REPORTE TAREA 1
ALGORITMOS Y COMPLEJIDAD

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

Benjamín Daza Jiménez

27 de abril de 2025

21:20

Resumen

Se realizará un análisis mediante la implementación de algoritmos de ordenamiento y algoritmos de multiplicación de matrices, utilizando distintos datasets para ejecutar pruebas y obtener resultados comparativos. ¿Realmente existe una diferencia entre estas implementaciones? Los resultados se presentarán mediante gráficos que permitan visualizar claramente las diferencia entre los algoritmos analizados. Finalmente, este análisis permitirá obtener una visión más objetiva de cómo influyen estos algoritmos en operaciones que puedan ser comunes en distintos contextos computacionales.

Índice

1. Introducción **2**

2. Implementaciones **3**

3. Experimentos **4**

4. Conclusiones **10**

A. Apéndice 1 **11**

1. Introducción

La eficiencia de los algoritmos es un factor fundamental para el rendimiento de las aplicaciones. El objetivo de este informe es implementar cuatro tipos de ordenamiento y dos tipos de multiplicación de matrices, con el fin de contrastar su rendimiento en distintas pruebas exhaustivas con conjutos de datos de tamaño variable, teniendo una visión completa de cada algoritmo.
Los algoritmos considerados para el análisis de ordenamiento son:

- Selection Sort: algoritmo que ordena una lista buscando el menor (o mayor) elemento de cada pasada, colocándolo en su posición correcta.
- Merge Sort: algoritmo que divide el arreglo en mitades iguales hasta que queden subarreglos de 1 elemento, los fusiona en orden, comparando pares de elementos.
- Quick Sort: algoritmo que elige un pivote, luego reordena el arreglo dejando a los elementos menores al pivote a la izquierda y los mayores a la derecha. Se aplica recursivamente a los subarreglos.
- Función Sort de c++: Combina Quick Sort, Heap Sort e Insertion Sort, adaptándose al tipo de dato.

En cuanto a la multiplicación de matrices se evaluará:

- Algoritmo de Naive: multiplica matrices siguiendo la definición clásica de la multiplicación.
- Algoritmo de Strassen: divide cada matriz en 4 submatrices. Calcula 7 productos especiales y los usa para construir la matriz resultado.

Se espera que los tiempos de ejecución de la función Sort de c++, Merge Sort y Quick Sort sean similares en la mayoría de los casos, ya que estos algoritmos comparten una complejidad promedio de O(n log n). Sin embargo, se anticipa un tiempo mucho mayor para el algoritmo de Selection sort, debido a su complejidad cuadrática. Para la multiplicación de matrices, se espera el algoritmo de Strassen sea más rápido que el algoritmo de Naive, ya que posee una complejidad menor al cúbico.
Se busca comprobar que los algoritmos de ordenamiento y de multiplicación de matrices presentan diferencias significativas en sus tiempos de ejecución, especialmente al trabajar con datasets de gran tamaño. Esto permitirá evidenciar cómo la eficiencia algorítmica impacta directamente en el rendimiento computacional a mayor escala.

2. Implementaciones

En el siguiente url se ecnuentra el repositorio con los algoritmos trabajados.

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

3. Experimentos

Los casos de prueba realizados fueron ejecutados en un notebook con procesador Intel Core i5-13420H a 2.10GHz, 16GB de memoria RAM DDR4 y almacenamiento SSD NVMe. Las pruebas se realizaron en un entorno de Windows 11 versión 10.0.26100 de 64bits, ejecutado solamente desde el equipo mencionado, permitiendo obtener mediciones comparables en cuando al rendimiento de los algoritmos analizados.

3.1. Dataset (casos de prueba)

Los datasets para algoritmos de ordenamiento de un arreglo unidimensional están compuestos de números enteros almacenados en archivos de nombre {n}_{t}_{d}_{m}.txt

- n hace referencia a la cantidad de elementos, pertenecuenco al conjunto N = {10^1,10^3,10^5,10^7}.
- t hace referencia al tipo de matriz, perteneciendo al conjunto T = {ascendente, descendente, aleatorio}.
- d hace referencia al conjunto dominio de cada elemento del arreglo d = {D1,D7}, donde D1 implica que el dominio es {0,1,2,...,9} y D7 implica que el dominio es {0,1,2,...,10^7}.
- m hace referencia a la muestra aleatoria y pertenece al conjunto M={a,b,c}

Los datasets para algoritmos de multiplicación de matrices cuadradas están compuestos de n números en n filas almacenados en archivos de nombre {n}_{t}_{d}_{m}_1.txt y {n}_{t}_{d}_{m}_2.txt

- n hace referencia a la dimensión de la matriz (n filas y n columnas) y pertenece al conjunto N = {10^4, 10^6, 10^8, 10^{10}}.
- t hace referencia al tipo de matriz, y pertenece al conjunto T = {dispersa, diagonal, densa}.
- d hace referencia al conjunto 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,...,9}.
- m hace referencia a la muestra aleatoria y pertenece al conjunto M = {a,b,c}.

Tener una variedad de datasets es importante para evaluar correctamente algoritmos de ordenamiento y para la multiplicación de matrices. El tamaño y dimensiones del conjunto de datos permite analizar cómo escalan con el tiempo los algoritmos. A su vez, la distribución de los datos afecta al rendimiento, especialmente en algoritmos que dependen de comparaciones o tienen estructuras distintas. Además, algunos algoritmos funcionan bien con datos ordenados, pero mal con otros casos. La aleatoriedad nos asegura que el algoritmo no esté optimizado para un paso en particular, teniendo una evaluación más completa.

3.2. Resultados

Para visualizar los resultados ejecutados en otro equipo, se debe realizar “make run” desde:

- `../code/matrix_multiplication` ->ejecutar multiplicación de matrices.
- `../code/sorting` ->ejecutar ordenamiento.

Al terminar la ejecución, los resultados estarán en:

- `../code/matrix_multiplication/data/matrix_output` ->contiene carpetas (naive y strassen) con los resultados de las multiplicaciones en archivos de texto.
- `../code/sorting/data/array_output` ->contiene carpetas (mergesort, quicksort, selectionsort y sort) con los resultados de los ordenamientos en archivos de texto.

Además, se pueden revisar los tiempos y el uso de memoria en:

- `../code/matrix_multiplication/data/measurements` ->archivos de texto para cada implementación de multiplicación de matrices.
- `../code/sorting/data/measurements` ->archivos de texto para cada implementación de ordenamiento.

Cabe destacar que la ejecución de estos algoritmos puede arrojar resultados distintos en cada implementación, ya que se generan nuevos datasets tanto para la multiplicación de matrices como para el ordenamiento. Además, el hardware del equipo en que se ejecuten los programas influye significativamente en el desempeño en términos de tiempo y memoria. No obstante, la tendencia general de los resultados no debería verse alterada, permitiendo replicar el análisis presentado en este informe sin modificar las conclusiones teóricas.

Para el análisis de los algoritmos de ordenamiento, se realizaron pruebas utilizando distintos datasets, lo que permitió ampliar el estudio sobre su comportamiento. Inicialmente, como se muestra en la Figura 1, los tiempos de ejecución fueron similares hasta datasets de tamaño 100.000, donde comienza a evidenciarse una diferencia considerable en el tiempo requerido por selectionsort, y una leve diferencia respecto a quicksort.

Al aumentar las pruebas a datasets con 10.000.000 de elementos, se observa claramente el desempeño de mergesort y sort. Es importante señalar que no se realizaron pruebas con quicksort en estos últimos datasets debido a errores de segmentation fault en cada archivo, ni con selectionsort, dado que su ejecución supera las 5 horas de ejecución por archivo.

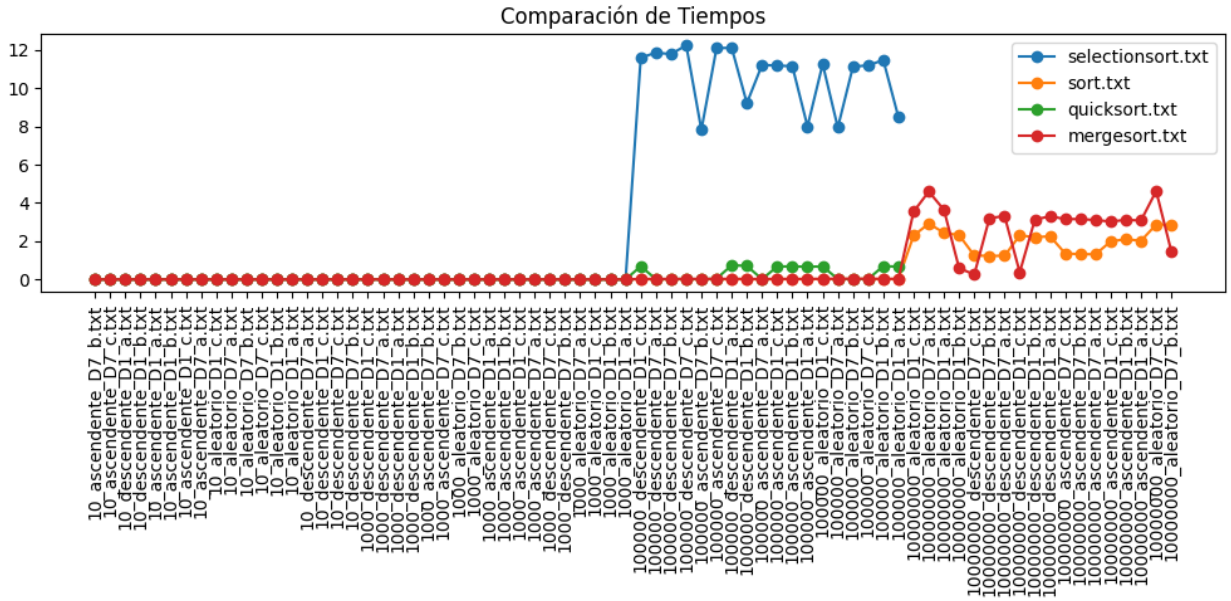

Figura 1: Tiempo en segundos empleado por cada ordenamiento

El uso de memoria para los archivos fue similar en todos los casos analizados hasta los datasets de tamaño 100.000 como se puede observar en la Figura 2. No obstante, no fue posible comparar el uso de memoria de quicksort y selectionsort en datasets de 10.000.000 elementos, ya que no se realizaron dichas pruebas. En particular, quicksort presentó errores de segmentation fault en todas las ejecuciones con estos últimos datasets, debido a que el uso de memoria superó el límite de stack asignado por el sistema. Esta situación se explica por la gran cantidad de llamadas recursivas provocadas por una mala elección de pivotes, generando una gran profundidad en la recursión.

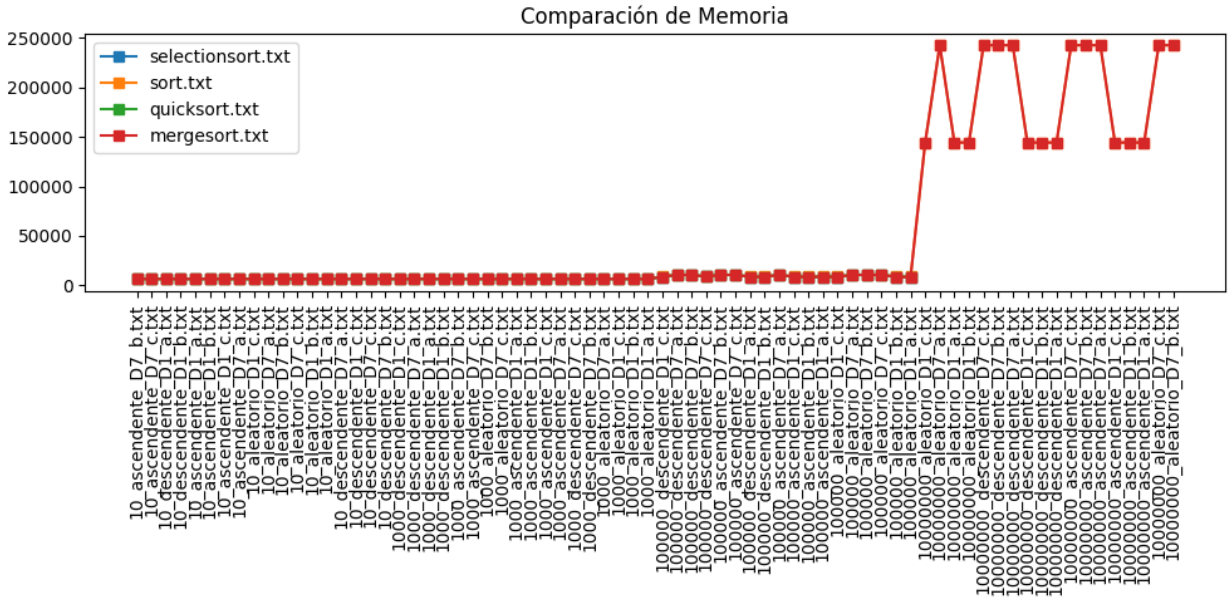

Figura 2: Memoria en kilobytes utilizada por cada ordenamiento

Para el análisis de los algoritmos de multiplicación de matrices se realizaron pruebas utilizando distintos datasets, lo que permitió ampliar el estudio sobre su comportamiento. Como se observa en la Figura 3, los tiempos de ejecución fueron similares para ambas implementaciones en los primeros tamaños de matrices. Sin embargo, a partir de matrices de tamaño 1024×1024, el algoritmo de Strassen aumentó significativamente el tiempo de ejecución, lo cual resulta curioso considerando que, teóricamente, su orden de complejidad es menor que el de naive. Esto se explica porque Strassen realiza muchas operaciones costosas que no son eficientes para datasets relativamente pequeños. Se proyecta que los resultados favorables para Strassen se observarán en matrices de tamaño aún mayor, donde naive incremente considerablemente sus tiempos de ejecución.

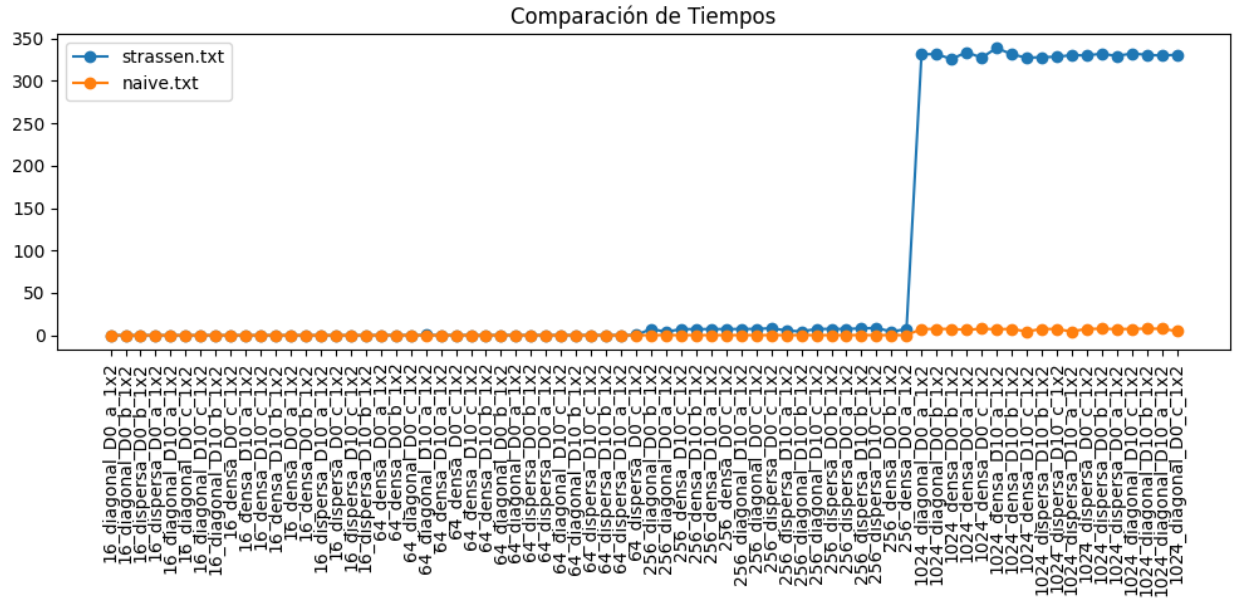

Figura 3: Tiempo en segundos empleado por cada multiplicación

El uso de memoria para los archivos fue similar en todos los casos analizados, como se observa en la Figura 4. Se observa una leve diferencia en el consumo de memoria por parte de strassen, se debe a las operaciones internas adicionales que realiza en comparación con naive. No obstante, esta diferencia podría incrementar a medida que aumente el tamaño de los datasets.

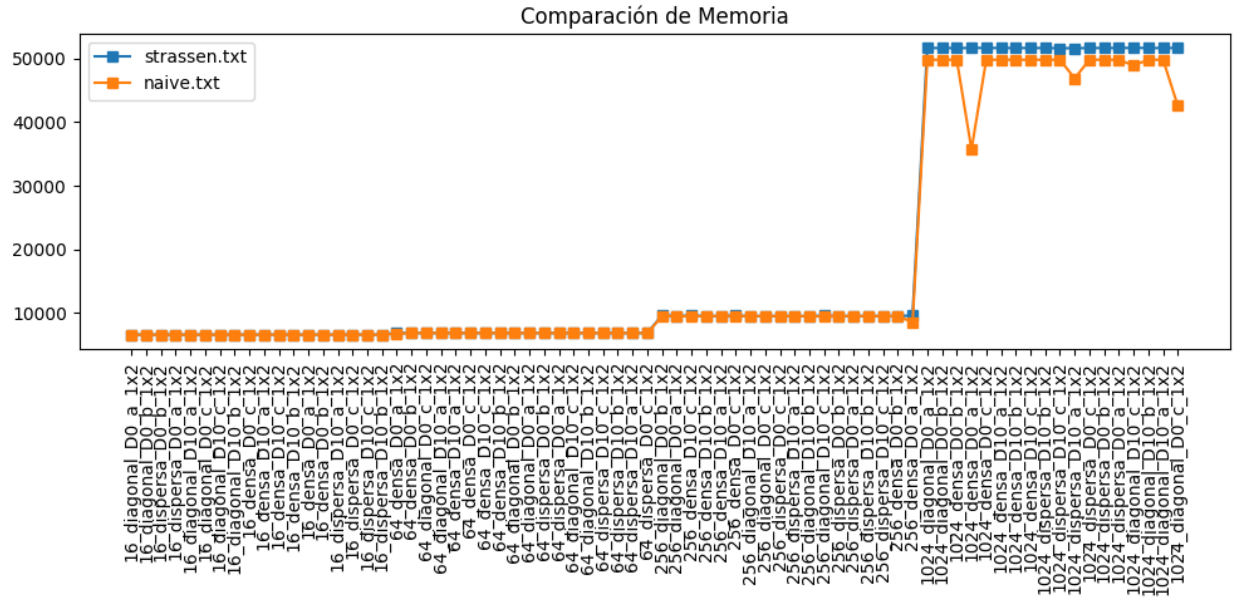

Figura 4: Memoria en kulobytes utilizada por cada multiplicación

4. Conclusiones

A partir de los datos generados y los tiempos obtenidos, se concluye que los algoritmos selectionsort y sort de C++ presentados en este informe son considerablemente más eficientes en tiempos de ejecución en comparación con SelectionSort y Quicksort en datasets de gran tamaño. Esta diferencia se debe a la forma en que están implementados, priorizando la división de tareas, la optimización de operaciones y una gestión más eficiente de grandes volúmenes de datos.

La complejidad temporal de estos algoritmos juega un papel fundamental en su desempeño. Mientras que los algoritmos cuadráticos según [1], como SelectionSort, tienen una complejidad de O(n^2), lo que provoca un aumento significativo en los tiempos de ejecución con grandes volúmenes de datos, algoritmos como Quicksort, Selectionsort y el sort de C++ tienen una complejidad de O(nlog(n)), lo que reduce drásticamente el tiempo de ejecución. Esta diferencia en la tasa de crecimiento explica por qué los algoritmos más eficientes presentan tiempos de ejecución considerablemente menores, como se evidenció en los casos de prueba. En los resultados obtenidos, los algoritmos eficientes lograron tiempos de ejecución reducidos, a excepción de Quicksort, el cual mostró un uso de memoria excesivo.

Un aspecto curioso de los resultados es la diferencia en los tiempos de Quicksort, que mostró un mayor uso de memoria y tiempos de ejecución. Esto se debe a la mala elección del pivote, lo que genera un comportamiento cercano al peor caso O(n^2), aumentando significativamente tanto el tiempo como la memoria utilizada.

Por otro lado, en la multiplicación de matrices, los resultados son contrarios a lo esperado. A pesar de que el algoritmo de Strassen tiene un orden de O($n^{2,81}$), se esperaban tiempos de ejecución menores que los de naive el cual tiene un orden de $O(n^3)$. Sin embargo, las diferencias en los tiempos se deben a que Strassen realiza más operaciones que el método tradicional, ya que incluye la división de matrices en submatrices y la combinación de resultados, lo que genera una sobrecarga y reduce su eficiencia. Por lo tanto, Strassen es más efectivo para matrices mucho más grandes que las que se probaron en este informe.

Finalmente, al realizar el análisis de los métodos de ordenamiento de datos y multiplicación de matrices, se concluye que cada método tiene sus propias ventajas y desventajas, y su eficiencia puede variar según las circunstancias y datos medidos. Comprender estas características nos permitirá escoger el algoritmo más adecuado según las necesidades específicas para la implementación.

A. Apéndice 1

Referencias

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 3.ª ed. MIT Press, 2009.