2511. Maximum Enemy Forts That Can Be Captured

Easy Array Two Pointers

Problem Description

You are given an array `forts` which consists of integers representing the positions of forts. The integer values can be -1, 0, or 1. Here's what they represent:

- A `-1` means no fort is present at that position.
- A `0` indicates an enemy fort is present at that position.
- A `1` indicates a fort under your command is present at that position.

Your objective is to move your army from one of your forts to an empty position. The conditions for the movement are:

- Your army must travel over positions containing only enemy forts (denoted by `0`).
- You cannot pass over an empty position or a position with your fort during this move.

The question asks you to calculate the maximum number of enemy forts you can capture during such a move. If it is not possible to move your army or you don't have any forts under your command, then the maximum number is `0`.

Intuition

To solve this problem, we can perform a linear scan over the array `forts` with the following approach:

- Start from the first element, iterate through the array and search for a fort under your command (a `1` in the array).
- Once a fort under your command is found, search for the next closest enemy fort (a `0`), and keep counting the number of enemy forts in between.
- If another fort under your command is found (another `1`), then the number of enemy forts captured during this move would be the count of `0`s found in between minus one (since we lose one position when we land on the second `1`).
- Keep track of the maximum count found throughout the entire iteration.
- Continue this process until the end of the array is reached.

This algorithm efficiently computes the maximum number of enemy forts that can be captured because it accounts for all possible movements from each of your forts to any other eligible position in a single pass. By always choosing the move that captures the most forts, we ensure that we find the maximum possible value.

Solution Approach

The provided solution uses a simple one-pass algorithm to iterate over the array of forts. We make use of a while loop that runs until the end of the array. There are two pointers used in the process: `i` and `j`. Pointer `i` is used to locate forts under our command, and `j` is used to find the next fort under our command after capturing enemy forts.

Here's a step-by-step approach to the solution:

1. Initialize two variables, `i` and `ans`, to 0. The variable `i` serves as a pointer to the current position in `forts`, and `ans` stores the maximum number of enemy forts captured so far.
2. Use a while loop to traverse the `forts` array from the current position identified by `i` until the end of the array (`n` is the length of `forts`).
3. Inside the loop, initiate a nested while loop that starts with `j = i + 1`. The variable `j` scans forward to count enemy forts:
   - If `forts[i]` is `1` (your fort), traverse through the array starting from `i + 1` to find a subsequent `0` (enemy fort). During traversal, increment `j` until a `0` is not encountered (either reach another `1` or `-1`).
   - If the loop ends at a position `j` where `forts[j]` is `1`, it means an enemy fort was captured. Thus, calculate the number of captured forts as `j - i - 1`. The `-1` subtracts the position of the ending `1`, which is not an enemy fort.
   - Update the maximum number of enemy forts captured, `ans`, if the current count (`j - i - 1`) is greater than the previous value of `ans`.
4. After processing from position `i` to `j`, move the pointer `i` to position `j` and continue the algorithm until all positions are scanned. This ensures no potential fort positions are skipped and that we consider all possibilities.
5. Finally, return the value of `ans`, which holds the maximum number of enemy forts captured in any possible move.

This simple algorithm is optimal because it ensures that:

- Only forts under your command (`1`) are considered as the starting point for an invasion.
- Only sequences consisting solely of enemy forts (`0`) are considered for capturing.
- The number of checks is the lowest possible, as we only care about streaks of enemy forts between your forts, achieving O(n) time complexity.

Example Walkthrough

Let's use a small example to illustrate the solution approach given the following array `forts`:

```
forts = [1, 0, 0, -1, 1, 0, 1]
```

With this array, follow the solution approach step by step:

1. **Initialize Variables**: Start with `i = 0` and `ans = 0`.
2. **Outer While Loop (Traverse `forts`)**: Begin loop from the first position.
3. **Inner While Loop (Locate Next Fort)**:
   - Position `i = 0` has value `1`, indicating a fort under your command.
   - Move to `j = i + 1`, looking for consecutive enemy forts (`0`).
4. **Capture Enemy Forts**:
   - Moving ahead from position `j = 0`: We have `forts[1] = 0`, `forts[2] = 0`, and the next position `forts[3]` is `-1`, so we stop here since we cannot move over an empty position. Therefore, no enemy forts were captured in this sequence.
   - Move the pointer `i` to position `3`. Repeat for the next portion of the array.
5. **Repeat the Process**:
   - Since `forts[3] = -1`, we move to `i = 4`.
   - At `i = 4` (`forts[4] = 1`), we have a fort under our control once again.
   - Start the inner loop from `j = i + 1 = 5`.
   - Move ahead: `forts[5] = 0` is an enemy fort.
   - However, at `j = 5 + 1 = 6`, we reach `forts[6] = 1`, another fort under our control, and we stop.
6. **Update Maximum Captured Enemy Forts**:
   - In this sequence, we have captured `j - i - 1 = 6 - 4 - 1 = 1` enemy fort.
   - Update `ans` to the maximum of itself and the current count: `ans = max(ans, 1)`.
   - Now, `ans = 1`.
7. **Continue Until the End**:
   - Move `i` to `j = 6` and continue, but we're already at the end of the array.
8. **Conclude with Results**:
   - Having traversed the entire array, `ans = 1` is the maximum number of enemy forts we can capture during a move.

This example shows that with the given solution approach, we can calculate the maximum number of enemy forts that the army can capture in one move, which is `1` for this particular array.

Solution Implementation

Python

```python
class Solution:
    def captureForts(self, forts: List[int]) -> int:
        n = len(forts)  # Get the length of the forts list.
        i = ans = 0  # Initialize pointer i to 0 and answer 'ans' to 0.

        # Iterate through the list of forts.
        while i < n:
            j = i + 1  # Set the pointer j to the next position after i.

            # If the current fort at position i is a friendly fort (non-zero).
            if forts[i]:
                # Find the next friendly fort (non-zero) starting from position j.
                while j < n and forts[j] == 0:
                    j += 1

                # If such a fort is found and the sum of the values at i and j is 0,
                # which means they cancel each other out.
                if j < n and forts[i] + forts[j] == 0:
                    # Calculate the distance between the forts, excluding the start and end points,
                    # and update the answer 'ans' if this distance is greater.
                    ans = max(ans, j - i - 1)

            # Move the starting pointer i to the position of j for the next iteration.
            i = j

        # Return the maximum number of captured forts.
        return ans
```

Java

```java
class Solution {

    // Method to find the maximum number of zeroes between two non-zero numbers in an array,
    // where the non-zero numbers sum up to zero.
    public int captureForts(int[] forts) {
        // Length of the array representing forts.
        int fortsLength = forts.length;

        // This will hold the final answer: the maximum number of zeroes between capturing forts.
        int maxZeroes = 0;

        // Index to iterate through forts array.
        int i = 0;

        // Iterate over the array.
        while (i < fortsLength) {
            // Potential second fort index.
            int nextFortIndex = i + 1;

            // Check if current fort at index i is not a zero.
            if (forts[i] != 0) {
                // Find the next non-zero fort.
                while (nextFortIndex < n && forts[nextFortIndex] == 0) {
                    ++nextFortIndex;
                }

                // Check if the end of the array hasn't been reached and if the sum of the
                // two non-zero forts is zero, implying opposite teams.
                if (nextFortIndex < n && forts[i] + forts[nextFortIndex] == 0) {
                    // Calculate the current number of zeroes.
                    int zeroCount = nextFortIndex - i - 1;

                    // Update maximum zeroes if the current count exceeds it.
                    maxZeroes = Math.max(maxZeroes, zeroCount);
                }
            }

            // Move to the next potential fort.
            i = nextFortIndex;
        }

        // Return the maximum number of zeroes between two capturing forts.
        return maxZeroes;
    }
}
```

C++

```cpp
class Solution {
public:
    // This function calculates the maximum number of consecutive fortresses captured.
    int captureForts(vector<int>& forts) {
        // Get the number of fortresses
        int fortCount = forts.size();
        // Initialize the answer to 0
        int maxCaptures = 0;
        // Start iterating through the fortresses
        int currentIndex = 0;

        // Loop through all the fortresses
        while (currentIndex < fortCount) {
            // Next index starts just after the current one
            int nextIndex = currentIndex + 1;

            // Only check fortresses that have not been captured (non-zero values)
            if (forts[currentIndex] != 0) {
                // Skip all the captured fortresses (zeros) until a non-captured fortress is found
                while (nextIndex < fortCount && forts[nextIndex] == 0) {
                    ++nextIndex;
                }
                // Check if we found a fortress and the sum of the current and next fortresses is zero
                if (nextIndex < fortCount && forts[currentIndex] + forts[nextIndex] == 0) {
                    // Calculate the number of fortresses between current and next (exclusive)
                    int captures = nextIndex - currentIndex - 1;
                    // Update max captures if this is the largest so far
                    maxCaptures = max(maxCaptures, captures);
                }
            }
            // Move to the next index (could be the next non-captured fortress or the end of vector)
            currentIndex = nextIndex;
        }

        // Return the maximum number of fortresses that can be captured consecutively
        return maxCaptures;
    }
};
```

TypeScript

```typescript
function captureForts(forts: number[]): number {
    const fortCount = forts.length; // Total number of forts
    let maxDistance = 0; // This will hold the maximum distance between two forts
    let currentIndex = 0; // Index to traverse the forts array

    // Loop through the array of forts
    while (currentIndex < fortCount) {
        let nextIndex = currentIndex + 1; // Index for the next fort

        // Ensure the current fort is not zero (zero implies the fort has already been captured)
        if (forts[currentIndex] !== 0) {

            // Find the next non-zero fort
            while (nextIndex < fortCount && forts[nextIndex] === 0) {
                nextIndex++;
            }

            // Check if the current and next non-zero forts sum up to zero and if true, calculate the distance between them
            if (nextIndex < fortCount && forts[currentIndex] + forts[nextIndex] === 0) {
                // Update the maximum distance if the current distance is greater
                maxDistance = Math.max(maxDistance, nextIndex - currentIndex - 1);
            }
        }

        // Move to the next segment of forts
        currentIndex = nextIndex;
    }

    // Return the maximum distance found
    return maxDistance;
}


class Solution:
    def captureForts(self, forts: List[int]) -> int:
        n = len(forts)  # Get the length of the forts list.
        i = ans = 0  # Initialize pointer i to 0 and answer 'ans' to 0.

        # Iterate through the list of forts.
        while i < n:
            j = i + 1  # Set the pointer j to the next position after i.

            # If the current fort at position i is a friendly fort (non-zero).
            if forts[i]:
                # Find the next friendly fort (non-zero) starting from position j.
                while j < n and forts[j] == 0:
                    j += 1

                # If such a fort is found and the sum of the values at i and j is 0,
                # which means they cancel each other out.
                if j < n and forts[i] + forts[j] == 0:
                    # Calculate the distance between the forts, excluding the start and end points,
                    # and update the answer 'ans' if this distance is greater.
                    ans = max(ans, j - i - 1)

            # Move the starting pointer i to the position of j for the next iteration.
            i = j

        # Return the maximum number of captured forts.
        return ans
```

Time and Space Complexity

The provided Python function `captureForts` calculates the maximum distance between pairs of forts, with the condition that the sum of the strengths of a pair of forts is zero, and there are only zeroes between them.

Time Complexity:

The time complexity of the function is `O(n)`, where `n` is the length of the `forts` list. This is because the function iterates through the list only once with a single while-loop. Within the while-loop, the inner while-loop also iterates through portions of the list, but overall each element is visited at most once by either the outer or the inner loop. There are no nested loops that depend on the size of the input, hence the linear time complexity.

Space Complexity:

The space complexity is `O(1)` since there are only a constant number of variables used (`n`, `i`, `j`, `ans`) that do not depend on the size of the input list. No additional space that grows with the input size is allocated during the execution of the function.