Bhartiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India

(Autonomous College Affiliated to University of Mumbai)

NAME:	SANIYA BANGARE
UID:	2021300009
SUBJECT	DAA
EXPERIMENT NO:	4
DATE OF PERFORMANCE	5/03/2023
AIM:	To find the minimum matrix chain multiplications required.
THEORY:	Let we have "n" number of matrices A1, A2, A3 ……… An and dimensions are d0 x d1, d1 x d2, d2 x d3 ………… d_{n-1} x d_n (i.e Dimension of Matrix **A_i is d_{i-1} x d_i** Solving a chain of matrix that, Ai A_{i+1} A_{i+2} A_{i+3} ……. A_j = (A_i A_{i+1} A_{i+2} A_{i+3} ……. A_k) (A_{k+1} A_{k+2} ……. A_j) + d_{i-1} d_k d_j where **i <= k < j.** Here total i to j matrices, Matrix i to k and Matrix k+1 to j should be solved in recursive way and finally these two matrices multiplied and these dimensions d_{i-1} d_k d_j (number of multiplications needed) added. The variable k is changed i to j. M[i, j] indicates that if we split from matrix i to matrix j then minimum number of scalar multiplications required. M [i , j] = { 0 ; when i=j ; [means it is a single matrix . If there is only one matrix no need to multiply with any other. So 0 (zero) multiplications required.] = { min { M[i, k] + M[k+1, j] + d_{i-1} d_k d_j } where i <= k< j

Time Complexity

If there are n number of matrices we are creating a table contains [(n) (n+1)] / 2 cells that is in worst case total number of cells n*n = n^2 cells we need calculate = **O (n^2)**

For each one of entry we need find minimum number of multiplications taking worst (it happens at last cell in table) that is Table [1,4] which equals to **O (n)** time.

Finally **O (n^2) * O (n) = O (n^3)** is time complexity.

Space Complexity

We are creating a table of n x n so space complexity is **O (n^2).**

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] + pi-1 pk pj
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.

PROGRAM:

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

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

int MatrixChainMultiplication(int p[], int n)
{
    int m[n][n];
    int i, j, k, L, q;

    for (i = 1; i < n; i++)
        m[i][i] = 0;
    for (L = 2; L < n; L++)
    {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; 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;
        }
    }

    return m[1][n - 1]; // returning the final answer which is
M[1][n]
}

int main()
{
    int n, i;
    printf("Enter number of matrices\n");
    scanf("%d", &n);

    n++;

    int arr[n];
```

```
    printf("Enter dimensions \n");

    for (i = 0; i < n; i++)
    {
        printf("Enter d%d :: ", i);
        scanf("%d", &arr[i]);
    }

    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
MatrixChainMultiplication(arr, size));

    return 0;
}
```

RESULT:

```
Enter number of matrices
4
Enter dimensions
Enter d0 :: 12
Enter d1 :: 5
Enter d2 :: 2
Enter d3 :: 16
Enter d4 :: 20
Minimum number of multiplications is 1240
```

CONCLUSION:

After running the matrix chain multiplication code experiment, it can be observed that the algorithm effectively computes the optimal sequence of matrix multiplications in terms of minimizing the number of scalar multiplications required.
The experiment showed that the running time of the algorithm increases significantly as the number of matrices in the chain increases. This is because the number of possible ways to parenthesize the matrices increases exponentially with the

	number of matrices. However, despite the exponential growth in the number of possible parenthesizations, the algorithm can find the optimal solution in a reasonable amount of time, even for relatively large chains of matrices. This is because the dynamic programming approach used by the algorithm avoids redundant calculations and uses previously computed values to solve subproblems efficiently