Name	**Advait Ravi Sapkal**
UID No.	**2021700055**
Experiment No.	**5**

AIM:	To implement the solution to the Matrix Chain Multiplication problem.
Program 1	
PROBLEM STATEMENT:	Given a chain A1; A2; A3; … An of n matrices, where for i = 1; 2; … n, matrix Ai has dimensions pi-1 X pi, fully parenthesize the product A1*A2*A3 … *An in a way that minimizes the number of scalar multiplications. Also, show the parenthesizing used.
ALGORITHM:	

Matrix Chain Multiplication Problem

Idea:

1. Characterize:

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

2. Recurse:

This recursive equation assumes that we know the value of k, which we do not. There are only $j - i$ possible values for k, however, namely $k = i, i+1, \ldots, j-1$. Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \,. \end{cases} \qquad (15.7)$$

The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i,j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i,j]$ equals a value k such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

Algorithm:

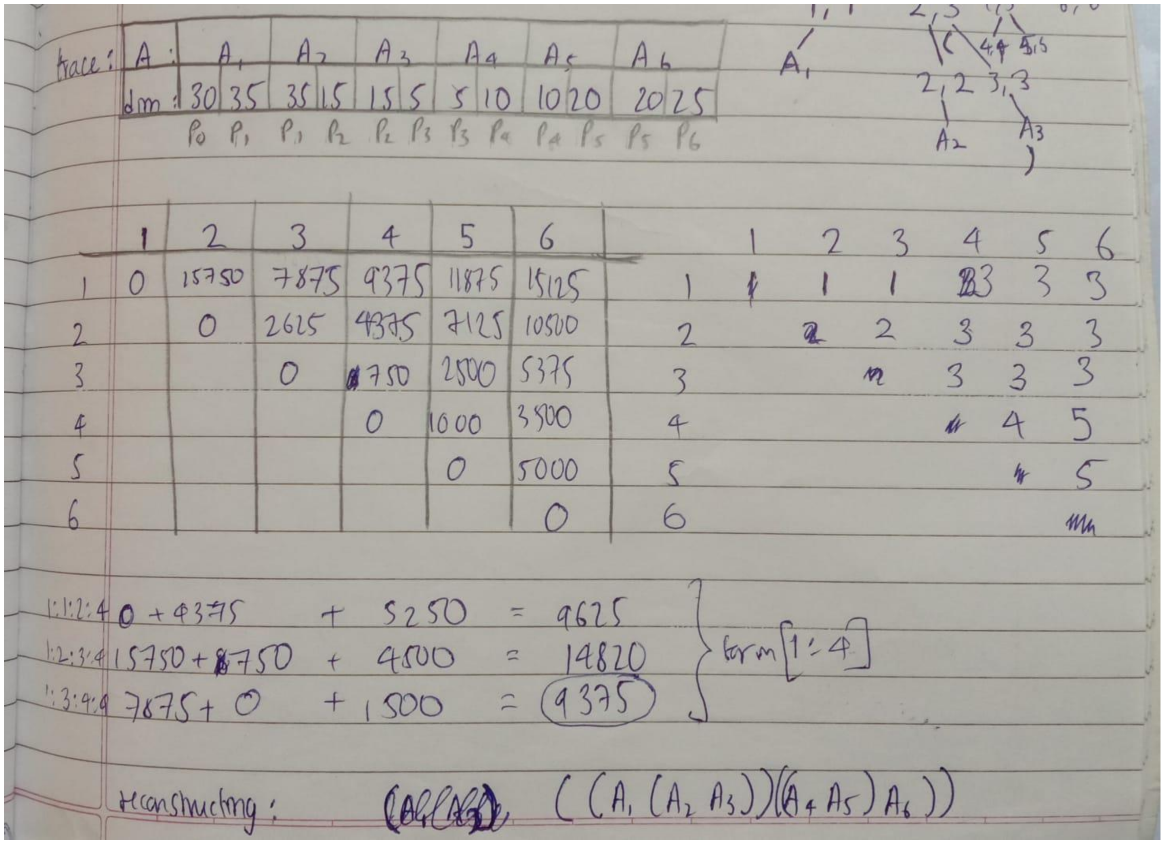
3. **Compute:**

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1 . . n, 1 . . n] and s[1 . . n − 1, 2 . . n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            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 m and s
```

4. **Construct:**

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Trace:



trace:

A :	A_1		A_2		A_3		A_4		A_5		A_6	
dim :	30	35	35	15	15	5	5	10	10	20	20	25
	P_0	P_1	P_1	P_2	P_2	P_3	P_3	P_4	P_4	P_5	P_5	P_6

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

1:1:2:4 0 + 4375 + 5250 = 9625

1:2:3:4 15750 + 8750 + 4500 = 14820

1:3:4:4 7875 + 0 + 1500 = 9375

for m[1:4]

reconstructing : $((A_1(A_2A_3))((A_4A_5)A_6))$

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<limits.h>

//utils
void print_matrix(int n,int mat[n][n]){
    for(int i=1;i<n;i++){
        for(int j=1;j<n;j++){
            if(i>j)printf("         ");
            else printf(" %7d ",*(*(mat+i)+j));
        }
        printf(" \n");
    }

}

//constructing the solution
void print_optimal_parentheses(int n,int s[n][n],int i,int j){
```

	`` if(i==j){ printf("A%d",i); }else{ printf("("); print_optimal_parentheses(n,s,i,s[i][j]); print_optimal_parentheses(n,s,s[i][j]+1,j); printf(")"); } } //computing the solution void matrix_chain_multiplication(int dimensions[],int n){ //initialization int m[n+1][n+1]; int s[n+1][n+1]; for(int i=0;i<n+1;i++){ m[i][i]=0; s[i][i]=0; } //diagonal traversal int len=2; while(len<n+1){ int second=len; int first=1; while(second<n+1){ int min=INT_MAX; int mink=0; int k=first; while(k<second){ int temp_cost= m[first][k] + m[k+1][second] + dimensions[first- 1]*dimensions[k]*dimensions[second]; if(temp_cost<min){ min=temp_cost; mink=k; } k++; } m[first][second]=min; s[first][second]=mink; ```

```c
            first++;
            second++;
        }
        len++;
    }

    //printing results
    printf("\nthe cost matrix is: \n");
    print_matrix(n+1,m);
    printf("\nthe k matrix is: \n");
    print_matrix(n+1,s);

    printf("\noptimal cost is: %d\n",m[1][n]);

    printf("\noptimal parentheses is:\n\n");
    print_optimal_parentheses(n+1,s,1,n);
    printf(" \n");
}


int main(){
    int n;
    printf("enter number of matrices:\n");
    scanf("%d",&n);
    int dimensions[n+1];
    printf("enter dimension array:\n");
    for(int i=0;i<n+1;i++)scanf("%d",&dimensions[i]);
    matrix_chain_multiplication(dimensions,n);
}
```

Analysis:

- We are filling the upper half of an nxn matrix, where n is the number of matrices. For each place we are varying k from i to j-1 to check all possibilities. The cost can be said to be:

$$\sum_{l=2}^{l=n} \sum_{j=l}^{j=6} \sum_{i=1}^{i=n-l+1} (j-i)*c$$

which when evaluated comes out to be an expression in n of degree 3.
∴ the time complexity is $\Omega(n^3) = \Theta(n^3) = O(n^3)$

- the extra space required is the two tables of size nxn ∴ total auxillary space = $O(n^2) = \Theta(n^2) = \Omega(n^2)$

RESULT:

Output:

```
d:\DAA\matrix_chain>cd "d:\DAA\matrix_chain\" && gcc mcm.c
enter number of matrices:
6
enter dimension array:
30 35 15 5 10 20 25

the cost matrix is:
       0    15750     7875     9375    11875    15125
                0     2625     4375     7125    10500
                         0      750     2500     5375
                                  0     1000     3500
                                           0     5000
                                                    0

the k matrix is:
       0        1        1        3        3        3
                0        2        3        3        3
                         0        3        3        3
                                  0        4        5
                                           0        5
                                                    0

optimal cost is: 15125

optimal parentheses is:

((A1(A2A3))((A4A5)A6))

d:\DAA\matrix_chain>
```

Conclusions:

1. The cost of multiplying n matrices without this dp optimization is $O(2^n)$ which is highly improved to $O(n^3)$ after dp optimization.
2. The cost without any recursion/dp would have been even higher (the matrices are multiplied linearly as it is)
3. The extra space needed is $O(n^2)$ which is okay.
4. The time and space complexities are tight, and they don't change for any best or worst case. $O(n^3)$, $\Theta(n^3)$, $\Omega(n^3)$. Similarly, we have for space complexity: $O(n^2)$, $\Theta(n^2)$, $\Omega(n^2)$.
5. The algorithm is highly useful and has many applications wherever multiple matrix multiplications are concerned, such as AI, physics, etc.