Multiplicación secuencial de matrices

1.- Para este punto de la práctica se decidió implementar una clase lectora de matrices y una clase con la información de una matriz, la cual lleva el nombre de ManejoArchivoMatriz y DatosMatriz, la clase ManejoArchivoMatriz contiene dos métodos que llevan el nombre de LecturaArchivo y DatosArchivos, el primer método hace la lectura de cualquier archivo que se encuentre en la carpeta Matrices y nos retorna un objeto del tipo DatosMatriz, el método DatosArchivos nos proporciona las dimensiones de las matrices que están en los archivos que llevan el sufijo de "Matriz", esto con la finalidad de que el usuario pueda seleccionar las matrices que el quiera y que respete la condición de que al querer multiplicar dos matrices de cualquier tamaño el numero de columnas de la primera debe coincidir con el numero de filas de la segunda.

La clase DatosMatriz nos proporciona la facultad de generar un objeto que contiene la matriz, el numero de columnas y el número de filas, esto se hizo para un control más cómodo de las matrices.

El código 2.2.2.1 nos indica cual es la secuencia optima de multiplicación entre matrices, ahora comprobemos el funcionamiento de este código mediante un ejemplo:

Ejemplo

Supongamos que queremos multiplicar las siguientes 4 matrices: $A_0, A_1, A_2\ y\ A_3$ las cuales le corresponden las siguientes dimensiones: 5x4, 4x6, 6x2 y 2x7 respectivamente, entonces hagamos los cálculos, primeramente, tenemos que buscar la multiplicación de una sola matriz, , en donde el coste es evidentemente 0, teniendo así el siguiente resultado:

Para M:

M	0	1	2	3
0	0			
1		0		
2			0	
3				0

Para S:

S	0	1	2	3
0	-1			
1		-1		
2			-1	
3				-1

Ahora para el siguiente paso calculemos el coste de multiplicar dos matrices, es decir, vamos a multiplicar $A_0\ con\ A_1, A_1\ con\ A_2\ y\ A_2\ con\ A_3$, el coste para la primera multiplicación es 5x4x6 que es igual a 120, el coste para la segunda multiplicación

es 4x6x2 que es igual a 48 y el coste para la última multiplicación es 6x2x7 que es igual a 84, teniendo así el siguiente resultado:

Para M:

M	0	1	2	3
0	0	120		
1		0	48	
2			0	84
3				0

Para S:

S	0	1	2	3
0	-1	0		
1		-1	1	
2			-1	2
3				-1

Ahora nos corresponde calcular el coste de multiplicar tres matrices, es decir, vamos a multiplicar $A_0 con\ A_1 con\ A_2, y\ A_1 con\ A_2 con\ A_3$, sin embargo, nos encontramos de que hay dos formas de realizar la multiplicación las cuales son tomando las primeras dos matrices o las ultimas dos, lo cual nos daría dos resultados por lo que tendríamos lo siguiente: $\min\{A_0 * (A_1 * A_2), (A_0 * A_1) * A_2\}$, lo cual es igual a $\min\{0 + 48 + 5 * 4 * 2, 120 + 0 + 5 * 6 * 2\} = \min\{88,180\}$, por lo que el mínimo seria 88 y la forma óptima de multiplicar seria $A_0 * (A_1 * A_2)$ y así para la siguiente multiplicación tendríamos lo siguiente: $\min\{A_1 * (A_2 * A_3), (A_1 * A_2) * A_3\}$, lo cual es igual a $\min\{0 + 84 + 4 * 6 * 7, 48 + 0 + 4 * 2 * 7\} = \min\{252,104\}$, por lo que el mínimo seria 104 y la forma óptima de multiplicar seria $(A_1 * A_2) * A_3$, teniendo así el siguiente resultado:

Para M:

M	0	1	2	3
0	0	120	88	
1		0	48	104
2			0	84
3				0

Para S:

S	0	1	2	3
0	-1	0	0	
1		-1	1	2
2			-1	2
3				-1

Finalmente, el último paso que es multiplicar las 4 matrices, teniendo 3 formas de realizar dicha multiplicación las cuales son las siguientes: $\min\{A_0 * (A_1 * A_2 * A_3), (A_0 * A_1) * (A_2 * A_3), (A_0 * A_1 * A_2) * A_3\}$, lo que es igual a $\min\{0 + 104 + 5 * 4 *$

$7{,}120 + 84 + 5 * 6 * 7{,}88 + 0 + 5 * 2 * 7\} = \min\{244{,}414{,}158\}$, por lo que el mínimo seria 158 y la forma óptima de multiplicar seria $(A_0 * A_1 * A_2) * A_3$, teniendo así el siguiente resultado:

Para M:

M	0	1	2	3
0	0	120	88	158
1		0	48	104
2			0	84
3				0

Para S:

S	0	1	2	3
0	-1	0	0	2
1		-1	1	2
2			-1	2
3				-1

Así vemos que para multiplicar esas cuatro matrices la forma óptima es $(A_0 * A_1 * A_2) * A_3$, pero la forma óptima de esta suma $A_0 * A_1 * A_2$ es esta: $A_0 * (A_1 * A_2)$, con lo que la forma optima final es: $(A_0 * (A_1 * A_2)) * A_3$, que como vemos en la imagen 1 es el mismo resultado. Las celdas que no cuentan con un numero pueden ser llenadas con cualquier otro valor, en el caso del programa se llenan con -1.

```
Las dimensiones del archivo Matriz1 son: 7x7
Las dimensiones del archivo Matriz2 son: 7x2
Las dimensiones del archivo Matriz3 son: 2x2
Las dimensiones del archivo Matriz4 son: 2x6
Las dimensiones del archivo Matriz5 son: 5x6
Las dimensiones del archivo Matriz6 son: 5x4
Las dimensiones del archivo Matriz7 son: 4x6
Las dimensiones del archivo Matriz8 son: 6x2
Las dimensiones del archivo Matriz9 son: 2x7

Cuantas matrices quiere usar? : 4
Introduzca el numero de los archivos a usar (Disponibles del 1-9):
Indice del archivo 1:
6
Indice del archivo 2:
7
Indice del archivo 3:
8
Indice del archivo 4:
9

M =
0 120 88 158
-1 0 48 104
-1 -1 0 84
-1 -1 -1 0
S =
-1 0 0 2
-1 -1 1 2
-1 -1 -1 2
-1 -1 -1 -1
La secuencia optima de multiplicacion es:
((A[0] (A[1] A[2])) A[3])
```

Imagen 1. Secuencia optima de multiplicación para el ejemplo

De esta manera comprobamos que el programa 2.2.2.1 nos proporciona la forma óptima de multiplicar las matrices, evidentemente se hicieron más pruebas, pero considero suficiente el mostrar un ejemplo con el proceso de cómo se haría *a mano* este problema.

Sanchez Mendez Edmundo Josué
Análisis de Algoritmos

2.- En este punto nos damos cuenta de que el código 2.2.2.2 que es la multiplicación matricial normal está diseñado únicamente para multiplicación para dos matrices, con lo cual se adaptó para realizar la multiplicación de n matrices, sin embargo, en el proceso, me arrojaba un error a la hora de multiplicar las matrices con lo que decidí modificar el método multiplicar de la clase AritmeticaMatricial.

Código anterior al cambio:

```
public Matriz multiplicar( Matriz M1, Matriz M2 ) {
        int [] xy1 = M1.length();
        int [] xy2 = M2.length();
        if( xy1[1]!=xy2[0] ) {
                System.out.println( "No se puede realizar la multiplicacion de " );
                System.out.println( M1 + " y " + M2 );
                return null;
        }
        matriz = new Matriz(xy1[0], xy2[1]);
        for( int y=0; y<xy1[0]; y++ ) {
                for( int x=0; x<xy2[1]; x++ ) {
                        int valor = 0;
                        for( int k=0; k<xy1[1]; k++ ) {
                                valor += (M1.getElemento(k, x) * M2.getElemento(y, k));
                                //System.out.println( M1.getElemento(k, x) + " " +
                                // M2.getElemento(y, k) + " " + valor );
                        }
                        //System.out.println( valor );
                        matriz.addElemento(x, y, valor);
                }
        }
return matriz;
}
```

Código posterior al cambio:

```
public Matriz multiplicar(Matriz M1, Matriz M2) {
        int[] xy1 = M1.length();
        int[] xy2 = M2.length();
        if (xy1[1] != xy2[0]) {
            System.out.println("No se puede realizar la multiplicacion de ");
            System.out.println(M1 + " y " + M2);
            return null;
        }
        matriz = new Matriz(xy1[0], xy2[1]);
        for (int y = 0; y < xy1[0]; y++) {
            for (int x = 0; x < xy2[1]; x++) {
                int valor = 0;
                for (int k = 0; k < xy1[1]; k++) {
                    valor += (M1.getElemento(k, y) * M2.getElemento(x,k));
                }
                matriz.addElemento(y, x, valor);
            }
        }
        return matriz;
}
```

De esta forma ya no saltaba ningún error al querer realizar las multiplicaciones

En la clase ProbadorDeMatriz la cual corresponde a la multiplicación matricial normal se decidió crear dos métodos los cuales nos permite realizar una multiplicación normal, es decir, de la forma como usualmente se haría que es multiplicar siguiendo solo el orden con las que tenemos las matrices y el segundo método aplica la multiplicación con orden óptimo.

Para poder visualizar de manera clara se decidió poner temporizador de tiempo cuando se hace las multiplicaciones y el resultado es lo que se espera, el método

que aplica la multiplicación con orden optimo es más rápido que el método que no la aplica, un ejemplo de ello lo podemos ver en la imagen 2.

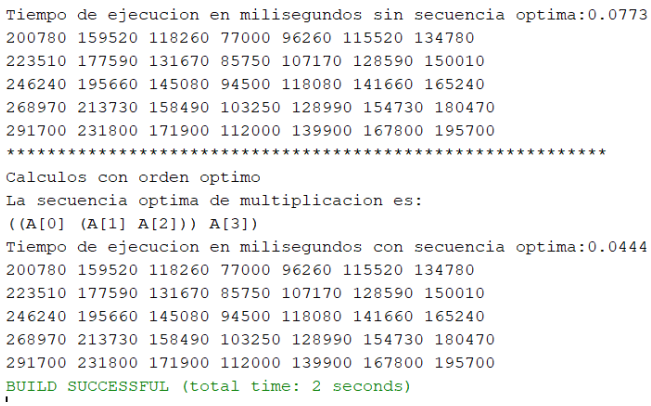

Imagen 2. Comparación del método multiplicación matricial normal usando orden optimo y sin usar el orden óptimo.

Como vemos hay una ligera diferencia y esto debido a que no es el único proceso que esta haciendo el CPU de mi computadora, pero si hubiera un CPU dedicado solo a hacer ese proceso la diferencia seria mas notoria.

Ahora vemos el código 2.2.2.3 que es la multiplicación matricial por el algoritmo de Strassen, antes de nada al investigar sobre el algoritmo de Strassen se encontró que este algoritmo está diseñado para multiplicar matrices cuadradas y de potencia de dos, es decir, matrices que comparten el mismo número de filas y de columnas y ademas ese número de filas y columnas corresponden a una potencia de 2, sin embargo, es posible adaptar este algoritmo para cualquier cantidad de matrices que queramos multiplicar, lo que hay que hacer es llenar una matriz cuadrada de $N = 2^n$, donde N es el número de filas y columnas. Nota: Esa N resulta ser la N más grande para todas las matrices a multiplicar. Ejemplo: Supongamos querer multiplicar 3 matrices de orden : 2x4, 4x6 y 6x8, la N que obtendríamos seria N=8 ya que es una potencia de 2 y es mayor o igual al numero de fila o columna mas grande entre las matrices a multiplicar

Ahora lo que resta hacer es, tomar una a una las matrices y llenar una nueva con tamaño NxN, en donde los datos de la matriz original conservarán el mismo lugar y los datos faltantes serán igual a 0. Para poder cumplir con lo anterior se creó un método llamado CumplirCondiconesdeArreglo en la clase Strassen, el cual hace que todas las matrices que queramos multiplicar cumplan con las condiciones anteriormente mencionadas. Es importante mencionar que tanto las matrices como la matriz resultante de las multiplicacion tendrán *n* columnas y filas de ceros en los extremos de la matriz que resulte de la operación.

Una vez aclarado las condiciones del algoritmo de Strassen, al igual que la Multiplicacion Matricial Normal se hicieron dos métodos uno usando la secuencia optima de multiplicacion y otro en donde no, los resultados son los esperados.

Veamos la Imagen 3, que es en donde no se ocupa la secuencia optima de multiplicacion. Como vemos resulta ser que el algoritmo de orden n^3 es más rápido que el de orden n^2.8, esto es perfectamente posible ya que no hay gran diferencia entre las ordenes que tienen estos algoritmos, sin embargo veamos la Imagen 4 que es donde ya se ocupa la secuencia optima de multiplicacion, como vemos la secuencia optima difiere con la Imagen 2 y eso es debido a la corrección de las matrices que se hace para que tengan el mismo numero de filas y columnas y que sea potencia de 2. Ahora comparando los resultados de la Imagen 3 son diferentes al de la Imagen 4, nos damos cuenta de que hay una diferencia entre estos dos algoritmos, evidentemente si usamos el orden optimo para multiplicacion es mucho mejor y de menor coste computacional algo importante actualmente.

```
Calculos sin orden optimo
Tiempo de ejecucion en milisegundos sin secuencia optima algoritmo de multiplicacion de matrices, orden n^3:0.0985
 200780 159520 118260 77000 96260 115520 134780 0
 223510 177590 131670 85750 107170 128590 150010 0
 246240 195660 145080 94500 118080 141660 165240 0
 268970 213730 158490 103250 128990 154730 180470 0
 291700 231800 171900 112000 139900 167800 195700 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
************************************************************
Pasando las matrices a matrices cuadradas
Tiempo de ejecucion en milisegundos sin secuencia optima algoritmo de Strassen, orden n^2.8:0.2582
 200780 159520 118260 77000 96260 115520 134780 0
 223510 177590 131670 85750 107170 128590 150010 0
 246240 195660 145080 94500 118080 141660 165240 0
 268970 213730 158490 103250 128990 154730 180470 0
 291700 231800 171900 112000 139900 167800 195700 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
************************************************************
```

Imagen 3. Comparación del método multiplicación matricial normal y algoritmo de Stressen sin usar el orden óptimo.

```
La secuencia optima de multiplicacion es:
(A[0] (A[1] (A[2] A[3])))
Tiempo de ejecucion en milisegundos con secuencia optima algoritmo de multiplicacion de matrices, orden n^3:2.072175073948E8
 200780 159520 118260 77000 96260 115520 134780 0
 223510 177590 131670 85750 107170 128590 150010 0
 246240 195660 145080 94500 118080 141660 165240 0
 268970 213730 158490 103250 128990 154730 180470 0
 291700 231800 171900 112000 139900 167800 195700 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
************************************************************
Pasando las matrices a matrices cuadradas
Tiempo de ejecucion en milisegundos con secuencia optima algoritmo de Strassen, orden n^2.8:0.2021
 200780 159520 118260 77000 96260 115520 134780 0
 223510 177590 131670 85750 107170 128590 150010 0
 246240 195660 145080 94500 118080 141660 165240 0
 268970 213730 158490 103250 128990 154730 180470 0
 291700 231800 171900 112000 139900 167800 195700 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
```

Imagen 4. Comparación del método multiplicación matricial normal y algoritmo de Stressen usando el orden óptimo.

En conclusión para este punto nos damos cuenta del correcto funcionamiento del código 2.2.2.1 y de la teoría en sí, ya que el orden optimo que nos proporciona cambia los tiempos en lo que se tarde en hacer esa multiplicacion, en donde, ese cambio es a favor de nosotros ya que estamos en la búsqueda de algoritmos que su coste computacional sea el menor posible, claramente hay una diferencia entre la multiplicación matricial normal y el algoritmo de Strassen ya que las ordenes son diferentes ya que tienen orden n^3 y n^2.8 respectivamente, como vemos el algoritmo de Strassen es más rápido que la multiplicacion matricial normal. Ahora

nos preguntamos el ¿Por qué?, bueno esa pregunta será contestada en el siguiente punto explicando la teoría correspondiente.

3.- Ahora justifiquemos el algoritmo de Strassen (código 2.2.2.3) con la teoría correspondiente:

Antes que nada, la multiplicación de matrices "a mano" se llevó siempre a cabo multiplicando y sumando los elementos de cada fila y cada columna, obteniendo el valor de aquel elemento que perteneciera a la fila y a la columna en cuestión, este algoritmo es el 2.2.2.2. Hasta la llegada de las ciencias de la computación, y la búsqueda por encontrar el algoritmo más rápido posible, es decir, un algoritmo de menor complejidad, este problema de multiplicar matrices fue estudiado.

El algoritmo de multiplicación de matrices tradicional tenía una complejidad de $\Theta(n^3)$. El surgimiento de diversas técnicas de programación llevó al intento de lograr un algoritmo de una complejidad menor aplicando diversas técnicas, pues el manejo de matrices es una parte fundamental de la informática, un ejemplo de su uso es el tratamiento de imágenes, computacionalmente hablando, es el tratamiento de matrices de enormes dimensiones.

Es importante mencionar que el algoritmo de Strassen parte del hecho de que las matrices son cuadradas y de lado igual a una potencia de dos, a fin de que se puedan dividir correctamente.

Fue Volker Strassen en 1969 quien señaló que si se usara el paradigma de dividir y vencer no era el óptimo. Aunque solo es ligeramente más rápido que el algoritmo estándar para la multiplicación de matrices, su artículo comenzó la búsqueda de algoritmos aún más rápidos.

¿Cómo logró Strassen hacer que la complejidad de su algoritmo fuera subcúbica? La razón principal radica en que, para cada subproblema, tan solo requiere de siete multiplicaciones. El resto de los aspectos del algoritmo tienen aproximadamente la misma complejidad, como el coste de combinar las submatrices.

El algoritmo de Strassen mejora tan sólo levemente la complejidad del algoritmo tradicional para multiplicación de matrices. La combinación de las sub-soluciones las lleva a cabo de exactamente igual forma. La clave reside en que Strassen encontró la forma para tener que llevar a cabo tan solo siete productos de matrices para resolver cada subproblema. Si llevamos a cabo las sustituciones pertinentes, veremos cómo la secuencia de sumas y restas que se llevan a cabo con las submatrices equivale a las distintas sumas de productos que utilizaba el primer algoritmo recursivo.

A continuación, vamos a analizar el algoritmo de Strassen para averiguar exactamente en qué medida mejora al algoritmo tradicional. Comenzando con la recurrencia, sabemos que, para cada problema, debemos llevar a cabo siete productos. Es decir, que, de cada problema, van a surgir siempre ocho

subproblemas. El tiempo necesario para "generar" estos subproblemas, es decir, para dividir las matrices, lo consideraremos constante (Θ(1)).

Nos queda saber el tiempo necesario para fusionar las soluciones parciales (submatrices) para generar una solución mayor. Si consultamos el código 2.2.2.3, veremos cómo el procedimiento para fusionar es fundamental en este algoritmo. Puesto que el procedimiento aquí es exactamente el mismo que entonces, su complejidad sigue siendo proporcional al número de entradas de la matriz, es decir, (Θ(n^2)).

Sabiendo todo esto, y que los casos base tratan simplemente de multiplicar, sumar y restar números, y no matrices, podemos formar la fórmula de recurrencia del algoritmo de Strassen:

Número de subproblemas por cada problema: a = 7.

Tamaño de cada subproblema: b = n/2.

Tiempo invertido en dividir las matrices: D(n) = Θ(1). Esto es despreciable frente a C(n).

Tiempo invertido en fusionar las submatrices: C(n) = Θ().

T(n) = Θ(1) si n = 1

T(n) = 7T(n/2) + Θ(n^2) en el resto

Por el método maestro, podríamos concluir de inmediato que la complejidad de este algoritmo es subcúbica. El valor de a (7) es mayor que el valor de , y por tanto estamos ante el tercer caso que contempla el método maestro. La complejidad de este algoritmo es Θ(n^(log7)). Puesto que el valor del está entre 2.80 y 2.81, concluimos que la complejidad es Θ(n^2.80).

Al hacer la investigación se descubrió que el primer intento para reducir la complejidad del algoritmo normal para la multiplicacion de matrices fracaso, dicho primer algoritmo hacia uso del paradigma divide y vencer y no lograba poder reducir la complejidad, después de la aparición del algoritmo de Stressen se desarrollaron mas algoritmos aún más rápidos como el complejo algoritmo de Coppersmith–Winograd de Shmuel Winograd en 2010.

Nota sobre el funcionamiento del proyecto: La clase MultiplicacionMatrices nos permite probar los dos algoritmos, primero el algoritmo de multiplicación matricial normal y el segundo es multiplicacion matricial usando el algoritmo de Strassen.