<https://leetcode.com/problems/remove-boxes/description/>

You are given several boxes with different colors represented by different positive numbers.

You may experience several rounds to remove boxes until there is no box left. Each time you can choose some continuous boxes with the same color (i.e., composed of k boxes, k >= 1), remove them and get k \* k points.

Return *the maximum points you can get*.

**Example 1:**

Input: boxes = [1,3,2,2,2,3,4,3,1]

Output: 23

Explanation:

[1, 3, 2, 2, 2, 3, 4, 3, 1]

----> [1, 3, 3, 4, 3, 1] (3\*3=9 points)

----> [1, 3, 3, 3, 1] (1\*1=1 points)

----> [1, 1] (3\*3=9 points)

----> [] (2\*2=4 points)

**Example 2:**

Input: boxes = [1,1,1]

Output: 9

**Example 3:**

Input: boxes = [1]

Output: 1

**Constraints:**

* 1 <= boxes.length <= 100
* 1 <= boxes[i] <= 100

**Attempt 1: 2023-09-06**

**Solution 1: Native DFS (360 min, TLE 20/63)**

class Solution {

public int removeBoxes(int[] boxes) {

int n = boxes.length;

return helper(boxes, 0, n - 1, 0);

}

private int helper(int[] boxes, int i, int j, int k) {

if(i > j) {

return 0;

}

int result = (k + 1) \* (k + 1) + helper(boxes, i + 1, j, 0);

for(int m = i + 1; m <= j; m++) {

if(boxes[i] == boxes[m]) {

result = Math.max(result, helper(boxes, i + 1, m - 1, 0) + helper(boxes, m, j, k + 1));

}

}

return result;

}

}

Time Complexity: O(N!)

Space Complexity: O(N!)

**Solution  2: DFS + Memoization (10 min)**

class Solution {

public int removeBoxes(int[] boxes) {

// dp[i][j] means the maximum points we can get by removing

// subarray boxes[i,j] (both inclusive).

// -> dp[i][j][k] means the maximum points possible by removing

// the boxes of subarray boxes[i,j] with k boxes attached to

// its left of the same color as boxes[i]

int n = boxes.length;

Integer[][][] memo = new Integer[n][n][n];

return helper(boxes, 0, n - 1, 0, memo);

}

private int helper(int[] boxes, int i, int j, int k, Integer[][][] memo) {

if(i > j) {

return 0;

}

if(memo[i][j][k] != null) {

return memo[i][j][k];

}

int result = (k + 1) \* (k + 1) + helper(boxes, i + 1, j, 0, memo);

for(int m = i + 1; m <= j; m++) {

if(boxes[i] == boxes[m]) {

result = Math.max(result, helper(boxes, i + 1, m - 1, 0, memo) + helper(boxes, m, j, k + 1, memo));

}

}

return memo[i][j][k] = result;

}

}

Time Complexity: O(N^3 ~ N!)

Space Complexity: O(N^3)

**Solution  3: DP (??? min, not able to reproduce individually on this one !!!)**

class Solution {

public int removeBoxes(int[] boxes) {

// dp[i][j] means the maximum points we can get by removing

// subarray boxes[i,j] (both inclusive).

// -> dp[i][j][k] / T(i, j, k) which denotes the maximum points

// possible by removing the boxes of subarray boxes[i, j] with

// k boxes attached to its left of the same color as boxes[i].

int n = boxes.length;

int[][][] dp = new int[n][n][n];

// T(i, i, k) = (k + 1) \* (k + 1): only one box left in the

// subarray but we've already got k boxes of the same color

// attached to its left, so the total number of boxes of the

// same color is (k + 1) and the maximum point is (k + 1) \* (k + 1).

for(int i = 0; i < n; i++) {

for(int k = 0; k <= i; k++) {

dp[i][i][k] = (k + 1) \* (k + 1);

}

}

for(int window = 1; window < n; window++) {

for(int j = window; j < n; j++) {

int i = j - window;

// Not very understand on why range of k can be select

// between [0,i] ? How we sure that we have k at most i ?

// Because we assume all k boxes attached to ith box left

// has same color as ith box (which also initialized in

// previous for loop section)?

for(int k = 0; k <= i; k++) {

int result = (1 + k) \* (1 + k) + dp[i + 1][j][0];

for(int m = i + 1; m <= j; m++) {

if(boxes[i] == boxes[m]) {

result = Math.max(result, dp[i + 1][m - 1][0] + dp[m][j][k + 1]);

}

}

dp[i][j][k] = result;

}

}

}

return n == 0 ? 0 : dp[0][n - 1][0];

}

}

Time Complexity: O(N^4)

Space Complexity: O(N^3)

**Refer to**

<https://leetcode.com/problems/remove-boxes/solutions/101310/java-top-down-and-bottom-up-dp-solutions/>

Since the input is an array, let's begin with the usual approach by breaking it down with the original problem applied to each of the subarrays.

Let the input array be boxes with length n. Define T(i, j) as the maximum points one can get by removing boxes of the subarray boxes[i, j] (both inclusive). The original problem is identified as T(0, n - 1) and the termination condition is as follows:

* T(i, i - 1) = 0: no boxes so no points.
* T(i, i) = 1: only one box left so the maximum point is 1.

Next let's try to work out the recurrence relation for T(i, j). Take the first box boxes[i](i.e., the box at index i) as an example. What are the possible ways of removing it? (Note: we can also look at the last box and the analyses turn out to be the same.)

If it happens to have a color that you dislike, you'll probably say "I don't like this box so let's get rid of it now". In this case, you will first get 1 point for removing this poor box. But still you want maximum points for the remaining boxes, which by definition is T(i + 1, j). In total your points will be 1 + T(i + 1, j).

But later after reading the rules more carefully, you realize that you might get more points if this box (boxes[i]) can be removed together with other boxes of the same color. For example, if there are two such boxes, you get 4 points by removing them simultaneously, instead of 2 by removing them one by one. So you decide to let it stick around a little bit longer until another box of the same color (whose index is m) becomes its neighbor. Note up to this moment all boxes from index i + 1 to index m - 1 would have been removed. So if we again aim for maximum points, the points gathered so far will be T(i + 1, m - 1). What about the remaining boxes?

At this moment, the boxes we left behind consist of two parts: the one at index i (boxes[i]) and those of the subarray boxes[m, j], with the former bordering the latter from the left. Apparently there is no way applying the definition of the subproblem to the subarray boxes[m, j], since we have some extra piece of information that is not included in the definition. **In this case, I shall call that the definition of the subproblem is not self-contained and its solution relies on information external to the subproblem itself**.

Another example of problem that does not have self-contained subproblems is [leetcode 312. Burst Balloons](https://leetcode.com/problems/burst-balloons/" \l "/description), where the maximum coins of subarray nums[i, j] depend on the two numbers adjacent to nums[i] on the left and to nums[j] on the right. So you may find some similarities between these two problems.

Problems without self-contained subproblems usually don't have well-defined recurrence relations, which renders it impossible to be solved recursively. The cure to this issue can sound simple and straightforward: **modify the definition of the problem to absorb the external information so that the new one is self-contained**.

So let's see how we can redefine T(i, j) to make it self-contained. First let's identify the external information. On the one hand, from the point of view of the subarray boxes[m, j], it knows nothing about the number (denoted by k) of boxes of the same color as boxes[m]to its left. On the other hand, given this number k, the maximum points can be obtained from removing all these boxes is fixed. Therefore the external information to T(i, j) is this k. Next let's absorb this extra piece of information into the definition of T(i, j) and redefine it as T(i, j, k) which denotes the maximum points possible by removing the boxes of subarray boxes[i, j] with k boxes attached to its left of the same color as boxes[i]. Lastly let's reexamine some of the statements above:

* Our original problem now becomes T(0, n - 1, 0), since there is no boxes attached to the left of the input array at the beginning.
* The termination conditions now will be: **a**. T(i, i - 1, k) = 0: no boxes so no points, and this is true for any k (you can interpret it as nowhere to attach the boxes).**b**. T(i, i, k) = (k + 1) \* (k + 1): only one box left in the subarray but we've already got k boxes of the same color attached to its left, so the total number of boxes of the same color is (k + 1) and the maximum point is (k + 1) \* (k + 1).
* The recurrence relation is as follows and the maximum points will be the larger of the two cases: **a**. If we remove boxes[i] first, we get (k + 1) \* (k + 1) + T(i + 1, j, 0) points, where for the first term, instead of 1 we again get (k + 1) \* (k + 1) points for removing boxes[i] due to the attached boxes to its left; and for the second term there will be no attached boxes so we have the 0 in this term. **b**. If we decide to attach boxes[i] to some other box of the same color, say boxes[m], then from our analyses above, the total points will be T(i + 1, m - 1, 0) + T(m, j, k + 1), where for the first term, since there is no attached boxes for subarray boxes[i + 1, m - 1], we have k = 0 for this part; while for the second term, the total number of attached boxes for subarray boxes[m, j] will increase by 1 because apart from the original k boxes, we have to account for boxes[i]now, so we have k + 1 for this term. But we are not done yet. What if there are multiple boxes of the same color as boxes[i] within subarray boxes[i + 1, j]? We have to try each of them and choose the one that yields the maximum points. Therefore the final answer for this case will be: max(T(i + 1, m - 1, 0) + T(m, j, k + 1)) where i < m <= j && boxes[i] == boxes[m].

Before we get to the actual code, it's not hard to discover that there is overlapping among the subproblems T(i, j, k), therefore it's qualified as a DP problem and its intermediate results should be cached for future lookup. Here each subproblem is characterized by three integers (i, j, k), all of which are bounded, i.e, 0 <= i, j, k < n, so a three-dimensional array (n x n x n) will be good enough for the cache.

Finally here are the two solutions, one for top-down DP and the other for bottom-up DP. From the bottom-up solution, the time complexity will be O(n^4) and the space complexity will be O(n^3).

**Top-down DP:**

public int removeBoxes(int[] boxes) {

int n = boxes.length;

int[][][] dp = new int[n][n][n];

return removeBoxesSub(boxes, 0, n - 1, 0, dp);

}

private int removeBoxesSub(int[] boxes, int i, int j, int k, int[][][] dp) {

if (i > j) return 0;

if (dp[i][j][k] > 0) return dp[i][j][k];

int i0 = i, k0 = k; // Need to record the intial values of i and k in order to apply the following optimization

for (; i + 1 <= j && boxes[i + 1] == boxes[i]; i++, k++); // optimization: all boxes of the same color counted continuously from the first box should be grouped together

int res = (k + 1) \* (k + 1) + removeBoxesSub(boxes, i + 1, j, 0, dp);

for (int m = i + 1; m <= j; m++) {

if (boxes[i] == boxes[m]) {

res = Math.max(res, removeBoxesSub(boxes, i + 1, m - 1, 0, dp) + removeBoxesSub(boxes, m, j, k + 1, dp));

}

}

dp[i0][j][k0] = res; // When updating the dp matrix, we should use the initial values of i, j and k

return res;

}

**Bottom-up DP:**

public int removeBoxes(int[] boxes) {

int n = boxes.length;

int[][][] dp = new int[n][n][n];

for (int j = 0; j < n; j++) {

for (int k = 0; k <= j; k++) {

dp[j][j][k] = (k + 1) \* (k + 1);

}

}

for (int l = 1; l < n; l++) {

for (int j = l; j < n; j++) {

int i = j - l;

for (int k = 0; k <= i; k++) {

int res = (k + 1) \* (k + 1) + dp[i + 1][j][0];

for (int m = i + 1; m <= j; m++) {

if (boxes[m] == boxes[i]) {

res = Math.max(res, dp[i + 1][m - 1][0] + dp[m][j][k + 1]);

}

}

dp[i][j][k] = res;

}

}

}

return (n == 0 ? 0 : dp[0][n - 1][0]);

}

**Side notes**: In case you are curious, for the problem "**leetcode 312. Burst Balloons**", the external information to subarray nums[i, j]is the two numbers (denoted as left and right) adjacent to nums[i] and nums[j], respectively. If we absorb this extra piece of information into the definition of T(i, j), we have T(i, j, left, right)which represents the maximum coins obtained by bursting balloons of subarray nums[i, j]whose two adjacent numbers are left and right. The original problem will be T(0, n - 1, 1, 1)and the termination condition is T(i, i, left, right) = left \* right \* nums[i]. The recurrence relations will be: T(i, j, left, right) = max(left \* nums[k] \* right + T(i, k - 1, left, nums[k]) + T(k + 1, j, nums[k], right))where i <= k <= j(here we interpret it as that the balloon at index k is the last to be burst. Since all balloons can be the last one so we try each case and choose one that yields the maximum coins). For more details, refer to [dietpepsi 's post](https://discuss.leetcode.com/topic/30746/share-some-analysis-and-explanations).

**Refer to**

<https://leetcode.com/problems/remove-boxes/solutions/1402561/c-java-python-top-down-dp-clear-explanation-with-picture-clean-concise/>

**Idea**

* This is really a hard problem, so feel free to enjoy it if you can't solve this problem. I will try my best to explain in the easiest way.
* Let dp(l, r, k) denote the maximum points we can get in boxes[l..r] if we have extra k boxes which is the same color with boxes[l] in the left side.

For example: boxes = [3, 3, 1, 3, 3]

The dp(l=3, r=4, k=2) is the maximum points we can get in boxes[3..4] if we have extra 2 boxes the same color with boxes[3] in the left side, it's the same as we find the maximum points in boxes = [3, 3, 3, 3].

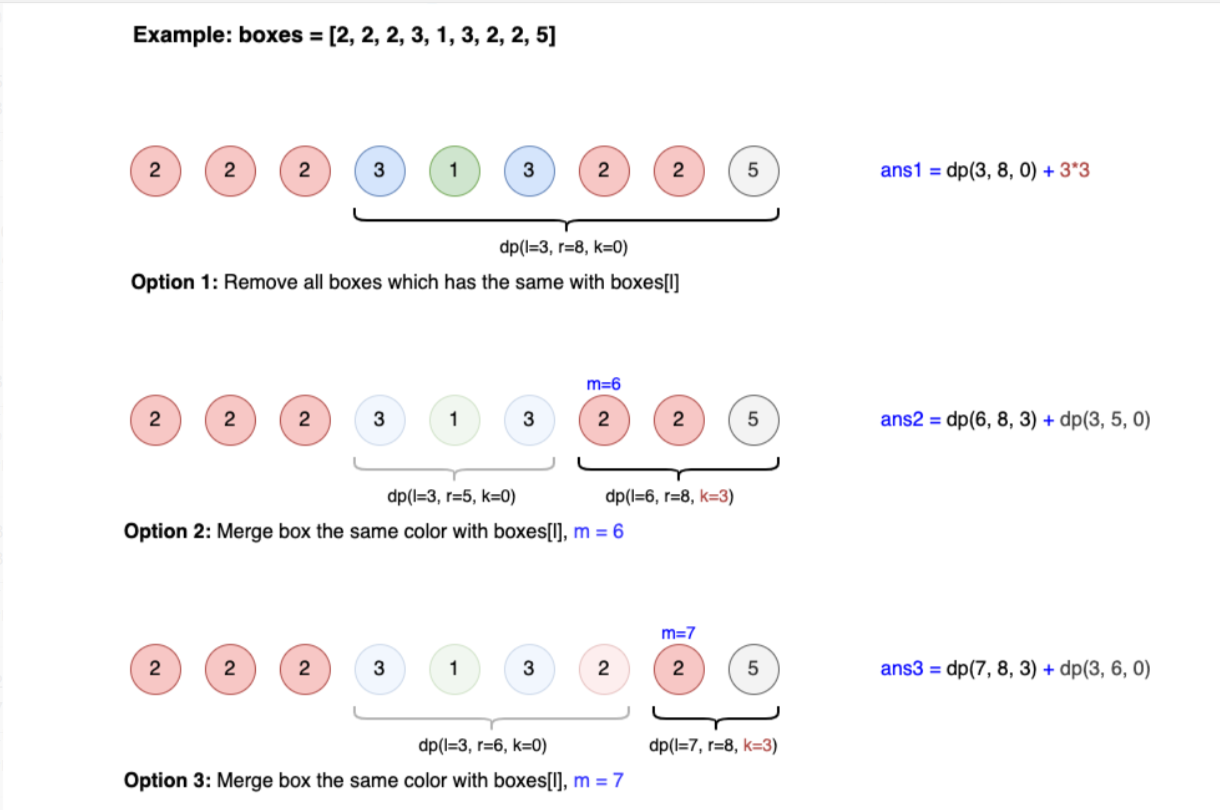
* Since (a+b)^2 > a^2 + b^2, where a > 0, b > 0, so it's better to greedy to remove all contiguous boxes of the same color, instead of split them.
* So we increase both l and k while boxes[l+1] == boxes[l].
* Now, we have many options to consider:

1. Option 1, remove all boxes which has the same with boxes[l], total points we can get is dp(l+1, r, 0) + (k+1)\*(k+1) (k left boxes and the lth box have the same color)
2. Other options, we try to merge non-contiguous boxes of the same color together, by:

Find the index j, where l+1 <= j <= r so that boxes[j] == boxes[l].

Total points we can get is dp(j, r, k+1) + dp(l+1, j-1, 0)

* Choose the option which has the maximum score we can get.



class Solution {

int[][][] memo;

public int removeBoxes(int[] boxes) {

int n = boxes.length;

memo = new int[n][n][n];

return dp(boxes, 0, n - 1, 0);

}

int dp(int[] boxes, int l, int r, int k) {

if (l > r) return 0;

if (memo[l][r][k] > 0) return memo[l][r][k];

int lOrg = l, kOrg = k;

while (l+1 <= r && boxes[l] == boxes[l+1]) { // Increase both `l` and `k` if they have consecutive colors with `boxes[l]`

l += 1;

k += 1;

}

int ans = (k+1) \* (k+1) + dp(boxes, l+1, r, 0); // Remove all boxes which has the same with `boxes[l]`

for (int m = l+1; m <= r; ++m) // Try to merge non-contiguous boxes of the same color together

if (boxes[m] == boxes[l])

ans = Math.max(ans, dp(boxes, m, r, k+1) + dp(boxes, l+1, m-1, 0));

return memo[lOrg][r][kOrg] = ans;

}

}

**Complexity**

* Time: O(N^4), where N <= 100 is length of boxes array.
* Explain: There is total N^3 dp states, they are dp[0][0][0], dp[0][0][1],..., dp[n][n][n], each state needs a loop O(N) to compute the result. So total complexity is O(N^4).
* Space: O(N^3)

**Refer to**

<https://grandyang.com/leetcode/546/>

刚开始看这道题的时候，感觉跟之前那道 [Zuma Game](http://www.cnblogs.com/grandyang/p/6759881.html) 很像，于是就写了一个暴力破解的方法，结果 TLE 了。无奈之下只好上网搜大神们的解法，又看了 fun4LeetCode 大神写的帖子，之前那道 [Reverse Pairs](http://www.cnblogs.com/grandyang/p/6657956.html) 就是参考的 fun4LeetCode 大神的帖子，惊为天人，这次又是这般精彩，大神请收下我的膝盖。那么下面的解法就大部分参考 [fun4LeetCode 大神的帖子](https://discuss.leetcode.com/topic/84687/java-top-down-and-bottom-up-dp-solutions)来讲解吧。在之前帖子 [Reverse Pairs](http://www.cnblogs.com/grandyang/p/6657956.html) 的讲解中，大神归纳了两种重现模式，这里也试着看能不能套用上。对于这种看来看去都没思路的题来说，抽象建模的能力就非常的重要了。对于题目中的具体场景啊，具体代表的东西都可忽略不看，这样能帮助我们接近问题的本质，这道题的本质就是一个数组，每次消去一个或多个数字，并获得相应的分数，让求最高能获得的分数。而之前那道 [Zuma Game](http://www.cnblogs.com/grandyang/p/6759881.html) 也是给了一个数组，让往某个位置加数字，使得相同的数字至少有3个才能消除，二者是不是很像呢，但是其实解法却差别很大。那道题之所以暴力破解没有问题是因为数组的长度和给定的数字个数都有限制，而且都是相对较小的数，那么即便遍历所有情况也不会有太大的计算量。而这道题就不一样了，既然不能暴力破解，那么对于这种玩数组和子数组的题，刷题老司机们都会优先考虑用动态规划 Dynamic Programming 来做吧。既然要玩子数组，肯定要限定子数组的范围，那么至少应该是个二维的 dp 数组，其中 dp[i][j] 表示在子数组 [i, j] 范围内所能得到的最高的分数，那么最后返回 dp[0][n-1] 就是要求的结果。

那么对于 dp[i][j]，如果移除 boxes[i] 这个数字，那么总得分应该是 1 + dp[i+1][j]，但是通过分析题目中的例子，能够获得高积分的 trick 是，移除某个或某几个数字后，如果能使得原本不连续的相同数字变的连续是坠好的，因为同时移除的数字越多，那么所得的积分就越高。那么假如在 [i, j] 中间有个位置m，使得 boxes[i] 和 boxes[m] 相等，那么就不应该只是移除 boxes[i] 这个数字，而是还应该考虑直接移除 [i+1, m-1] 区间上的数，使得 boxes[i] 和 boxes[m] 直接相邻，那么获得的积分就是 dp[i+1][m-1]，那么剩余了什么，boxes[i] 和 boxes[m, j] 区间的数，此时无法处理子数组 [m, j]，因为有些信息没有包括在 dp 数组中， **此类的题目归纳为不自己包含的子问题，其解法依赖于一些子问题以外的信息** 。这类问题通常没有定义好的重现关系，所以不太容易递归求解。为了解决这类问题， **我们需要修改问题的定义，使得其包含一些外部信息，从而变成自包含子问题** 。

那么对于这道题来说，无法处理 boxes[m, j] 区间是因为其缺少了关键信息，我们不知道 boxes[m] 左边相同数字的个数k，只有知道了这个信息，那么m的位置才有意义，所以 dp 数组应该是一个三维数组 dp[i][j][k]，表示区间 [i, j] 中能获得的最大积分，当 boxes[i] 左边有k个数字跟其相等，那么目标就是要求 dp[0][n-1][0] 了，而且也能推出 dp[i][i][k] = (1+k) \* (1+k) 这个等式。那么来推导重现关系，对于 dp[i][j][k]，如果移除 boxes[i]，那么得到 (1+k)\*(1+k) + dp[i+1][j][0]。对于上面提到的那种情况，当某个位置m，有 boxes[i] == boxes[m] 时，也应该考虑先移除 [i+1,m-1] 这部分，得到积分 dp[i+1][m-1][0]，然后再处理剩下的部分，得到积分 dp[m][j][k+1]，这里k加1点原因是，移除了中间的部分后，原本和 boxes[m] 不相邻的 boxes[i] 现在相邻了，又因为二者值相同，所以k应该加1，因为k的定义就是左边相等的数字的个数。讲到这里，那么DP方法最难的状态转移方程也就得到了，那么代码就不难写了，需要注意的是，这里的 C++ 的写法不能用 vector 来表示三维数组，好像是内存限制超出，只能用C语言的写法，由于C语言数组的定义需要初始化大小，而题目中说了数组长度不会超 100，所以就用 100 来初始化，参见代码如下：

解法一：

class Solution {

public:

int removeBoxes(vector<int>& boxes) {

int n = boxes.size();

int dp[100][100][100] = {0};

return helper(boxes, 0, n - 1, 0, dp);

}

int helper(vector<int>& boxes, int i, int j, int k, int dp[100][100][100]) {

if (j < i) return 0;

if (dp[i][j][k] > 0) return dp[i][j][k];

int res = (1 + k) \* (1 + k) + helper(boxes, i + 1, j, 0, dp);

for (int m = i + 1; m <= j; ++m) {

if (boxes[m] == boxes[i]) {

res = max(res, helper(boxes, i + 1, m - 1, 0, dp) + helper(boxes, m, j, k + 1, dp));

}

}

return dp[i][j][k] = res;

}

};

下面这种写法是上面解法的迭代方式，但是却有一些不同，这里需要对 dp 数组的部分值做一些初始化，将每个数字的所有k值的情况的积分都先算出来，然后在整体更新三维 dp 数组的时候也很有意思，并不是按照原有的顺序更新，而是块更新，先更新 dp[1][0][k], dp[2][1][k], dp[3][2][k]….，再更新 dp[2][0][k], dp[3][1][k], dp[4][2][k]…., 再更新 dp[3][0][k], dp[4][1][k], dp[5][2][k]….，之前好像也有一道是这样区域更新的题，但是博主想不起来是哪一道了，以后想起来了再来补充吧，参见代码如下：

解法二：

class Solution {

public:

int removeBoxes(vector<int>& boxes) {

int n = boxes.size();

int dp[n][n][n] = {0};

for (int i = 0; i < n; ++i) {

for (int k = 0; k <= i; ++k) {

dp[i][i][k] = (1 + k) \* (1 + k);

}

}

for (int t = 1; t < n; ++t) {

for (int j = t; j < n; ++j) {

int i = j - t;

for (int k = 0; k <= i; ++k) {

int res = (1 + k) \* (1 + k) + dp[i + 1][j][0];

for (int m = i + 1; m <= j; ++m) {

if (boxes[m] == boxes[i]) {

res = max(res, dp[i + 1][m - 1][0] + dp[m][j][k + 1]);

}

}

dp[i][j][k] = res;

}

}

}

return n == 0 ? 0 : dp[0][n - 1][0];

}

};