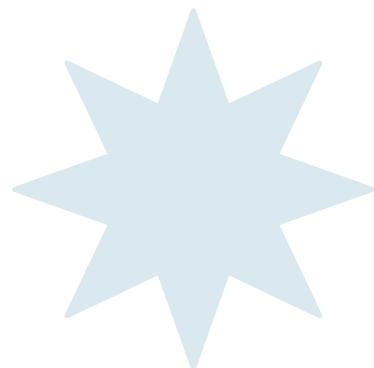




UNIVERSIDAD TECNICA  
FEDERICO SANTA MARIA

# Tarea 1

Algoritmos y  
Complejidad



# Dividir y conquistar

Geraldine Cornejo Valenzuela  
ROL: 202173529-1

# Introducción

Este informe persigue realizar una evaluación experimental de distintos algoritmos de ordenamiento y de multiplicación de matrices. Así, se realizó la implementación de algunos de los algoritmos más conocidos para este tipo de problemas, además de contrastar su funcionamiento con implementaciones provistas por los lenguajes de programación.

Los algoritmos a estudiar son:

- De ordenamiento: Selection Sort, MergeSort, QuickSort y función de biblioteca estándar.
- Multiplicación de matrices: Cúbico tradicional, Cúbico optimizado y Strassen.

Los dataset consisten en:

- De ordenamiento: Archivo .txt con un enteros a utilizar en el vector.
- Multiplicación de matrices: Archivo .txt con las matrices a multiplicar.

Las herramientas utilizadas son:

- Lenguajes de programación: C++ (algoritmos) y Python3 (generación de datasets)

Conclusiones preliminares:

La elección del algoritmo a utilizar depende del tamaño del problema a resolver, dado que tanto algoritmos cúbicos como cuadráticos son adecuados para problemas pequeños, pero cuando escala la cantidad de datos se debe optar por un algoritmo más avanzado y por ende, más eficiente. Es por esto que se debe elegir delicadamente el algoritmo adecuado que optimice la localidad de los datos, ya que esto impactará significativamente el rendimiento general de los algoritmos, como en el caso del algoritmo QuickSort o MergeSort para ordenamiento y Strassen para multiplicación de matrices.



# 02

# Descripción de Algoritmos

## 2.1. Algoritmos de Ordenamiento.

### 2.1.1. Selection Sort (Algoritmo Cuadrático)

- Descripción: Selection Sort es un algoritmo de ordenamiento simple que funciona seleccionando el elemento más pequeño de la lista y colocándolo en su posición correcta, repitiendo este proceso para cada elemento de la lista.
- Complejidad:
  - Mejor caso:  $O(n^2)$  (el algoritmo siempre realiza comparaciones, incluso si la lista ya está ordenada).
  - Peor caso:  $O(n^2)$  (cuando la lista está invertida).
  - Espacial:  $O(1)$  (es in-place, ya que no requiere espacio adicional).
- Comparación con la función de biblioteca: Es significativamente menos eficiente que las funciones de ordenamiento en las bibliotecas estándar, como `std::sort` en C++.

### 2.1.2. MergeSort

- Descripción: MergeSort es un algoritmo de ordenamiento basado en la técnica de divide y vencerás. Divide la lista en dos mitades, ordena cada mitad recursivamente y luego las fusiona.
- Complejidad:
  - Mejor caso:  $O(n \log n)$ .
  - Peor caso:  $O(n \log n)$  (incluso cuando la lista está invertida).
  - Espacial:  $O(n)$  (requiere espacio adicional para las sublistas durante la fusión).
- Comparación con la función de biblioteca: Es más eficiente que Selection Sort y generalmente más rápido que QuickSort en casos específicos, pero ocupa más memoria.

### 2.1.3. QuickSort

- Descripción: QuickSort es un algoritmo de ordenamiento rápido que selecciona un pivote y reordena los elementos de la lista de manera que todos los elementos menores que el pivote queden a la izquierda y los mayores a la derecha. Luego aplica recursivamente el proceso.
- Complejidad:
  - Mejor caso:  $O(n \log n)$  (cuando el pivote divide la lista en partes casi iguales).
  - Peor caso:  $O(n^2)$  (cuando el pivote divide la lista de manera muy desigual, como en una lista ya ordenada).
  - Espacial:  $O(\log n)$  (versión in-place) o  $O(n)$  si se usa un arreglo adicional.
- Comparación con la función de biblioteca: `std::sort` en C++ está basado en QuickSort y otras optimizaciones. Su rendimiento es casi siempre excelente.



Códigos



#### 2.1.4. Función de biblioteca estándar (`std::sort`)

- Descripción: La función `std::sort` en C++ implementa una versión optimizada de QuickSort (generalmente una combinación de QuickSort, MergeSort y otros algoritmos) que mejora el rendimiento en la mayoría de los casos.
- Complejidad:
  - Mejor caso:  $O(n \log n)$ .
  - Peor caso:  $O(n \log n)$  (gracias a las optimizaciones internas que evitan los problemas típicos de QuickSort puro).
  - Espacial:  $O(1)$  en versiones in-place.
- Comparación: Es considerablemente más rápido que los algoritmos cuadráticos y ofrece un equilibrio sólido entre velocidad y uso de memoria.

### 2.2. Algoritmos de Multiplicación de Matrices

#### 2.2.1. Algoritmo Iterativo Cúbico Tradicional

- Descripción: Es el algoritmo básico de multiplicación de matrices que sigue la fórmula estándar  $C[i][j] = \sum_{k=1}^n A[i][k] \times B[k][j]$ . Itera a través de cada elemento de las matrices para calcular la matriz resultante.
- Complejidad:
  - Tiempo:  $O(n^3)$ , ya que hay tres bucles anidados para recorrer las matrices.
  - Espacial:  $O(n^2)$  para almacenar la matriz resultante.
- Comparación: Aunque es fácil de implementar, no es eficiente para matrices grandes.

#### 2.2.2. Algoritmo Iterativo Cúbico Optimizado

- Descripción: Este algoritmo es una optimización del algoritmo cúbico tradicional, donde la segunda matriz se transpone para mejorar la localidad de los datos en caché, reduciendo el costo de acceso a memoria.
- Complejidad:
  - Tiempo:  $O(n^3)$ , pero con un mejor comportamiento debido al acceso más eficiente a la memoria.
  - Espacial:  $O(n^2)$ , ya que requiere espacio adicional para la matriz transpuesta.
- Comparación: Es más rápido que el algoritmo cúbico tradicional, pero sigue siendo  $O(n^3)$ .

#### 2.2.3. Algoritmo de Strassen

- Descripción: El algoritmo de Strassen mejora la multiplicación de matrices al reducir el número de multiplicaciones necesarias mediante un enfoque recursivo que divide las matrices en submatrices más pequeñas.
- Complejidad:
  - Tiempo:  $O(n \log 27) \approx O(n^{2.81})$ , lo que lo hace más rápido que el algoritmo cúbico.
  - Espacial:  $O(n^2)$ , pero necesita espacio adicional para las submatrices temporales.
- Comparación: Es más rápido que los métodos iterativos cúbicos, especialmente para matrices grandes, aunque requiere más memoria.



Códigos

# Descripción de Datasets

## 3.1 Algoritmos de Ordenamiento

El formato del archivo para este tipo de algoritmos será un archivo .txt que tendrá los datos en formato de texto. Como en los códigos se trabaja ordenando vectores, la forma más óptima de tener los datos de prueba es en una columna de enteros, donde cada fila representa un elemento del vector y la cantidad de filas es la cantidad de elementos en el vector.

La entrada de datos será el archivo ya explicado y las salidas consistirán en mostrar el vector original, luego el tiempo que le tomó al algoritmo ordenar dicho vector y por último, el vector ordenado.

Cada archivo de código .cpp procesa el dataset mediante una función auxiliar que lee el archivo y guarda cada número como un elemento en el vector a ordenar.

```
dataset.txt
Archivo Editar Ver
369
795
356
651
565
199
370
129
268
825
842
368
668
8

Ln 7, Col 4 | 53 caracteres. | 100% | Windows (CRL) | UTF-8
```

# Descripción de Datasets

## 3.2 Algoritmos de Multiplicación de Matrices

El formato del archivo para este tipo de algoritmos será un archivo .txt que tendrá los datos en formato de texto. Como en los códigos se trabaja la multiplicación de matrices, la forma más óptima de tener los datos de prueba es en la forma que se muestra en la imagen, donde la primera fila se encuentra la dimensión de la matriz A y luego, para el ejemplo, las siguientes 4 líneas representan la matriz separando cada elemento por un espacio. Luego se representa la matriz B de la misma manera. Se consideró utilizar matrices cuadradas de la misma dimensión para poder hacer la comparación de rendimiento entre todos los algoritmos, ya que Strassen funciona bajo la multiplicación de matrices cuadradas de la misma dimensión y que su dimensión sea una potencia de 2 (que sea una potencia de dos queda bajo responsabilidad de lo que se ingrese por la terminal al ejecutar el generador de dataset).

Cada archivo de código .cpp procesa el dataset mediante una función auxiliar que lee el archivo y guarda las matrices en un vector de vectores, donde el vector principal contiene cada columna en un subvector.

The screenshot shows a Windows Notepad window with the title 'dataset: Bloc de notas'. The menu bar includes Archivo, Edición, Formato, Ver, and Ayuda. The content of the text file is as follows:

```
4 4
71 69 88 6
71 80 16 18
68 51 60 18
99 33 94 13
4 4
96 76 33 73
31 97 92 93
47 90 39 44
1 6 52 81
```

At the bottom of the window, there is a status bar with the text 'Línea 10, columna 100%' and encoding information 'Windows (CRLF) UTF-8'.

# 04

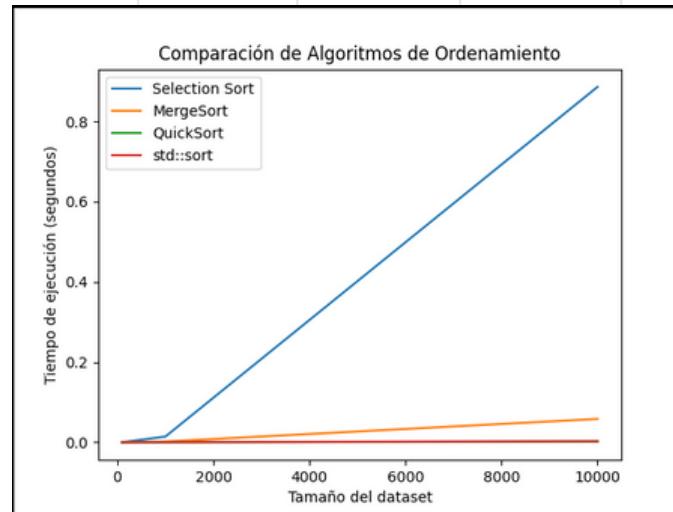


## Resultados Experimentales

### 4.1 Algoritmos de Ordenamiento

Los resultados temporales de los algoritmos se despliegan en la siguiente tabla y gráfico. Las mediciones fueron consideradas dados 10 mediciones y luego tomando el promedio de estas.

Dataset (tamaño)	std::sort	Selection Sort	MergeSort	QuickSort
100 elementos	0,0000127	0,0001531	0,0001503	0,0000092
1.000 elementos	0,0001894	0,0141991	0,0018330	0,0001383
10.000 elementos	0,0026113	0,8864563	0,0582050	0,0017175



Selection Sort: Algoritmo de complejidad  $O(n^2)$ .

MergeSort y QuickSort: Ambos más eficientes que el algoritmo anterior, con una complejidad de  $O(n \log n)$

Función de biblioteca estándar: Algoritmo similar al QuickSort, ya que está optimizado por el mismo lenguaje.

Analizando el gráfico, podemos ver como el algoritmo Selection Sort para tamaños de datos pequeños, no requiere tanto tiempo de ejecución, pero cuando los conjuntos de datos crecen el rendimiento decae rápidamente, aumentando su tiempo de ejecución, como se observa en el gráfico, la serie azul crece rápidamente a medida que crece el dataset, luego se puede ver la serie naranja que es mucha más eficiente que la explicada anteriormente dado que se trata del algoritmo MergeSort, el cual posee una complejidad de tiempo más estable en el peor caso, sin embargo, QuickSort se nota más eficiente (ver en el gráfico serie roja, ya que serie verde queda tapada por esta, sin embargo es poco apreciable la diferencia entre estas) debido a su menor uso de memoria y por ende, mejor aprovechamiento de la caché.

Para el caso de los 10.000 elementos se aprecia notablemente como SelectionSort es el más lento de los algoritmos a diferencia de los otros tres.

Por ende, algoritmos con mejor complejidad (QuickSort y MergeSort) superan a los de peor complejidad (Selection Sort en este caso) conforme el tamaño de los datos aumentan.

# 04

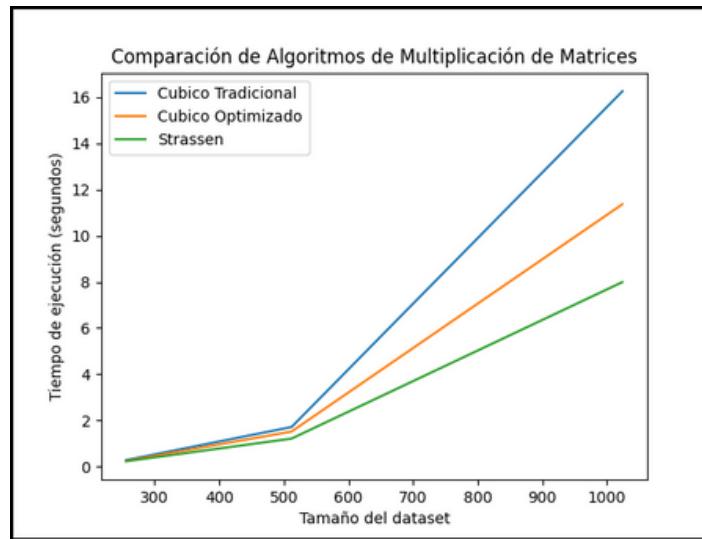


## Resultados Experimentales

### 4.2 Algoritmos de Multiplicación de Matrices

Los resultados temporales de los algoritmos se despliegan en la siguiente tabla y gráfico. Las mediciones fueron consideradas dados 10 mediciones y luego tomando el promedio de estas.

Dimensiones	Cúbico Tradicional	Cúbico Optimizado	Strassen
256x256	0,2795749	0,2435940	0,2287030
512x512	1,7117330	1,5153220	1,2097830
1024x1024	16,2580100	11,3642600	7,9914340



Considerando las restricciones que impone el algoritmo de Strassen los dataset son matrices cuadradas con dimensiones potencias de 2.

Algoritmo cúbico tradicional: Tiene una complejidad de  $O(n^3)$ , por lo que los tiempos de ejecución crecerán rápidamente conforme aumente el tamaño de las matrices.

Algoritmo optimizado: Se nota una mejora en la eficiencia comparado con el tradicional debido a la optimización por localidad de los datos (transponer la segunda matriz), ya que se utiliza un acceso secuencial a las filas de la matriz.

Strassen: A partir de matrices de tamaño mediano a grande (como 512x512 o 1024x1024), Strassen debería ser más eficiente que los otros dos debido a su complejidad de  $O(n^{2.81})$ , esto debido a que necesita de una llamada recursiva menos (7) a diferencia de los cúbicos.

Si bien para las matrices de 256x256 (dataset más pequeño probado) no hay mucha diferencia en tiempo de ejecución (ver tabla para más detalle), a medida que crecen los conjuntos de datos se ve que Strassen es mucho más eficiente.

Es por esto que Strassen es más eficiente en matrices grandes, más notoriamente en matrices de dimensión 1024x1024 o mayor, mientras que los algoritmos cúbicos, tradicional y optimizado, pierden eficiencia rápidamente.

# Conclusiones

# 05

El análisis asintótico es importante para entender el comportamiento de los algoritmos a medida que crece el tamaño del conjunto de datos. A pesar de esto, se debe tener en cuenta factores externos al programa que ejecute el algoritmo, como la arquitectura del sistema y las características de la caché, ya que a modo de experimento intente ejecutar el algoritmo de Strassen en dos terminales a la vez, y ambos resultados dieron el doble de tiempo de ejecución que si hubiera ejecutado una sola vez el programa. Así, el análisis asintótico se basa en modelos teóricos pero estos no capturan los detalles prácticos de la implementación.

A modo general, los algoritmos que resultaron más eficientes fue debido a su uso eficiente de memoria caché, además de minimizar el acceso a memoria que resulta costoso.

- En los algoritmos de ordenamiento, QuickSort es el más eficiente, debido a que usa menos memoria y aprovecha mejor la caché, sin embargo hay que tener precaución dado que en el peor de los casos, este algoritmo llega a ser menos eficiente.
- En los algoritmos de multiplicación de matrices, Strassen es el más eficiente, debido a que presenta una menor complejidad teórica, sin embargo su implementación es más compleja (código proporcionado por ChatGPT) y en el caso de matrices menores a las de 256x256 suele ser menos eficiente que los algoritmos cúbicos.

Las implementaciones in-place es ventajosa por el uso que se le da a la memoria, pero suelen ser más complicadas, dado que se trabaja modificando la memoria original sin necesidad de reservar un espacio adicional, como en el caso del algoritmo QuickSort. En cambio, las implementaciones no-in-place aumentan el uso de memoria y por ende el tiempo de ejecución, pero son más sencillos de implementar, como el caso del algoritmo MergeSort. Dependiendo del caso se evalúa cuál implementación utilizar.

La optimización, como en el caso de las multiplicaciones de matrices, pueden tener un gran impacto como el reducir aproximadamente a la mitad el tiempo de ejecución de un programa. En el caso del algoritmo cúbico optimizado se toma la decisión de transponer la matriz para alinear las filas de la matriz transpuesta con las columnas de la matriz que se está multiplicando, esto permite un acceso a memoria mucho más eficaz y eficiente, ya que reduce los fallos de caché y mejora el rendimiento del algoritmo en la práctica.

En general, la optimización es un proceso que se debería perseguir constantemente, ya que trae consigo muchos beneficios en la implementación de algoritmos, como mejorar el uso de los recursos, reduciendo uso de memoria y disminuyendo el tiempo de procesamiento, mejorar la experiencia de usuario, debido a lo que demora la respuesta, reducción de la complejidad, minimizando cálculos innecesarios y mejorando la estructura de datos, entre otros.

La optimización y la implementación eficiente juegan un papel crucial en el rendimiento real de los algoritmos