Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	4
Aim	To understand and implement Chain MatrixMultiplication.
Name	CHINMAY PRASHANT JADHAV
UID No.	2021300046
Class & Division	COMPS A(C)

THEORY:

Chain matrix multiplication is a problem in linear algebra that involves multiplying multiple matrices together. The order in which the matrices are multiplied affects the number of scalar multiplications required to compute the result, and therefore affects the efficiency of the algorithm.

A common strategy for optimizing the computation of matrix products is to use dynamic programming. The idea is to break down the problem into smaller subproblems, and then reuse the solutions to those subproblems in order to compute the final result. In the case of chain matrix multiplication, this involves computing the optimal order in which to multiply the matrices.

Let's say we have a sequence of matrices A1, A2, ..., An. We can define the problem of computing their product as follows:

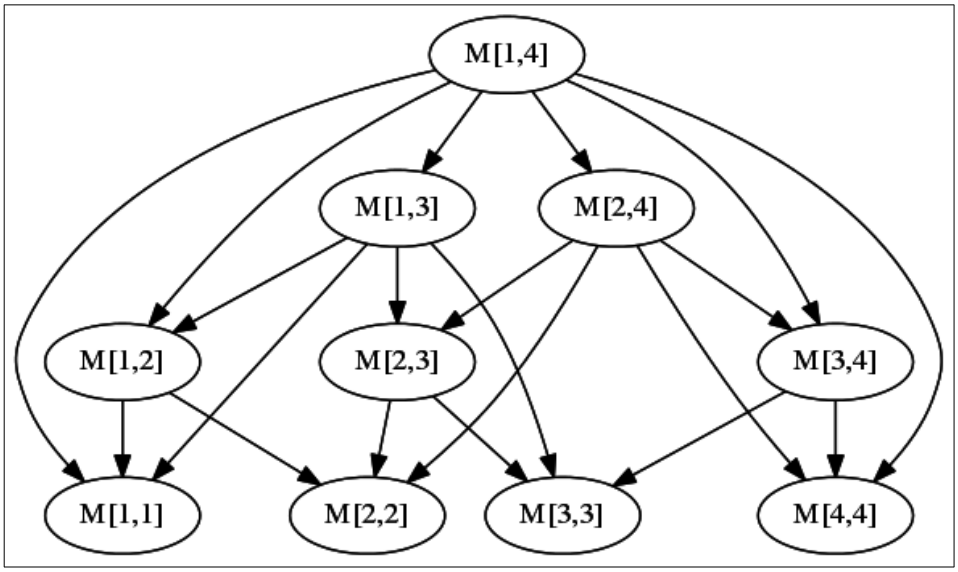
- Find the optimal way to parenthesize the product A1A2...An such that the total number of scalar multiplications is minimized.

For example, if we have matrices A, B, and C, we can multiply them in two different orders:

- (AB)C, which requires 2 scalar multiplications to compute AB and then 2 more to compute the final product.
- A(BC), which requires 2 scalar multiplications to compute BC and then 2 more to compute the final product.

The first order is therefore more efficient.

We can solve this problem using dynamic programming by defining a table M where M[i,j] is the minimum number of scalar multiplications required to compute the product of matrices AiAi+1...Aj. We can then fill in the entries of the table in a bottom-up fashion, using the following recurrence:

	M[i,j] = min(M[i,k] + M[k+1,j] + pi-1pkpj) where k is a value between i and j-1, and pi-1pkpj is the number of scalar multiplications required to compute the product AiAi+1...Ak and Ak+1Ak+2...Aj. The optimal solution can then be found by tracing back through the table M. This algorithm has a time complexity of O(n^3) and a space complexity of O(n^2), where n is the number of matrices in the sequence. 

CODE:

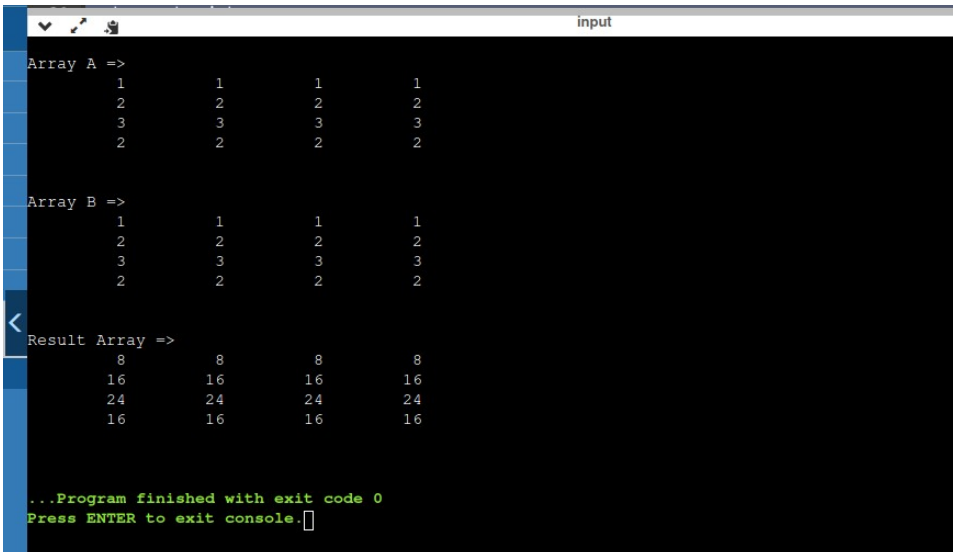
```
#include <bits/stdc++.h>
using namespace std;

#define ROW_1 4
#define COL_1 4

#define ROW_2 4
#define COL_2 4

void print(string display, vector<vector<int> > matrix,
int start_row, int start_column, int end_row,
int end_column)
{
cout << endl << display << " =>" << endl;
for (int i = start_row; i <= end_row; i++) {
for (int j = start_column; j <= end_column; j++) {
cout << setw(10);
```

```
cout << matrix[i][j];
}
cout << endl;
}
cout << endl;
return;
}

vector<vector<int> >
add_matrix(vector<vector<int> > matrix_A,
vector<vector<int> > matrix_B, int split_index,
int multiplier = 1)
{
for (auto i = 0; i < split_index; i++)
for (auto j = 0; j < split_index; j++)
matrix_A[i][j]
= matrix_A[i][j]
+ (multiplier * matrix_B[i][j]);
return matrix_A;
}

vector<vector<int> >
multiply_matrix(vector<vector<int> > matrix_A,
vector<vector<int> > matrix_B)
{
int col_1 = matrix_A[0].size();
int row_1 = matrix_A.size();
int col_2 = matrix_B[0].size();
int row_2 = matrix_B.size();

if (col_1 != row_2) {
cout << "\nError: The number of columns in Matrix "
"A must be equal to the number of rows in "
"Matrix B\n";
return {};
}

vector<int> result_matrix_row(col_2, 0);
vector<vector<int> > result_matrix(row_1,
result_matrix_row);

if (col_1 == 1)
result_matrix[0][0]
= matrix_A[0][0] * matrix_B[0][0];
else {
```

```
int split_index = col_1 / 2;

vector<int> row_vector(split_index, 0);

vector<vector<int> > a00(split_index, row_vector);
vector<vector<int> > a01(split_index, row_vector);
vector<vector<int> > a10(split_index, row_vector);
vector<vector<int> > a11(split_index, row_vector);
vector<vector<int> > b00(split_index, row_vector);
vector<vector<int> > b01(split_index, row_vector);
vector<vector<int> > b10(split_index, row_vector);
vector<vector<int> > b11(split_index, row_vector);

for (auto i = 0; i < split_index; i++)
for (auto j = 0; j < split_index; j++) {
a00[i][j] = matrix_A[i][j];
a01[i][j] = matrix_A[i][j + split_index];
a10[i][j] = matrix_A[split_index + i][j];
a11[i][j] = matrix_A[i + split_index]
[j + split_index];
b00[i][j] = matrix_B[i][j];
b01[i][j] = matrix_B[i][j + split_index];
b10[i][j] = matrix_B[split_index + i][j];
b11[i][j] = matrix_B[i + split_index]
[j + split_index];
}

vector<vector<int> > p(multiply_matrix(
a00, add_matrix(b01, b11, split_index, -1)));
vector<vector<int> > q(multiply_matrix(
add_matrix(a00, a01, split_index), b11));
vector<vector<int> > r(multiply_matrix(
add_matrix(a10, a11, split_index), b00));
vector<vector<int> > s(multiply_matrix(
a11, add_matrix(b10, b00, split_index, -1)));
vector<vector<int> > t(multiply_matrix(
add_matrix(a00, a11, split_index),
add_matrix(b00, b11, split_index)));
vector<vector<int> > u(multiply_matrix(
add_matrix(a01, a11, split_index, -1),
add_matrix(b10, b11, split_index)));
vector<vector<int> > v(multiply_matrix(
add_matrix(a00, a10, split_index, -1),
add_matrix(b00, b01, split_index)));
```

```
vector<vector<int> > result_matrix_00(add_matrix(
add_matrix(add_matrix(t, s, split_index), u,
split_index),
q, split_index, -1));
vector<vector<int> > result_matrix_01(
add_matrix(p, q, split_index));
vector<vector<int> > result_matrix_10(
add_matrix(r, s, split_index));
vector<vector<int> > result_matrix_11(add_matrix(
add_matrix(add_matrix(t, p, split_index), r,
split_index, -1),
v, split_index, -1));

for (auto i = 0; i < split_index; i++)
for (auto j = 0; j < split_index; j++) {
result_matrix[i][j]
= result_matrix_00[i][j];
result_matrix[i][j + split_index]
= result_matrix_01[i][j];
result_matrix[split_index + i][j]
= result_matrix_10[i][j];
result_matrix[i + split_index]
[j + split_index]
= result_matrix_11[i][j];
}

a00.clear();
a01.clear();
a10.clear();
a11.clear();
b00.clear();
b01.clear();
b10.clear();
b11.clear();
p.clear();
q.clear();
r.clear();
s.clear();
t.clear();
u.clear();
v.clear();
result_matrix_00.clear();
result_matrix_01.clear();
result_matrix_10.clear();
result_matrix_11.clear();
```

```
}
return result_matrix;
}

int main()
{
vector<vector<int> > matrix_A = { { 1, 1, 1, 1 },
{ 2, 2, 2, 2 },
{ 3, 3, 3, 3 },
{ 2, 2, 2, 2 } };

print("Array A", matrix_A, 0, 0, ROW_1 - 1, COL_1 - 1);

vector<vector<int> > matrix_B = { { 1, 1, 1, 1 },
{ 2, 2, 2, 2 },
{ 3, 3, 3, 3 },
{ 2, 2, 2, 2 } };

print("Array B", matrix_B, 0, 0, ROW_2 - 1, COL_2 - 1);

vector<vector<int> > result_matrix(
multiply_matrix(matrix_A, matrix_B));

print("Result Array", result_matrix, 0, 0, ROW_1 - 1,
COL_2 - 1);
}

// Time Complexity: T(N) = 7T(N/2) + O(N^2) => O(N^Log7)
// which is approximately O(N^2.8074) Code Contributed By:
// lucasletum
```

RESULT:	

CONCLUSION:	Chain matrix multiplication is an important problem in linear algebra that involves finding the optimal order in which to multiply a sequence of matrices in order to minimize the total number of scalar multiplications required to compute their product. This problem can be solved efficiently using dynamic programming by computing a table of minimum scalar multiplications required to compute all subproblems, and then using this table to compute the final result. The time complexity of the algorithm is O(n^3) and the space complexity is O(n^2), where n is the number of matrices in the sequence. Efficiently computing the optimal order of matrix multiplication is a fundamental problem in computer science and has many applications in areas such as computer graphics, scientific computing, and machine learning.