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

ESTRUCTURAS DE DATOS Y ALGORITMOS AVANZADOS

**BOLETÍN N°1**

SEPTIEMBRE 2023

**ERICH GERMÁN GRÜTTNER DÍAZ**

**Índice**

[1. Introducción 3](#_Toc144557501)

[2. Descripción de los algoritmos a ser comparados 4](#_Toc144557502)

[2.1 Ordenamiento 4](#_Toc144557503)

[2.1.1 Selection Sort 4](#_Toc144557504)

[2.1.2 Merge Sort 5](#_Toc144557505)

[2.1.3 Quick Sort 6](#_Toc144557506)

[2.1.4 Sort Interno C++ 7](#_Toc144557507)

[2.1.5 Tabla resumen algoritmos de ordenamiento a analizar en este informe 8](#_Toc144557508)

[3. Descripción de los datasets 9](#_Toc144557509)

[3.1 Datasets para algoritmos de ordenamiento 9](#_Toc144557510)

[3.1.1 Inputs 9](#_Toc144557511)

[3.1.2 Outputs 11](#_Toc144557512)

[3.1.3 Archivos para graficar 11](#_Toc144557513)

[4. Resultados experimentales 12](#_Toc144557514)

[4.1 Medición rendimiento algoritmos de ordenamiento 13](#_Toc144557515)

[4.1.1 Caso 1: Datos desordenados repetidos 13](#_Toc144557516)

[4.1.2 Caso 2: Datos desordenados con valores únicos 14](#_Toc144557517)

[4.1.3 Caso 3: Datos ordenados repetidos 15](#_Toc144557518)

[4.1.4 Caso 4: Datos ordenados reversa (descendente) 16](#_Toc144557519)

[4.1.5 Caso 5: Datos parcialmente ordenados 17](#_Toc144557520)

[5. Conclusiones 18](#_Toc144557521)

[5.1 Algoritmos de ordenamiento 18](#_Toc144557522)

[6. Referencias 19](#_Toc144557523)

# 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 Búsqueda simple

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 Búsqueda binaria

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](#uno)].

Tiene una complejidad de tiempo de O(n2), 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(n2), 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 Búsqueda galopante

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](#dos)]

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.

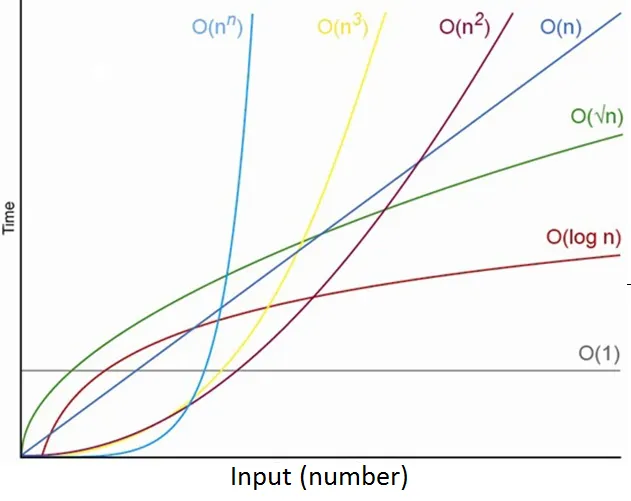
Pseudocódigo:

|  |
| --- |
| 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 |

### 2.1.5 Tabla resumen algoritmos de ordenamiento a analizar en este informe

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nombre** | **Mejor caso** | **Promedio** | **Peor caso** | **Estable** |
| Selection sort | n2 | n2 | n2 | No |
| Merge sort | n log n | n log n | n log n | Sí |
| Quicksort | n log n | n log n | n2 | No |
| SortInterno (Introsort) | n log n | n log n | n log n | No |

Crecimiento de los algoritmos según su orden de complejidad [[5](#cinco)]

****

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

# 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](#nueve)]

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](#diez)]

### 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](#once)]

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](#diecisiete)]

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

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

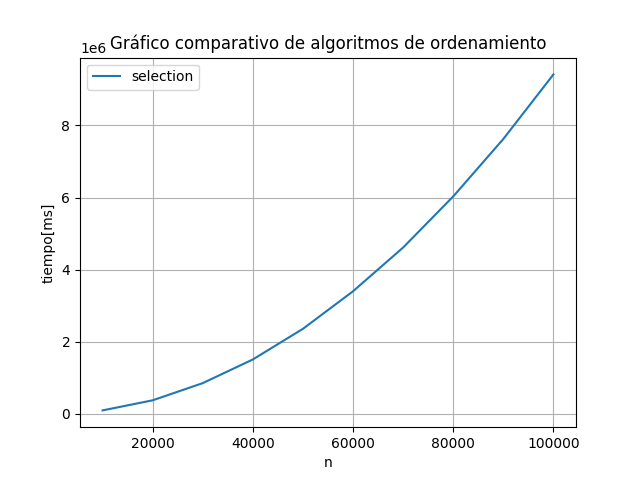
Observaciones:

* 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.

# 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(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.

# 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