REPORTE TAREA 1
ALGORITMOS Y COMPLEJIDAD

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

Miguel Rivero Medrano

7 de septiembre de 2025

16:22

Resumen

Este reporte analiza experimentalmente distintos algoritmos de ordenamiento y de multiplicación de matrices bajo escenarios que simulan condiciones de estrés computacional. El objetivo fue contrastar su comportamiento práctico con las complejidades teóricas asociadas. A partir de los resultados obtenidos y de las gráficas generadas, se observa que la elección del algoritmo más adecuado depende en gran medida del caso de uso específico. Si bien las tendencias teóricas se cumplen en términos generales, las constantes ocultas y el manejo de memoria tienen un impacto decisivo en el rendimiento real, lo que puede volver poco convenientes a ciertos algoritmos en instancias de tamaño moderado.

Índice

1. Introducción

El análisis de algoritmos no se queda solo en lo que la teoría predice con la notación asintótica. En la práctica influyen muchos otros factores: las constantes ocultas, el manejo de memoria, la forma en que se implementa cada rutina e incluso las características del hardware donde se ejecuta. Por eso resulta necesario complementar el estudio teórico con experimentos que muestren cómo se comportan realmente los algoritmos en condiciones concretas.

En esta tarea se trabajó con dos problemas clásicos: el ordenamiento de arreglos de enteros y la multiplicación de matrices cuadradas. Para el primero se consideraron distintos enfoques, desde métodos básicos como `InsertionSort`, hasta algoritmos eficientes como `MergeSort`, `QuickSort`, el experimental `PandaSort` y la función optimizada `std::sort`. Para el segundo se compararon las aproximaciones `Naive` y `Strassen`, dos alternativas que ilustran la diferencia entre un método directo y uno que busca reducir la complejidad asintótica a costa de mayor uso de memoria.

El objetivo principal es contrastar lo que se espera en teoría con lo que realmente ocurre al ejecutar los algoritmos sobre conjuntos de datos generados bajo distintos escenarios. De esta forma, se busca mostrar que la elección de un algoritmo no depende solo de su complejidad matemática, sino también de las condiciones específicas del problema y de los recursos disponibles en la práctica.

2. Implementaciones

La extensión máxima para esta sección es de 1 página.

Sólo agregar la url del repositorio de GitHub donde se encuentra el código fuente de la tarea. Recordar que el repositorio **debe ser privado**, ya que de lo contrario cualquier persona podrá acceder a su código y cometer plagio, siendo usted responsable de ello.

https://github.com/INF221-20252/tarea-1-RMiguel0.git

3. Experimentos

> "*Non-reproducible single occurrences are of no significance to science.*"
>
> —Popper, 2005 [1]

Para esta experiencia se utilizó el siguiente hardware: un CPU Ryzen 5 5600X a 3.60 GHz, 16 GB de memoria RAM DDR4, y un disco sólido SSD M.2 PCIe NVMe. El ambiente de desarrollo consistió en *Visual Studio Code* sobre Windows 11, mientras que la ejecución se llevó a cabo en un entorno virtual Ubuntu WSL 24.04.3 LTS, compilando con `g++` versión 13.3.0.

En cuanto a las herramientas utilizadas, se emplearon diferentes librerías de C++ tanto estándar como específicas del entorno UNIX/Linux, entre las cuales destacan:

- `<functional>`: Permite definir funciones anónimas ("lambda") y almacenarlas en estructuras. En el código se utiliza para declarar el vector de algoritmos y ejecutarlos de manera genérica, independientemente de su firma particular.
- `<chrono>`: Proporciona mediciones precisas de tiempo mediante el uso de `high_resolution_clock`. En este trabajo se empleó para registrar el tiempo de ejecución (en milisegundos) de cada algoritmo.
- `<sys/resource.h>`: Ofrece acceso a recursos del sistema (CPU, memoria, entre otros). En nuestro caso, se utilizó en conjunto con el sistema de archivos `/proc` de Linux para medir el uso de memoria de los procesos en ejecución.
- `<unistd.h>`: Permite obtener información del sistema operativo, en particular el tamaño de página de memoria mediante `getpagesize()`, dato esencial para convertir las mediciones de memoria del archivo `/proc` a kilobytes.

De manera general, ambos programas desarrollados (`sorting.cpp` y `matrix_multiplication.cpp`) siguen una misma lógica. En primer lugar, recorren las carpetas de entrada correspondientes, las cuales contienen los archivos generados previamente mediante *scripts* de Python. Posteriormente, los datos son cargados en estructuras adecuadas: vectores en el caso del ordenamiento y matrices en el caso de la multiplicación. La función `record` es la encargada de recibir estos datos, ejecutar los algoritmos definidos y registrar sus resultados experimentales.

Dentro de `record` se declaran los algoritmos utilizando funciones "lambda". Esto permite estandarizar la forma de invocar cada implementación, aun cuando difieren en su interfaz original, garantizando así una ejecución uniforme y reduciendo posibles sesgos en las mediciones.

En cuanto a las limitaciones observadas, fue necesario restringir algunos casos de prueba. En el ordenamiento, los algoritmos `quickSort` e `insertionSort` solo se ejecutaron hasta un tamaño máximo de 10^5 elementos. Esto se debió a que, para instancias mayores, `quickSort` producía errores de `Segmentation fault (Core dumped)`, mientras que `insertionSort` presentaba tiempos de ejecución excesivamente largos que lo volvían impráctico.

En contraste, los algoritmos de multiplicación de matrices no presentaron este tipo de inconvenientes, siendo estables incluso para tamaños de matriz mayores, aunque con tiempos de ejecución relativamente altos.

3.0.1. Metodología general

Los inputs fueron generados automáticamente mediante *scripts* de Python provistos en el enunciado. Según cada problema se recorren los directorios;

- **Sorting**: Archivos con listas de enteros.
- **Multiplicación de matrices**: Archivos con matrices cuadradas.

Luego la función `record` recibe los datos y ejecuta todos los algoritmos definidos. Esta función constituye el núcleo de la experiencia, pues concentra la lógica de ejecución, medición y almacenamiento de resultados.

En el caso de la multiplicación de matrices, una vez que se cargan las matrices con `leer_matriz`, se extrae el nombre del archivo y se normaliza para usarlo como etiqueta del experimento. Por ejemplo, los inputs *1024_dispersa_D10_c_1.txt* y *1024_dispersa_D10_c_2.txt* producen como salida un archivo en `data/matrix_output/` llamado *1024_dispersa_D10_c_Naive.txt* si el algoritmo ejecutado fue `Naive`, que contiene la matriz resultante de la multiplicación. De manera análoga, en `data/measurements/` se genera un archivo *1024_dispersa_D10_c_multiplication_measurements.txt* donde se almacenan los tiempos de ejecución en milisegundos y las estimaciones de uso de memoria para cada algoritmo (Naive y Strassen).

Para garantizar equidad experimental, ambos algoritmos se encapsulan en funciones "lambda" con la misma firma, lo que permite generalizar su ejecución en un mismo bucle. Adicionalmente, se registran tanto mediciones empíricas (RSS del proceso antes y después) como estimaciones teóricas de memoria según la complejidad espacial de cada algoritmo. Esto asegura consistencia entre ejecuciones y otorga un marco más riguroso para comparar resultados.

El comportamiento es análogo en el caso de ordenamiento de arreglos: la función `record` recibe el archivo de entrada, lo carga en un vector, identifica el tamaño del arreglo y ejecuta cada algoritmo definido (InsertionSort, MergeSort, QuickSort, entre otros) sobre una copia idéntica de los datos. Los resultados ordenados se guardan en `data/array_output/`, mientras que los tiempos y consumos de memoria se registran en `data/measurements/`.

Cabe destacar que en el caso de los algoritmos de ordenamiento, la función `record` distingue entre aquellos que son in-place (InsertionSort, QuickSort, SortStd), cuyo consumo adicional de memoria es despreciable, y aquellos que requieren memoria auxiliar (MergeSort con $O(n)$ o PandaSort con $O(\sqrt{n})$), donde se hace un ajuste explícito para reflejar la complejidad teórica. De esta forma, los resultados obtenidos no sólo registran el comportamiento observado, sino que también incorporan la naturaleza propia de cada algoritmo.

Este ajuste explícito consiste en que, en el caso de MergeSort y PandaSort, la medición de memoria reportada combina dos componentes: (i) la variación observada en el RSS del proceso antes y después

de la ejecución, y (ii) un ajuste teórico que refleja la complejidad espacial propia de cada algoritmo ($O(n)$ para MergeSort y $O(\sqrt{n})$ para PandaSort). De esta forma, los gráficos no representan únicamente el uso real de memoria en tiempo de ejecución, sino una estimación ajustada que busca reflejar de manera más fiel el comportamiento esperado según su análisis teórico.

En el caso particular del algoritmo de Strassen, fue necesario realizar un ajuste explícito en la estimación de memoria. A diferencia del método `Naive`, que requiere únicamente tres matrices de tamaño $n \times n$ (dos de entrada y una de salida), Strassen divide las matrices en cuatro bloques y realiza siete multiplicaciones de sub-bloques en lugar de las ocho que se harían con el enfoque estándar. Sin embargo, este ahorro en operaciones aritméticas se consigue a costa de crear matrices temporales adicionales en cada nivel recursivo, tanto para las sumas y restas intermedias como para almacenar los productos parciales $M_1, \ldots, M_7$.

Si se analiza un nivel de la recursión, los arreglos temporales ocupan $O(n^2)$ espacio adicional. Dado que en el siguiente nivel se trabaja con submatrices de tamaño $n/2 \times n/2$, el coste extra en memoria se reduce a $O((n/2)^2) = O(n^2/4)$, y así sucesivamente en cada paso de la recursión. Esto conduce a la serie geométrica

$$n^2 \left(1 + \tfrac{1}{4} + \tfrac{1}{16} + \cdots\right),$$

cuyo valor converge a $\frac{4}{3}n^2$. Es decir, Strassen mantiene el mismo orden asintótico $O(n^2)$ en consumo de memoria que el algoritmo Naive, pero con una constante mayor, aproximadamente un 33 % adicional.

En la implementación de este trabajo, este análisis se refleja aplicando un factor de corrección de $\frac{4}{3}$ a la estimación de memoria base B, de modo que

$$M_{\text{Strassen}} \approx \tfrac{4}{3} B,$$

lo cual constituye un modelo más fiel al comportamiento real del algoritmo que asumir simplemente el doble de memoria.

3.1. Dataset (casos de prueba)

Los casos de prueba considerados en esta tarea fueron generados a partir de los programas oficiales entregados, y se dividen en dos familias: *ordenamiento de arreglos unidimensionales* y *multiplicación de matrices cuadradas*. Cada instancia se codifica mediante un nombre de archivo de la forma `{n}_{t}_{d}_{m}.txt`, donde cada parámetro controla una característica del dataset y, en consecuencia, influye en el comportamiento y las mediciones de los algoritmos evaluados.

Ordenamiento de arreglos. Los arreglos se definen por cuatro parámetros:

- n: número de elementos, con valores en $\{10^1, 10^3, 10^5, 10^7\}$. El tamaño del arreglo incide directamente en la complejidad temporal: algoritmos cuadráticos como `InsertionSort` se vuelven impracticables en 10^7 elementos, mientras que algoritmos de orden $O(n \log n)$ (como `MergeSort` o `Quicksort`) mantienen tiempos manejables.

- t: tipo de orden inicial (*ascendente, descendente, aleatorio*). Este parámetro afecta de manera crítica a algoritmos dependientes del orden inicial, como `InsertionSort` (muy eficiente en arreglos casi ordenados, pero costoso en descendentes).
- d: dominio de los valores ($D1 = \{0,\ldots,9\}$ o $D7 = \{0,\ldots,10^7\}$). Dominios pequeños generan muchos valores repetidos, lo cual impacta la estabilidad y las comparaciones internas del algoritmo.
- m: muestra aleatoria (*a,b,c*), que permite obtener variabilidad en los casos aún con los mismos parámetros, asegurando que los resultados no dependan de una única instancia no representativa.

Multiplicación de matrices. En este caso, cada par de archivos define dos matrices cuadradas M_1, M_2 cuyos parámetros son:

- n: dimensión de la matriz, con valores en $\{2^4, 2^6, 2^8, 2^{10}\}$. El tamaño domina tanto el tiempo como la memoria: mientras `Naive` ejecuta $O(n^3)$ operaciones, `Strassen` reduce a $O(n^{\log_2 7}) \approx O(n^{2,81})$, pero con sobrecosto en memoria.
- t: tipo de matriz (*dispersa, diagonal, densa*). Matrices dispersas o diagonales reducen el número de operaciones efectivas, lo que se traduce en tiempos más bajos que en el caso denso, donde prácticamente todos los elementos participan en los cálculos.
- d: dominio de valores ($D0 = \{0,1\}$ o $D10 = \{0,\ldots,9\}$). Un dominio más amplio genera más diversidad numérica y menor probabilidad de estructuras degeneradas (ej. filas nulas), lo que puede impactar tanto en la dificultad del cálculo como en la validez de comparaciones entre algoritmos.
- m: muestra aleatoria (*a,b,c*), que permite disponer de tres variantes de cada configuración, evitando sesgos por un caso aislado y favoreciendo un análisis estadístico más robusto.

En conjunto, estos parámetros aseguran una batería de pruebas diversa que cubre desde instancias pequeñas y estructuradas hasta entradas de gran escala y aleatorias, lo cual permite evaluar no sólo la complejidad teórica de los algoritmos, sino también su rendimiento práctico bajo diferentes condiciones.

3.2. Resultados

En esta sección presentamos los resultados experimentales obtenidos para los dos problemas estudiados: *ordenamiento de arreglos* y *multiplicación de matrices*.

Reproducibilidad. Los datos de medición (tiempo y memoria) se guardan en la carpeta `data/measurements/` de cada módulo, y las salidas (arreglos o matrices) en `data/array_output/` y `data/matrix_output/`. Los gráficos se generan con el script `plot_generator.py` y se almacenan en `data/plots/` dentro de cada módulo (`sorting` y `matrix_multiplication`).

Ordenamiento de arreglos

Tiempo. En la Figura 1 (izquierda) se observa que `std::sort` es sistemáticamente el más rápido en todo el rango de n, seguido de `MergeSort` y `QuickSort` (cuando alcanza a ejecutarse). La tendencia de estas tres curvas es coherente con $O(n \log n)$. `InsertionSort` exhibe el crecimiento cuadrático esperado, volviéndose impracticable para $n \geq 10^5$. `PandaSort` escala peor que los $O(n \log n)$, con tiempos sensiblemente mayores en tamaños grandes.

Memoria. En la Figura 1 (derecha) se aprecia que `InsertionSort`, `QuickSort` y `std::sort` son casi *in-place* (overhead adicional despreciable), mientras que `MergeSort` presenta el consumo lineal $O(n)$ que exige su arreglo auxiliar. `PandaSort` muestra un consumo notoriamente superior en n grandes, consistente con su diseño con estructuras auxiliares adicionales.

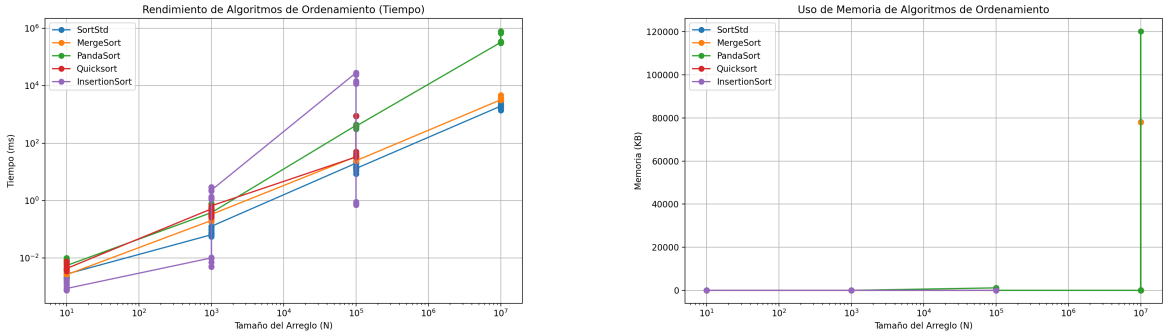

Figura 1: Ordenamiento: (izq.) tiempo de ejecución vs. tamaño del arreglo; (der.) memoria adicional usada.

Multiplicación de matrices

Tiempo. La Figura 2 (izquierda) muestra que el método `Naive` resulta más rápido que `Strassen` en los tamaños evaluados (hasta $n = 2^{10}$). Pese a la mejor complejidad asintótica de Strassen ($O(n^{\log_2 7}) \approx O(n^{2,81})$), en nuestra escala finita su *overhead* domina: muchas sumas/restas de subbloques, llamadas recursivas y manejo de temporales elevan las constantes ocultas y penalizan la localidad de caché. Sin un umbral híbrido para conmutar a `Naive` en subproblemas pequeños, el punto de cruce donde Strassen superaría a `Naive` no se alcanza en esta infraestructura.

Memoria. En la Figura 2 (derecha) `Strassen` utiliza más memoria que `Naive`. Aunque ambos conservan orden $O(n^2)$ en espacio, Strassen requiere temporales por nivel recursivo (sumas/restas y productos parciales), lo que incrementa la constante asociada al término cuadrático. Esto se refleja empíricamente en una brecha consistente a favor de `Naive`.

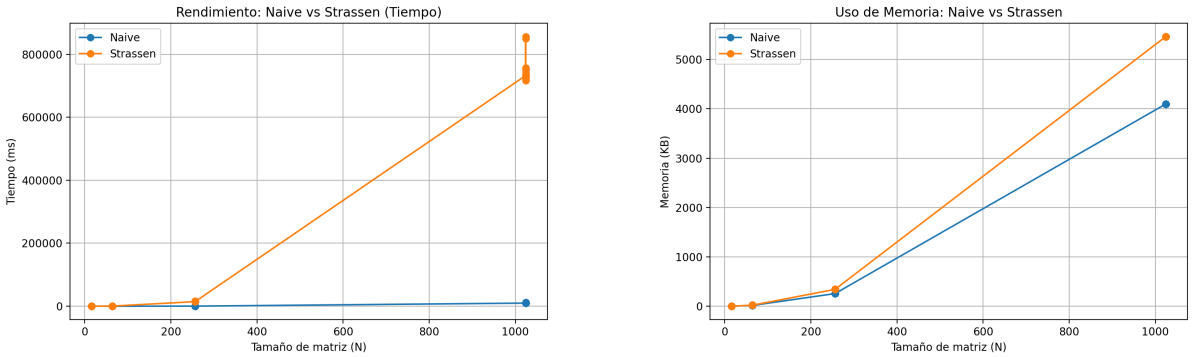

Figura 2: Multiplicación de matrices: (izq.) tiempo de ejecución; (der.) memoria adicional.

4. Conclusiones

El análisis experimental permitió contrastar las complejidades teóricas de los algoritmos con su desempeño práctico. En el caso del ordenamiento, los resultados confirmaron la superioridad de los algoritmos $O(n \log n)$ frente a los cuadráticos: `InsertionSort` se volvió impracticable en instancias grandes, mientras que `MergeSort`, `QuickSort` y especialmente `std::sort` mostraron tiempos eficientes y un uso de memoria acorde a sus diseños. El algoritmo `PandaSort`, en cambio, evidenció un comportamiento inferior, lo que subraya la dificultad de competir con métodos clásicos ya optimizados.

En cuanto a la multiplicación de matrices, el algoritmo `Naive` resultó más eficiente en los tamaños considerados, a pesar de su complejidad asintótica $O(n^3)$. Esto se debe a que `Strassen`, aunque teóricamente más rápido ($O(n^{2,81})$), introduce un costo adicional significativo en operaciones intermedias y uso de memoria, lo que impide que su ventaja teórica se materialice en escalas moderadas. Además, la necesidad de múltiples matrices temporales genera un sobrecosto en espacio que se refleja empíricamente en los gráficos.

En síntesis, los experimentos evidencian que la complejidad asintótica no basta para predecir el rendimiento real de un algoritmo: las constantes ocultas, el manejo de memoria y los detalles de implementación son determinantes en la práctica. Los resultados cumplen con el objetivo de la tarea al conectar teoría y práctica, mostrando cómo los límites del análisis asintótico se revelan en escenarios experimentales concretos.

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] K. Popper. *The Logic of Scientific Discovery*. Routledge Classics. Taylor & Francis, 2005. ISBN: 9781134470020. URL: https://books.google.cl/books?id=LWSBAgAAQBAJ.