|  |  |  |
| --- | --- | --- |
|  |  |  |

FUNDAMENTOS DE ESTRUCTURAS DE DATOS

Y ANÁLISIS DE ALGORITMOS

**INFORME N°3**

**“Diccionarios, Tablas Hash vs Árboles de búsqueda”**

Junio 2023

**ERICH GERMÁN GRÜTTNER DÍAZ**

**Índice**

[1. Introducción 4](#_Toc133954985)

[2. Descripción de los algoritmos a ser comparados 5](#_Toc133954986)

[2.1 Ordenamiento 5](#_Toc133954987)

[2.1.1 Selection Sort 5](#_Toc133954988)

[2.1.2 Merge Sort 6](#_Toc133954989)

[2.1.3 Quick Sort 7](#_Toc133954990)

[2.1.4 Sort Interno C++ 8](#_Toc133954991)

[2.1.5 Tabla resumen algoritmos de ordenamiento a analizar en este informe 9](#_Toc133954992)

[2.2 Multiplicación de matrices 10](#_Toc133954993)

[2.2.1 Algoritmo iterativo cúbico tradicional 11](#_Toc133954994)

[2.2.2 Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos 12](#_Toc133954995)

[2.2.3 Algoritmo Strassen 13](#_Toc133954996)

[3. Descripción de los datasets 14](#_Toc133954997)

[3.1 Datasets para algoritmos de ordenamiento 14](#_Toc133954998)

[3.1.1 Inputs 14](#_Toc133954999)

[3.1.2 Outputs 16](#_Toc133955000)

[3.1.3 Archivos para graficar 16](#_Toc133955001)

[3.2 Datasets para multiplicación de matrices 17](#_Toc133955002)

[3.1.1 Inputs 17](#_Toc133955003)

[3.1.2 Outputs 19](#_Toc133955004)

[3.1.3 Archivos para graficar 19](#_Toc133955005)

[4. Resultados experimentales 20](#_Toc133955006)

[4.1 Medición rendimiento algoritmos de ordenamiento 21](#_Toc133955007)

[4.1.1 Caso 1: Datos desordenados repetidos 21](#_Toc133955008)

[4.1.2 Caso 2: Datos desordenados con valores únicos 22](#_Toc133955009)

[4.1.3 Caso 3: Datos ordenados repetidos 23](#_Toc133955010)

[4.1.4 Caso 4: Datos ordenados reversa (descendente) 24](#_Toc133955011)

[4.1.5 Caso 5: Datos parcialmente ordenados 25](#_Toc133955012)

[4.2 Medición rendimiento para multiplicación de matrices 26](#_Toc133955013)

[4.2.1 Caso 1: Matrices cuadradas 26](#_Toc133955014)

[4.2.2 Caso 2: Matrices cuadradas con tamaño potencia de 2 27](#_Toc133955015)

[4.2.3 Caso 3: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas variables, columnas fijas) 28](#_Toc133955016)

[4.2.4 Caso 4: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas y columnas variables) 29](#_Toc133955017)

[4.2.5 Caso 5: Matrices rectangulares con matriz A (filas y columnas variables) y matriz B (filas variables y columnas fijas) 30](#_Toc133955018)

[5. Conclusiones 31](#_Toc133955019)

[5.1 Algoritmos de ordenamiento 31](#_Toc133955020)

[5.2 Multiplicación de matrices 32](#_Toc133955021)

[6. Referencias 33](#_Toc133955022)

# 1. Introducción

**Se introduce de manera correctitud y coherente el tema a tratar. Se menciona el objetivo general de la tarea y una breve descripción de las conclusiones obtenidas**

**La temática y las conclusiones se introducen de manera adecuada**

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 las estructuras de datos

**Se describen, como mínimo, las estructuras de datos comparadas: Árbol binario, tabla hash con hashing abierto, tabla hash con hashing cerrado. Cada estructura debe ser descrita en un párrafo, incluyendo referencias bibliográficas de ser necesario.**

**Se presenta de manera correcta la descripción de las 3 estructuras de datos solicitadas**

## 2.1 Árbol binario

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.

## 2.2 Tabla Hash con hashing abierto

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.

## 2.3 Tabla Hash con hashing cerrado

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.

# 3. Código y documentación

**Se entrega el código fuente de las tres estructuras de datos y su documentación**

**Tanto el código como la documentación son adecuados al contexto de la tarea. Se entenderá como documentación adecuada a aquella que está contenida dentro del mismo código o en un archivo adjunto.**

# 4. Datasets y diseño experimental

**Descripción del dataset, métodos de lectura/escritura y setup experimental**

**Se describe de manera adecuada y concisa el dataset utilizado en la evaluación experimental. Además, se describe el equipo computacional utilizado. Cómo mínimo incluye el tipo de procesador, tamaño de memoria RAM y tamaño de memorias caché.**

**Input 1.csv -> 1000 registros**

**Input 2.csv -> 5000 registros**

**Input 3.csv -> 10.000 registros**

**Input 4.csv -> 15.000 registros**

**Input 5.csv -> 29245 registros**

**university;user\_id;user\_name;number\_tweets;friends\_count;followers\_count;created\_at**

**uvalpochile;1183020785033760773;dvm\_ucsc;462;712;578;"Sat Oct 12 14:05:55 +0000 2019"**

input 1, 85KB

input 2, 418KB

input 3, 816KB

input 4, 1,2MB

input 5, 2,4MB

## 4.1 D|atasets 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](#nueve)]

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

### 4.1.1 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 |

### 4.1.2 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](#diez)]

### 4.1.3 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](#once)]

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

# 5. Resultados experimentales

**Incluye una sección describiendo los experimentos realizados, resumiendo de manera adecuada los resultados obtenidos por medio de tablas y figuras.**

**Los resultados entregados están bien explicados y resumidos. Se entregan resultados de tiempo de ejecución y espacio de las estructuras. Además, se utilizan los dos tipos de claves en la comparación: User ID y User Name.**

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](#diecisiete)] y para multiplicación de matrices en [[18](#dieciocho)].

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](#quince)] y los gráficos asociados a la multiplicación de matrices se encuentran en [[16](#dieciseis)].

## 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(n2)
* 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(n2)
* 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(n2)
* 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(n2)
* 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(n2)
* 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(n2).

### 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(n2). 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(n2). 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(n2). Existe una mínima diferencia al comenzar, que luego se hace evidente con la victoria de “Transpose” por mejor desempeño.

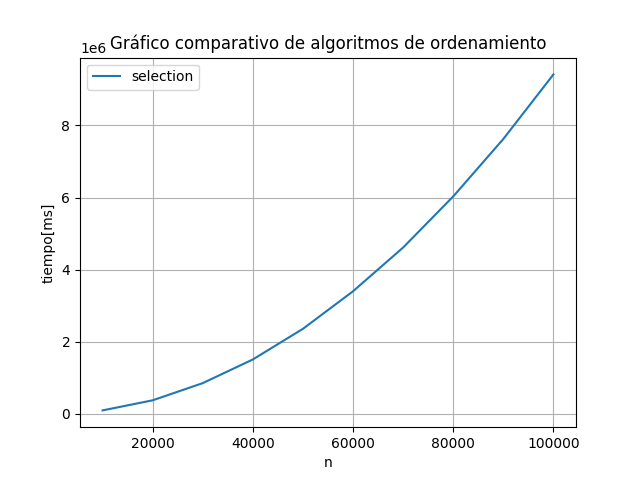
# 6. Conclusiones

Incluye una secciones con conclusiones y reflexiones generadas a partir de desarrollo del trabajo.

Las conclusiones son correctas y están respaldadas por los resultados mostrados en la sección de experimentos.

## 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(n2):



* 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.

# 7. 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