NAME:	Maureen Miranda
UID:	2022300060
SUBJECT	DAA
EXPERIMENT NO :	4
DATE OF PERFORMANCE	1/03/24
DATE OF SUBMISSION	10/03/24
AIM:	**Implementation of matrix chain multiplication using Dynamic**
OBJECTIVE	1. To understand the concept of dynamic programming 2. To understand the concept of matrix multiplication 3. To implement the matrix chain multiplication 4. To optimize the solution of matrix chain multiplication

THEORY:

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial .

Matrix Chain Multiplication

Matrix Chain Multiplication is an algorithm that is applied to determine the lowest cost way for multiplying matrices. The actual multiplication is done using the standard way of multiplying the matrices, i.e., it follows the basic rule that the number of rows in one matrix must be equal to the number of columns in another matrix. Hence, multiple scalar multiplications must be done to achieve the product. To brief it further, consider matrices A, B, C, and D, to be multiplied; hence, the multiplication is done using the standard matrix multiplication. There are multiple combinations of the matrices found while using the standard approach since matrix multiplication is associative.

Matrix Chain Multiplication Pseudocode

```
MATRIX-CHAIN-MULTIPLICATION(p)
n = p.length ─ 1

let m[1...n, 1...n] and s[1...n ─ 1, 2...n] be new matrices
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] + pi-1pkpj
if q < m[i, j]
m[i, j] = q
s[i, j] = k
return m and s
```

CODE:

```
import java.util.Scanner;

class Main {
static char name;
// Function for printing the optimal parenthesization of a matrix
chain product
static void printParenthesis(int i, int j, int n,
int[][] dp) {
// If only one matrix left
if (i == j) {
System.out.print(name++);
return;
}
System.out.print("(");
printParenthesis(i, dp[i][j], n, dp);
printParenthesis(dp[i][j] + 1, j, n, dp);
System.out.print(")");
}
```

```
static void matrixChainOrder(int p[], int n) {
int[][] m = new int[n][n];
int[][] dp = new int[n][n];
// cost is zero when multiplying one matrix.
for (int i = 1; i < n; i++)
m[i][i] = 0;
// L is chain length.
for (int L = 2; L < n; L++) {
for (int i = 1; i < n - L + 1; i++) {
int j = i + L - 1;
m[i][j] = Integer.MAX_VALUE;
for (int k = i; k <= j - 1; k++) {
// q = cost/scalar multiplications
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;
dp[i][j] = k;
}
}
}
}

System.out.println("\nOptimal Cost is : " + m[1][n - 1]);
// Print matrix chain table (m)
System.out.println("\nMatrix Chain Table (m):");
for (int i = 1; i < n; i++) {
for (int j = 1; j < n; j++) {
System.out.print(m[i][j] + "\t");
}
System.out.println();
}
// Print split point table (dp or k table)
System.out.println("\nK Table :");
for (int i = 1; i < n; i++) {
for (int j = 1; j < n; j++) {
System.out.print(dp[i][j] + "\t");
}
System.out.println();
}
// The first matrix is printed as 'A', next as 'B'
name = 'A';
System.out.print("\nOptimal Parenthesization is : ");
printParenthesis(1, n - 1, n, dp);
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Enter the number of matrices : ");
int n = sc.nextInt();
int arr[] = new int[n + 1]; // +1 because dimensions start from 0
to n
System.out.println("Enter the dimensions of each matrix :");
for (int i = 0; i <= n; i++) {
arr[i] = sc.nextInt();
}
matrixChainOrder(arr, n + 1); // +1 because dimensions start from
```

0 to n
System.out.println();
sc.close();
}
}

Output

```
Enter the number of matrices : 6
Enter the dimensions of each matrix :
30 35 15 5 10 20 25

Optimal Cost is : 15125

Matrix Chain Table (m):
0       15750   7875    9375    11875   15125
0       0       2625    4375    7125    10500
0       0       0       750     2500    5375
0       0       0       0       1000    3500
0       0       0       0       0       5000
0       0       0       0       0       0

K Table :
0       1       1       3       3       3
0       0       2       3       3       3
0       0       0       3       3       3
0       0       0       0       4       5
0       0       0       0       0       5
0       0       0       0       0       0

Optimal Parenthesization is : ((A(BC))((DE)F))
```

SOLVING:

Maureen Miranda

formula:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & i < j \end{cases}$$

$P = \{30, 35, 15, 5, 10, 20, 25\}$

Matrix Chain table

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

K Table

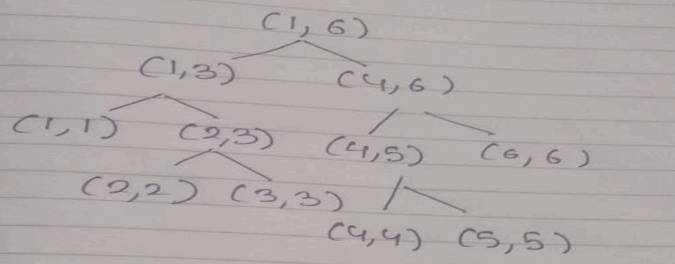
	1	2	3	4	5	6
1	0	1	1	3	3	3
2	0	0	2	3	3	3
3	0	0	0	3	3	3
4	0	0	0	0	4	5
5	0	0	0	0	0	5
6	0	0	0	0	0	0

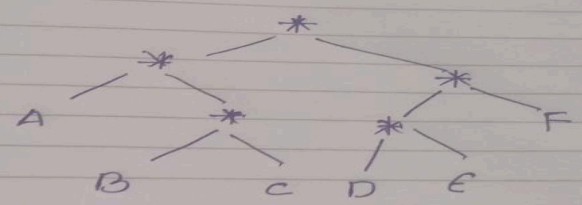
$m[2,3]$ ($k=2$) $= m[2,2] + m[3,3] + p_1 p_2 p_3$

$= 0 + 0 + 35 \times 15 \times 5$

$= 2625$ $\quad \therefore k = 2$

Similarly for the rest.





$(A \cdot (B \cdot C)) \cdot ((D \cdot E) \cdot F)$

optimal cost = 15125

RESULT:

1. Best Case : O(n^3)
• In the best case, the time complexity of the dynamic programming approach for matrix chain multiplication is O(n^3), where n is the number of matrices. This occurs when the input matrices are already in the optimal order, and the algorithm only needs to fill the dp Table.

2. Average Case : O(n^3)
• The average case time complexity of the dynamic programming approach for matrix chain multiplication is also O(n^3). This is because, on average, the algorithm will need to consider all subproblems and fill the dp table, leading to cubic time complexity.

3. Worst Case : O(n^3)
• The worst case time complexity of the dynamic programming approach for matrix chain multiplication is O(n^3). This occurs when the input matrices are in the worst possible order, and the algorithm needs to consider all subproblems and fill the dp table, resulting in the highest time complexity.

CONCLUSION:

Dynamic programming is employed in matrix chain multiplication to achieve a time complexity of O(n^3), where 'n' represents the number of matrices in the chain. This efficiency stems from the algorithm's capability to decompose the problem into smaller subproblems and retain their solutions, reducing redundant computations. Through iterative population of cost and split point matrices, the algorithm identifies the optimal sequence for matrix multiplication. This method guarantees a manageable overall time complexity, rendering dynamic programming a potent approach for addressing intricate optimization tasks such as matrix chain multiplication.