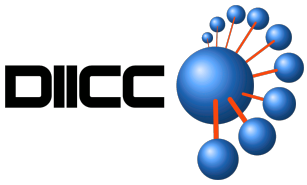


FUNDAMENTOS DE ESTRUCTURAS DE DATOS
Y ANÁLISIS DE ALGORITMOS

INFORME N°1

"ALGORITMOS DE ORDENAMIENTO Y MULTIPLICACIÓN DE MATRICES"

MAYO 2023

ERICH GERMÁN GRÜTTNER DÍAZ

Índice

1. Introducción

Este informe tiene como objetivo presentar una revisión sobre los algoritmos de ordenamiento y multiplicación de matrices más comunes e importantes en el ámbito de la informática. Los algoritmos de ordenamiento permiten la organización de datos de forma eficiente, mientras que los algoritmos de multiplicación de matrices son fundamentales en numerosas áreas de la ciencia y la ingeniería.

La medición de rendimiento de algoritmos es un proceso importante en la evaluación y optimización de algoritmos informáticos. La eficiencia de un algoritmo se puede medir en términos de tiempo de ejecución y consumo de recursos, como memoria y energía.

Una forma común de medir el rendimiento de un algoritmo es calcular su complejidad temporal y espacial. La complejidad temporal se refiere a la cantidad de tiempo que un algoritmo tarda en completar una tarea en función del tamaño de la entrada. La complejidad espacial, por otro lado, se refiere a la cantidad de memoria que el algoritmo requiere para procesar la entrada.

Una vez que se ha calculado la complejidad temporal y espacial de un algoritmo, se pueden realizar pruebas empíricas para verificar si los resultados teóricos se cumplen en la práctica. Para ello, se puede realizar una serie de pruebas utilizando diferentes conjuntos de datos de entrada, registrando el tiempo de ejecución y el consumo de recursos en cada caso.

Es importante tener en cuenta que la medición de rendimiento de un algoritmo no solo implica evaluar su eficiencia, sino también su precisión y exactitud. Un algoritmo que es rápido pero produce resultados inexactos no es útil en la mayoría de los casos.

Se espera experimentar con los algoritmos: Selection sort, Merge sort y Quick sort. Y, a su vez, a modo de complemento, se utilizará el comando sort contenido dentro de la librería STL.

Respecto a las matrices se experimentará con la multiplicación tradicional, la técnica de matriz traspuesta y el algoritmo de Strassen.

A través de las pruebas con distintos datasets, generados para este fin, se presentan con resultados, gráficos y un breve análisis del comportamiento observado.

2. Descripción de los algoritmos a ser comparados

2.1 Ordenamiento

Los algoritmos de ordenamiento son una parte esencial de la ciencia de la computación y se utilizan para organizar un conjunto de datos en un orden específico. Estos algoritmos son útiles en una amplia variedad de aplicaciones, desde la organización de archivos en una computadora hasta la búsqueda de elementos en una base de datos.

Cada algoritmo tiene su propia estrategia para ordenar los datos, pero todos comparten el objetivo común de organizar los elementos en una secuencia ordenada de acuerdo con un criterio determinado, como el orden numérico o alfabético. La elección del algoritmo de ordenamiento correcto depende del tamaño del conjunto de datos, la complejidad de los datos y la velocidad requerida para el procesamiento.

2.1.1 Selection Sort

Es un algoritmo de ordenamiento de comparación en el lugar, es decir, que ordena los datos en el mismo input sin requerir de alguna estructura externa [1].

Tiene una complejidad de tiempo de $O(n^2)$, lo que lo hace ineficiente para listas largas, y generalmente tiene peor desempeño que el “Insertion Sort”.

Se destaca por su simplicidad y tiene ventajas de desempeño sobre otros algoritmos más complejos en ciertas situaciones, particularmente cuando la memoria auxiliar es limitada.

Su funcionamiento es el siguiente:

- Buscar el mínimo elemento de la lista de entrada
- Intercambiarlo con el primero
- Buscar el siguiente mínimo en el resto de la lista
- Intercambiarlo con el segundo

Y en general:

- Buscar el mínimo elemento entre una posición i y el final de la lista
- Intercambiar el mínimo con el elemento de la posición i

Pseudocódigo:

```
funcion selection_sort(A):
    n = longitud(A)
    para i de 0 a n-1:
        minimo = i
        para j de i+1 a n:
            si A[j] < A[minimo]:
                minimo = j
        si minimo != i:
            intercambiar A[i] y A[minimo]
    retornar A
```

La complejidad temporal del algoritmo es de $O(n^2)$, lo que lo hace menos eficiente que otros algoritmos de ordenamiento como Quick sort o Merge sort, especialmente cuando se trata de arreglos muy grandes. Sin embargo, es fácil de implementar y puede ser útil en situaciones en las que la cantidad de elementos es relativamente pequeña.

2.1.2 Merge Sort

Es un algoritmo eficiente, de propósito general, y del tipo de ordenamiento basado en comparaciones. La mayoría de sus implementaciones producen ordenamientos estables, lo que significa que el orden de los elementos es el mismo en el input que en el output. Es de tipo “dividir para conquistar”, y fue inventado por John Von Neumann en 1945. [2]

Funciona dividiendo la lista a ordenar en dos mitades iguales y luego ordenándolas por separado. Una vez que ambas mitades están ordenadas, el algoritmo combina las dos sublistas ordenadas para obtener una lista ordenada completa.

El proceso de combinación se realiza comparando el primer elemento de cada sublista y seleccionando el más pequeño. Este elemento se coloca en la posición correspondiente en la nueva lista ordenada, y el proceso continúa hasta que todas las sublistas se hayan combinado.

El Merge Sort es un algoritmo de ordenamiento estable y tiene una complejidad de tiempo promedio de O(n log n), lo que lo hace adecuado para listas grandes. También es fácilmente paralelizable, lo que lo hace ideal para su uso en sistemas con múltiples núcleos de procesamiento.

Pseudocódigo:

```
Función mergeSort(lista)
    Si la longitud de la lista es 1 o menos, devuelve la lista
    Divide la lista en dos sublistas de tamaño aproximadamente igual
    Llamada recursiva a mergeSort en cada sublista
    Combina las dos sublistas ordenadas en una lista ordenada
    Devuelve la lista ordenada

Función combinar(sublistaIzquierda, sublistaDerecha)
    Crea una lista vacía para almacenar la lista combinada
    Mientras haya elementos en ambas sublistas
        Si el primer elemento de la sublista izquierda es menor o igual que el de la
derecha
            Agrega el primer elemento de la sublista izquierda a la lista combinada
            Elimina el primer elemento de la sublista izquierda
        De lo contrario
            Agrega el primer elemento de la sublista derecha a la lista combinada
            Elimina el primer elemento de la sublista derecha
    Agrega los elementos restantes de la sublista izquierda a la lista combinada
    Agrega los elementos restantes de la sublista derecha a la lista combinada
    Devuelve la lista combinada
```

2.1.3 Quick Sort

Es un algoritmo de ordenamiento muy eficiente que utiliza la técnica de "dividir para conquistar". Fue desarrollado por Tony Hoare en 1959 y es uno de los algoritmos de ordenamiento más utilizados en la actualidad [3].

El algoritmo Quick Sort funciona dividiendo la lista a ser ordenada en dos partes, la mitad superior y la mitad inferior, con respecto a un elemento elegido como pivote. Luego, se ordenan ambas mitades de forma recursiva, utilizando el mismo proceso de partición. La partición consiste en seleccionar un elemento del subconjunto de la lista y colocar todos los elementos menores que el pivote a su izquierda y todos los mayores a su derecha. El pivote se ubica en su posición final en la lista ordenada y se repite este proceso en cada subconjunto de la lista hasta que toda la lista esté ordenada.

El proceso de selección del pivote es importante para la eficiencia del algoritmo. Una forma común de hacerlo es elegir el primer elemento de la lista, aunque esto puede llevar a un rendimiento ineficiente en casos específicos. Otra estrategia es elegir un elemento al azar de la lista, lo que aumenta la probabilidad de una partición equilibrada.

Quick Sort es un algoritmo muy rápido y eficiente, con una complejidad de tiempo promedio de O(n log n), lo que lo hace adecuado para listas grandes. Sin embargo, su complejidad de tiempo en el peor caso puede ser $O(n^2)$, lo que lo hace menos adecuado para listas con muchos elementos repetidos. También es posible que el Quick Sort no sea tan fácilmente paralelizable como otros algoritmos de ordenamiento.

En general, el Quick Sort es una excelente opción para ordenar grandes cantidades de datos en poco tiempo, siempre y cuando se seleccione adecuadamente el pivote y se implemente correctamente.

Pseudocódigo: (asume que la lista está ordenada de forma ascendente)

```
Función quickSort(lista, izquierda, derecha)
    Si izquierda < derecha
        pivote = partición(lista, izquierda, derecha)
        Llamada recursiva a quickSort en la lista de la izquierda del pivote
        Llamada recursiva a quickSort en la lista de la derecha del pivote

Función partición(lista, izquierda, derecha)
    Selecciona el último elemento de la lista como pivote
    i = izquierda - 1
    Para j de izquierda a derecha - 1
        Si lista[j] <= pivote
            Incrementa i
            Intercambia lista[i] y lista[j]
    Intercambia lista[i+1] y lista[derecha]
    Devuelve i + 1
```

2.1.4 Sort Interno C++

Es una función genérica de C++, presente en la librería estándar (STL), y que permite ordenamiento por comparación [4].

Una llamada a sort debe ejecutar no más de O(N Log N) comparaciones cuando se aplica a un rango de N elementos

Se incluye en la librería <algorithm> y requiere 3 argumentos: RandomAccessIterator first, RandomAccessIterator last y Compare comp. Los primeros definen el inicio y el final de una secuencia de valores. El último debe ser alcanzable desde el primero aplicando repetidamente un incremento del operador. Mientas que el tercer argumento denota el predicado de la comparación, definiendo un ordenamiento débil estricto sobre los elementos de la secuencia a ordenar. Es opcional y si no es ingresado, se asume el “menor-que” (<).

Diferentes implementaciones utilizan diferentes algoritmos. La librería GNU estándar de C++, por ejemplo, utiliza un algoritmo de 3 partes híbrido: Introsort es ejecutado primero a una máxima profundidad dada por 2 x $\log_2 n$, donde n es el número de elementos, seguido por un Insertion Sort sobre el resultado.

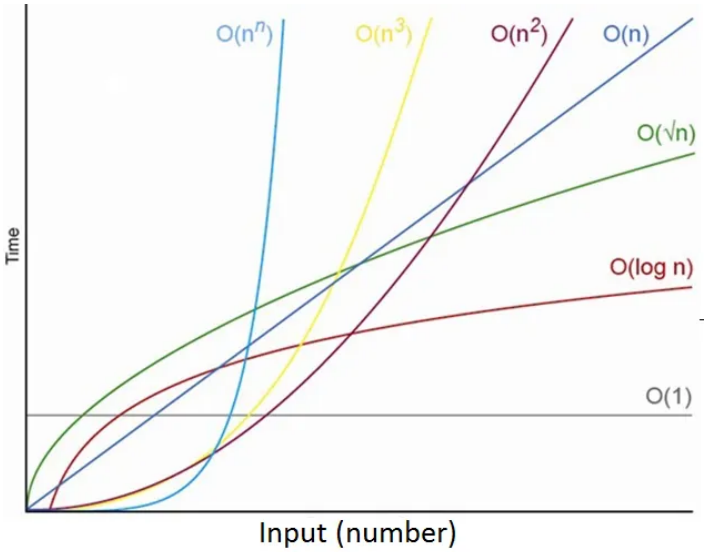
Ejemplo usando vectores en C++

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {

  vector<int> vec = { 23, 5, -10, 0, 0, 321, 1, 2, 99, 30 };

  sort(vec.begin(), vec.end());

}
```

2.1.5 Tabla resumen algoritmos de ordenamiento a analizar en este informe

Nombre	Mejor caso	Promedio	Peor caso	Estable
Selection sort	n^2	n^2	n^2	No
Merge sort	n log n	n log n	n log n	Sí
Quicksort	n log n	n log n	n^2	No
SortInterno (Introsort)	n log n	n log n	n log n	No

Crecimiento de los algoritmos según su orden de complejidad [5]



Este gráfico permitirá validar (o no) el comportamiento, en rendimiento, de los diversos algoritmos en comparación a los resultados obtenidos.

2.2 Multiplicación de matrices

La multiplicación de matrices es una operación matemática fundamental que permite combinar dos o más matrices para producir una nueva matriz. Este proceso es utilizado en una amplia variedad de aplicaciones en las matemáticas, la física, la ingeniería y la computación. El primer acercamiento a su cálculo se le atribuye a Jacques Philippe Marie Binet, en 1812. [6]

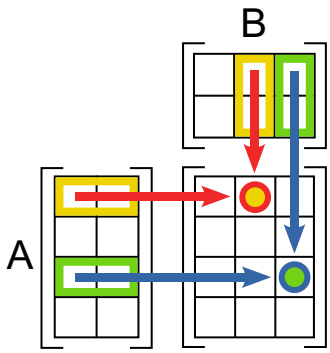
Dadas dos matrices A y B, tales que $A \in \mathbb{R}^{mxn}$ y $B \in \mathbb{R}^{nxp}$, la multiplicación de A por B, que se denota AB, es una matriz con m filas y p columnas cuya (i,j)-ésima entrada es:

$$\sum_{r=1}^{n} a_{ir} b_{rj}$$

$$\boldsymbol{AB} = \begin{pmatrix} \boldsymbol{a_{11}} & \cdots & \boldsymbol{a_{1n}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{a_{m1}} & \cdots & \boldsymbol{a_{mn}} \end{pmatrix} \begin{pmatrix} \boldsymbol{b_{11}} & \cdots & \boldsymbol{b_{1p}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{b_{n1}} & \cdots & \boldsymbol{b_{np}} \end{pmatrix}$$

$$\boldsymbol{AB} = \begin{pmatrix} \boldsymbol{a_{11}b_{11}} + & \cdots & \boldsymbol{+ a_{1n}b_{n1}} \cdots & \boldsymbol{a_{1n}b_{n1}} + & \cdots & \boldsymbol{+ a_{1n}b_{np}} \\ & \vdots & & & \vdots & \\ \boldsymbol{a_{m1}b_{11}} + & \cdots & \boldsymbol{+ a_{mn}b_{n1}} \cdots & \boldsymbol{a_{m1}b_{1p}} + & \cdots & \boldsymbol{+ a_{mn}b_{np}} \end{pmatrix}$$

Los resultados en las posiciones marcadas dependen de las filas y columnas de sus respectivos colores [7]:



El cálculo tradicional de multiplicación de matrices puede ser un proceso intensivo en términos de cálculo, especialmente cuando se multiplican matrices grandes. Para mejorar la eficiencia del proceso, se han desarrollado algoritmos y técnicas de cálculo más avanzados, como el algoritmo de Strassen. Estas técnicas utilizan métodos más sofisticados para reducir el número de operaciones necesarias para multiplicar matrices grandes, lo que puede mejorar significativamente la eficiencia del proceso.

La multiplicación de matrices es útil en una amplia variedad de aplicaciones. En la física, se utiliza para calcular las transformaciones de coordenadas en sistemas de referencia diferentes. En la ingeniería, se utiliza para calcular los efectos de los materiales y las cargas en las estructuras. En la informática, se utiliza para procesar y analizar grandes cantidades de datos en sistemas de inteligencia artificial y aprendizaje automático.

En el presente informe, se consideraron los siguientes 3 algoritmos para el cálculo de multiplicación de matrices: algoritmo iterativo cúbico tradicional, algoritmo iterativo cúbico optimizado para mantener la localidad de los datos y algoritmo de Strassen.

2.2.1 Algoritmo iterativo cúbico tradicional

El algoritmo de multiplicación estándar es el método más común para multiplicar dos matrices. Este algoritmo utiliza la regla de multiplicación de matrices y se basa en la técnica de "fila por columna" para multiplicar los elementos de las matrices.

Funciona multiplicando cada elemento de una fila de la matriz A por cada elemento de una columna de la matriz B y sumando los productos resultantes. Este proceso se repite para cada fila de A y cada columna de B, y los resultados se combinan para formar la matriz resultante C.

Es un algoritmo fácil de implementar y se puede utilizar para multiplicar matrices de cualquier tamaño. Sin embargo, este algoritmo puede ser ineficiente para matrices grandes, ya que requiere un gran número de operaciones de multiplicación y suma. Además, el orden en que se multiplican las matrices puede afectar el número de operaciones necesarias para realizar la multiplicación.

Pseudocódigo:

```
funcion multiplicar_matrices(A, B):
    si el número de columnas de A es diferente al número de filas de B:
        imprimir "No se pueden multiplicar las matrices"
        retornar
    sino:
        crear una matriz C con tamaño de filas de A y columnas de B
        para i desde 0 hasta el número de filas de A:
            para j desde 0 hasta el número de columnas de B:
                suma = 0
                para k desde 0 hasta el número de columnas de A:
                    suma = suma + A[i][k] * B[k][j]
                C[i][j] = suma
        retornar C
```

2.2.2 Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos

Este algoritmo es una variante del algoritmo de multiplicación estándar que utiliza la matriz transpuesta de B para multiplicar las filas de A por las filas de B. Puede ser muy eficiente en algunas situaciones, especialmente cuando las matrices son grandes y dispersas (tienen muchos ceros), pero puede ser menos eficiente que el algoritmo de multiplicación estándar cuando las matrices son pequeñas o densas.

Pseudocódigo:

```
funcion multiplicar_matrices_transpuesta(A, B):
    si el número de columnas de A es diferente al número de filas de B:
        imprimir "No se pueden multiplicar las matrices"
        retornar
    sino:
        crear una matriz C con tamaño de filas de A y columnas de B
        B_transpuesta = transponer(B)
        para i desde 0 hasta el número de filas de A:
            para j desde 0 hasta el número de columnas de B:
                producto_punto = 0
                para k desde 0 hasta el número de columnas de A:
                    producto_punto += A[i][k] * B_transpuesta[j][k]
                C[i][j] = producto_punto
        retornar C
```

El algoritmo cúbico optimizado para mantener la localidad de los datos puede ser más eficiente que el algoritmo de multiplicación estándar en ciertas situaciones debido a la forma en que utiliza la memoria y la CPU. La transposición puede mejorar el uso de la caché y reducir la cantidad de datos que deben leerse de la memoria.

Sin embargo, es importante tener en cuenta que la eficiencia del algoritmo de multiplicación de matrices usando transpuesta depende del tamaño y la densidad de las matrices, así como de las características específicas de la arquitectura de la CPU y la memoria. En algunos casos, el algoritmo de multiplicación estándar puede ser más rápido o más eficiente que el algoritmo de multiplicación usando transpuesta. Por lo tanto, es importante evaluar cuidadosamente el rendimiento de ambos algoritmos para determinar cuál es el más adecuado para una tarea específica.

2.2.3 Algoritmo Strassen

El algoritmo de Strassen es un algoritmo eficiente para la multiplicación de matrices grandes y se basa en la división y conquista. Este algoritmo divide las matrices en submatrices más pequeñas, realiza cálculos recursivamente en estas submatrices y luego combina los resultados para obtener el resultado final. [8]

Volker Strassen publicó este algoritmo en 1969. Pese a que es ligeramente más rápido que el algoritmo tradicional, fue el primero en señalar que el enfoque estándar no es óptimo.

Pseudocódigo:

```
funcion multiplicar_matrices_strassen(A, B):
    n = tamaño de A (o B), siendo n una potencia de 2
    si n == 1:
        retornar el producto de A y B
    sino:
        # División
        A11, A12, A21, A22 = dividir(A)
        B11, B12, B21, B22 = dividir(B)

        # Conquista
        P1 = multiplicar_matrices_strassen(A11 + A22, B11 + B22)
        P2 = multiplicar_matrices_strassen(A21 + A22, B11)
        P3 = multiplicar_matrices_strassen(A11, B12 - B22)
        P4 = multiplicar_matrices_strassen(A22, B21 - B11)
        P5 = multiplicar_matrices_strassen(A11 + A12, B22)
        P6 = multiplicar_matrices_strassen(A21 - A11, B11 + B12)
        P7 = multiplicar_matrices_strassen(A12 - A22, B21 + B22)

        # Combinación
        C11 = P1 + P4 - P5 + P7
        C12 = P3 + P5
        C21 = P2 + P4
        C22 = P1 - P2 + P3 + P6

        # Combinar submatrices en matriz resultado
        C = combinar(C11, C12, C21, C22)

        retornar C
```

3. Descripción de los datasets

3.1 Datasets para algoritmos de ordenamiento

A continuación, se presentan los distintos sets de datos utilizados para realizar las evaluaciones de rendimiento de algoritmos de ordenamiento.

Para ello se construyó una herramienta en C++ generadora de archivos de texto de input, ubicada dentro del repositorio. [9]

Mediante un menú básico, se ofrece al usuario la opción de 5 tipos de datasets, los que se describen a continuación.

3.1.1 Inputs

Los archivos de **input** tienen formato .txt, nombre "input" + número correlativo de archivo, y tienen la siguiente estructura interna:

inputxx.txt

Tamaño vector
Dato1
...
Dato n

Para el presente informe se consideraron los siguientes parámetros:

- **10** archivos por generación
- El primer archivo contiene **10.000** registros, el segundo **20.00**0 y así hasta el último que contiene **100.000**
- El rango de números en el vector va desde el número **0** hasta el **10.000**

Para cada uno de esos datasets, se consideraron 5 tipos de "desorden":

- **Desordenado repetido:** números al azar dentro del vector de salida, pero sin restringir la aparición de dos o más veces del mismo número.
- **Desordenado único**: números al azar dentro del vector de salida, pero con la condición de que solamente aparezcan una vez en el registro.
- **Ordenado repetido**: la misma generación del vector "desordenado repetido", pero esta vez se deja ordenado en forma ascendente para su procesamiento.
- **Ordenado reversa (descendente)**: la misma generación del vector "desordenado repetido", pero esta vez se deja ordenado en forma descendente para su procesamiento
- **Parcialmente ordenado**: la misma generación del vector "desordenado repetido", pero se le aplica un orden parcial a la mitad de los registros

Por lo tanto, la estructura de datasets utilizada quedó organizada de la siguiente forma:

	input1.txt	input2.txt	...	input10.txt
Desordenado repetido (carpeta INPUT1)	10000	20000	...	100000
	Dato 1	Dato 1	...	Dato 1
	...	...	...	...
	Dato 10000	Dato 20000	...	Dato 100000

	input1.txt	input2.txt	...	input10.txt
Desordenado único (carpeta INPUT 2)	10000	20000	...	100000
	Dato 1	Dato 1	...	Dato 1
	...	...	...	...
	Dato 10000	Dato 20000	...	Dato 100000

	input1.txt	input2.txt	...	input10.txt
Ordenado repetido (carpeta INPUT3)	10000	20000	...	100000
	Dato 1	Dato 1	...	Dato 1
	...	...	...	...
	Dato 10000	Dato 20000	...	Dato 100000

	input1.txt	input2.txt	...	input10.txt
Ordenado reversa (carpeta INPUT 4)	10000	20000	...	100000
	Dato 1	Dato 1	...	Dato 1
	...	...	...	...
	Dato 10000	Dato 20000	...	Dato 100000

	input1.txt	input2.txt	...	input10.txt
Parcialmente ordenado (carpeta INPUT 5)	10000	20000	...	100000
	Dato 1	Dato 1	...	Dato 1
	...	...	...	...
	Dato 10000	Dato 20000	...	Dato 100000

3.1.2 Outputs

Los archivos de **output** tienen formato .txt, nombre "output" + número correlativo de archivo, y tienen siguiente la siguiente estructura interna:

output.txt

INICIO
Algoritmo seleccionado: mergesort
Vector inicial:
Dato 1
...
Dato n
Resultado:
Dato 1 (procesado)
...
Dato n (procesado)

Se pueden encontrar en el repositorio en [10]

3.1.3 Archivos para graficar

Para facilitar la generación de gráficos para la observación del rendimiento de los algoritmos, se crearon archivos del tipo CSV con el siguiente formato:

nombre_algoritmo_ordenamiento_results.csv

n, tiempo[ms]
valor 1, tiempo 1
...
valor n, tiempo n

Estos archivos se pueden acceder en el repositorio en [11]

Están agrupados por generación, vale decir, para este experimento se consideraron carpetas csv1, csv2, csv3, csv4 y csv5.

3.2 Datasets para multiplicación de matrices

A continuación, se presentan los distintos sets de datos utilizados para realizar las evaluaciones de rendimiento para multiplicación de matrices.

Para ello se construyó una herramienta en C++ generadora de archivos de texto de input, ubicada dentro del repositorio. [12]

Mediante un menú básico, se ofrece al usuario la opción de 5 tipos de datasets, los que se describen a continuación.

3.1.1 Inputs

Los archivos de **input** tienen formato .txt, nombre "input" + número correlativo de archivo, y tienen la siguiente estructura interna:

inputxx.txt

n,m,k
Matriz A
...
Matriz B

Donde n, m son las columnas y filas de la matriz A, y k son las filas de la matriz B (se asume como m la cantidad de columnas de la matriz B).

Para el presente informe se consideraron los siguientes parámetros:

- **10** archivos por generación
- El rango de números en el vector va desde el número **0** hasta el **10.000**
- Los tamaños varían desde 100 hasta 1000, y en el caso de potencia de 2, hasta 1024.

Para cada uno de los datasets, se consideraron los siguientes 5 casos:

- Matrices cuadradas (n=m=k)
- Matrices cuadradas con tamaño potencia de 2 (n=m=k, potencia de 2)
- Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas variables, columnas fijas) (n fijo, m creciente, k fijo)
- Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas y columnas variables) (n fijo, m creciente, k creciente)
- Matrices rectangulares con matriz A (filas y columnas variables) y matriz B (filas variables y columnas fijas) (n creciente, m creciente y k fijo)

Por lo tanto, la estructura de datasets utilizada quedó organizada de la siguiente forma:

	input1.txt	input2.txt	...	input10.txt
Matrices cuadradas (carpeta INPUT1)	100 100 100	200 200 200	...	1000 1000 1000
	Matriz A	Matriz A	...	Matriz A
	...	...	...	...
	Matriz B	Matriz B	...	Matriz B

	input1.txt	input2.txt	...	input10.txt
Matrices cuadradas 2^n (carpeta INPUT 2)	2 2 2	4 4 4	...	1024 1024 1024
	Matriz A	Matriz A	...	Matriz A
	...	...	...	...
	Matriz B	Matriz B	...	Matriz B

	input1.txt	input2.txt	...	input10.txt
Matrices rectangulares 1 (carpeta INPUT3)	100 100 100	100 200 100	...	100 1000 100
	Matriz A	Matriz A	...	Matriz A
	...	...	...	...
	Matriz B	Matriz B	...	Matriz B

	input1.txt	input2.txt	...	input10.txt
Matrices rectangulares 2 (carpeta INPUT 4)	100 100 100	100 200 200	...	100 1000 10000
	Matriz A	Matriz A	...	Matriz A
	...	...	...	...
	Matriz B	Matriz B	...	Matriz B

	input1.txt	input2.txt	...	input10.txt
Matrices rectangulares 3 (carpeta INPUT 5)	100 100 100	200 200 100	...	1000 1000 100
	Matriz A	Matriz A	...	Matriz A
	...	...	...	...
	Matriz B	Matriz B	...	Matriz B

3.1.2 Outputs

Los archivos de **output** tienen formato .txt, nombre "output" + número correlativo de archivo, y tienen siguiente la siguiente estructura interna:

output.txt

n: valor
m: valor
k: valor
Matriz A
...
Matriz B
...
Resultado
...

Se pueden encontrar en el repositorio en [13].

3.1.3 Archivos para graficar

Para facilitar la generación de gráficos para la observación del rendimiento de los algoritmos, se crearon archivos del tipo CSV con el siguiente formato:

nombre_algoritmo_ordenamiento_results.csv

n, tiempo[ms]
valor 1, tiempo 1
...
valor n, tiempo n

Estos archivos se pueden acceder en el repositorio en [14].

Están agrupados por generación, vale decir, para este experimento se consideraron carpetas csv1, csv2, csv3, csv4 y csv5.

4. Resultados experimentales

Para la realización de las pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria. El chip M1 tiene 8 núcleos (4 de alta eficiencia a 3.2 GHz + 4 de alto rendimiento a 2.0 GHz) y una velocidad de transferencia de 50Gb por segundo.

El código fuente para algoritmos de ordenamiento está disponible en [17] y para multiplicación de matrices en [18].

Forma de realizar las mediciones (ordenamiento y multiplicación de matrices)

- Generación de datasets
- Utilización de script bash para la llamada de cada dataset

Ejemplo, para la ejecución de script de multiplicación de matrices:

```
num_datasets=5

for (( i=1; i <= $num_datasets; ++i ))
do
    make num_dataset=$i

    python3 plot.py csv/csv$i/standard_results.csv
    python3 plot.py csv/csv$i/transpose_results.csv
    python3 plot.py csv/csv$i/strassen_results.csv
    python3 plot_dos.py csv/csv$i/standard_results.csv csv/csv$i/transpose_results.csv
    python3 plot_todos.py csv/csv$i/standard_results.csv csv/csv$i/transpose_results.csv
csv/csv$i/strassen_results.csv

done
```

Este script ejecuta el llamado a make y luego muestra en pantalla los gráficos generados a partir de los archivos CSV producidos.

Para calcular el tiempo de ejecución, se utilizó la librería Chrono de C++:

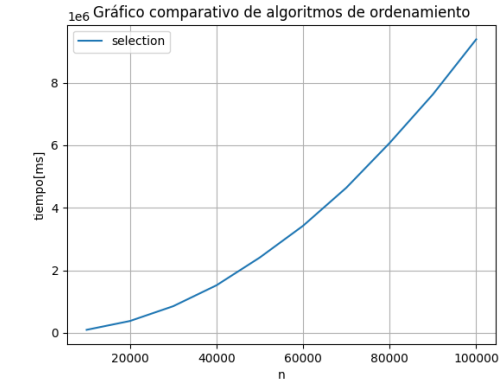
```
long long execution_time_ms(Func function, const vector<int> &A, string alg) {
  auto start_time = std::chrono::high_resolution_clock::now();
  function(A, alg);
  auto end_time = std::chrono::high_resolution_clock::now();
  return std::chrono::duration_cast<std::chrono::microseconds>(end_time -
start_time).count();
}
```

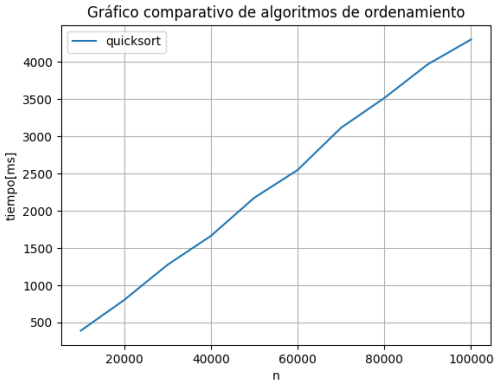
Donde “alg” corresponde al algoritmo a medir (ej: Quick sort)

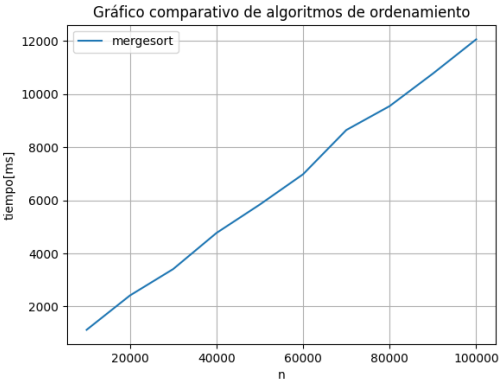
Para graficar se usó la librería matplotlib de Python. Los gráficos para algoritmos de ordenamiento se encuentran en [15] y los gráficos asociados a la multiplicación de matrices se encuentran en [16].

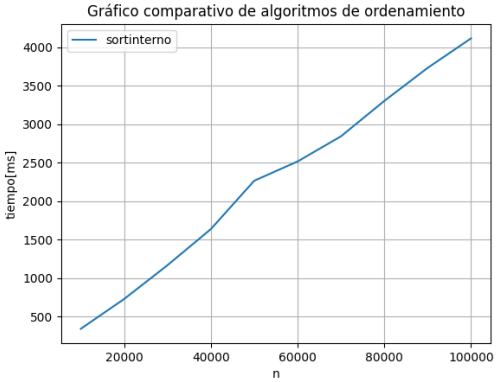
4.1 Medición rendimiento algoritmos de ordenamiento

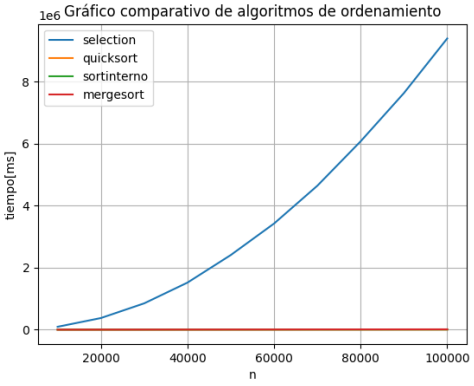
4.1.1 Caso 1: Datos desordenados repetidos

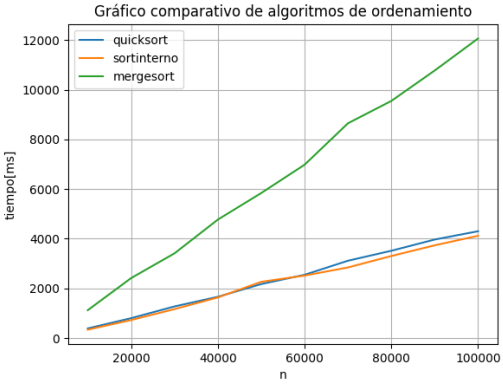








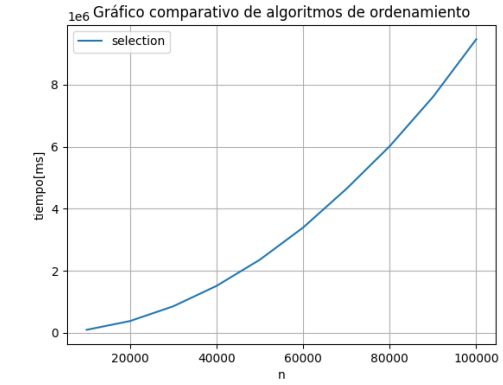


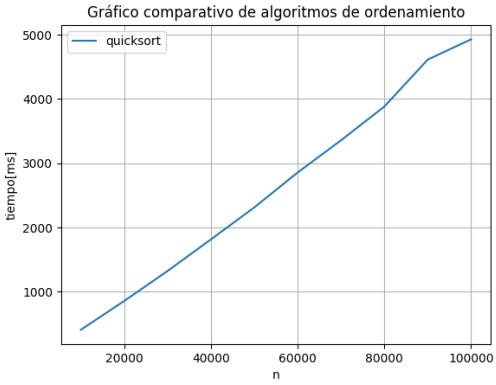


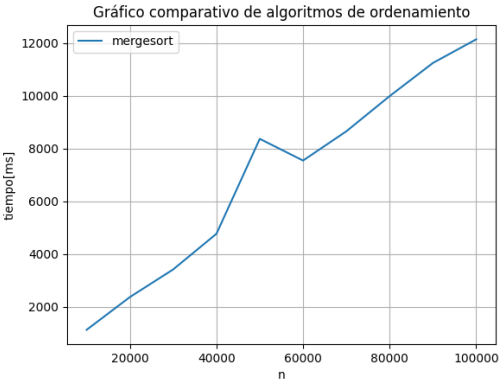
Observaciones:

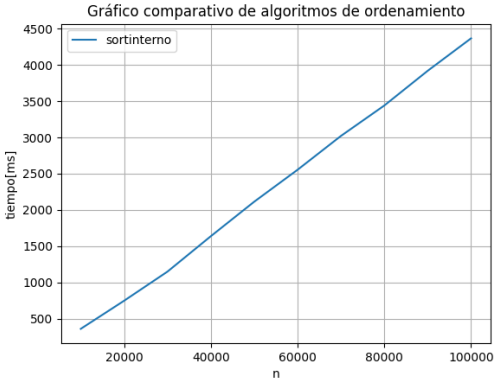
- El algoritmo “Selection” es el de peor rendimiento al ser de orden $O(n^2)$
- Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. De ello se puede inferir que el peor fue “Mergesort”, y que “Quicksort” y “SortInterno” tuvieron un desempeño muy similar.

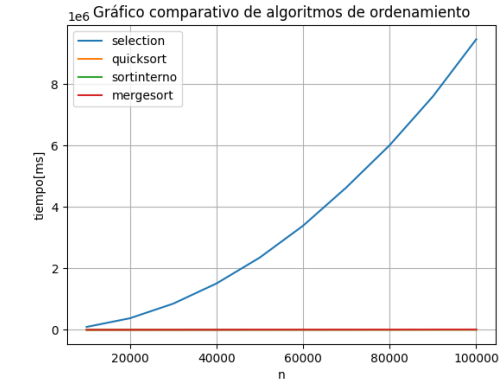
4.1.2 Caso 2: Datos desordenados con valores únicos

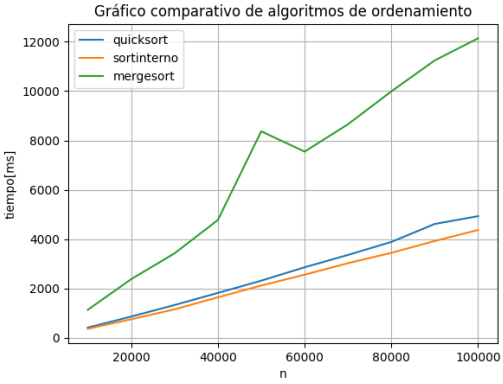








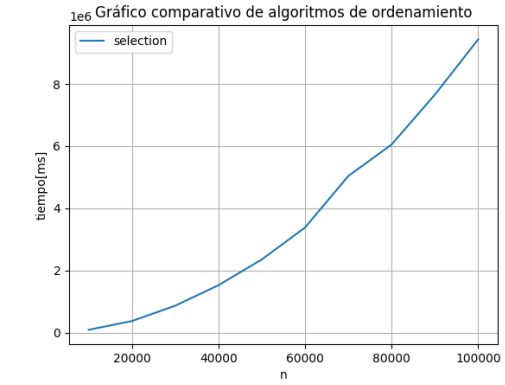


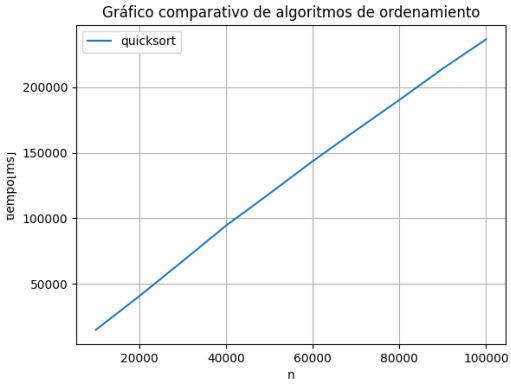


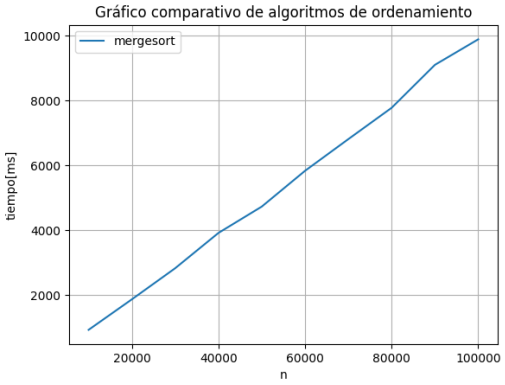
Observaciones:

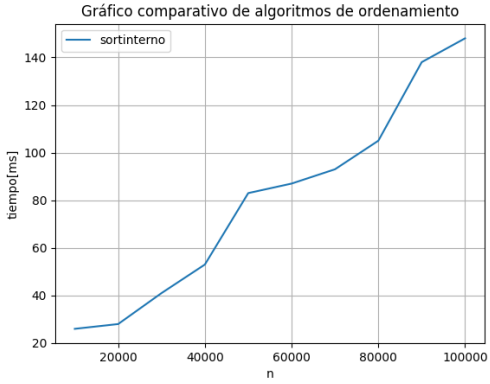
- El algoritmo “Selection” es el de peor rendimiento al ser de orden $O(n^2)$
- Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. De ello se puede inferir que el peor fue “Mergesort”, y que el “SortInterno” fue ligeramente superior al “Quicksort”.

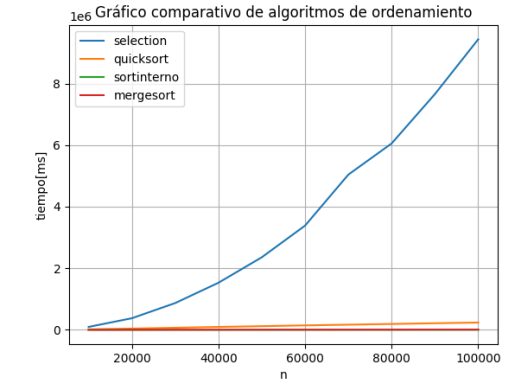
4.1.3 Caso 3: Datos ordenados repetidos

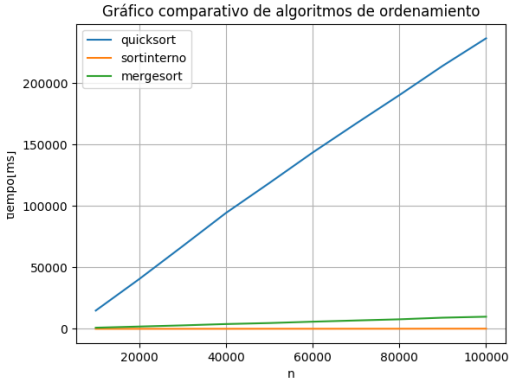








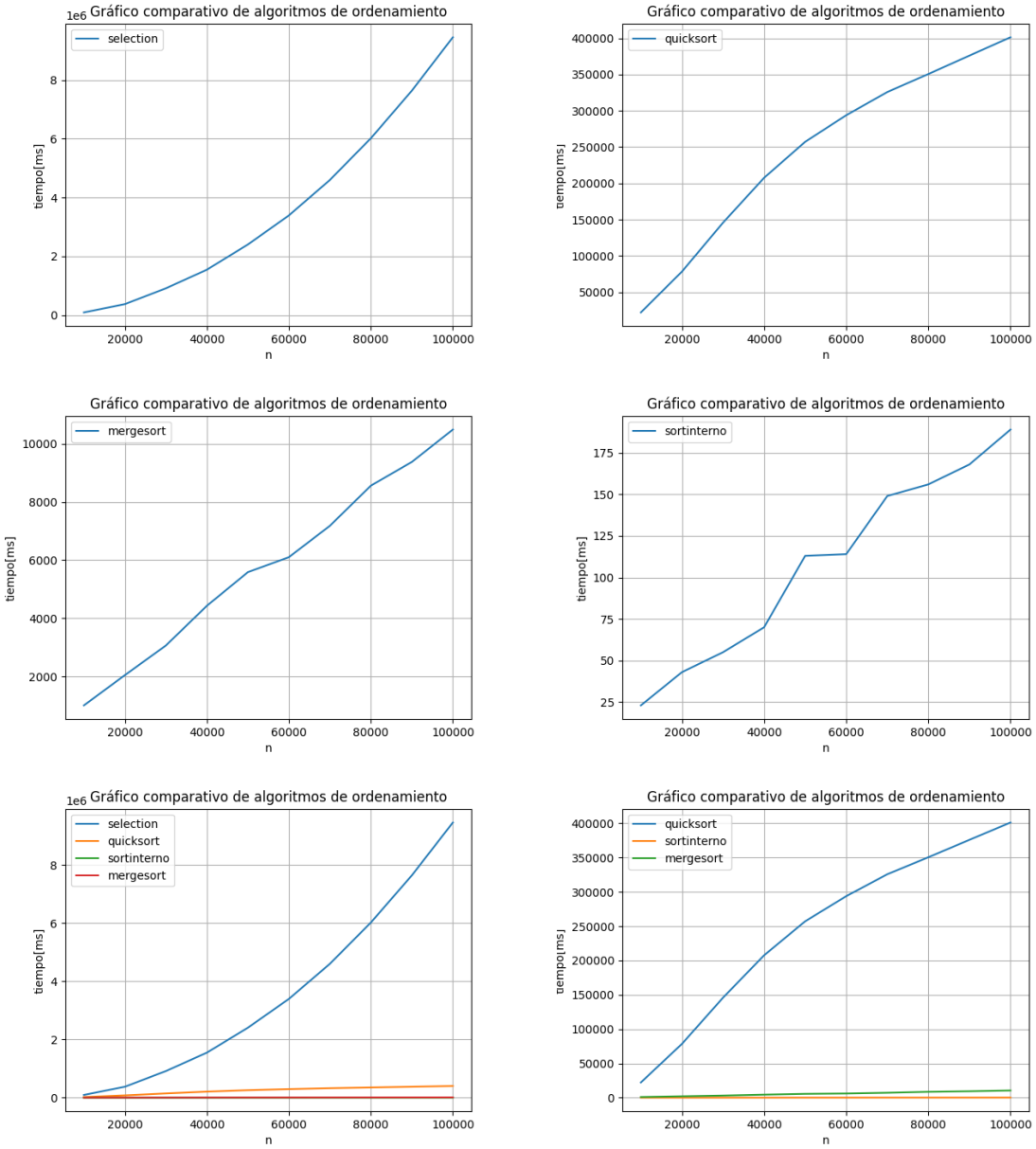




Observaciones:

- El algoritmo “Selection” es el de peor rendimiento al ser de orden $O(n^2)$
- Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. En este caso “Quicksort” tuvo el peor rendimiento, siendo el mejor “SortInterno” seguido por “Mergesort”.

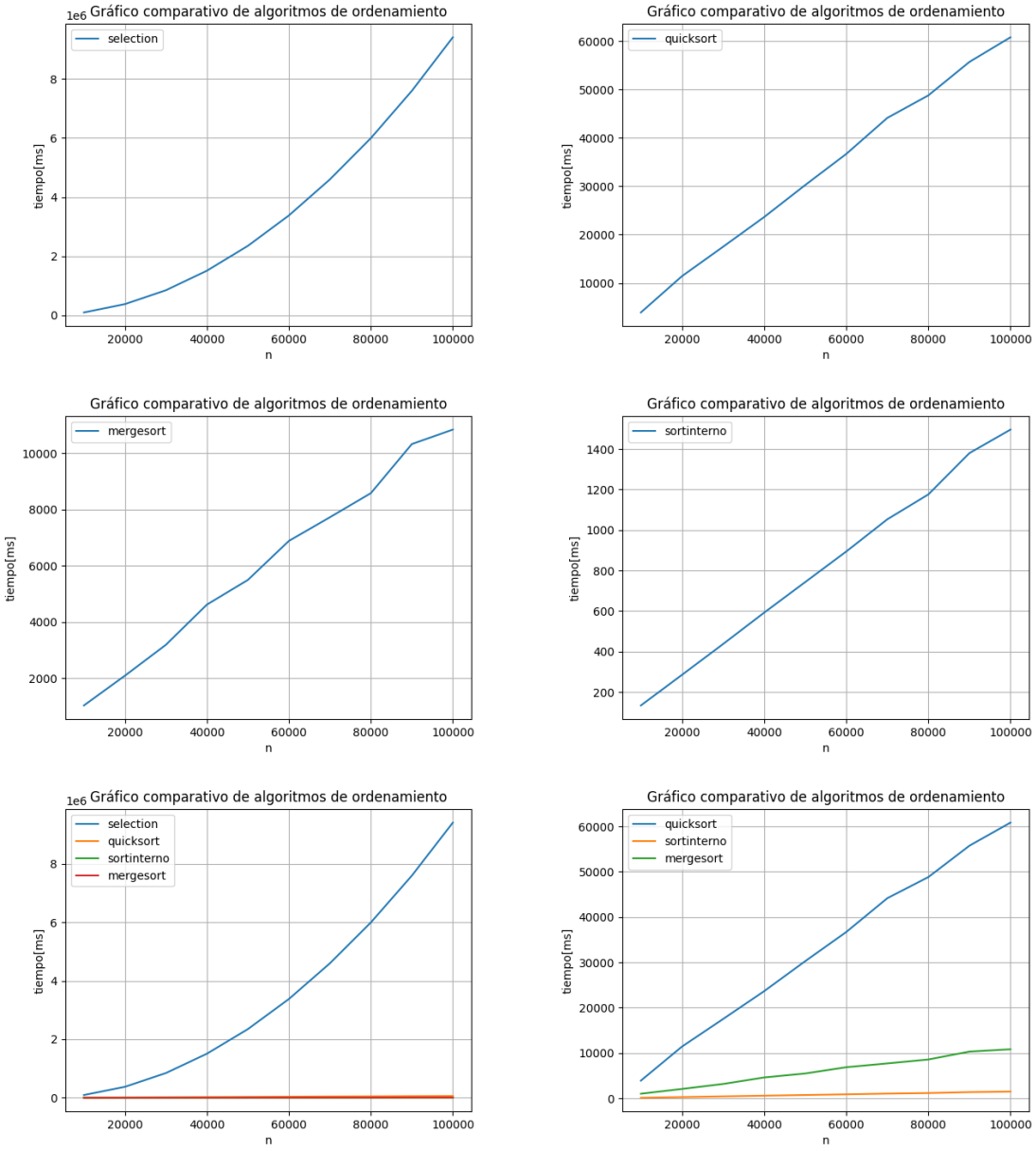
4.1.4 Caso 4: Datos ordenados reversa (descendente)



Observaciones:

- El algoritmo “Selection” es el de peor rendimiento al ser de orden $O(n^2)$
- Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. Pese a la cercanía del desempeño en el gráfico, “SortInterno” sigue siendo el que tiene mejor desempeño por una amplia diferencia.

4.1.5 Caso 5: Datos parcialmente ordenados

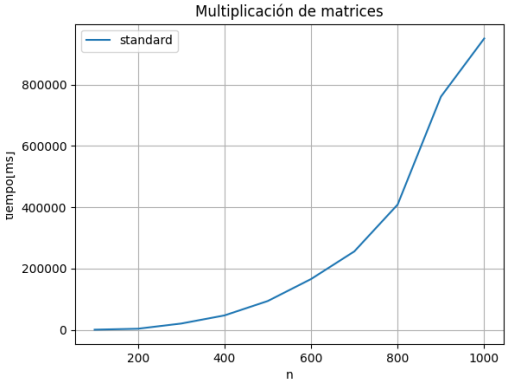


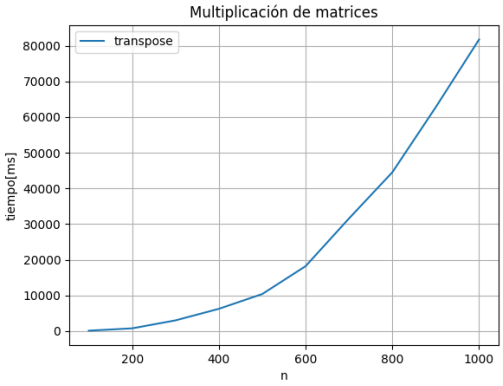
Observaciones:

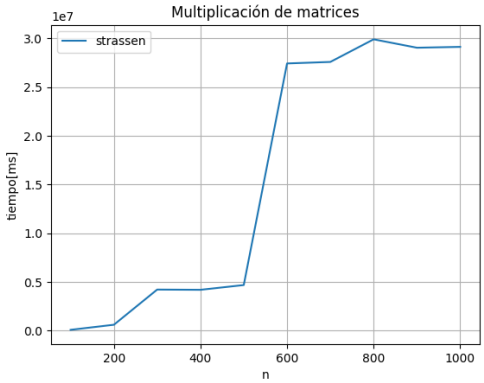
- El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n^2)
- Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. Para tamaños menores (20.000) no es tan notoria como al llegar a tamaños grandes (100.000), donde nuevamente “Sort Interno” tiene mejor desempeño.

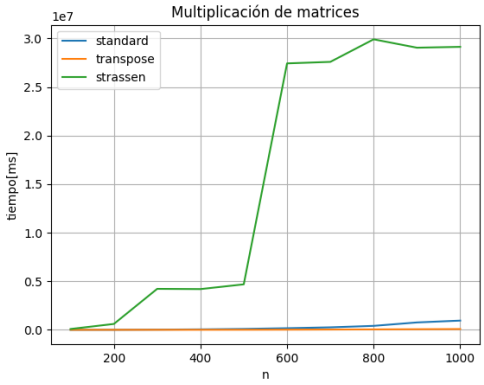
4.2 Medición rendimiento para multiplicación de matrices

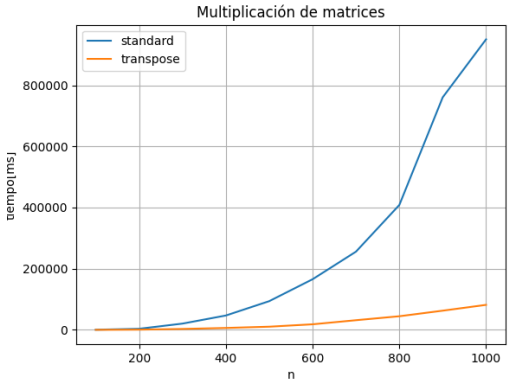
4.2.1 Caso 1: Matrices cuadradas







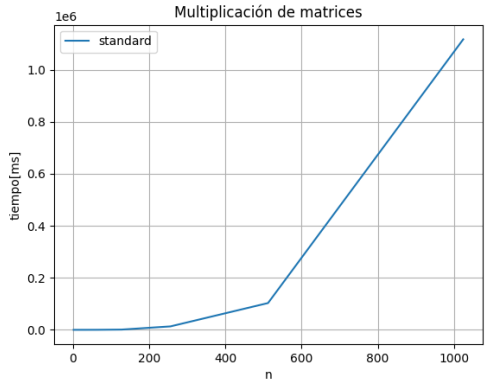


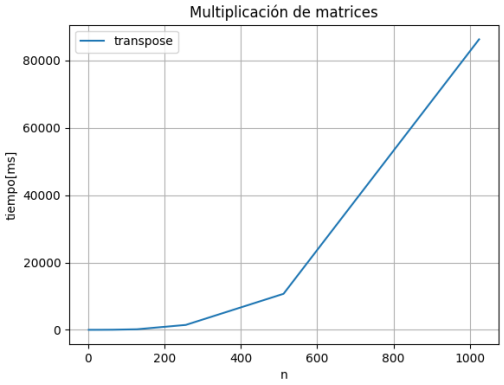


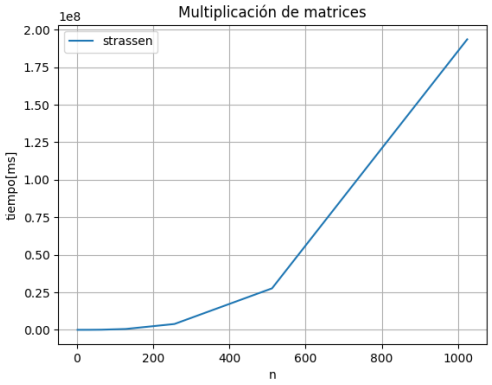
Observaciones:

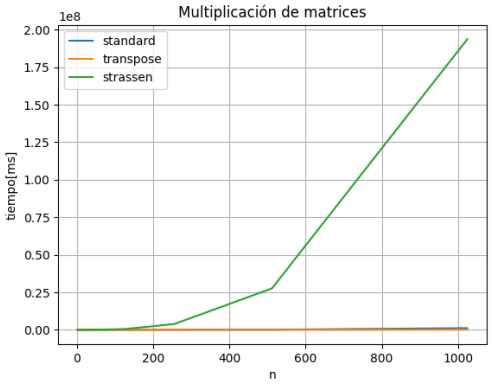
- El algoritmo “Strassen” tiene un muy bajo desempeño. De acuerdo al gráfico, al pasar del tamaño 600 la diferencia con los otros algoritmos se hace demasiado evidente.
- Se separó el gráfico de “Strassen” para comparar el desempeño de los otros dos. Se infiere que el algoritmo “Standard” se acerca a $O(n^2)$.

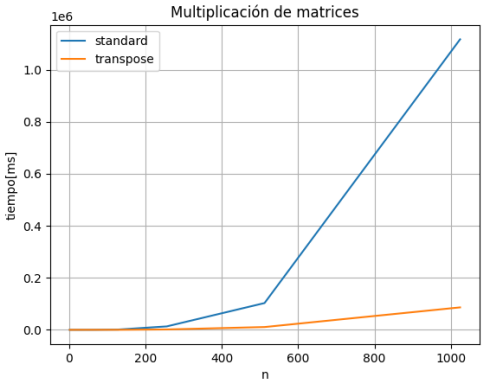
4.2.2 Caso 2: Matrices cuadradas con tamaño potencia de 2







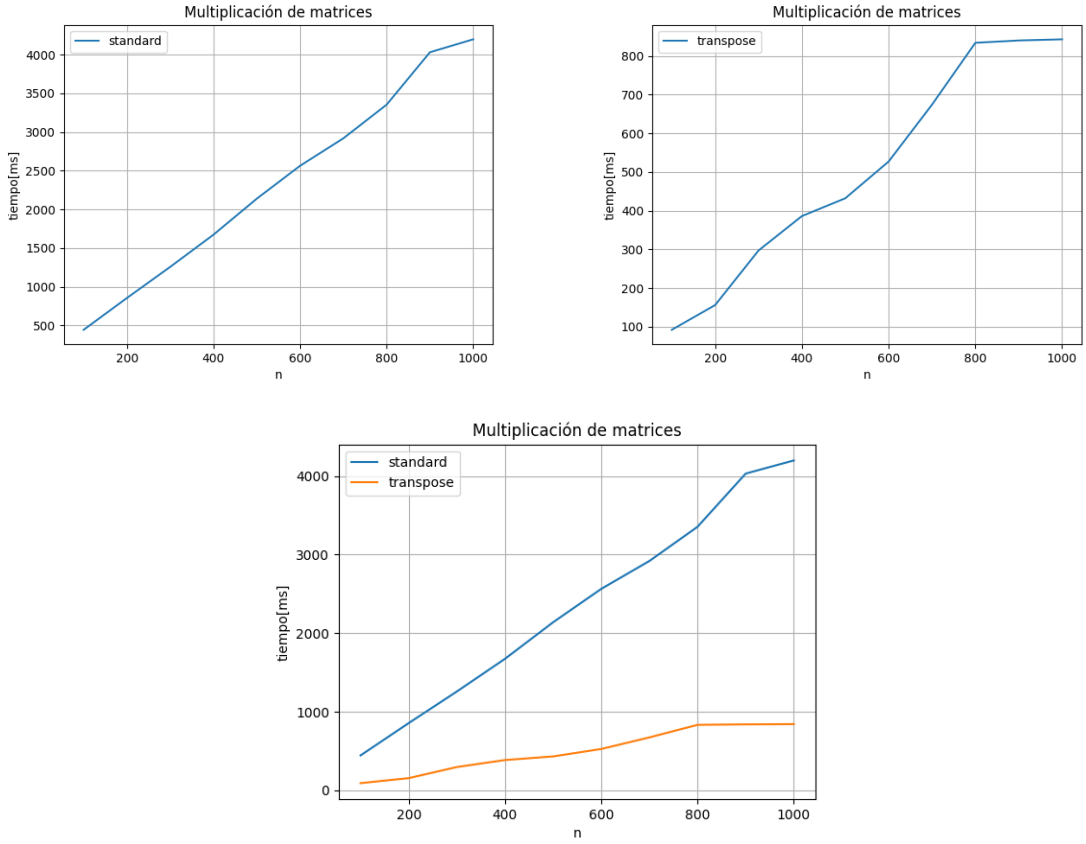




Observaciones:

- El algoritmo “Strassen”, si bien tiene bajo desempeño, en esta ocasión no se aleja tanto del rendimiento de sus otros competidores.
- Se separó el gráfico de “Strassen” para comparar el desempeño de los otros dos. A tamaños pequeños no se ve gran diferencia, pero al llegar a los 1000 la diferencia es notable.

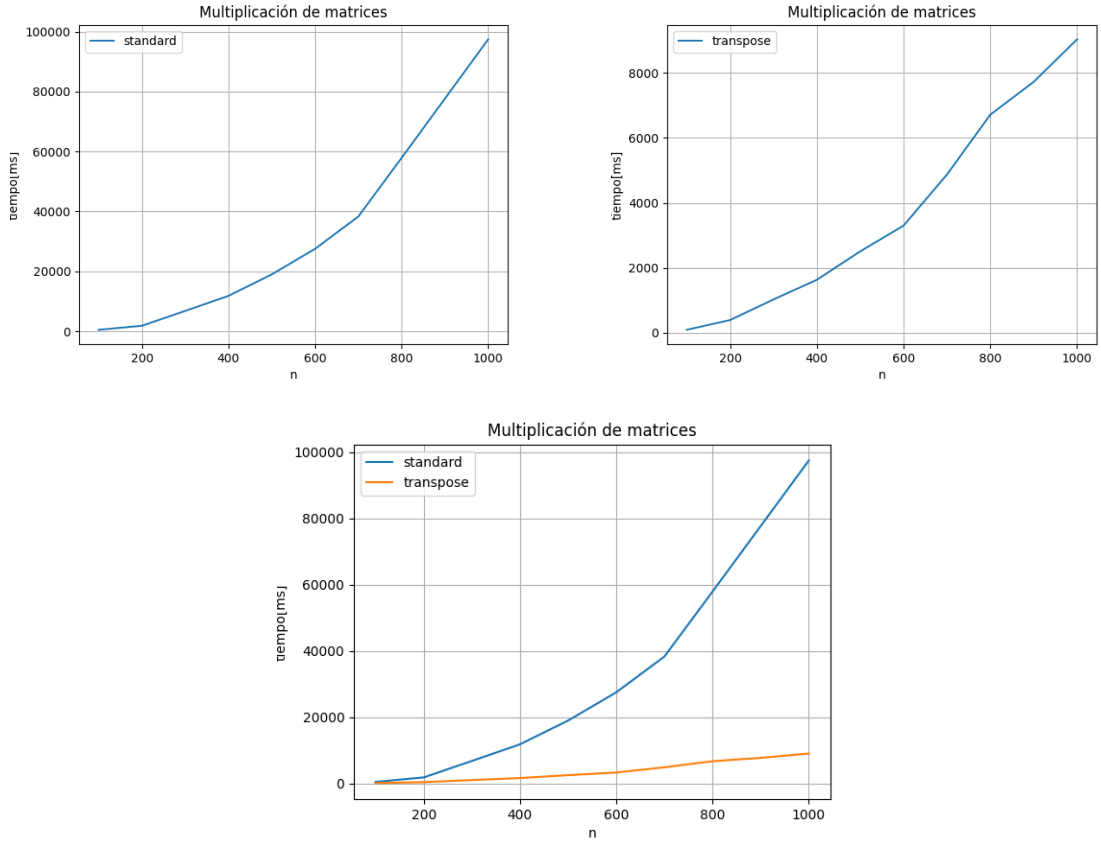
4.2.3 Caso 3: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas variables, columnas fijas)



Observaciones:

- No se dan las condiciones para realizar la ejecución del algoritmo “Strassen”, por lo que fue eliminado del análisis
- Se infiere que el algoritmo “Standard” se acerca a $O(n^2)$. En este caso, desde el primer momento “Transpose” es el claro ganador.

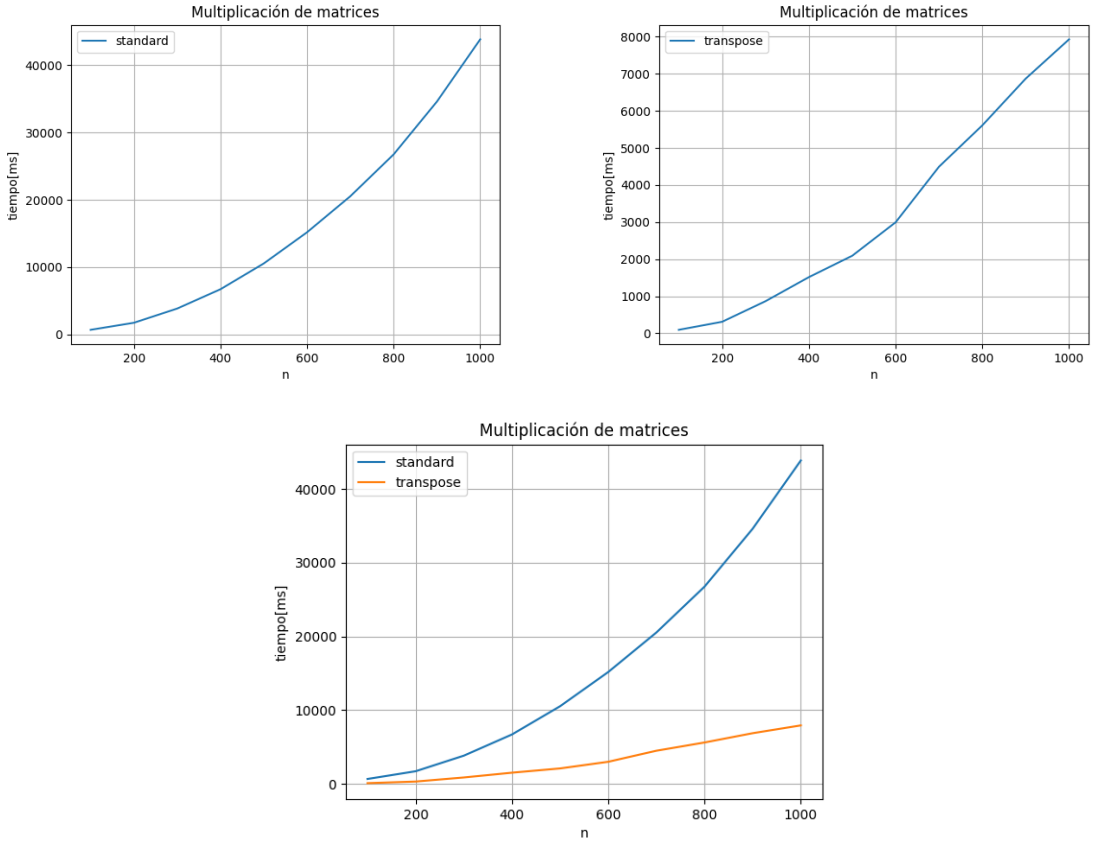
4.2.4 Caso 4: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas y columnas variables)



Observaciones:

- No se dan las condiciones para realizar la ejecución del algoritmo “Strassen”, por lo que fue eliminado del análisis
- Se infiere que el algoritmo “Standard” se acerca a $O(n^2)$. A tamaños pequeños de matrices la diferencia en rendimiento no es tan grande.

4.2.5 Caso 5: Matrices rectangulares con matriz A (filas y columnas variables) y matriz B (filas variables y columnas fijas)



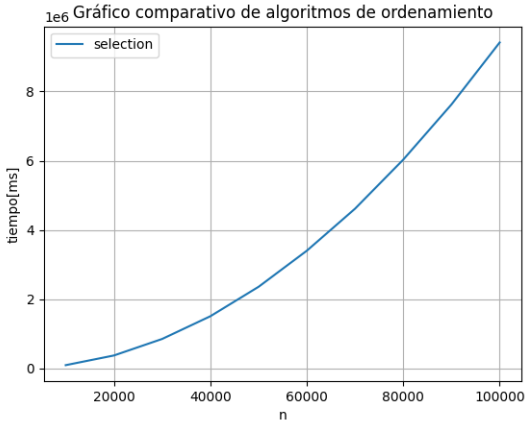
Observaciones:

- No se dan las condiciones para realizar la ejecución del algoritmo “Strassen”, por lo que fue eliminado del análisis
- Se infiere que el algoritmo “Standard” se acerca a $O(n^2)$. Existe una mínima diferencia al comenzar, que luego se hace evidente con la victoria de “Transpose” por mejor desempeño.

5. Conclusiones

5.1 Algoritmos de ordenamiento

- El algoritmo “Selection Sort” es el de peor desempeño luego de todas las pruebas. En los gráficos se ve claramente su orden cuadrático $O(n^2)$:



- El algoritmo “Sort Interno” fue el de mejor desempeño luego de todas las pruebas
- El algoritmo “Quicksort” tuvo un desempeño similar al “Sort Interno” cuando los datos venían **desordenados con datos únicos o repetidos**. Su peor desempeño lo tuvo con datos **ordenados.**
- El algoritmo “Mergesort” ocupó generalmente el segundo lugar en cuando a tiempo de procesamiento.
- El algoritmo “Sort Interno” es el de más fácil implementación en C++, ya que no requiere la programación del algoritmo y solamente precisa del comando sort.
- La diferencia de rendimiento entre cada algoritmo se hace más evidente al aumentar el tamaño de la muestra.
- Se esperaba, de acuerdo a su definición, que los algoritmos “Quicksort”, “Merge sort”y “Sort Interno” tuviesen rendimientos similares, sin embargo, eso no ocurrió.

Por lo tanto, se puede concluir que, si bien existen algoritmos con mejor o peor desempeño, debe ser analizado en primera instancia el tipo de input a procesar, su tamaño y su grado de desorden.

5.2 Multiplicación de matrices

- El algoritmo “Strassen” para matrices cuadradas no tuvo un buen desempeño comparado con el algoritmo “Standard” y “Transpuesta”, sin embargo para matrices cuadradas de tamaño potencia de dos, y considerando matrices de pequeño tamaño, sí podría considerase competitivo respecto al tiempo.
- Respecto a los otros dos algoritmos, el de mejor desempeño siempre fue el de “Transpuesta”, destacándose aún más con matrices de gran tamaño.

5.3 Construcción de aplicación

- Es factible la construcción de una herramienta en lenguaje C++, y que mediante complementos de automatización (makefile y scripts), permita el procesamiento y medición del rendimiento de diferentes algoritmos.
- Como complemento a esta herramienta es recomendable la construcción, como se hizo en el presente informe, de un “generador de datasets” que permita variar en tamaño y en tipo de desorden el input de los diferentes algoritmos.

6. Referencias

[1] Wikipedia, Selection sort. [En línea]. Disponible: https://en.wikipedia.org/wiki/Selection_sort

[2] Wikipedia, Merge sort. [En línea]. Disponible: https://en.wikipedia.org/wiki/Merge_sort

[3] Wikipedia, Quick sort. [En línea]. Disponible: https://es.wikipedia.org/wiki/Quicksort

[4] Wikipedia, Sort STL. [En línea]. Disponible: https://en.wikipedia.org/wiki/Sort_(C%2B%2B)

[5] Medium, Algorithm Analysis & Time Complexity Simplified. [En línea]. Disponible: https://randerson112358.medium.com/algorithm-analysis-time-complexity-simplified-cd39a81fec71

[6] Wikipedia, Multiplicación de matrices. [En línea]. Disponible: https://es.wikipedia.org/wiki/Multiplicación_de_matrices

[7] De User:Bilou - Trabajo propio, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1538693

[8] Github, Explicación del algoritmo de Strassen. [En línea]. Disponible: https://alu0100881677.github.io/DAA_L2_1_Strassen/Strassen.html

[9] Github, Herramienta para la generación de datasets para algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/datasets

[10] Github, Outputs de ejecución de algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/output

[11] Github, Archivos para graficar. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/csv

[12] Github, Herramienta para la generación de datasets para multiplicación de matrices. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/datasets

[13] Github, Outputs de ejecución de algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/output

[14] Github, Archivos para graficar. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/csv

[15] Github, Gráficos para algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/gr%C3%A1ficos

[16] Github, Gráficos para multiplicación de matrices. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/gr%C3%A1ficos

[17] Github, Código fuente herramienta análisis algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code

[18] Github, Código fuente herramienta análisis algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code