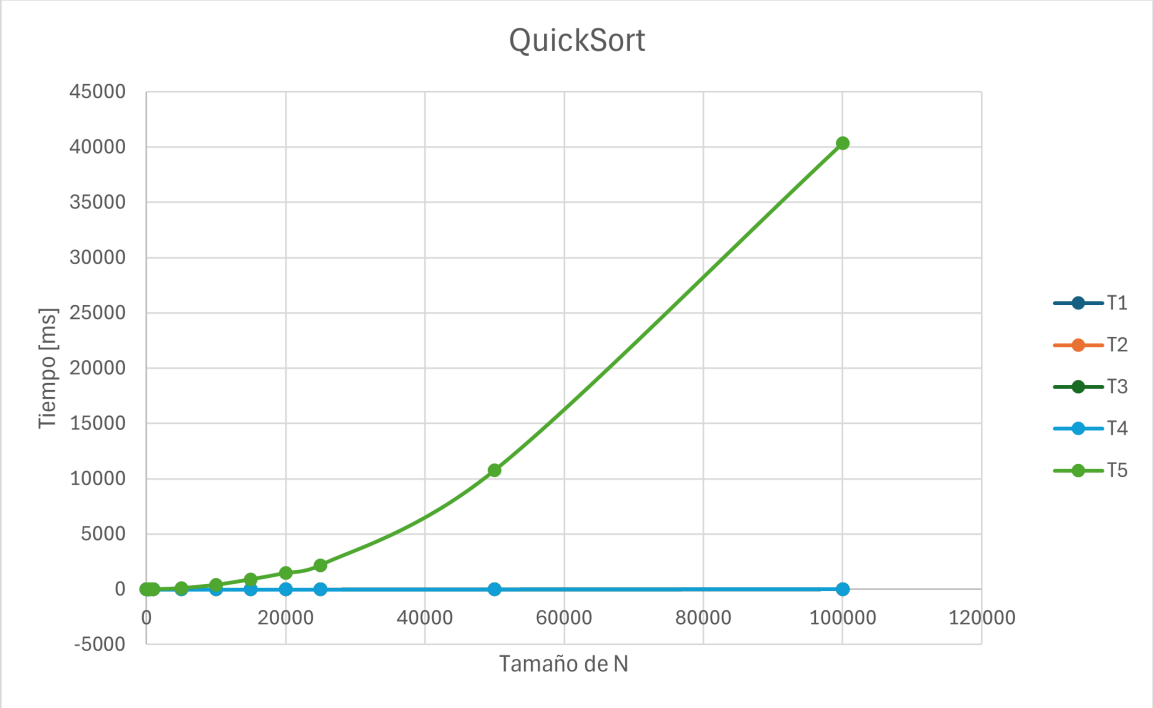


Figure 6: Quick Sort

En este gráfico de las comparaciones de los tiempos de ejecución podemos ver claramente que en el peor caso de Quick Sort cuando los datos están ordenados (**T5**), el tiempo de ejecución se vuelve cuadrático ($O(n^2)$), donde ni si quiera se pueden ver los demás tiempos de ejecución. A continuación se mostrará un gráfico donde no se considera a T5.

Gráfico de los tiempos de ejecuón de Quick Sort, sin considerar a Time5.

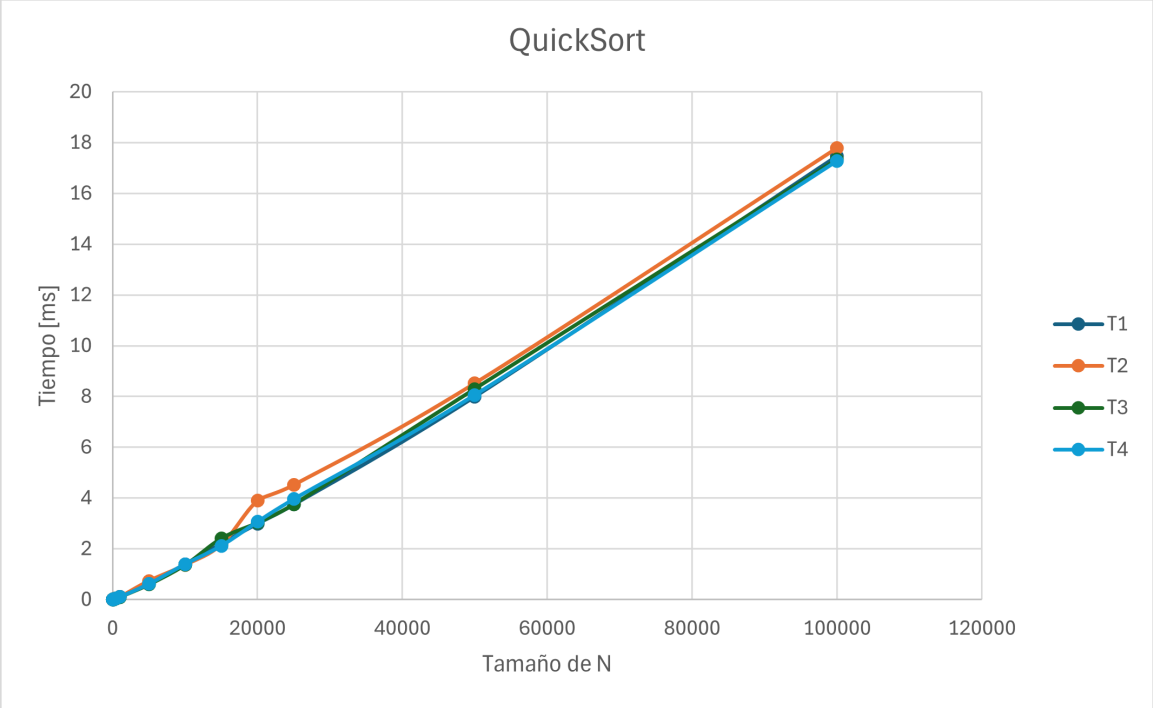


Figure 7: Quick Sort

Ahora sin considerar a T5 se pueden ver de mejor manera los otros casos de Quick Sort, donde las variaciones de tiempo son muy similares, si somos un poco más detallados podemos observar que entre estos el que más se demora es T2, que corresponde al dataset de los datos semi ordenados, el cual en mi caso contiene el 70% de los datos ordenados alrededor del centro del listado de números.

Table 5: Tiempos de Ejecución de la Función Sort de C++

N	Time1 [ms]	Time2 [ms]	Time3 [ms]	Time4 [ms]	Time5 [ms]
10	0,001062	0,000881	0,000721	0,000911	0,000511
50	0,00507	0,004488	0,004298	0,00531	0,003036
100	0,009839	0,008867	0,009287	0,010018	0,005911
200	0,020949	0,018765	0,020519	0,020428	0,011942
500	0,130995	0,116548	0,072555	0,059391	0,030748
1000	0,125093	0,107711	0,121397	0,124994	0,066584
5000	0,759741	0,653572	0,753018	0,760302	0,506387
10000	1,63975	1,38038	1,58921	1,62993	0,971154
15000	2,53284	2,21571	2,40597	2,55664	1,3667
20000	3,54702	3,03348	3,3929	3,52203	2,01157
25000	4,54215	3,82793	4,20089	4,54992	2,4815
50000	9,64372	8,27607	9,01487	9,64053	5,35914
100000	20,5372	17,1827	18,942	20,2468	11,8378

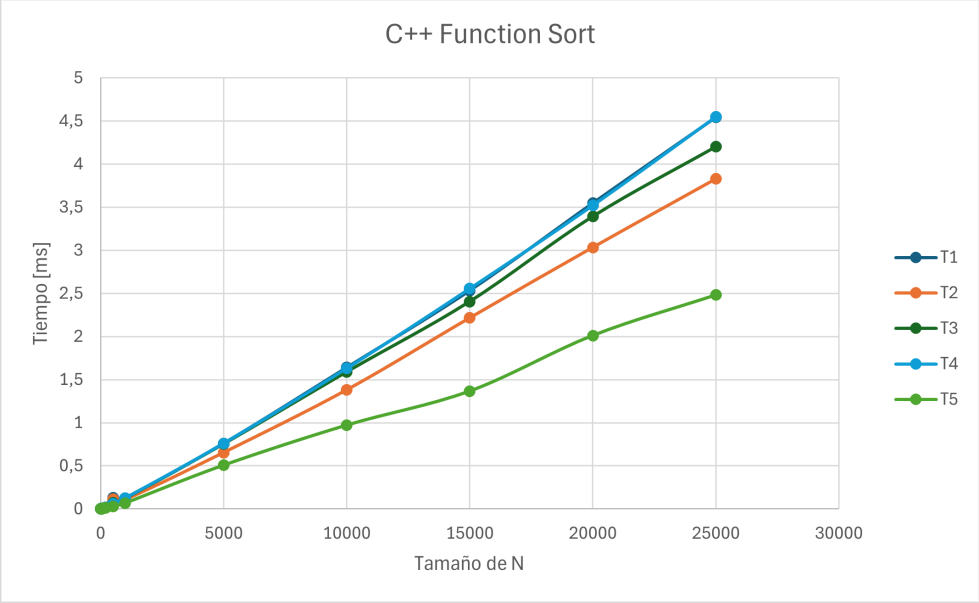


Figure 8: C++ Function Sort

Este gráfico correspondiente a los tiempos de ejecución de la función Sort de la librería estándar de C++, podemos ver que a diferencia del Quick Sort implementado, los mejores casos se dan cuando el listado ya está ordenado (T5) y semi ordenado (T2), esto se debe principalmente a la implementación híbrida de de la función Sort de C++, la cual puede detectar estos casos y usar otro tipo de algoritmos de ordenamiento cuando el listado ya está ordenado.

Comparación general, usando Time1 = Random Dataset.

Table 6: Comparación General (Random dataset)

N	BubbleSort [ms]	SelectionSort [ms]	MergeSort [ms]	QuickSort [ms]	C++ FunSort [ms]
10	0,000671	0,000782	0,003877	0,000481	0,001062
50	0,009718	0,007283	0,014207	0,003316	0,00507
100	0,038622	0,024175	0,033522	0,007514	0,009839
200	0,204072	0,086561	0,05927	0,014848	0,020949
500	0,951117	0,59916	0,153387	0,047409	0,130995
1000	4,19906	2,01239	0,364601	0,100879	0,125093
5000	106,711	48,8976	1,72665	0,645218	0,759741
10000	429,06	191,695	3,6247	1,39298	1,63975
15000	964,318	434,006	5,53867	2,20708	2,53284
20000	1737,9	823,119	7,51931	2,99435	3,54702
25000	2697,09	1219,16	9,63574	3,74176	4,54215

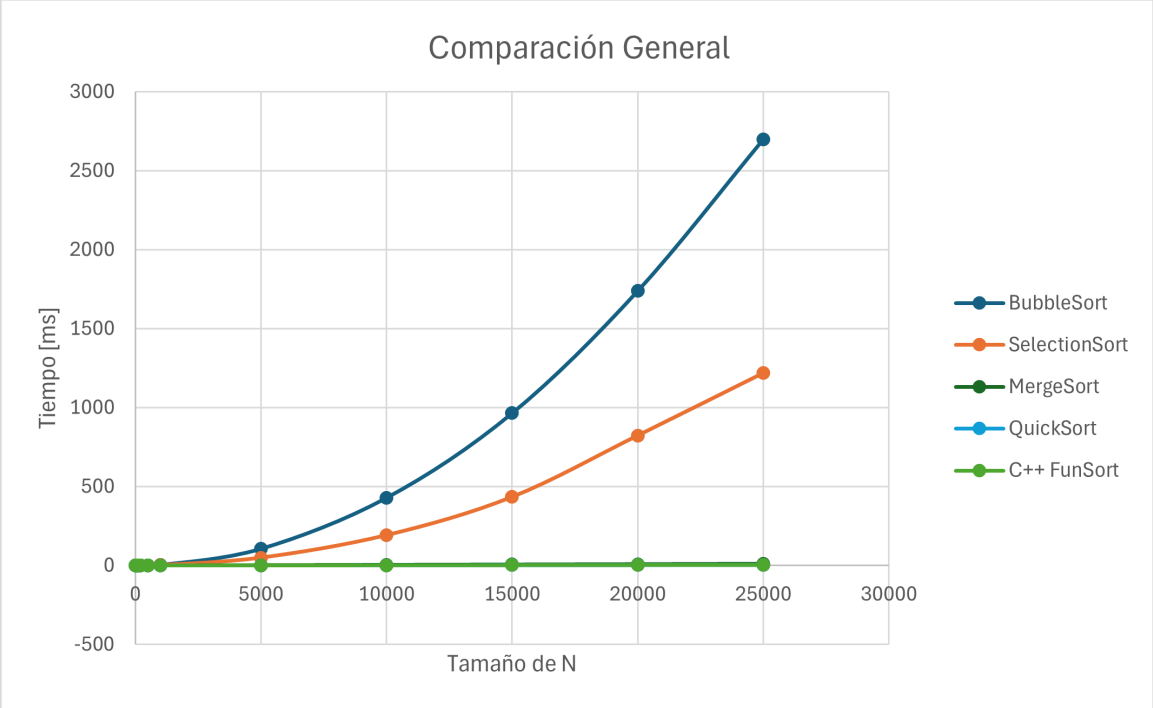
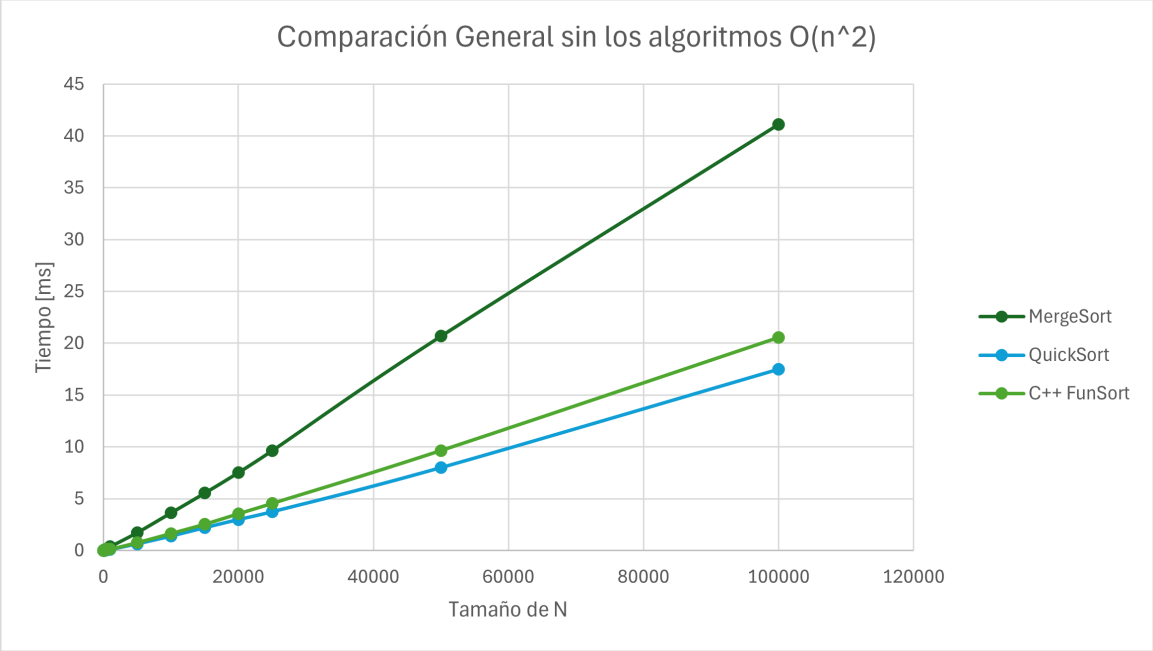


Figure 9: General Sort

Bubble Sort y Selection Sort son los que más tiempo se demoran debido a su complejidad algorítmica de $O(N^2)$, lo que impide ver la comparación de los algoritmos más rápidos (o los que tienen una complejidad menor). A continuación se mostrará un gráfico de comparaciones entre los algoritmos sin contar a Bubble Sort y Selection Sort:

Gráfico de Comparación sin considerara a los algoritmos cuadráticos:



En este gráfico podemos ver que la función de Quick Sort implementada es más rápida que la función Sort de la librería estándar de C++, y luego le sigue Merge Sort. La implementación de Quick Sort podría ser más rápida que la función Sort de C++ debido a varios factores. Uno de ellos podría ser debido a que la implementación de Quick Sort está mejor adaptada a las características específicas de los datos (que estén desordenaos), mientras que Sort de C++ está optimizada para los casos generales, como también para cuando están ordenados. Además, Sort de C++ realiza diversas optimizaciones internas que pueden añadir complejidad y, en algunos casos, ralentizar el proceso. también es posible que incluya verificaciones adicionales, lo que podría impactar en su rendimiento. Sin embargo Sort de C++ está diseñado para ser eficiente en una variedad de escenarios, mi implementación personalizada de Quick Sort puede superar su rendimiento en contextos específicos. Ya que como vimos anteriormente el peor caso de Quick Sort es cuando el arreglo está ordenado, lo que no pasa con la función Sort de C++, donde en ese caso, es la que menos tiempo demora.
La razón de que Merge Sort es más lento, a pesar de que tiene la misma complejidad que QuickSort en el caso promedio $O(nlog(n))$ (aunque Merge Sort es $O(nlog(n))$ en todos sus casos), se puede deber principalmente a que QuickSort es un algoritmo de tipo In-place, es decir que utiliza una cantidad constante de espacio, mientras que Merge Sort no lo es, sin embargo Merge Sort si es un algoritmo estable.

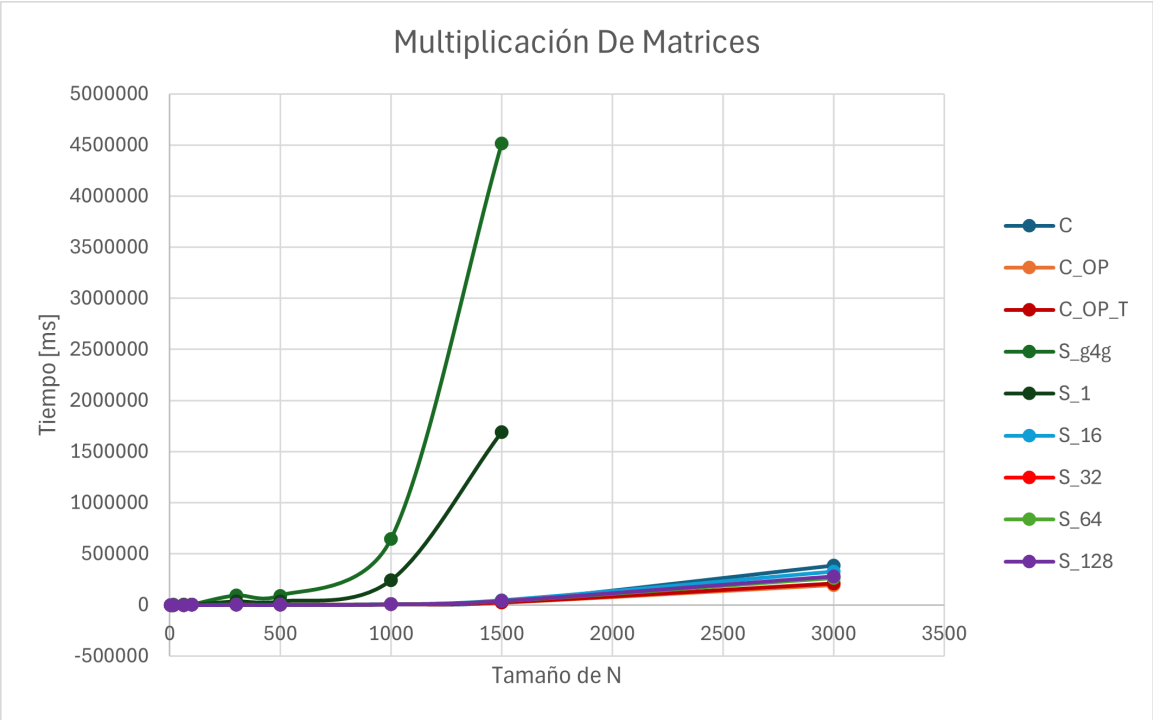
Resultados de los algoritmos de Multiplicación
Matrices cuadradas de tipo $[N][N] \times [N][N]$
Primero explicaré que significa cada nombre de las tablas:

N: Corresponde al tamaño de la fila de la matriz (o columna ya que son matrices cuadradas) que se usó en cada caso.
C: Corresponde al algoritmo de multiplicación Cúbico tradicional.
C_OP: Corresponde al algoritmo Cúbico Optimizado, pero sin considerar el tiempo de transponer la matriz B.
C_OP_T: Corresponde al algoritmo Cúbico Optimizado, pero ahora SI considerando el tiempo de transponer la matriz B.
S_g4g: Corresponde al algoritmo de Strassen para multiplicar matrices full recursivo (base = 1) de GeeksForGeeks.
S_1: Corresponde al algoritmo de Strassen pero con hoja de tamaño 1, es decir que se multiplica el único elemento de la matriz A, por el único elemento de la matriz B, este caso fue realizado para la comparación a nivel de recursión con S_g4g.
S_16: Algoritmo de multiplicación de matrices de Strassen con hoja de tamaño 16.
S_32 Algoritmo de multiplicación de matrices de Strassen con hoja de tamaño 32.
S_64 Algoritmo de multiplicación de matrices de Strassen con hoja de tamaño 64.
S_128 Algoritmo de multiplicación de matrices de Strassen con hoja de tamaño 128.

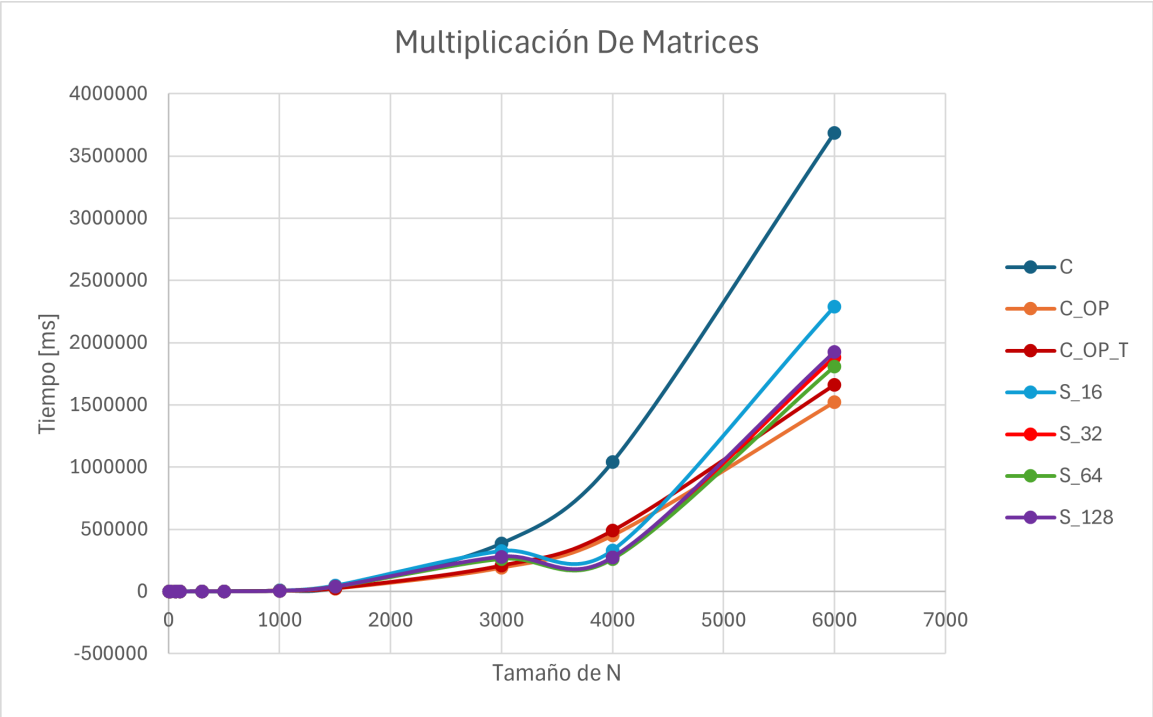
TL: Significa Time Limit, este se usa en **S_g4g** y **S_1**, ya que luego se empezó a demorar demasiado, fuera de tiempos razonables para las comparaciones, en el peor caso medido fue para **S_g4g** con **N** de 1500, en el cual se demoró 4,51E+06 [ms] que son 75,17 [min], lo que es mas de 1 hora esperando a que el algoritmo pare de ejecutarse.

Table 7: Comparación de Tiempos de Ejecución para Diferentes N

N	C [ms]	C_OP [ms]	C_OP_T [ms]	S_g4g [ms]	S_1 [ms]	S_16 [ms]	S_32 [ms]	S_64 [ms]	S_128 [ms]
4	0,001883	0,001462	0,002906	0,115155	0,05856	0,011682	0,018004	0,014147	0,014978
16	0,035106	0,032541	0,039413	5,46161	2,223	0,092774	0,073547	0,055053	0,054662
64	2,00689	1,83500	2,061420	258,725	102,596	2,65843	2,18075	2,04611	1,99967
100	7,19115	6,97748	7,74282	1834,24	695,064	18,9241	15,926	17,2736	15,4694
300	193,364	188,518	205,806	91096,7	34333,4	989,9520	772,0690	745,0320	788,9970
500	924,465	889,741	961,213	89912,2	34207,4	936,346	816,372	752,003	793,458
1000	8604,44	6967,89	7653,48	643429	241789	6598,59	5501,37	5339,94	5534,45
1500	29415,4	23563,4	25733,6	4,51E+06	1,69E+06	46558	38384,8	37886,3	38598,9
3000	386932	191337	208354	TL	TL	324903	271555	261410	279468
4000	1,04E+06	450360	488193	TL	TL	329733	271537	259976	273693
6000	3,68E+06	1,52E+06	1,66E+06	TL	TL	2,29E+06	1,88E+06	1,81E+06	1,92E+06



Como podemos notar **S_g4g** y **S_1** tienen los peores tiempos de ejecución, se demoran mucho más que los demás, los cuales no se alcanzan a diferenciar bien, el peor tiempo de ejecución lo tiene **S_g4g** esto debido principalmente a que usa recursión hasta matriz base = 1, lo que provoca un overhead excesivo, y además sobrecarga la memoria Caché de la CPU para este tipo de operaciones. Luego le sigue en tiempo S_1, que al llegar al nivel de hoja 1 cuando la matriz solo tiene un único elemento y se realiza una multiplicación de estos dos elementos del tipo $C[0][0] += A[0][0] \times B[0][0]$. si bien esto es mejor, no logra usar de mejor forma el método de Strassen para números grandes. La razón de por qué S_1 es mejor que S_g4g, se debe a la forma en la cual está estructurado el código, y el uso de llamas por referencia en las funciones. A continuación se mostrará el gráfico pero sin considerar a **S_g4g** y **S_1**.

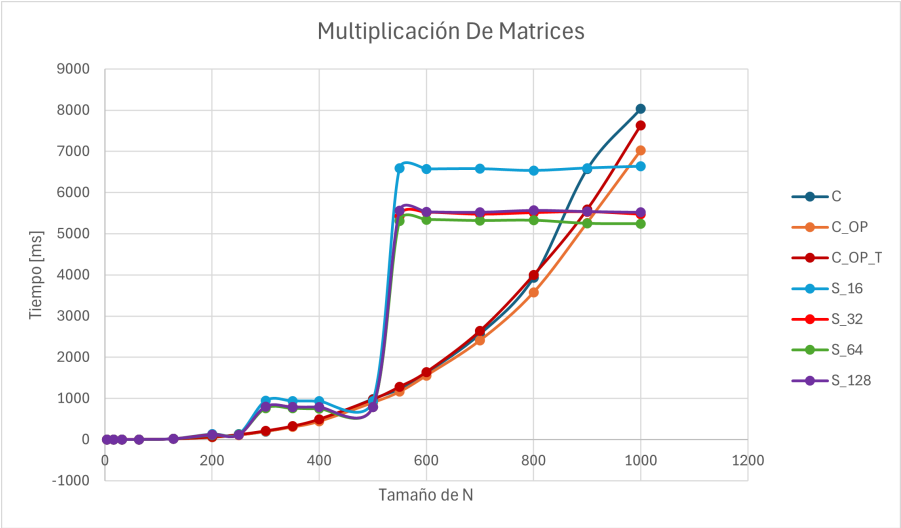


Los análisis que podemos hacer a partir de este gráfico que es que el método de Strassen con distintos tamaños de hojas, mejora para números grandes en comparación al Cúbico tradicional C, en donde los mejores tiempos de Strassen son para hojas de nivel 32 y 64, aunque también es muy similar con tamaño de hoja de 128. Sin embargo, si comparamos el método de Strassen con hojas con el algoritmo Cúbico Optimizado, solo en algunos rangos de números logra ser mejor que **C_OP**. También si comparamos **C_OP** con **C_OP_T** el cual considera el tiempo de transponer la matriz B (o matriz 2), no varían demasiado para números pequeños, sin embargo para números más grandes ya se comienza a ver una diferencia entre los tiempos, obviamente con **C_OP_T** demorándose un poco más, debido a que considera el tiempo de transponer la matriz B.

Se realizó una comparación más exhaustiva para los métodos de multiplicación de matrices para un rango mas reducido, entre un N de tamaño 4 y 1000, solamente para ver como se comportaba el método de Strassen con distintos niveles de hoja en este intervalo de N más bajo, y para compararlo con los algoritmos cúbico tradicional y el cúbico optimizado.

Table 8: Comparación de Tiempos de Ejecución para Diferentes N 1000

N	C [ms]	C_OP [ms]	C_OP_T [ms]	S_16 [ms]	S_32 [ms]	S_64 [ms]	S_128 [ms]
4	0,001894	0,001513	0,003236	0,015419	0,014417	0,014177	0,028243
16	0,034575	0,032461	0,039304	0,051877	0,055473	0,056576	0,056545
32	0,243475	0,396773	0,272590	0,372598	0,287438	0,308327	0,286086
64	2,41572	1,96664	2,07537	2,62697	2,56172	2,00353	2,00538
128	15,1345	14,6497	16,2329	20,2299	16,3668	16,4723	15,3963
200	57,4807	55,6317	64,6743	132,829	119,561	105,096	111,481
250	112,101	108,491	120,417	134,763	111,53	124,237	112,372
300	196,479	198,761	210,591	943,631	783,076	762,717	799,738
350	313,363	304,387	326,507	938,796	791,563	759,634	787,107
400	459,88	443,272	495,582	935,012	772,555	749,975	788,887
500	985,183	897,263	975,316	936,239	790,151	789,244	788,052
550	1233,77	1162,12	1283,96	6586,29	5420,69	5308,58	5562,17
600	1622,39	1555,28	1641,78	6571,4	5515,86	5340,27	5530,65
700	2564,25	2408,22	2641,53	6576,06	5471,52	5317,12	5520,34
800	3929,97	3574,09	3996,76	6533,34	5509,52	5323,89	5566,23
900	6566,77	5264,71	5590,71	6594,17	5533,43	5248,42	5538,13
1000	8036,87	7019,97	7634,87	6635,7	5470,19	5241,89	5518,54

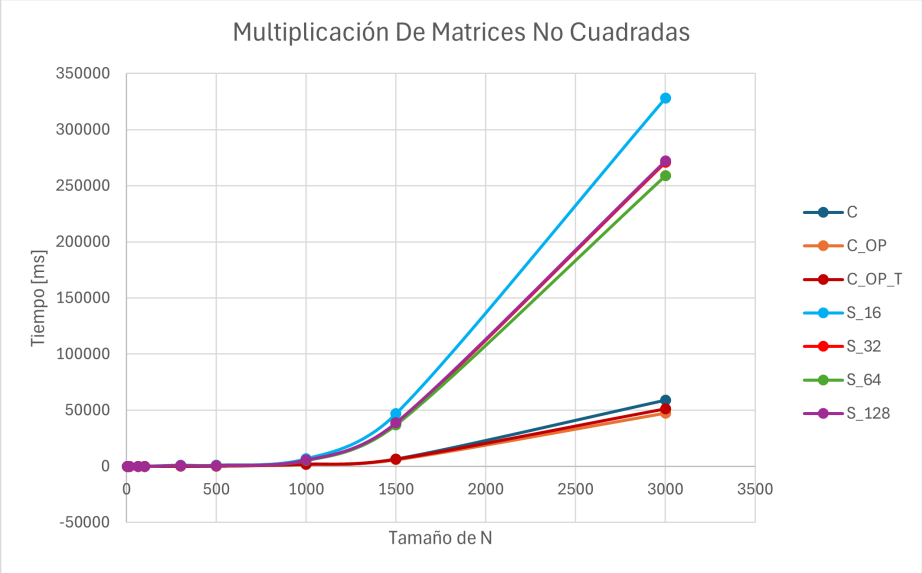


Para este gráfico el comportamiento de Strassen es bastante extraño por así decirlo, debido a que en ciertos rangos como lo es entre el 500 y el 1000, los tiempos de ejecución del algoritmo son cercanos a ser constantes, a pesar de haber variado el tamaño de las matrices (y haberlo probado varias veces), sin embargo luego de este pequeño intervalo, la función de strassen crece, como podemos notar después del intervalo 250 y 450 aproximadamente. Pero de todas formas para este nivel más pequeño de los tamaños de las matrices, en general es mejor el algoritmo cúbico tradicional o el optimizado. Las mejoras son notorias para tamaños de matrices más grandes. Este comportamiento de Strassen se puede deber principalmente por efectos de Caché y localidad de los datos.

Matrices No cuadradas de tipo $[N][M] \times [M][M]$

Table 9: Comparación de Tiempos de Ejecución

N	C [ms]	C_OP [ms]	C_OP_T [ms]	S_16 [ms]	S_32 [ms]	S_64 [ms]	S_128 [ms]
4	0,001543	0,001413	0,002285	0,013866	0,016321	0,011491	0,013425
16	0,01097	0,010319	0,013095	0,052969	0,058028	0,057097	0,05347
64	0,493373	0,479767	0,532836	3,00312	2,13552	2,05743	2,03088
100	1,82233	1,80885	1,98302	18,6533	15,4123	14,6998	15,4377
300	48,4125	47,3807	52,0754	955,758	787,493	750,545	795,152
500	224,324	219,121	238,150	939,259	784,656	753,723	787,929
1000	1838,37	1757,61	1902,48	6675,28	5488,47	5253,15	5554,33
1500	6439,39	5873,82	6430,44	46875,2	38419,9	37070,1	38900
3000	58776,9	47265,7	51335,4	327933	270972	259140	271923



Para las matrices que no son cuadradas, el método tradicional cúbico, así como los cúbicos optimizados resultan ser mucho mejores que el método de Strassen, esto debido principalmente a que el método de Strassen no está pensado para matrices que no sean cuadradas, sin embargo las matrices se pueden ajustar como es el caso que yo realicé, lo que hace este ajuste es rellenar las matrices con ceros hasta que ambas matrices sean cuadradas, esto sin embargo provoca que el método de strassen realice operaciones con números iguales a 0, en las cuales se pierde tiempo de ejecución, así como para matrices o sectores que sean solo ceros va a realizar la recurrencia de igual manera. Por lo que a pesar de que este enfoque funciona, se debería seguir modificando el código de Strassen para identificar cuando las matrices sean ceros y no realizar las operaciones o recurrencia en estos casos. Sin embargo para probar estos casos de matrices no cuadradas el ajuste funciona. Mientras que para los otros métodos no es necesario realizar algún tipo de ajuste, por esta razón son mejores en el tiempo de ejecución.

5 Conclusiones

El análisis comparativo de los algoritmos de ordenamiento y multiplicación de matrices realizado en este trabajo nos permitió evaluar la relación entre el análisis asintótico teórico y el comportamiento real en la práctica. Si bien el análisis de complejidad nos brinda una idea clara del rendimiento esperado en diferentes casos, los experimentos mostraron que factores como la localidad de datos, la optimización del uso de memoria y las implementaciones concretas de cada algoritmo juegan un rol decisivo en los tiempos de ejecución.

En el caso de los algoritmos de ordenamiento, como Bubble Sort y Selection Sort, observamos que su complejidad cuadrática $O(n^2)$ se traduce en un desempeño muy pobre en datasets de gran tamaño, lo que confirma las predicciones asintóticas. Por otro lado, algoritmos como Merge Sort y Quick Sort, ambos con complejidad $O(nlogn)$, demostraron ser significativamente más eficientes que los cuadráticos lo que también se corrobora con la notación asintótica. Sin embargo, es importante notar que Quick Sort, aunque es asintóticamente equivalente a Merge Sort en promedio, sufre un gran impacto en su rendimiento en el peor de los casos, donde su complejidad es $O(n^2)$. Este comportamiento fue evidenciado al ordenar datasets completamente ordenados, confirmando así la importancia de considerar tanto el mejor como el peor caso en análisis prácticos.
En cuanto a la multiplicación de matrices, el algoritmo cúbico tradicional y el optimizado presentaron tiempos de ejecución relativamente consistentes con su complejidad $O(n^3)$. Sin embargo, el algoritmo de Strassen, a pesar de tener una mejor complejidad asintótica de $O(n^{2.81})$, demostró que su ventaja no es tan evidente en tamaños de matrices pequeños o cuando no se ajusta adecuadamente la implementación para arquitecturas modernas. Si consideramos el caso base cuando la matriz es de tamaño 1, es el algortimo que peor rendimiento tiene. Esto revela la importancia de ajustar el nivel de recursión o aplicar técnicas híbridas (como el uso del algoritmo cúbico tradicional en matrices pequeñas, o la función de la librería estándar de C++ con Introsort) para obtener un mejor rendimiento en la práctica.
Uno de los factores clave en la evaluación de estos algoritmos fue la diferencia entre implementaciones in-place y no-in-place. Algoritmos como Quick Sort, que ordena los datos en su lugar, aprovechan mejor la localidad de los datos y minimizan el uso de memoria, mientras que Merge Sort, aunque estable y con mejor comportamiento en algunos casos, requiere espacio adicional, lo que afecta su rendimiento.
Finalmente, los experimentos mostraron que la preservación de la localidad de los datos, especialmente en los algoritmos de multiplicación de matrices, es crucial para optimizar el uso de la memoria caché. La optimización cúbica mediante la transposición de matrices demostró ser una mejora notable en tiempos de ejecución, a pesar de mantener la misma complejidad teórica.
En conclusión, si bien el análisis asintótico proporciona una guía sólida para evaluar el comportamiento de los algoritmos, los detalles de implementación, el manejo de la memoria y las características del hardware son determinantes para obtener un rendimiento óptimo en la práctica.

6 Bibliografía

Bubble Sort.
https://www.geeksforgeeks.org/bubble-sort-in-cpp/
https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

Selection Sort
https://www.geeksforgeeks.org/selection-sort-algorithm-2/
https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-selection-sort/

Merge Sort
https://www.geeksforgeeks.org/cpp-program-for-merge-sort/
https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/

Quick Sort.
https://www.geeksforgeeks.org/quick-sort-algorithm/
Quick Sort Demostración.
https://medium.com/enjoy-algorithm/quicksort-one-the-fastest-sorting-algorithms-2685fb0910c5
https://users.dcc.uchile.cl/~bebustos/apuntes/cc3001/Ordenacion/
https://en.wikipedia.org/wiki/Quicksort
https://www.redalyc.org/pdf/2570/257059815007.pdf

C++ Sort Function.
https://cplusplus.com/reference/algorithm/sort/
https://en.wikipedia.org/wiki/Introsort
Teorema Maestro.
https://es.wikipedia.org/wiki/Teorema_maestro

- Material visto en clases y en ayudantías.

Multiplicación de Matrices Método iterativo cúbico tradicional
https://www.geeksforgeeks.org/c-program-multiply-two-matrices/
https://en.wikipedia.org/wiki/Matrix_multiplication
Multiplicación de matrices iterativo cúbico optimizado
https://www.geeksforgeeks.org/program-to-find-transpose-of-a-matrix/
https://youtu.be/QGYvbsHDPxo
https://learn.microsoft.com/en-us/cpp/parallel/amp/walkthrough-matrix-multiplication?view=msvc-170
Método de multiplicación de matrices de Strassen
https://www.geeksforgeeks.org/strassens-matrix-multiplication/
https://es.wikipedia.org/wiki/Algoritmo_de_Sch%C3%B6nhage-Strassen#:~:text=El%20Algoritmo%20de%20Schonhage%2DStrassen,la%20pr%C3%A1ctica%20para%20matrices%20grandes.
https://en.wikipedia.org/wiki/Strassen_algorithm
https://es.wikipedia.org/wiki/Algoritmo_de_Strassen
https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344