Burst Balloons

Given `n` balloons, indexed from `0` to `n-1`. Each balloon is painted with a number on it represented by array `nums`. You are asked to burst all the balloons. If the you burst balloon `i` you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of `i`. After the burst, the `left` and `right` then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:
(1) You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.
(2) 0 ≤ `n` ≤ 500, 0 ≤ `nums[i]` ≤ 100

Example:

Given `[3, 1, 5, 8]`

Return `167`

```
    nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
   coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

See here for a better view

Be Naive First

When I first get this problem, it is far from dynamic programming to me. I started with the most naive idea the backtracking.

We have n balloons to burst, which mean we have n steps in the game. In the i th step we have n-i balloons to burst, i = 0~n-1. Therefore we are looking at an algorithm of O(n!). Well, it is slow, probably works for n < 12 only.

Of course this is not the point to implement it. We need to identify the redundant works we did in it and try to optimize.

Well, we can find that for any balloons left the maxCoins does not depends on the balloons already bursted. This indicate that we can use memorization (top down) or dynamic programming (bottom up) for all the cases from small numbers of balloon until n balloons. How many cases are there? For k balloons there are C(n, k) cases and for each case it need to scan the k balloons to compare. The sum is quite big still. It is better than O(n!) but worse than O(2^n).

Better idea

We then think can we apply the divide and conquer technique? After all there seems to be many self similar sub problems from the previous analysis.

Well, the nature way to divide the problem is burst one balloon and separate the balloons into 2 sub sections one on the left and one one the right. However, in this problem the left and right become adjacent and have effects on the maxCoins in the future.

Then another interesting idea come up. Which is quite often seen in dp problem analysis. That is reverse thinking. Like I said the coins you get for a balloon does not depend on the balloons already burst. Therefore instead of divide the problem by the first balloon to burst, we divide the problem by the last balloon to burst.

Why is that? Because only the first and last balloons we are sure of their adjacent balloons before hand!

For the first we have `nums[i-1]*nums[i]*nums[i+1]` for the last we have `nums[-1]*nums[i]*nums[n]`.

OK. Think about n balloons if i is the last one to burst, what now?

We can see that the balloons is again separated into 2 sections. But this time since the balloon i is the last balloon of all to burst, the left and right section now has well defined boundary and do not affect each other! Therefore we can do either recursive method with memoization or dp.

Final

Here comes the final solutions. Note that we put 2 balloons with 1 as boundaries and also burst all the zero balloons in the first round since they won't give any coins. The algorithm runs in O(n^3) which can be easily seen from the 3 loops in dp solution.

Java D&C with Memoization

```java
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] memo = new int[n][n];
    return burst(memo, nums, 0, n - 1);
}

public int burst(int[][] memo, int[] nums, int left, int right) {
    if (left + 1 == right) return 0;
    if (memo[left][right] > 0) return memo[left][right];
    int ans = 0;
    for (int i = left + 1; i < right; ++i)
        ans = Math.max(ans, nums[left] * nums[i] * nums[right]
        + burst(memo, nums, left, i) + burst(memo, nums, i, right));
    memo[left][right] = ans;
    return ans;
}
// 12 ms
```

Java DP

```java
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] dp = new int[n][n];
    for (int k = 2; k < n; ++k)
        for (int left = 0; left < n - k; ++left) {
            int right = left + k;
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = Math.max(dp[left][right],
                nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][right]);
        }

    return dp[0][n - 1];
}
// 17 ms
```

C++ DP

```
int maxCoinsDP(vector<int> &iNums) {
    int nums[iNums.size() + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int dp[n][n] = {};
    for (int k = 2; k < n; ++k) {
        for (int left = 0; left < n - k; ++left)
            int right = left + k;
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = max(dp[left][right],
                     nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][rig
ht]);
        }

    return dp[0][n - 1];
}
// 16 ms
```

Python DP

```
def maxCoins(self, iNums):
    nums = [1] + [i for i in iNums if i > 0] + [1]
    n = len(nums)
    dp = [[0]*n for _ in xrange(n)]

    for k in xrange(2, n):
        for left in xrange(0, n - k):
            right = left + k
            for i in xrange(left + 1,right):
                dp[left][right] = max(dp[left][right],
                       nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][r
ight])
    return dp[0][n - 1]

# 528ms
```

written by dietpepsi original link here

Solution 2

```
int maxCoins(vector<int>& nums) {
    int N = nums.size();
    nums.insert(nums.begin(), 1);
    nums.insert(nums.end(), 1);

    // rangeValues[i][j] is the maximum # of coins that can be obtained
    // by popping balloons only in the range [i,j]
    vector<vector<int>> rangeValues(nums.size(), vector<int>(nums.size(), 0));

    // build up from shorter ranges to longer ranges
    for (int len = 1; len <= N; ++len) {
        for (int start = 1; start <= N - len + 1; ++start) {
            int end = start + len - 1;
            // calculate the max # of coins that can be obtained by
            // popping balloons only in the range [start,end].
            // consider all possible choices of final balloon to pop
            int bestCoins = 0;
            for (int final = start; final <= end; ++final) {
                int coins = rangeValues[start][final-1] + rangeValues[final+1][en
d]; // coins from popping subranges
                coins += nums[start-1] * nums[final] * nums[end+1]; // coins from
final pop
                if (coins > bestCoins) bestCoins = coins;
            }
            rangeValues[start][end] = bestCoins;
        }
    }
    return rangeValues[1][N];
}
```

written by The_Duck original link here

Solution 3

This solution is inspired by The_Duck with his C++ solution

https://leetcode.com/discuss/72186/c-dynamic-programming-o-n-3-1100-ms-with-comments

However, I would give an explanation based on my own understanding.

The basic idea is that we can find the maximal coins of a subrange by trying every possible final burst within that range. Final burst means that we should burst balloon i as the very last one and burst all the other balloons in whatever order. dp[i][j] means the maximal coins for range [i...j]. In this case, our final answer is dp[0][nums.length - 1].

When finding the maximal coins within a range [start...end], since balloon i is the last one to burst, we know that in previous steps we have already got maximal coins of range[start .. i - 1] and range[i + 1 .. start], and the last step is to burst ballon i and get the product of balloon to the left of i, balloon i, and ballon to the right of i. In this case, balloon to the left/right of i is balloon start - 1 and balloon end + 1. Why? Why not choosing other balloon in range [0...start - 1] and [end + 1...length] because the maximal coins may need other balloon as final burst?

In my opinion, it's because this subrange will only be used by a larger range when it's trying for every possible final burst. It will be like [larger start.....start - 1, [start .. end] end + 1/ larger end], when final burst is at index start - 1, the result of this sub range will be used, and at this moment, start - 1 will be there because it's the final burst and end + 1 will also be there because is out of range. Then we can guarantee start - 1 and end + 1 will be there as adjacent balloons of balloon i for coins. That's the answer for the question in previous paragraph.

```java
public class Solution {
public int maxCoins(int[] nums) {
    if (nums == null || nums.length == 0) return 0;

    int[][] dp = new int[nums.length][nums.length];
    for (int len = 1; len <= nums.length; len++) {
        for (int start = 0; start <= nums.length - len; start++) {
            int end = start + len - 1;
            for (int i = start; i <= end; i++) {
                int coins = nums[i] * getValue(nums, start - 1) * getValue(nums,
end + 1);
                coins += i != start ? dp[start][i - 1] : 0; // If not first one,
we can add subrange on its left.
                coins += i != end ? dp[i + 1][end] : 0; // If not last one, we ca
n add subrange on its right
                dp[start][end] = Math.max(dp[start][end], coins);
            }
        }
    }
    return dp[0][nums.length - 1];
}

private int getValue(int[] nums, int i) { // Deal with num[-1] and num[num.length
]
    if (i < 0 || i >= nums.length) {
        return 1;
    }
    return nums[i];
}
```

}

written by Alexpanda original link here