546. Remove Boxes

March 28, 2017 | 8.7K views

Average Rating: 4.17 (18 votes)

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: 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)
```

Constraints:

- `1 <= boxes.length <= 100`
- `1 <= boxes[i] <= 100`

Solution

Approach #1 Brute Force Approach[Time Limit Exceeded]

Algorithm

The brute force approach is very obvious. We try removing every possible element of the given array and calculate the points obtained for the rest of the array in a recursive manner.

Java

```java

public class Solution {
    public int removeBoxes(int[] boxes) {
        return remove(boxes);
    }
    public int remove(int[] boxes)
    {
        if(boxes.length==0)
            return 0;
        int res=0;
        for(int i=0,j=i+1;i<boxes.length;i++)
        {
            while(j<boxes.length && boxes[i]==boxes[j])
                j++;
            int[] newboxes=new int[boxes.length-(j-i)];
            for(int k=0,p=0;k<boxes.length;k++)
            {
                if(k==i)
                    k=j;
                if(k<boxes.length)
                    newboxes[p++]=boxes[k];
            }
            res=Math.max(res,remove(newboxes)+(j-i)*(j-i));
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(n!)$. $f(n)$ be the time to find the solution of n boxes with n different colors, then obviously $f(n) = n * f(n-1)$ which results in the $n!$ time complexity.
- Space complexity : $O(n^2)$. The recursive tree goes upto a depth of n, with every level consisting of upto n newBoxes elements.

Approach #2 Using DP with Memorization[Accepted]

Algorithm

The problem with the previous approach is that it involves a lot of recomputations. e.g. Consider the array `[3, 2, 1, 4, 4, 4]`. In this case, we try to remove 3 and calculate the cost for the remaining array, in which we try removing 2 first leading to the point calculation for the subarray `[1, 4, 4, 4]`. The same happens in the second iteration in which we try to remove 2 first and then remove 3. We can prune the depth of the recursion tree a lot by using memorization.

But the problem of memorization isn't simple in this case. We can't simply use the start and end index of the array to determine the maximum number of points which that subarray will eventually lead to. This is because the points obtained by using the subarray depend not only on the subarray but also on the removals done prior to reaching the current subarray, which aren't even a part of the subarray. e.g. Consider the array `[3, 2, 1, 4, 4, 2, 4, 4]`. The points obtained for the subarray `[3, 2, 1, 4]` depend on whether the element 2(index 5) has been already removed or not, since it eventually determines the number of 4's which will be combined together to determine the potential points obtained for the currently considered subarray.

Thus, in order to preserve this information, we need to add another dimension to the memorization array, which tells us how many similar elements are combined together from the end of the current subarray. We make use of a dp array, which is used to store the maximum number of points that can be obtained for a given subarray with a specific number of similar elements at the end. For an entry in dp[l][r][k], l represents the starting index of the subarray, r represents the ending index of the subarray and k represents the number of elements similar to the r^{th} element following it which can be combined to obtain the point information to be stored in dp[l][r][k]. e.g.

This can be better understood with the following example. Consider a subarray $[x_l, x_{l+1}, .., x_i, .., x_r, 6, 6, 6]$. For this subarray, if x_r=6, the entry at dp[l][r][3] represents the maximum points that can be obtained using the subarray $boxes[l:r]$ if three 6's are appended with the trailing x_r.

Now, let us look at how to fill in the dp. Consider the same suabrray as mentioned above. For filling in the entry, dp[l][r][k], we firstly make an initial entry in dp[l][r][k], which considers the assumption that we will firstly combine the last $k+1$ similar elements and then proceed with the remaining subarray. Thus, the initial entry becomes:

$\text{dp[l][r][k]} = dp[l][r-1][0] + (k+1)*(k+1)$. Here, we combined all the trailing similar elements, so the value 0 is passed as the k value for the recursive function, since no similar elements to the $(r-1)^{th}$ element exist at its end.

But, the above situation isn't the only possible solution. We could obtain a better solution for the same subarray $boxes[l:r]$ for making the entry into dp[l][r][k], if we could somehow combine the trailing similar elements with some extra similar elements lying between $boxes[l:r]$.

Thus, we look for the elements within $boxes[l:r]$, which could be similar to the trailing k elements, which in turn are similar to the r^{th} element. Whenever such an element $boxes[i]$ is found, we check if the new solution could lead to more points by using the same array. If so, we update the entry at dp[l][r][k].

To get a clearer understanding of the above statment, consider the same subarray again: $[x_l, x_{l+1}, .., x_i, .., x_r, 6, 6, 6]$. If $x_i = x_r = 6$, we could eventually be benefitted by combining x_i and x_r by removing the elements lying between them, since now we can bring $k+2$ similar elements together. By removing the in-between lying elements($[x_{i+1}, x_{i+2}, ..., x_{r-1}]$, the maximum points we can obtain are given by: dp[i+1][r-1][0]. Now, the points obtained from the remaining array $[x_l, x_{l+1}, .., x_i, x_r, 6, 6, 6]$ are given by: dp[l][i][k+1], which is quite clear now.

Thus equation for dp updation becomes:

$\text{dp[l][r][k]} = max(\text{dp[l][r][k]}, \text{dp[l][i][k+1]} + \text{dp[i+1][r-1][0]})$.

At the end, the entry for dp[0][n-1][0] gives the required result. In the implementation below, we've made use of `calculatePoints` function which is simply a recursive function used to obtain the dp values.

Java

```java
class Solution {

    public int removeBoxes(int[] boxes) {
        int[][][] dp = new int[100][100][100];
        return calculatePoints(boxes, dp, 0, boxes.length - 1, 0);
    }

    public int calculatePoints(int[] boxes, int[][][] dp, int l, int r, int k) {
        if (l > r) return 0;
        if (dp[l][r][k] != 0) return dp[l][r][k];
        while (r > l && boxes[r] == boxes[r - 1]) {
            r--;
            k++;
        }
        dp[l][r][k] = calculatePoints(boxes, dp, l, r - 1, 0) + (k + 1) * (k + 1);
        for (int i = l; i < r; i++) {
            if (boxes[i] == boxes[r]) {
                dp[l][r][k] = Math.max(dp[l][r][k], calculatePoints(boxes, dp, l, i, k + 1) +
calculatePoints(boxes, dp, i + 1, r - 1, 0));
            }
        }
        return dp[l][r][k];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$ dp array of size n^3 is filled.
- Space complexity : $O(n^3)$ dp array is of size n^3.

Rate this article:

Previous Next

Comments: 10

lixx2100 ★ 2634 March 30, 2017 1:21 AM Report

The runtime of the code is indeed `O(n^4)`, and the bound is tight in the worst case. Consider the following test case of `100` numbers in total:

```
1, 2, 1, 2,  ... 1, 2, 1, 2 (25 copies of 1, 2) followed by
```

Read More

2 Share Reply

SHOW 1 REPLY

meganlee ★ 1094 June 18, 2018 11:17 AM

A good video tutorial for explaining the dp state transfer. The video is recursion+memoization, but behind the scene, idea is the same. https://youtu.be/U8Ru-ZpfHfA 中文视频教程

Read More

2 Share Reply

SHOW 1 REPLY

vinod23 ★ 461 April 2, 2017 3:19 PM

Approach-2 has been updated. Thanks.

1 Share Reply

suneelg ★ 3 April 2, 2017 2:52 AM

Lets say if there was a follow up question to print the boxes in the order you pick them to achieve maximum score, how can we do that? Eg. if the input was [1,2,1], it should print: 2, 1, 1.

1 Share Reply

vinod23 ★ 461 April 1, 2017 2:44 AM Report

@htrinh yes you are right. I have updated it. Thanks

1 Share Reply

htrinh ★ 0 March 31, 2017 12:07 AM

The time complexity of the brute-force approach should be n!, not 2^n.
Let f(n) be the time to find the solution of n boxes with n different colors, then obviously f(n) = n * f(n-1) which results in the n! time complexity overall.

0 Share Reply

quadpixels ★ 96 March 30, 2017 7:03 AM

I agree with @htrinh that the While loop should be modified accordingly. The While loop seems to be considering repetitions on the Left of the region instead of the right region.

As a side note, to consider the question with "how many repetitions there are to the LEFT of the region", the while loop is the same as in the code, but the transition part is to be changed to:

Read More

0 Share Reply

htrinh ★ 0 March 30, 2017 3:27 AM

The WHILE loop should be in order to match with the solution description and be accepted:
while (r > l && boxes[r] == boxes[r - 1]) {
r--;
k++;
}

Read More

0 Share Reply

indrajohn ★ 20 July 6, 2020 7:14 PM

Approach 2 is very niche.

0 Share Reply

lonewy2005 ★ 0 March 13, 2020 11:02 PM

May I propose that the brute-force time complexity is actually bigger than n's factorial?
My understanding is that n! only represents the number of leaf level nodes, but there are still calculations at the non-leaf nodes. Therefore the total complexity should be sum of all nodes: both leaf and non-leaf ones.
E.g.: for n=3, here is what happened:

Read More

0 Share Reply