Solution Review: The Matrix Chain Multiplication

In this lesson, we will solve the matrix chain multiplication problem with different techniques of dynamic programming.

We'll cover the following

Solution 1: Simple recursion

```
import numpy as np
def minMultiplications(dims):
    if len(dims) <= 2:
        return 0
    minimum = np.inf
    for i in range(1,len(dims)-1):
        minimum = min(minimum, minMultiplications(dims[0:i+1]) + minMultiplications(dims[i:]) +
                    dims[0] * dims[-1] * dims[i])
    return minimum

print(minMultiplications([3, 3, 2, 1, 2]))
```

Explanation

We can easily imagine `dims` to be a list of matrices. Then each recursive call tries to find the optimal placement of two parentheses between these matrices. So, let's

say we have the following sequence of matrices: $A_1 A_2 A_3 A_4$. Each recursive call tries to place two parentheses, so we have the following possibilities:

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

Thus, we make recursive calls for all these possibilities and finally choose the best one from among them.

Look at the following visualization for more clarification.

minMultiplications([3, 3, 2, 1, 2])

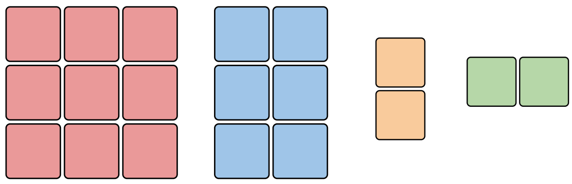
minMultiplications([3, 3, 2, 1, 2])

1 of 16

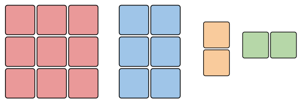
minMultiplications([3, 3, 2, 1, 2])

Let's build a sequence of matrices out of the given dimensions minMultiplications([3, 3, 2, 1])

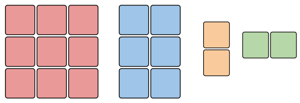
dims:	3	3	2	1	2

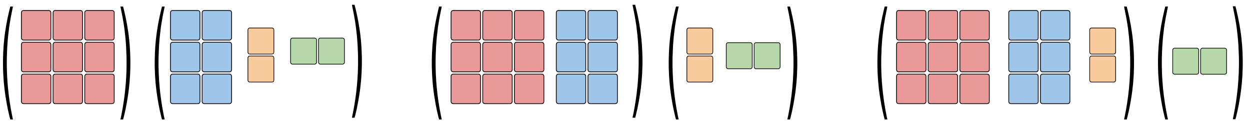


Let's build a sequence of matrices out of the given dimensions

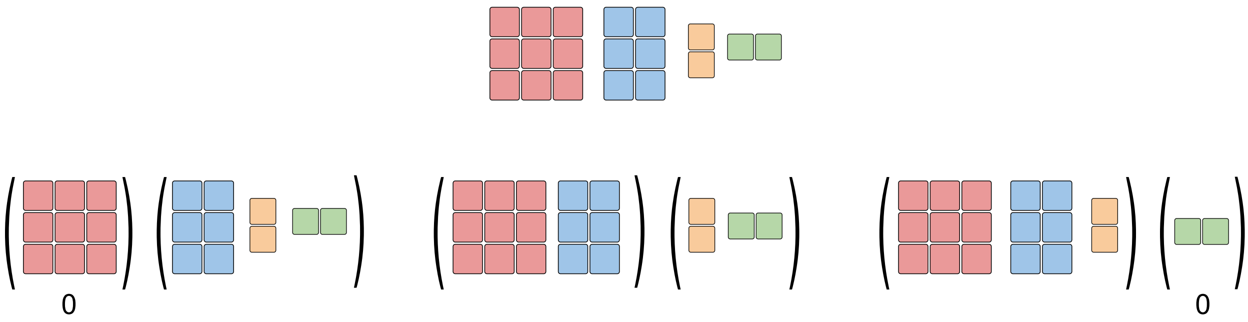


Let's run our algorithm on this sequence of matrices now

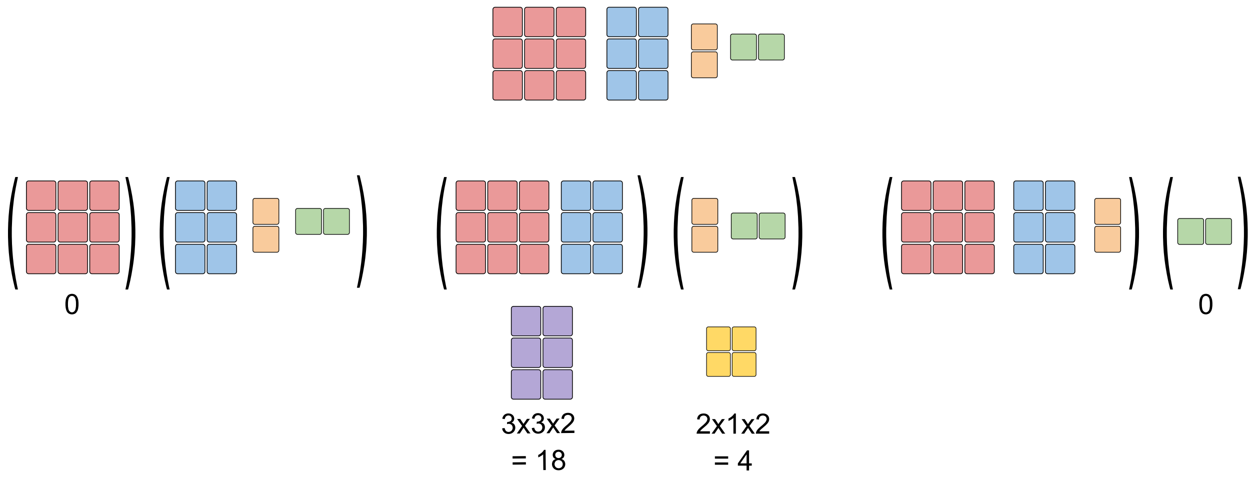




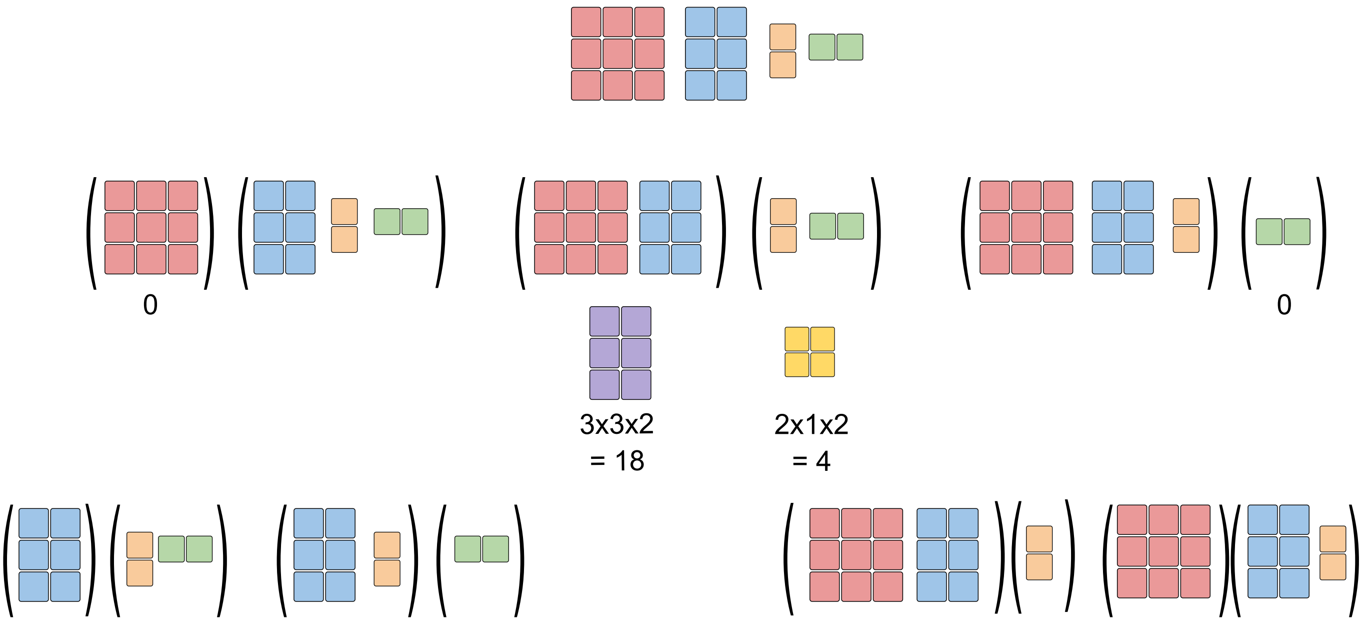
As we discussed there are three ways to put parentheses in a chain of four matrices



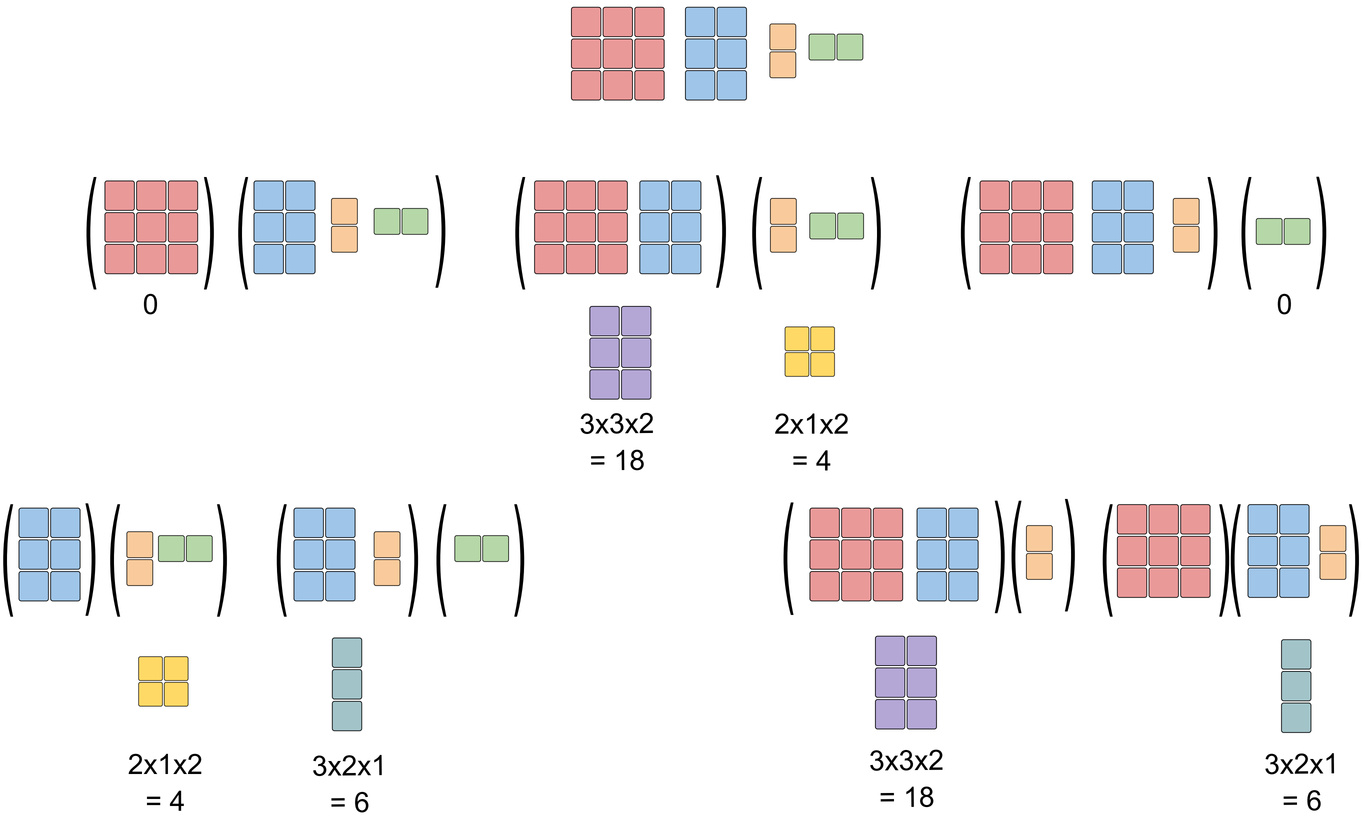
For the single matrices in parentheses, there will be no multiplication so we can return 0



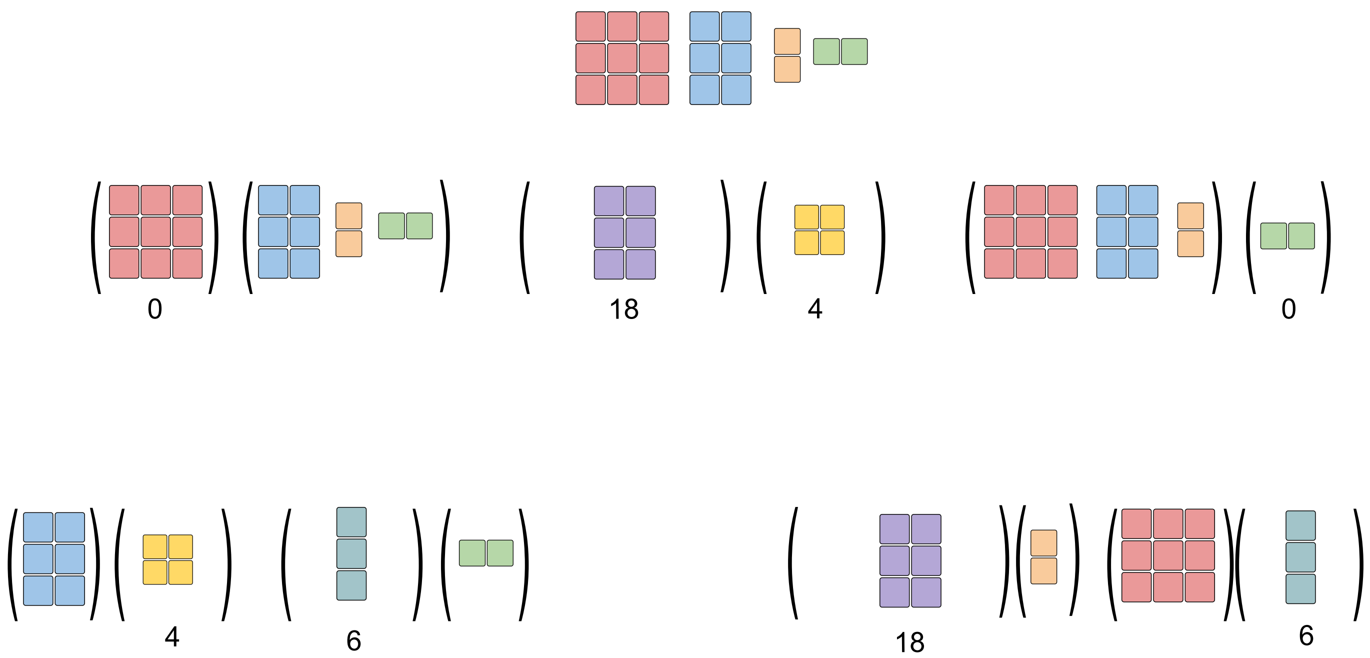
For the cases, when there are only 2 matrices, we can calculate number of primitive multiplications



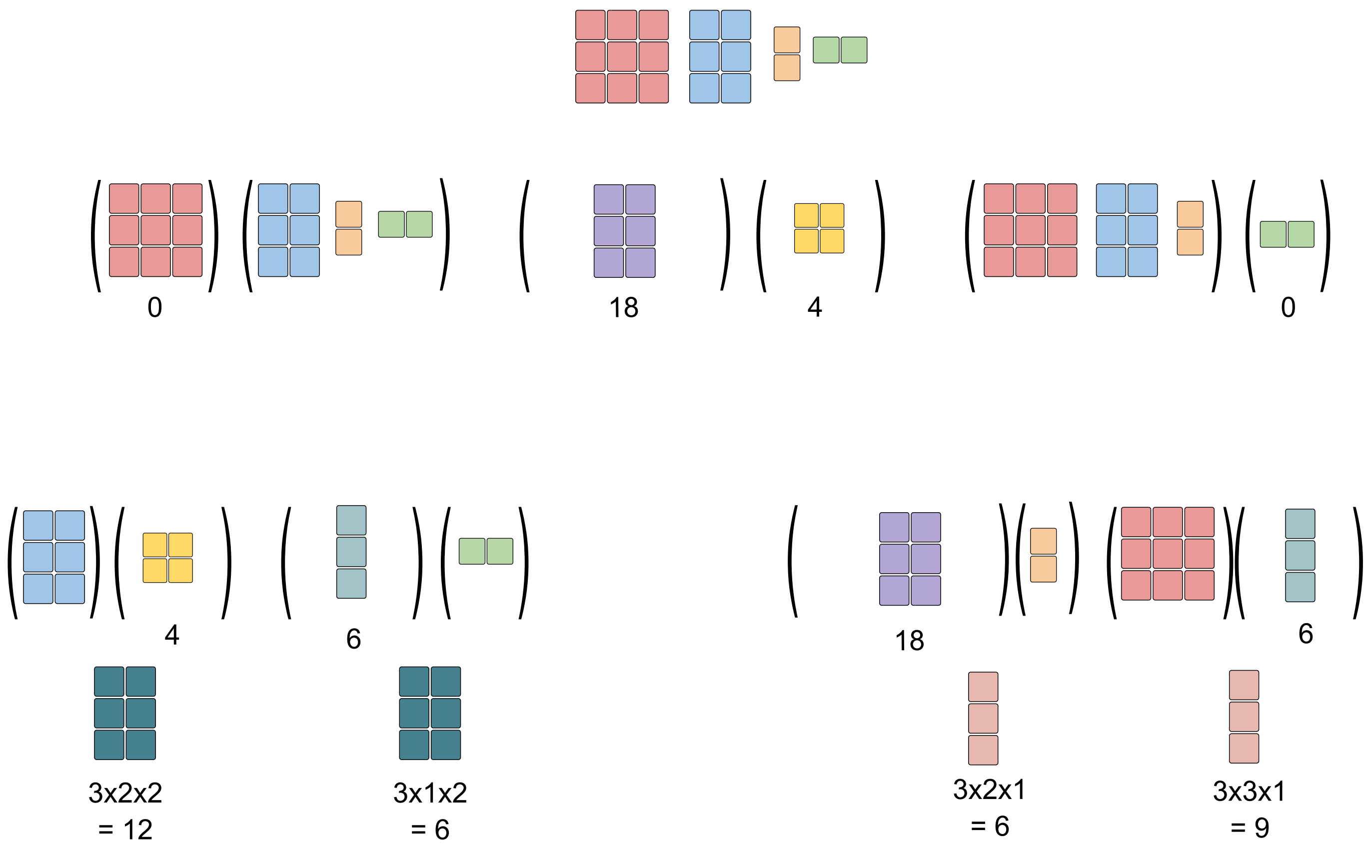
For the other cases, we can make recursive calls again



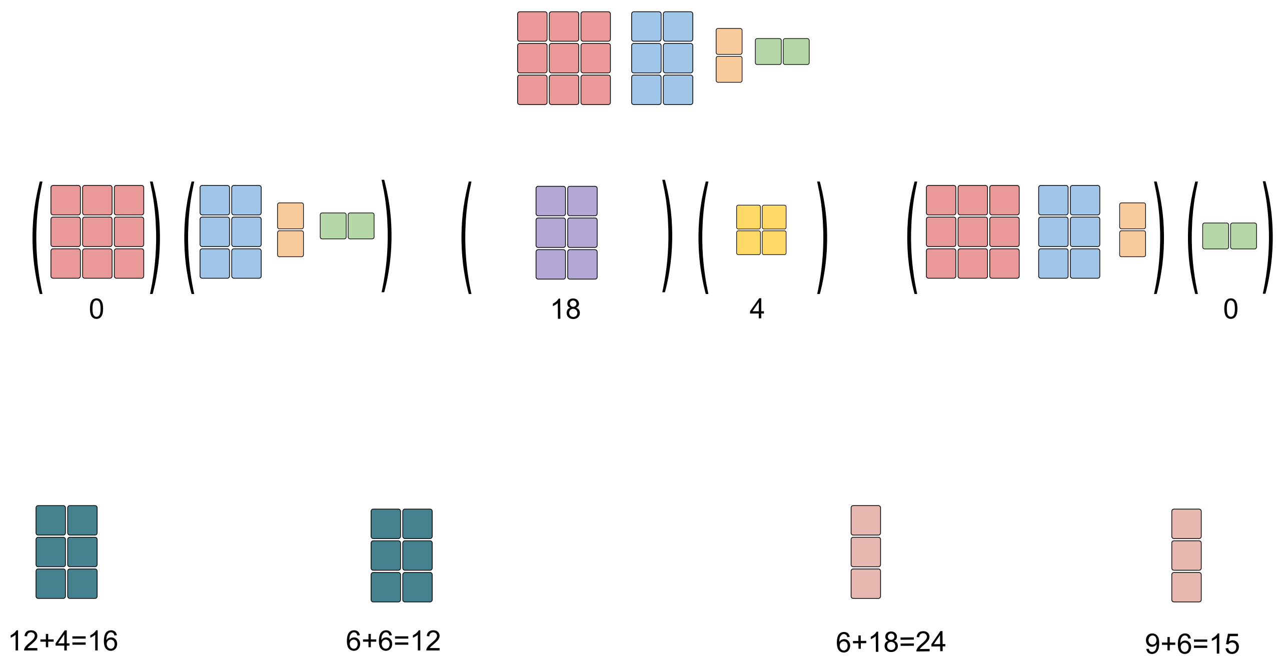
And by finding number of multiplications, we have completed recursive step. Now let's do backtracking



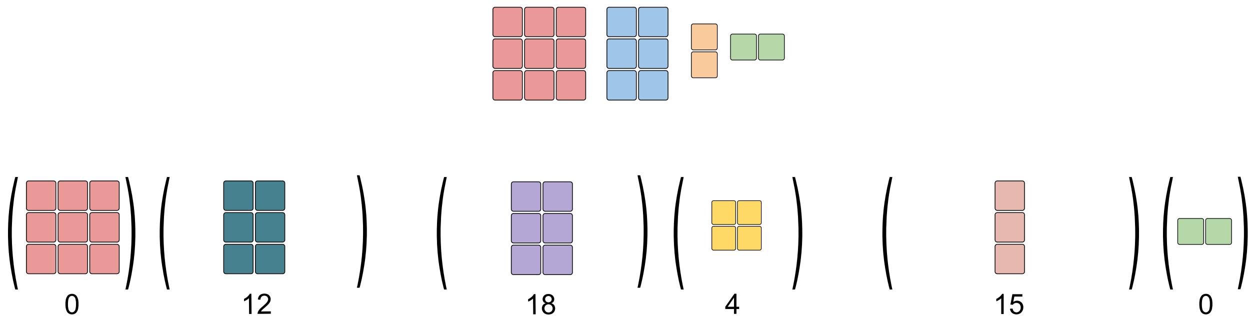
At each step we will choose the recursive call with least number of multiplications. If there is only one option we don't have a choice.



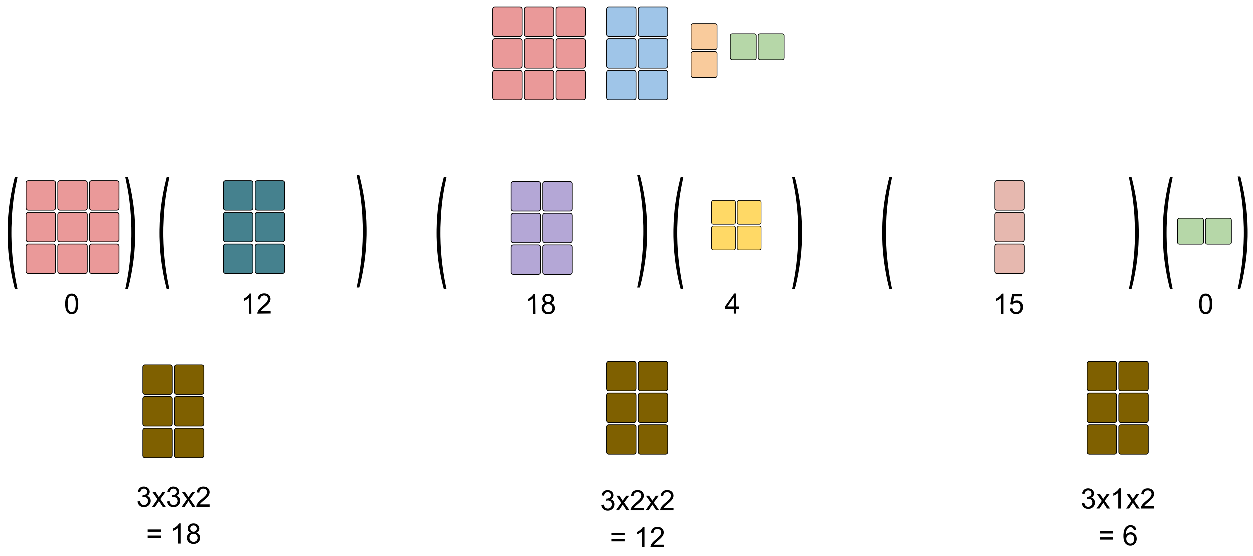
Calculate the number of multiplications is multiplying matrices returned from recursive calls, when returning add recursive calls results too



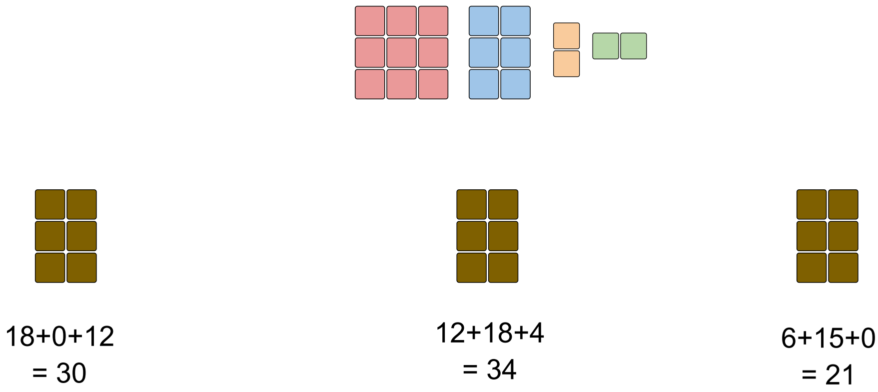
Calculate the number of multiplications is multiplying matrices returned from recursive calls, when returning add recursive calls results too



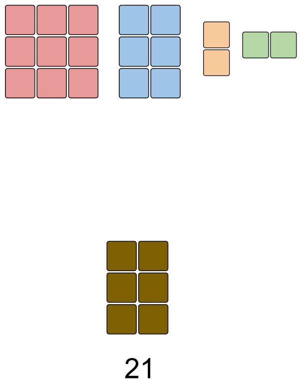
From the results of recursive calls choose the one with minimum multiplications



Again calculate the number of multiplications is multiplying matrices returned from recursive calls, when returning add recursive calls results too



Again calculate the number of multiplications is multiplying matrices returned from recursive calls, when returning add recursive calls results too



Thus total least number of multiplications required to multiply this sequence is 21

16 of 16

Time complexity

At each position, we have the choice to place or not place the parentheses. This way, the total number of possible arrangements is bound by *O(2^n)*. At each point, we have to do this at n places, so the overall time complexity becomes *O($n2^n$)*. However, since we are using the indexing operation to get the subarrays of `dims` in *line 7-8*, this operation has a time complexity of *O(n)*. The overall time complexity would be **O(n^2 2^n)**.

We can easily reduce the time complexity to a simple *O($n2^n$)* if we use two variables, `i` and `j`, as an abstraction to using indexing for getting a subarray. Using `i` and `j` instead of indexing would also make memoization easy for us. Thus, look at the following implementation of the algorithm without using the

Thus, look at the following implementation of the algorithm without using the

indexing operation of Python. Everything else is the same, we have just used additional variables `i` and `j` to avoid *O(n)* indexing operation.

```
import numpy as np
def minRecursive(dims, i, j):
    if j-i <= 2:
        return 0
    minimum = np.inf
    for k in range(i+1, j-1):
        minimum = min(minimum, minRecursive(dims, i, k+1) + minRecursive(dims, k, j) +
                      dims[i]*dims[j-1]*dims[k])
    return minimum
def minMultiplications(dims):
    return minRecursive(dims, 0, len(dims))

print(minMultiplications([3, 3, 2, 1, 2]))
```

The time complexity of this algorithm as we discussed earlier is $\mathbf{O(n2^n)}$.

Solution 2: Top-down dynamic programming

Let's see how this problem satisfies both conditions of dynamic programming

Optimal substructure

For solving this problem for n matrices:

$A_1A_2A_3....An$

If we know the optimal solutions to following subproblems, we just need to take the minimum of the results:

A_1 and $A_2A_3A_4....A_n$

A_1A_2 and $A_3A_4A_5....A_n$

$A_1A_2A_3$ and $A_4A_5A_6....A_n$

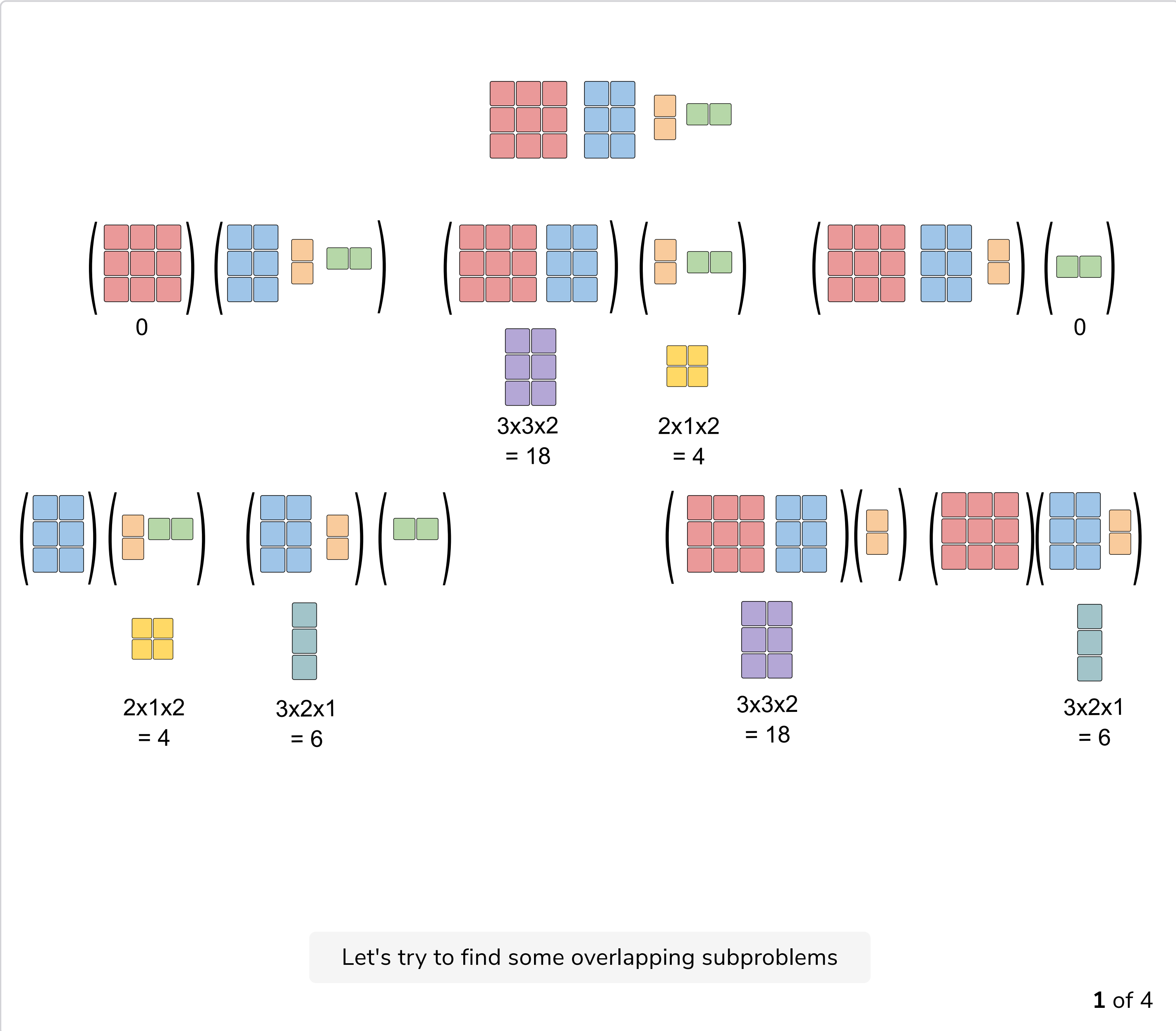
and so on.

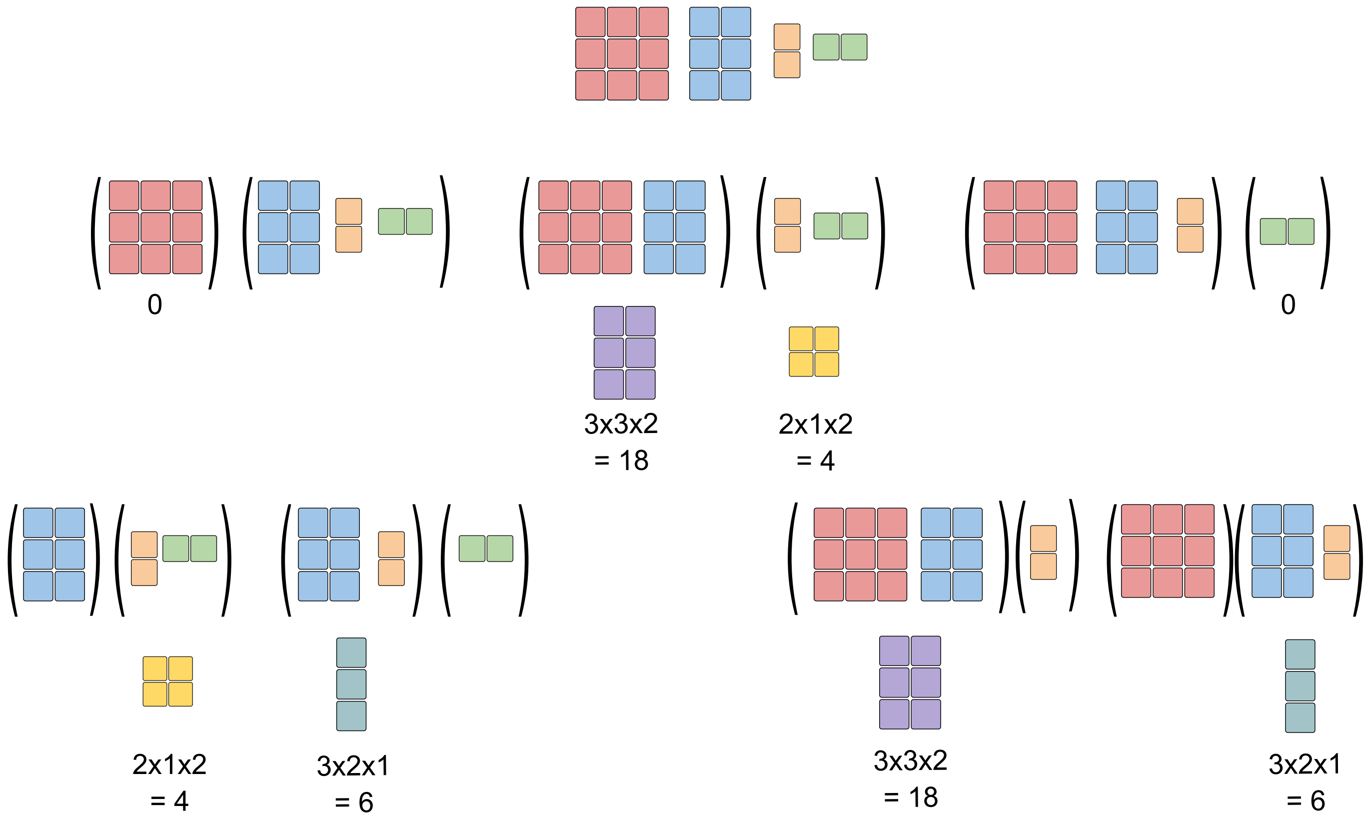
Thus, this problem obeys the property of optimal substructure.

Overlapping subproblems

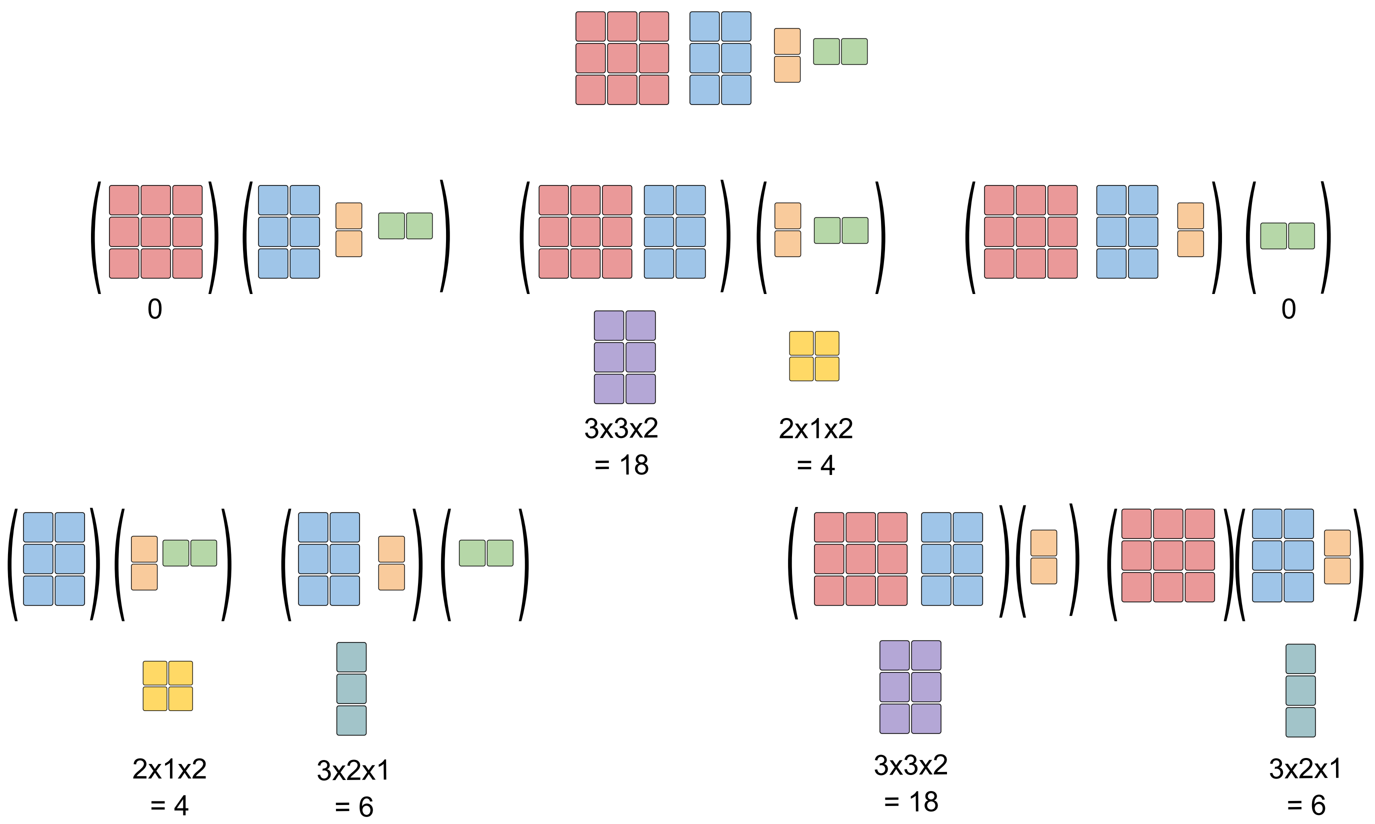
You can see in the above example of only four matrices we have many repeating subproblems. This would increase exponentially as we increase the number of matrices.

Below is a visualization to show overlapping subproblems.

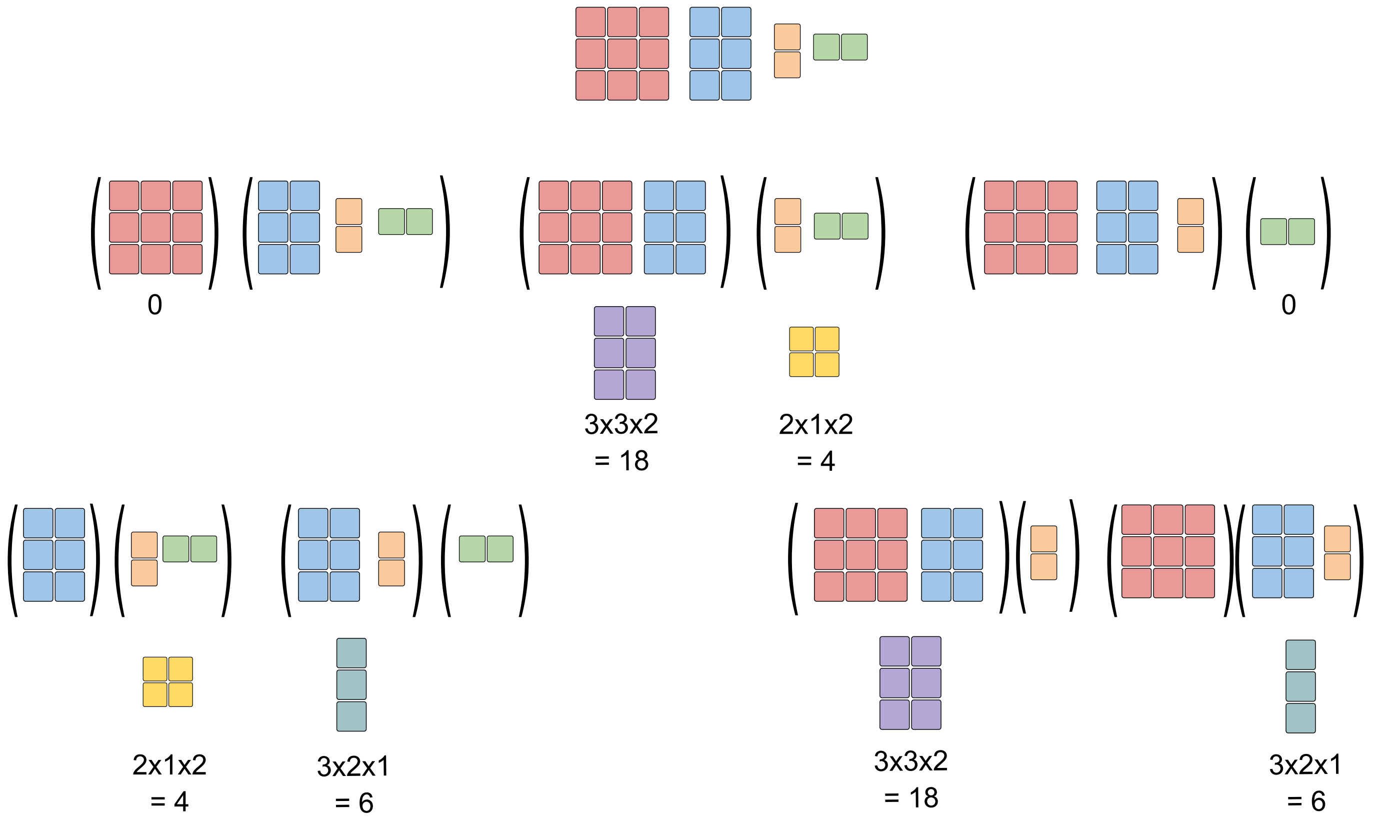




One subproblem repeating twice



Another subproblem repeating twice



Repeating computations due to overlapping subproblems.

4 of 4

```
import numpy as np

def minRecursive(dims, i, j, memo):
    if j-i <= 2:
        return 0
    if (i,j) in memo:
        return memo[(i,j)]
    minimum = np.inf
    for k in range(i+1, j-1):
        minimum = min(minimum, minRecursive(dims, i, k+1, memo) + minRecursive(dims, k, j, memo) +
                      dims[i]*dims[j-1]*dims[k])
    memo[(i,j)] = minimum
    return minimum

def minMultiplications(dims):
    memo = {}
    return minRecursive(dims, 0, len(dims), memo)

print(minMultiplications([3, 3, 2, 1, 2]))
```

Explanation

Well, we have seen this many times now. We are simply storing all our evaluated results in the `memo` table and looking it up before the evaluation. The important bit is our choice of key for memoization, which we have created by using a tuple of `i` and `j`. Since `i` and `j` can uniquely identify a subarray from `dims`, a tuple of these two variables fits perfectly for the key. If we had used an indexing approach to get the subarray, we wouldn't be able to memoize our results. This is because lists cannot be used as a key in dictionaries.

Time and space complexity

Every recursive call loops over the list of `dims` which is *O(n)*. How many calls will we have? We need to see how many `i` - `j` pairs there are. The number of `i` - `j` pairs would be bound by $O(n^2)$. Thus, the overall time complexity would be $\mathbf{O(n^3)}$. Since we store $O(n^2)$ pairs in `memo`, the space complexity would be $\mathbf{O(n^2)}$.

Solution 3: Bottom-up dynamic programming

```
import numpy as np

def minMultiplications(dims):

    dp = [[0 for _ in range(len(dims))] for _ in range(len(dims))]

    for l in range(2,len(dims)):
        for i in range(1,len(dims)-l+1):
            j = i+l-1
            dp[i][j] = np.inf
            for k in range(i, j):
                temp = dp[i][k]+ dp[k+1][j] + dims[i-1]*dims[k]*dims[j]
                if temp < dp[i][j]:
                    dp[i][j] = temp
    return dp[1][-1]
print(minMultiplications([3, 3, 2, 1, 2]))
```

Explanation

This might look like a rather daunting problem, but if you look at it from the perspective of the bottom-up approach, the ideas are similar to any other problem we've gone over so far. Let's discuss the layout of the `dp` table, it is a 2-d list of dimensions $n \times n$ (where n is the length of `dims`), where `dp[i][j]` for any `i`, `j` $< n$, denotes the minimum number of multiplications required to multiply a chain of matrices formed between `i` and `j`. Now, we need to find a sequence in which we fill the `dp` table so that no value is needed before it is evaluated. We know from our previous solution that we should start from the base case of a single matrix's multiplication. This we have covered in our initialization of `dp` by setting everything to 0 (*line 5*). Next, we need to handle the cases for multiplications of the chains of size 2, since these will be used by all the bigger problems. After 2, we will need to fill for chains of size 3, and so on (*lines 7 - 14*). The nested for loops are simply calculating the optimal answer in the same way as the previous solutions, i.e., by finding the minimum cumulative value from all the subproblems (*lines 11 - 14*).

Let's look at a visualization of this algorithm.

minMultiplications([3, 3, 2, 1, 2])

minMultiplications([3, 3, 2, 1, 2])

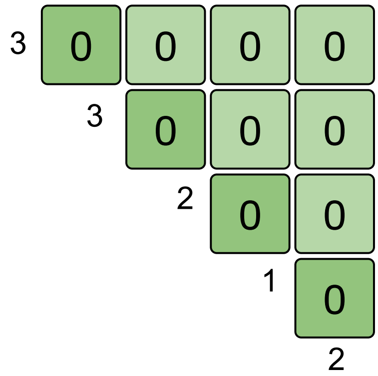
1 of 16

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

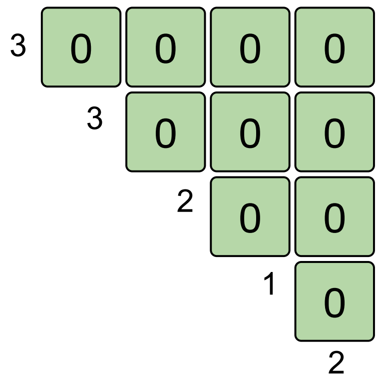
Let's build dp table out of this

3	0	0	0	0
	3	0	0	0
		2	0	0
			1	0
				2

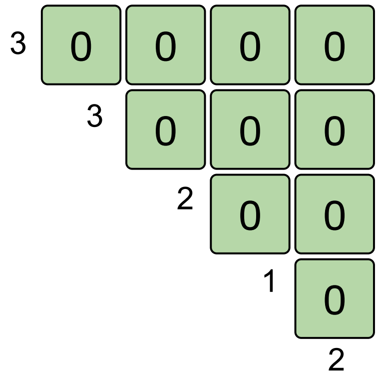
We only need one triangular half of this matrix so let's ignore the other half, we also don't require the first row, We have also annotated dims.



Every diagonal starting from left, denotes a step size (l). first diagonal represents a step size of 1, i.e. multiplication of a chain containing a single matrix, therefore it will be 0

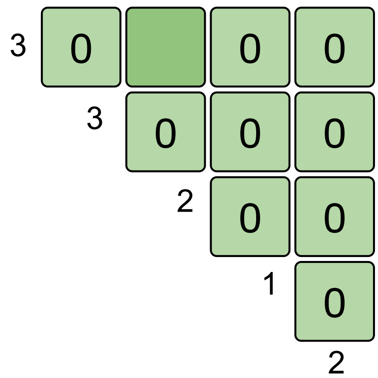


From the next diagonal we will start filling as follows: choose the division that results in smallest number of multiplications

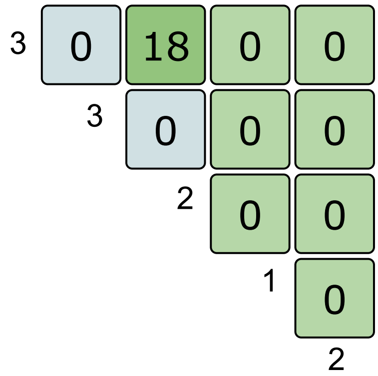


Number of multiplications in multiplying two chains
given by i,j,k
= dp[i][j] + dp[j+1][k] + dims[i]*dims[j]*dims[k]

The number of multiplications for multiplying two matrix chain given given by cuts I,j,k

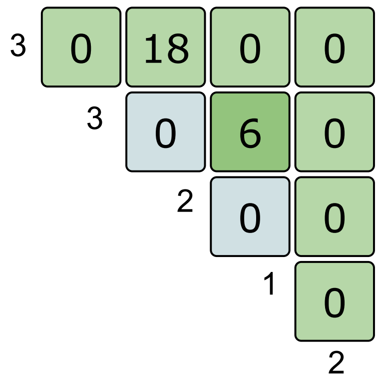


dp[1][2]



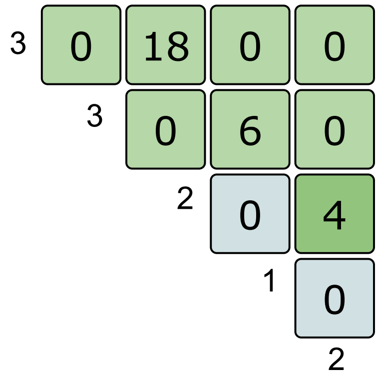
dp[1][2] = dp[1][1] + dp[2][2] + 3x3x2
dp[1][2] = 18

dp[1][2]



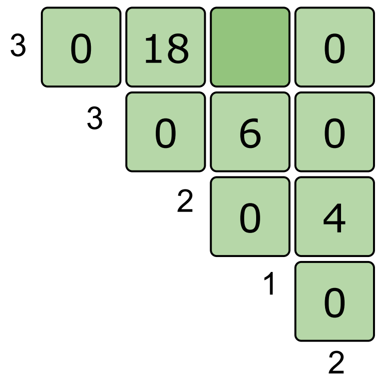
dp[2][3] = dp[2][2] + dp[3][3] + 3x2x1
dp[2][3] = 6

dp[2][3]

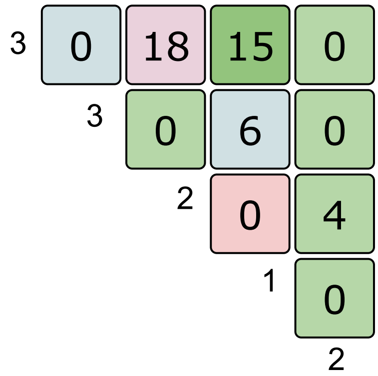


dp[3][4] = dp[3][3] + dp[4][4] + 2x1x2
dp[3][4] = 4

dp[3][4]

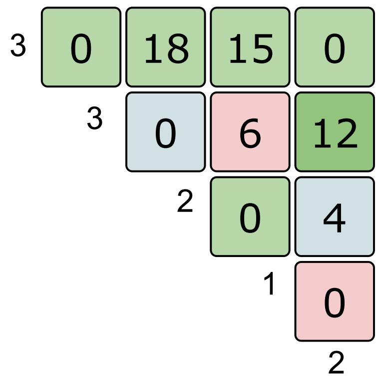


Moving on to the step size of 3



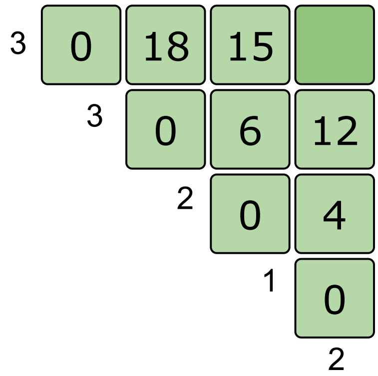
dp[1][3] = min{dp[1][1] + dp[2][3] + 3x3x1,
dp[1][2] + dp[3][3] + 3x2x1}
dp[3][4] = min{15, 24} = 15

dp[1][3]

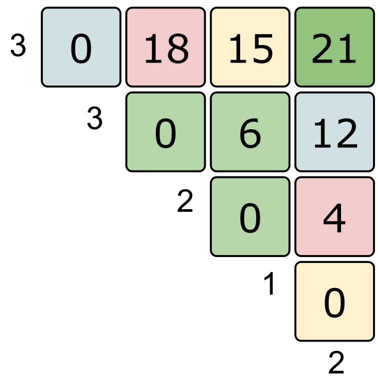


dp[2][4] = min{dp[2][2] + dp[3][4] + 3x2x2,
dp[2][3] + dp[4][4] + 3x1x2}
dp[3][4] = min{16, 12} = 12

dp[1][3]

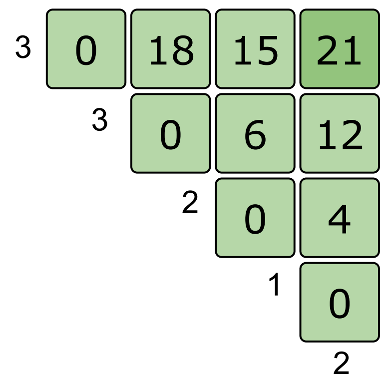


Moving on to the step size of 4



dp[1][4] = min{dp[1][1] + dp[2][4] + 3x3x2,
dp[1][2] + dp[3][4] + 3x2x2,
dp[1][3] + dp[4][4] + 3x1x2}
dp[1][4] = min{30, 34, 21} = 21

dp[1][4]



Thus optimal number of multiplications is 21

16 of 16

Time and space complexity

The time complexity would be $\mathbf{O(n^3)}$. There are n number of steps, for each step we have to evaluate problems bounded by $O(n)$. And each of these problems might depend on n other subproblems. Thus, we get $\mathbf{O(n^3)}$. The space complexity is $\mathbf{O(n^2)}$ as we can see in the visualization also.

Since a problem can depend on any of the $O(n^2)$ subproblems, we cannot reduce space complexity below this bound.

In the next lesson, we will work on another interesting dynamic programming problem.