

|  |  |
| --- | --- |
| **Name** | Vedant Deshmukh |
| **UID No.** | 2021300025 |
| **Class** | COMPS A (B batch) |
| **Experiment No.** | 4 |

**Aim:** Experiment on Dynamic Programming

* Take matrix count as input (up to 10)
* Generate random matrix whose order is between 15 to 46.
* Fill all matrix with random distribution of values 1 and 0.
* Determine the optimal parenthesizing of matrices.
* Perform matrix multiplication according to that parenthesizing.

# Theory:

* **Dynamic Programming**

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have two properties:

1. Overlapping subproblems
2. Optimal substructure property

If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution (typically bottom-up fashion)
4. Construct an optimal solution from computed information.



# Algorithm:

1. Matrix Chain Multiplication cost calulation MCM(p)

n = p.length – 1

let m[1 .. n, 1 .. n] and s[1 .. n – 1, 2 .. n] be new tables for i = 1 to n

m[i, i] = 0 for l = 2 to n

for i = 1 to n – l + 1

j = i + l – 1

m[i, j] = infinity for k = i to j – 1

q = m[i, k] + m[k + 1, j] + p[i – 1] \* p[k] \* p[j] if q < m[i, j]

m[i, j] = q

s[i, j] = k

return m and s

1. Determining Optimal Parenthesization OptimalParens(s, i, j)

if i == j

return “M” + i else

return “(“ + OptimalParens(s, i, s[i][j]) + “\*“ + OptimalParens(s, s[i][j] + 1, j);



1. Convert Infix to Postfix

INFIX-TO-POSTFIX(E):

S <- empty stack

P <- empty list for postfix expression for each token t in E do:

if t is an operand, append t to P

if t is a left parenthesis, push t onto S if t is a right parenthesis, then:

while the top of S is not a left parenthesis

pop operators from S and append them to P pop the left parenthesis from S and discard it

if t is an operator, then:

while there is an operator on top of S with greater or equal precedence pop it from S and append it to P

push t onto S

while S is not empty, pop operators from S and append them to P return P

1. Postfix Evaluation EVALUATE-POSTFIX(P):

S <- empty stack

for each token t in P do:

if t is a number, push t onto S

if t is the multiplication operator, then: pop the top two numbers a and b from S compute a \* b

push the result onto S

the final result is the only element left on stack S return the result



1. Matrix Multiplication

MatrixMul(a, b, n)

Let c be resultant matrix of size n x n For i = 1 to n

for j = 1 to n

c[i, j] = 0

for k = 1 to n

c[i, j] = c[i, j] + a[i, k] \* b[k, j]

return c

# Code:

#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++) {

int 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;



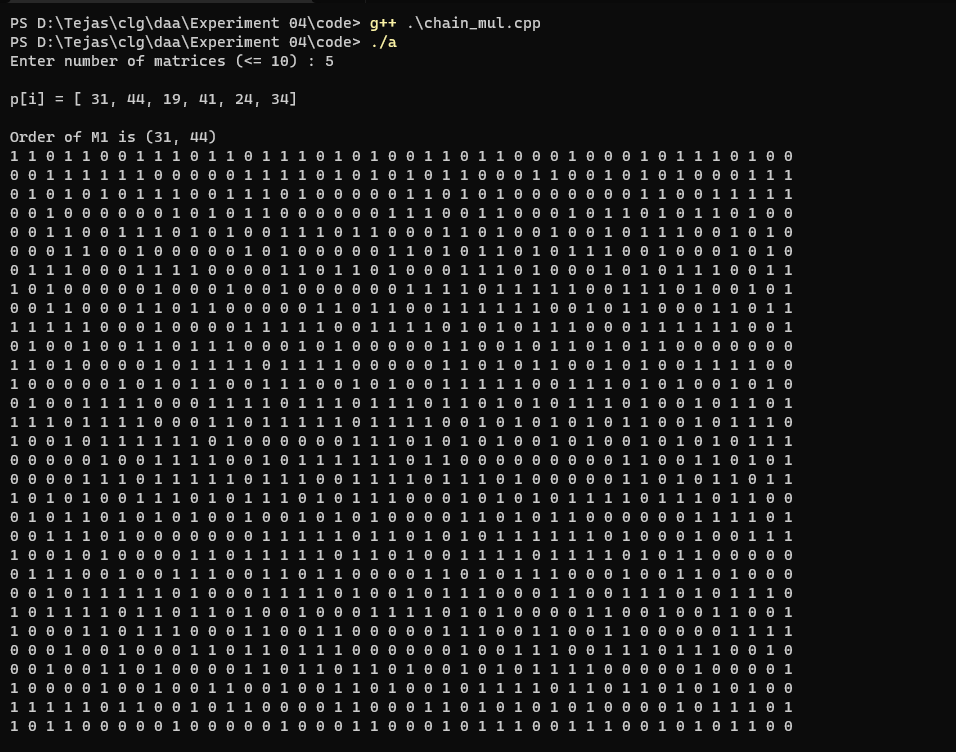
cout << "Actual Multiplication count: " << Matrix::mul\_count << endl;

return 0;

}

# Output:

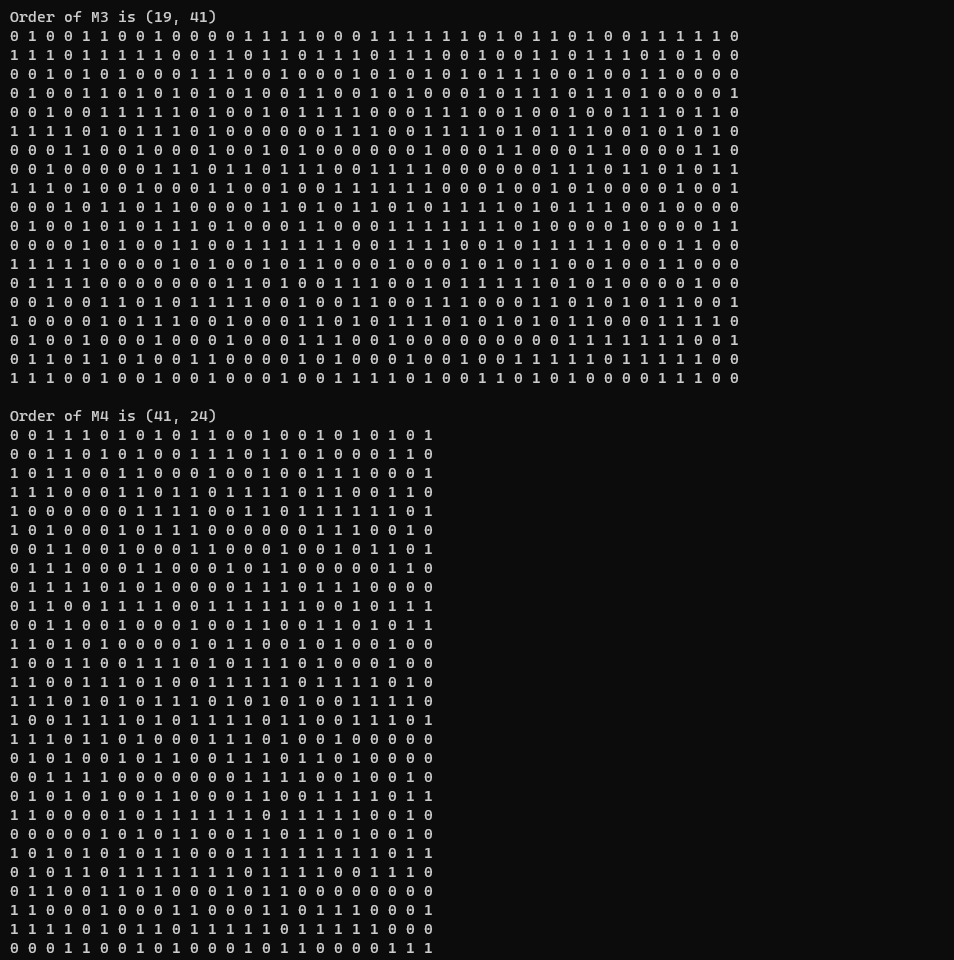
1. Number of Matrices = 5







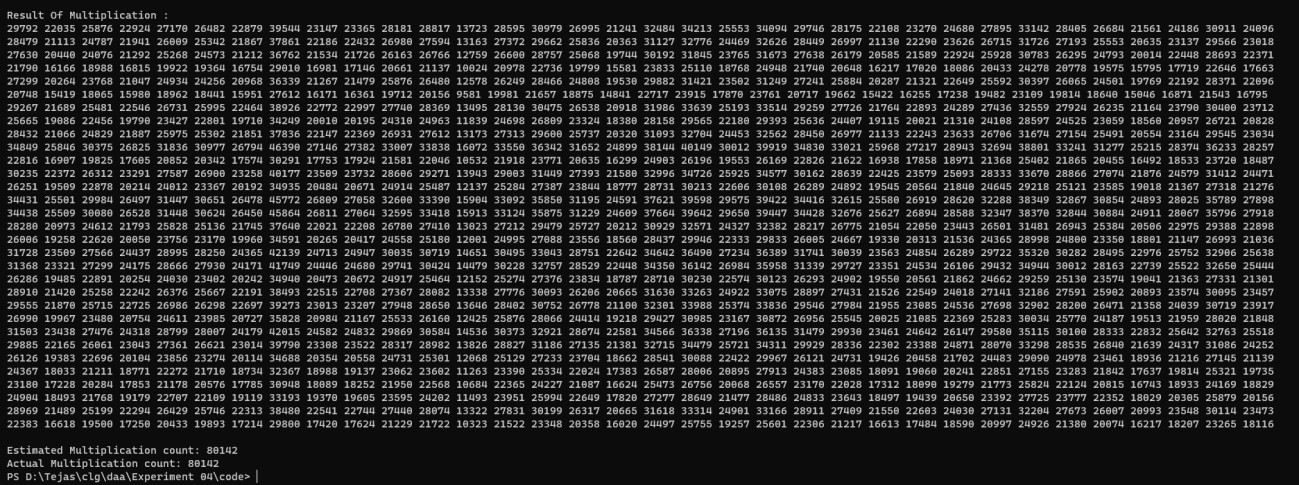






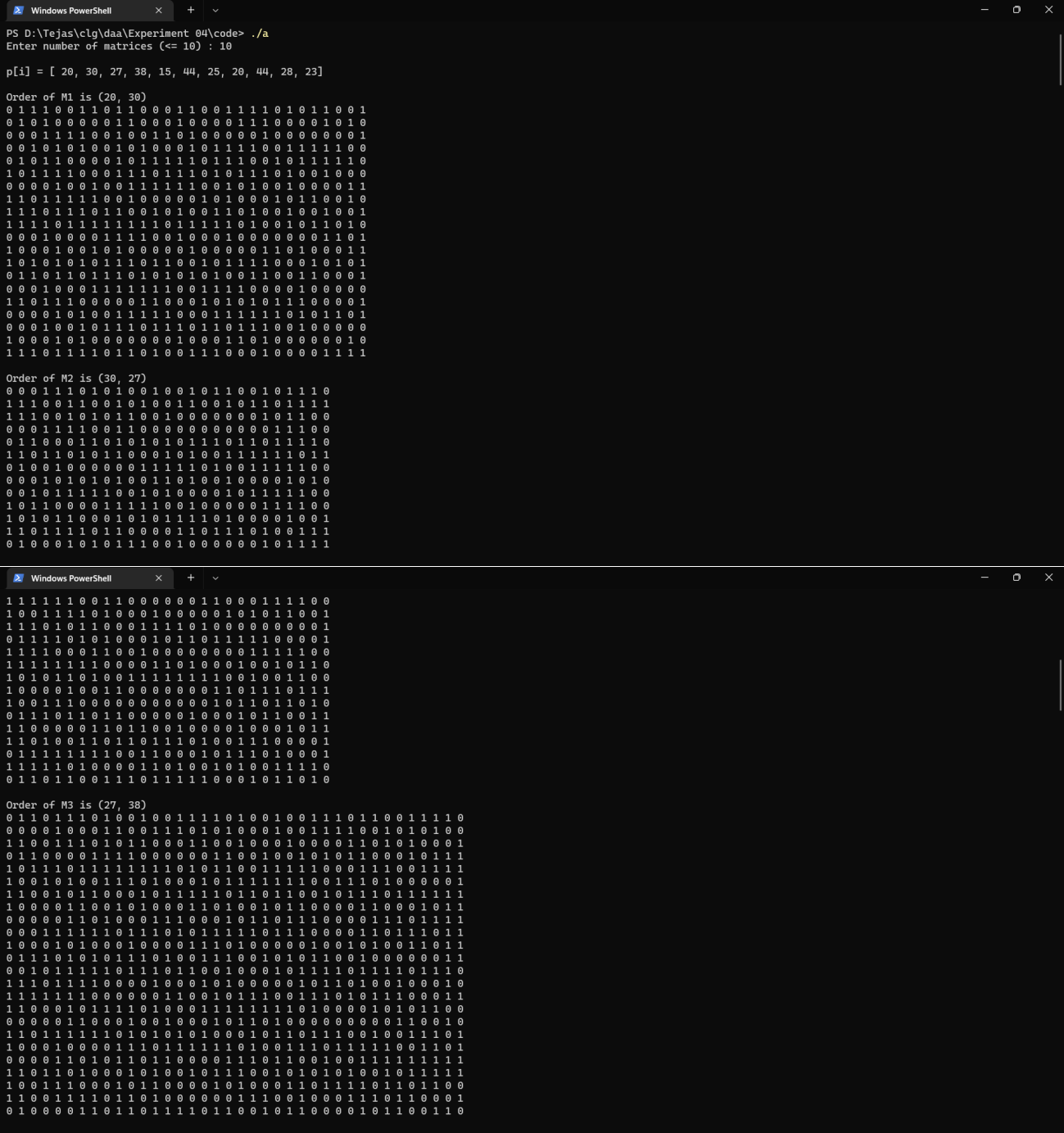




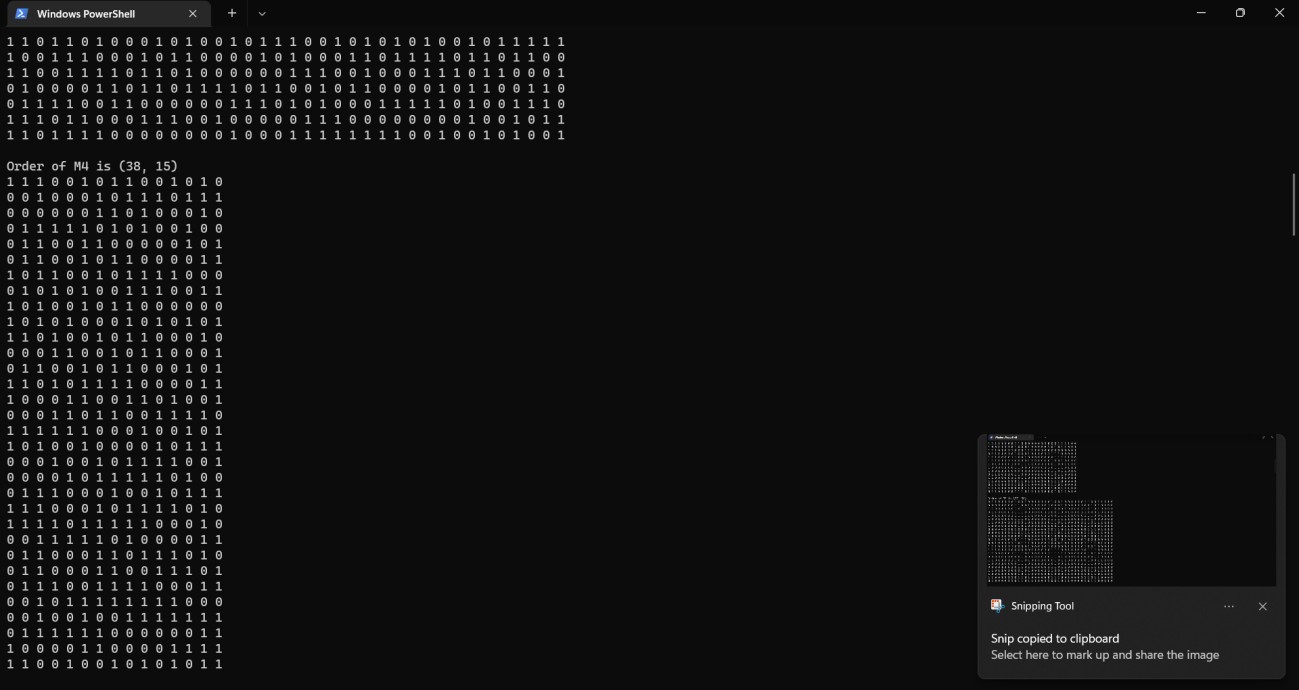


1. Number of Matrices = 10

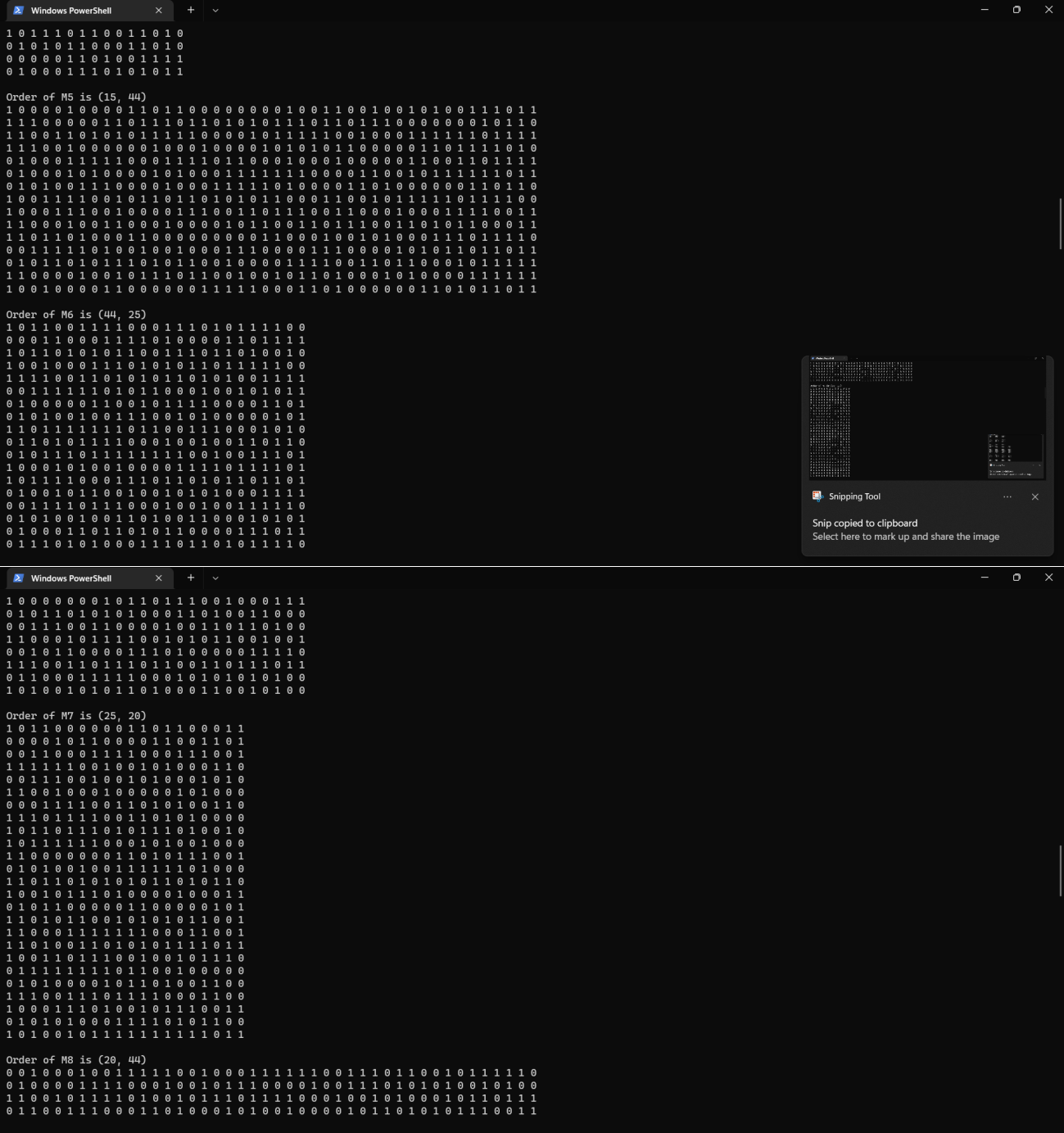




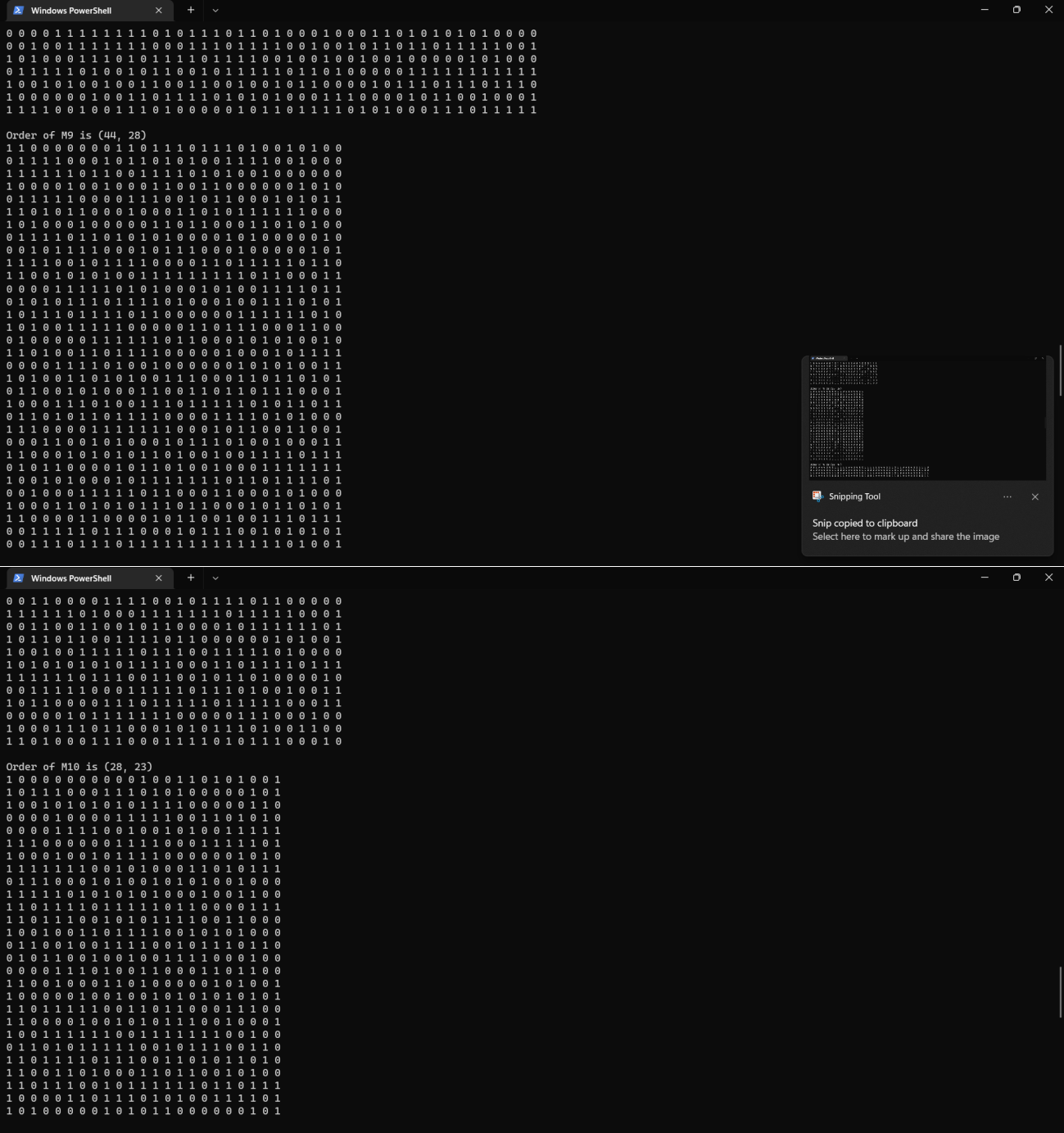




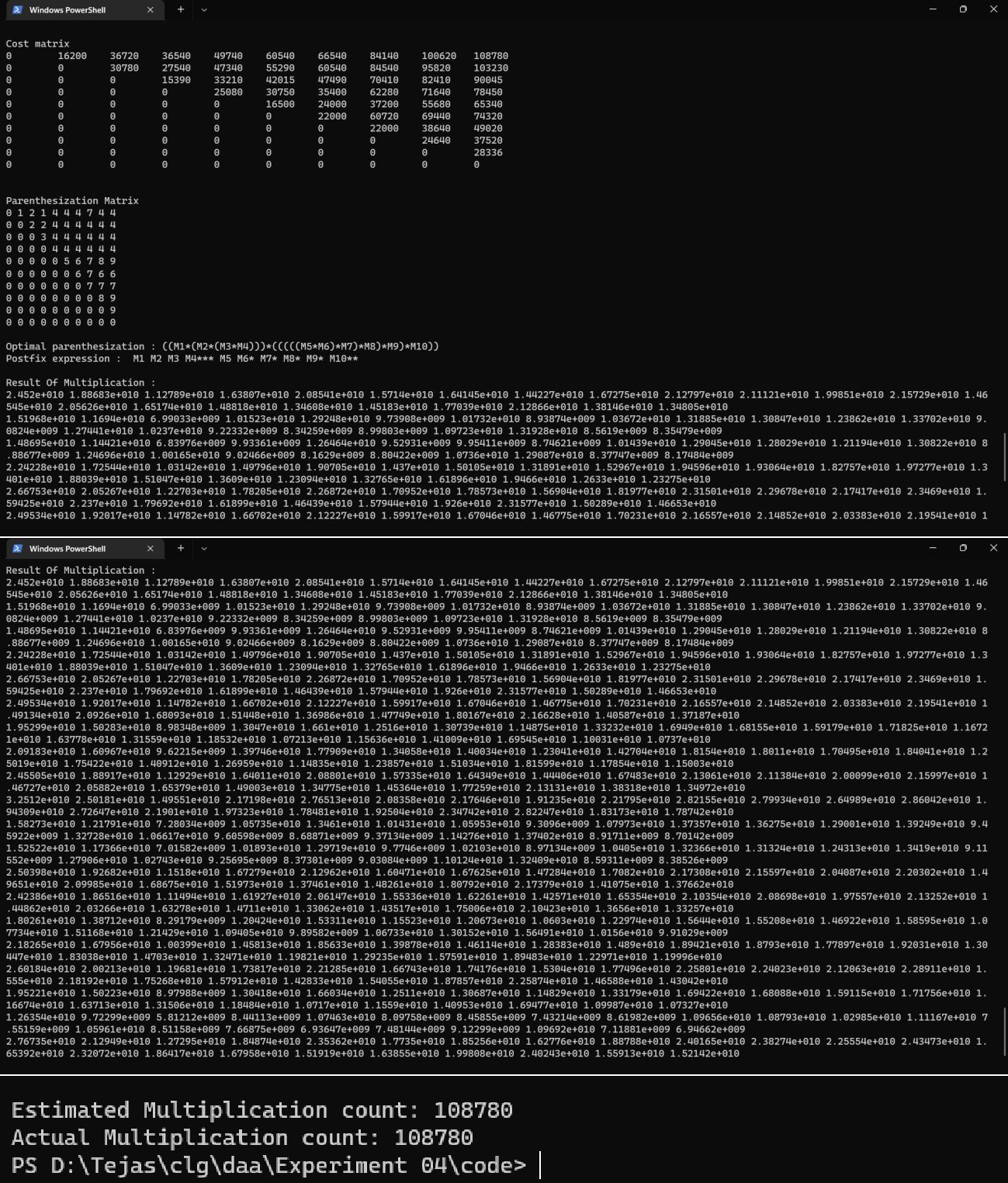














# Observation:

* The cost table tells the minimum cost for each pair of multiplication possible.
* The actual cost is equal to the minimum cost estimated by matrix chain multiplication.
* The parantehsization table also tells the cost for not only the entire multiplication but also any intermediate multiplication, hence, complying to the optimal substructure property.

# CONCLUSION:

After conducting this experiment, i have learnt to use dynamic programming approach to determine the optimal matric chain multiplication paranthesization. I have also learnt how the dynamic programming approach solves the problem and the importance of the optimal substructure property along with using tables to lookup previously calculated values.