Experiment No.	4
Aim	To implement dynamic algorithms
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

Theory:
Matrix chain multiplication problem is a classic problem in computer science and mathematics that involves finding the most efficient way to multiply a chain of matrices. Given a sequence of matrices with different dimensions, the goal is to determine the order of multiplication that minimizes the total number of scalar multiplications required.

The problem is usually defined as follows: Given a chain of matrices A1, A2, A3, ..., An, where the dimensions of matrix Ai are given as d[i-1] x d[i] for i = 1, 2, ..., n, the task is to find the order of multiplication that minimizes the total number of scalar multiplications needed to multiply the entire chain of matrices.

The matrix chain multiplication problem has an optimal substructure property, which allows for the use of dynamic programming to solve it efficiently. One common approach is to use a dynamic programming algorithm, such as the matrix chain multiplication algorithm, also known as the dynamic programming-based approach, which has a time complexity of O(n^3), where n is the number of matrices in the chain.

The matrix chain multiplication problem has applications in various fields such as computer graphics, optimization, and scientific computing, where efficient matrix multiplication is required for large-scale computations.

Dynamic Programming Approach:-

Step 1: The structure of an optimal parenthesization

For our first step in the dynamic-programming paradigm, we find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. In the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

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 $A_{k+1} A_{k+2} \cdots A_j$.

Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. We have seen that any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.

Step 2: A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \; .$$

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

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

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 $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Step 3: Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (15.7) to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 \cdots A_n$. As we saw for the rod-cutting problem, and as we shall see in Section 15.3, this recursive algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.

Observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence (15.7) recursively, we compute the optimal cost by using a tabular, bottom-up approach. (We present the corresponding top-down approach using memoization in Section 15.3.)

We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$. The procedure uses an auxiliary table $m[1 \mathinner{..} n, 1 \mathinner{..} n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1 \mathinner{..} n-1, 2 \mathinner{..} n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We shall use the table s to construct an optimal solution.

In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i, j]$. Equation (15.7) shows that the cost $m[i, j]$ of computing a matrix-chain product of $j-i+1$ matrices depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices. That is, for $k = i, i+1, \ldots, j-1$, the matrix $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. For the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain.

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1 \mathinner{..} n-1, 2 \mathinner{..} n]$ gives us the information we need to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}.

Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$. We can determine the earlier matrix multiplications recursively, since $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1,n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$. The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$.

Algorithm:

MATRIX-CHAIN-ORDER(p)

```
1. n = p.length - 1
2. let m[1…n,1…n] and s[1…n-1,2…n] be new tables
3. for i = 1 to n
4.      m[i,i] = 0
5. for len = 2 to n // len is the chain length
6.      for i = 1 to n - len + 1
7.           j = i + len - 1
8.           m[i,j] = infinity
9.           for k = i to j - 1
10.               q = m[I,k] + m[k+1, j] + $p_{i-1}.p_k.p_j$
11.               if q < m[i,j]
12.                    m[i,j] = q
13.                    s[i,j] = k
14 return m and s
```

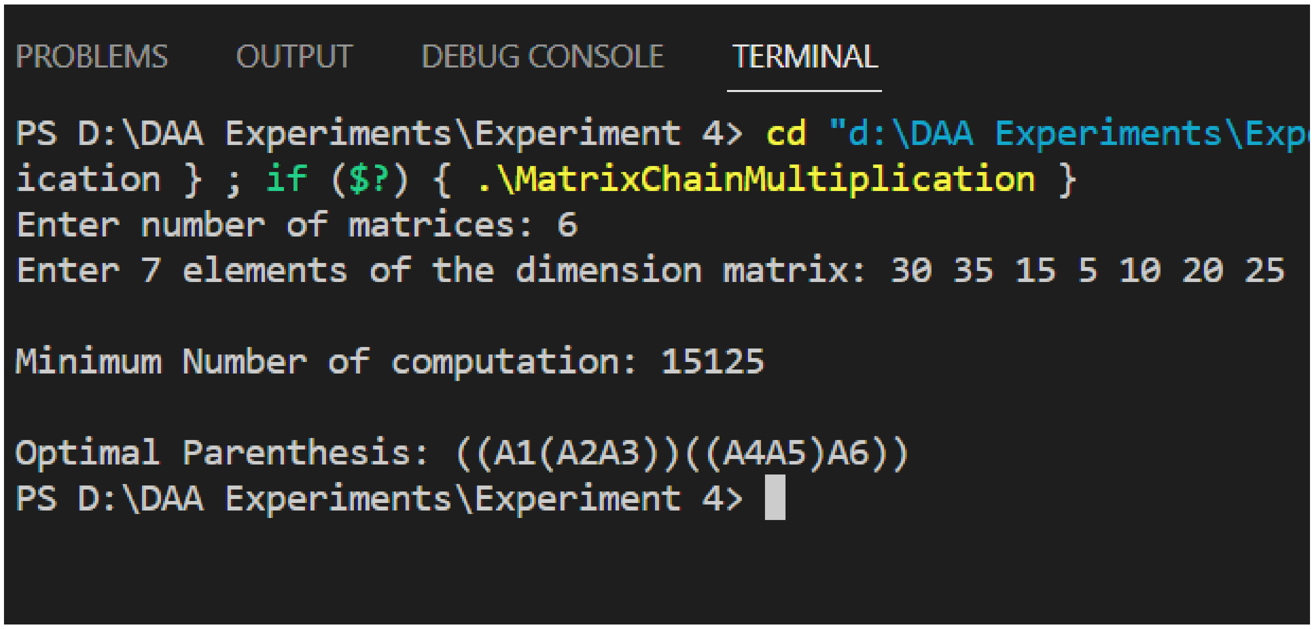
PRINT-OPTIMAL-PARENS(s, i, j)

```
1. if i == j
2.      print "A"$_i$
3. else print "("
4.      PRINT-OPTIMAL-PARENS(s, i, s[i,j])
5.      PRINT-OPTIMAL-PARENS(s, s[i,j]+1, j)
6.      print ")"
```

Program:

```c
#include <stdio.h>
#define MAX 100

int m[MAX][MAX], s[MAX][MAX];

void printOptimalParenthesis(int i, int j) {
    if (i == j)
        printf("A%d", i);
    else {
        printf("(");
        printOptimalParenthesis(i, s[i][j]);
        printOptimalParenthesis(s[i][j] + 1, j);
        printf(")");
    }
}

int main() {
    int n = 4, i, j, k, len, q;
    printf("Enter number of matrices: ");
    scanf("%d",&n);

    int p[n+1];

    printf("Enter %d elements of the dimension matrix: ",(n+1));
    for(i=0;i<n+1;i++){
        scanf("%d",&p[i]);
    }

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

    for (len = 2; len <= n; len++) {
        for (i = 1; i <= n - len + 1; i++) {
            j = i + len - 1;
            m[i][j] = 99999999;
            for (k = i; k < j; 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;
                }
            }
        }
    }
    printf("Optimal Parenthesis: ");
    printOptimalParenthesis(1, n);

    return 0;
}
```

Results:



```
PS D:\DAA Experiments\Experiment 4> cd "d:\DAA Experiments\Exp
ication } ; if ($?) { .\MatrixChainMultiplication }
Enter number of matrices: 6
Enter 7 elements of the dimension matrix: 30 35 15 5 10 20 25

Minimum Number of computation: 15125

Optimal Parenthesis: ((A1(A2A3))((A4A5)A6))
PS D:\DAA Experiments\Experiment 4> 
```

Conclusion:

I have understood the dynamic programming approach towards solving the matrix chain multiplication problem and the four main steps involved: (i) Characterize the structure of an optimal solution, (ii) Recursively define the value of an optimal solution, (iii) Compute the value of an optimal solution, typically in a bottom-up fashion and (iv) Construct an optimal solution from computed information.