312. Burst Balloons

June 18, 2019 | 18.2K views

Average Rating: 3.74 (46 votes)

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:

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

Example:

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

Solution

Intuition

At first glance, the obvious solution is to find every possible order in which balloons can be burst. Since at each step we burst one balloon, the number of possibilities we get at each step are $N \times (N-1) \times (N-2) \times ... \times 1$, giving us a time complexity of $O(N!)$. We can make a small improvement to this by caching the set of existing balloons. Since each balloon can be burst or not burst, and we are incurring extra time creating a set of balloons each time, we are still looking at a solution worse than $O(2^N)$.

There are two techniques we will use to optimize our solution:

1. Divide and Conquer
   - After bursting balloon `i`, we can divide the problem into the balloons to the left of `i` (`nums[0:i]`) and to the right of `i` (`nums[i+1:]`).
   - To find the optimal solution we check every optimal solution after bursting each balloon.
   - Since we will find the optimal solution for every range in nums, and we burst every balloon in every range to find the optimal solution, we have an $O(N^2) * O(N) = O(N^3)$ solution
   - However, if we try to divide our problem in the order where we burst balloons first, we run into an issue. As balloons burst, the adjacency of other balloons changes. We are unable to keep track of what balloons the endpoints of our intervals are adjacent to. This is where the second technique comes in.
2. Working Backwards
   - Above, we start with all the balloons and try to burst them. This causes adjacency issues. Instead, we can start with no balloons and add them in reverse order of how they were popped. Each time we add a balloon, we can divide and conquer on its left and right sides, letting us add new balloons in between.
   - This gets rid of adjacency issues. For the left interval, we know that the left boundary stays the same, and we know that our right boundary is the element we just added. The opposite goes for the right interval. We compute the coins added from adding balloon `i` with `nums[left] * nums[i] * nums[right]`.

Approach 1: Dynamic Programming (Top-Down)

Algorithm

To deal with the edges of our array we can reframe the problem into only bursting the non-edge balloons and adding ones on the sides.

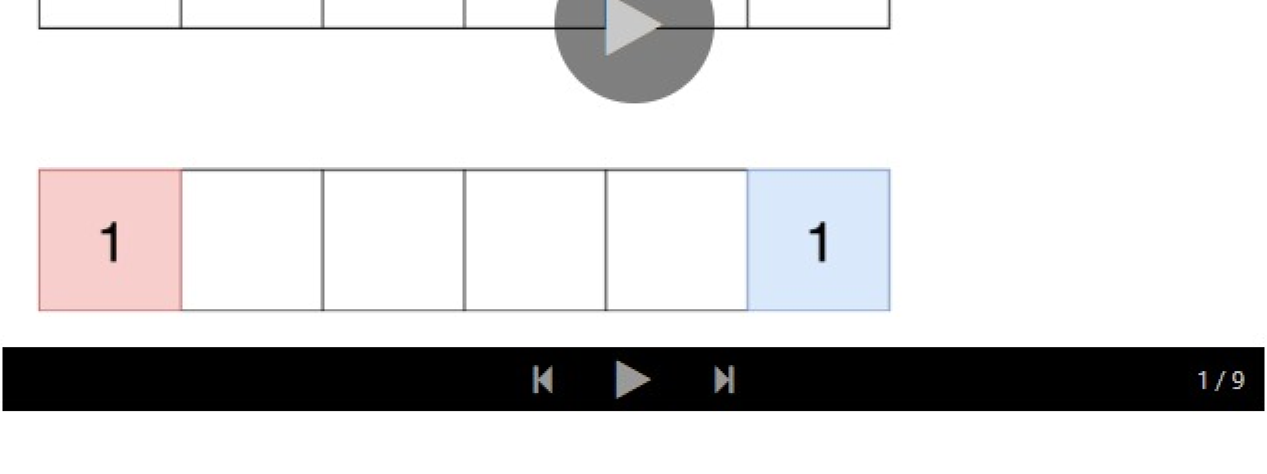
We define a function `dp` to return the maximum number of coins obtainable on the open interval (left, right). Our base case is if there are no integers on our interval (`left + 1 == right`), and therefore there are no more balloons that can be added. We add each balloon on the interval, divide and conquer the left and right sides, and find the maximum score.

The best score after adding the `i` th balloon is given by:

```
nums[left] * nums[i] * nums[right] + dp(left, i) + dp(i, right)
```

`nums[left] * nums[i] * nums[right]` is the number of coins obtained from adding the `i` th balloon, and `dp(left, i) + dp(i, right)` are the maximum number of coins obtained from solving the left and right sides of that balloon respectively.

An illustration of how we divide and conquer the left and right sides:



Implementation

Java Python Copy

```python
from functools import lru_cache

class Solution:
    def maxCoins(self, nums: List[int]) -> int:

        # reframe the problem
        nums = [1] + nums + [1]

        # cache this
        @lru_cache(None)
        def dp(left, right):

            # no more balloons can be added
            if left + 1 == right: return 0

            # add each balloon on the interval and return the maximum score
            return max(nums[left] * nums[i] * nums[right] + dp(left, i) + dp(i, right)
for i in range(left+1, right))

        # find the maximum number of coins obtained from adding all balloons from (0,
len(nums) - 1)
        return dp(0, len(nums)-1)
```

Complexity Analysis

- Time complexity : $O(N^3)$. We determine the maximum score from all (left, right) pairs. Determining the maximum score requires adding all balloons in (left, right), giving $O(N^2) * O(N) = O(N^3)$
- Space complexity : $O(N^2)$ to store our cache

Approach 2: Dynamic Programming (Bottom-Up)

Algorithm

Instead of caching our results in recursive calls we can build our way up to `dp(0, len(nums)-1)` in a bottom-up manner.

Implementation

Java Python Copy

```python
class Solution:
    def maxCoins(self, nums: List[int]) -> int:

        # reframe problem as before
        nums = [1] + nums + [1]
        n = len(nums)

        # dp will store the results of our calls
        dp = [[0] * n for _ in range(n)]

        # iterate over dp and incrementally build up to dp[0][n-1]
        for left in range(n-2, -1, -1):
            for right in range(left+2, n):
                # same formula to get the best score from (left, right) as before
                dp[left][right] = max(nums[left] * nums[i] * nums[right] + dp[left][i] +
dp[i][right] for i in range(left+1, right))

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

Complexity Analysis

- Time complexity : $O(N^3)$. There are N^2 (left, right) pairs and it takes `O(N)` to find the value of one of them.
- Space complexity : $O(N^2)$. This is the size of `dp`.