Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Experiment No. 4

Name: Om Bhutki

UID: 2022301003

Subject: DAA

Experiment No: 4

Aim – Experiment to implement matrix chain multiplication.

Objective: To understand dynamic programming approach

Theory:

Matrix chain multiplication (or the **matrix chain ordering problem**[1]) is an optimization problem concerning the most efficient way to multiply a given sequence of matrices. The problem is not actually to *perform* the multiplications, but merely to decide the sequence of the matrix multiplications involved. The problem may be solved using dynamic programming. There are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices *A*, *B*, *C*, and *D*, there are five possible options:

$((AB)C)D = (A(BC))D = (AB)(CD) = A((BC)D) = A(B(CD))$.

The idea is to break the problem into a set of related subproblems that group the given matrix to yield the lowest total cost.

Following is the recursive algorithm to find the minimum cost:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the price of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices ABCD, we compute the cost required to find each of (A)(BCD), (AB)(CD), and (ABC)(D), making recursive calls to find the minimum cost to compute ABC, AB, CD, and BCD and then choose the best one. Better still, this yields the minimum cost and demonstrates the best way of doing the multiplication

---

Algorithm:

MATRIX-CHAIN-ORDER (p)

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n    // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

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

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

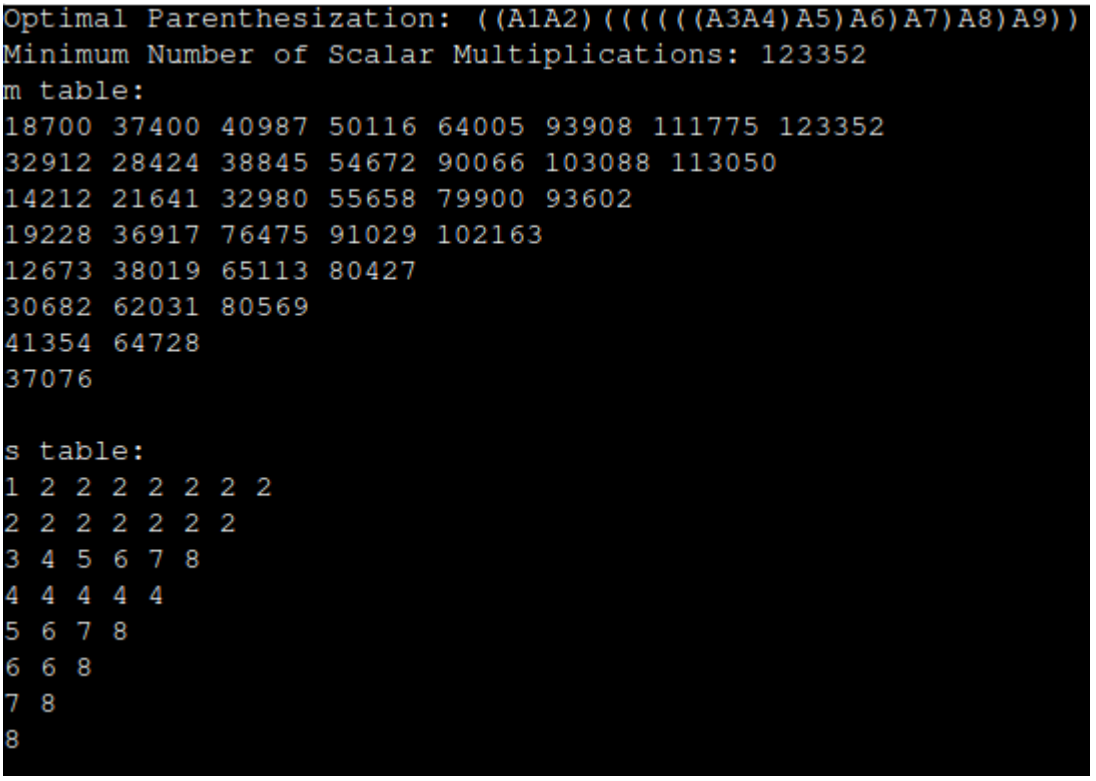
Program:

```
#include <iostream>

#include <climits>

#include <random>

#include <ctime>

using namespace std;

void matrixChainOrder(int p[], int n, int m[][100], int s[][100]) {

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] = INT_MAX;

 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;

    }

 }

  }

 }

}



void printOptimalParenthesis(int s[][100], int i, int j) {

 if(i == j)

    cout << "A" << i;
```

```
    else {
        cout << "(";
        printOptimalParenthesis(s, i, s[i][j]);
        printOptimalParenthesis(s, s[i][j]+1, j);
        cout << ")";
    }
}
int main() {
    int p[10];
    srand ( time(NULL) );
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> distr(15, 46);
    for(int i=0; i<10; ++i)
        p[i] = distr(gen);
    int n = sizeof(p)/sizeof(p[0]) - 1;
    generateMatrices(p);
    int m[100][100];
    int s[100][100];
    matrixChainOrder(p, n, m, s);
    cout << "Optimal Parenthesization: ";
    printOptimalParenthesis(s, 1, n);
    cout << endl;
    cout << "Minimum Number of Scalar Multiplications: " << m[1][n] << endl;
    cout << "m table:";
    for(int a = 0; a < 10; a++)
    {
        for(int b = 0; b < 10; b++)
        {
            if(m[a][b] == 0){continue;}
            cout << m[a][b] << " ";
```

```
    }
    cout << endl;
  }

  cout << "s table:";
      for(int a = 0; a < 10; a++)
  {
    for(int b = 0; b < 10; b++)
    {
      if(s[a][b] == 0){continue;}
      cout << s[a][b] << " ";
    }
    cout << endl;
  }
      return 0;
}
```

Output with random p values:



```
Optimal Parenthesization: (A1(A2(A3(((A4A5)A6)(A7(A8A9))))))
Minimum Number of Scalar Multiplications: 215446
m table:
22475 42224 70824 113984 131924 159874 209324 215446
23374 56550 99294 115778 148200 201890 198771
35464 78000 93756 128414 182260 178094
45760 72800 101868 153296 159556
45760 94952 149812 135608
44720 99268 95128
51428 71208
45494

s table:
1 1 3 3 3 6 7 1
2 3 3 3 6 6 2
3 3 3 6 6 3
4 5 6 7 6
5 6 6 5
6 6 6
7 7
8
```

Inference: Hence, after performing the experiment I have observed that the order of matrix while multiplication is critical while multiplying the matrices. Determining the optimal way to parenthesize the matrices will significantly decrease the number of scalar multiplications needed to obtain the final result. For example

If A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then

computing $(AB)C$ needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while computing $A(BC)$ needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Therefore I found out that finding an optimal way to multiply will significantly reduce running times especially with large matrix values.

Conclusion: Thus, by performing this experiment I understood the significance of matrix chain multiplication and was also able to implement it.