I collaborated with:

Problem
If you are familiar with matrix multiplication, you can skip down to **The Problem**; otherwise read the next few paragraphs first.
Given two vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ the *dot product* (a.k.a. *inner product*) of u and v is given by $u \cdot v = u_1v_1 + \ldots + u_nv_n$. Note that it takes n scalar multiplications to compute the dot product of two vectors of length n.
Now suppose that A and B are $n \times q$ and $q \times m$ matrices (arrays), respectively. The *product* of A and B is the $n \times m$ matrix P for which $P[i, j]$ is the dot product of row i of A with column j of B. Note that this is well-defined since the rows of A and the columns of B are all vectors of length q. This operation is referred to as *matrix multiplication*.
If you aren't familiar with matrix multiplication, don't worry. Here are the only things you need to know to answer this question.

- Computing the product AB by directly using the definition above requires nqm scalar multiplications: There are nm entries, and each requires $O(q)$ multiplications to compute.
- In general, $AB \neq BA$: Matrix multiplication is *not* commutative, so don't even try!
- Matrix multiplication is *associative*: ((A B) C) = (A (B C)). That is, a product $A_1 \ldots A_k$ can be parenthesized in any order without changing the value of the result.

The Problem
While the order of evaluation of a chain $A_1 \ldots A_k$ of matrices may not affect the value of the result, it can greatly affect the time required to compute the result! For example. If A, B, C have sizes 3×20, 20×4 and 4×10 respectively, then

Computing $((AB)C)$ takes $3 \cdot 20 \cdot 4$ scalar multiplications for AB, which is a 3×4 matrix, plus $3 \cdot 4 \cdot 10$ scalar multiplications for multiplying AB by C, giving a total of 240 + 120 = 360 scalar multiplications.

Computing $(A(BC))$ takes $20 \cdot 4 \cdot 10$ scalar multiplications for BC, which is a 20×10 matrix, plus $3 \times 20 \times 10$ scalar multiplications for multiplying A by (BC), giving a total of 800 + 600 = 1400 scalar multiplications.

So, consider a product $A_1 \ldots A_k$ of matrices where each A_i has size $r_i \times c_i$. Assume that $c_i = r_{i+1}$ for $i = 1 \ldots n-1$, so that the product of two consecutive matrices is well-defined. We call an order of evaluation (a particular parenthesizing) of the product $A_1 \ldots A_k$ *optimal* if it uses the minimum number of scalar multiplications.

[a] Design a dynamic programming algorithm to compute the number of scalar multiplications in an optimal order of evaluation for $A_1 \ldots A_k$. Justify its correctness. (Hint: Think about the final matrix multiplication that happens in the ordering.)

[b] Determine the time and space complexity of your algorithm.

[c] Describe how you would modify your algorithm to report the (or an) optimal order. What is the complexity of this algorithm.

Solution
We will have to find the final multiplication so that the number of scalar multiplications is minimized. We can think of a start matrix A_s and an end matrix A_t and figure out where we will last multiply. We must check every possible end multiplication of matrices from s to t, and pick the one that minimizes multiplications. The following recurrence relation allows us to do that:
$opt[s, t] = \min_{s \leq k \leq t-1}(opt[s, k] + r_s * c_k * c_t + opt[k + 1, t])$

Proof. Base Case: Let our chain of matrices be A_1. We cannot multiply A_1 with any other matrix, so there are zero scalar multiplications. Since the start matrix and the end matrix are the same, then our algorithm will output zero. Thus, our algorithm holds for this case.
Inductive Hypothesis: Assume our algorithm outputs the minimized number of scalar multiplications of up to k matrices for $k \geq 1$.

Algorithm 1 computes the number of scalar multiplications in an optimal order of evaluation

```
function OPT(s,n)
    if s = n then
        opt[n, n] = 0
    else
        for k from s to n − 1 do
            opt[n] = min(opt[s, k] + r_s * c_k * c_n + opt[k + 1, n])
        end for
    end if
end function
```

Inductive Step: We must show that our algorithm minimizes the number of scalar multiplications for $k + 1$ matrices. We know that there must be a final matrix multiplication that happens in the ordering. This final multiplication can happen anywhere from the first matrix, to the $k + 1$ matrix. We must divide our series of matrices into two and see which are the last two matrices we are multiplying. To do this, we find the most optimal solutions for every possible split of our series. Note that this split creates two series of size less than $k + 1$. From our inductive hypothesis, we know that we have the optimal solution for these sub-problems. We then take the optimal solutions of these two series and add the number of multiplications involved in multiplying the last two martrices which is $r_1 * c_j * c_{k+1}$ where j corresponds to the last matrix of the first sequence. We have combined both halves of our series so that the number of scalar multiplications is minimized. Thus, our algorithm works for all cases. □

b) Time Complexity: Split $O(n)$ * n^2 entries $O(n^2)$ = $O(n^3)$
Space Complexity: $O(n^2)$
c) Once we find a minimum split, we know that we cannot split any longer so we have a pair of matrices we are multiplying. Each grouping of matrices is adding parenthesis, so we just add parenthesis over our recursive calls.
Run TIme: is still $O(n^3)$
Space: $O(n^2)$