Optimal Parenthesization for Matrix Chain Product

Given a chain of N matrices $A_1, A_2, \cdots, A_N$, where for $i = \{1, 2, \cdots, N\}$ matrix A_i has dimension $d_{i-l} \times d_i$, your task is to fully parenthesize the product $A_1 \times A_2 \times \cdots \times A_n$ in a way that minimizes the number of scalar multiplications required to evaluate this expression. Note that evaluating $A_1 \times A_2$ requires $d_0 \times d_1 \times d_2$ scalar multiplications.

For example, the matrix chain product $A_1 \times A_2 \times A_3$ can be parenthesized and evaluated in two possible ways:

1. $((A_1 \times A_2) \times A_3)$ requiring $d_0 \times d_1 \times d_2 + d_0 \times d_2 \times d_3$ scalar multiplications,
2. $(A_1 \times (A_2 \times A_3))$ requiring $d_0 \times d_1 \times d_3 + d_1 \times d_2 \times d_3$ scalar multiplications.

Question 1: Given a list d which represents the chain of matrices such that matrix A_i is of dimension $d[i-1]$ by $d[i]$, write a function, OptProduct, to solve this problem using a top-down dynamic programming approach. **Hint**: in each step, guess what should be outermost multiplication. Then write the base case and the recursive relation.

Subproblems: OptProduct(i, j) the optimal parenthesization of $A_i \cdots A_{j-1}$
Guess: the outermost multiplication

$$(A_i \times \cdots \times A_{k-1}) \times (A_k \times \cdots \times A_{j-1})$$

Recursive Relation:

$$OptProduct(i,j) = \begin{cases} 0, & \text{if } j = i+1 \\ \min_{i+1 \le k \le j-1}\Big(OptProduct(i,k) + OptProduct(k,j) + d[i-1]d[k-1]d[j-1]\Big), & \text{otherwise.} \end{cases}$$

Original Problem: OptProduct(1, len(d))

```python
def OptProduct(i, j, memo=None):
    if memo is None:
        memo = {}
    if (i, j) in memo:
        return memo[(i,j)]
    else:
        if j == i+1:
            cost = 0
        else:
            cost = min([OptProduct(i, k, memo) + OptProduct(k, j, memo) \
                       + d[i-1]*d[j-1]*d[k-1] for k in range(i+1, j)])
        memo[(i,j)] = cost
    return cost
```

Question 2: Analyze the running time of your code.

Let N = len(d). We have:
subproblems: $O(N^2)$
computation time per subproblem: $O(N)$
total running time: $O(N^3)$

Question 3: Write a function, OPTPRODUCTBOTTOMUP, to solve this problem using a bottom-up approach. **Hint**: find out the order in which you must solve the subproblems and determine what needs to be stored in a table.

```
def OptProductBottomUp():
    cost = [[math.inf for i in range(len(d))] for j in range(len(d))]

    for k in range(1, len(d)+1):
        for i in range(len(d)-k):
            if k == 1:
                cost[i][i+k] = 0
                continue
            for j in range(i+1, i+k):
                current_cost = cost[i][j] + cost[j][i+k] + d[i]*d[j]*d[i+k]
                if current_cost < cost[i][i+k]:
                    cost[i][i+k] = current_cost

    return cost[0][len(d)-1]
```

Question 4: Analyze the running time of your code.

The loops are nested three deep, and each loop index (i, j, and k) takes on at most $N-1$ values. Thus, the total running time of this algorithm is $O(N^3)$ and it requires $O(N^2)$ auxiliary space.