In this article, we're going to be looking at the problem [Maximum Score from Performing Multiplication Operations](https://leetcode.com/problems/maximum-score-from-performing-multiplication-operations/). We can tell this is a DP problem because it is asking for a maximum score, and every time we choose to use a number from nums, it affects all future possibilities. Let's solve this problem with the framework:

1. A **function or array** that answers the problem for a given state

Since we're doing top-down, we need to decide on two things for our function dp. What state variables we need to pass to it, and what it will return. We are given two input arrays:  nums and multipliers. The problem says we need to do m operations, and on the *ith* operation, we gain score equal to multipliers[i] times a number from either the left or right end of nums, which we remove after the operation. That means we need to know 3 things for each operation:

1. How many operations have we done so far; this tells us what number from multipliers we will be using?
2. The index of the leftmost number remaining in nums.
3. The index of the rightmost number remaining in nums.

We can use one state variable i, to indicate how many operations we have done so far, which means multipliers[i] is the current multiplier to be used. For the leftmost number remaining in nums, we can use another state variable,  left, that indicates how many left operations we have done so far. If we have done, say 3 left operations, if we were to do another left operation we would use nums[3] (because nums is 0-indexed). We can say the same thing for the rightmost remaining number - let's use a state variable right that indicates how many right operations we have done so far.

It may seem like we need all 3 of these state variables, but we can formulate an equation for one of them using the other two. If we know how many elements we have picked from the leftside, left, and we know how many elements we have picked in total,  i, then we know that we must have picked i - left elements from the rightside. The original length of nums is n, which means the index of the rightmost element is right = n - 1 - (i - left). Therefore, we only need 2 state variables:  i and left, and we can calculate right inside the function.

Now that we have our state variables, what should our function return? The problem is asking for the maximum score from some number of operations, so let's have our function dp(i, left) return the maximum possible score if we have already done i total operations and used left numbers from the left side. To answer the original problem, we should return  dp(0, 0).

Table

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

Text, application

Description automatically generated with medium confidence

2. A **recurrence relation** to transition between states

At each state, we have to perform an operation. As stated in the problem description, we need to decide whether to take from the left end (nums[left]) or the right end (nums[right]) of the current nums. Then we need to multiply the number we choose by multipliers[i], add this value to our score, and finally remove the number we chose from nums. For implementation purposes, "removing" a number from nums means incrementing our state variables i and left so that they point to the next two left and right numbers.

Let mult=multipliers[i] and right = nums.length - 1 - (i - left). The only decision we have to make is whether to take from the left or right of nums.

* If we choose left, we gain mult⋅nums[left] points from this operation. Then, the next operation will occur at  (i + 1, left + 1).  i gets incremented at every operation because it represents how many operations we have done, and left gets incremented because it represents how many left operations we have done. Therefore, our total score is mult⋅nums[left] + dp(i + 1, left + 1).
* If we choose right, we gain mult⋅nums[right] points from this operation. Then, the next operation will occur at (i + 1, left). Therefore, our total score is mult⋅nums[right] + dp(i + 1, left).

Since we want to maximize our score, we should choose the side that gives more points. This gives us our recurrence relation:

dp(i, left) = max(

mult⋅nums[left] + dp(i + 1, left + 1), mult⋅nums[right] + dp(i + 1, left)

)

Where mult⋅nums[left]+dp(i + 1, left + 1) represents the points we gain by taking from the left end of nums plus the maximum points we can get from the remaining nums array and mult⋅nums[right]+dp(i + 1, left) represents the points we gain by taking from the right end of nums plus the maximum points we can get from the remaining nums array.

3. **Base cases**

The problem statement says that we need to perform m operations. When i equals m, that means we have no operations left. Therefore, we should return 0.

### op-down Implementation

Let's put the 3 parts of the framework together for a solution to the problem.

Protip: for Python, the [functools](https://docs.python.org/3/library/functools.html) module provides super handy tools that automatically memoize a function for us. We're going to use the @lru\_cache decorator in the Python implementation.

If you find yourself needing to memoize a function in an interview and you're using Python, check with your interviewer if using modules like functools is OK.

This particular problem happens to have very tight time limits. For Java, instead of using a hashmap for the memoization, we will use a 2D array. For Python, we're going to limit our cache size to 2000.

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

### Bottom-up Implementation

In the bottom-up implementation, the array works the same way as the function from top-downdp[i][left] represents the max score possible if i operations have been performed and left left operations have been performed.

Earlier in the explore card, we learned that while bottom-up is typically faster than top-down, it is often harder to implement. This is because the order in which we iterate needs to be precise. You'll see in the implementations below that we use the same math to calculate right, and the same recurrence relation but we need to iterate backwards starting from m (because the base case happens when i equals m). We also need to initialize dp with one extra row so that we don't go out of bounds in the first iteration of the outer loop.

Text

Description automatically generated

Text, letter

Description automatically generated

The time and space complexity of both implementations is  *O*(*m^2*) where m is the length of multipliers. We will talk about more in depth about time and space complexity at the end of this chapter.

Up Next

Try the next two problems on your own. The first one is a very classical computer science problem and popular in interviews. If you get stuck, come back here for hints:

**1143. Longest Common Subsequence**

Let dp(i, j) represent the longest common subsequence between the string text1 up to index i and text2 up to index j.

If text1[i] == text2[j], then we should use this character, giving us 1 + dp(i - 1, j - 1). Otherwise, we can either move one character back from text1, or 1 character back from text2. Try both.

If i or j becomes less than 0, then we're out of bounds and should return 0.

**221. Maximal Square**

dp[row][col] represent the largest possible square whose bottom right corner is on matrix[row][col].

Any square with a \text{0}0 cannot have a square on it, and should be ignored. Otherwise, let's say you had a 3x3 square. Look at the bottom right corner of this 3x3 square. What do the squares above, to the left, and to the up-left of this square have in common? All 3 of those squares are the bottom-right square of a square that is (at least) 2x2.

Just make sure to stay in bounds.