BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	Balla Mahadev Shrikrishna
UID no.	2023300010
Experiment No.	6

AIM:	To implement Matrix Chain Multiplication (Dynamic Programming) for multiplying matrices using Strassen's Matrix Multiplication
Program 1	
PROBLEM STATEMENT :	**Problem Definition & Assumptions –** The aim of this experiment is two-fold. First, it finds the efficient way of multiplying a sequence of k matrices (called Matrix Chain Multiplication) using Dynamic Programming. The chain of multiplication M 1 x M 2 x M 3 x M 4 x...x M k may be computed in $(2N!)/((NN+1)!\ N!) = (2N \text{ combination } N)/(N+1)$ ways due to associative property where $NN = kk - 1$ of matrix multiplication. Consider the optimization problem of efficiently multiplying a randomly generated sequence of 10 matrices (M 1, M 2, M 3, M 4 ,..., M 10) using Dynamic programming approach. The dimension of these matrices are stored in an array p[i] for i = 0 to 10, where the dimension of the matrix M i is (p[i-1] x p[i]). All p[i] are randomly generated and they are in powers of twos (i.e. 2k for some k). For example, *pp*[0. .10] = (8, 16, 16, 64, 32, 32, 64, 16, 16, 8, 16). All ten matrices are generated randomly and each matrix value can be between 0 and 1. Determine following values of Matrix Chain Multiplication (MCM) using Dynamic Programming: 1) m[1..10][1..10] = Two dimension matrix of optimal solutions (No. of multiplications) of all possible matrices M 1... M 10 2) s[1..9][2..10] = Two dimension matrix of optimal solutions (parenthesizations) of all combinations of matrices M 1...M 10 3) the optimal solution (i.e.parenthesization) for the multiplication of all ten matrices M 1x M 2x M 3xM 4 x...x M 10 Find the running time of 10 matrices using regular matrix multiplication and Strassen's Matrix Multiplication as a trivial sequence i.e. ((((((((M 1x M 2)x M 3) xM 4 x...x M 10) and the sequence of matrix multiplication suggested by Matrix Chain Multiplication in Step No. 3

	Input – Each student has to generate dimension of 10 matrices using rand() function and store them in p[i]. All p[i] for i=0 to 10 are randomly generated and they are in powers of twos (i.e. 2 k for some k). All ten matrices are generated randomly and each matrix value can be between 0 and 1. **Submission and Output –** 1) Part 1 – Find Optimal Parenthesization a) m[1..10][1..10]= 2D Matrix of optimal solutions (No. of multiplications) of all possible matrices M1... M 10 b) s[1..9][2..10] = 2D Matrix of optimal solutions (parenthesizations) of all combinations M1...M 10 c) The optimal solution (i.e.parenthesization) for the multiplication of all matrices M1x M 2x M 3xM 4 x...x M 10 d) Print the time required to multiply ten matrices using four combinations as discussed above. 2) Part 2 – Use optimal parenthesizations in Part 1 to multiply ten matrices using regular matrix multiplication 3) Part 3 – Use optimal parenthesizations in Part 1 to multiply matrices using Strassen's Matrix Multiplication

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <limits.h>
#include <string.h>

typedef struct {
        double** data;
        int rows;
        int cols;
} Matrix;

// Creates a new matrix with given dimensions
Matrix createMatrix(int rows, int cols) {
        Matrix M;
        M.rows = rows;
```

```
	M.cols = cols;
	M.data = (double**)malloc(rows * sizeof(double*));
	for (int i = 0; i < rows; i++) {
	M.data[i] = (double*)malloc(cols * sizeof(double));
	}
	return M;
}

// Free memory allocated for a matrix
void freeMatrix(Matrix M) {
	for (int i = 0; i < M.rows; i++) {
	free(M.data[i]);
	}
	free(M.data);
}

// Generates a matrix with random values between 0 and 1
Matrix generateRandomMatrix(int rows, int cols) {
	Matrix M = createMatrix(rows, cols);
	for (int i = 0; i < rows; i++) {
	for (int j = 0; j < cols; j++) {
	M.data[i][j] = (double)rand() / RAND_MAX; // Random value [0,1]
	}
	}
	return M;
}

// Prints matrix with formatted output
void printMatrix(Matrix M) {
	for (int i = 0; i < M.rows; i++) {
	for (int j = 0; j < M.cols; j++) {
	printf("%8.4f ", M.data[i][j]); // 4 decimal places, 8 width
	}
	printf("\n");
	}
}

// Standard O(n³) matrix multiplication - triple loop for matrix
```

```
multiplication
Matrix regularMultiply(Matrix A, Matrix B) {
        int r = A.rows;
        int c = B.cols;
        int inner = A.cols;
        Matrix C = createMatrix(r, c);

        for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
        C.data[i][j] = 0.0;
        for (int k = 0; k < inner; k++) {
                C.data[i][j] += A.data[i][k] * B.data[k][j];
        }
        }
        }
        return C;
}

// Matrix addition
Matrix addMatrix(Matrix A, Matrix B) {
        int r = A.rows, c = A.cols;
        Matrix C = createMatrix(r, c);
        for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
        C.data[i][j] = A.data[i][j] + B.data[i][j];
        }
        }
        return C;
}

// Matrix subtraction
Matrix subMatrix(Matrix A, Matrix B) {
        int r = A.rows, c = A.cols;
        Matrix C = createMatrix(r, c);
        for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
        C.data[i][j] = A.data[i][j] - B.data[i][j];
        }
```

```
    }
    return C;
}

// Extracts a submatrix from given position
Matrix getSubmatrix(Matrix A, int row, int col, int size) {
    Matrix sub = createMatrix(size, size);
    // Copy elements from original matrix
    for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
    sub.data[i][j] = A.data[row + i][col + j];
    }
    }
    return sub;
}

// Copies submatrix into another matrix at given position
void setSubmatrix(Matrix* C, Matrix sub, int row, int col) {
    int size = sub.rows;
    // Copy elements to target matrix
    for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
    C->data[row + i][col + j] = sub.data[i][j];
    }
    }
}

// Pads matrix with zeros to make it square of size n×n
Matrix padMatrix(Matrix A, int n) {
    Matrix B = createMatrix(n, n);
    // Copy original elements, pad with zeros
    for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
    B.data[i][j] = (i < A.rows && j < A.cols) ? A.data[i][j] : 0.0;
    }
    }
    return B;
}
```

```
// Removes padding to restore original dimensions
Matrix unpadMatrix(Matrix A, int rows, int cols) {
	Matrix B = createMatrix(rows, cols);
	// Copy only the original portion
	for (int i = 0; i < rows; i++) {
	for (int j = 0; j < cols; j++) {
	B.data[i][j] = A.data[i][j];
	}
	}
	return B;
}

// Strassen's algorithm for square matrices (recursive)
Matrix strassenMultiplySquare(Matrix A, Matrix B) {
	int n = A.rows;
	Matrix C = createMatrix(n, n);

	// Base case: 1×1 matrix
	if (n == 1) {
	C.data[0][0] = A.data[0][0] * B.data[0][0];
	return C;
	}

	int newSize = n / 2;

	// Divide matrices into 4 submatrices each
	Matrix A11 = getSubmatrix(A, 0, 0, newSize);
	Matrix A12 = getSubmatrix(A, 0, newSize, newSize);
	Matrix A21 = getSubmatrix(A, newSize, 0, newSize);
	Matrix A22 = getSubmatrix(A, newSize, newSize, newSize);

	Matrix B11 = getSubmatrix(B, 0, 0, newSize);
	Matrix B12 = getSubmatrix(B, 0, newSize, newSize);
	Matrix B21 = getSubmatrix(B, newSize, 0, newSize);
	Matrix B22 = getSubmatrix(B, newSize, newSize, newSize);

	// Compute the 7 products recursively
```

	Matrix M1 = strassenMultiplySquare(addMatrix(A11, A22), addMatrix(B11, B22));
	Matrix M2 = strassenMultiplySquare(addMatrix(A21, A22), B11);
	Matrix M3 = strassenMultiplySquare(A11, subMatrix(B12, B22));
	Matrix M4 = strassenMultiplySquare(A22, subMatrix(B21, B11));
	Matrix M5 = strassenMultiplySquare(addMatrix(A11, A12), B22);
	Matrix M6 = strassenMultiplySquare(subMatrix(A21, A11), addMatrix(B11, B12));
	Matrix M7 = strassenMultiplySquare(subMatrix(A12, A22), addMatrix(B21, B22));
	
	// Compute result submatrices using Strassen's formulas
	Matrix C11 = addMatrix(subMatrix(addMatrix(M1, M4), M5), M7);
	Matrix C12 = addMatrix(M3, M5);
	Matrix C21 = addMatrix(M2, M4);
	Matrix C22 = addMatrix(subMatrix(addMatrix(M1, M3), M2), M6);
	
	// Combine submatrices into final result
	for (int i = 0; i < newSize; i++) {
	for (int j = 0; j < newSize; j++) {
	C.data[i][j] = C11.data[i][j];
	C.data[i][j + newSize] = C12.data[i][j];
	C.data[i + newSize][j] = C21.data[i][j];
	C.data[i + newSize][j + newSize] = C22.data[i][j];
	}
	}
	
	// Free all temporary matrices
	freeMatrix(A11); freeMatrix(A12); freeMatrix(A21); freeMatrix(A22);
	freeMatrix(B11); freeMatrix(B12); freeMatrix(B21); freeMatrix(B22);
	freeMatrix(M1); freeMatrix(M2); freeMatrix(M3); freeMatrix(M4);
	freeMatrix(M5); freeMatrix(M6); freeMatrix(M7);
	freeMatrix(C11); freeMatrix(C12); freeMatrix(C21); freeMatrix(C22);
	
	return C;

```
}

// Strassen's algorithm with padding for non-square matrices
Matrix strassenMultiply(Matrix A, Matrix B) {
	int r1 = A.rows;
	int c1 = A.cols;
	int r2 = B.rows;
	int c2 = B.cols;

	// Find smallest power of 2 that can contain all dimensions
	int n = r1;
	if (c1 > n) n = c1;
	if (r2 > n) n = r2;
	if (c2 > n) n = c2;

	int mSize = 1;
	while (mSize < n) mSize *= 2;

	// Pad matrices to make them square with power-of-2 dimensions
	Matrix A_padded = padMatrix(A, mSize);
	Matrix B_padded = padMatrix(B, mSize);
	Matrix C_padded = strassenMultiplySquare(A_padded, B_padded);
	Matrix C = unpadMatrix(C_padded, r1, c2);

	// Free temporary padded matrices
	freeMatrix(A_padded);
	freeMatrix(B_padded);
	freeMatrix(C_padded);

	return C;
}

// Dynamic programming solution for matrix chain ordering
void matrixChainOrder(int* p, int n, long long** m, int** s) {
	// Initialize diagonal (single matrix costs 0)
	for (int i = 1; i <= n; i++) {
	m[i][i] = 0;
	}
```

```
    // Fill DP tables for chain lengths from 2 to n
    for (int l = 2; l <= n; l++) {
    for (int i = 1; i <= n - l + 1; i++) {
    int j = i + l - 1;
    m[i][j] = LLONG_MAX;
    // Try all possible split points
    for (int k = i; k < j; k++) {
        long long q = m[i][k] + m[k + 1][j] + (long long)p[i - 1] *
p[k] * p[j];
        if (q < m[i][j]) {
        m[i][j] = q;
        s[i][j] = k; // Store optimal split point
        }
    }
    }
    }
}

// Recursively prints optimal parenthesization
void printOptimalParens(int** s, int i, int j) {
    if (i == j) {
    printf("M%d", i); // Base case: single matrix
    }
    else {
    printf("(");
    printOptimalParens(s, i, s[i][j]); // Left subexpression
    printf(" X ");
    printOptimalParens(s, s[i][j] + 1, j); // Right subexpression
    printf(")");
    }
}

// Multiplies matrix chain using optimal parenthesization
Matrix multiplyChainOptimal(Matrix* matrices, int** s, int i, int j, int
useStrassen) {
    if (i == j) {
    return matrices[i - 1]; // Base case: return single matrix
```

```
    }
    int k = s[i][j]; // Optimal split point

    // Recursively multiply left and right parts
    Matrix A = multiplyChainOptimal(matrices, s, i, k, useStrassen);
    Matrix B = multiplyChainOptimal(matrices, s, k + 1, j, useStrassen);
    Matrix result;

    // Use specified multiplication algorithm
    if (useStrassen) {
    result = strassenMultiply(A, B);
    }
    else {
    result = regularMultiply(A, B);
    }

    // Free intermediate matrices if they were created
    if (i != k) freeMatrix(A);
    if (k + 1 != j) freeMatrix(B);
    return result;
}

// Multiplies matrix chain in trivial left-to-right order
Matrix multiplyChainTrivial(Matrix* matrices, int n, int useStrassen) {
    Matrix result = matrices[0];
    for (int i = 1; i < n; i++) {
    Matrix temp;
    // Use specified multiplication algorithm
    if (useStrassen) {
    temp = strassenMultiply(result, matrices[i]);
    }
    else {
    temp = regularMultiply(result, matrices[i]);
    }

    // Free previous result if it wasn't the first matrix
    if (i > 1) freeMatrix(result);
    result = temp;
```

```
        }
        return result;
}

int main() {
        srand(time(0)); // Seed random number generator
        int n = 10; // Number of matrices in chain
        int p[n + 1]; // Array of matrix dimensions

        int possible[] = { 8, 16, 32, 64 }; // Possible matrix dimensions
(powers of 2 for Strassen)
        for (int i = 0; i < n + 1; i++) {
        p[i] = possible[rand() % 4]; // Randomly select dimensions
        }

        printf("Matrix dimensions array p: ");
        for (int i = 0; i < n + 1; i++) {
        printf("%d ", p[i]);
        }
        printf("\n");

        // Generate random matrices with specified dimensions
        Matrix* matrices = (Matrix*)malloc(n * sizeof(Matrix));
        for (int i = 0; i < n; i++) {
        int rows = p[i];
        int cols = p[i + 1];
        matrices[i] = generateRandomMatrix(rows, cols);
        }

        // Allocate m and s matrices for matrix chain ordering
        long long** m = (long long**)malloc((n + 1) * sizeof(long long*));
        int** s = (int**)malloc((n + 1) * sizeof(int*));
        for (int i = 0; i <= n; i++) {
        m[i] = (long long*)malloc((n + 1) * sizeof(long long));
        s[i] = (int*)malloc((n + 1) * sizeof(int));
        }

        matrixChainOrder(p, n, m, s); // Compute optimal matrix chain
```

ordering

```
        // Print DP tables
        printf("\nMatrix m (Optimal Multiplication Costs):\n");
        for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
        if (j < i)
                printf("%8s", "0"); // Lower triangle is unused
        else
                printf("%8lld", m[i][j]); // Cost from i to j
        }
        printf("\n");
        }

        printf("\nMatrix s (Optimal Splits):\n");
        for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
        if (j <= i)
                printf("%4s", "0"); // Lower triangle is unused
        else
                printf("%4d", s[i][j]); // Optimal split point
        }
        printf("\n");
        }

        printf("\nOptimal Parenthesization: ");
        printOptimalParens(s, 1, n);
        printf("\n");

        clock_t start, end;
        double duration;

        start = clock();
        Matrix trivialRegular = multiplyChainTrivial(matrices, n, 0);
        end = clock();
        double durationTrivialRegular = ((double)(end - start)) /
CLOCKS_PER_SEC * 1000;
```

```
    start = clock();
    Matrix optimalRegular = multiplyChainOptimal(matrices, s, 1, n, 0);
    end = clock();
    double durationOptimalRegular = ((double)(end - start)) /
CLOCKS_PER_SEC * 1000;

    start = clock();
    Matrix trivialStrassen = multiplyChainTrivial(matrices, n, 1);
    end = clock();
    double durationTrivialStrassen = ((double)(end - start)) /
CLOCKS_PER_SEC * 1000;

    start = clock();
    Matrix optimalStrassen = multiplyChainOptimal(matrices, s, 1, n,
1);
    end = clock();
    double durationOptimalStrassen = ((double)(end - start)) /
CLOCKS_PER_SEC * 1000;

    // Print timing results
    printf("\nTiming Results (in milliseconds):\n");
    printf("1. Trivial order using Regular Multiplication: %.2f ms\n",
durationTrivialRegular);
    printf("2. Trivial order using Strassen Multiplication: %.2f ms\n",
durationTrivialStrassen);
    printf("3. Optimal order using Regular Multiplication: %.2f ms\n",
durationOptimalRegular);
    printf("4. Optimal order using Strassen Multiplication: %.2f ms\n",
durationOptimalStrassen);

    // Free allocated memory
    for (int i = 0; i < n; i++) {
    freeMatrix(matrices[i]);
    }
    free(matrices);

    for (int i = 0; i <= n; i++) {
    free(m[i]);
```

```
        free(s[i]);
        }
        free(m);
        free(s);

        freeMatrix(trivialRegular);
        freeMatrix(optimalRegular);
        freeMatrix(trivialStrassen);
        freeMatrix(optimalStrassen);

        return 0;
}
```

RESULT:

```
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/exp6$ gcc mcm.c
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/exp6$ ./a.out
Matrix dimensions array p: 64 8 8 32 32 64 16 8 8 64 32

Matrix m (Optimal Multiplication Costs):
       0    4096   18432   26624   59392   41984   38400   38912   71680   69632
       0       0    2048   10240   26624   33792   34304   34816   38912   53248
       0       0       0    8192   24576   32768   33792   34304   38400   52736
       0       0       0       0   65536   49152   32768   33792   50176   58368
       0       0       0       0       0   32768   24576   25600   41984   50176
       0       0       0       0       0       0    8192    9216   41984   41984
       0       0       0       0       0       0       0    1024    9216   21504
       0       0       0       0       0       0       0       0    4096   18432
       0       0       0       0       0       0       0       0       0   16384
       0       0       0       0       0       0       0       0       0       0

Matrix s (Optimal Splits):
   0   1   1   1   1   1   1   1   1   1
   0   0   2   2   4   2   2   2   8   8
   0   0   0   3   4   5   6   7   8   8
   0   0   0   0   4   4   4   4   8   8
   0   0   0   0   0   5   5   5   8   8
   0   0   0   0   0   0   6   6   8   8
   0   0   0   0   0   0   0   7   8   8
   0   0   0   0   0   0   0   0   8   8
   0   0   0   0   0   0   0   0   0   9
   0   0   0   0   0   0   0   0   0   0

Optimal Parenthesization: (M1 X ((M2 X (((((M3 X M4) X M5) X M6) X M7) X M8)) X (M9 X M10)))

Timing Results (in milliseconds):
1. Trivial order using Regular Multiplication: 5.05 ms
2. Trivial order using Strassen Multiplication: 385.27 ms
3. Optimal order using Regular Multiplication: 0.76 ms
4. Optimal order using Strassen Multiplication: 164.78 ms
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/exp6$
```

CONCLUSION:

* Matrix multiplicat^{n}

Input : $A = [a_{ij}]$, $B = [b_{ij}]$
Output : $C = [c_{ij}]$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Running time $= \Theta(n^3)$

Divide & Conquer - Idea:
$n \times n$ matrix $= 2 \times 2$ matrix of $(n/2) \times (n/2)$ submatrices

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C \quad = \quad A \cdot B$$

$$\left.\begin{array}{l} r = ae + bg \ , \ s = af + bh \\ t = ce + dg \ , \ u = cf + dh \end{array}\right\} \begin{array}{l} 4 \text{ addit}^{n}\text{s.} \\ 8 \text{ multiplicat}^{n}\text{s.} \end{array}$$

$$T(n) = \underbrace{8}_{\text{submatrices}} \cdot T\underbrace{(n/2)}_{\text{size of submatrix}} + \underbrace{\Theta(n^2)}_{\text{work adding submatrices}}$$

$$\underbrace{T(n) = \Theta(n^3)}_{\text{found using Master Th}^{m}} \implies \text{no improvement}$$

- Strassen's Idea
Multiply 2X2 matrices with only 7 recursive mults.

$P_1 = a \cdot (f-h)$
$P_2 = (a+b) \cdot h$
$P_3 = (c+d) \cdot e$
$P_4 = d \cdot (g-e)$
$P_5 = (a+d) \cdot (e+h)$
$P_6 = (b-d) \cdot (g+h)$
$P_7 = (a-c) \cdot (e+f)$

$r = P_5 + P_4 - P_2 + P_6$
$s = P_1 + P_2$
$t = P_3 + P_4$
$u = P_5 + P_1 - P_3 - P_7$

7 mults., 18 add/subs.
Note: No reliance on commututivity ~~of~~ of mult.

On solving for r we get the same value as found in divide & conquer approach.

1) Divide: divide 2x2 matrices in $(n/2) \times (n/2)$ submatrices.
2) Conquer: perform 7 mults recursively on the submatrices.
3) Combine: Form matrix C using add/subs.

Analysis: $T(n) = 7\,T(n/2) + \Theta(n^2)$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Longrightarrow \text{Case 1}: T(n) = \Theta(n^{\lg 7})$$

2.81 isn't much smaller than 3 but bcoz the difference is in the exp., its impact on running time is significant. This algo. beats the ordinary algo. on today's machines for $n \geq 32$ or so.

* Matrix-Chain Multiplicatn

- paranthesize in a way that minimizes the no. of scalar multiplicatns.
- For a chain of matrices $\langle A_1, A_2, A_3, A_4 \rangle$, we can paranthesize the product in 5 distinct ways:

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$

- Matrix-Multiply:

```
if A.cols ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows X B.cols
    for i=1 to A.rows
```

- Given a chain of n matrices $\langle A_1, A_2, A_3, \ldots A_n \rangle$
- Counting no. of ~~paranthes~~ paranthesizatns of a seq. of n matrices by $P(n)$.
- The split b/w two subproducts may occur b/w k-th and (k+1)-st matrices for any k.

$$P(n) = \begin{cases} 1 & ; \text{if } n=1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & ; \text{if } n \geq 2. \end{cases}$$

soln to recurrence is $\Omega(4^n / n^{3/2})$.

catalan no.

- $P(n) = C(n) = \frac{(2n)!}{n!\,(n+1)!}$

no. of operators

- Applying DP
  - → characterize struct. of optimal sol^n.
  - → recursively define value of optimal sol^n.
  - → compute value of an optimal sol^n in a bottom-up fashion.
  - → construct optimal sol^n from computed info.

Step 1: Optimal Structure

Find optimal substruct & then use it to construct an optimal sol^n to the problem from optimal sol^ns to subproblems.

$A_{i....j}$: the matrix resulting from evaluating the product $A_i A_{i+1} ..... A_{i+j}$

We must split the product b/w A_k & A_{k+1} for some int. k in the range $i \le k \le j$.

i.e., for some val of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1.....j}$ & then multiply them together to produce final product $A_{i.....j}$.

Step 2: Recursive Sol^n

Can define $m[i,j]$ recursively as follows
→ If $i=j$, $m[i,j]=0$ for $1 \le i = j \le n$
→ When $i<j$, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$

	The eqⁿ assumes that we know value of k, which we don't. There are only j-i possible values for k, however, namely k=i, i+1, ..., j-1. Optimal paranthesizatⁿ must use one of these values for k, so need to check them all to find the best. ∴ Recursive defⁿ for min. cost of paranthesizing product $A_i A_{i+1} \ldots A_j$ becomes $m[i,j] = \begin{cases} 0 & ; \text{if } i=j \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & ; \text{if } i<j \end{cases}$