Informe: Algoritmos Dividir y Conquistar

Carlos Rabutia Riveros

INF-221 Algoritmo y complejidad

2024-2

Introducción

El presente informe aborda una evaluación exhaustiva de diversos algoritmos de ordenamiento y multiplicación de matrices, dos problemas fundamentales en el campo de la informática teórica y aplicada. Los algoritmos analizados incluyen tanto implementaciones propias como aquellas provistas por bibliotecas estándar de C++, abarcando métodos clásicos como Selection Sort, Merge Sort y Quick Sort, así como el algoritmo de Strassen para la multiplicación de matrices, comparándolos contra métodos tradicionales y optimizados.

Utilizando un conjunto variado de datasets que simulan condiciones prácticas “desde pequeños arreglos ordenados hasta grandes matrices desordenadas”, este estudio no solo mide la eficiencia y el rendimiento de cada algoritmo, sino que también explora cómo distintas condiciones de los datos impactan estos resultados. Las pruebas se realizaron midiendo tiempos de ejecución en microsegundos, proporcionando una base cuantitativa para el análisis comparativo.

Las conclusiones preliminares indican que, aunque el análisis teórico asintótico proporciona una buena base para entender el comportamiento general de los algoritmos, los resultados prácticos revelan la influencia significativa de la implementación específica y las características del dataset en el rendimiento final. Este informe subraya la importancia de combinar evaluaciones teóricas con pruebas empíricas para seleccionar o adaptar algoritmos a aplicaciones específicas en el mundo real.

Descripción de los algoritmos

Selection Sort

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
El algoritmo de ordenamiento por selección encuentra el mínimo elemento en cada iteración y lo coloca en la posición correcta del arreglo.

Costo temporal:

- **Mejor caso:** $O(n^2)$ (siempre realiza el mismo número de comparaciones, independientemente del orden inicial del arreglo)
- **Peor caso:** $O(n^2)$ (igualmente, el número de operaciones es el mismo)

Funciones específicas:

- **std::swap**: Intercambia los valores de dos elementos. Utiliza un algoritmo simple de intercambio que tiene un costo de O(1) para cada operación.

Merge Sort

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
El algoritmo de Merge Sort ordena un subarreglo en el rango de índices [left..right]. Primero, divide el arreglo en dos mitades hasta que cada subarreglo tiene un solo elemento o está vacío. Luego, une las mitades ordenadas usando la función de fusión "merge", que combina dos subarreglos en uno solo, manteniendo el orden.

Costo temporal:

- **Mejor caso:** O(n log n) (cuando el arreglo está bien distribuido y el algoritmo divide y fusiona eficientemente)
- **Peor caso:** O(n log n) (siempre divide en mitades y fusiona en tiempo logarítmico)

Funciones específicas:

- **std::vector**: Usado para manejar los subarreglos temporales y el arreglo principal. La gestión de memoria y el acceso a elementos tienen un costo de O(1) para operaciones básicas.
- **merge**: Fusiona dos subarreglos en vec[left..right].
    - **Costo**: O(n), No In-place.

Quicksort

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
El algoritmo de Quick Sort ordena un arreglo utilizando el método de partición. Primero, divide el arreglo en dos subarreglos basados en un pivote: los elementos menores al pivote se colocan a la izquierda y los mayores a la derecha. Luego, aplica el mismo proceso de partición de forma recursiva en ambos subarreglos hasta que el arreglo esté completamente ordenado.

Costo temporal:

- **Mejor caso:** O(n log n) (cuando el pivote divide el arreglo de manera equilibrada en cada paso de partición).
- **Peor caso:** $O(n^2)$ (si el pivote es el menor o mayor elemento en cada partición, resultando en particiones desequilibradas).

Funciones específicas:

- **std::swap**: Intercambia los valores de dos elementos en el arreglo.
    - **Costo**: O(1) por cada operación de intercambio.
- **partition**: Realiza la partición del arreglo alrededor del pivote.
    - **Costo**: O(n) para recorrer y reorganizar el arreglo en función del pivote.

Sort C++

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
La función Sorting utiliza std::sort de la biblioteca estándar de C++ para ordenar un vector. Esta función de ordenación es una implementación eficiente que se basa en un algoritmo de ordenación rápido, como Quick Sort, Heap Sort o Intro Sort.

Costo temporal:

- **Mejor caso:** O(n log n) (cuando el algoritmo logra un ordenamiento eficiente con particiones equilibradas).
- **Peor caso:** O(n log n) (la función está diseñada para ofrecer un rendimiento en tiempo logarítmico en el peor de los casos, utilizando algoritmos adaptativos como Intro Sort).

Funciones específicas:

- **std::sort**: Ordena los elementos en un rango definido por iteradores.
    - **Costo**: O(n log n) en promedio. std::sort utiliza el algoritmo Intro Sort, que combina Quick Sort, Heap Sort y, en algunos casos, Insertion Sort para obtener un rendimiento óptimo y garantizar la eficiencia en la mayoría de los escenarios.

Algoritmo iterativo cúbico tradicional

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
El algoritmo de multiplicación de matrices utiliza un triple bucle para calcular el producto de dos matrices clásicamente. Primero, verifica si las dimensiones de las matrices son compatibles para la multiplicación. Luego, realiza la multiplicación y suma de las componentes correspondientes para obtener la matriz resultante.

Costo temporal:

- **Mejor caso:** $O(n^3)$ (donde n es el tamaño de las matrices, en el mejor de los casos se realizan las mismas operaciones que en el peor caso).
- **Peor caso:** $O(n^3)$ (en el peor de los casos se realizan todas las multiplicaciones y sumas necesarias, con triple bucle anidado).

Funciones específicas:

- **std::vector**: Utilizado para almacenar las matrices y el resultado. Las operaciones básicas de acceso y modificación tienen un costo de O(1).
- **mulMat**: Realiza la multiplicación de matrices con un triple bucle.
    - **Costo**: $O(n^3)$, donde n es el tamaño de las matrices (suponiendo matrices de dimensiones n x n), debido a los tres bucles anidados que recorren filas y columnas.

Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos (transponiendo la segunda matriz)

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
El algoritmo realiza la multiplicación de dos matrices de manera optimizada mediante la transposición de la segunda matriz. Primero, transpone la segunda matriz para facilitar la multiplicación. Luego, multiplica la primera matriz con la matriz transpuesta utilizando un triple bucle, lo que mejora la eficiencia de acceso a los elementos.

Costo temporal:

- **Mejor caso:** $O(n^3)$ (donde n es el tamaño de las matrices, ya que el tiempo total depende del número de elementos en la matriz y la optimización no cambia el orden de complejidad).
- **Peor caso:** $O(n^3)$ (en el peor de los casos, la multiplicación y transposición involucran bucles anidados que recorren todos los elementos de las matrices).

Funciones específicas:

- **transpose**: Transpone la matriz original intercambiando filas por columnas.
    - **Costo**: O(m * n), donde m es el número de filas y n es el número de columnas de la matriz original.
- **mulOp**: Multiplica la primera matriz con la matriz transpuesta de la segunda.
    - **Costo**: $O(n^3)$, ya que realiza una multiplicación de matrices con un triple bucle anidado, a pesar de la optimización mediante transposición.

Algoritmo de Strassen

(se encuentra en la carpeta Algoritmo documentado)

Descripción:
El algoritmo realiza la multiplicación de matrices usando el método de Strassen para submatrices, lo que reduce la complejidad en comparación con la multiplicación clásica. Primero, divide las matrices en submatrices y utiliza siete multiplicaciones de submatrices en lugar de ocho, combinándolas con sumas y restas. La función add_matrix realiza la suma componente a componente de dos matrices, con un multiplicador opcional para la segunda matriz, facilitando las combinaciones en el algoritmo de Strassen.

Costo temporal:

- **Mejor caso:** $O(n^{\log_2(7)}) \approx O(n^{2.81})$ (la complejidad temporal se mejora comparado con el algoritmo clásico, pero depende de la implementación de la división en submatrices y combinaciones).
- **Peor caso:** $O(n^{\log_2(7)}) \approx O(n^{2.81})$ (el tiempo total en el peor de los casos sigue la misma complejidad debido a la reducción de operaciones necesarias en comparación con la multiplicación clásica).

Funciones específicas:

- **add_matrix**: Suma dos matrices componente a componente, con un multiplicador opcional para la segunda matriz.

    - **Costo:** $O(n^2)$ (donde n es el tamaño de las submatrices, ya que recorre todos los elementos de las matrices involucradas).

- **multiply_matrix**: Multiplica dos matrices utilizando el algoritmo de Strassen.
    - **Costo:** O(n^log2(7)) ≈ O(n^2.81) (mejora en comparación con la multiplicación clásica debido a la reducción en el número de multiplicaciones necesarias).

Descripción de los datasets

Un dataset es un conjunto de datos relacionados que se recopilan y organizan para su análisis. En este contexto lo ocupamos como pruebas de rendimiento, un dataset puede incluir una variedad de arreglos o matrices con diferentes características. Los datasets se utilizan para evaluar y comparar el rendimiento de los algoritmos bajo diferentes condiciones, como tamaño del conjunto de datos y distribución de los valores, por lo que se presentan los usados a continuación (Todos los datasets se encuentran en archivos .txt).

Datasets para algoritmos de ordenamiento

1 Arreglo tamaño 10 con números del 0 al 1000 ordenado ascendentemente

- Este dataset pequeño permite evaluar la eficiencia de los algoritmos en un caso ideal donde los datos ya están ordenados. Es útil para comprobar si el algoritmo puede manejar eficientemente datos ya ordenados sin realizar trabajo adicional innecesario.

2 Arreglo tamaño 10 con números del 1000 al 10000000 ordenado descendentemente

- Al igual que el anterior, pero en orden descendente, permite probar cómo los algoritmos manejan datos en el orden opuesto. Es útil para analizar cómo los algoritmos gestionan listas que están completamente desordenadas desde el punto de vista inverso.

3 Arreglo tamaño 10 con números del 0 al 1000 desordenado

- Este dataset permite evaluar el desempeño de los algoritmos en un caso típico de desorden. Es importante para probar la capacidad del algoritmo para ordenar datos que no tienen un patrón evidente.

4 Arreglo tamaño 1000 con números del 1000 al 10000000 desordenado

- Similar al anterior, pero con un tamaño de arreglo mayor. Permite evaluar la eficiencia y el tiempo de ejecución de los algoritmos en un dataset más grande y desordenado, reflejando situaciones más realistas.

5 Arreglo tamaño 1000 con números del 0 al 1000 ordenado ascendentemente

- Permite observar cómo los algoritmos manejan conjuntos de datos grandes que ya están ordenados. Esto es importante para entender si los algoritmos implementan optimizaciones para estos casos.

6 Arreglo tamaño 1000 con números del 1000 al 10000000 ordenado descendentemente

- Similar a los anteriores datasets ordenados, pero con un tamaño de arreglo más grande y en orden descendente. Esto ayuda a evaluar la robustez y la capacidad de adaptación de los algoritmos a diferentes patrones de datos.

7 Arreglo tamaño 1000 con números del 0 al 1000 desordenado

- Permite probar la capacidad del algoritmo para ordenar grandes cantidades de datos desordenados, proporcionando una visión de su eficiencia en escenarios de desorden más amplios.

8 Arreglo tamaño 1000 con números del 1000 al 10000000 desordenado

- Evaluación de la eficiencia del algoritmo en un conjunto de datos grande y desordenado, proporcionando una perspectiva sobre su rendimiento con datos extensos y aleatorios.

9 Arreglo tamaño 100000 con números del 0 al 1000 ordenado ascendentemente

- Permite observar cómo los algoritmos manejan grandes conjuntos de datos que ya están ordenados. Este dataset es útil para evaluar el impacto del tamaño del arreglo en el rendimiento de los algoritmos en casos ideales.

10 Arreglo tamaño 100000 con números del 1000 al 10000000 ordenado descendentemente

- Similar al anterior pero con datos ordenados en orden descendente, ayuda a probar la eficacia de los algoritmos en conjuntos de datos grandes y completamente desordenados desde el punto de vista inverso.

11 Arreglo tamaño 100000 con números del 0 al 1000 desordenado

- Proporciona una prueba exhaustiva del rendimiento de los algoritmos en grandes cantidades de datos desordenados, revelando cómo se comportan con conjuntos de datos extensos y aleatorios.

12 Arreglo tamaño 100000 con números del 1000 al 10000000 desordenado

- Es el dataset más grande y desordenado, ideal para evaluar la eficiencia y escalabilidad de los algoritmos en escenarios de desorden extensivo y variado.

Datasets para algoritmos de multiplicación de matrices

En el análisis de algoritmos de multiplicación de matrices, se han utilizado matrices cuadradas de diferentes tamaños para evaluar su rendimiento, multiplicándose entre sí mismas. A continuación, se observa cada uno de los datasets empleados:

Matriz 4x4 con números entre 0 y 1000

- Esta matriz pequeña permite evaluar la eficiencia y la corrección de los algoritmos de multiplicación de matrices en un caso simple y controlado. Dado su tamaño reducido, facilita la verificación manual de los resultados y permite observar el comportamiento básico de los algoritmos.

Matriz 64x64 con números entre 0 y 1000

- Este tamaño de matriz es suficientemente grande para proporcionar una evaluación más realista del rendimiento de los algoritmos. Permite observar cómo los algoritmos manejan un mayor volumen de datos y cómo se comportan en términos de eficiencia y tiempo de ejecución en un escenario más complejo.

Matriz 128x128 con números entre 0 y 1000

- Utilizar matrices de 128x128 permite probar los algoritmos en un entorno aún más grande y desafiante. Además, el uso de potencias de 2 (4, 64, 128) es importante porque el algoritmo de Strassen, en particular, requiere que las matrices sean potencias de 2 para funcionar correctamente. Esto asegura que todas las pruebas sean coherentes con los requisitos del algoritmo de Strassen y permite una comparación directa entre los distintos métodos de multiplicación de matrices bajo condiciones homogéneas.

Resultados Experimentales

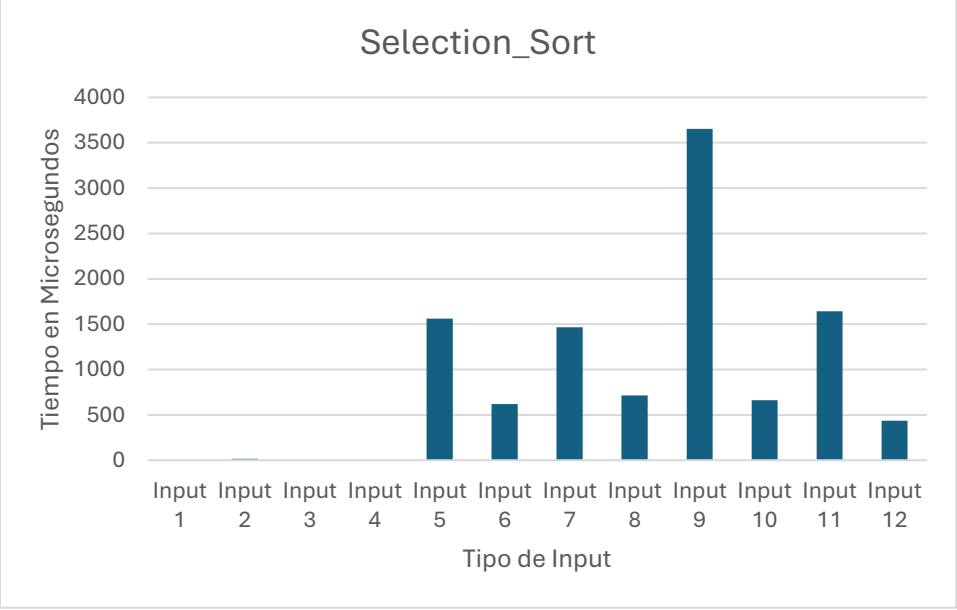
Los resultados se enuncian en gráficos de Tipo de input vs Tiempo de ejecución (en microsegundos), y al costado del gráfico su respectivo análisis y observaciones captadas de los resultados (Cabe destacar que se obtuvieron los resultados de ocupar la función Chrono de C++, con la cual se pudo medir el tiempo de ejecución de cada algoritmo)

Selection Sort

Este algoritmo muestra tiempos de ejecución considerables en arreglos grandes y desordenados (Input 8, 9, 12), lo que indica su ineficiencia para manejar grandes volúmenes de datos desordenados.

Para arreglos pequeños y ya ordenados (Input 1, 2, 3), el algoritmo demuestra ser muy eficiente, casi no requiriendo tiempo adicional, lo que sugiere que maneja bien pequeñas cantidades de datos ordenados o parcialmente ordenados.

La variabilidad en los tiempos de ejecución entre arreglos pequeños y grandes es muy notable, destacando la no escalabilidad del algoritmo para tamaños de datos incrementados.

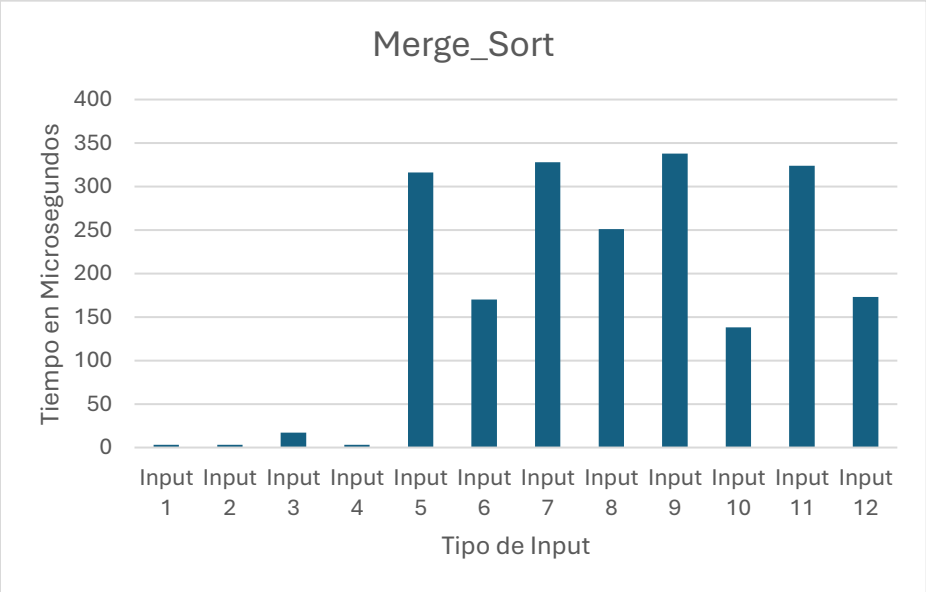


Merge Sort

Exhibe una consistencia relativa en tiempos de ejecución a través de diferentes condiciones de datos (desordenados, ordenados ascendente y descendentemente), lo que muestra su robustez y eficiencia en variados escenarios, como se observa en los Inputs 4, 5, y 6.

Aunque generalmente más eficiente que Selection Sort en arreglos grandes, sus tiempos incrementan con el tamaño del arreglo, aunque no tanto como Selection Sort, indicando mejor escalabilidad.

Los tiempos de ejecución más bajos en datasets pequeños y ordenados (Inputs 1 y 5) sugieren que el Merge Sort es eficaz para manejar eficientemente datos ya organizados, con una sobrecarga computacional mínima.

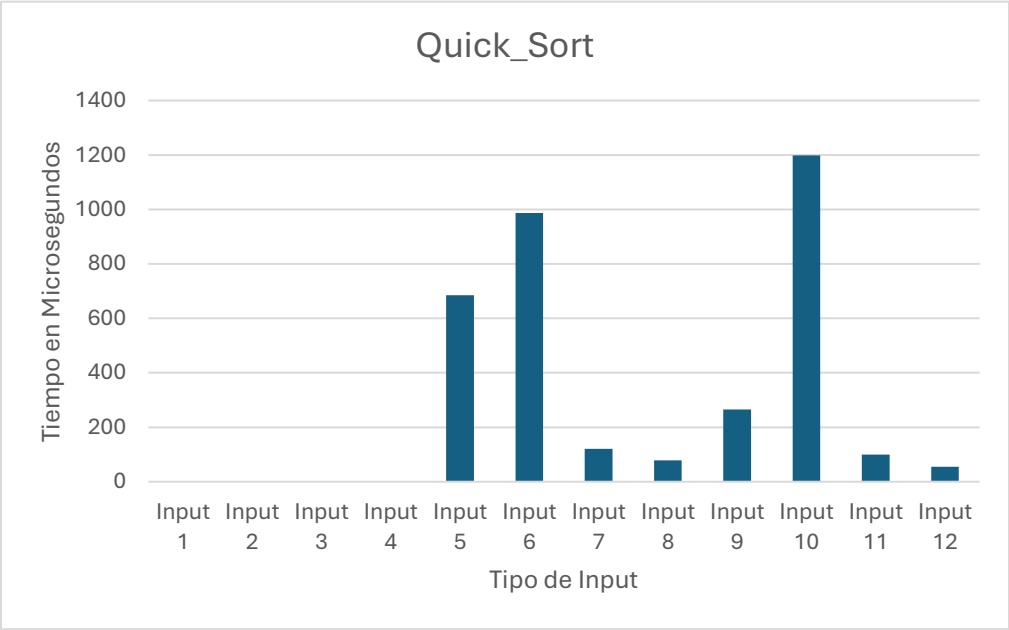


Quick Sort

En general, muestra tiempos de ejecución eficientes en casi todos los datasets, particularmente en Inputs 7 y 8, indicando buena adaptabilidad y eficiencia para manejar datos desordenados de tamaño moderado.

Sin embargo, en ciertos casos como el Input 10, el tiempo de ejecución es significativamente alto, lo que podría indicar casos de peor escenario (pivotes mal elegidos) donde su rendimiento degrada.

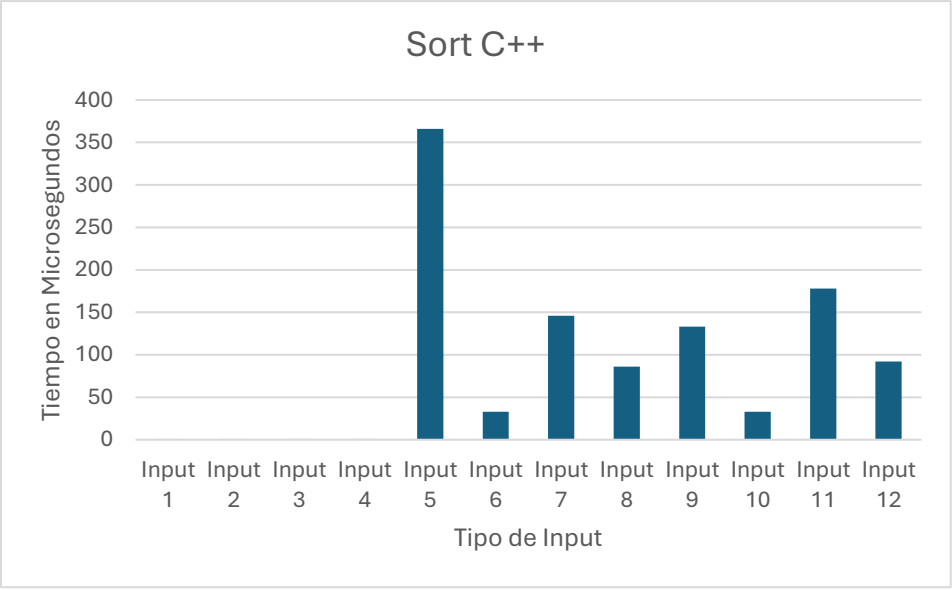
Su rendimiento en arreglos pequeños y grandes muestra que es competitivamente eficiente, aunque con ciertas inconsistencias que pueden depender de la naturaleza del dataset y la elección de pivote.



Sort C++

A través de los diferentes datasets, desde pequeños arreglos ordenados hasta grandes arreglos desordenados, este algoritmo mantiene una consistencia en tiempos bajos de ejecución. Esto sugiere que el algoritmo implementa estrategias internas eficaces para ajustarse a la naturaleza de los datos, como se observa en el desempeño constante en Inputs 4, 5, y 6.

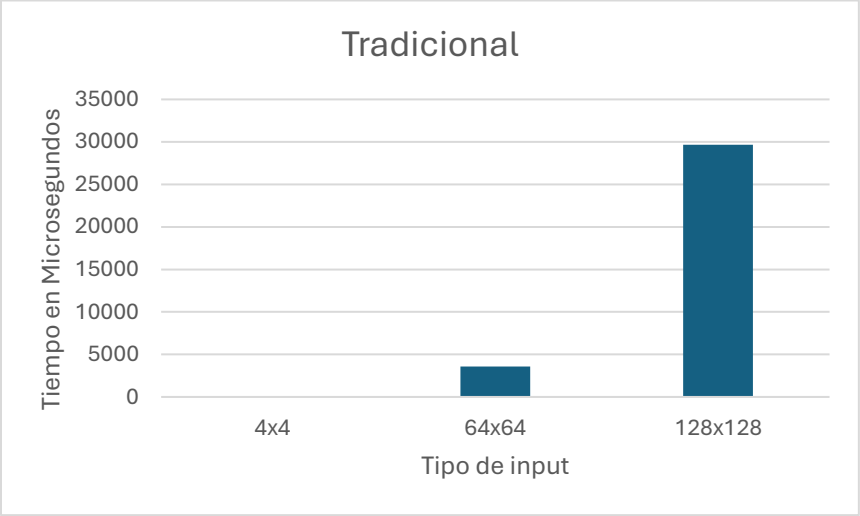
El algoritmo de ordenamiento de C++ muestra una notable eficiencia en casi todos los inputs. Destaca en arreglos grandes y desordenados (Input 9 y 12), con tiempos significativamente menores en comparación con otros algoritmos como Selection Sort y Quick Sort, lo que refleja su optimización y adaptabilidad a diversos tamaños y condiciones de datos.



Algoritmo iterativo cúbico tradicional

Tiene un rendimiento constante y predecible que escala linealmente con el tamaño de la matriz, como se observa en las matrices 4x4 a 128x128. Esto indica una buena escalabilidad lineal sin optimizaciones específicas para casos más pequeños o grandes.

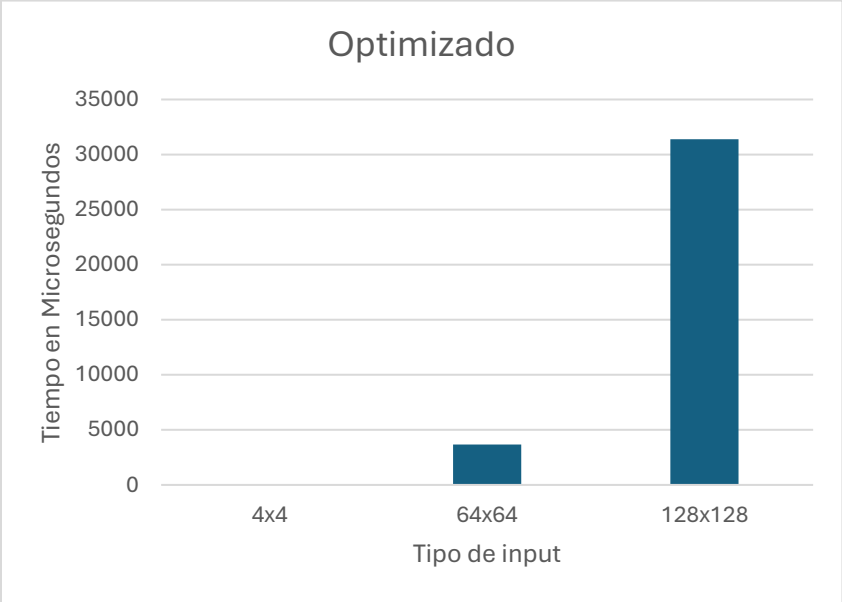
Comparado con el optimizado y Strassen, el tradicional es significativamente más rápido en matrices pequeñas (4x4), sugiriendo que las complejidades adicionales en otros algoritmos no se justifican para tamaños reducidos.



Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos (transponiendo la segunda matriz)

Muestra un rendimiento similar al tradicional en matrices pequeñas y medianas, pero no ofrece ventajas significativas en tiempos de ejecución, lo que puede indicar que las optimizaciones son menos efectivas bajo ciertas condiciones o tamaños de matriz.

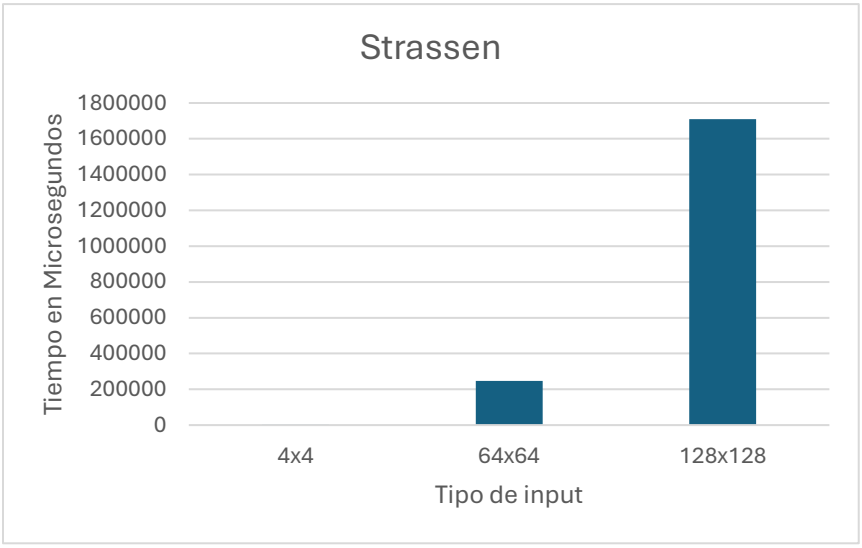
Su consistencia en diferentes tamaños de matrices sugiere una buena implementación general sin degradaciones específicas en ciertos tamaños.



Algoritmo de Strassen

Es considerablemente más lento en matrices pequeñas (4x4), donde la complejidad del algoritmo no compensa su tamaño. Sin embargo, se vuelve extremadamente costoso en términos de tiempo cuando se escala a 128x128, destacando que este método puede no ser eficiente para todos los tamaños de matriz debido a su alto costo computacional en la división y conquista de submatrices.

Este algoritmo es conocido por su eficiencia teórica en grandes matrices, pero los datos muestran que puede no ser el más adecuado para todas las situaciones prácticas, especialmente en matrices más pequeñas o cuando el overhead computacional no se justifica.



Conclusiones

Este informe ha realizado una evaluación exhaustiva de diversos algoritmos de ordenamiento y multiplicación de matrices a través de una serie de pruebas estructuradas utilizando datasets variados. Los resultados obtenidos permiten extraer varias conclusiones importantes respecto a la eficiencia, escalabilidad y comportamiento práctico de estos algoritmos bajo diferentes condiciones.

Aunque el análisis asintótico ofrece una visión teórica del comportamiento de los algoritmos, los resultados experimentales muestran que las características específicas de implementación y el tipo de datos pueden influir significativamente en el rendimiento. Por ejemplo, mientras que algoritmos como Quick Sort muestran buena eficiencia teórica, en la práctica, situaciones como la elección de pivotes pueden afectar drásticamente su desempeño. Esto subraya la importancia de considerar el análisis asintótico junto con evaluaciones empíricas para predecir el comportamiento real de los algoritmos.

Las diferencias en los tiempos de ejecución entre algoritmos como el Sorting de C++, Merge Sort y Selection Sort demuestran cómo las optimizaciones y técnicas de implementación pueden mejorar la eficiencia. En particular, el algoritmo de Sorting de C++ destacó por su consistencia y rendimiento superior en todos los datasets, lo que refleja la efectividad de las optimizaciones internas que probablemente incluyen mejoras en la preservación de la localidad de los datos y técnicas de ordenamiento adaptativo.

La eficiencia de los algoritmos de multiplicación de matrices también mostró variaciones considerables, especialmente en el caso del algoritmo de Strassen en comparación con métodos tradicionales y optimizados, lo que ilustra cómo la complejidad de implementación puede no justificar el uso de métodos teóricamente más rápidos en todas las situaciones prácticas.

El comportamiento de los algoritmos cambió notablemente entre los diferentes tipos de datasets. Por ejemplo, los algoritmos mostraron variaciones en su eficiencia dependiendo si los datos estaban ordenados o desordenados, y según el tamaño del dataset. Esto es crucial para la selección de un algoritmo adecuado en aplicaciones específicas, donde el tipo de datos y su tamaño pueden determinar la elección del algoritmo más eficiente.

En conclusión, mientras que el análisis teórico provee una base para entender el comportamiento potencial de los algoritmos, este informe subraya la importancia de realizar pruebas empíricas para capturar las complejidades y variaciones en el rendimiento real. La implementación específica y las características del dataset juegan roles críticos en el desempeño de los algoritmos, lo que sugiere la necesidad de un enfoque balanceado que combine teoría y práctica para la selección y optimización de algoritmos en aplicaciones del mundo real.

Fuentes de Información

Algoritmo Selection Sort y Cúbico interativo optimizado:

Apunte de profesora Raquel Pezoa Rivera

Algoritmo Quick Sort
https://www.geeksforgeeks.org/quick-sort-algorithm/?ref=gcse_ind

Algoritmo Merge Sort
https://www.geeksforgeeks.org/cpp-program-for-merge-sort/?ref=gcse_ind

Algoritmo Interativo Cúbico tradicional

https://www.geeksforgeeks.org/c-program-multiply-two-matrices/

Algoritmo Strassen

https://www.geeksforgeeks.org/strassens-matrix-multiplication/?ref=gcse_ind