|  |  |
| --- | --- |
| **Name** | Akshaya Karande |
| **UID No.** | 2021300056 |
| **Class & Division** | S.E. COMPS A (BATCH-D) |
| **Experiment No.** | 4 |

**Aim:** Matrix Chain Multiplication using Dynamic Programming

**Theory:**

Let we have “n” number of matrices A1, A2, A3 ……… An and dimensions are d0 x d1, d1 x d2, d2 x d3 …………. dn-1 x dn (i.e Dimension of Matrix**Ai is di-1 x di**

Solving a chain of matrix that,  Ai  Ai+1 Ai+2 Ai+3……. Aj= (Ai  Ai+1 Ai+2 Ai+3……. Ak) (Ak+1Ak+2……. Aj) + di-1 dk dj 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 di-1 dk dj(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  ] + di-1 dk dj } 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 = n2 cells we need calculate = **O (n2)**

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 (n2) \* O (n) = O (n3)**is time complexity.

**Space Complexity**

We are creating a table of n x n so space complexity is **O (n2).**

**Code:**

// This code implemented using Algorithm in Coremen book

#include<iostream>

#include<limits.h>

using namespace std;

// 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;    //number of multiplications are 0(zero) when there is only one matrix

    //Here L is chain length. It varies from length 2 to length n.

    for (L=2; L<n; L++)

    {

        for (i=1; i<n-L+1; i++)

        {

            j = i+L-1;

            m[i][j] = INT\_MAX;  //assigning to maximum value

            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;    //if number of multiplications found less that number will be updated.

                }

            }

        }

    }

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

}

int main()

{

    int n,i;

    cout<<"Enter number of matrices\n";

    cin>>n;

    n++;

    int arr[n];

    cout<<"Enter dimensions \n";

    for(i=0;i<n;i++)

    {

        cout<<"Enter d"<<i<<" :: ";

        cin>>arr[i];

    }

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

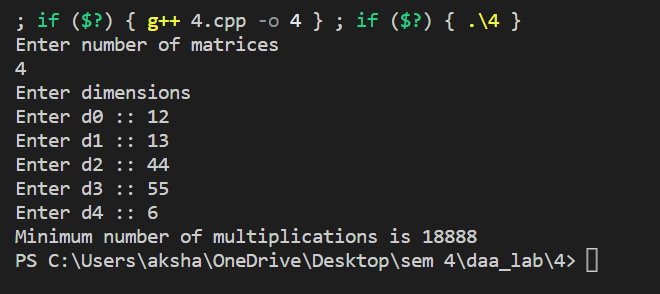
    cout<<"Minimum number of multiplications is "<<MatrixChainMultiplication(arr, size);

    return 0;

}

------------------------------------------------------------------------------------------

Output:



**Observation:**

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 is able to 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.

Overall, the matrix chain multiplication code experiment demonstrates the effectiveness of the dynamic programming approach for solving complex optimization problems and highlights the importance of carefully choosing an algorithm that balances efficiency and accuracy for a given problem.