REPORTE TAREA 1
ALGORITMOS Y COMPLEJIDAD

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

Javier Miranda Lanyon

28 de abril de 2025

20:18

Resumen

Este trabajo presenta un análisis experimental del rendimiento de algoritmos clásicos de ordenamiento (sort, selection sort, quicksort, mergesort) y multiplicación de matrices (naive y Strassen). Se utilizaron benchmarks cuidadosamente generados y un ambiente controlado de pruebas en un computador Ryzen 3 3200G con 16 GB de RAM ejecutando Ubuntu 24.04.2 LTS. Las implementaciones fueron compiladas bajo la norma C++17. Se presentan resultados de tiempo de ejecución y uso de memoria, visualizados mediante gráficos, permitiendo comparar el comportamiento de cada algoritmo bajo distintas condiciones de entrada. Se concluye con una interpretación crítica de los resultados.

Índice

1. Introducción

El presente informe se enmarca en el campo del **Análisis y Diseño de Algoritmos** en Ciencias de la Computación, disciplina fundamental que estudia tanto la eficiencia teórica como el comportamiento práctico de las soluciones algorítmicas a problemas computacionales. El estudio de algoritmos permite optimizar recursos computacionales críticos, como el tiempo de ejecución y el consumo de memoria, elementos esenciales en el desarrollo de aplicaciones eficientes a gran escala. Durante décadas, investigaciones clásicas como las de Cormen et al. han establecido las bases del análisis asintótico; sin embargo, en la práctica, el rendimiento real puede diferir significativamente del comportamiento predicho, especialmente cuando se consideran factores como la estructura de los datos, el entorno de ejecución o los costos de acceso a memoria.

En este contexto, surge una **brecha** relevante entre el análisis teórico y el comportamiento experimental de los algoritmos. Si bien la notación asintótica (por ejemplo, $\mathcal{O}(n \log n)$ o $\mathcal{O}(n^2)$) proporciona información sobre la tendencia de crecimiento para entradas grandes, no siempre refleja de manera precisa la eficiencia en tamaños de entrada típicos en aplicaciones reales. Este informe busca precisamente explorar esa diferencia, enfocándose en algoritmos de **ordenamiento** (Merge Sort, Quick Sort y Selection Sort) y **multiplicación de matrices** (Naive y Strassen), evaluando su comportamiento práctico frente a distintos tipos de matrices (densas, diagonales y dispersas).

El propósito principal de este estudio es comparar experimentalmente estos algoritmos en términos de **tiempo de ejecución** y **consumo de memoria**, utilizando un entorno controlado que permita garantizar la **reproducibilidad** de los resultados. A diferencia del tratamiento meramente teórico abordado en clases, aquí se pretende validar, mediante experimentación sistemática, cómo se manifiestan en la práctica las diferencias de eficiencia entre algoritmos, y en qué condiciones un algoritmo puede resultar preferible a otro.

2. Implementaciones

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

- **Sort:** Utilizando `std::sort` sacado de la biblioteca de C++
- **Selection Sort:** Implementación clásica basada en selección de mínimo
- **Quicksort:** Versión iterativa optimizada con median-of-three y insertion sort
- **Mergesort:** Implementación con control explícito de memoria
- **Multiplicación Naive:** Triple bucle anidado
- **Multiplicación Strassen:** Método de divide y vencerás basado en 7 productos

3. Experimentos

- **Procesador:** AMD Ryzen 3 3200G
- **Memoria RAM:** 16 GB DDR4 3000 Mhz
- **Almacenamiento:** SSD 512 GB
- **Sistema Operativo:** Ubuntu 24.04.2 LTS
- **Compilador:** g++ con flags `-std=c++17`

El formato de entrada para los casos de algoritmos de ordenamiento consistió en distintos arreglos almacenados en archivos `.txt`, generados mediante el código provisto por los ayudantes.

Para los algoritmos de multiplicación de matrices, el formato de entrada consistió en dos archivos `.txt` que contienen las matrices correspondientes, también generados con el código entregado por los ayudantes.

La ejecución de los programas se realizó utilizando un `Makefile`, compilando con el comando `make` en un entorno Linux (como WSL o consola nativa), y posteriormente ejecutando los binarios generados: `./sorting` para los algoritmos de ordenamiento, y `./matrix_multiplication` para los algoritmos de matrices.

3.1. Dataset (casos de prueba)

Se utilizaron los datasets provistos en el enunciado de la tarea, generados mediante scripts de Python. Estos incluyen:

- **Ordenamiento:** entradas de tamaño 10, 1000 y 100000 con distribuciones aleatoria, ascendente y descendente.
- **Multiplicación de matrices:** matrices de tamaño 16, 64, 256 y 1024, con patrones densos, dispersos y diagonales.

Se tuvieron que borrar los archivos de sorting de 10000000 ya que simplemente el tiempo de ejecucion de cada programa era muy alto.

3.2. Resultados

3.3. Guía para replicar el benchmark

A continuación, se describe el procedimiento paso a paso para replicar el mismo benchmark utilizando el código proporcionado en el repositorio de GitHub:

1. Ejecutar los algoritmos de ordenamiento y multiplicación de matrices, siguiendo las instrucciones detalladas en la sección de Experimentos. La compilación se realiza mediante `make` y la ejecución mediante los binarios generados (`./sorting` y `./matrix_multiplication`).

2. Una vez finalizada la ejecución de los algoritmos, se generarán automáticamente los archivos `bench.txt` que contienen los resultados de medición (tiempo y memoria), los cuales se almacenan en la carpeta `Data/Measurements`.

3. Posteriormente, acceder a la carpeta de scripts, específicamente al directorio `plot_generator`, y ejecutar el script de Python correspondiente fijarte que la direccion este bien puesta para la generación de gráficos. Este paso puede realizarse directamente desde la consola, siempre que los datos de medición estén disponibles.

4. Las imágenes generadas se almacenarán automáticamente en el directorio `Data/plots`, listas para su inclusión en el informe final.

3.3.1. Algoritmos de Ordenamientos

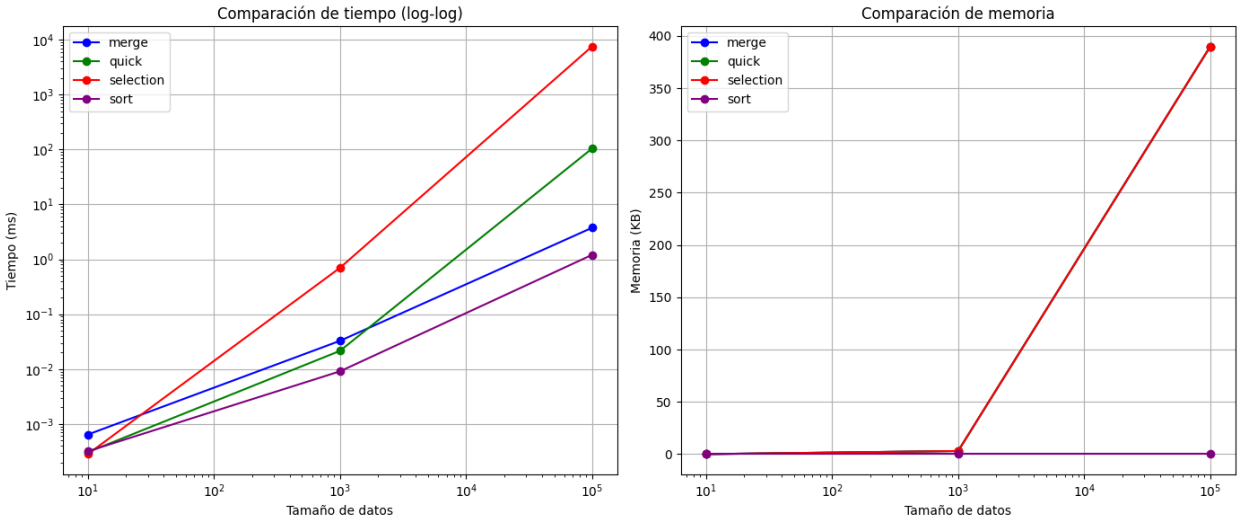

Figura 1: Tiempos de ejecución y memoria para algoritmos de ordenamiento.

Se observa que **Selection Sort** es el algoritmo que presenta el mayor crecimiento en el tiempo de ejecución a medida que aumenta el tamaño de la entrada, lo cual es coherente con su complejidad temporal de orden $\mathcal{O}(n^2)$. Esta tendencia se hace especialmente evidente en entradas de gran tamaño, donde su desempeño se vuelve considerablemente ineficiente en comparación con otros algoritmos.

En contraste, los algoritmos **QuickSort**, **std::sort** (de la biblioteca estándar de C++), y **MergeSort** muestran un comportamiento mucho más eficiente, gracias a sus complejidades temporales promedio de $\mathcal{O}(n \log n)$. Entre ellos, **QuickSort** exhibe una ligera desventaja en ciertos casos particulares, dado que puede alcanzar su peor caso de complejidad $\mathcal{O}(n^2)$ en situaciones específicas, como entradas ya ordenadas o altamente desbalanceadas, si no se implementan estrategias de pivoteo adecuadas.

No obstante, en el escenario general evaluado, **QuickSort** y **std::sort** se comportaron de manera muy competitiva, validando su aplicabilidad práctica en una amplia gama de problemas de ordenamiento. De manera destacada, **std::sort** se consolidó como el algoritmo más eficiente en nuestras mediciones, superando ligeramente a **MergeSort**. Esta ventaja se debe a que `std::sort` implementa una combinación optimizada de `QuickSort`, `HeapSort` y `InsertionSort`, conocida como *Introsort*, lo que le permite adaptarse dinámicamente al patrón de la entrada y minimizar el tiempo de ejecución en la mayoría de los casos prácticos. Aunque **MergeSort** ofrece una garantía teórica de $\mathcal{O}(n \log n)$ en todos los casos, su sobrecarga de memoria adicional y operaciones de copia lo hacen marginalmente menos competitivo que **std::sort** en ambientes de alto rendimiento como el evaluado.

En cuanto al consumo de memoria, se observa que todos los algoritmos de ordenamiento analizados (**std::sort**, **Selection Sort**, **QuickSort** y **MergeSort**) presentan un uso muy acotado de memoria adicional respecto al tamaño de las entradas procesadas.

De acuerdo a las mediciones obtenidas, para instancias de tamaño pequeño (10 a 1000 elementos), la memoria utilizada se mantuvo en torno a los 0 a 3 KB, mientras que para entradas de tamaño 100,000 elementos, el consumo de memoria alcanzó aproximadamente los 390 KB. Estos valores reflejan un comportamiento prácticamente lineal en la relación entre el tamaño del input y la memoria ocupada.

El **std::sort** de la biblioteca estándar de C++ destacó por su mínima utilización de memoria adicional, aprovechando que su implementación basada en *Introsort* opera principalmente de manera *in-place* en memoria, requiriendo sólo una sobrecarga constante.

En contraste, **MergeSort** presentó un consumo similar de memoria, aunque su naturaleza intrínseca requiere la creación de arreglos temporales para realizar las operaciones de mezcla, lo que podría penalizar su eficiencia espacial en aplicaciones donde el uso de memoria sea crítico.

Por su parte, **Selection Sort** y **QuickSort** también mantuvieron un perfil de consumo de memoria bajo, acorde a sus implementaciones clásicas que, en su mayoría, modifican el arreglo original sin recurrir a estructuras auxiliares adicionales.

En general, los resultados experimentales evidencian que todos los algoritmos implementados son altamente eficientes en términos de uso de memoria, manteniéndose dentro de márgenes prácticos para sistemas modernos incluso con volúmenes de datos elevados.

Para una mejor visualización de las diferencias en el comportamiento de los algoritmos de ordenamiento en términos de tiempo de ejecución y consumo de memoria, se presentan gráficos individuales en el Apéndice (ver Figuras **??** a **??**). Esta separación permite observar de manera más clara cómo varía la eficiencia de `std::sort`, `QuickSort`, `Selection Sort` y `MergeSort` conforme aumenta el tamaño de la entrada.

3.3.2. Multiplicación de Matrices

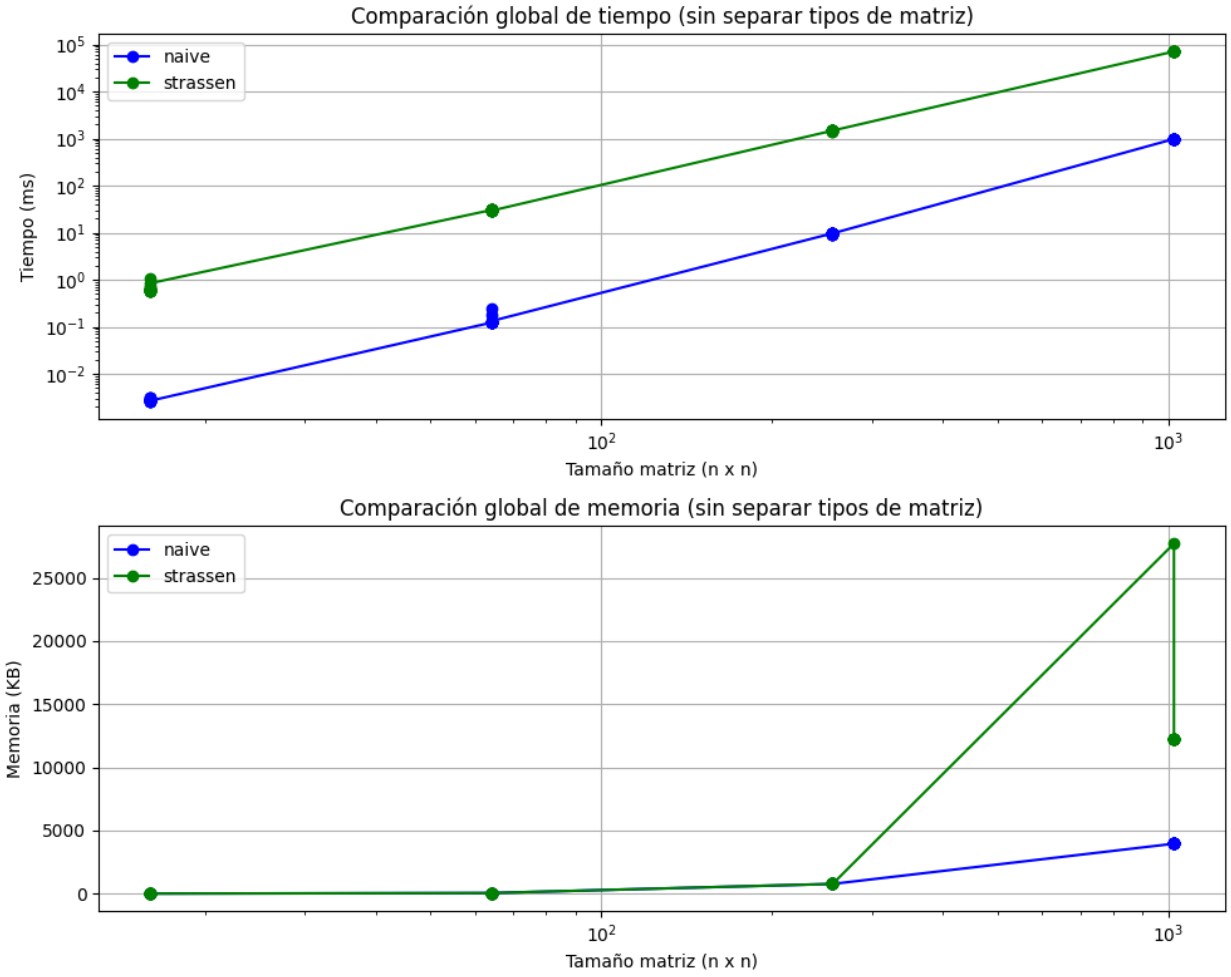

Figura 2: Comparación de tiempos y memoria para métodos de multiplicación de matrices.

En el caso de los algoritmos de multiplicación de matrices, se observaron diferencias notables tanto en consumo de memoria como en tiempo de ejecución entre el método **Naive** y el método de **Strassen**.

En cuanto al consumo de memoria, el algoritmo **Naive**, basado en el enfoque tradicional de triple bucle anidado, presentó un uso muy controlado de recursos. Para matrices de tamaño 1024×1024, el consumo de memoria fue de aproximadamente 3956 KB, mientras que en tamaños pequeños como 16×16 se mantuvo en apenas 3 KB. Esto se debe a que **Naive** trabaja directamente sobre los arreglos de entrada y el arreglo resultado, sin necesidad de generar estructuras auxiliares adicionales.

Por otro lado, el método de **Strassen** evidenció un consumo de memoria mucho mayor, alcanzando alrededor de 12,288 KB para matrices de 1024×1024. Esto se explica porque **Strassen** requiere múltiples submatrices en cada nivel de recursión para realizar las combinaciones de sumas, restas y productos parciales, lo que incrementa sustancialmente su demanda de memoria.

Respecto al tiempo de ejecución, se registró que **Naive** fue consistentemente más rápido que **Strassen** en todos los tamaños de matrices analizados. Por ejemplo, para instancias de 1024×1024, **Naive** completó

las operaciones en aproximadamente 970 ms, mientras que **Strassen** tardó más de 70,000 ms (alrededor de 70 segundos), evidenciando una diferencia de dos órdenes de magnitud.

Este resultado contrasta con la expectativa teórica de que **Strassen**, con una complejidad asintótica de $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2,81})$, debería superar a **Naive** ($\mathcal{O}(n^3)$) para tamaños de matrices muy grandes. Sin embargo, debido al considerable overhead de operaciones adicionales (sumas y restas de submatrices) y gestión intensiva de memoria intermedia, el algoritmo **Strassen** no siempre se ve beneficiado por las arquitecturas modernas de los computadores. Esto se debe a la sobrecarga en operaciones de suma y el manejo de submatrices, que no siempre aprovechan de manera óptima el acceso eficiente a la memoria cache, ni se adaptan fácilmente a las técnicas de paralelización, factores que juegan un papel fundamental en el rendimiento práctico en hardware contemporáneo.

En resumen, para los tamaños de entrada considerados en este informe, **Naive** no solo fue más eficiente en consumo de memoria, sino que además obtuvo tiempos de ejecución notablemente menores que **Strassen**, posicionándose como la opción preferente en escenarios de matrices de tamaño moderado.

4. Conclusiones

Los resultados experimentales obtenidos a lo largo de este informe permiten reflexionar de manera crítica sobre la relación entre la complejidad teórica de los algoritmos y su comportamiento práctico en ambientes computacionales reales.

En primer lugar, en el ámbito de los algoritmos de ordenamiento, se corroboró que las estrategias optimizadas, como `std::sort` y `MergeSort`, mantienen un desempeño altamente eficiente en términos de tiempo de ejecución y consumo de memoria para una amplia gama de tamaños de entrada. El análisis también evidenció que, aunque teóricamente `QuickSort` ofrece buenas prestaciones promedio, en ciertos casos específicos puede degradar su rendimiento, comportamiento que resulta relevante para aplicaciones críticas donde el control del peor caso es prioritario.

De manera destacada, la experiencia empírica reafirmó la importancia de considerar el impacto de la implementación y del entorno de ejecución: `std::sort`, gracias a su estrategia híbrida (*Introsort*), logró superar marginalmente a `MergeSort`, demostrando que en la práctica pequeñas optimizaciones pueden resultar decisivas en escenarios de alto rendimiento.

Respecto a los algoritmos de multiplicación de matrices, los experimentos subrayan que la complejidad asintótica, aunque esencial para un análisis teórico, no garantiza necesariamente un mejor desempeño práctico en contextos finitos. El algoritmo `Naive`, pese a su complejidad $\mathcal{O}(n^3)$, superó ampliamente a `Strassen` en tiempos de ejecución y eficiencia en el uso de memoria para los tamaños de matrices considerados. Este fenómeno se explica por el considerable overhead de operaciones adicionales en `Strassen`, y por el hecho de que las arquitecturas modernas de computadores favorecen algoritmos con patrones de acceso a memoria más lineales y simples.

En suma, los hallazgos de este estudio refuerzan la noción de que el análisis experimental es una herramienta indispensable para validar y contextualizar las predicciones teóricas de los algoritmos. Además, resaltan la importancia de considerar no sólo la complejidad asintótica, sino también factores prácticos como la eficiencia espacial, el acceso a memoria, y las características de la infraestructura, a la hora de seleccionar algoritmos para aplicaciones reales.

A. Apéndice 1

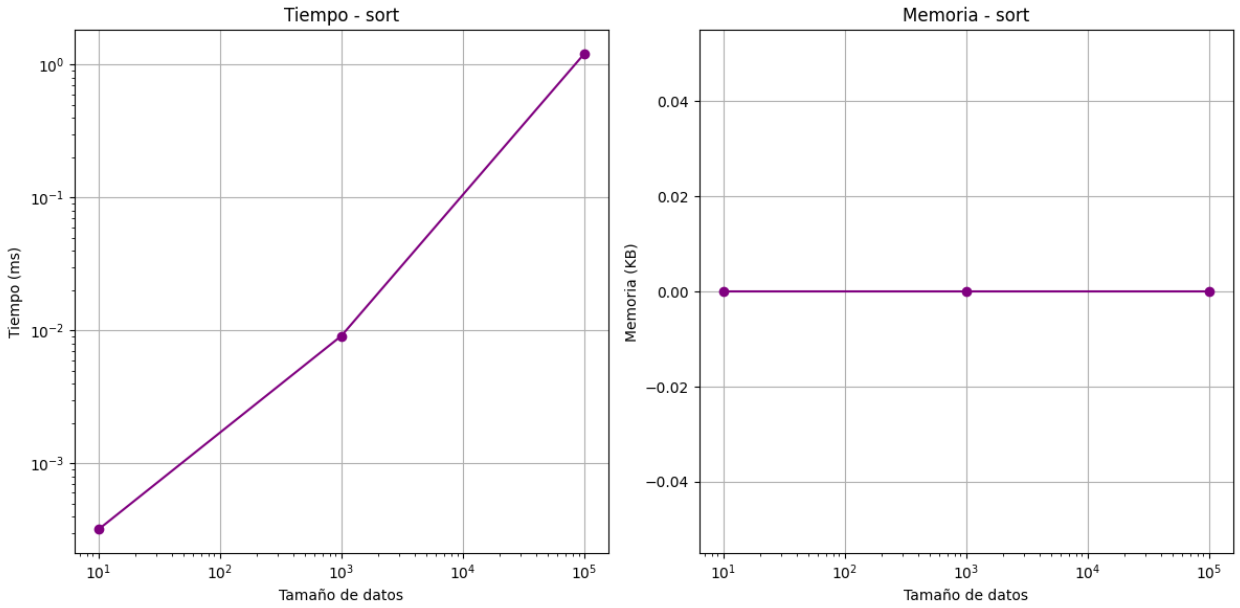

Figura 3: Tiempos de ejecución y memoria de Std::sort.

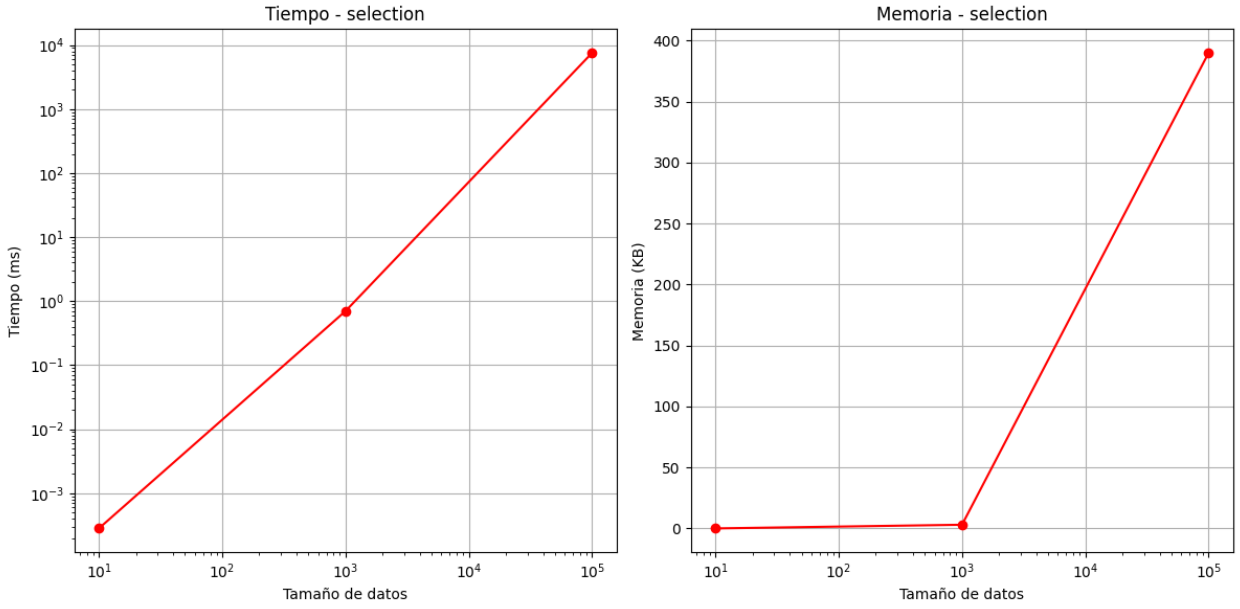

Figura 4: Tiempos de ejecución y memoria de Selection Sort.

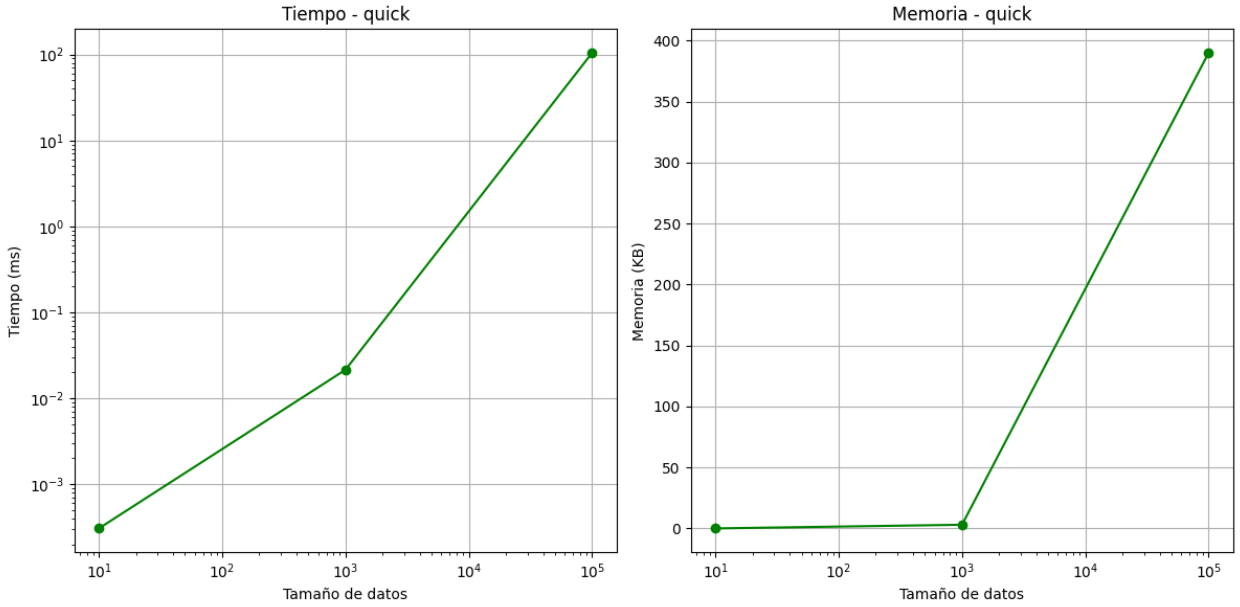

Figura 5: Tiempos de ejecución y memoria de Quick Sort.

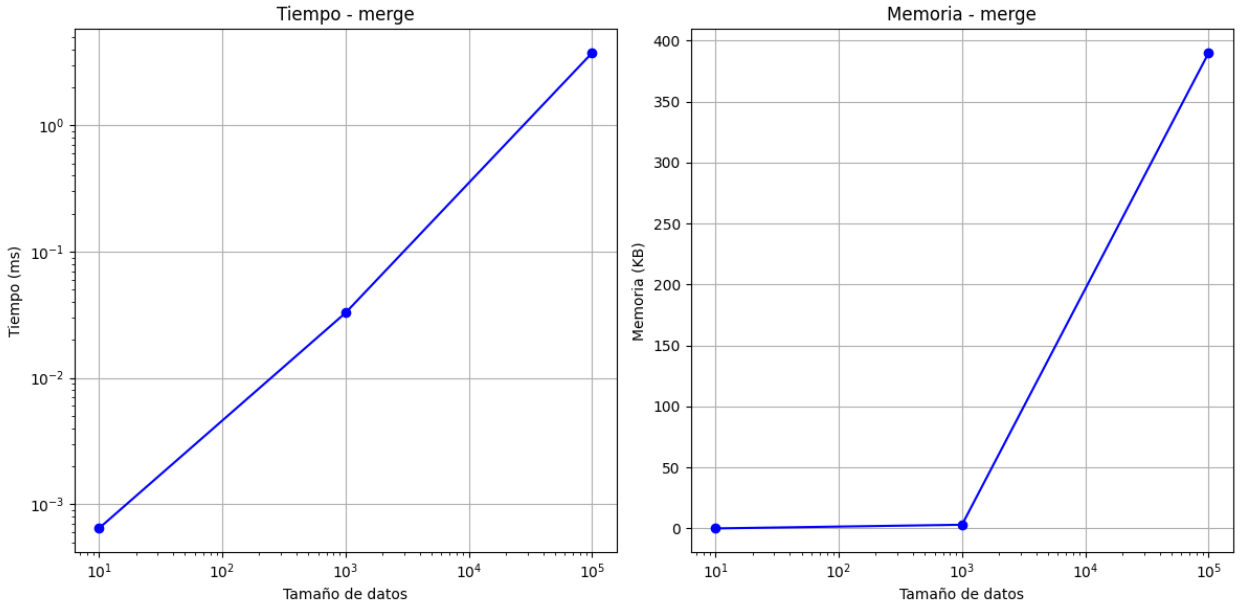

Figura 6: Tiempos de ejecución y memoria de Merge Sort