Name: Rishi Tiku

Class: SE DS

UID: 2021700067

Subject: DAA LAB

Experiment: 5

Aim: To solve Matrix Chain Multiplication problem using Dynamic Programming approach.

Problem:

Given the dimension of a sequence of matrices in an array **arr[]**, where the dimension of the **i^{th}** matrix is **(arr[i-1] * arr[i])**, the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

Matrix Chain Multiplication using Recursion:

We can solve the problem using recursion based on the following facts and observations:

*Two matrices of size **m*n** and **n*p** when multiplied, they generate a matrix of size **m*p** and the number of multiplications performed are **m*n*p**.*

*Now, for a given chain of **N** matrices, the first partition can be done in **N-1** ways. For example, sequence of matrices A, B, C and D can be grouped as (A)(BCD), (AB)(CD) or (ABC)(D) in these 3 ways.*

*So a range **[i, j]** can be broken into two groups like **{[i, i+1], [i+1, j]}, {[i, i+2], [i+2, j]}, . . . , {[i, j-1], [j-1, j]}**.*

- *Each of the groups can be further partitioned into smaller groups and we can find the total required multiplications by solving for each of the groups.*
- *The minimum number of multiplications among all the first partitions is the required answer.*

The time complexity of the solution is exponential.

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

If observed carefully you can find the following two properties:

1) Optimal Substructure: *In the above case, we are breaking the bigger groups into smaller subgroups and solving them to finally find the minimum number of multiplications. Therefore, it can be said that the problem has optimal substructure property.*

2) Overlapping Subproblems: *We can see in the recursion tree that the same subproblems are called again and again and this problem has the Overlapping Subproblems property.*

So Matrix Chain Multiplication problem has both properties of a dynamic programming problem. So recomputations of same subproblems can be avoided by constructing a temporary array ***dp[][]*** *in a bottom up manner.*

Algorithm

DP Approach:

- 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.

Printing Parenthesis:

// Prints parenthesization in subexpression (i, j)

- printParenthesis(i, j, bracket[n][n], name)

  //If only one matrix left in current segment

  - if (i == j)
    - print name;
    - name++;
    - return;
  - print "(";

    // Recursively put brackets around subexpression
    // from i to bracket[i][j].

  - printParenthesis(i, bracket[i][j], bracket, name);

    // Recursively put brackets around subexpression
    // from bracket[i][j] + 1 to j.

  - printParenthesis(bracket[i][j]+1, j, bracket, name);
  - print ")";

Code

```c
#include <limits.h>
#include <stdio.h>

int A[100][2];

int Input()
{
    int n;
    printf("Enter number of Matrices.\n");
    scanf("%d", &n);

    printf("Matrix %d: m*n \n", 1);
    printf("Enter m.\n");
    scanf("%d", &A[0][0]);
    printf("Enter n.\n");
    scanf("%d", &A[0][1]);

    for(int i = 1; i<n; i++)
    {
        printf("Matrix %d: %d*n \n", i+1, A[i-1][1]);
        A[i][0] = A[i-1][1];
        printf("Enter n.\n");
        scanf("%d", &A[i][1]);
    }

    return n;
}

void Display(int size, int m[][size])//Accepts a sq matrix
{
    for(int i = 1; i<size; i++)
    {
        for(int j = 1; j<size; j++)
            printf("%d\t", m[i][j]);
        printf("\n");
    }

}
```

```
void printParenthesis(int i, int j, int n, int* bracket, char *
name)
{
    // If only one matrix left in current segment
    if (i == j) {
        printf("%c",(*name));
        (*name)++;
        return;
    }

    printf("(");

    // Recursively put brackets around subexpression from i to
bracket[i][j].
    // "*((bracket+i*n)+j)" is similar to bracket[i][j]

    printParenthesis(i, *((bracket + i * n) + j), n, bracket, name);

    // Recursively put brackets around subexpression from
bracket[i][j] + 1 to j.
    printParenthesis(*((bracket + i * n) + j) + 1, j, n, bracket,
name);
    printf(")");
}

int MatrixChainOrder(int p[], int n)
{

    int m[n][n];
    int brac[n][n];

    int i, j, k, L, q;
    // i, j, k = loop, L = length of chain in loop, q used for min
cost calculation

    for(i = 0; i<n; i++)
    {
        for(j = 0; j<n; j++)
        {
            m[i][j] = 0;
            brac[i][j] = 0;
        }
```

```
    }
    for (i = 1; i < n; i++)
        m[i][i] = 0;
    // L is chain length.
    for (L = 2; L < n; L++) {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j]
                    + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]){
                    m[i][j] = q;
                    brac[i][j] = k;
                }
            }
        }
    }
    Display(n, m);
    printf("\n\n");
    Display(n, brac);

    for(i = n-1; i>2; i--)
    {
        int a = brac[1][i];

    }
    char name = 'A';
    printParenthesis(1, n - 1, n, (int*)brac, &name);
    printf("\n");

    return m[1][n - 1];
}

// Driver  code
int main()
{
    int N = Input();
```

```
    int D[N+1];

    D[0] = A[0][0];

    for(int k = 1; k<N; k++)
    {
        D[k] = A[k][0];
    }

    D[N] = A[N-1][1];

    printf("Minimum number of multiplications is %d ",
           MatrixChainOrder(D, N+1));

    getchar();
    return 0;
}

//4 3 2 4 2 5
```

Output

```
Enter number of Matrices.
4
Matrix 1: m*n
Enter m.
3
Enter n.
2
Matrix 2: 2*n
Enter n.
4
Matrix 3: 4*n
Enter n.
2
Matrix 4: 2*n
Enter n.
5
0       24      28      58
0       0       16      36
0       0       0       40
0       0       0       0


0       1       1       3
0       0       2       3
0       0       0       3
0       0       0       0
((A(BC))D)
Minimum number of multiplications is 58
```

Conclusion

Matrix Chain Multiplication is solved using Dynamic Programming Iterative Approach.