2212. Maximum Points in an Archery Competition

Medium Bit Manipulation Array Backtracking Enumeration Leetcode Link

Problem Description

Alice and Bob participate in an archery competition with a unique scoring system. They each shoot `numArrows` arrows at a target divided into sections numbered from 0 to 11. For each section, a player earns points equal to the section number if they shoot strictly more arrows into that section than the opponent, and no points are awarded if both competitors shoot an equal number of arrows and that number is zero.

The challenge is to create a strategy for Bob to shoot his arrows to maximize his points, given the fixed number of arrows he can shoot (`numArrows`) and an array `aliceArrows` detailing how many arrows Alice shot in each section.

The goal is to return the array `bobArrows` representing the number of arrows Bob should shoot at each scoring section to maximize his points total. Bob's strategy doesn't have to be unique; any optimal configuration satisfying the points maximization condition is a valid answer as long as the sum of `bobArrows` equals `numArrows`.

Intuition

The problem is essentially about making strategic decisions to outscore Alice while working with a limited resource: a fixed number of arrows. Since higher-scoring sections contribute more to the total score, it's generally advantageous for Bob to focus on winning these sections, provided the cost (number of arrows needed) does not exceed the potential gain in points.

However, we must balance this approach with the constraint on the total number of arrows Bob has. The strategy thus becomes a variant of the knapsack problem, where each section is like an item with a 'weight' (number of arrows needed to win the section) and 'value' (the score of the section), and the total number of arrows Bob has is like the capacity of the knapsack.

To arrive at the optimal solution, we can use a bit mask to represent whether or not Bob should compete in each individual section (1 for compete, 0 for not compete). We iterate over all possible combinations (states) Bob could take, and for each state, calculate the total points earned and the number of arrows used. We keep track of the state that yields the highest score without exceeding the total number of arrows Bob can use.

The solution code then converts the optimal state into a specific distribution of arrows, ensuring that the sections Bob won are those indicated by the state, and that all leftover arrows (if any) are assigned to the 0 section, which has no score and hence will not change Bob's total points.

Solution Approach

The solution involves a bit manipulation and brute-force approach, iterating over all possible states that represent shooting strategies for Bob. For each state, we check if it is possible for Bob to use his arrows to gain more points than Alice with the `numArrows` available.

Here's the step-by-step breakdown of the implementation, referencing the supplied Python code:

1. **Initialization**: We know there are 12 sections on the target, so `n = len(aliceArrows)` ensures we work with all sections. We initialize `state` to 0 to track the best combination of movements for Bob to beat or tie Alice's score, and `mx` to -1 to keep track of the maximum points Bob can score.

2. **Bitmask Enumeration**: We loop through all `2^n` (from `1` to `1 << n`, exclusive of `1 << n`) possible combinations of sections using a bitmask `mask`. Each bit in the mask indicates whether Bob should try to win that section (bit is 1) or not (bit is 0).

3. **Scoring Calculation**: Inside the loop, we initialize `cnt` to track the number of arrows Bob is shooting under this state and `points` to track the score acquired. We iterate through all sections, and if the bitmask indicates Bob should compete in a section `i` (using `(mask >> i) & 1`), we add the number of arrows Alice shot in this section plus one to `cnt` and increment `points` with the section number `i`.

4. **State and Score Updating**: If the number of arrows `cnt` is less than or equal to `numArrows` (meaning Bob can actually apply this strategy without running out of arrows), and the accumulated `points` of this state are higher than the maximum points `mx` seen so far, we update `state` with the current bitmask and `mx` with the current `points`.

5. **Construct the Result Array**: After finding the optimal state, we initialize an array `ans` of zeros with `n` slots to represent the number of arrows Bob will shoot in each section. We go through each section again and if the state indicates that Bob should win that section, we set `ans[i]` with the number Alice shot plus one (`alice + 1`), and subtract that amount from `numArrows`. We assign all remaining arrows to section `0`, which does not contribute any points (`ans[0] = numArrows`).

6. **Returning the Result**: Finally, we return the array `ans`, which represents the strategy that Bob should follow to shoot his arrows to maximize his score with respect to Alice's recorded shots.

Through this approach, we have ensured Bob's possible strategies are exhaustively considered and the most beneficial one is chosen, in compliance with the constraints provided by the number of arrows Bob has (`numArrows`) and the result of Alice's shots (`aliceArrows`).

Example Walkthrough

Let's use a small example to illustrate the solution approach described. Let's say Bob has `numArrows = 5` arrows he can shoot and Alice's shots in each section are given by the array `aliceArrows = [0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0]`.

We're looking to maximize Bob's score by deciding how many arrows Bob should shoot in each section.

1. **Initialization**: We have `n = 12` sections on the target. Initially, `state = 0` and `mx = -1`.

2. **Bitmask Enumeration**: We go through all possible `2^12` combinations of sections. For illustration, we'll consider two example states (bitmasks): `0b000000000101` where Bob chooses to compete in sections 2 and 0; and `0b000000100001`, where Bob chooses to compete in sections 0 and 3.

3. **Scoring Calculation**:

   - For the state `0b000000000101`, Bob would need to shoot `aliceArrows[2]+1 = 2` arrows to win section 2 and no arrows in section 0 since Alice shot 0 arrows there. This means he uses 2 arrows and scores 2 points.
   - For the state `0b000000100001`, Bob needs `aliceArrows[3]+1 = 3` arrows to win section 3. This uses 3 arrows and gets 3 points.

4. **State and Score Updating**:

   - With the first state `0b000000000101`, Bob uses 2 arrows out of 5, scoring 2 points. This is our current maximum.
   - With the second state `0b000000100001`, Bob uses 3 arrows scoring 3 points. This is now our new maximum, so `state = 0b000000100001` and `mx = 3`.

5. **Construct the Result Array**: Based on the state `0b000000100001`, the array `ans` would start as `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`. We update it to have 3 arrows in section 3 (`ans[3] = 3`), and since Bob has 2 arrows remaining, we put them in section 0 (`ans[0] = 2`) for no additional points.

6. **Returning the Result**: The final `ans` array representing Bob's optimal strategy is `[2, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0]`.

Through this example, we followed the solution approach to maximize Bob's score by distributing his 5 arrows across the sections, taking into account Alice's shots. The outcome showed that competing in sections with a higher score (i.e., section 3 worth 3 points) is generally more beneficial than competing in several lower-scoring sections.

Python Solution

```python
class Solution:
    def maximumBobPoints(self, num_arrows: int, alice_arrows: List[int]) -> List[int]:
        # Length of Alice's arrows array to determine the number of score sections
        num_sections = len(alice_arrows)

        # Initialize the variables to keep track of the best state and maximum score achievable
        best_state = 0
        max_points = -1

        # Iterate over all the possible combinations of winning score sections
        for mask in range(1 << num_sections):
            # 'cnt' will hold the total number of arrows used and 'points' will hold the total score
            cnt = points = 0

            # Check each score section to see if Bob can win it
            for i, alice in enumerate(alice_arrows):
                # If the ith bit of mask is set, Bob wins this score section
                if (mask >> i) & 1:
                    cnt += alice + 1  # Increment arrows count by Alice's arrows plus one
                    points += i       # Increment Bob's total points

            # If the current combination uses less or equal arrows than available and maximizes the points
            if cnt <= num_arrows and max_points < points:
                best_state = mask
                max_points = points

        # Create an array 'ans' to represent the number of arrows Bob will use for each score section
        ans = [0] * num_sections

        # Distribute the arrows Bob uses as per the best combination found
        for i, alice in enumerate(alice_arrows):
            if (best_state >> i) & 1:
                ans[i] = alice + 1  # Set the number of arrows for the ith section
                num_arrows -= ans[i]   # Decrement the number of remaining arrows

        # Bob uses all remaining arrows in section 0, as it won't give any points to Bob
        ans[0] += num_arrows

        return ans

```

Java Solution

```java
class Solution {
    // Main method to compute the max points Bob can score and the distribution of arrows.
    public int[] maximumBobPoints(int numArrows, int[] aliceArrows) {
        int n = aliceArrows.length; // Number of arrow targets
        int maxPoints = -1; // The maximum points Bob can earn
        int bestState = 0; // The binary representation of the best combination of targets

        // Iterate over all possible combinations of targets
        for (int mask = 1; mask < (1 << n); ++mask) {
            int arrowCount = 0; // Number of arrows used in the current combination
            int points = 0; // Points scored in the current combination

            // Check each bit in the mask
            for (int i = 0; i < n; ++i) {
                if (((mask >> i) & 1) == 1) {
                    // If the bit is set, Bob wins the target, and we need to use one more arrow than Alice
                    arrowCount += aliceArrows[i] + 1;
                    // Add the index to the points because that's the score for each target
                    points += i;
                }
            }

            // Update max points and best state only if this mask uses fewer arrows than Bob has and scores more points
            if (arrowCount <= numArrows && maxPoints < points) {
                bestState = mask;
                maxPoints = points;
            }
        }

        // Create an array to hold the number of arrows Bob should shoot at each target
        int[] result = new int[n];
        for (int i = 0; i < n; ++i) {
            if (((bestState >> i) & 1) == 1) {
                // If the target is part of the best combination, use the required number of arrows
                result[i] = aliceArrows[i] + 1;
                numArrows -= result[i]; // Subtract the used arrows from the available total
            }
        }

        // If Bob has any arrows left after beating Alice, add them all to the score for the 0th target
        result[0] += numArrows;

        // Return the final distribution of arrows across the targets
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Method to calculate the maximum points Bob can achieve,
    // along with the distribution of arrows for each score.
    vector<int> maximumBobPoints(int numArrows, vector<int>& aliceArrows) {
        int n = aliceArrows.size(); // The number of segments in the target.
        int bestState = 0; // State to track the best combination of segments that Bob can choose.
        int maxPoints = -1; // The maximum points that Bob can score.

        // Iterate over all possible combinations of segments.
        // (1 << n) creates a bitmask for all segments.
        for (int mask = 1; mask < (1 << n); ++mask) {
            int arrowsUsed = 0; // Arrows Bob used to win the segments.
            int pointsEarned = 0; // Points scored by Bob.

            // Check each segment in the current combination.
            for (int i = 0; i < n; ++i) {
                // If the bit for the current segment is set in the mask,
                // it means Bob tries to win this segment.
                if ((mask >> i) & 1) {
                    arrowsUsed += aliceArrows[i] + 1; // Arrows required to win over Alice.
                    pointsEarned += i; // Add the points of the current segment to the total.
                }
            }

            // Update the best state if the current combination uses less or equal
            // number of arrows than Bob has and offers more points.
            if (arrowsUsed <= numArrows && maxPoints < pointsEarned) {
                bestState = mask;
                maxPoints = pointsEarned;
            }
        }

        // Construct the optimal distribution of arrows for each segment.
        vector<int> optimalDistribution(n); // Vector to store the answer.
        for (int i = 0; i < n; ++i) {
            // Assign the arrows to the segments that are included in the best state.
            if ((bestState >> i) & 1) {
                optimalDistribution[i] = aliceArrows[i] + 1;
                numArrows -= optimalDistribution[i];
            }
        }

        // Add the remaining arrows to the lowest scoring segment, as they don't add to Bob's points.
        optimalDistribution[0] += numArrows;

        // Return the final arrow distribution that ensures the maximum points scored.
        return optimalDistribution;
    }
};

```

Typescript Solution

```typescript
function maximumBobPoints(numArrows: number, aliceArrows: number[]): number[] {
    // Defines the depth-first search (DFS) function for finding the optimal distribution.
    const depthFirstSearch = (currentArrows: number[], index: number, remainingArrows: number): number[] => {
        // Base case: when there are no more types of arrows or no arrows left.
        if (index < 0 || remainingArrows === 0) {
            currentArrows[0] += remainingArrows; // Assign remaining arrows to the 0-index score.
            return currentArrows;
        }

        // Recursive case: compute the result without considering the current type of arrow.
        const resultWithoutCurrent = depthFirstSearch([...currentArrows], index - 1, remainingArrows);

        if (remainingArrows > aliceArrows[index]) { // If Bob can win over Alice's count for this arrow type.
            currentArrows[index] = aliceArrows[index] + 1; // Increase Bob's count to one more than Alice for this type.

            // Compute the result with the current arrow type included.
            const resultWithCurrent = depthFirstSearch(currentArrows, index - 1, remainingArrows - aliceArrows[index] - 1);

            // Compare total points from both scenarios using reduce to sum scores.
            if (resultWithCurrent.reduce((acc, value, idx) => acc + (value > 0 ? idx : 0), 0) >=
                resultWithoutCurrent.reduce((acc, value, idx) => acc + (value > 0 ? idx : 0), 0)) {
                return resultWithCurrent; // If with current yields a higher or equal score, choose that branch.
            }
        }

        // If not taking the current arrow or the result without current scores higher, choose it.
        return resultWithoutCurrent;
    };

    // Start the depth-first search from the highest score type with all arrows available.
    return depthFirstSearch(new Array(12).fill(0), 11, numArrows);
}

```

Time and Space Complexity

The given code aims to maximize the points Bob can score over Alice by choosing specific targets to shoot given a limited number of arrows. The total number of targets is denoted by `n`.

Time Complexity:

The time complexity of the code is determined by the main loop iterating over all the subsets of the targets that could be shot by Bob (`2^n` possible states), and the inner loop which sums the arrows needed and the points scored for each subset.

- Main loop: iterates over `2^n` subsets (`2^n` times)
- Inner loop: iterates over `n` targets to calculate `cnt` and `points` (takes `O(n)` for each subset)

Multiplying the number of iterations of both loops gives us the overall time complexity of the code, which will be `O(n * 2^n)`.

Space Complexity:

The space complexity of the code involves storing the best state (`state`), maximum points (`mx`), the final answer (`ans` array), and the variables used in loop iterations.

- The `ans` array uses `O(n)` space.
- The variables `state`, `mx`, `mask`, `cnt`, and `points` use `O(1)` space each.

Since `ans` is the most significant space consumption, the overall space complexity is `O(n)`.