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(
boxes,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 va
lues 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(d
p, boxes, m, j, k + 1));
                }
            }

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


    }
```

written by wihoho2 original link here

Solution 3

```
class Solution {
public:
    int mem[100][100][100]; // initialized to 0, mem[left][right][k] means value from boxes[left]~boxes[right] followed by
    // k same color boxes. Follow does not mean strictly consecutive boxes, for example, [1, 3, 2, 3, 4], 3 can be
    // followed by the other 3 because we can remove 2 first

    int removeBoxes(vector<int>& boxes) {
        return DFS(boxes,0,boxes.size()-1,0);
    }

    int DFS(vector<int>& boxes, int l,int r,int k){
        if (l>r) return 0;
        if (mem[l][r][k]) return mem[l][r][k]; // if we have calculated this DFS result, return it

        mem[l][r][k] = DFS(boxes,l,r-1,0) + (k+1)*(k+1); // box[l][r] result is box[l][r-1]+(k+1)^2
        for (int i=l; i<r; i++) // go through each box from left
            if (boxes[i]==boxes[r]) // check for same color box as boxes[r]
                mem[l][r][k] = max(mem[l][r][k], DFS(boxes,l,i,k+1) + DFS(boxes,i+1,r-1,0)); // if we found same color box,
                // then we have a chance to get a higher value by group boxes[l]~boxes[i] and boxes[r] together, plus the
                // value from boxes[i+1]~boxes[r-1]
        return mem[l][r][k];
    }
};
```

written by luming89 original link here