**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: 13/02/2023**

**Experiment No. 11**

# Title: Matrix Chain Multiplication

**Theory/Description of the Problem Statement:**

Matrix Chain Multiplication is a problem in computer science that involves finding the most efficient way to multiply a chain of matrices. Given a sequence of matrices A1, A2, ..., An, where the dimensions of matrix Ai are p[i-1] x p[i] for i = 1, 2, ..., n, the goal is to compute the product of these matrices in a way that requires the fewest number of scalar multiplications.

The problem can be formulated as an optimization problem, where the goal is to find the optimal sequence of matrix multiplications that minimizes the total number of scalar multiplications required. This can be achieved using dynamic programming, by computing the optimal substructure of the problem and building up a table of subproblems to solve the original problem.

The optimal substructure of the problem is defined by the fact that the optimal parenthesization of a matrix chain can be split into two subchains, each of which is itself optimally parenthesized. Specifically, for any matrix chain Ai, Ai+1, ..., Aj, we can split it into two subchains, Ai, ..., Ak and Ak+1, ..., Aj, where k is between i and j-1. The optimal parenthesization of the entire matrix chain is then obtained by recursively computing the optimal parenthesizations of the two subchains and combining them in the most efficient way.

**Algorithm :**

* Create a function matrixChainMemoised with parameters p, i, and j
* If i is equal to j, return 0
* If dp[i][j] is not equal to -1, return dp[i][j]
* Set dp[i][j] to INT\_MAX
* Create a loop with variable k starting from i to j-1
* Within the loop, set dp[i][j] to minimum of dp[i][j], matrixChainMemoised(p, i, k) + matrixChainMemoised(p, k + 1, j) + p[i - 1] \* p[k] \* p[j]
* Return dp[i][j]
* Create a function MatrixChainOrder with parameters p and n
* Initialize i to 1 and j to n-1
* Return matrixChainMemoised(p, i, j)
* In the main function
* Create an integer array arr with elements {1, 2, 3, 4}
* Get the size of the array n
* Initialize dp with -1
* Call MatrixChainOrder function with parameters arr and n and print the output

**Pseudo Code :**

* function matrixChainMemoised(p, i, j, dp):
* if i == j:
* return 0
* if dp[i][j] != -1:
* return dp[i][j]
* dp[i][j] = INFINITY
* for k from i to j-1:
* q = matrixChainMemoised(p, i, k, dp)
* + matrixChainMemoised(p, k+1, j, dp)
* + p[i-1] \* p[k] \* p[j]
* dp[i][j] = min(dp[i][j], q)
* return dp[i][j]
* function MatrixChainOrder(p, n):
* initialize dp with -1
* return matrixChainMemoised(p, 1, n-1, dp)
* main:
* initialize arr with {1, 2, 3, 4}
* n = length of arr
* result = MatrixChainOrder(arr, n)
* print "Minimum number of multiplications is", result

**Analysis of the Algorithm**

The algorithm for solving the Matrix Chain Multiplication problem using dynamic programming involves filling in the entries of the table M in a bottom-up fashion. We start by filling in the diagonal entries of the table with zeros, since a single matrix requires no scalar multiplications. Then, we fill in the remaining entries of the table by computing the minimum number of scalar multiplications required to multiply the subchains of matrices that make up the matrix chain. The optimal solution to the problem is then given by M[1,n].

**Time Complexity:**

The time complexity of the algorithm is O(n^3), where n is the number of matrices.

This is because the algorithm uses nested loops to iterate through all possible values of k, resulting in a time complexity of O(n^3)

**Space Complexity:**

The space complexity of the algorithm is O(n^2), where n is the number of matrices.

This is due to the use of the dp array, which has a size of n x n.

The dp array is used to store the results of the subproblems that have already been solved.

Therefore, the space complexity of the algorithm depends only on the number of matrices and not on the dimensions of the matrices.

**Experiment and result:**

Code:

// C++ program using memoization

#include <bits/stdc++.h>

using namespace std;

int dp[100][100];

// Function for matrix chain multiplication

int matrixChainMemoised(int\* p, int i, int j)

{

    if (i == j)

    {

        return 0;

    }

    if (dp[i][j] != -1)

    {

        return dp[i][j];

    }

    dp[i][j] = INT\_MAX;

    for (int k = i; k < j; k++)

    {

        dp[i][j] = min(

            dp[i][j], matrixChainMemoised(p, i, k)

                    + matrixChainMemoised(p, k + 1, j)

                    + p[i - 1] \* p[k] \* p[j]);

    }

    return dp[i][j];

}

int MatrixChainOrder(int\* p, int n)

{

    int i = 1, j = n - 1;

    return matrixChainMemoised(p, i, j);

}

// Driver Code

int main()

{

    int arr[] = { 1, 2, 3, 4 };

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

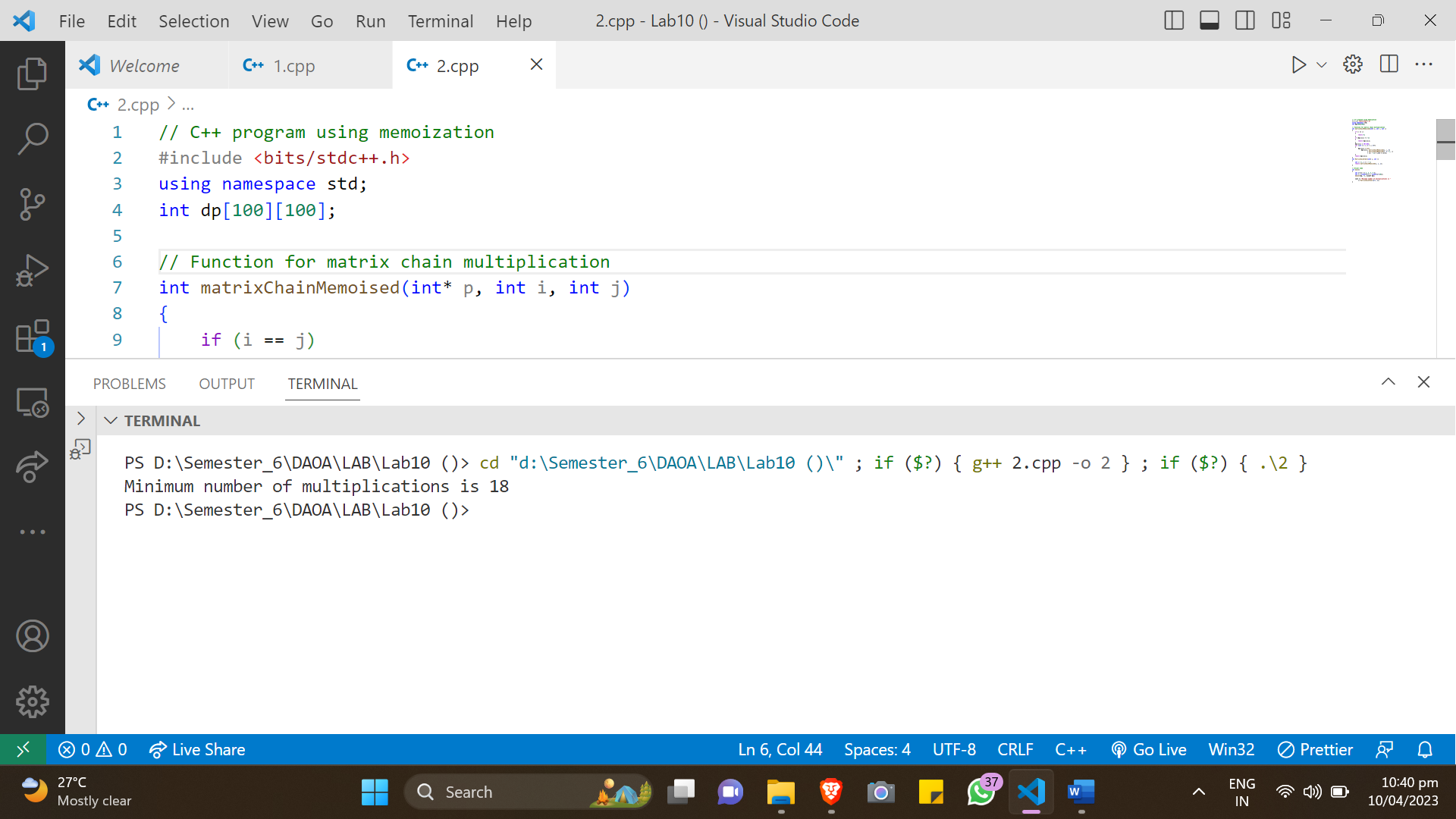
    memset(dp, -1, sizeof dp);

    cout << "Minimum number of multiplications is "

        << MatrixChainOrder(arr, n);

}

Output:



**Conclusions:**

In conclusion, the Matrix Chain Multiplication algorithm is an efficient way to multiply a chain of matrices. It works by breaking down the problem into subproblems and storing the results of these subproblems in a table using dynamic programming. 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. The algorithm is particularly useful in applications where large numbers of matrices need to be multiplied, such as in computer graphics and scientific simulations.