312. Burst Balloons

Hard Array Dynamic Programming

Problem Description

You are presented with a hypothetical situation involving `n` balloons, each painted with a number. These balloons are indexed from `0` to `n - 1` and the numbers painted on them are given in an array called `nums`. Your task is to "burst" all of these balloons in such a way that allows you to collect the maximum number of coins possible.

The main rule to remember is that when you burst a balloon at index `i`, you collect a number of coins equal to the product of the numbers on three balloons: the one at `i`, the one immediately before it (`i - 1`), and the one immediately after it (`i + 1`). If there is no balloon at `i - 1` or `i + 1` (which would be the case if `i` is at the start or end of the array), then you pretend there is a balloon with the number `1` painted on it.

The challenge is to find the sequence of bursting balloons that yields the highest total number of coins.

Intuition

To arrive at a solution for this problem, consider that the last balloon you burst would allow you to collect `nums[0] * nums[i] * nums[n-1]` coins, where `i` is the index of this last balloon. Since `nums[0]` and `nums[n-1]` would be out of bounds after bursting all other balloons, we treat them as `1`.

This suggests a dynamic programming approach where we consider smaller subproblems and build up to the full solution. The intuition behind the provided solution is to transform the problem into considering the maximum coins that can be collected by bursting the balloons within a subinterval of the original array.

By iteratively expanding these intervals and calculating the maximum number of coins that can be collected when bursting the last balloon in these subintervals, we can use previously solved smaller intervals to help calculate larger intervals. This concept is similar to matrix chain multiplication, where optimal solutions to smaller subproblems can be used to construct the final solution.

The idea is to iteratively consider all possible subintervals and all possible last balloons to burst within each subinterval. By doing so, we use the dynamic programming (DP) array `dp` to store the maximum coins obtained for each subinterval, so that we do not have to re-calculate the coins for those intervals again. The process starts from the smallest intervals and works its way up to the interval covering the entire range.

In the implementation, the `dp` array is first initialized to store zero values. Then, for each subinterval length `l`, starting at `2` and going up to `n`, we calculate the maximum coins for starting subintervals at index `i` and ending at `j`, using all possible last balloons `k` to find the best option. The final answer is then taken from the `dp` entry that covers the entire range, excluding the artificial balloons at both ends (that only serve the purpose of simplifying boundary conditions).

Solution Approach

The solution approach involves dynamic programming, which is a method for solving complex problems by breaking them down into simpler subproblems. The solution uses a 2D array, often referred to as a "DP table", to store the maximum number of coins that can be collected by bursting balloons within a specific subinterval of the array.

Let's delve into how the solution is implemented:

1. We first add a `1` at the beginning and the end of the `nums` array. This step simplifies our calculations for cases when we burst the first or the last balloon in the original array.
2. We define `n` as the length of this new array, now including the two additional balloons with a `1` painted on them.
3. We initialize a 2D array called `dp` with dimensions `n` by `n`, setting all values to `0`. `dp[i][j]` will store the maximum coins that can be collected by bursting all balloons between index `i` and `j`.
4. We iterate through all possible lengths of subintervals `l`, starting from `2`. We choose `2` because a subinterval must contain at least one balloon to burst (situated between the two `1`s we added).
5. For each subinterval length (`l`), we find the start (`i`) and end (`j`) of the subinterval. We restrict our subinterval to be within the bounds of the array.
6. For each subinterval from `i` to `j`, we consider each balloon `k` as the last balloon to burst in this subinterval. We calculate the coins collected from bursting balloon `k` last, which is `nums[i] * nums[k] * nums[j]`, and add to it the coins collected from the subintervals `[i, k]` and `[k, j]` which we have already calculated and stored in our `dp` table previously.
7. We update `dp[i][j]` with the maximum number of coins found for bursting all balloons between `i` and `j`.

The nested loops in the code iterate over subintervals and the potential last balloons that could be burst in that interval. The innermost loop picks the best choice for the last balloon to maximize the coins collected.

Finally, the result is the value in `dp[0][n-1]`, where `n-1` is the index after the last artificial `1`.

This solution effectively builds the solution from smaller intervals to larger ones and uses previously computed results for efficiency, avoiding redundant calculations.

Example Walkthrough

Let's consider an example with the following balloon numbers: `nums = [3, 1, 5, 8]`. Following the given solution approach:

1. We first add `1` at the beginning and the end to get `[1, 3, 1, 5, 8, 1]` to account for the boundaries.
2. The length `n` of the new array is `6`.
3. We initialize a `6x6` DP table with zeros. This table will help us remember the best solutions for subproblems.
4. We consider subinterval lengths `l` starting from `2`, because a subinterval should have at least one balloon.
5. For `l = 2`, we have subintervals `[1,3]`, `[2,4]`, `[3,5]`, and `[4,6]`:
   - For subinterval `[1,3]`, the only balloon to burst is `3` (as `1` and `1` are our added boundaries), and it gives `1 * 3 * 1 = 3` coins.
   - Similarly, for `[2,4]`, bursting `1` would give `3 * 1 * 5 = 15` coins, and so on for the other subintervals.

   Our DP table is updated for these subintervals.
6. With `l = 3`, we have subintervals `[1,2,4]`, `[2,3,5]`, and `[3,4,6]`:
   - Take `[1,2,4]`, if we decide to burst balloon `2` last, we get `1 * 1 * 5 + dp[1][2] + dp[2][4] = 5 + 3 + 15 = 23` coins. For `[2,3,5]`, the last burst `3` gives `3 * 5 * 1 + dp[2][3] + dp[3][5]` coins, and so on.

   We update our DP table accordingly.
7. This process is continued, expanding the size of subintervals until `l = n - 1`, where `n` is the length of our new array, and our subinterval is the entire array.

For each subinterval at each `l`, we consider all possible balloons that could be burst last, calculate the coins we can collect, and choose the maximum among them to update our DP table.

The result would then be in `dp[0][n-1]` which in this case will store the maximum number of coins we can collect from `[1, 3, 1, 5, 8, 1]`.

Walkthrough example with detailed subinterval analysis:

1. `l=2`:
   - Subinterval `[1,3]` (burst 3): `dp[1][3] = 1 * 3 * 1 + dp[1][2] + dp[3][3] = 3`
   - Subinterval `[2,4]` (burst 1): `dp[2][4] = 3 * 1 * 5 + dp[2][2] + dp[4][4] = 15`
   - Subinterval `[3,5]` (burst 5): `dp[3][5] = 1 * 5 * 8 + dp[3][4] + dp[5][5] = 40`
   - Subinterval `[4,6]` (burst 8): `dp[4][6] = 5 * 8 * 1 + dp[4][5] + dp[6][6] = 40`
2. `l=3`:
   - Subinterval `[1,2,4]` (choose balloon 2 to burst last for max coins): `dp[1][4] = max(calculate for bursting 2 last, calculate for bursting 3 last) = max(1*1*5+dp[1][2]+dp[3][4], 1*3*5+dp[1][3]+dp[4][4]) = max(23, 15+0+0) = 23`
   - Subinterval `[2,3,5]` (choose balloon 3 to burst last for max coins): `dp[2][5] = max(calculate for bursting 3 last, calculate for bursting 4 last) = max(3*5*1+dp[2][3]+dp[4][5], 3*1*8+dp[2][4]+dp[5][5]) = max(40+15, 15+40+0) = 55`
   - Subinterval `[3,4,6]` (choose balloon 4 to burst last for max coins): `dp[3][6] = max(calculate for bursting 4 last, calculate for bursting 5 last) = max(1*1*1+dp[3][4]+dp[5][6], 1*5*1+dp[3][5]+dp[6][6]) = max(1+0+0, 40+0+0) = 40`

And continue this way until the entire array is covered, resulting in `dp[1][5]` for the maximum coins.

By the end of the dynamic programming process, we find that the maximum number of coins that can be earned by bursting all the balloons is `dp[0][5]`.

Python Solution

```python
class Solution:
    def maxCoins(self, nums):
        """
        Calculate the maximum coins that can be obtained by bursting the balloons wisely.

        :param nums: A list of integer representing the number of balloons
        :type nums: List[int]
        :return: The maximum coins obtained
        :rtype: int
        """
        # Add a balloon with value 1 to each end of the list to simplify calculations
        nums = [1] + nums + [1]
        # Calculate the length of the new nums list
        n = len(nums)
        # Initialize the dp (Dynamic Programming) table with zeros
        dp = [[0] * n for _ in range(n)]

        # Fill the dp table
        # l represents the length of the range we're considering
        for l in range(2, n):
            # i represents the start of the range
            for i in range(n - l):
                # j represents the end of the range
                j = i + l
                # Test each possible position 'k' for the balloon to burst last in the range (i, j)
                for k in range(i + 1, j):
                    # Calculate the maximum coins by adding the coins obtained from bursting the current balloon
                    # (nums[i]*nums[k]*nums[j]) plus the maximum coins from the two subranges (i, k) and (k, j)
                    dp[i][j] = max(
                        dp[i][j],  # Current max coins for range (i, j)
                        dp[i][k] + dp[k][j] + nums[i] * nums[k] * nums[j]  # Max coins if balloon k is burst last
                    )
        # The maximum coins obtained will be in the top right corner of the dp table
        return dp[0][-1]

```

Java Solution

```java
class Solution {

    // Method to calculate the maximum coins after bursting all the balloons.
    public int maxCoins(int[] nums) {
        // Initialize an extended array that includes 1s at both ends.
        int[] extendedNums = new int[nums.length + 2];
        extendedNums[0] = 1;
        extendedNums[extendedNums.length - 1] = 1;

        // Copy the original nums array into the extendedNums array, starting at index 1.
        System.arraycopy(nums, 0, extendedNums, 1, nums.length);

        // Length of the extended array.
        int n = extendedNums.length;

        // Create a DP table to store the results of subproblems.
        int[][] dp = new int[n][n];

        // Iterate over all possible lengths (starting from 2 because we need at least
        // one balloon between i and j to calculate coins).
        for (int length = 2; length < n; ++length) {
            // Iterate over the start index of the subarray.
            for (int i = 0; i + length < n; ++i) {
                // End index of the subarray.
                int j = i + length;

                // Iterate over all possible last balloons to burst between i and j.
                for (int k = i + 1; k < j; ++k) {
                    // Update the DP table with the maximum coins from either the previously
                    // computed number or the new number of coins obtained by bursting the
                    // current balloon last, along with previously computed subresults.
                    dp[i][j] = Math.max(dp[i][j],
                            dp[i][k] + dp[k][j] + extendedNums[i] * extendedNums[k] * extendedNums[j]);
                }
            }
        }

        // Final answer is the maximum coins we can get for the entire array,
        // which is stored in dp[0][n - 1].
        return dp[0][n - 1];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        // Add 1 before the first and after the last element to handle edge cases
        nums.insert(nums.begin(), 1);
        nums.push_back(1);

        // Calculate the size of the modified vector
        int n = nums.size();

        // Initialize a 2D DP array with 0s
        vector<vector<int>> dp(n, vector<int>(n, 0));

        // Iterate over the balloons in windows of increasing size
        for (int windowLength = 2; windowLength < n; ++windowLength) {
            // i is the start index of the current window
            for (int i = 0; i + windowLength < n; ++i) {
                // j is the end index of the current window
                int j = i + windowLength;
                // k is the index of the last balloon to be burst in the current window
                for (int k = i + 1; k < j; ++k) {
                    // Update the DP table with the maximum coins obtained by bursting
                    // the last balloon at index k in the current window (subarray [i, j])
                    // The equation considers the coins obtained from bursting balloons in
                    // the subarrays [i, k] and [k, j], along with the coins from bursting
                    // the k-th balloon with its adjacent balloons at indices i and j
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[i] * nums[k] * nums[j]);
                }
            }
        }

        // The optimal solution for the entire array is stored in dp[0][n - 1]
        return dp[0][n - 1];
    }
};

```

Typescript Solution

```typescript
function maxCoins(nums: number[]): number {
    // Obtain the length of the input array
    let numberOfBalloons = nums.length;

    // Initialize a 2D array `dp` (Dynamic Programming table) for memoization
    // with dimensions (numberOfBalloons + 1) x (numberOfBalloons + 2),
    // defaulting all cells to 0.
    let dp = Array.from({ length: numberOfBalloons + 1 }, () => new Array(numberOfBalloons + 2).fill(0));

    // Pad the array of nums with 1 on both ends to simplify edge cases.
    nums.unshift(1);
    nums.push(1);

    // Reverse iterate over the array to calculate maximum coins using bottom-up approach,
    // starting from the second to last element and going backwards.
    for (let left = numberOfBalloons - 1; left >= 0; --left) {
        for (let right = left + 2; right < numberOfBalloons + 2; ++right) {
            // Check all possible balloons that can be burst between left and right
            // excluding the boundaries, and compute the maximum coins that can be obtained.
            for (let k = left + 1; k < right; ++k) {
                // Calculate coins obtained by bursting k-th balloon between balloons at 'left' and 'right',
                // and add the coins obtained from previously burst balloons on both sides.
                let coins = nums[left] * nums[k] * nums[right] + dp[left][k] + dp[k][right];
                dp[left][right] = Math.max(coins, dp[left][right]); // Store the maximum in the DP table.
            }
        }
    }
    // Return the maximum coins obtained by bursting all balloons except the first and last added ones.
    return dp[0][numberOfBalloons + 1];
}

```

Time and Space Complexity

Time Complexity

The given code implements the dynamic programming approach to solve the burst balloons problem. The time complexity of the code is dependent on the nested loops used in the algorithm.

- There is an outer loop that iterates through all possible lengths of the subarrays (denoted by `l`), which goes from 2 to `n`. So, it iterates `n - 2` times.
- For every possible length, there is a nested loop iterating over the start indices of the subarrays (denoted by `i`), and this runs for `n - l` iterations for each `l`.
- Inside the second loop, there is an innermost loop that iterates through all possible k positions where the balloon can be burst, between the start and end of the subarrays (from `i + 1` to `j - 1` where `j = i + l`).

The combination of these nested loops looks something like a cubic number of operations, giving us a time complexity of `O(n^3)`, where `n` is the number of balloons (including the two added ones).

Space Complexity

The space complexity is mainly dictated by the 2D dynamic programming array `dp`, which has a size of `n * n`.

As a result, the space complexity of the code is `O(n^2)`, where `n` is the total number of elements in the augmented array `nums`.