MATRIX CHAIN MULTIPLICATION PROBLEM

USING DYNAMIC PROGRAMMING

A PROJECT REPORT

Submitted by
Tanmay Agrawal(19BCT0134)
Abhay Sharma (19BCT0125)

Course Code: CSE2003
Course Title: DATA STRUCTURES AND ALGORITHMS

Under the guidance of
ARUN KUMAR G
Assistant Professor(Senior),
School Of Computer Science & Engineering,
VIT University, Vellore.

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Aim

The Goal of the project is to decide the sequence of matrix multiplications involved to perform said calculations efficiently.

Object

The main object of this project is to obtain an efficient algorithm for multiplying multiple matrices in a particular order and to also do so in a fast process.

Problem Statement and commercial application

Matrix multiplications is used in many optimizing processes for rendering images and videos but when the quality of image increases the time required to render it also increases because the process involves multiplication of multiple matrices to achieve the desired look.

The first image is the actual image taken from a camera, whereas the second one is the image formed after removing lens distortion.

This process uses matrix chain multiplication. This project is thus made to make the process more efficient.

Abstract

Matrix Chain Multiplication(also known as the matrix chain ordering problem) is one of the most popular examples of dynamic programing being used to solve optimization issues.

This project includes an introduction to the methods of matrix chain multiplication, the criteria used to decide the most efficient way and an example of how the above mentioned methods have been put to use. The algorithm and code for the said problem is also included.

Introduction

Matrix Chain Multiplication or MCOP helps in deciding the most optimum way in which multiplication should be carried out. This optimisation problem can be shown with the help of dynamic programming.It mainly focuses on the sequence in which multiplication should be carried out rather than the actual multiplication. Input includes a chain of matrices to be multiplied and output includes a parrenthizing of the chain. The main objective is to reduce the number of steps in multiplication.

it is said that matrix multiplication is associative but not commutative in nature. For matrix multiplication, if we have matrices A(m*n) and B(n*p), then the product will be AB(m*p). In order to demonstrate the commutative and associative property of matrix multiplication, the following examples are demonstrated.

NOT COMMUTATIVE

If $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

then

$$AB = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1)(5)+(2)(7) & (1)(6)+(2)(8) \\ (3)(5)+(4)(7) & (3)(6)+(4)(8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

and

$$BA = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} (5)(1)+(6)(3) & (5)(2)+(6)(4) \\ (7)(1)+(8)(3) & (7)(2)+(8)(4) \end{bmatrix} = \begin{bmatrix} 23 & 34 \\ 31 & 46 \end{bmatrix}$$

which clearly demonstrates that
matrix multiplication is not commutative.

ASSOCIATIVE

Matrix multiplication is associative.

i.e., (AB)C = A(BC)

Example :

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}, B = \begin{bmatrix} 2 & 3 \\ 1 & 5 \end{bmatrix} \text{and } C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$AB = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ 1 & 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 1\times 2+3\times 1 & 1\times 3+3\times 5 \\ 2\times 2+1\times 1 & 2\times 3+1\times 5 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 2+3 & 3+15 \\ 4+1 & 6+5 \end{bmatrix} \Rightarrow \begin{bmatrix} 5 & 18 \\ 5 & 11 \end{bmatrix}$$

$$(AB)C \Rightarrow \begin{bmatrix} 5 & 18 \\ 5 & 11 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 5\times 1+18\times 0 & 5\times 0+18\times 1 \\ 5\times 1+11\times 0 & 5\times 0+11\times 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 5+0 & 0+18 \\ 5+0 & 0+11 \end{bmatrix} \Rightarrow \begin{bmatrix} 5 & 18 \\ 5 & 11 \end{bmatrix}$$

$$BC = \begin{bmatrix} 2 & 3 \\ 1 & 5 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 2\times 1+3\times 0 & 2\times 0+3\times 1 \\ 1\times 1+5\times 0 & 1\times 0+5\times 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 2+0 & 0+3 \\ 1+0 & 0+5 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 3 \\ 1 & 5 \end{bmatrix}$$

$$A(BC) \Rightarrow \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ 1 & 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 1\times 2+3\times 1 & 1\times 3+3\times 5 \\ 2\times 2+1\times 1 & 2\times 3+1\times 5 \end{bmatrix}$$

Hence, (AB)C=A(BC) and matrix multiplication is associative.

Optimization

The process of selecting the best alternative of all, fulfilling all the criteria, is called optimisation.

Principle Of Optimality:

An optimal policy has the property that when whatever the initial state and initial decision are, the remaining decision must constitute an optimal policy with regard to the state resulting from the first decision.

Dynamic Programming

Dynamic programming is a method for solving a complex problem by dividing it down into a collection of simpler some problems solving each of the subproblems just once, and storing their solutions. Whenever the subproblem occurs, instead of computing the solution, the previously computer solution can be inferred, thus saving competition time.

Example- multiplication of 2 x 4 x 5 x 10

we parenthese like say,(((2 x 4) x 5) x 10)

we now solve 2 x 4=8 and store it. ((2 x 4) x 5) is computed as 8 x 5=40. Finally, (((2 x 4) x 5) x 10)=40 x 10=400

so instead of again beginning to calculate from scratch we use the already computed values to reduce computing time.

When developing a dynamic programming algorithm we follow a sequence of four steps:

- <u>Characterize the structure of an optimal solution</u>: We roughly define the logic to be followed in order to get a solution.
- <u>Recursively defined the rate of an optimal solution</u>: The logic is redefined such that for a particular step the logic calls itself with its sub part (recursion)
- <u>Compute the value of an optimal solution, typically in a bottom up fashion</u>: Since the logic is recursive, it device the complex problem into very small parts, then start solving the problem and start wrapping them up into larger subparts, and finally, the complete solution. but the solution is just a final way by which we can get the optimal solution
- <u>Construct an optimal solution from computer information</u>: If it is asked to find the optimal solution, then we start putting the values of the solution we achieved from previous step to get the optimal solution.

Matrix Chain Multiplication

A matrix multiplication, as we have discussed before, is associative but not commutative. When two matrices A(mxn) and B(nxp) are multiplied, the total cost of multiplication is m x n x p, because n elements from m rows of A are multiplied to n elements of B. Suppose there are 4 matrices to be multiplied, M(2x3), N(3x4), O(4x5), P(5x6). There can be various ways of grouping them: (MxN)x(OxP) or (Mx(NxO))xP or ((MxN)xO)xP and so on.

for case 1, cost is: 2x3x4 + 4x5x6 +2x4x6 =192

for case 2, cost is: 3x4x5 + 2x3x5 + 2x5x6=150

for case 3, cost is: 2x3x4 + 2x4x5 + 2x5x6=124

out of just these three cases we can see case 3 is more optimized as compared to case 1 and 2.

Therefore this program is basically finding out the most optimized parenthesizing.

When we are asked to multiply a sequence of matrices:

$$A_i, A_{i+1}, \ldots.., A_{j-1}, A_j$$

The possible approaches to parenthesize the matrices might be-

- ➔ Hit and Trial method: a bad approach because probability to get a correct guess is very low. It is inefficient as well as very difficult to apply in programming.
- ➔ Naive Approach: another bad approach, to try out all the possible combinations, then pick the most efficient or optimal one. This is very time consuming and unrealistic for very long sequences. The time complexity is $\mathbf{\Omega}(4^n/n^{3/2})$.

The better way is using Dynamic Programming.

PROCESS: Check if the problem has optimal Structure.

Let us assume we have an optimal solution for $A_i \ldots A_j$ with parenthesis: $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$.

If there is a better way to multiply $(A_i \ldots A_k)$, we would get a more optimal solution, This would be a contradiction, as we already started that we have optimal solution for $A_i \ldots A_j$.

Therefore, This problem has optimal structure (proof by Contradiction).

Now, for an efficient solution, we need to find out which “k” returns the fewest number of multiplication.

So, we define a recursive formula:

where M[i,j] is the cost of multiplying matrices from A_i to A_j

and, $M[i,j] = M[i,k] + M[k+1, j] + P_{i-1} P_k P_j$ where P denotes dimension.

Example: $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ be the final two matrices to be multiplied.

$(A_i \ldots A_k)$ has dimension 2x3 and cost, say, 1000.

$(A_{k+1} \ldots A_j)$ has dimension 3x5 and cost, say, 100.

Now, M[i,j] = 1000 + 100 + 2x3x5= 1130.

To find best “k”, we iterate k from i to k<j as i<=k<j, and thus:

$$M[i,j] \begin{cases} 0, \text{ if } i=j \\ \min_{i<=k<j} \{M[i,j] = M[i,k] + M[k+1,j] + P_{i-1} P_k P_j\} \end{cases}$$

Another approach can be the recursive matrix chain, which works on the principle of divide and conquer. If the subchain becomes optimized, the main chain is tend to become optimized.

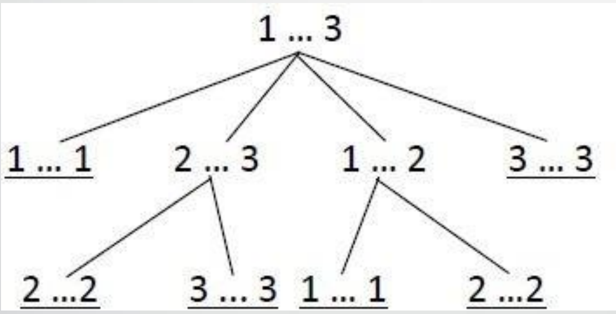
ALGORITHM:

```
/*
        P is an array storing details about dimension of matrices.
        i is the initial index of string(chain) of matrices.
        j is the final index of chain.
*/
```

```
MatrixChainRecursive(p[],i,j)

{
        if(i==j)
                return 0;
        initialize m[i,j] to greatest integer;  for
        every k  E        [i,j-1].
        {
                declare cost = MatrixChainRecursive (p,i,k)
                        + MatrixChainRecursive (p,k+1,j) + P[i-1] * P[k] * P[j];
                if(cost < M[i,j])
                        M[i,j] =cost;
                return (m[i,j]);
        }
}
```

The time complexity is $O(2^n)$.

So, the recursion tree for say, MatrixChainRecursive(p,1,3) locks like-



It is observed that there are repetition of multiplication. When j>>i, a lot of time would be wasted calculating same data over and over again. Thus, this wastage of time can be minimized by storing them. Which is exactly what we do in Dynamic Programming. Thus, Memorised Matrix Chain Multiplication is a better option.

Logic Behind the Algorithm of Memoized Matrix Chain Multiplication

The formula for matrix chain multiplication is

$$M[i,j] \begin{cases} 0, \text{ if } i=j \\ \min_{i<=k<j} \{M[i,j] = M[i,k] + M[k+1,j] + P_{i-1}P_kP_j\} \end{cases}$$

Suppose the matrices are

A_1 A_2 A_3 4x10 A_4 A_5

10x3 3x12 12x20 20x7

From the table

	1	2	3	4	5
1	0	120	264	1080	1344
2	x	0	360	1320	1350
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

where

$P_0 = 4$ $P_1 = 10$ $P_2 = 3$ $P_3 = 12$ $P_4 = 20$ $P_5 = 7$

According to the formula

$M[1,1] = M[2,2] = M[3,3] = M[4,4] = M[5,5] = 0$

$M[1,2] = \min_{1\le k<2}\{M[1,1] + M[2,2] + 4 \times 10 \times 3\}$

$= \min_{1\le k<2} \{0+0+120\}$

$= 120$

$M[2,3] = \min_{2\le k<3}\{M[2,2] + M[3,3] + 10 \times 3 \times 12\}$

$= \min_{2\le k<3} \{0+0+360\}$

$= 360$

$M[3,4] = \min_{3\le k<4}\{M[3,3] + M[4,4] + 3 \times 12 \times 20\}$

$= \min_{3\le k<4} \{0+0+720\}$

$= 720$

$M[4,5] = \min_{4\le k<5}\{M[4,4] + M[5,5] + 12 \times 20 \times 27\}$

$= \min_{4\le k<5} \{0+0+1680\}$

$= 1680$

$M[1,3] = \min_{1\le k<3}\{M[1,1] + M[2,3] + P_0P_1P_2 , M[1,2] + M[3,3] + P_0P_2P_3 \}$

$= \min_{1\le k<3}\{ 0+360+480 , 120+0+144\}$

$= \min_{1\le k<3}\{ 840 , 264\}$

$= 264$

$M[2,4] = \min_{2\le k<4}\{M[2,2] + M[3,4] + P_1P_2P_4 , M[2,3] + M[4,4] + P_1P_3P_4 \}$

$= \min_{2\le k<4}\{ 0+720+600 , 360+0+2400\}$

$= \min_{1\le k<3}\{ 1320 , 2760\}$

$= 1320$

And so on.

now I can stop the table as shown above and from that we can see that we can multiply a1 a5 in as few as 1344 multiplication operations.

in order to put a multiplication brackets we must focus on the selected k values.

when k = 2

$M[1,5] = M[1,2] + M[3,5] + P_0P_2P_5 = 1344$

So $(A_1 \times A_2)(A_3 \times A_4 \times A_5)$

Now we can make this more simple by considering M[3,5],

when k=4

$M[3,5] = M[3,4] + M[5,5] + P_2P_4P_5 = 1140$

So $(A_1 \times A_2)((A_3 \times A_4) \times A_5)$

Cost: $(120)(720 \times A_5)$

$120 + 720 + 420 + 84 = 1344$

Therefore, these parenthesis are the optimal way to have the fewest number of multiplication operations.

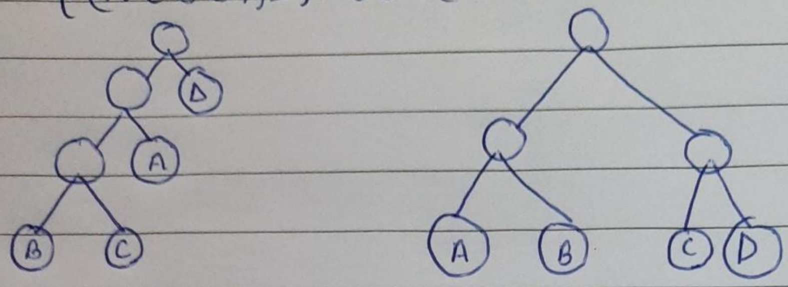
Algorithm For Finding Order Of Matrix Chain Multiplication

- /*
  - P is array storing details about the dimension of matrices. n is the number of elements in P.
  - Let P = [2,3,4,5], then the given matrices are of order {2,3},{3,4},{4,5}. And, n=4,
- */
- MatrixChainMultiplicationOrder(P[ ],n)
- {
  - Declare a matrix of order nxn to store costs - M[n][n];
  - Declare another matrix to store bracket information - Bracket[n][n]; Initialize the diagonal elements of sub-matrix M[1…n][1…n] to 0;
  - for every len belonging to [2,n]
  - {
    - for every i E [i + n-len+1]
    - {
      - initialize j = i + len-1;
      - maximize M[i][j] to greatest integer;
      - For every k E [i,j-1]
      - {
        - declare cost = M[i][k] + M[k+1][j] + P[i-1] * P[k] * P[j];
        - if (cost < M[i][j])
- {
- M[i][j]=cost; //Minimum cost
- Bracket[i][j]=k //Stores where to split for min cost
- }
      - }
    - }
  - }
  - print "Optimal costs is "<< M[1][n-1];
- }
- Time Complexity is $O(n^3)$.

Algorithm For Printing Parenthesis

```
/*
        i - The initial index of bracket matrix to be considered.  j -
        The final index of bracket matrix to be considered.  n - The
        total number of matrices in matrix chain.
        Bracket[n][n] - The bracket matrix storing information.
        Mname - The name of matrices in chain to identify them(starts from A).

        print parenthesization in subexpression(i,j)
*/
PrintParentesis(i , j , bracket[n][n] , Mname)
{
        if (i==j)           //only one matrix is left in current segment
        {
                print Mname;
                increment Mname;
                return;
        }
        else
        {
                print ”(“;

                //Recursion put brackets around subexpression.
                //From i to Bracket[i][j].

                PrintPrenthesis( i , Bracket[i][j] , Bracket , Mname);

                //Recursive put brackets around subexpression.
                //From Bracket[i][j] + 1 to j.

                PrintPrenthesis( Bracket[i][j] + 1 , j , Bracket , Mname);
                print “)”;
        }
}
```

Consider 4 matrices A, B, C, D then they can be grouped as
((A (B C)) D) or (A B) (C D)



Eg. $A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$ $B = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$
(2 x 3) (3 x 4)

So $A.B = \begin{bmatrix} m & n & o & p \\ q & r & s & t \end{bmatrix}$
(2 x 4)

So cost = 2 x 4 x 3
⇒ If 2 matrix of order (m x n) and (n x p) are multiplied then the cost is (m x n x p).
and if we are multiplying further cost keeps getting added.

A	B	C	D

$$2 \times \boxed{3 \quad 3} \times \boxed{4 \quad 4} \times \boxed{5 \quad 5} \times 6$$

$d_0 \quad d_1 \quad d_2 \quad d_3 \quad d_4$

(A B) ¦ (C D)
⇒ we are moving from (i, k) (k+1, j)

Code

```
#include<iostream>
 #include<conio.h>
#define INT_MAX 999999999
using namespace std;
void PrintParenthesis(int i, int j, int n, int *bracket, char &Mname)
{
	if(i==j)
	{
		cout<<Mname++;
		return;
	}
	cout<<"(";
	 PrintParenthesis(i,*((bracket+i*n)+j),n,bracket,Mname);
	PrintParenthesis(*((bracket+i*n)+j)+1,j,n,bracket,Mname);
	cout<<")";
}
void MCMO(int p[], int n)
{
	int M[n][n];
	int bracket[n][n];
	for(int i=1;i<n;i++)
		M[i][i]=0;
	for(int len=2;len<n;len++)
	    for(int i=1;i<n-len+1;i++)
		{
			int j=i+len-1;
			M[i][j]=INT_MAX;
			for(int k=i;k<=j-1;k++)
			{
				int cost=M[i][k]+M[k+1][j]+p[i-1]*p[k]*p[j];
				if(cost<M[i][j])
				{
					M[i][j]=cost;
					bracket[i][j]=k;
				}
			}
```

```
		}
	cout<<"\nThe Cost-Matrix is given by:\n";
	for(int i=1;i<n;i++)
	{
		for(int j=1;j<n;j++)
		{
			if(i>j)
				cout<<"X\t";
			else
				cout<<M[i][j]<<"\t";
		}
		cout<<endl;
	}
	cout<<"\nOptimal Cost:"<<M[1][n-1]<<endl;
	cout<<"Optimal Parenthesization:";
	char Mname='A';
	PrintParenthesis(1,n-1,n,(int *)bracket,Mname);
}
int main()
{
	int n;
	cout<<"Enter number of matrices:";
	cin>>n;
	int arr[n+1];
	cout<<"Enter dimensions of the matrices:"<<endl;
	for(int i=0;i<=n;i++)
		cin>>arr[i];
	MCMO(arr,n+1);
	getch();
	return 0;
}
```

Output

```
Enter number of matrices:10
Enter dimensions of the matrices:
12
30
32
31
14
23
45
24
8
12
10

The Cost-Matrix is given by:
0       11520   23424   28632   32496   44916   57876   41464   42616   43384
X       0       29760   27328   36988   60718   67018   38584   41464   41944
X       X       0       13888   24192   48538   54250   30904   33976   34424
X       X       X       0       9982    34020   40026   22968   25944   25916
X       X       X       X       0       14490   29610   19496   20840   21576
X       X       X       X       X       0       24840   16920   19128   19720
X       X       X       X       X       X       0       8640    12960   13200
X       X       X       X       X       X       X       0       2304    2880
X       X       X       X       X       X       X       X       0       960
X       X       X       X       X       X       X       X       X       0

Optimal Cost:43384
Optimal Parenthesization:((A(B(C(D(E(F(GH)))))))(IJ))
```

```
Enter number of matrices:3
Enter dimensions of the matrices:
10 20 30 40

The Cost-Matrix is given by:
0       6000    18000
X       0       24000
X       X       0

Optimal Cost:18000
Optimal Parenthesization:((AB)C)
```

```
Enter number of matrices:5
Enter dimensions of the matrices:
2 4 5 7 8 9

The Cost-Matrix is given by:
0       40      110     222     366
X       0       140     364     652
X       X       0       280     640
X       X       X       0       504
X       X       X       X       0

Optimal Cost:366
Optimal Parenthesization:((((AB)C)D)E)
```

The HU And Shing Algorithm

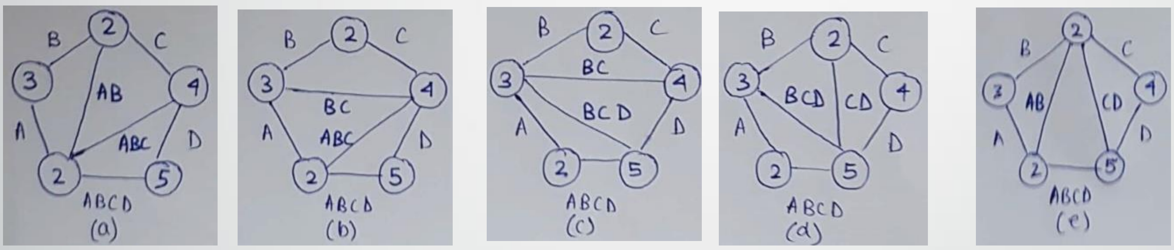
This algorithm was published in 1981 by T.C.Hu and M.T.Shing. It is very efficient algorithm with a time complexity of only **O(n.log n).** But the logic as well as the algorithm itself is very complicated.

In this algorithm they showed how the matrix chain multiplication can be transformed into the problem of triangulation of regular polygon, where, the bottom side, called base, represents final result.

Their Lemma 1 states that any order of multiplying (n-1) matrices corresponds to a partition of an n-gon.

For example, there are five sides: A,B,C,D and final result ABCD.

A is 2x3, B is 3x2, C is 2x4, and D is 4x5.



(a) ((AB)C)D = 2x3x2 + 2x2x4 + 2x4x5 = 68 multiplications
(b) (A(BC))D = 3x2x4 + 2x3x4 + 2x4x5 = 88 multiplications
(c) A((BC)D) = 3x2x4 + 5x3x4 + 2x3x5 = 114 multiplications
(d) A(B(CD)) = 2x4x5 + 2x3x5 + 2x3x5 = 100 multiplications
(e) (AB)(CD) = 2x3x2 + 2x4x5 + 2x2x5 = 72 multiplications

Thus, the minimum cost is obtained for case(a).

Hence, optimal parenthesization is ((AB)C)D.

Conclusion

Using dynamic programming, we can minimize the total number of multiplication (costs) while multiplying a chain of matrices, thus making the calculation time shorter, and program efficient.

Bibliography

1) Introduction to Algorithm - CLRS
2) https://www.sciencedirect.com/science/article/pii/0022247X7890166X .
3) https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/
4) https://www.thoughtco.com/principle-of-optimality-definition-1147078