Name:Kunal Virendra Bhatia

UID:2021300010

Subject:DAA

Experiment No:4

Aim – Experiment to implement matrix chain multiplication.

Objective: To understand dynamic programming approach

Theory:

Matrix chain multiplication (or the **matrix chain ordering problem**[1]) is an optimization problem concerning the most efficient way to multiply a given sequence of matrices. The problem is not actually to *perform* the multiplications, but merely to decide the sequence of the matrix multiplications involved. The problem may be solved using dynamic programming. There are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices *A*, *B*, *C*, and *D*, there are five possible options:

((*AB*)*C*)*D* = (*A*(*BC*))*D* = (*AB*)(*CD*) = *A*((*BC*)*D*) = *A*(*B*(*CD*)).

The idea is to break the problem into a set of related subproblems that group the given matrix to yield the lowest total cost.

Following is the recursive algorithm to find the minimum cost:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the price of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices `ABCD`, we compute the cost required to find each of `(A) (BCD)`, `(AB)(CD)`, and `(ABC)(D)`, making recursive calls to find the minimum cost to compute `ABC`, `AB`, `CD`, and `BCD` and then choose the best one. Better still, this yields the minimum cost and demonstrates the best way of doing the multiplication

---

Algorithm:

Dynamic Programming Solution for Matrix Chain Multiplication using Tabulation (Iterative Approach):

In iterative approach, we initially need to find the number of multiplications required to multiply two adjacent matrices. We can use these values to find the minimum multiplication required for matrices in a range of length 3 and further use those values for ranges with higher lengths.

Build on the answer in this manner till the range becomes [0, N-1]. Follow the steps mentioned below to implement the idea:

Iterate from l = 2 to N-1 which denotes the length of the range: Iterate from i = 0 to N-1:
Find the right end of the range (j) having l matrices.
Iterate from k = i+1 to j which denotes the point of partition.

Multiply the matrices in range (i, k) and (k, j).

This will create two matrices with dimensions arr[i-1]*arr[k] and arr[k]*arr[j].

The number of multiplications to be performed to multiply these two matrices (say X) are arr[i-1]*arr[k]*arr[j].

The total number of multiplications is dp[i][k]+ dp[k+1][j] + X. The value stored at dp[1][N-1] is the required answer.

Program:

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

class Matrix {
  public:
   float** m;
   int row;
   int col;

   Matrix(int r, int c) {
       m = new float*[r];
       for (int i = 0; i < r; i++) {
           m[i] = new float[c];
       }

       for (int i = 0; i < r; i++) {
           for (int j = 0; j < c; j++) {
               m[i][j] = 0;
           }
       }

       row = r;
       col = c;
   }

   void fill_random_in_range(int min, int max) {
       for (int i = 0; i < row; i++) {
           for (int j = 0; j < col; j++) {
               m[i][j] = rand() % (max - min + 1) + min;
           }
       }
   }

   static void print(Matrix* matrix, bool skip_zero = false, int w = 1) {
       int start = skip_zero ? 1 : 0;
       for (int i = start; i < matrix->row; i++) {
           for (int j = start; j < matrix->col; j++) {
               cout << left << setw(w) << matrix->m[i][j] << " ";
           }
           cout << endl;
       }
       cout << endl;
   }

   static long mul_count;
   static Matrix* multiply(Matrix* a, Matrix* b) {
       Matrix* c = new Matrix(a->row, b->col);

       for (int i = 0; i < a->row; i++) {
           for (int j = 0; j < b->col; j++) {
               float sum = 0;
```

```
                for (int k = 0; k < a->col; k++) {
                    sum += a->m[i][k] * b->m[k][j];
                    mul_count++;
                }
                c->m[i][j] = sum;
            }
        }
        return c;
    }
};

long Matrix::mul_count = 0;

void print_array(int* a, int n) {
    cout << "[ ";
    for (int i = 0; i < n; i++) {
        if (i == n - 1)
            cout << a[i];
        else
            cout << a[i] << ", ";
    }
    cout << "]" << endl;
}

int* gen_matrix_orders_in_range(int num, int min, int max) {
    int* p = new int[num + 1];
    srand(time(0));
    for (int i = 0; i <= num; i++) {
        p[i] = rand() % (max - min + 1) + min;
    }
    return p;
}

string optimal_parenthesization(Matrix* s, int i, int j) {
    if (i == j) {
        return "M" + to_string(i);

    } else {
        return "(" + optimal_parenthesization(s, i, s->m[i][j]) + "*" +
               optimal_parenthesization(s, s->m[i][j] + 1, j) + ")";
    }
}

string matrix_chain(int* p, int n, Matrix* m, Matrix* s) {
    int t = 1;
    for (int i = 1; i <= n - 1; i++) {
        for (int j = 1; j + t <= n; j++) {
            // j and (j + t) are indices of m
            // k = j to (j + t) - 1
            int min = INT_MAX;
            for (int k = j; k <= j + t - 1; k++) {
                int cost = m->m[j][k] + m->m[k + 1][j + t] +
```

```
                        p[j - 1] * p[k] * p[j + t];

                if (cost < min) {
                    min = cost;
                    m->m[j][j + t] = min;
                    s->m[j][j + t] = k;
                }
            }
        }
        t++;
    }

    return optimal_parenthesization(s, 1, n);
}

string to_postfix(string infix) {
    string postfix = "";
    vector<char> stack;

    for (int i = 0; i < infix.size(); i++) {
        char ch = infix[i];
        if (ch == '(') {
            stack.push_back(ch);
        } else if (ch == '*') {
            stack.push_back('*');
        } else if (ch == ')') {
            while (stack[stack.size() - 1] != '(') {
                postfix = postfix + stack.back();
                stack.pop_back();
            }
            stack.pop_back();
        } else {
            if (ch == 'M') {
                postfix += " ";
            }
            postfix = postfix + ch;
        }
    }

    while (stack.size() != 0) {
        char pop = stack.back();
        postfix = postfix + pop;
        stack.pop_back();
    }

    return postfix;
}

long count_normal = 0;

Matrix* eval_matrix_normal_mul(string postfix, Matrix** m_arr) {
    vector<Matrix*> eval;
```

```
    Matrix::mul_count = 0;

    for (int i = 0; i < postfix.size(); i++) {
        char ch = postfix[i];

        if (ch == 'M' || ch == ' ') {
            continue;
        }
        if (ch == '*') {
            Matrix* b = eval.back();
            eval.pop_back();

            Matrix* a = eval.back();
            eval.pop_back();

            Matrix* c = Matrix::multiply(a, b);

            eval.push_back(c);

        } else if (ch >= '1' || ch <= '9') {
            int index = ch - '0';
            if (ch == '1' && postfix[i + 1] == '0') {
                index = 10;
                i++;
            }
            eval.push_back(m_arr[index]);
        }
    }
    return eval.back();
}

int main() {
    int MATRIX_COUNT = 0;

    cout << "Enter number of matrices (<= 10) : ";
    cin >> MATRIX_COUNT;

    int* p = gen_matrix_orders_in_range(MATRIX_COUNT, 15, 46);

    cout << "\np[i] = ";
    print_array(p, MATRIX_COUNT + 1);
    cout << endl;

    Matrix** M = new Matrix*[MATRIX_COUNT + 1];

    for (int i = 1; i <= MATRIX_COUNT; i++) {
        M[i] = new Matrix(p[i - 1], p[i]);
        M[i]->fill_random_in_range(0, 1);
        cout << "Order of M" << i << " is (" << M[i]->row << ", " << M[i]->col
             << ")" << endl;
        Matrix::print(M[i]);
```

```
    }

    Matrix* m = new Matrix(MATRIX_COUNT + 1, MATRIX_COUNT + 1);
    Matrix* s = new Matrix(MATRIX_COUNT + 1, MATRIX_COUNT + 1);

    string optimum_inorder = matrix_chain(p, MATRIX_COUNT, m, s);

    cout << "\nCost matrix" << endl;
    Matrix::print(m, true, 8);

    cout << "\nParenthesization Matrix" << endl;
    Matrix::print(s, true);

    cout << "Optimal parenthesization : " << optimum_inorder << endl;

    string optimum_postfix = to_postfix(optimum_inorder);
    cout << "Postfix expression : " << optimum_postfix << endl;

    cout << "\nResult Of Multiplication : " << endl;

    Matrix::print(eval_matrix_normal_mul(optimum_postfix, M));

    cout << "Estimated Multiplication count: " << m->m[1][m->col - 1] << endl;
    cout << "Actual Multiplication count: " << Matrix::mul_count << endl;

    return 0;
}
```

Output with random p values:

```
Cost matrix
0       24840   33480   40920
0       0       26496   49312
0       0       0       17856
0       0       0       0


Parenthesization Matrix
0 1 2 3
0 0 2 3
0 0 0 3
0 0 0 0

Optimal parenthesization : (((M1*M2)*M3)*M4)
Postfix expression :  M1 M2* M3* M4*
```

Inference:

I have observed that the order of matrix while multiplication is critical while multiplying the matrices. Determining the optimal way to parenthesize the matrices will significantly decrease the number of scalar multiplications needed to obtain the final result.

Conclusion:
Thus, by performing this experiment I understood the significance of matrix chain multiplication and was also able to implement it.