Tarea 2

CC3001 – Algoritmos y Estructuras de Datos

Nombre:	Sebastián Parra
Profesor:	Patricio Poblete
Auxiliares:	Sven Reisenegger
	Gabriel Flores
Ayudantes:	Fabián Mosso
	Gabriel Chandía
	Matías Risco
	Matías Ramirez

Introducción

El problema a resolver consiste en determinar el orden óptimo de multiplicación de *n* matrices con distintas dimensiones. Este problema surge debido a que para poder multiplicar una matriz A de p filas por q columnas, con una matriz B de q filas por r columnas, se necesita realizar un total de $p \times q \times r$operaciones, lo que implica que el número de operaciones a realizar al multiplicar más de dos matrices varía según el orden (cumpliendo la propiedad de asociatividad) en que se realizan las operaciones.

A modo de facilitar la comprensión del problema, se plantea el siguiente ejemplo: Se tienen las matrices A (100x10), B (10x100), y C (100x10) y se quiere realizar la operación $A \times B \times C$. Existen dos formas de hacerlo: La primera consiste en multiplicar primer A con B, requiriendo $100 \times 10 \times 100$operaciones obteniendo una matriz de (100x100), y luego el resultado multiplicarlo por C, lo cual requiere de $100 \times 10 \times 100 + 100 \times 10 \times 100 = 200000$ operaciones. La segunda consiste en multiplicar B por C, requiriendo $10 \times 100 \times 10$ operaciones obteniendo una matriz de (10x10), y luego premultiplicar el resultado por A, lo cual requiere un total de $10 \times 100 \times 10 + 100 \times 10 \times 10 = 20000$ operaciones, valor considerablemente menor que el método anterior.

La tarea a resolver consiste en diseñar un programa que reciba como entrada un *string* con las *n+1* dimensiones (en orden) de las *n* matrices a multiplicar, separadas por un espacio y que en su salida devuelva un *string* con la parentización óptima para el problema, representando cada matriz con un punto ("."). Además, se prohíbe el uso de las bibliotecas Array y ArrayList en la implementación de la solución, y se solicita que la entrada sea recibida a través de la entrada estándar de Java y pueda responder a varias instancias seguidas, cada una en una línea distinta.

En cuanto a la solución propuesta, ésta se puede dividir en los siguientes pasos: Primero se extraen los enteros del string de entrada y se guardan en un arreglo. Luego, se utiliza una estructura tipo lista enlazada, pero con punteros al nodo anterior y posterior, para guardar las *n* matrices ingresadas. Posteriormente, se utiliza un método visto en el apunte para calcular una matriz cuyos coeficientes (i,j) indiquen la posición que debiese tener el paréntesis que divide de manera óptima la multiplicación de las matrices $A_i$hasta la A_j en dos subproblemas. Una vez calculada la matriz, se procede a determinar recursivamente la posición de los paréntesis del problema principal y todos sus subproblemas, agregándolos a la lista enlazada donde se encontraban las matrices. Finalmente, se recorre la lista enlazada completa para formar el *string* de salida e imprimirlo.

Diseño de la Solución

Para llegar a la solución implementada, primero se estudió el método `multiplicacionCadenaMatrices` presentado en la sección 2 del apunte, el cual dado un vector de dimensiones de las matrices a multiplicar, calcula mediante programación dinámica las matrices m[i,j] y s[i,j], que determinan el costo mínimo de multiplicar de la matriz $A_i$hasta la A_j y la ubicación del paréntesis que minimiza el costo de esta multiplicación respectivamente. Luego, se pensó utilizar la matriz s[i,j] entregada por el algoritmo para programar un método que resuelva recursivamente las ubicaciones de los paréntesis del problema principal y todos sus subproblemas. Sin embargo, rápidamente se llegó a la conclusión que se necesitaría de un tipo de dato que permitiese manejar arreglos de tamaño variable a los que se pueda agregar elementos tanto a la derecha como a la izquierda de elementos ya existentes, por lo que se implementó la clase `ordenMultiplicacion` que funciona como una lista enlazada, pero contiene punteros para sus vecinos izquierdo y derecho (figura 1).

Figura 1: Ejemplo de objeto tipo `ordenMultiplicacion`

De esta manera, se inicializa el algoritmo ingresando todas las matrices en una lista (ver ejemplo en figura 2), y luego una vez calculada la matriz s[i,j] se comienzan a agregar recursivamente las parentizaciones mediante el método `agregarElemento` que permite ingresar un nodo a la izquierda o derecha de un nodo existente. Finalmente, se diseña un método encargado de leer todos los valores de la lista, de izquierda a derecha, y generar la salida mediante la concatenación de los valores de todos los nodos.

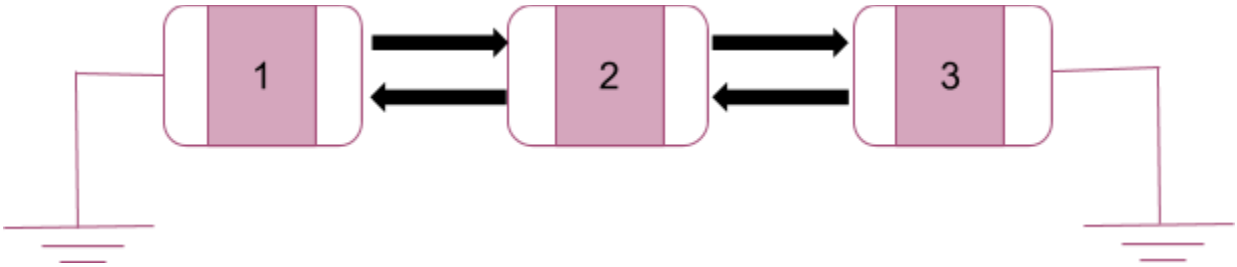

Figura 2: Ejemplo de inicialización de lista con 3 matrices.

En cuanto a los supuestos y casos de borde abordados, se asumió que el usuario respetaría el formato de la entrada de la forma "t_0 t_1 … t_n", pero se contaron los casos de 0, 1, 2 y 3 dimensiones como casos de borde: Si el usuario no ingresa ninguna dimensión o sólo ingresa una, se mostrará un mensaje en pantalla indicando que la entrada es inválida y se dará la opción de ingresar otra entrada. En cambio, si el usuario ingresa dos (sólo 1 matriz) o tres (sólo 2 matrices, caso trivial) dimensiones, se imprimirá en pantalla el *string* "`(.)`" o "`(..)`" según corresponda.

Implementación

A continuación, en la figura 3 se muestra el extracto de la función `main()` que se encarga de producir la salida a partir de la entrada. Posteriormente se irá explicando en detalle cada bloque por separado.

```java
//Leer una linea de strings
String linea = entrada.nextLine();

//Obtener un arreglo de los strings de las dimensiones de las
//matrices
String[] dimensiones = linea.split(" ");
if(dimensiones.length <= 1) {
    System.out.println("Por favor ingrese una entrada valida");
    continue;
}
int[] dims = new int[dimensiones.length];

//Parsear arreglo de strings a enteros
for(int i = 0; i < dims.length; ++i) {
    dims[i] = Integer.parseInt(dimensiones[i]);
}

//Crear lista de matrices
ordenMultiplicacion listaMatrices = agregarMatrices(dims.length-1);

//Crear matriz que indica el costo optimo de
//multiplicar de la mariz Ai hasta la Aj en numero de operaciones
int[][] matrizCosto = new int[dims.length][dims.length];

//Crear matriz que indica la ubicacion del parenesis para
//multiplicar de manera optima de la mariz Ai hasta la Aj
int[][] lugarParentesis = new int[dims.length][dims.length];

//Rellenar las matrcies de costo y de parentesis utilizando el
//del apunte
multiplicacionCadenaMatrices(dims, matrizCosto, lugarParentesis);

//Agregar los parentesis a la lista de matrices e imprimir resultado
listaMatrices.parentesisOptimo(1, dims.length-1, lugarParentesis);
listaMatrices.imprimirString();
```

Figura 3: Extracto de la función `main()` que resuelve el probema.

Parte 0: Diseñando la clase `ordenMultiplicacion`

Al diseñar la clase, se decidió por designar los valores como enteros, ya que esto permite realizar operaciones matemáticas y comparativos sobre éstos si llega a ser necesario. Debido a esto, se decide codificar las matrices por un entero que indique si índice (ver ejemplo en la figura 2) y los paréntesis por un cero si corresponde al paréntesis derecho ")" y un -1 si corresponde al paréntesis izquierdo "(". Este formato se encuentra debidamente comentado dentro del código, como se puede apreciar en la figura 4.

```
public class ordenMultiplicacion {
    //Clase de tipo lista enlazada cuyos nodos pueden representar una matriz
    //o un parentesis. Si el nodo es una matriz, su valor sera su indice.
    //Si es un parentesis, si valor sera 0 si corresponde al caracter ')'
    //y -1 si corresponde a '('
    private int val;
    private ordenMultiplicacion sig;
    private ordenMultiplicacion ant;
```

Figura 4: Declaración de los atributos de la clase `ordenMultiplicacion` y el formato a utilizar para la representación de caracteres.

Parte 1: Obteniendo un formato útil a partir de la entrada

Una vez obtenido el *string* con la entrada codificada como corresponde, se procede a utilizar el método `split()` para transformar la entrada a un arreglo de *strings* donde cada índice almacenará una dimensión de las matrices. Luego, si se reconoce la entrada como válida (o sea, si existe más de una dimensión en el arreglo), se recorre el vector para convertir cada uno de sus valores en enteros utilizando la función `parseInt()` y así poder realizar operaciones matemáticas con ellos. Finalmente, se calcula la cantidad de matrices a multiplicar a partir de la cantidad de dimensiones ingresadas, y mediante el método `agregarMatrices()` se crea un objeto tipo `ordenMultiplicacion` con todas las matrices indexadas (como en el ejemplo de la figura 2), retornando un puntero a la cabecera de la lista (en este caso, la primera matriz).

Parte 2: Calculando la cantidad mínima de operaciones y la ubicación óptima de los paréntesis

En esta parte se comienza generando las matrices de costo (`matrizCosto`) y de parentización (`lugarParentesis`) que serán utilizadas, junto al vector de enteros con las dimensiones de las matrices obtenido en la parte anterior, como argumentos en la función `multiplicacionCadenaMatrices` mostrada en el apunte del curso. Como para este problema no importa la cantidad de operaciones necesarias fuera del cálculo de la parentización óptima, se modificó esta función para no retornar nada, pero fuera de este detalle, como se puede ver en la figura 4, la función permanece intacta.

```java
//INICIO METODO DEL APUNTE
//Metodo estatico, resuelve utilizando programacion dinamica el problema de
//encontrar la posicion de parentesis que optimiza la multiplicacion de n
//matrices y todos sus subproblemas. El arreglo p corresponde a un vector con
//con las dimensiones de las matrices a multiplicar, la matriz m[i,j] indica el
//costo minimo de multiplicar de la matriz i a la matriz j, y la matriz s[i,j]
//indica la posicion que debe tener el parentesis para multiplicar optimamente
//de la matriz i a a matriz j (no retorna nada)
public static void multiplicacionCadenaMatrices(int[] p, int[][] m, int[][] s) {
    // Matriz Ai tiene dimensiones p[i-1] x p[i] para i = 1..n
    // El primer indice para p es 0 y el primer indice para m y s es 1
    int n = p.length - 1;
    for (int i = 1; i <= n; i++) {
        m[i][i] = 0;
    }
    for (int l = 2; l <= n; l++) {
        for (int i = 1; i <= n - l + 1; i++) {
            int j = i + l - 1;
            m[i][j] = Integer.MAX_VALUE;
            for (int k = i; k <= j-1; k++) {
                int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
    return;
}
//FIN METODO DEL APUNTE
```

Figura 5: Función `multiplicacionCadenaMatrices` implementada en la solución.

Parte 3: Ingresando los paréntesis a la lista de matrices

En esta sección se utiliza el método `parentesisOptimo()`, cuya implementación se puede apreciar en la figura 6, que a partir de la matriz de parentización calculada en la parte anterior, determina la ubicación que debe tener el paréntesis si se quiere multiplicar de la matriz en la posición i hasta la posición j, luego agrega los paréntesis necesarios a la lista de matrices llamando al método `agregarParentesis()`, y finalmente llama recursivamente a encontrar la parentización óptima para resolver la multiplicación de los dos subproblemas restantes (desde i hasta la posición del paréntesis, y desde la posición del paréntesis hasta j).

```java
//Metodo de instancia, determina la posicion optima del parentesis para
//multiplicar de la matriz inicio a la matriz fin a partir de una matriz de
//parentesis s[][], lo agrega a la lista de matrices, y luego resuelve los
//subproblemas restantes (no retorna nada)
public void parentesisOptimo(int inicio, int fin, int[][] s) {
    //Determinar posicion optima del parentesis
    int k = s[inicio][fin];

    //Caso base: Si solo es una matriz, no hacer nada
    if(inicio == fin) {
        return;
    }
    else {
        //Agregar los parentesis y resolver los subproblemas
        this.agregarParentesis(inicio, fin, k);
        this.parentesisOptimo(k+1, fin, s);
        this.parentesisOptimo(inicio, k, s);
        return;
    }
}
```

Figura 6: Implementación del método `parentesisOptimo()`

En cuanto al método `agregarParentesis()`, éste recibe los índices de inicio y fin del problema o subproblema, junto con el índice en el cual se desea colocar los paréntesis. Su implementación se puede dividir en 2 pasos: Primero se recorre la lista y se guardan los nodos que contienen las ubicaciones de las matrices en los índices de inicio, fin y parentización. Luego, se revisan los casos de borde en donde sólo existe una matriz hacia algún lado de donde se desee ubicar el paréntesis, ya que si este es el caso no se debiesen colocar paréntesis hacia ese lado (por ejemplo, si se quieren multiplicar 3 matrices y se desea poner la parentización después de la segunda matriz, sólo tiene sentido poner los paréntesis que van desde la matriz 1 a la matriz 2, ya que poner un paréntesis encerrando sólo a la matriz 3 sería redundante). Su implementación se puede encontrar en la figura 7.

```java
//Metodo de instancia, agrega los parentesis necesarios sabiendo que para
//multiplicar de manera optima las matrices desde ini hasta fin se
//requiere poner un parentesis en la matriz k (no retorna nada)
public void agregarParentesis(int ini, int fin, int k) {
    //Paso 1: Guardar nodos que tengan las matrices ini, fin y k
    ordenMultiplicacion inicio = this;
    ordenMultiplicacion division = this;
    ordenMultiplicacion termino = this;
    while (termino.val != fin) {
        if (termino.val == ini) {
            inicio = termino;
        }
        if(termino.val == k) {
            division = termino;
        }
        termino = termino.sig;
    }

    //Paso 2: Revisar si hay mas de una matriz a la derecha de la division
    if((fin - k) > 1) {
        division.agregarElemento(-1, "der");
        termino.agregarElemento(0, "der");
    }
    //Paso 3: Revisar si hay mas de una matriz a la izquierda de la division
    if(k != ini) {
        division.agregarElemento(0, "der");
        inicio.agregarElemento(-1, "izq");
    }
    return;
}
```

Figura 7: Implementación del método `agregarParentesis()`

Parte 4: Imprimiendo la salida con el formato solicitado

Finalmente, en esta sección se utiliza el método `imprimirString()`, que se encarga de recorrer la lista de matrices y paréntesis hasta llegar al primer nodo (que no necesariamente será el que corresponde a la primera matriz, ya que se pudieron haber puesto paréntesis antes de ésta), y luego recorre toda la lista de izquierda a derecha, convirtiendo los valores de cada nodo en un *string* de acuerdo a la codificación solicitada en el enunciado ("." para matrices, "(" y ")" para paréntesis). Así, la salida a imprimir corresponde simplemente a la concatenación de todos los carácteres más dos paréntesis adicionales que encierran toda la expresión. Su implementación se puede apreciar en la figura 8.

```java
//Metodo de instancia, dada una lista tipo ordenMultiplicacion, imprime sus
//Valores siguiendo la codificación indicada en el enunciado
public void imprimirString() {
    //Paso 0: Inicializar string con un parentesis izquierdo
    String salida = "(";

    //Paso 1: Recorrer lista hasta el primer nodo
    ordenMultiplicacion actual = this;
    while(actual.ant != null) {
        actual = actual.ant;
    }

    //Paso 2: Recorrer lista hasta el ultimo valor e ir agregando caracteres
    while (actual != null) {
        if(actual.val == 0) {
            salida += ")";
        }
        else if(actual.val == -1) {
            salida += "(";
        }
        else {
            salida += ".";
        }
        actual = actual.sig;
    }

    //Paso 3: Agregar parentesis final
    salida += ")";
    System.out.println(salida);
}
```

Figura 8: Implementación de la función `imprimirString()`

Resultados

A continuación, en la figura 9 se presentan los resultados obtenidos para los casos de borde tratados y los ejemplos mostrados en el enunciado.

```
sparra@sparra-Aspire-E5-575G:~/Desktop/Algortimos/Tarea2/bin$ java ordenMultipli
cacion
Ingrese secuecias(s) de strings:

Por favor ingrese una entrada valida
1
Por favor ingrese una entrada valida
1 2
(.)
Ejecucion terminada
1 2 2
(..)
Ejecucion terminada
2 3 5 4
((..).)
Ejecucion terminada
2 3 5 4 1 2
((.(.(..))).)
Ejecucion terminada
```

Figura 9: Resultados obtenidos para a) Entradas con 0 y 1 dimensión (casos invalidos) b) Entradas con 2 y 3 dimensiones (casos triviales) c) Ejemplos mostrados en el enunciado.

Conclusiones

A través del trabajo realizado se puede concluir que se ha resuelto el problema propuesto, pudiendo incluso agregar condiciones de borde que, si bien no debiesen ser nunca introducidas por un usuario, es bueno evitar estas suposiciones para poder lograr un programa más robusto. Sin embargo, se debe notar que se podría trabajar aún más en esta área, ya que actualmente el programa no está diseñado para tratar otros casos de entradas invalidas (como, por ejemplo, entradas que no sigan el formato propuesto en el enunciado).

Además, al desarrollar este programa se pudo evidenciar un ejemplo práctico de la utilidad de la programación dinámica como una herramienta para reducir operaciones redundantes y en general disminuir tiempos de ejecución, lo cual es un problema que toma gran importancia para aplicaciones que se basan en modelos complejos o necesitan procesar una gran cantidad de datos, ya que en estos ejemplos los tiempos de ejecución pueden escalar enormemente.

Finalmente, se destaca también el aprendizaje práctico logrado en cuanto al diseño de TDA's que puedan ayudar a resolver un problema dado, especialmente debido a las restricciones impuestas sobre el uso de las librerías Array y ArrayList.