Matrix chain multiplication problem

An example of Dynamic programming

Question-

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

Example:

Input: p[] = {10, 20, 30, 40, 30}

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Then idea used in recursion is:

1. Take the sequence matrices and separate it into 2 subsequences.

2. Find the minimum cost of multiplying out each subsequence.

3. Add these costs together and add in the cost of multiplying the two result matrices.

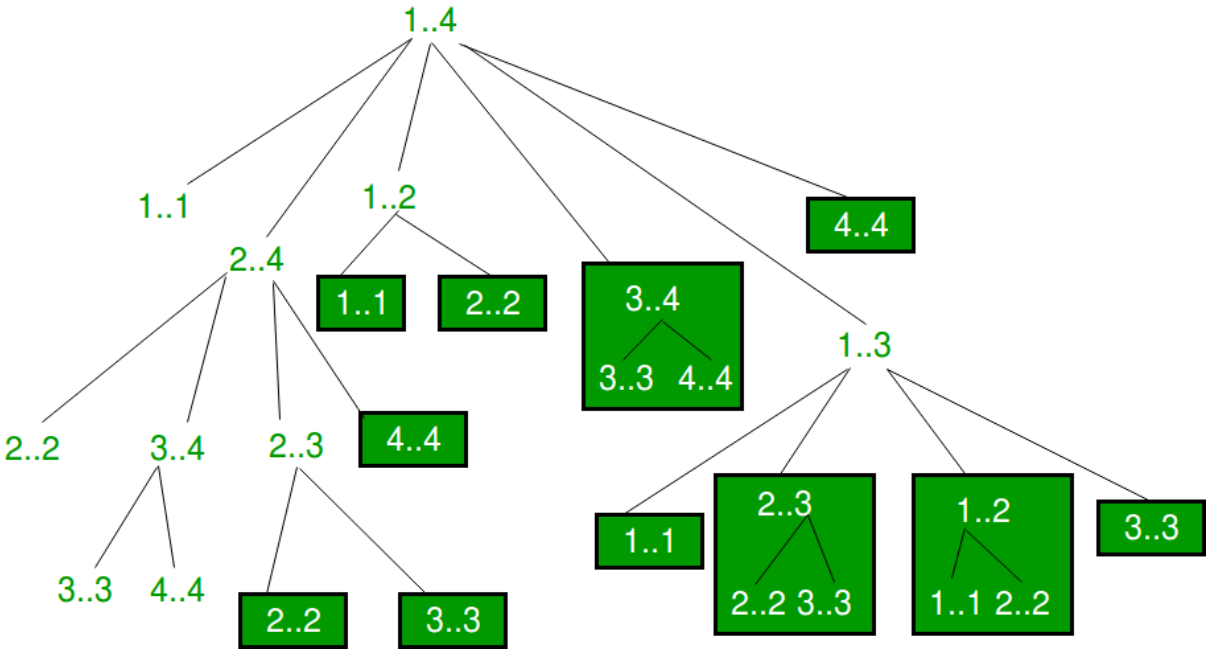
4. Do this for each possible position at which the sequence of matrices can be split and take the minimum over all of them.

Recursion Algorithm:

```
int MatrixChainMultiplication(int dims[], int i, int j)
{
        // base case: one matrix
        if (j <= i + 1)
                return 0;

        // stores minimum number of scalar multiplications (i.e., cost)
        // needed to compute the matrix M[i+1]...M[j] = M[i..j]
        int min = INT_MAX;

        // take the minimum over each possible position at which the
        // sequence of matrices can be split

        /*
                (M[i+1]) x (M[i+2].....M[j])
                (M[i+1]M[i+2]) x (M[i+3.....M[j])
                ...
                ...
                (M[i+1]M[i+2].....M[j-1]) x (M[j])
        */

        for (int k = i + 1; k <= j - 1; k++)
        {
                // recur for M[i+1]..M[k] to get a i x k matrix
                int cost = MatrixChainMultiplication(dims, i, k);

                // recur for M[k+1]..M[j] to get a k x j matrix
                cost += MatrixChainMultiplication(dims, k, j);

                // cost to multiply two (i x k) and (k x j) matrix
                cost +=         dims[i] * dims[k] * dims[j];

                if (cost < min)
                        min = cost;
        }

        // return min cost to multiply M[j+1]..M[j]
        return min;
}
```

The time complexity of the above solution is exponential as we are doing a lot of redundant work. For example the matrix ABCD we will make a recursive call to find the best cost for computing both ABC and AB. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called

two times. We can see that there are many subproblems being called more than once.



Memoization algorithm:

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

    /* For simplicity of the program, one
    extra row and one extra column are
    allocated in m[][]. 0th row and 0th
    column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar
    multiplications needed to compute the
    matrix A[i]A[i+1]...A[j] = A[i..j] where
    dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying
    // one matrix.
    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;
            }
        }
    }

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

The time complexity of above solution is $O(n^3)$.