Burst Balloons (/category/392/burst-balloons)

/ Share some analysis and explanations (/topic/30746.rss)

317

dietpepsi (/user/dietpepsi)
(/user/dietpepsi) Reputation: ★ 2,721

See here for a better view (http://algobox.org/burst-balloons/)

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

```
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

```
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][right]);
        }

    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][right])
    return dp[0][n - 1]

# 528ms
```

10 months ago (/post/32431)

JAVA **7.5k** (/tags/java) **DYNAMIC-PROGRAMMING** **1.2k** (/tags/dynamic-programming) **DIVIDECONQUER** **115** (/tags/divideconquer) **MEMOIZATION** **47** (/tags/memoization)

Log in to reply (/login) (https://leetcode.com/problems/burst-balloons)

0

S (/user/stpeterh) **stpeterh (/user/stpeterh)**
Reputation: ★ 64

@peisi, Thanks for providing this nice problem and your nice analysis and solution for it. I tried a python solution based on your java code. However, I got a TLE error. Could it be a result of the TLE threshold? Or, any suggestion about improving it?

BTW, I tried `collections.defaultdict` . It still does not work.

Dynamic Programming: