REPORTE TAREA 1
ALGORITMOS Y COMPLEJIDAD

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

Sergio Cárcamo Naranjo

8 de septiembre de 2025

15:55

Resumen

*Esta investigación realiza un **Análisis experimental de la complejidad computacional** de algoritmos de ordenamiento y multiplicación de matrices. Se implementaron en C++ cinco algoritmos de ordenamiento (Insertion Sort, Merge Sort, Quick Sort con selección aleatoria de pivote, Panda Sort y Sort de la STL) y dos de multiplicación de matrices (Naive y Strassen), midiéndose su rendimiento en términos de tiempo de ejecución y uso de memoria para distintos tamaños y tipos de entrada. Los resultados se visualizaron mediante gráficos generados en Python, permitiendo contrastar el comportamiento práctico con la complejidad teórica esperada.*

Se concluye que, si bien la complejidad teórica predice el crecimiento asintótico, factores como la implementación, el lenguaje y la estructura de los datos introducen variaciones significativas en el desempeño real. Este estudio sienta las bases para futuras investigaciones sobre optimizaciones aplicadas y el análisis de algoritmos en escenarios de gran escala.

Índice

1. Introducción

El análisis de algoritmos es un pilar fundamental en las Ciencias de la Computación, permitiendo evaluar y predecir el comportamiento de las soluciones de software en términos de eficiencia y consumo de recursos. En particular, el estudio de algoritmos de ordenamiento y multiplicación de matrices representa un área clásica y bien establecida, donde la complejidad teórica es ampliamente conocida y documentada. Sin embargo, la brecha entre el desempeño teórico y el comportamiento práctico de un algoritmo bajo condiciones reales de ejecución (considerando factores como la arquitectura del hardware, gestión de memoria caché y el patrón de acceso a datos) motiva la necesidad de un análisis experimental riguroso.

Este reporte se enmarca en el campo del **Análisis y Diseño de Algoritmos**, específicamente en la evaluación realista de su desempeño. Mientras que el análisis teórico provee cotas superiores e inferiores de complejidad, el análisis experimental permite validar estas cotas, identificar parámetros ocultos y observar el comportamiento real en escenarios concretos, lo cual es crucial para la selección informada de algoritmos en aplicaciones prácticas. Existen estudios previos que comparan el rendimiento de algoritmos clásicos, pero estos suelen realizarse en contextos controlados y con configuraciones específicas, dejando espacio para explorar su comportamiento en diferentes dominios de datos y tamaños de entrada, particularmente en el contexto más limitado que estudiantes de pregrado como nosotros tenemos.

Tenemos como objetivo realizar un **análisis experimental** del tiempo de ejecución y uso de memoria de cinco algoritmos de ordenamiento: Insertion Sort, Merge Sort, Quick Sort (con la mediana como pivote), un algoritmo inventado denominado Panda Sort, y el algoritmo de la librería estándar de C++ (‘std::sort‘) y dos algoritmos de multiplicación de matrices: el método Naive y el algoritmo de Strassen. La **pregunta central** que guía este análisis es: ¿hasta qué punto el rendimiento práctico de estos algoritmos, medido experimentalmente, se alinea con sus complejidades teóricas esperadas bajo diferentes distribuciones de datos (arreglos ordenados, inversos, aleatorios; matrices densas, dispersas, diagonales) y distintos tamaños de entrada?

El propósito de este reporte es doble: primero proveer una comparación cuantitativa del desempeño de estos algoritmos en una variedad de casos de prueba, generando evidencia empírica que complemente el análisis teórico visto en el curso; y segundo introducir y evaluar el desempeño de Panda Sort, un algoritmo de ordenamiento original desarrollado para esta tarea, cuya lógica y diseño se documentan y analizan. La **novedad** de este trabajo en el contexto de un curso de pregrado radica en la integración de un algoritmo propio (Panda Sort) dentro de un conjunto de métodos clásicos, permitiendo una comparación directa y en igualdad de condiciones, así como en la aplicación de una metodología de medición sistemática para ambos problemas (ordenamiento y multiplicación de matrices) que considera no solo el tiempo de ejecución sino también el consumo de memoria.

2. Implementaciones

https://github.com/INF221-20252/tarea-1-frostodev

3. Experimentos

El hardware utilizado en este estudio consiste de:

- **Procesador**: Intel(R) Core(TM) i5-9400 CPU @ 2.90GHz
- **RAM**: 16 GB DDR4-2666 MT/s Single Channel
- **GPU**: NVIDIA GeForce GTX 1060 6GB
- **SSD**: 256 GB WDC WDS240G2G0C-00AJM0 NVMe
- **HDD**: 14 TB WDC WD142PURP-85B6HY0 SATA

Respecto al software y toolchain usados, estos consisten de:

- Ubuntu 24.04.3 LTS (WSL2) sobre Microsoft Windows 11 Enterprise IoT LTSC
- GNU gcc 13.3.0 (Ubuntu 13.3.0-6ubuntu2 24.04)
- GNU Make 4.3 x86_64-pc-linux-gnu
- Python 3.13.7

3.1. Dataset (casos de prueba)

Para los algoritmos de ordenamiento de arreglos, la entrada corresponde a **72** archivos de texto en el formato `{n}_{t}_{d}_{m}.txt`:

- n hace referencia a la cantidad de elementos (o largo del arreglo) y pertenece al conjunto $\mathcal{N} = \{10^1, 10^3, 10^5, 10^7\}$.
- t hace referencia al tipo de matriz, y pertenece al conjunto $\mathcal{T} = \{\text{ascendente}, \text{descendente}, \text{aleatorio}\}$.
- d hace referencia al conjunto dominio de cada elemento del arreglo. $d = \{D1, D7\}$, donde $D7$ implica que el dominio es $\{0,1,2,3,4,5,6,7,8,9\}$ y $D7$ que el dominio es $\{0,1,2,3,...,10^7\}$.
- m hace referencia a la muestra aleatoria (o caso de prueba) y pertenece al conjunto $\mathcal{M} = \{a, b, c\}$.

Cada uno de estos archivos contiene exactamente **un** arreglo, con sus elementos separados por un espacio.

Para los algoritmos de multiplicación de matrices, la entrada corresponde a **144** archivos, `{n}_{t}_{d}_{m}_1.txt` y `{n}_{t}_{d}_{m}_2.txt`, que corresponden a las matrices M_1 y M_2 respectivamente, y cuyo formato es el siguiente:

- n hace referencia a la dimensión de la matriz (n filas y n columnas) y pertenece al conjunto $\mathcal{N} = \{2^4, 2^6, 2^8, 2^{10}\}$.
- t hace referencia al tipo de matriz, y pertenece al conjunto $\mathcal{T} = \{\text{dispersa}, \text{diagonal}, \text{densa}\}$.
- d hace referencia al dominio de cada coeficiente de la matriz $d = \{D0, D10\}$, donde $D0$ implica que el dominio es $\{0,1\}$ y $D10$ que el dominio es $\{0,1,2,3,4,5,6,7,8,9\}$.
- m hace referencia a la muestra aleatoria (o caso de prueba) y pertenece al conjunto $\mathcal{M} = \{a, b, c\}$.

Estos archivos se tratan de **a pares** para generar la salida, y los elementos de cada casilla están separados por espacios.

Para generar los archivos de entrada para ambos experimentos, se utiliza un script específico para cada experimento, llamado ‘array_generator.py‘ y ‘matrix_generator.py‘ respectivamente.

3.2. Resultados

Los resultados de mediciones de tiempo y memoria para ambos experimentos se realizaron durante la ejecución, guardando los datos en el directorio ‘data/measurements‘ correspondiente a cada experimento.

En una buena porción de los experimentos, la memoria utilizada por cada algoritmo es demasiado pequeña para ser medida eficazmente. Debido a ello, algunas mediciones reportan 0 KB. Sin embargo, debido a la escala de los gráficos generados, no se presentan incosistencias mayores.

Para una visualización más directa y comprensible de los resultados, se generan gráficos usando las librerías **pandas** y **matplotlib** de Python. Se debe considerar que los gráficos usan **escala logarítmica**.

Comenzando con el análisis de algoritmos de ordenamiento, se generan tres gráficos, cada uno correspondiente al tipo de arreglo de entrada (ascendiente, descendiente, aleatorio). Estos son los resultados:

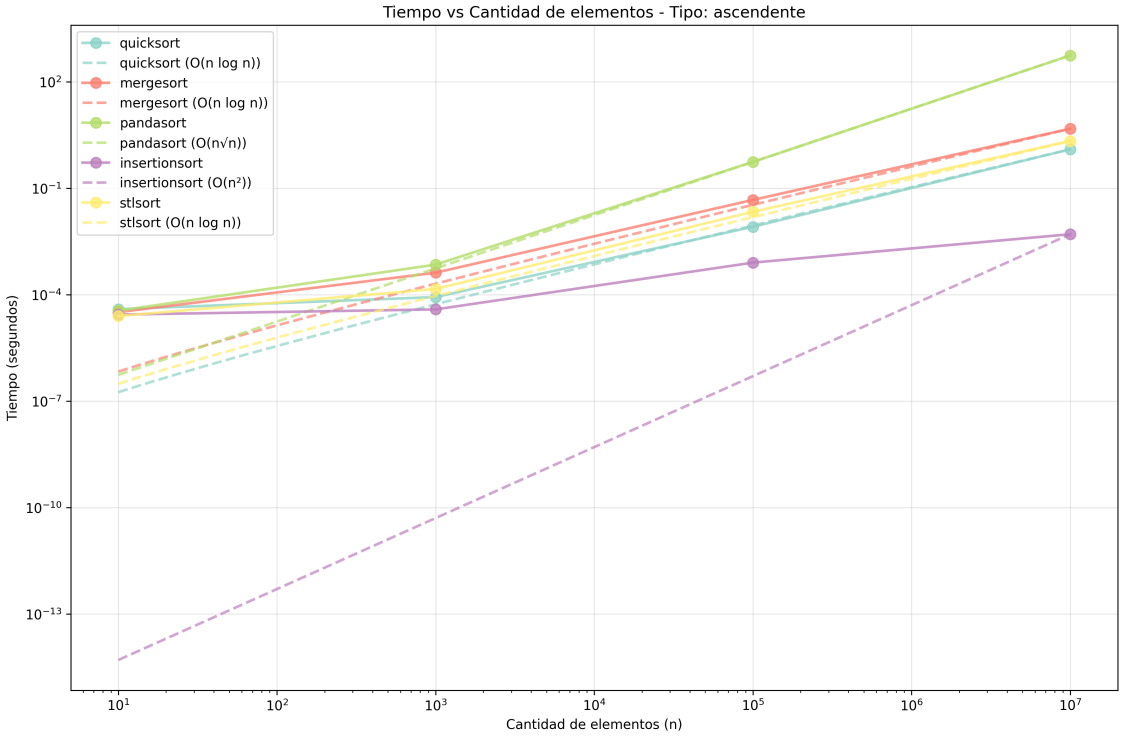

Figura 1: Medición de tiempos de ordenamiento de arreglo ascendente

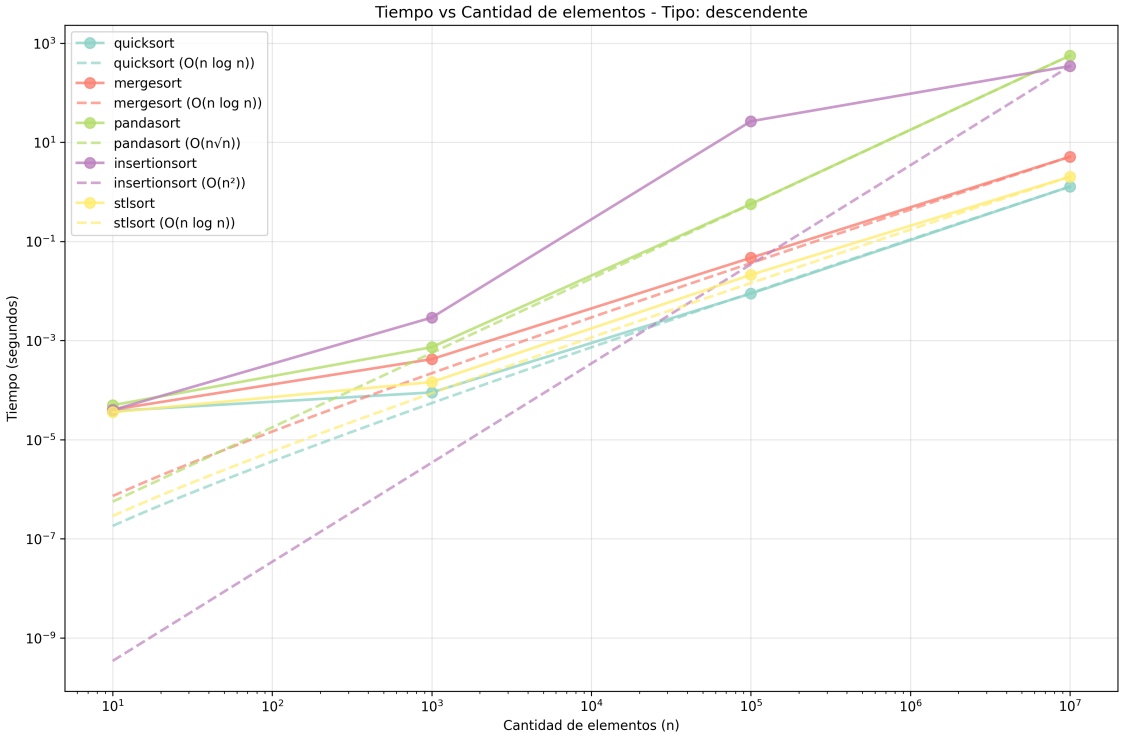

Figura 2: Medición de tiempos de ordenamiento de arreglo descendente

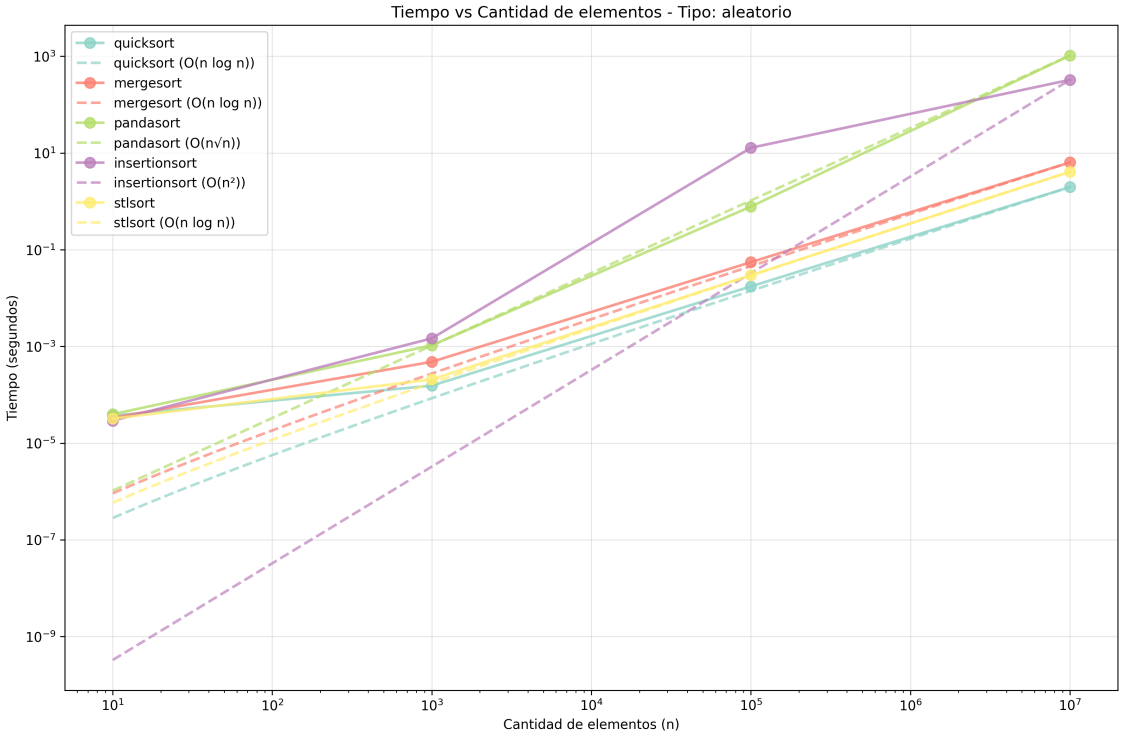

Figura 3: Medición de tiempos de ordenamiento de arreglo aleatorio

Debemos considerar además que en el caso de **Insertion Sort**, la cantidad de datos se truncó a 500.000 elementos, debido a que considerando su complejidad teórica cuadrática y el hardware actual, se tomaría varios días en ordenar un único arreglo.

En un análisis visual inmediato, podemos observar que para tamaños pequeños de arreglos el tiempo de ejecución es prácticamente despreciable, con valores menores a **0.1s** en la mayoría de arreglos de tamaño 10^4. Por otro lado para arreglos de gran tamaño los tiempos de ejecución crecen de forma significativa conforme el tamaño del arreglo aumenta.

En un análisis más profundo, podemos notar que los gráficos respetan en términos generales las complejidades teóricas de cada algoritmo, correspondiente a los casos mejor, peor y promedio (ascendente, descendente y aleatorio respectivamente).

Ahora, para el análisis de algoritmos de multiplicación de matrices, generamos dos gráficos, uno combinando la complejidad temporal de ambos algoritmos y comparándola con la teórica, y el otro mostrando el uso de memoria (se debe considerar n como el tamaño/cantidad de filas/columnas):

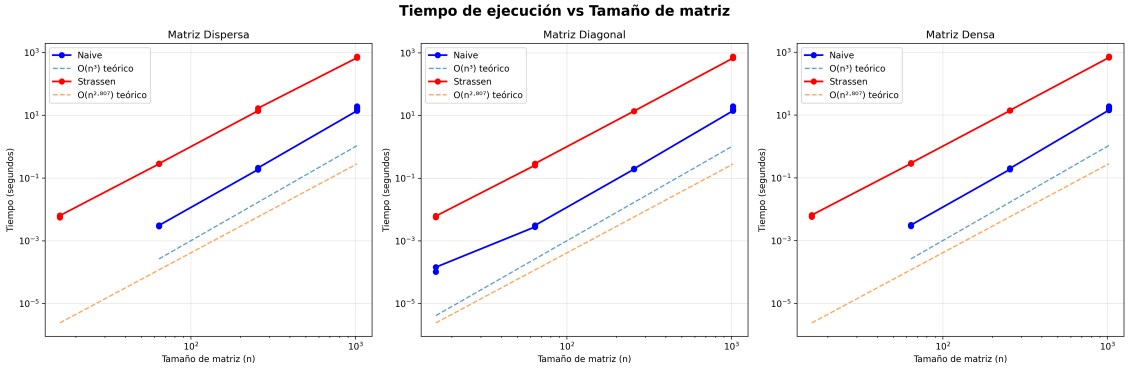

Figura 4: Medición de tiempos de multiplicación de matrices de ambos algoritmos

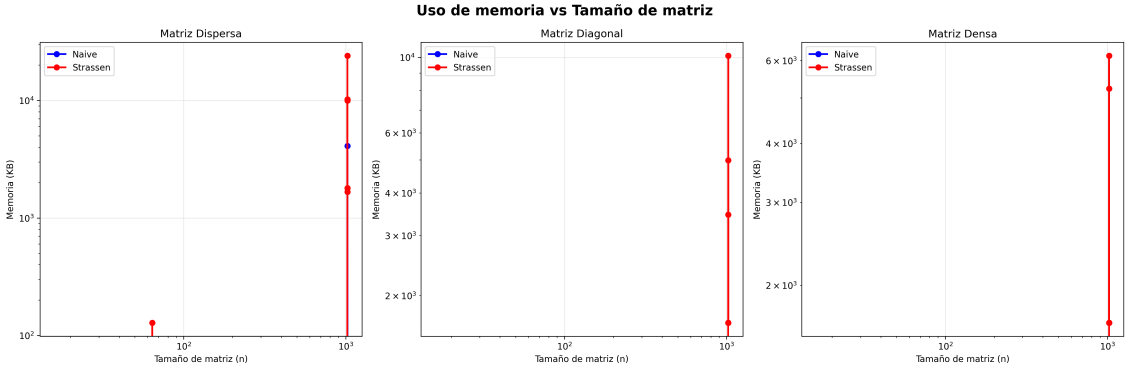

Figura 5: Comparación uso de memoria en ambos algoritmos

Podemos notar que ambos algoritmos respetan en gran medida su complejidad teórica. Sin embargo, el algoritmo **Strassen**, según los resultados de este expermiento, parece ser menos eficiente en terminos generales que el algoritmo Naive.

Esto es debido a que la implementación actual de Strassen genera una sobrecarga (overhead) excesivo, pues para matrices más pequeñas se recomienda utilizar Naive, y como este algoritmo es de tipo **dividir y conquistar**, siempre se llegará a aplicar recursivamente el algoritmo a matrices pequeñas.

Esta ineficiencia (en este caso particular) es por diseño, pues el propósito de este reporte es estudiar la complejidad para una implementación "pura"de Strassen, que no tenga que recurrir a Naive para las submatrices pequeñas.

4. Conclusiones

Para finalizar, los resultados de este estudio experimental permiten concluir que la **eficiencia práctica** de cada algoritmo depende no solo de su complejidad teórica o de la implementación, sino también (y de manera crucial) de las **características de la entrada**, confirmando así la hipótesis inicial de que existe una **elección ideal** del algoritmo según el contexto de uso.

En el caso de los algoritmos de **ordenamiento**, se observa que:

- **MergeSort** y **std::sort** demuestran robustez y eficiencia consistente en todos los órdenes iniciales, con un comportamiento cercano a $O(n \log n)$.
- **QuickSort**, aunque eficiente en promedio, confirma su sensibilidad a la elección del pivote, degradándose a $O(n^2)$ en el peor caso (entrada descendente), a menos que se use una estrategia avanzada de selección.
- **InsertionSort** resulta competitivo para entradas pequeñas o casi ordenadas ($O(n)$), pero su ineficiencia en casos promedio y peores casos ($O(n^2)$) lo limita severamente.
- **PandaSort**, aunque de orden superior teórico ($O(n\sqrt{n})$), mostró un comportamiento interesante en ciertos escenarios, lo que sugiere que su diseño merece un análisis más profundo en futuros trabajos.

Respecto a la **multiplicación de matrices**:

- El algoritmo **Naive** confirmó su complejidad cúbica $O(n^3)$ en todos los casos, siendo simple pero costoso para dimensiones grandes.
- **Strassen**, con complejidad $O(n^{\log_2 7})$, mostró ventajas claras para matrices de gran tamaño, aunque con *overhead* constante que lo hace menos eficiente para matrices pequeñas.

Estos resultados refuerzan la importancia del **análisis experimental** como complemento al teórico, permitiendo validar, contrastar y contextualizar el comportamiento de los algoritmos bajo condiciones reales. Además, se destaca la relevancia de considerar no solo el tiempo de ejecución, sino también el uso de memoria y las particularidades de la implementación.

En síntesis, la elección del algoritmo debe basarse en un equilibrio entre su garantía teórica, su adaptabilidad al tipo de entrada, y los recursos computacionales disponibles. Esta tarea constituye un primer paso en el desarrollo de criterios fundados para la selección informada de algoritmos en aplicaciones prácticas.

A. Apéndice 1

Algoritmo	Ascendente	Descendente	Aleatorio (Promedio)
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n \log n)^{\dagger}$	$O(n^2)$	$O(n \log n)$
std::sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
PandaSort	$O(n\sqrt{n})$	$O(n\sqrt{n})$	$O(n\sqrt{n})$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$

Cuadro 1: Complejidad temporal de algoritmos de ordenamiento según el orden inicial del arreglo.

Algoritmo	Mejor Caso	Peor Caso	Caso Promedio
Naive	$O(n^3)$	$O(n^3)$	$O(n^3)$
Strassen	$O(n^{\log_2 7})$	$O(n^{\log_2 7})$	$O(n^{\log_2 7})$

Cuadro 2: Complejidad temporal de algoritmos de multiplicación de matrices.

† Depende del método de selección de pivote. En el peor caso (pivote mal elegido), QuickSort puede degradarse a $O(n^2)$.