Programación dinámica:

Multiplicación encadenada de matrices

- **Imar Abreu Díaz**
- **Carlos García González**
- **Richard Morales Luis**

Índice.

Introducción.

Existe una serie de problemas cuyas soluciones pueden ser expresadas recursivamente, y la manera más natural de resolverlos es mediante un algoritmo recursivo. Sin embargo, el tiempo de ejecución de la solución recursiva, normalmente es de orden exponencial y por tanto impracticable. Esto puede mejorarse substancialmente mediante la Programación Dinámica.

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas. Tiene una mayor implicación en problemas de optimización. En este tipo de problemas se puede presentar distintas soluciones, pero lo que se desea encontrar es el valor óptimo.

En este trabajo abordaremos el algoritmo de multiplicación de matrices encadenadas haciendo uso de programación dinámica. Las multiplicaciones de matrices y otras operaciones, tienen uso en el tratamiento y procesamiento de imágenes. Es muy útil para la resolución de sistemas de ecuaciones con un número muy elevado de variables, ya que resulta más fácil su implementación mediante un ordenador. El cálculo numérico, también se basa en gran parte de estas operaciones, al igual que aplicaciones como MATLAB y Octave. Otro uso que actualmente se utiliza, es el cálculo de microarrays, en el área de bioinformática.

La operación que realizaremos en este trabajo, el de multiplicar matrices de forma consecutiva, tiene una complejidad alta. Si se usase un algoritmo de fuerza bruta, su tiempo de ejecución sería de $4^n/n$, siendo n el número de matrices a multiplicar. Al ser un tiempo exponencial, se trata de una solución prácticamente intratable como nombramos anteriormente. Por el contrario, al atajar el problema mediante un algoritmo de programación dinámica, este tiempo de ejecución se ve reducido a n^3. Esta diferencia en el tiempo de ejecución, deja claro la mejoría al usar programación dinámica en este problema.

Descripción del problema.

Supongamos que tenemos un conjunto de matrices que queremos multiplicar de tal manera que tengamos:

$M = M_1 \times M_2 \times M_3 \times \ldots \times M_n$.

Sabemos que la operación del producto es asociativa y por ello podemos multiplicar las diferentes matrices en el orden que queramos respetando siempre las condiciones para que ambas se puedan multiplicar.
Para ello la colocación de los paréntesis marcará el orden diferente en el que se pueden realizar las operaciones.
Según el orden de las multiplicaciones, el número total operaciones escalares necesarias puede variar considerablemente.

Ejemplo: Supongamos las matrices A,B,C y D. Cuyas dimensiones son:

- A = 13 x 5
- B = 5 x 89
- C = 89 x 3
- D = 3 x 34

Con estos ejemplos podemos comprobar lo anteriormente afirmado.
Para el caso de la multiplicación donde el orden de las operaciones es:

((AB)C)D -> Tiene 10582 = 13x5x89 + 13x89x3 + 13x3x34.
(AB)(CD) -> Tiene 54201 = 13x5x89 + 89x3x34 + 13x89x34.
(A(BC))D -> Tiene 2886 = 5x89x3 + 15x5x3 + 13x3x34
A((BC)D) -> Tiene 4055 = 5x89x3 + 5x3x34 + 13x5x34
A(B(CD)) -> Tiene 26418 = 89x3x34 + 5x89x34 + 13x5x34

SOLUCIÓN DEL PROBLEMA.

Tras observar esto, la idea que nos viene es la de conseguir una combinación de tal manera que se minimice el número de operaciones para una entrada de N Matrices multiplicables entre sí.

SOLUCIÓN 1. Sin programación dinámica.

Para ello una solución podría ser intentar encontrar todas las posibles combinaciones con su respectivo número de operaciones y escoger el menor.

Si analizamos esta posible solución al problema si tenemos entradas pequeñas por ejemplo N=1 o N=2 el tiempo T(n) = 1 pues simplemente habría que multiplicar ambas matrices sin tener que contar ninguna combinación.

Por otro lado, en el caso de que N>2 tendremos que realizar la primera multiplicación por n-1 sitios distintos.

Ejemplo:

Para un N=3 con matrices A, B, C las posibles multiplicaciones serían:

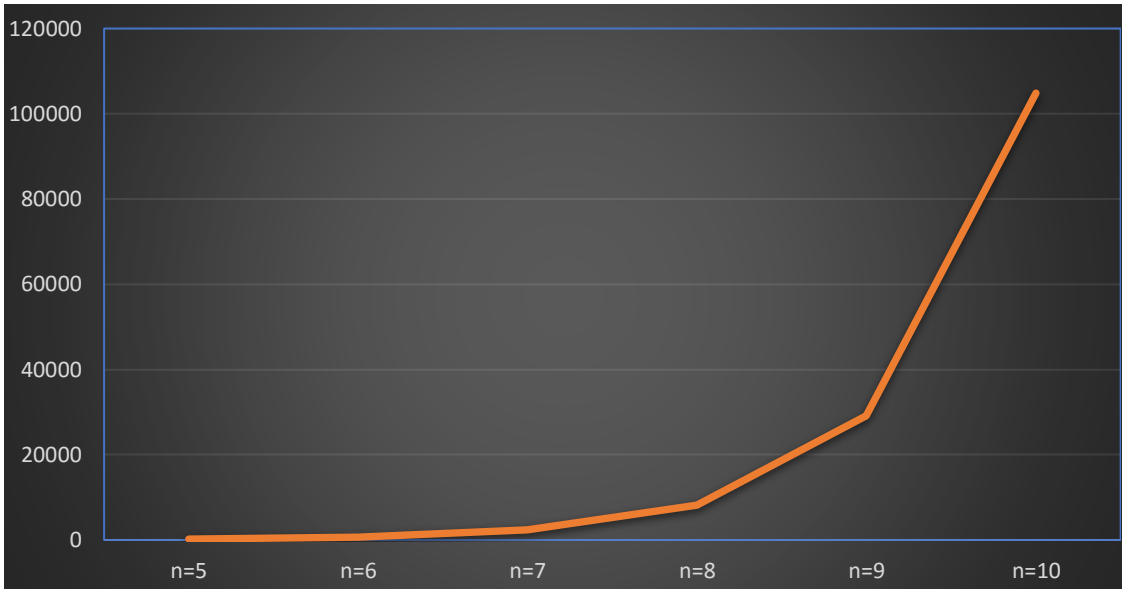
(AB)C o A(BC).

Por tanto, tendríamos un tiempo T(n) = $\sum_{i=1}^{n-1} T(i)T(n-i)$. De manera que el valor obtenido resolviendo dicha ecuación es:

T(n) que estaría en $O(4^n/n^2)$.

Como para cada ordenación tenemos un tiempo $\Theta(n)$ dicha solución requerirá de un tiempo:

$T(n) = O(4^n/n)$.



SOLUCIÓN 2. Con programación dinámica.

Para utilizar un algoritmo de programación dinámica deberemos escoger una o varias estructuras de datos que nos permitan almacenar la cantidad de operaciones que debemos realizar con las diferentes combinaciones.
Para ello usaremos una matriz **NMulti(i,j)** que contendrá el número mínimo de productos escalares desde la matriz i hasta la j donde **(i ≤ j).**
Además, usaremos otra estructura de datos en forma de vector **D[0…N]** para guardar los tamaños de las matrices que usaremos para hallar el número de operaciones siguiendo el cálculo de la primera parte.

Supongamos que tenemos un conjunto de N matrices a multiplicar. La tabla estará formada por n filas y n columnas donde las posiciones i,j contendrán el valor óptimo de operaciones para multiplicar las matrices desde la i hasta la j.
Como el producto de matrices no es conmutativo solo podremos seguir un orden concreto a la hora de multiplicar las matrices. Es decir, no es lo mismo realizar la multiplicación de AxB que BxA. Con estos datos podemos llegar a la conclusión que nuestra tabla final solo se rellenará por la mitad pues la otra mitad es imposible de resolver debido a que el orden en la multiplicación de las matrices sí importa.
La tabla la rellenaremos por diagonales, donde identificaremos cada una con la letra "s", donde el valor de la misma irá desde 1 hasta n-1 y será igual a j-i. Por ejemplo la diagonal s=0 contendrá los valores M(i,i) y será igual a 0 porque se corresponde con el número de operaciones desde la matriz i hasta la matriz i, al ser la misma matriz, no realiza ninguna multiplicación y por ello el valor será 0. Esto ocurrirá para todos los valores de la diagonal principal. Para las diagonales s>0 tendremos que realizar las operaciones desde la matriz "i" hasta la matriz "i+s". Ahora bien, el producto de dichas matrices se puede hacer de diferentes maneras y esto varía depende el corte (colocación de un paréntesis) que hagamos. Podemos realizar un corte hasta la matriz M(i+s-1) separando en dos partes izquierda y derecha las matrices. La parte izquierda iría desde la matriz M_i hasta la matriz M_k siendo k un valor entre i y j. ($i \leq k<j$).

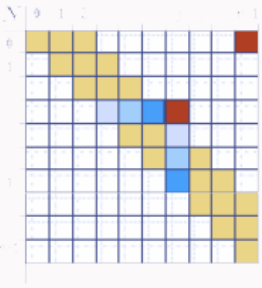
Y la parte derecha iría desde M_{k+1} hasta M_{i+s}. Tras realizar los diferentes productos habría que multiplicar ambas matrices resultantes para obtener el resultado de la matriz final. Como ya sabemos hallar el número de operaciones de un producto de dos matrices de tamaño PxQ y QxS = PxQXS operaciones para para resolver ese producto, simplemente deberemos hallar el mínimo número de operaciones según para todos los cortes posibles.
De tal manera que la ecuación quedaría de la siguiente forma:

$$\text{NMulti}(i,j) = \begin{cases} 0 & \rightarrow i=j \\ \min\{ M[i,k] + M[k+1,j] + D[i-1]xD[k]xD[j]\} & \rightarrow i<j \end{cases}$$

Donde se representa cada corte y su posterior unión en cada una de las matrices.

Pseudocódigo

Implementación:



```
for (i=1; i<=n; i++)
    m[i,i] = 0;

for (s=2; s<=n; l++)
    for (i=1; i<=n-s+1; i++) {
        j = i+s-1;
        m[i,j] = ∞;
        for (k=i; k<=j-1; k++) {
            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;
            }
        }
    }
```

Implementación:

```
// Suponiendo que hemos calculado previamente s[i,j]…

MultiplicaCadenaMatrices (A, i, j)
{
  if (j>i) {
     x = MultplicaCadenaMatrices (A, i, s[i,j]);
     y = MultplicaCadenaMatrices (A, s[i,j]+1, j);
     return MultiplicaMatrices(x, y);
  } else {
     return A[i];
  }
}
```

Ejemplo:

Si aplicamos esto al ejemplo del principio obtendremos que:
El valor del vector D{13,5,89,3,34}.
Los valores de la tabla serían los siguientes:

Implementación:

```
// Suponiendo que hemos calculado previamente s[i,j]…

MultiplicaCadenaMatrices (A, i, j)
{
  if (j>i) {
     x = MultplicaCadenaMatrices (A, i, s[i,j]);
     y = MultplicaCadenaMatrices (A, s[i,j]+1, j);
     return MultiplicaMatrices(x, y);
  } else {
     return A[i];
  }
}
```

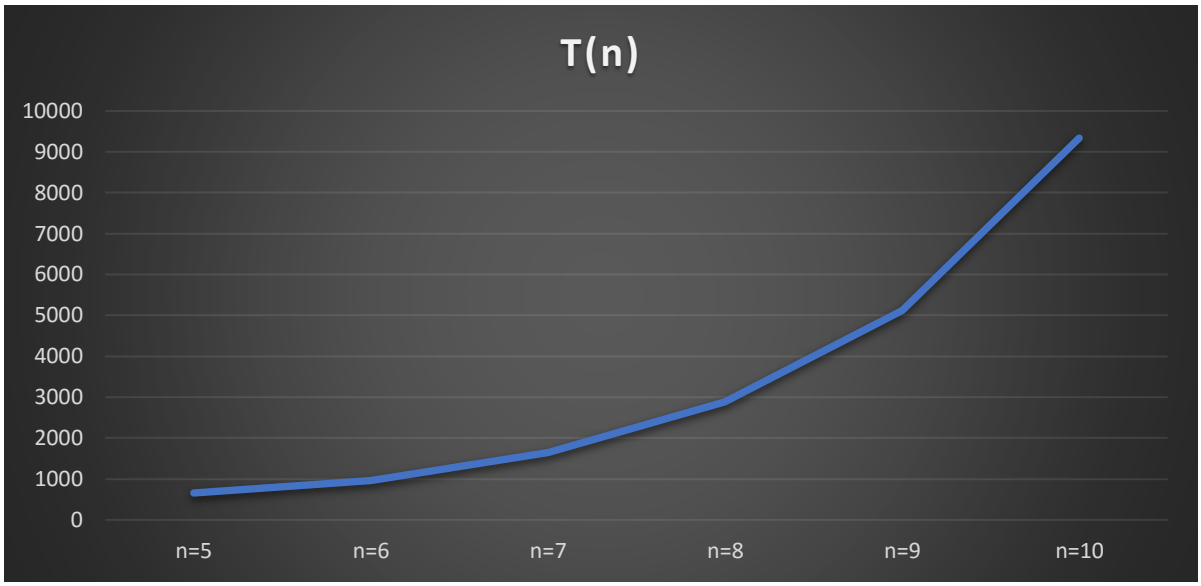
	J=1	2	3	4	
I=1	0	5785	1530	2856	S=3
2	-	0	1335	1845	S=2
3	-	-	0	9078	S=1
4	-	-	-	0	S=0

Complejidad.

Sabemos que para un s>0 hay que calcular n-s elementos en la diagonal. Para cada uno de ellos tendremos que hacer "s" posibilidades dadas por los distintos valores de "k" por ello el tiempo de ejecución del algoritmo esta en:

$\sum_{s=1}^{n-1}(n-s)s = n\ \sum_{s=1}^{n-1} s - \sum_{s=1}^{n-1} s^2$ = **n^2 (n-1)/2 – n(n-1)(2n-1)/6**

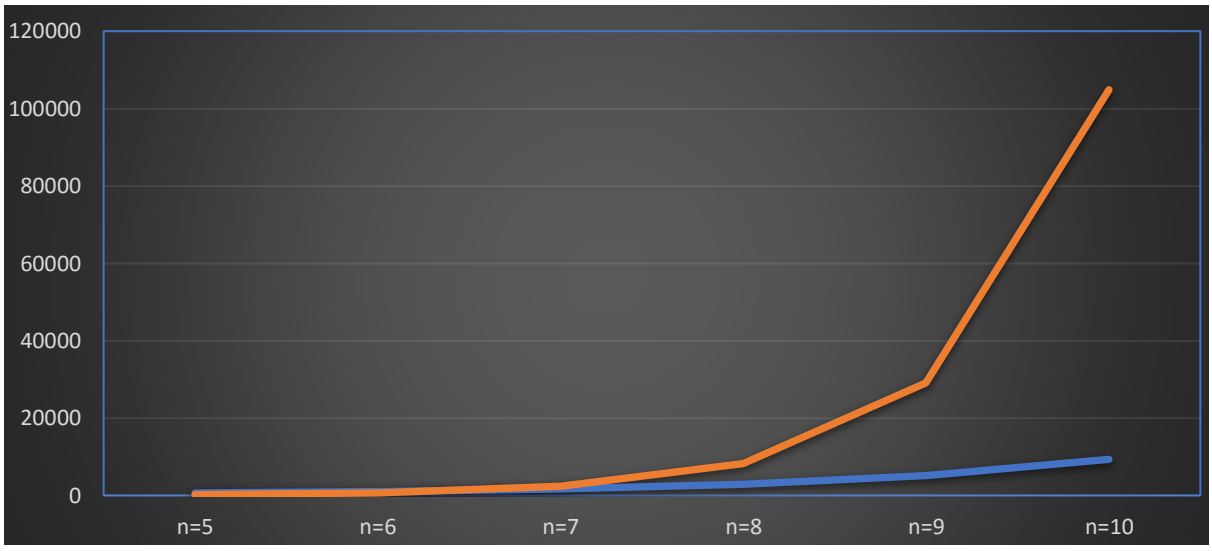
= (n^3 –n)/6
Por ello podemos concluir que el algoritmo tiene una complejidad:
T(n) = O(n^3).



Conclusiones.

Como se ha podido observar en las pruebas, el algoritmo de fuerza bruta es bueno hasta una cierta entrada N en la cual el tiempo que requiere se dispara llegando a ser intolerable. Por otro lado, vemos como el algoritmo de programación dinámica para una entrada pequeña es un poco menos eficiente que el algoritmo de fuerza bruta, pero a medida que la entrada crece resulta ser mucho más eficiente debido a que no realiza tantas operaciones y combinaciones como haría el de fuerza bruta. Con esto cabe concluir que, aunque sacrifiquemos memoria en las estructuras de datos al final el tiempo de ejecución es mucho menor y por ello más eficiente.

Como podemos ver en la gráfica:



Bibliografía

- https://es.wikibooks.org/wiki/Optimizaci%C3%B3n_del_Producto_de_Matrices
- http://dis.unal.edu.co/profesores/jgomez/courses/algorithmics/documents/progDinamica.pdf
- http://elvex.ugr.es/decsai/algorithms/slides/6%20Dynamic%20Programming.pdf