REPORTE TAREA 1
ALGORITMOS Y COMPLEJIDAD

«Más allá de la notación asintótica: Análisis experimental de algoritmos de ordenamiento y multiplicación de matrices.»

Alejandro Rojo

28 de abril de 2025

00:40

Resumen

Este trabajo presenta un análisis experimental comparativo entre algoritmos de ordenamiento (SelectionSort, MergeSort y QuickSort) y de multiplicación de matrices (Naive y Strassen), evaluando la brecha entre su complejidad teórica y su rendimiento práctico. Implementados en C++, los algoritmos fueron sometidos a pruebas con conjuntos de datos de tamaños crecientes (hasta $n = 10^7$ para ordenamiento y matrices de 1024×1024) y características variadas (aleatorios, ordenados, densos, diagonales y dispersos). Los resultados confirman que SelectionSort se vuelve inviable para $n \geq 10^5$, mientras que los algoritmos de complejidad $O(n \log n)$ mantienen eficiencia incluso con grandes volúmenes de datos, siendo `std::sort` el más rápido. Para la multiplicación de matrices, Strassen ofrece ventajas marginales solo en casos específicos ($n \approx 256$), pues la sobrecarga de recursión y gestión de memoria contrarresta sus beneficios teóricos en dimensiones mayores. Se concluye que la elección óptima de algoritmo depende no solo de su complejidad asintótica, sino también de las características del hardware y los patrones de datos, sugiriéndose la exploración de implementaciones híbridas para un alto rendimiento.

Índice

1. Introducción

El presente informe se enmarca en el campo del Análisis y Diseño de Algoritmos en Ciencias de la Computación, destacando la importancia de evaluar la eficiencia y complejidad de diferentes métodos en contextos reales [1]. Con ello se busca contextualizar el problema y establecer los antecedentes necesarios para comprender la relevancia de comparar técnicas que, si bien cuentan con fundamentos teóricos sólidos, presentan comportamientos distintos en la práctica, ya que factores como el acceso a memoria y las características del hardware pueden influir significativamente en su rendimiento.

A pesar del amplio conocimiento que existe sobre algoritmos de ordenamiento y multiplicación de matrices, persiste la necesidad de evaluar comparativamente técnicas de distinto coste computacional bajo condiciones experimentales controladas.En este trabajo concretamente se evaluaran experimentalmente dos grupos de algoritmos:

- **Algoritmos de Ordenamiento:** Selection Sort, Mergesort y Quicksort.
- **Algoritmos de Multiplicación de Matrices:** Naive y Strassen.

Se conoce que **SelectionSort** opera con una complejidad $O(n^2)$, lo que lo hace ineficiente para grandes volúmenes de datos. Por otro lado, **Mergesort** ofrece una complejidad de $O(nlogn)$ a costa de una mayor complejidad en su implementación,y **Quicksort**, aunque generalmente resulta muy eficiente con una complejidad promedio de $O(nlogn)$, puede presentar una complejidad de $O(n^2)$ en su peor caso debido a una mala elección del pivote. En lo que respecta a la multiplicación de matrices, el método tradicional (Naive) tiene una complejidad de $O(n^3)$, mientras el algoritmo de Strassen reduce la complejidad teórica a aproximadamente $O(n^{2,81})$.

El propósito de este informe es realizar una evaluación experimental de los cinco algoritmos, midiendo su tiempo de ejecución en diversos escenarios de datos: aleatorios, parcialmente ordenados y en el caso de matrices, de tamaños crecientes. Para ello, se implementaron manualmente todos los algoritmos en C++ y se utilizó la biblioteca `<chrono>` para obtener mediciones precisas de los tiempos de ejecucion de cada algoritmo. De este modo, se busca identificar las diferencias entre la complejidad teórica y el desempeño real, así como destacar las ventajas y limitaciones de cada enfoque.

2. Implementaciones

Las implementaciones de todos los codigos descritos en el infrme estaran presentes en sigueinte repositorio de GitHub, asi como los archivos que ejecutaran todos los casos de prueba (sorting.cpp y matrixMmultiplioation.cpp)

https://github.com/pinkyig/INF221-2025-1-TAREA-1

3. Experimentos

Para asegurar la reproducibilidad de los experimentos y con el fin de que otros puedan verificar y extender estos resultados,se detalla a continuación la infraestructura utilizada.

El hardware consiste en:
- un procesador Apple M1,
- 8 GB de memoria RAM
- almacenamiento en SSD NVMe.

El entorno software corresponde a MacOS 15.4.1 Sequoia, compilador `c++17`, y las librerías estándar de C++ junto con el módulo `<chrono>` para la toma de tiempos. Todos los scripts de generación de datos y de gráficas fueron escritos en Python 3.8, empleando `matplotlib` y automatizados mediante un `Makefile`.

3.1. Dataset (casos de prueba)

Los casos de prueba empleados en este informe fueron los proporcionados en el enunciado, generados por los scripts en Python entregados originalmente. No se consideraron tamaños de entrada adicionales ni variaciones externas. Los conjuntos cubren:

- **Ordenamiento:**

  - Tipo de datos: Los datos de los archivos de prueba estan conformados por solo numeros enteros.

  - Tamaño de la muestra: Los tamanios de muestra abarcan $n = 10; 1000; 100000; 10000000$, con distintos largo de los numeros siendo de 1 a 7 dígitos.

  - Distribución: ascendente, descendente y aleatoria, cada una con 3 casos por cada tamaño y distribución de los datos

  - Método de generación: Todos y cada uno de los casos de prueba se generan con el script presente en `/code/sorting/scripts/array_generator.py`

- **Multiplicación de matrices:**

  - Tipo de datos: Los datos de los archivos de prueba estan conformados por solo numeros enteros.

  - Tamaño de la muestra: Los tamaños de muestra abarcan $n = 16, 64, 256, 1024$, con distintos tipos de números, pudiendo los valores ser binarios (0 y 1) o Decimales (0 al 9).

  - Distribución: Matices Densas, Diagonales y Dispersas, cada una con 3 casos por cada tamaño y distribución de los datos

  - Método de generación: Todos y cada uno de los casos de prueba se generan automaticamente con el script presente en `code/matrix_multiplication/scripts/matrix_generator.py`

3.2. Resultados

Los tiempos de ejecución y el uso de memoria se registraron en archivos CSV ubicados en `code/sorting/data/` `code/matrix_multiplication/data/`.

Las gráficas de los resultados se generan automáticamente con los scripts de Python definidos en el `Makefile`, ejecutando el objetivo `plots`, lo que permite reproducir las visualizaciones bajo cualquier infraestructura similar.

Resultados ordenamientos

A continuación se muestran los resultados principales. Cada Gráfico comparara los cuatro algoritmos para un tipo de entrada, representando en el eje X el tamaño n y en el eje Y el tiempo medio de ejecución (ms)

Datos aleatorios

Probando con datos aleatorios obtuvimos en promedio estos tiempos de ejecucion

Algoritmo	SelectionSort(µs)	MergeSort(µs)	QuickSort(µs)	`std::sort`(µs)
10	0	0.002	0	0
1000	2	0.156	0.053	0.066
100000	9498	8.17	3.50	2.80
10 000 000	$+\infty$	987	461	364

Cuadro 1: Tiempos de ejecución promedio para datos aleatorios

Como se aprecia en el cuadro 1, los tiempos de ejecución promedio para entradas ordenadas ascendente muestran una clara diferencia entre el comportamiento cuadrático de `SelectionSort` y las curvas $O(n \log n)$ de los demás algoritmos. Esta misma información se representa de forma visual en la Figura 1, donde la pendiente más pronunciada de la línea de `SelectionSort` contrasta con las trayectorias más suaves de `MergeSort`, `QuickSort` y `std::sort`.

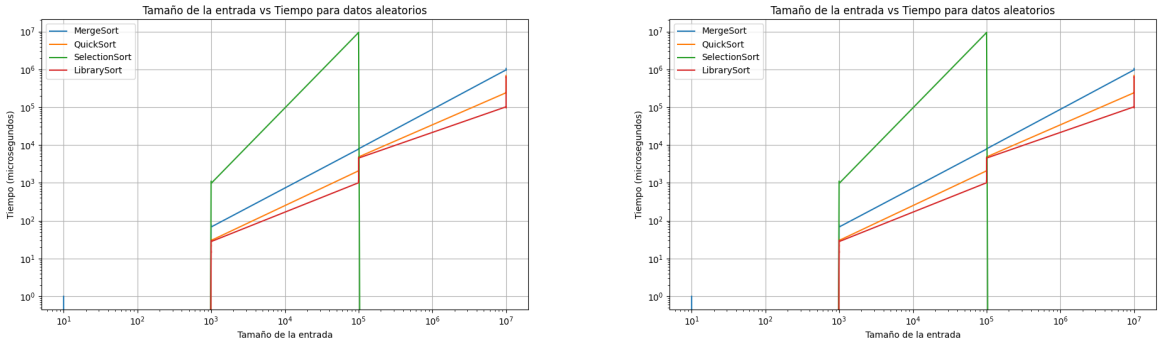

Figura 1: Ejemplo de scatterplot hecho con matplotlib.

Datos ordenados de forma Ascendente

Probando con datos ordenados de manera ascendente, obtuvimos en promedio estos tiempos de ejecución

Algoritmo	SelectionSort(µs)	MergeSort(µs)	QuickSort(µs)	`std::sort`(µs)
10	0	0.001	0	0
1000	1.96	0.105	0.012	0.006
100 000	9432	5.69	0.942	0.212
10 000 000	$+\infty$	722	153	18.04

Cuadro 2: Tiempos de ejecución promedio para datos ascendentes

Como se observa en la Tabla 2, para entradas ordenadas ascendentemente los tiempos de `SelectionSort` igualmente crecen de manera drástica en comparación con los algoritmos $O(n \log n)$.

Para $n = 10$ los cuatro métodos tardan prácticamente 0 ms, pero ya para $n = 10^3$ `SelectionSort` registra 1,96 ms frente a 0,105 ms de `MergeSort`, 0,012 ms de `QuickSort` y 0,006 ms de `std::sort`. Al aumentar a $n = 10^5$, `SelectionSort` alcanza 9432 ms, mientras que `MergeSort`, `QuickSort` y `std::sort` se quedan en 5,69 ms, 0,942 ms y 0,212 ms, respectivamente. Para $n = 10^7$, `SelectionSort` resulta inviable ($+\infty$), en contraste con los 722 ms, 153 ms y 18,04 ms de `MergeSort`, `QuickSort` y `std::sort`.

Esta misma diferencia de escalado queda reflejada de manera más intuitiva en la Figura 2, donde la curva de `SelectionSort` muestra una pendiente mucho más pronunciada en comparación con las trayectorias más planas de los algoritmos $O(n \log n)$. Así, la gráfica confirma visualmente la penalización del comportamiento cuadrático de `SelectionSort` incluso en el "mejor caso" de entradas ya ordenadas.

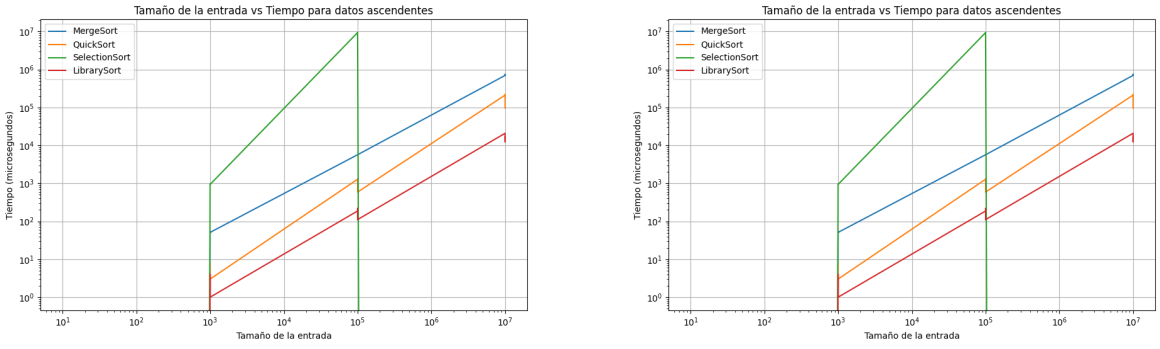

Figura 2: Ejemplo de scatterplot hecho con matplotlib.

Datos ordenados de forma descendente

Probando con datos ordenados de manera descendente,obtuvimos en promedio estos tiempos de ejecución

Algoritmo	SelectionSort(µs)	MergeSort(µs)	QuickSort(µs)	`std::sort`(µs)
10	0	0.001	0.0001	0
1000	1.62	0.091	0.009	0.008
100 000	9422	5.81	0.924	0.188
10 000 000	$+\infty$	746	142	18.66

Cuadro 3: Tiempos de ejecución promedio para datos descendentes

Como se aprecia en la Tabla 3, cuando los datos llegan ya ordenados de forma descendente, el rendimiento de `SelectionSort` se ve gravemente afectado, hasta el punto de volverse inviable para tamaños muy grandes. En contraste, `MergeSort`, `QuickSort` y `std::sort` conservan su comportamiento $O(n \log n)$, sufriendo únicamente un incremento moderado en los tiempos de ejecución. Esta disparidad en la escalada de tiempos queda de manifiesto en la Figura 3, donde las curvas de los algoritmos que usan dividir y conquistar muestran pendientes suaves frente a la pronunciada pendiente observada en `SelectionSort`.

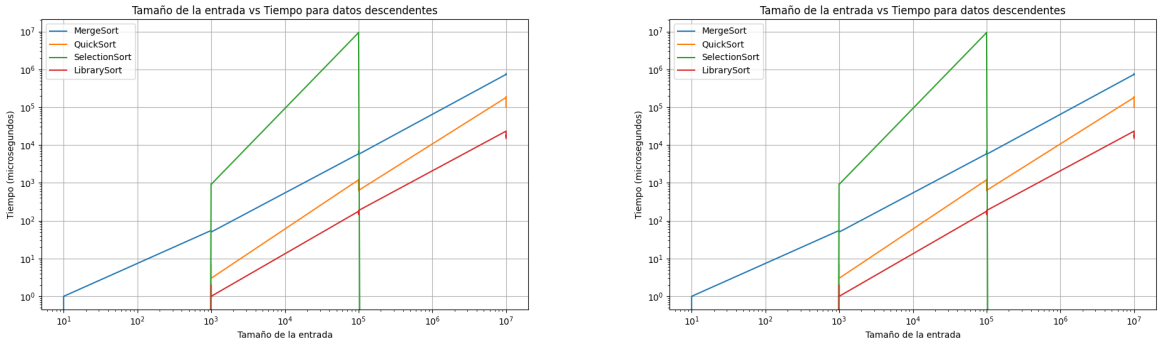

Figura 3: Ejemplo de scatterplot hecho con matplotlib.

Resultados Multiplicación de Matrices

A continuación se presentan los resultados principales de los dos algoritmos de multiplicación de matrices implementados. Cada gráfico compara dichos algoritmos para un tipo de entrada distinto, representando en el eje X el tamaño de la matriz n considerando matrices cuadradas de dimensión $(n \times n)$ y en el eje Y el tiempo medio de ejecución (en milisegundos).

A continuación se muestran los gráficos detallados para cada tipo de entrada y un breve análisis de sus implicaciones prácticas.

Matrices Densas

Probando con matrices densas se obtuvieron en prodmedio los siguientes tiempos de ejecucion

Tamanño $(n \times n)$	Naive(ms)	Strassen(ms)
16	0.132	0.345
64	2.63	2.88
256	102	98
1024	6459	6650

Cuadro 4: Tiempos de ejecución promedio para matrices densas

La Tabla 4 muestra los tiempos promedio de ejecución obtenidos para la multiplicación de matrices densas utilizando los algoritmos Naive y Strassen. Se observa que, para tamaños pequeños, ambos algoritmos presentan tiempos similares, aunque Strassen resulta ligeramente más lento en = n=16 debido al mayor overhead de su estructura recursiva. Sin embargo, a medida que el tamaño de las matrices crece, las diferencias se atenúan e incluso Strassen logra superar marginalmente al método Naive en n=256. en la figura 4 Para tamaños mayores,como n=1024, ambos algoritmos exhiben tiempos de ejecución similares, lo que sugiere que el impacto teórico de Strassen en la reducción de complejidad se ve amortiguado por factores prácticos como el manejo de memoria y la sobrecarga de recursión.

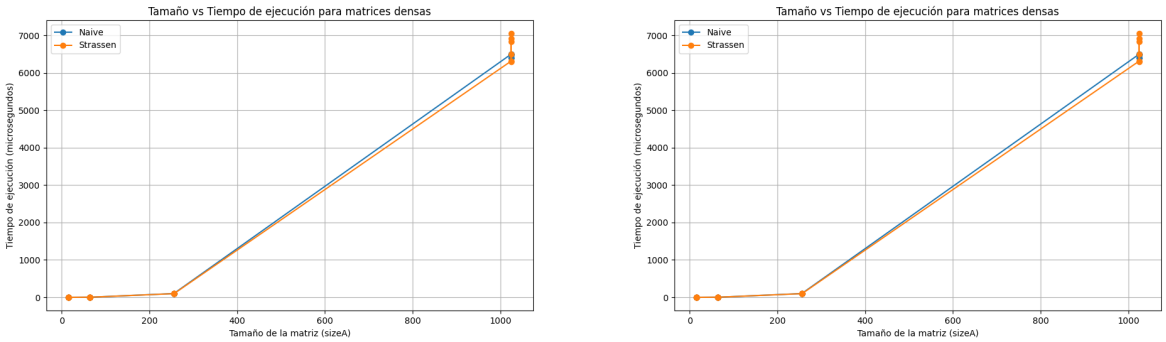

Figura 4: Ejemplo de scatterplot hecho con matplotlib.

Matrices Diagonales

Probando con Matrices Diagonales se obtuvieron en prodmedio los siguientes tiempos de ejecución

Tamanño ($n \times n$)	Naive(ms)	Strassen(ms)
16	0.065	0.161
64	1.59	1.86
256	102	96.71
1024	6498	6623

Cuadro 5: Tiempos de ejecución promedio para matrices diagonales

La Tabla 5 recoge los tiempos promedio de ejecución para multiplicación de matrices diagonales, y el Gráfico 5 ilustra de forma visual esta evolución. Al comparar estos resultados con los obtenidos previamente para matrices densas, se aprecia que ambos algoritmos (Naive y Strassen) obtienen un marcado beneficio en el caso diagonal: los tiempos iniciales para $n = 16$ y $n = 64$ se reducen casi a la mitad respecto al escenario denso, reflejando la simplicidad del cómputo sobre estructuras con elementos mayoritariamente nulos. No obstante, para tamaños grandes ($n = 256$ y $n = 1024$) las diferencias se atenúan, y los tiempos convergen nuevamente a valores similares, puesto que la sobrecarga de recorrido y acceso a memoria acaba dominando el coste total. Este contraste evidencia cómo la naturaleza de la matriz influye en el rendimiento de los algoritmos y subraya la importancia de seleccionar métodos adaptados al patrón de datos.

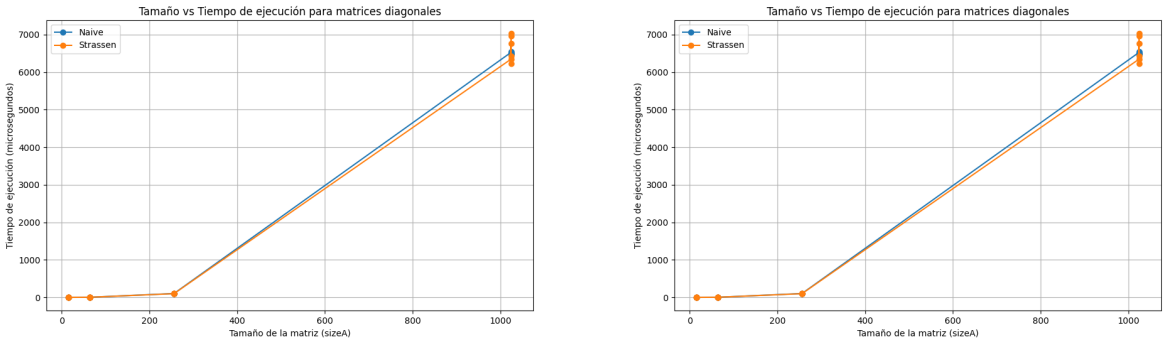

Figura 5: Ejemplo de scatterplot hecho con matplotlib.

Matrices Dispersa

Probando con Matrices Dispersas se obtuvieron en promedio los siguientes tiempos de ejecución

Tamanño ($n \times n$)	Naive(ms)	Strassen(ms)
16	0.055	0.144
64	1.561	1.782
256	104	113
1024	6555	6819

Cuadro 6: Tiempos de ejecución promedio para matrices dispersas

La Tabla 6 presenta los tiempos promedio de ejecución para la multiplicación de matrices dispersas, mientras que el Gráfico 6 ilustra la evolución de estos valores en función del tamaño n. En comparación con los resultados de matrices densas y diagonales, observamos que para tamaños reducidos ($n = 16$ y $n = 64$) los algoritmos Naive y Strassen se benefician de la escasez de elementos no nulos, mostrando tiempos incluso inferiores a los de las matrices diagonales, y muy por debajo de los obtenidos en el escenario denso. Sin embargo, al aumentar n (especialmente a partir de 256), las mejoras se diluyen: el método Naive alcanza valores de tiempo similares (e incluso ligeramente superiores) a los de matrices densas, y Strassen incurre en un costo extra por la gestión de la estructura dispersa, superando en algunos casos los tiempos registrados para matrices diagonales. Este comportamiento pone de manifiesto que, aunque la dispersidad aporta ventajas en cómputos pequeños, el sobrecoste de recorrer y verificar la ubicación de los ceros puede contrarrestar la ganancia teórica en instancias de gran tamaño.

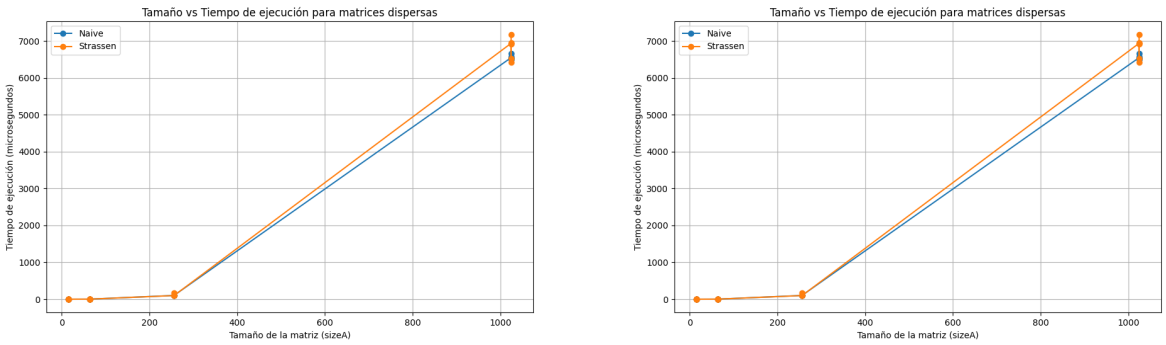

Figura 6: Tiempos de ejecución promedio para matrices dispersas

4. Conclusiones

En este trabajo se llevó a cabo una evaluación experimental de cinco algoritmos clásicos —tres de ordenamiento (SelectionSort, MergeSort y QuickSort, junto con la implementación de `std::sort`) y dos de multiplicación de matrices (Naive y Strassen)— sobre diversos tipos de entrada y tamaños crecientes. El objetivo planteado en la introducción—cuantificar la brecha entre complejidad teórica y rendimiento real bajo condiciones controladas—se ha alcanzado satisfactoriamente mediante la obtención de mediciones reproducibles y un análisis comparativo detallado.

Para los algoritmos de ordenamiento, los resultados confirman la penalización dramática del enfoque cuadrático de SelectionSort frente a las curvas $O(n \log n)$ de MergeSort, QuickSort y `std::sort`. En todos los escenarios (aleatorio, ascendente y descendente) SelectionSort se vuelve inviable a partir de $n \approx 10^5$, mientras que MergeSort y QuickSort mantienen tiempos moderados incluso para $n = 10^7$, destacando la robustez de `std::sort` como la opción más eficiente en promedio

Respecto a la multiplicación de matrices, el método Naive exhibe el crecimiento cúbico esperado, superando los 6 400 ms para $n = 1024$, mientras que Strassen logra ventajas marginales alrededor de $n = 256$, pero ve reducida su eficacia práctica en grandes dimensiones debido al overhead recursivo y al coste de memoria adicional. Al explorar matrices diagonales y dispersas, se confirmó que la escasez de elementos no nulos acelera ambos algoritmos en rangos pequeños ($n \leq 64$), aunque a partir de $n \geq 256$ las mejoras teóricas quedan amortiguadas por el coste de acceso y verificación de ubicaciones nulas.

En conjunto, estos hallazgos ponen de manifiesto que la elección de algoritmo debe basarse no solo en su orden de complejidad asintótica, sino también en características prácticas del hardware (cache, gestión de memoria) y del patrón de datos. Para aplicaciones con matrices de gran tamaño, una implementación híbrida o bloqueada que combine lo mejor de Naive y Strassen, o el uso de bibliotecas optimizadas, podría ofrecer ventajas adicionales.

Finalmente, los resultados alcanzados responden al planteamiento inicial al demostrar que las discrepancias entre teoría y práctica son sustanciales: la sobrecarga de gestión de memoria y recursividad puede anular las ganancias teóricas en escenarios reales, y solo un análisis experimental riguroso permite identificar la opción óptima en cada contexto. Como trabajo futuro, se sugiere analizar variantes paralelas y basadas en BLAS para extender la validez de estas conclusiones a entornos de alto rendimiento.

A. Apéndice 1

Aquí puede agregar tablas, figuras u otro material que no se incluyó en el cuerpo principal del documento, ya que no constituyen elementos centrales de la tarea. Si desea agregar material adicional que apoye o complemente el análisis realizado, puede hacerlo en esta sección.

> Esta sección es solo para material adicional. El contenido aquí no será evaluado directamente, pero puede ser útil si incluye material que será referenciado en el cuerpo del documento. Por lo tanto, asegúrese de que cualquier elemento incluido esté correctamente referenciado y justificado en el informe principal.

Referencias

[1] Chris Mack y Lola Muxamedova. *How to Write a Good Scientific Paper*. Nov. de 2019.