0546. Remove Boxes

A. Madhur

Introduction

My interest stems from the fact that I have encountered problems of (partially) similar nature — particularly those involving pattern recognition — in the domains of Predictive Modeling and Operational Research. These problems often exhibit a common structure: They are governed by sequential dependencies (although, in real-world scenarios, such dependencies may neither be necessary nor evident) and reward-driven dynamics, in which each decision or observation directly influences the potential outcomes of subsequent states.

Problem Description

URL: `https://leetcode.com/problems/remove-boxes/`

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.

Solution

A naive solution could be seen as:

```
class Solution {
    public Map<String, Integer> seen = new HashMap<>();

    public int rx(ArrayList<Integer> nums, int i) {
        if (nums.isEmpty() || i < 0 || i >= nums.size()) { return 0; }

        String key = nums.toString();

        if (seen.containsKey(key)) { return seen.get(key); }

        ArrayList<Integer> cloned = new ArrayList<>(nums);

        int value = nums.get(i), count = 1;
        int left = i - 1, right = i + 1;

        while (left >= 0 && nums.get(left) == value) {
          --left; ++count;
        }

        while (right < nums.size() && nums.get(right) == value) {
          ++right; ++count;
        }

        nums.subList(left + 1, right).clear();
```

```java

            int pick = (count * count) + rx(nums, left);
            int skip = rx(cloned, i + 1);

            int res = Math.max(pick, skip);
            seen.put(key, res);

            return res;
        }

        public int removeBoxes(int[] boxes) {
            ArrayList<Integer> nums = new ArrayList<>();
            for (int num: boxes) { nums.add(num); }

            return rx(nums, 0);
        }
}
```

This implementation follows the problem statement verbatim; however, it is not particularly efficient. Although its theoretical asymptotic time-complexity is $O(N^4)$, the creation of a new list on each iteration, combined with the use of expensive `toString()` operations, makes each lookup $O(N)$. Furthermore, the deletion of subsections also introduces an additional $O(k)$ cost. As a result, the overall effective time-complexity becomes $O(N^5)$.

Naturally, one can employ more efficient techniques, such as key serialization, hashing, in-place mutation, or backtracking, to reduce the time-complexity. However, these methods still revolve around essentially the same underlying approach, and thus do not fundamentally alter the asymptotic behaviour.

To achieve a truly efficient solution, one of the time-tested strategies is to introduce an additional metric. This invariably adds another dimension. Where, as we have seen, $O(N^4)$ often represents the lower bound when employing traditional dynamic programming methodologies. Therefore unless otherwise this new metric acts on a better algorithm, the time-complexity could become worse. Nevertheless, a natural idea for problems of this kind —where elements may be repeated— is to employ the frequency of elements as a guiding metric. Indeed, using a running frequency array combined with a modified LIS-esque (Longest Increasing Subsequence–like) approach may yield an optimal formulation.

Adopting such a method, however, entails considerable bookkeeping of indices and careful handling of the inclusion or exclusion of elements, particularly when they appear within a perfectly increasing frequency sequence. Nevertheless, this approach provides a useful invariant: The optimal value of such a sequence remains indifferent to the placement of singleton elements. Moreover, since this formulation rapidly curtails the solution space, the relative number of such singleton elements tends to increase over time (this reduction in solution space is also evident in the top-down approach).

In practical terms, for many problem instances, the traditional dynamic programming approach still yields an acceptable result on online judges. The method presented here builds the solution from the bottom up and is reminiscent of strategies commonly employed in palindrome-related problems. Here, the number of elements from one specific end is counted while they are successively grouped with adjacent elements of the same type, allowing the state transitions to capture not only the subarray boundaries but also the number of matching elements carried over from previous positions. This formulation enables the algorithm to reuse computed results efficiently, significantly reducing redundant computations (but not iterations) and leading to an overall tractable solution in practice.

```java
class Solution {
    public int removeBoxes(int[] boxes) {
        int n = boxes.length;
        int[][][] dp = new int[n][n][n];

        for (int len = 1; len < n + 1; ++len) {
            for (int left = 0; left < n - len + 1; ++left) {
                int right = left + len - 1;
                for (int k = left; k > -1; --k) {
                    int curr = (k + 1) * (k + 1);
                    if (left < right && dp[left + 1][right][0] > 0) {
                        curr += dp[left + 1][right][0];
                    }
                    for (int mid = left + 1; mid < right + 1; ++mid) {
                        if (boxes[mid] == boxes[left]) {
                            curr = Math.max(curr,
                                            dp[left + 1][mid - 1][0] +
                                            dp[mid][right][k + 1]);
                        }
                    }

                    dp[left][right][k] = curr;
                }
            }
        }

        return dp[0][n - 1][0];
    }
}
```

It’s top down variant could be seen as:

```java
class Solution {
    public int[][][] dp;
    public int[] nums;

    public int rx(int left, int right, int boxCount) {
        if (left > right) { return 0; }

        if (dp[left][right][boxCount] > 0) { return dp[left][right][boxCount]; }

        while (left < right && nums[right] == nums[right - 1]) {
            --right;
            ++boxCount;
        }

        if (dp[left][right][boxCount] > 0) { return dp[left][right][boxCount]; }

        int fromRight = (boxCount + 1) * (boxCount + 1) + rx(left, right - 1, 0);

        int fromLeft = 0;
        for (int mid = left; mid < right; mid++) {
            if (nums[mid] == nums[right]) {
                fromLeft = rx(left, mid, boxCount + 1) + rx(mid + 1, right - 1, 0);
                fromRight = Math.max(fromRight, fromLeft);
            }
        }

        return dp[left][right][boxCount] = fromRight;
    }

    public int removeBoxes(int[] boxes) {
        int n = boxes.length;
        dp = new int[n][n][n];
        nums = boxes;

        return rx(0, n - 1, 0);
    }
}
```

Notice the double calls to the memoized results, due to the parameter `boxCount` (which is not essential but also not trivial). In this approach, the recursion captures not only the current subarray boundaries but also how many boxes of the same color are contiguous to the right boundary and can be merged into the current segment. This additional state parameter allows the algorithm to "carry over" accumulated boxes of the same color across recursive calls, effectively reducing redundant recomputation (and iterations) and ensuring that all possible states are explored.