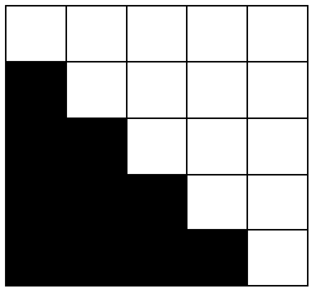
The Algorithm

For our dynamic programming algorithm to solve this problem, we will need two m×m 2-dimensional arrays, of which we will only use one half.



One of them is called M, which stores the minimum runtimes; the other is called S, which stores the multiplication orders.

Each matrix sequence multiplication can be broken down into a multiplication of two sequences. For example,

$$A_1 ((A_2 A_3) A_4)$$

can be broken down into the multiplication of the length one sequence A_1 and the length 3 sequence $((A_2 A_3) A_4)$. $((A_2 A_3) A_4)$ can in turn be turned into the multiplication of $(A_2 A_3)$ and A_4. $(A_2 A_3)$ at last will be the multiplication of two length one sequences A_2 and A_3.

We formulate our problem of multiplying a sequence of matrices starting at matrix i and ending at matrix j as

$$Opt(i, j)$$

. Opt(3, 6) means the optimal runtime and corresponding order of multiplying the sequence from matrix 3 to matrix 6. We want Opt(1,m) for this problem.

The row number of our two 2-dimensional arrays specify the starting matrix, the column number specifies the ending matrix (e.g. M[3,6] is the least runtime possible for multiplying the sequence starting at matrix 3 and ending at matrix 6; S[3,6] is the corresponding optimal order of multiplication). We want M[1,m] and S[1,m] for this problem.

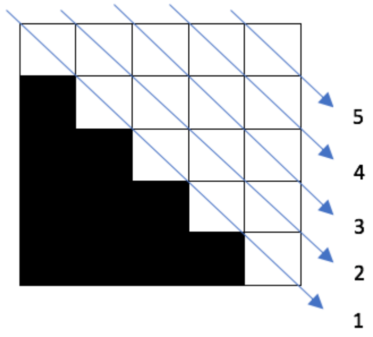
Pseudocode

```
Opt-Matrix-Multiplication:
    M[][] = new matrix[m][m]
    S[][] = new matrix[m][m]

    int diff = 0
    for i = m, m-1, ... 1
        for j = 1, 2, ... i
            starting_matrix = j
            ending_matrix = j+diff
            if (diff == 0) # on the diagonal of the matrices
                M[starting_matrix][ending_matrix] = 0
                S[starting_matrix][ending_matrix] = null
            else
                middle_matrix = a number between starting_matrix (inclusive)
                                                and ending_matrix (exclusive)
                such that

                new_opt = (M[starting_matrix][middle_matrix] +
                           M[middle_matrix + 1][ending_matrix] +
                           (number of rows of starting_matrix *
                           number of columns of middle_matrix *
                           number of columns of ending_matrix))

                is minimized.

                M[starting_matrix][ending_matrix] = new_opt
                S[starting_matrix][ending_matrix] = join(S[starting_matrix][middle_matrix],
                                                         S[middle_matrix+1][ending_matrix])
                                                    .prepend(middle_matrix)
            endif
        endfor
        diff++
    endfor
    minimum_runtime = M[1][m]
    order = S[1][m]
```

This part:

```
for i = m, m-1, ... 1
    for j = 1, 2, ... i
                .......
    diff++
```

graphically represented is this:



Proof of Correctness

Observe that the optimal multiplication ordering of a sequence of matrices of length > 1 is broken into two sequences at an optimal spot. Call the last matrix of the first sequence the middle matrix. A sequence of length three can be broken down into

$$A_1 (A_2 A_3)$$

or

$$(A_1 A_2) A_3$$

.

And $(A_1 A_2)$ is broken down into the multiplication of single matrices A_1 and A_2. Opt(a single matrix A_1) = A_1, which is the base case.
Each of the two optimally broken sequences can in turn be broken into their own optimal two sequences. Done recursively, this looks like a tree. The runtime of multiplying the two sequences is the sum of the runtimes from each sequence plus the runtime of multiplying these two sequences, which is the product of the number of rows of the starting matrix and the number of columns of the middle matrix and the number of columns of the ending matrix. The recursion can be summarized as

$$\mathrm{Opt}(\mathrm{matrix}_{\mathrm{start}}, \mathrm{matrix}_{\mathrm{end}}) = \begin{cases} 0, \qquad if\ matrix_{end} = matrix_{end} \\ \min\limits_{\substack{matrix_{start} \leq \\ matrix_{middle} < \\ matrix_{end}}} \begin{pmatrix} Opt(matrix_{start}, matrix_{middle}) + \\ Opt(matrix_{middle} + 1, matrix_{end}) + \\ number\ of\ rows\ of\ matrix_{start} * \\ number\ of\ columns\ of\ matrix_{middle} * \\ number\ of\ columns\ of\ matrix_{end} \end{pmatrix} \end{cases}$$

And the order of multiplications is the combined orders of the two sequences. We could store the index of the matrix that breaks up each sequence for each subproblem. When two sequences are combined, we combine each of their breaks and add the new break that results from combing them. So

$$A_1 (A_2 A_3)$$

Would have the break order as

$$[1,2]$$

Because the two sequences A_1 has break order [] (it's empty), and (A_2 A_3) has break order [2] because the sequence breaks after A_2. Combining them makes [2]. Add in the newest break after A_1, the solution is [1,2].

This is formulated as

$$\mathrm{Opt}-\mathrm{Order}(\mathrm{matrix}_{\mathrm{start}}, \mathrm{matrix}_{\mathrm{end}}) = \begin{cases} null, \qquad if\ matrix_{end} = matrix_{end} \\ join(Opt - Order(matrix_{start}, matrix_{middle}), \\ Opt - Order(matrix_{middle} + 1, matrix_{end})) \\ .prepend(matrix_{middle}) \\ \\ otherwise \\ (using\ the\ optimal\ matrix_{middle}\ as\ shown) \end{cases}$$

We still now prove by **strong induction** that the above returns the optimal runtime.

Base Case

We want to prove M[1][1] is the optimal runtime and S[1][1] describes the optimal ordering if there were only one matrix. The optimal runtime for one matrix should be 0 and the optimal ordering should be null because there is no multiplication going on. This is all true by definition.

Inductive Case

We need to prove M[i][j] = Opt(i, j) and S[i][j] describes the corresponding multiplication order. Since we're using strong induction, we can assume for all a < i and for all b < j, M[a][b] is the optimal runtime for the sequence starting at matrix a and ending at matrix b, and S[a][b] describes the optimal ordering of said sequence. Also observe that because of the order in which we fill in the cells, M[i][middle] and M[middle+1][j] is always already filled in: middle < j, so M[i][middle] is a cell in the same row as M[i][j] but an earlier column, thus already filled in;

and because middle $\geq$ i, M[middle+1][j] is at a row below M[i][j] but in the same column, thus also already filled in.

$$
\begin{aligned}
&M[i][j] = M[i][middle\_matrix] + \\
&\ M[middle\_matrix + 1][j] \\
&(number\ of\ rows\ of\ i * \\
&number\ of\ columns\ of\ middle\_matrix * \\
&number\ of\ columns\ of\ j) \\
\\
&= Opt(i, middle\_matrix) + \\
&Opt(middle\_matrix + 1, j) + \\
&(number\ of\ rows\ of\ i * \\
&number\ of\ columns\ of\ middle\_matrix * \\
&number\ of\ columns\ of\ j)
\end{aligned}
$$

Which was our recursive definition given above.

S keeps track of the order by recording the optimal breaks in each sequence. When you have a sequence from matrix i to matrix j broken up by middle_matrix, the order of the optimal orderings of the left and right sequences stay the same, so you combine their breaks. The break that happened at the current level also needs to be accounted for, so it needs to be added to the front because it is the most recent break.

Runtime Analysis

Since we update each of the upper half of our 2-dimensional array once, that would be O(n^2). Getting the optimal middle_matrix that breaks the sequence for each of those cells is O(n) because you could do a linear search for a middle_matrix that minimizes the recurrence. Altogether, it is **O(n^3)**.