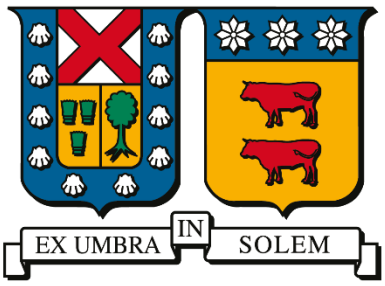
Informe N1 Algoritmos y Complejidad



UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Benjamín Ferrada Larach

202273061-7

7 de septiembre de 2024

Índice.

Índice

        - Rectangulares
4. **Resultados Experimentales**
    - Análisis de datos
    - Gráficos y visualizaciones
        - Dataset 10^4
        - Dataset 10^5
        - Dataset Matrices
5. **Conclusiones**
6. **Referencias**

Introducción.

El objetivo de este informe es aprender a implementar diferentes algoritmos de ordenamiento y de multiplicación de matrices, así como, comparar estos mismos creados por los ya provistos por los diferentes lenguajes de programación. Además, empezar a introducirse al uso de herramientas de profiling.

Los algoritmos evaluados en este informe son:

- De ordenamiento:
    1. Selection Sort.
    2. Mergesort.
    3. Quicksort.
    4. Función de sorting implementada en la biblioteca estándar de C++

- De multiplicación de matrices:
    1. Iterativo cúbico tradicional.
    2. Iterativo cúbico optimizado para mantener la localidad de los datos.
    3. De Strassen.

Las herramientas de profiling a usar serán valgrind para el uso de memoria, chrono para tomar el tiempo, gprof para generar un informe.

Para la creación de datasets, se usará Python y se guardarán en archivos de .csv. Además, para el análisis de datos, se usará Python con la librería Pandas y matplotlib para la generación de gráficos.

Descripción de los algoritmos.

Los algoritmos se encontrarán en el repositorio de Github, dentro del archivo "*first_report_algorithms/algorithms.cpp*", de esta forma fueron llamados en "*main.cpp*" y si son mencionados será el nombre de la función, además que cada algoritmo está separado por diferentes funciones que son diferenciadas en el código, por los algoritmos.

Estos algoritmos guardan cada resultado en diferentes .csv en la carpeta "*/resultados"* y son diferenciados por el nombre del archivo.

➢ Algoritmos de ordenamiento:

- **Selection Sort:** Es un algoritmo sencillo que selecciona el elemento más pequeño de una lista no ordenada y lo intercambia con el primer elemento. Luego, repite el proceso para el resto de la lista. Es fácil de entender, pero no muy eficiente para listas largas.

Mejor caso	Caso promedio	Peor caso
$O(n^2)$	$O(n^2)$	$O(n^2)$

- **Merge Sort:** Es un algoritmo más eficiente que divide la lista en dos partes, las ordena por separado y luego las fusiona en una lista ordenada. Su principal ventaja es que siempre tiene un buen rendimiento, incluso con grandes volúmenes de datos.

Mejor caso	Caso promedio	Peor caso
$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$

- **Quick Sort:** Similar al Merge Sort, divide la lista, pero lo hace eligiendo un "pivote". Organiza los elementos menores y mayores que el pivote en lados opuestos y luego los ordena recursivamente. En el caso del código el pivote elegido es aleatorio.

Mejor caso	Caso promedio	Peor caso
O(nlog(n))	***O(nlog(n))***	***$O(n^2)$***

- **Sort de C++:** Es el algoritmo de ordenamiento incorporado en la biblioteca estándar de C++. Usualmente, es una combinación de Quick Sort, Heap Sort y Merge Sort, optimizado para la mayoría de los casos.

Mejor caso	Caso promedio	Peor caso
O(nlog(n))	***O(nlog(n))***	***O(nlog(n))***

➢ *Algoritmos de multiplicación de matrices:*

- **Iterativo cúbico tradicional:** Es la forma más directa de multiplicar matrices. Se basa en tres bucles anidados que recorren las filas y columnas, multiplicando y sumando los elementos correspondientes. Aunque es fácil de implementar, no es el método más rápido.

Complejidad
$O(n^3)$

- **Iterativo cúbico optimizado para mantener la localidad de los datos:** Es una mejora del método tradicional, diseñado para mejorar la “localidad de los datos”. Esto significa que intenta aprovechar mejor la memoria caché del procesador para hacer la multiplicación más eficiente.

Complejidad
$O(n^3)$

- **De Strassen:** Es un algoritmo más avanzado que reduce la cantidad de operaciones necesarias para multiplicar matrices grandes. Utiliza una técnica de dividir y conquistar para hacer la multiplicación más rápido que el método iterativo tradicional, aunque puede ser más complicado de implementar correctamente.

Complejidad
$O(n^{2.81})$

Descripción de los datasets.

Los datasets se encontrarán en el repositorio de Github, dentro de la carpeta first_report_algorithms/datasets, de forma que ahora serán solo referenciados como ‘nombre_archivo.extensión’ en el informe :

Los datasets fueron creados en Python con la ayuda de la librería “random” y guardados en diferentes archivos .csv con ayuda de la función *“save_list”.*

- ## Listas:

  Fueron generados 4 tipos de listas:

  1. Lista desordenada con datos hasta 10^4:
     De archivo “*unsorted10^4.csv*”, esta lista tiene 10^4 enteros que están entre 0 y 10^4 de tamaño, todos generados mediante el uso de “randint”.

  2. Lista medianamente desordenada con datos hasta 10^4:’
     De archivo “*sorted_middle10^4.csv*”, esta lista tiene 10^4 enteros que están entre 0 y 10^4 de tamaño, todos generados mediante el uso de “randint”. A diferencia de la anterior, esta lista está un 40% ya ordenada, para evaluar el algoritmo en diferentes situaciones.

  3. Lista desordenada con datos hasta 10^5:
     De archivo “*unsorted10^5.csv*”, esta lista tiene 10^5 enteros que están entre 0 y 10^4 de tamaño, todos generados mediante el uso de “randint”.

  4. Lista medianamente desordenada con datos hasta 10^6:
     De archivo “*sorted_middle10^5.csv*”, esta lista tiene 10^4 enteros que están entre 0 y 10^4 de tamaño, todos generados mediante el uso de “randint”. A diferencia de la anterior, esta lista está un 60% ya ordenada, para evaluar el algoritmo en diferentes situaciones.

- ## Matrices:

  Fueron generados solo 2 tipos de matrices:

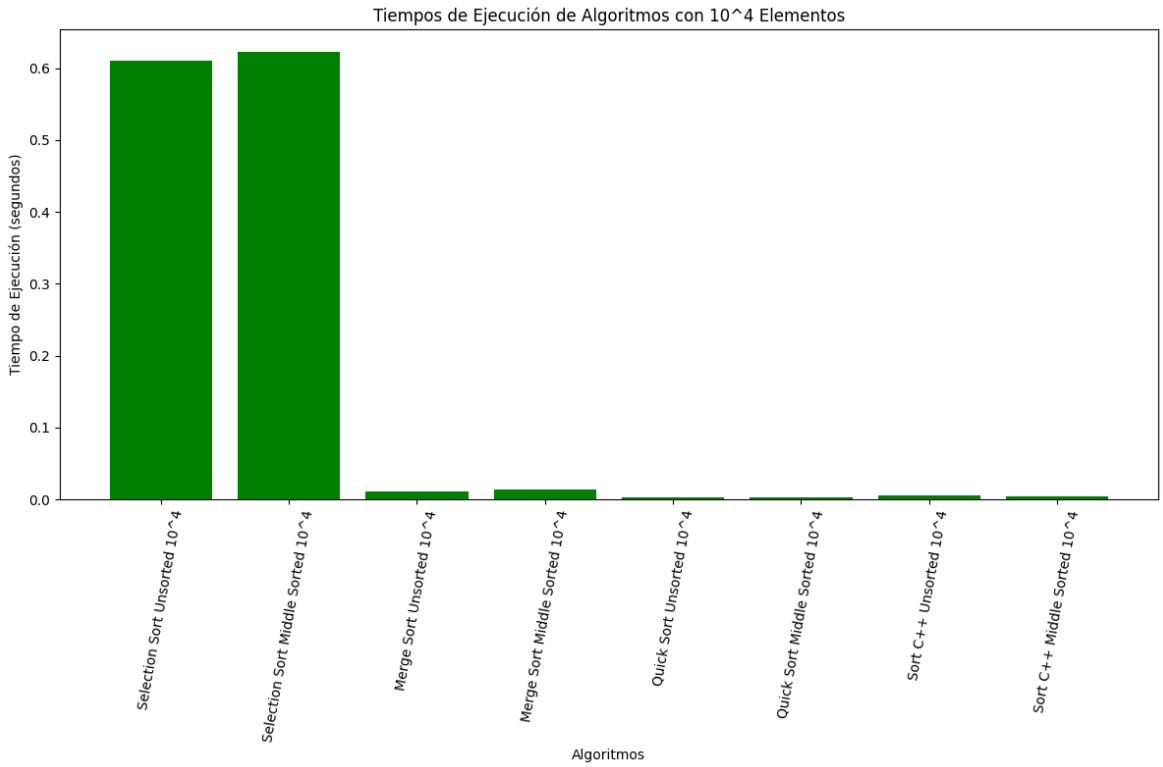
  1. Cuadradas:
     De archivo “*square_matrix.csv*”, esta es una matriz de 100x100, que contiene enteros entre 0 y 10^4.

  2. Rectangulares:
     De archivo “*rectangular_matrix.csv*”, esta es una matriz de 100x200, que contiene enteros entre 0 y 10^4.

Resultados experimentales.

Los datos fueron analizados con ayuda del archivo “*algorithmAnalysis.py*” que es un código programado en Python, y que lee los .csv guardados en la carpeta “*/tiempos”* y generados por el “*main.cpp”* separados por datasets con datos hasta 10^4, hasta 10^5 y las matrices. Que contienen el nombre y tiempo de ejecución de la siguiente forma “Nombre del Algoritmo | Tiempo de Ejecución (segundos)”. Estos son separados por pandas y con matplotlib se generan los siguientes gráficos guardados en la carpeta “*/visuals”*:

- **Dataset 10^4:**

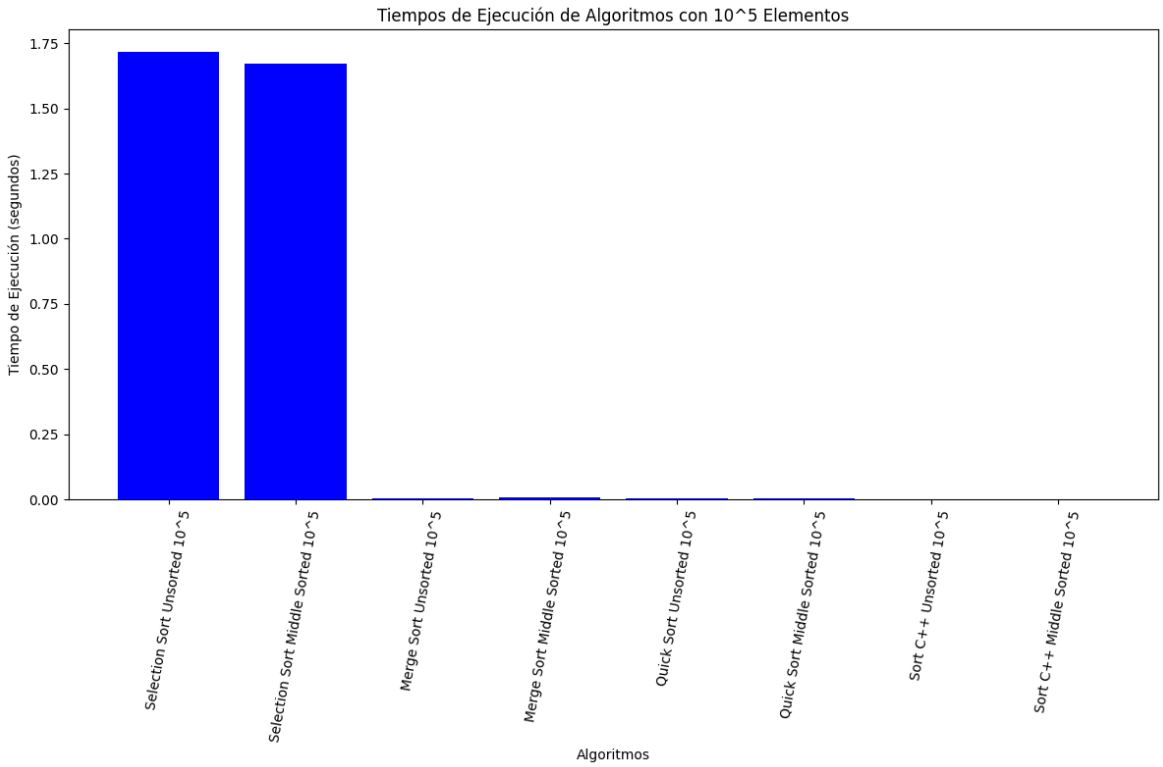


(Gráfico comparativo “Tiempos de Ejecución de Algoritmos con 10^4 Elementos.png”, Fig N1)

Nombre del Algoritmo	Tiempo de Ejecución (segundos)
Selection Sort Unsorted 10^4	0.0169911
Selection Sort Middle Sorted 10^4	0.0192305
Merge Sort Unsorted 10^4	0.00119621
Merge Sort Middle Sorted 10^4	0.00150914
Quick Sort Unsorted 10^4	0.00030983
Quick Sort Middle Sorted 10^4	0.000307084
Sort C++ Unsorted 10^4	9.9108e-05
Sort C++ Middle Sorted 10^4	0.000133084

("Tabla ejemplo de Tiempos de Ejecución de Algoritmos con 10^4 Elementos", (Datos obtenidos de prueba), Fig N2)

- **Dataset 10^5:**

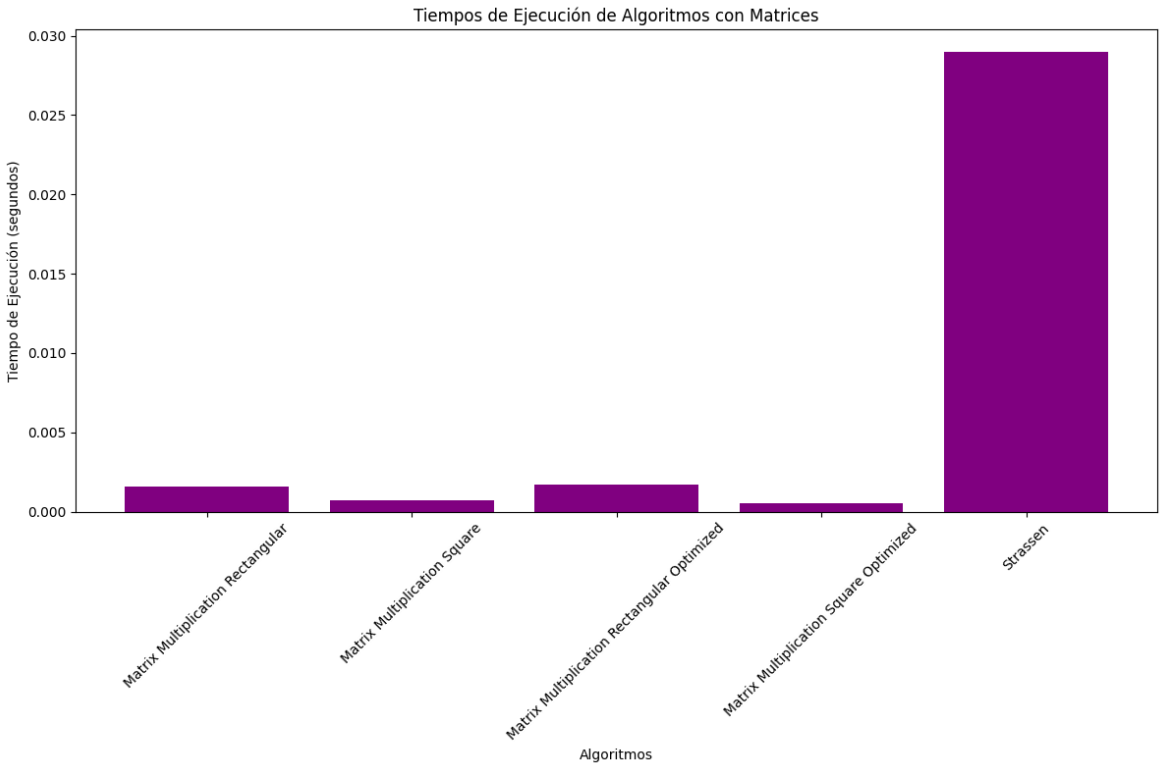


(Gráfico comparativo "Tiempos de Ejecución de Algoritmos con 10^5 Elementos.png", Fig N3)

Nombre del Algoritmo	Tiempo de Ejecución (segundos)
Selection Sort Unsorted 10^4	1.71805
Selection Sort Middle Sorted 10^4	1.67282
Merge Sort Unsorted 10^4	0.00484815
Merge Sort Middle Sorted 10^4	0.00899141
Quick Sort Unsorted 10^4	0.00327781
Quick Sort Middle Sorted 10^4	0.00300041
Sort C++ Unsorted 10^4	0.00121475
Sort C++ Middle Sorted 10^4	0.00124675

("Tabla ejemplo de Tiempos de Ejecución de Algoritmos con 10^5 Elementos", (Datos obtenidos de prueba), Fig N4)

- **Dataset Matrices:**



("Tiempos de Ejecución de Algoritmos con matrices.png", Fig N5)

Nombre del Algoritmo	Tiempo de Ejecución (segundos)
Matrix Multiplication Rectangular	0.00156847
Matrix Multiplication Square	0.000711032
Matrix Multiplication Rectangular Optimized	0.00173494
Matrix Multiplication Square Optimized	0.000534216
Strassen	0.0289652

("Tabla ejemplo de Tiempos de Ejecución de Algoritmos de multiplicación de matrices", (Datos obtenidos de prueba), Fig N6)

Conclusiones.

En conclusión, este informe presenta una evaluación detallada de varios algoritmos de ordenamiento y multiplicación de matrices, utilizando datasets de diferentes tamaños y tipos. A través del análisis de los tiempos de ejecución y el uso de herramientas de profiling, se puede observar que los algoritmos más avanzados, como Merge Sort y Quick Sort, tienen un rendimiento significativamente mejor que los algoritmos más simples como Selection Sort, especialmente cuando se trabaja con grandes volúmenes de datos.

En cuanto a la multiplicación de matrices, el algoritmo de Strassen mostró ser más eficiente en términos de complejidad teórica, aunque en la práctica, su desempeño no siempre supera a los métodos iterativos optimizados para matrices de tamaño moderado.

Finalmente, el uso de herramientas como valgrind, chrono y gprof permitió obtener una visión completa no solo del tiempo de ejecución, sino también del uso de recursos de memoria, proporcionando una base sólida para futuras optimizaciones.

Referencias.

https://chatgpt.com/

https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm/

https://www.topcoder.com/thrive/articles/strassenss-algorithm-for-matrix-multiplication

https://github.com/benyens/algoco