2498. Frog Jump II

Medium Greedy Array Binary Search Leetcode Link

Problem Description

You have an integer array `stones` where each element represents the position of a stone in a river, and the array is sorted in strictly increasing order. A frog starts on the first stone and wants to make it to the last stone before returning to the first stone. In doing so, the frog is allowed to jump to any stone, but only once per stone.

The length of a jump is calculated as the absolute difference between the positions of the current stone and the stone the frog jumps to. Thus, if the frog jumps from `stones[i]` to `stones[j]`, the length of the jump is `|stones[i] - stones[j]|`.

A path's cost is determined by the longest (maximum length) jump the frog makes on its way from the first stone to the last stone and back. The goal is to determine the minimum cost of a path, meaning, to find out the smallest possible maximum jump length that the frog can achieve while still reaching the last stone and coming back to the first one.

Intuition

The intuition behind the solution lies in understanding how the cost of a path is defined and making strategic jumps that minimize the maximum jump length. The fact that the area we cover is the river between the first and last stone, and we eventually need to return to the start, it is reasonable to consider that the costliest jumps will be at the beginning or end of the route. This is because the stones are sorted and there are no gaps, so the largest jumps will be between stones that are the farthest apart.

We notice that the frog has two critical long jumps: the first jump into the river and the last jump out of the river. The first jump can only be between `stones[0]` and `stones[1]`, and the last jump can be between any two consecutive stones since the frog must return to `stones[0]`.

Hence, we arrive at the idea of checking every pair of stones that could represent the last jump (every two consecutive stones), and for each such pair, calculate the longest jump that occurs if the frog were to make that particular pair its last jump sequence. We can then extract which of these calculated longest jumps represents the smallest maximum jump, yielding the minimum cost of a path.

The implementation of the solution iterates through the `stones` array, starting from the second index (since the first jump is predefined), and compares the length of the jump two stones apart, updating the answer to the maximum jump length seen so far. This effectively accounts for the costliest jump the frog makes if it decided to make that pair of stones the last jump of its return trip. By the end of the iteration, since all potential last jump pairs are accounted for, the `ans` variable holds the value of the minimum cost path.

Solution Approach

The solution uses a simple for-loop to iterate through the array of `stones` from the second stone onwards, and it performs a comparison operation for each element to find the maximum jump length.

The core algorithm doesn't utilize advanced data structures or complex patterns but relies on understanding the problem and employing a simple iterative approach to solve it.

Here's a breakdown of the implementation steps with a focus on understanding the algorithm and patterns used:

1. Initialize `ans` with the difference between the first two stones, since the frog's first jump from `stones[0]` to `stones[1]` is predetermined and represents the initial cost.
2. Iterate through the array of stones starting from index 2 (the third stone). For each stone at index `i`, consider the previous stone at index `i-1` and the one before it at index `i-2`.
3. Calculate the length of the jump from `stones[i - 2]` to `stones[i]` by taking the absolute difference: `stones[i] - stones[i - 2]`. This represents the potential maximum jump length if `stones[i - 1]` and `stones[i]` are considered as the last jump on the way back.
4. Update the `ans` if the current jump length is greater than the previously recorded maximum jump length. This is done using the `max` function.
5. Repeat this process until all possible jump lengths (for when each pair of consecutive stones could be the frog's last jump back to the start) have been considered.
6. At the end of the loop, `ans` holds the minimum cost of the path that the frog can take.

The reference solution doesn't apply complex algorithms or use additional data structures; it applies a direct approach that uses only simple array indexing and built-in functions. This illustrates an important pattern in many coding challenges: sometimes the straightforward solution is the most effective and efficient one.

Here's the essence of the code snippet that achieves this:

```python
def maxJump(self, stones: List[int]) -> int:
    ans = stones[1] - stones[0]
    for i in range(2, len(stones)):
        ans = max(ans, stones[i] - stones[i - 2])
    return ans
```

Note that the code above assumes that the input will always have more than two elements as per the constraints implied in the problem description.

Example Walkthrough

Let us consider a small example to illustrate the solution approach. Suppose we have the following integer array `stones`: `[2, 4, 7, 10]`, where each element signifies the stone's position in a river.

Following the logic described above:

1. Initialize `ans` with the difference between the first two stones.
   - `ans = stones[1] - stones[0]`
   - `ans = 4 - 2`
   - `ans = 2`
2. Proceed through the stones to find the maximum jump required if the last jump back were between the second stone and third stone, then the third stone and fourth stone.
3. We make a jump from `stones[0]` to `stones[2]` (from position 2 to position 7):
   - Current jump length = `stones[2] - stones[0]`
   - Current jump length = `7 - 2`
   - Current jump length = `5`
4. Since 5 (current jump length) > 2 (`ans`), we update `ans`.
   - `ans = 5`
5. Next, consider the stones at positions `stones[1]`, `stones[2]`, and `stones[3]` to simulate if the last jump is between `stones[2]` and `stones[3]`.
   - Current jump length would be between `stones[1]` and `stones[3]`: from position 4 to position 10.
   - Current jump length = `stones[3] - stones[1]`
   - Current jump length = `10 - 4`
   - Current jump length = `6`
6. Since 6 (current jump length) > 5 (`ans`), we update `ans` again.
   - `ans = 6`
7. We have now considered all possible last jump pairs, and thus, the `ans` value is the minimum possible maximum jump length the frog can achieve when completing its path.
   - The final `ans` = `6`

So, the minimum cost of a path which is the smallest maximum jump the frog has to make in this case is `6`.

Python Solution

```python
from typing import List

class Solution:
    def maxJump(self, stones: List[int]) -> int:
        # Initialize the maximum jump distance as the distance between the first two stones.
        max_jump = stones[1] - stones[0]

        # Iterate over the stones starting from the third stone.
        for i in range(2, len(stones)):
            # For each stone, calculate the jump distance from the stone two positions before it.
            # Update the maximum jump distance if the current distance is larger.
            max_jump = max(max_jump, stones[i] - stones[i - 2])

        # Return the maximum jump distance found.
        return max_jump

```

Java Solution

```java
class Solution {

    // Method to calculate the maximum jump distance between consecutive stones
    public int maxJump(int[] stones) {
        // Initialize the maximum jump to the distance between the first two stones
        int maxJumpDistance = stones[1] - stones[0];

        // Loop through the array starting from the third stone
        for (int i = 2; i < stones.length; ++i) {
            // Calculate the jump distance between the current stone and the stone two steps back
            int jumpDistance = stones[i] - stones[i - 2];

            // Update the maximum jump distance if the current jump is greater
            maxJumpDistance = Math.max(maxJumpDistance, jumpDistance);
        }

        // Return the maximum jump distance found
        return maxJumpDistance;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include the algorithm header for the 'max' function

class Solution {
public:
    // Function to determine the maximum jump between adjacent stones
    int maxJump(vector<int>& stones) {
        // Check if there are less than two stones, return 0 if true,
        // as no jump can be made with only one stone or no stones
        if (stones.size() < 2) {
            return 0;
        }

        // Initialize 'max_jump' to the jump between the first two stones
        int max_jump = stones[1] - stones[0];

        // Iterate from the third stone to the end of the vector
        for (int i = 2; i < stones.size(); ++i) {
            // Update 'max_jump' to be the maximum between its current value and
            // the difference between the current stone and the stone two places before
            max_jump = max(max_jump, stones[i] - stones[i - 2]);
        }

        // Return the maximum jump found
        return max_jump;
    }
};

```

Typescript Solution

```typescript
// Import the algorithm's max function equivalent in TypeScript
import { max } from 'lodash';

// Function to determine the maximum jump between adjacent stones
function maxJump(stones: number[]): number {
    // Check if there are fewer than two stones, return 0 if true,
    // as no jump can be made with only one stone or no stones
    if (stones.length < 2) {
        return 0;
    }

    // Initialize 'maxJump' to the jump between the first two stones
    let maxJump = stones[1] - stones[0];

    // Iterate from the third stone to the end of the array
    for (let i = 2; i < stones.length; i++) {
        // Update 'maxJump' to be the maximum between its current value and
        // the difference between the current stone and the stone two places before
        maxJump = max([maxJump, stones[i] - stones[i - 2]]);
    }

    // Return the maximum jump found
    return maxJump;
}

// Example on how to use the function
// let stones = [0, 3, 5, 9, 10];
// let result = maxJump(stones);
// console.log(result); // This would print the result of the maximum jump.

```

Time and Space Complexity

The given Python code snippet defines a method `maxJump` that determines the maximum jump between consecutive or alternate stones in a list of stones represented by their positions in `stones`.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the `stones` list. This is because there is a single `for` loop iterating through the `stones` array, starting at the second index and performing a constant time operation (calculating the maximum jump and updating `ans`) during each iteration.

Space Complexity

The space complexity of the code is `O(1)`, which means it is constant space complexity. Apart from the input list itself, the only additional storage used is a single variable `ans` to keep track of the current maximum jump, which does not rely on the size of the input.