1989. Maximum Number of People That Can Be Caught in Tag

Medium Greedy Array

Problem Description

In the game of tag described, we're given an array representing two types of players: those who are "it" (represented as `1`) and those who are not "it" (represented as `0`). Each player who is "it" has a specific reach (`dist`) which defines the range (inclusive) from their index position within which they can tag a player that is not "it". The goal is to calculate the maximum number of people who are not "it" that can be tagged by those who are "it".

The main challenge is to maximize the number of tags while considering the positions and the reach of the players who are "it". It's essentially a problem of finding the maximum pairing between the players who are "it" and those who are not, within the distance constraints set by `dist`.

Intuition

To arrive at the solution, consider that we must optimize the pairing between players who are "it" with those who are not. We can iterate through the `team` array, keeping track of two pointers: one for the current player who is "it" (`i`) and one for the potential player to catch (`j`).

As soon as we encounter a player who is "it", we look to find the nearest player who is not "it" within the legal range of `dist`. If such a player exists, we increment our answer `ans`, as this represents a successful tag, and move the `j` pointer forward to look for the next available player.

This approach naturally leads to a greedy solution, where we prioritize tagging the closest player who is not "it" for each "it" player, because this leaves the possibility open for more distant "it" players to tag those further away. We keep iterating until we either run out of "it" players or we exhaust checking all players who are not "it".

The solution code uses a single loop passing through the whole team, which ensures that the time complexity is linear with respect to the size of the `team` array.

Solution Approach

The implementation of the solution employs a single pass algorithm with a two-pointer technique, where one pointer (denoted as `i`) goes through the `team` array, and the other pointer (`j`) is used to find the nearest person to be caught.

Here's a step-by-step breakdown of the algorithm:

1. Initialize two variables: `ans` to accumulate the number of successful tags, and `j` to serve as the pointer for finding someone to catch.
2. Loop through the `team` array with the index `i`, representing the current person checking for potential tags.
3. When a person who is "it" (`team[i] == 1`) is encountered, start or continue the inner search with pointer `j`.
4. The inner loop increases the `j` pointer until it finds a person who is not "it" (`team[j] == 0`) and is within the catching range (`i - j <= dist`).
5. Once a person to catch is found, increase the answer counter `ans`, and increment `j` to avoid catching the same person more than once.
6. Repeat this process for each person who is "it" in the `team` array.
7. If `j` reaches the end of the array (`j >= n`), break out of the inner loop since there are no more people left to catch.
8. Continue until all "it" persons have been processed or no one left to catch.

In terms of data structures, only the original input array `team` is used, and no additional space is needed, making this an in-place algorithm with O(1) space complexity.

The time complexity is O(n), where n is the length of the `team` array, as each element in the `team` is visited up to twice – once by the `i` pointer and once by the `j` pointer. However, `j` does not reset after each iteration; it moves continuously forward.

The given implementation successfully captures the greedy nature of the problem, where each "it" person attempts to catch the nearest possible "not it" person in order to maximize the overall number of tags.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following setup for the game of tag:

- Team array: `team = [0, 1, 0, 0, 1, 0]`
- Reach distance: `dist = 2`

Our goal here is to maximize the number of people who are not "it" (represented by `0`) that can be tagged by the players who are "it" (represented by `1`).

Let's walk through the algorithm:

1. We start with `ans = 0` and `j = 0`, where `ans` will hold the number of successful tags and `j` is our pointer to look for the next player to catch.
2. Begin iterating through the `team` array (i = 0). Since `team[0]` is not "it", we do nothing and continue.
3. Move to `team[1]` (i = 1). Since `team[1]` is "it", we begin looking for the nearest player to catch using `j`.
4. Increase `j` until `j` reaches 2, satisfying the condition `team[j] == 0` and `i - j <= dist`. We successfully tag the player at `j = 2` and increment `ans` to 1.
5. We continue to iterate with `i` now at 2. But since `team[2]` is already tagged, we move forward.
6. At `team[3]` (i = 3), nothing happens since it's not "it".
7. At `team[4]` (i = 4), we find another "it" player. We start from the current position of `j = 3`.
8. Since the player at `j = 3` is within reach (`i - j <= dist`), we tag them as well, increment `j` to 4, and increment `ans` to 2.
9. Moving to `team[5]` (i = 5), we notice `j` is at the end of the array, so no more players can be tagged.

At the end of this walkthrough, `ans = 2`, indicating two players were successfully tagged by players who were "it". This reflects a maximized number of tags based on the "greedy" method of tagging the nearest untagged player within reach for each "it" player.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def catchMaximumAmountofPeople(self, team: List[int], dist: int) -> int:
        # Initialize count of pairs and the index for the team member to be paired with.
        maximum_pairs = 0
        team_member_index = 0

        # Get the number of people on the team.
        number_of_people = len(team)

        # Iterate through members of the team.
        for i, current_member in enumerate(team):
            # Check if the current member is ready to catch (indicated by 1).
            if current_member:
                # Move the team_member_index forward to find the next ready catcher within the allowed distance.
                while team_member_index < number_of_people and (team[team_member_index] or i - team_member_index > dist):
                    team_member_index += 1

                # When a valid team member is found within the distance 'dist',
                # increase the result and move team_member_index forward to pair this catcher.
                if team_member_index < number_of_people and abs(i - team_member_index) <= dist:
                    maximum_pairs += 1
                    team_member_index += 1

        # Return the maximum number of pairs of people.
        return maximum_pairs
```

Java

```java
class Solution {
    public int catchMaximumAmountOfPeople(int[] team, int dist) {
        int maxCatches = 0; // This variable stores the maximum number of people that can be caught
        int teamLength = team.length; // The length of the team array

        // Two pointers, i for catcher's position and j for nearest catchable runner
        for (int i = 0, j = 0; i < teamLength; ++i) {
            // Check if current position has a catcher
            if (team[i] == 1) {
                // Move j to the next catchable runner within the distance
                while (j < teamLength && (team[j] == 1 || i - j > dist)) {
                    ++j;
                }
                // If j is a valid runner, increment the catch count and move j to next position
                if (j < teamLength && Math.abs(i - j) <= dist) {
                    maxCatches++; // Increase number of people caught
                    j++; // Move j to the next position
                }
            }
        }
        // Return the computed maximum number of catches
        return maxCatches;
    }
}
```

C++

```cpp
#include <vector>
#include <cmath>

class Solution {
public:
    // Function to calculate the maximum number of people that can be caught
    int catchMaximumAmountOfPeople(vector<int>& team, int dist) {
        // Initialize the count of caught people as zero
        int countCaught = 0;
        // Get the total number of people
        const int numPeople = team.size();
        // Two-pointer method variables: i for catcher, j for catchee
        for (int i = 0, j = 0; i < numPeople; ++i) {
            // Isolate the cases where the current person is a catcher (team[i] == 1)
            if (team[i] == 1) {
                // Advance the catchee pointer 'j' to find a catchable person
                while (j < numPeople && (team[j] == 1 || i - j > dist)) {
                    ++j;
                }
                // If j is within bounds and within distance 'dist', a catch is possible
                if (j < numPeople && std::abs(i - j) <= dist) {
                    ++countCaught; // Increase the count of caught people
                    ++j; // Move the 'j' pointer forward to ensure a person can only be caught once
                }
            }
        }
        // Return the total count of people caught
        return countCaught;
    }
};
```

TypeScript

```typescript
// Function to calculate the maximum number of people that can be caught
function catchMaximumAmountOfPeople(team: number[], dist: number): number {
    // Initialize the count of people caught as zero
    let countCaught: number = 0;
    // Get the total number of people
    const numPeople: number = team.length;

    // Two-pointer method: 'i' for catcher, 'j' for the person being caught
    let i: number = 0, j: number = 0;
    while (i < numPeople) {
        // Isolate the case where the current person is a catcher (team[i] === 1)
        if (team[i] === 1) {
            // Advance the 'j' pointer to find a person who can be caught
            while (j < numPeople && (team[j] === 1 || Math.abs(i - j) > dist)) {
                j++;
            }
            // If 'j' is within bounds and within the distance 'dist', a catch is possible
            if (j < numPeople && Math.abs(i - j) <= dist) {
                countCaught++; // Increase the count of people caught
                j++; // Move the 'j' pointer forward to ensure a person can only be caught once
            }
        }
        i++; // Move to the next potential catcher
    }

    // Return the total count of people caught
    return countCaught;
}
```

```python
from typing import List

class Solution:
    def catchMaximumAmountofPeople(self, team: List[int], dist: int) -> int:
        # Initialize count of pairs and the index for the team member to be paired with.
        maximum_pairs = 0
        team_member_index = 0

        # Get the number of people on the team.
        number_of_people = len(team)

        # Iterate through members of the team.
        for i, current_member in enumerate(team):
            # Check if the current member is ready to catch (indicated by 1).
            if current_member:
                # Move the team_member_index forward to find the next ready catcher within the allowed distance.
                while team_member_index < number_of_people and (team[team_member_index] or i - team_member_index > dist):
                    team_member_index += 1

                # When a valid team member is found within the distance 'dist',
                # increase the result and move team_member_index forward to pair this catcher.
                if team_member_index < number_of_people and abs(i - team_member_index) <= dist:
                    maximum_pairs += 1
                    team_member_index += 1

        # Return the maximum number of pairs of people.
        return maximum_pairs
```

Time and Space Complexity

The given Python code is designed to count the maximum number of people a team can catch within a certain distance `dist`. The algorithm uses a greedy two-pointer approach.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the `team` list. This is because:

- The for-loop iterates through the list once, which contributes `O(n)`.
- Inside the loop, the while-loop moves the second pointer `j` forward until it finds a valid person to catch or reaches the end of the list. Each element is visited by the `j` pointer at most once throughout the entire execution of the algorithm. Therefore, the total number of operations contributed by the inner while-loop across all iterations of the for-loop is also `O(n)`.

Hence, the combined time complexity remains `O(n)` since both the outer for-loop and the cumulative operations of the inner while-loop are linear with respect to the size of the input list.

Space Complexity

The space complexity of the code is `O(1)`.

This is because the algorithm uses a constant amount of extra space for variables `ans`, `j`, `n`, and `i`. The space used does not depend on the input size and remains constant even as the input list `team` grows in length.