Dynamic Programming | Set 8 (Matrix Chain Multiplication)

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) = ....
```

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

```
    (AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
    A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
```

Clearly the first parenthesization requires less number of operations.

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.

```
  Input: p[] = {40, 20, 30, 10, 30}
  Output: 26000
  There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
  Let the input 4 matrices be A, B, C and D.  The minimum number of
  multiplications are obtained by putting parenthesis in following way
  (A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

  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

  Input: p[] = {10, 20, 30}
  Output: 6000
  There are only two matrices of dimensions 10x20 and 20x30. So there
```

```
 is only one way to multiply the matrices, cost of which is 10*20*30
```

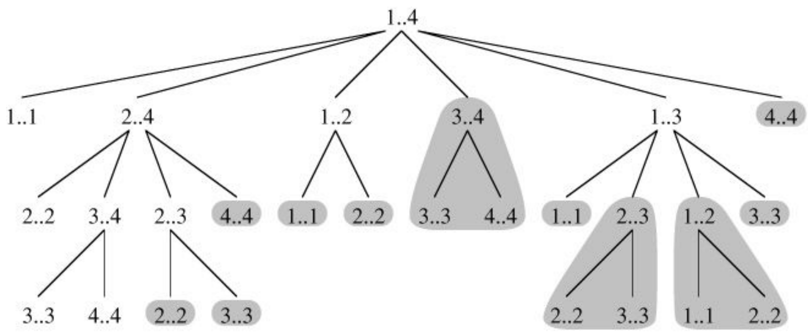
1) Optimal Substructure:
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 way to place first set of parenthesis: A(BCD), (AB)CD and (ABC)D. So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

2) Overlapping Subproblems
Following is a recursive implementation that simply follows the above optimal substructure property.

```
/* A naive recursive implementation that simply follows
 substructure property */
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between fir
    // recursively calculate count of multiplcations for
    // placement and return the minimum count
    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                           MatrixChainOrder(arr, 1, n-1))

    getchar();
    return 0;
}
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. 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.



Since same suproblems are called again, this problem has Overlapping Subprolems property. So Matrix Chain Multiplication problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for Matrix Chain Multiplication problem using Dynamic Programming.

```
// See the Cormen book for details of the following algo
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and 
       allocated in m[][].  0th row and 0th column of m[
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications
       the matrix A[i]A[i+1]...A[j] = A[i..j] where dime
       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[
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

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

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                       MatrixChainOrder(arr, size));

    getchar();
    return 0;
}
```

Time Complexity: O(n^3)
Auxiliary Space: O(n^2)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://en.wikipedia.org/wiki/Matrix_chain_multiplication
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm