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

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**

**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 y las matemáticas. 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. En este informe se discutirán los algoritmos más utilizados para cada tarea, sus ventajas y desventajas, y se presentarán algunos ejemplos para ilustrar su aplicación práctica.

Breve reseña de todo el trabajo realizado. Descripción a alto nivel de los problemas y algoritmos (implementados y provistos por bibliotecas), herramientas, fuentes de datos y conclusiones preliminares

Este primer informe persigue que los estudiantes realicen una evaluación experimental de distintos algoritmos de ordenamiento y de multiplicación de matrices. Así, se busca que los estudiantes sean capaces de implementar algunos de los algoritmos más conocidos para este tipo de problemas, además de contrastar su funcionamiento con implementaciones provistas por los lenguajes de programación. Adicionalmente, esta primera tarea permitirá a los estudiantes familiarizarse con herramientas de profiling y con la creación de códigos que les permitan medir las características solicitadas.

**2.- Descripción de los algoritmos a ser comparados**

**2.1 Ordenamiento**

**\*Intro algoritmos de ordenamiento**

se consigna adecuadamente sus costos de cómputo.

1. Entrega de todos los códigos como adjuntos al documento o enlaces a dónde encontrarlos en el mismo. El código debe estar documentado y seguir algún estándar de codificación adoptado por el estudiante.  
   Para cada algoritmo, especificar una descripción general y consignar el costo del mejor y peor caso.  
   Reseñar las funciones de bibliotecas estándar utilizadas ¿Qué algoritmos funcionan por debajo y cuáles son sus costos?

**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.

Tiene una complejidad de tiempo de O(n2), lo que lo hace ineficiente para listas largas, y generalmente tiene peor desempeño que el “InsertionSort”.

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 QuickSort o MergeSort, 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.

|  |  |
| --- | --- |
| **Clase** | **Algorimo de ordenamiento** |
| **Estructura de datos** | **Arreglo** |
| **Desempeño peor caso** | **O(n2) comparaciones,**  **O(n) intercambios** |
| **Desempeño mejor caso** | **O(n2) comparaciones,**  **O(1) intercambios** |
| **Desempeño promedio** | **O(n2) comparaciones,**  **O(n) intercambios** |

**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.

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 QuickSort**

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.

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.

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 log2n, 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());

**for** (size\_t i = 0; i < vec.size(); ++i) {

cout << vec[i] << ' ';

}

cout << '\n';

}

**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.

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 |

\*PONER DIAGRAMA

**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 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.

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**

**3.1.1 Inputs**

Para la construcción de datasets se consideraron las siguientes variables:

* Cantidad de archivos a generar
* Cantidad de elementos en cada vector
* Rango de números del vector
* Grado de orden/desorden:
  + Datos repetidos desordenados
  + Datos únicos desordenados
  + Datos repetidos ordenados ascendente
  + Datos repetidos ordenados descendente
  + Datos parcialmente ordenados (la mitad de los datos está desordenado, explicar)
  + Datos semi-ordenados (solo algunos elementos están desordenados, explicar)

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

|  |
| --- |
| inputxx.txt |
| **Tamaño vector** |
| **Dato1** |
| **…** |
| **Dato n** |

Se generaron carpetas para almacenar cada generación, con el siguiente formato: tipo de “desorden” + cantidad de archivos + número máximo permitido para pertenecer al vector.

Ejemplo: “desordenado\_repetido-10-10000”

Se construyó un generador para estos inputs, ubicado en el repositorio de código en XXX.

Para el presente informe se consideraron los siguientes parámetros:

* 10 archivos por generación
* El primer archivo contiene 10.000 registros, el segundo 20.000 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
* Se generó una muestra por cada tipo de “desorden”
* Poner como se construyó cada tipo de desorden
* Gráfico de cada set

**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)** |

**4.- Resultados experimentales**

Para la realización de pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria.

Forma de realizar las mediciones:

Se utiliza un parámetro de cantidad de ciclos de ejecución junto a un valor de incremento en las mediciones. Por ejemplo, un rango de 1 a 1000 ejecuciones, con incrementos de 100 en 100.

Por otra parte, cada una de eses ejecuciones comprende la realización de N experimentos (valor parametrizable), que permite calcular un promedio de tiempo de ejecución por bloque de experimentos.

Parámetros a considerar:

Cantidad de ejecuciones

Rango de incremento en el intervalo de ejecuciones

Cantidad de experimentos a realizar dentro del ciclo de ejecuciones

1.- Primer experimento:

* Probando variabilidad de número de experimentos
* Ejecución: rango 1 a 1000, cada 100
* 10 archivos desordenados con posibilidad de números repetidos

1° archivo 10.000 registros, 2° achivos 20.000…hasta llegar al décimo y último archivos con 100.000 registros

Poner tablas de inputs

Poner gráficos de ejecución

2.- Variabilidad de rango de ejecución

Caso1

//Parámetros de ejecución

n\_1 = 0;

N = 500; //1000

i = 10;//100 incremento

numero\_de\_experimentos = 10; //10

Desordenados repetidos

Caso2

Muestra: 10 archivos, totalmente desordenados, repetidos, 1 a 5000, cada 100

1. En esta sección se deberán mostrar los resultados comparativos de los distintos algoritmos en forma de tablas, gráficos y comentar, dentro del texto, observaciones que consideren que vale la pena resaltar en las tablas/gráficos.

**Esperados**

**Obtenidos**

**Gráficos**

**Ejecuciones:**

**Ordena**

**5.- Conclusiones**

1. **:** En esta sección se deberán consignar observaciones/comentarios/sentencias que evalúen el trabajo realizado, así como los resultados obtenidos. En especial, se aconseja abundar alrededor del poder predictivo del análisis asintótico del peor/mejor caso y sobre la importancia de aspectos relacionados con la implementación (i.e. algoritmos inplace vs no-inplace, preservación de la localidad de los datos, otros) en los resultados experimentales.

Ordenamiento:

Premisas:

* Expectativas de rendimiento de algoritmos
* Variedad de datasets, resultados diferentes

Mutiplicación de matrices

//Parámetros de ejecución

n\_1 = 1;

N = 10000; //1000

i = 100;//100 incremento

numero\_de\_experimentos = 10; //10

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Criterios Calificaciones Pts** | | | | | | | | | | | | | | | | | | | | |
| Algoritmos implementados/probados  Número de algoritmos tomados en cuenta en el trabajo | **3.5 pts Todos los algoritmos** 7 o más algoritmos tomados en cuenta | **3 pts 6 algoritmos** 6 algoritmos | **2.5 pts 5 algoritmos**  2 algoritmos no implementados | | | | **2 pts 4 algoritmos**  3 algoritmos no implementados | | **1.5 pts 3 algoritmos implementados** 4 algoritmos no implementados | | | **1 pts 2 algoritmos implementados** 5 algoritmos no implementados | | | **0.5 pts 1 algoritmo implementado** 6 algoritmos no implementados | | | | **0 pts Sin marcas** | 3.5 pts |
| Introducción  Correctitud y pertinencia de la introducción | **0.5 pts Introducción adecuada**  La introducción cumple su propósito y motiva adecuadamente el trabajo | | | | **0.3 pts Introducción con errores**  La introducción presenta errores o no motiva adecuadamente el trabajo | | | | | | | | **0 pts Sin introducción**  El trabajo no presenta introducción o ésta tiene serios errores conceptuales. | | | | | | | 0.5 pts |
| Descripción de algoritmos  Se describen los algoritmos y se sigue la especificación del enunciado de la tarea. | **1 pts Descripción completa y correcta** Se describen todos los algoritmos y se consigna adecuadamente sus costos de cómputo. | | **0.7 pts Descripción adecuada con errores u omisiones menores**  Se describen mas de 5 algoritmos y se consigna adecuadamente su complejidad | | | | | **0.4 pts Descripción deficiente**  Se describen menos de 4 algoritmos y/o existen varios errores en la complejidad de los mismos. | | | | | **0.1 pts Grabes errores en la descripción de los algoritmos**  O bien no se describen los algoritmos o existen errores serios en los costos reportados. | | | | **0 pts Sin descripción** El trabajo no describe los algoritmos solicitados. | | | 1 pts |
| Datasets Descripción y  características de los datasets | **1 pts Los datasets generados son diversos y adecuados** Se presentan datasets diversos y se justifica adecuadamente su inclusión en el trabajo | | | | | | **0.5 pts Datasets sencillos**  Los datasets utilizados son ingenuos y no permiten extraer conclusiones adecuadas del trabajo experimental. | | | | | | | | | **0 pts Sin descripción de los datasets** No se describen los datasets usados en el trabajo | | | | 1 pts |
| Resultados Experimentales  Se presenta una sección donde se muestran los resultados de ejecutar los distintos algoritmos a los datasets propuestos. | **2 pts Con marcas**  La sección presenta los resultados de ejecutar todos los algoritmos sobre todos los datasets presentados y utiliza un formato amigable, legible y claro. Los comentarios sobre los resultados son correctos. | | | **1.5 pts Sección con pequeños errores u omisiones** La sección presenta los resultados de ejecutar todos los algoritmos sobre todos los datasets presentados y o bien tiene problemas de presentación o contiene errores menores. | | | | | | **1 pts Resultados parcialmente logrados**  La sección presenta carencias considerables, su presentación es confusa o contiene errores considerables. | | | | **0.5 pts Resultados incompletos** La sección tiene serias deficiencias, presenta muy pocos resultados o es ininteligible o tiene errores muy importantes. | | | | **0 pts No existen resultados** No se presenta la sección. | | 2 pts |
| Conclusiones La sección de  conclusiones está correctamente presentada. Lo que allí se dice tiene relación directa y es argumentable a partir de lo presentado en las anteriores secciones. | **1 pts Conclusiones adecuadas**  La sección de conclusiones está correctamente presentada. Lo que allí se dice tiene relación directa y es argumentable a partir de lo presentado en las anteriores secciones. | | | | | **0.7 pts Conclusiones mayoritariamente adecuadas**  Las conclusiones son en general adecuadas pero se cometen omisiones o afirmaciones importantes para el trabajo. | | | | | **0.3 pts Conclusiones insuficientes** Las conclusiones no tienen relación o contradicen el contenido presentado con anterioridad o son muy limitadas. | | | | | **0 pts Sin conclusiones**  El trabajo no presenta conclusiones o éstas no tienen relación alguna con el contenido del mismo. | | | | 1 pts |
| Puntos totales: 9 | | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | | |

**REFERENCIAS**

**Selection Sort:**

[**https://en.wikipedia.org/wiki/Selection\_sort**](https://en.wikipedia.org/wiki/Selection_sort)

[**https://en.wikipedia.org/wiki/Sorting\_algorithm**](https://en.wikipedia.org/wiki/Sorting_algorithm)

<https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja>

**https://es.wikipedia.org/wiki/Algoritmo\_de\_ordenamiento**

**MergeSort**

**https://en.wikipedia.org/wiki/Merge\_sort**

https://es.wikipedia.org/wiki/Ordenamiento\_por\_mezcla

**QuickSort**

<https://es.wikipedia.org/wiki/Quicksort>

**Sort STL**

**https://en.wikipedia.org/wiki/Sort\_(C%2B%2B)**

**Matrices Algoritmo estándar**

[**https://es.wikipedia.org/wiki/Multiplicación\_de\_matrices**](https://es.wikipedia.org/wiki/Multiplicación_de_matrices)

[**https://en.wikipedia.org/wiki/Matrix\_multiplication**](https://en.wikipedia.org/wiki/Matrix_multiplication)

**Strassen:**

https://alu0100881677.github.io/DAA\_L2\_1\_Strassen/Strassen.html

[**https://es.wikipedia.org/wiki/Algoritmo\_de\_Strassen**](https://es.wikipedia.org/wiki/Algoritmo_de_Strassen)