Remove Boxes

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 (composed of k boxes, k >= 1), remove them and get `k*k` points.
Find the maximum points you can get.

Example 1:
Input:

```
[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)
```

Note: The number of boxes `n` would not exceed 100.

Solution 1

Getting memory limit errors for the last input, so sad. I read some of the top submissions and found out the reason: I was using STL vector instead of a C array....

Thanks to one of the top submission, which used the same idea as me, I have cleaned my code.

========== Explanation
==========
First Attempt
The initial thought is straightforward, try every possible removal and recursively search the rest. No doubt it will be a TLE answer. Obviously there are a lot of recomputations involved here. Memoization is the key then. But how to design the memory is tricky. I tried to use a string of 0s and 1s to indicate whether the box is removed or not, but still getting TLE.

One step further
I think the problem of the approach above is that there are a lot of *unnecessary* computations (not recomputations). For example, if there is a formation of `ABCDAA`, we know the optimal way is `B->C->D->AAA`. On the other hand, if the formation is `BCDAA`, meaning that we couldn't find an `A` before `D`, we will simply remove `AA`, which will be the optimal solution for removing them. Note this is true only if `AA` is at the end of the array. With naive memoization approach, the program will search a lot of unnecessary paths, such as `C->B->D->AA`, `D->B->C->AA`.

Therefore, I designed the memoization matrix to be `memo[l][r][k]`, the largest number we can get using `l` th to `r` th (inclusive) boxes with k same colored boxes as `r` th box appended at the end. Example, `memo[l][r][3]` represents the solution for this setting: `[b_l, ..., b_r, A,A,A]` with `b_r==A`.

The transition function is to find the maximum among all `b_i==b_r` for `i=l,...,r-1`:

```
memo[l][r][k] = max(memo[l][r][k], memo[l][i][k+1] + memo[i+1][r-1][0])
```

Basically, if there is one `i` such that `b_i==b_r`, we partition the array into two: `[b_l, ..., b_i, b_r, A, ..., A]`, and `[b_{i+1}, ..., b_{r-1}]`. The solution for first one will be `memo[l][i][k+1]`, and the second will be `memo[i+1][r-1][0]`. Otherwise, we just remove the last k+1 boxes (including `b_r`) and search the best solution for `l` th to `r-1` th boxes. (One optimization here: make `r` as left as possible, this improved the running time from **250ms** to **35ms**)

The final solution is stored in `memo[0][n-1][0]` for sure.

I didn't think about this question for a long time in the contest because the time is up. There will be a lot of room for time and space optimization as well. Thus, if you find any flaws or any improvements, please correct me.

```
class Solution {
public:
    int removeBoxes(vector<int>& boxes) {
        int n=boxes.size();
        int memo[100][100][100] = {0};
        return dfs(boxes,memo,0,n-1,0);
    }

    int dfs(vector<int>& boxes,int memo[100][100][100], int l,int r,int k){
        if (l>r) return 0;
        if (memo[l][r][k]!=0) return memo[l][r][k];

        while (r>l && boxes[r]==boxes[r-1]) {r--;k++;}
        memo[l][r][k] = dfs(boxes,memo,l,r-1,0) + (k+1)*(k+1);
        for (int i=l; i<r; i++){
            if (boxes[i]==boxes[r]){
                memo[l][r][k] = max(memo[l][r][k], dfs(boxes,memo,l,i,k+1) + dfs(bo
xes,memo,i+1,r-1,0));
            }
        }
        return memo[l][r][k];
    }
};
```

written by waterbucket original link here

Solution 2

When facing this problem, I am keeping thinking how to simulate the case when `boxes[i] == boxes[j]` when `i` and `j` are not consecutive. It turns out that the dp matrix needs one more dimension to store such state. So we are going to define the state as

```
dp[i][j][k] represents the max points from box[i] to box[j] with k boxes whose value
s equal to box[i]
```

The transformation function is as below

```
dp[i][j][k] = max(dp[i+1][m-1][1] + dp[m][j][k+1]) when box[i] = box[m]
```

So the Java code with memorization is as below. Kindly ask me any questions.

```
  public int removeBoxes(int[] boxes) {
        if (boxes == null || boxes.length == 0) {
            return 0;
        }

        int size = boxes.length;
        int[][][] dp = new int[size][size][size];

        return get(dp, boxes, 0, size-1, 1);
    }

    private int get(int[][][] dp, int[] boxes, int i, int j, int k) {
        if (i > j) {
            return 0;
        } else if (i == j) {
            return k * k;
        } else if (dp[i][j][k] != 0) {
            return dp[i][j][k];
        } else {
            int temp = get(dp, boxes, i + 1, j, 1) + k * k;

            for (int m = i + 1; m <= j; m++) {
                if (boxes[i] == boxes[m]) {
                    temp = Math.max(temp, get(dp, boxes, i + 1, m - 1, 1) + get(dp,
boxes, m, j, k + 1));
                }
            }

            dp[i][j][k] = temp;
            return temp;
        }


    }
```

written by wihoho2 original link here

Solution 3

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:

1. `T(i, i - 1) = 0`: no boxes so no points.
2. `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., 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 don't like, 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 (including `boxes[i]`), 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 there is another box of the same color (whose index is `m`) becomes its neighbor. Note at this moment all boxes between indices `i + 1` and `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?

For now, the boxes we left behind have 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, 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:

1. 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.
2. 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)`.
3. 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]`? 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` by `n` by `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 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) + remove
BoxesSub(boxes, m, j, k + 1, dp));
        }
    }

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

Bottom-up DP:

```java
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.

written by fun4LeetCode original link here