45. Jump Game II

Medium Greedy Array Dynamic Programming

Problem Description

In this problem, you're given an array of integers called `nums` where each element represents the maximum length of a forward jump you can make from that index. Your goal is to find out the minimum number of jumps required to reach the last element of the array, starting from the first element.

Imagine standing on a series of stepping stones stretched across a river, where each stone represents an index in the array and the number on the stone is the maximum number of stones you can skip forward with your next step. You want to cross the river using as few steps as possible, and all the stones are placed so it's guaranteed you can reach the other side.

The constraints are as follows:

- You are always starting at the first index.
- You can choose any number of steps to move forward as long as it doesn't exceed the number specified in the current index.
- You must reach to or beyond the final index of the array.

This is a path-finding problem where we're not only interested in reaching the end but also in minimizing the number of steps we take to get there.

Intuition

The solution to this problem lies in a greedy strategy, which means taking the best immediate step without worrying about the global optimum at each move.

First, understand that since you can always reach the last index, there's no need to worry about getting stuck -- it's only a matter of how quickly you can get there. Also, the farthest you can reach from the current step should always be considered for making a jump, as it gives more options for the next move.

With this in mind, you keep track of two important positions:

- The farthest you can reach (`mx`) from the current index or from a group of indexes you consider during a jump.
- The last position you jumped to (`last`), which is your starting line for considering where to jump next.

As you progress through the array, you keep updating `mx` to be the maximum of itself and the farthest you can reach from the current index (`i + nums[i]`). When you reach the `last` index, it means you've exhausted all possibilities for the current jump and must make another jump. This is when you set `last` to `mx`, because `mx` is the new maximum range of positions you can jump to from your current position, and increment the number of jumps you've made.

By iterating through the array with this strategy, you can determine the minimum number of jumps needed to reach the end of the array.

Solution Approach

The implementation of this solution uses the greedy algorithm approach to determine the minimum number of jumps. The greedy algorithm makes the locally optimal choice at each step with the hope of finding the global optimum, and in this problem, it aims to reach the farthest index possible with each jump, thereby minimizing the total number of jumps.

The following steps and variables are used to implement this strategy:

- `ans` is a counter to keep track of the number of jumps made.
- `mx` is a variable that holds the maximum distance that can be reached at the current position.
- `last` is a variable that keeps track of the farthest index reached with the last jump.

The algorithm works as follows:

1. Initialize `ans`, `mx`, and `last` to 0. These will store the current number of jumps, the maximum reachable index, and the last jumped-to index, respectively.
2. Iterate over the array `nums` except for the last element, because from the second to last element, you are always one jump away from the end:
   - Update `mx` to be the maximum of itself and the sum of the current index `i` and the current element's value `nums[i]`. This effectively sets `mx` to the farthest reachable index from the current index.
   - Check if the current index `i` is equal to `last`. If true, it means you've reached the maximum index from the previous jump, and you have to jump again.
   - When jumping again (the `if` condition is satisfied):
     - Increment `ans` because you've made a jump.
     - Update `last` to `mx`, as now you will calculate the reachable distance from this new index.

By working through the array with these rules, the `ans` variable reflects the minimum number of jumps needed when the loop ends. This greedy approach ensures that you make a jump only when necessary, which coincides with reaching the maximum distance from the previous jump.

The final result is obtained by returning the value of `ans` after the loop has finished executing. Since the greedy algorithm here always takes the farthest reaching option, it guarantees that the number of jumps is minimized.

Here is the solution code with comments corresponding to each step:

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        ans = mx = last = 0  # Step 1: Initialize variables
        for i, x in enumerate(nums[:-1]):  # Step 2: Iterate over the array except the last element
            mx = max(mx, i + x)  # Update mx to the farthest reachable index
            if last == i:  # Encountered when we reached the max distance from the last jump
                ans += 1  # Increment the number of jumps made
                last = mx  # Update last to the new maximum reachable index
        return ans  # Return the minimum number of jumps required
```

By faithfully following these steps, the code realizes the greedy solution approach and successfully solves the problem of finding the minimum number of jumps.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach using the greedy algorithm as described:

Suppose we have the array `nums = [2, 3, 1, 1, 4]`. We want to find the minimum number of jumps to reach the last element.

- We're starting at the first element (index 0), which is `2`. This means that from index 0, we can jump to index 1 or index 2. Our goal is to reach index 4 (the last index) with the minimum number of jumps.

Here's how the greedy algorithm will walk through this example:

1. Initialize `ans` (number of jumps), `mx` (maximum reachable index), and `last` (last jumped-to index) to 0.
   - `ans = 0`
   - `mx = 0`
   - `last = 0`
2. Begin iterating from index 0 to the second-to-last index, which is index 3 in this case. For each index `i`, we will do the following:

- **Index 0:**
  - The maximum we can reach from here is `0 + nums[0] = 0 + 2 = 2`. So `mx` is now 2.
  - We haven't reached `last` yet, so we don't increment `ans` or update `last`.
- **Index 1:**
  - Here, we can reach `1 + nums[1] = 1 + 3 = 4`, which is the last index. Now `mx` is updated to 4.
  - We've reached `last` (which is still 0), so it's time to make a jump.
    - Increment `ans` to 1. (`ans = 1`)
    - Update `last` to `mx`, which is 4. (`last = 4`)
- **Index 2:**
  - We can reach `2 + nums[2] = 2 + 1 = 3`, but `mx` is already 4, so there's no need to update `mx`.
  - We have not yet reached the new `last` (which is now 4), so no jumps are made.
- **Index 3:**
  - We can reach `3 + nums[3] = 3 + 1 = 4`, but again, `mx` is already 4.
  - We have not yet reached the new `last` (which is still 4), so no jumps are made.

Using the greedy approach, we've found that we can reach the last index with only 1 jump from index 1 directly to index 4. Therefore, the minimum number of jumps required to reach the last element is 1.

- The final answer `ans` is returned, which contains the value 1, indicating the minimum number of jumps needed to reach the end of the array.

The greedy algorithm works optimally in this case by maximizing the reach with every jump and only increasing the jump count when it is necessary.

Python Solution

```python
from typing import List

class Solution:
    def jump(self, nums: List[int]) -> int:
        # Initialize the jump count, the maximum reach, and the edge of the current range to 0.
        jump_count = max_reach = last_reach = 0

        # Iterate over the list excluding the last element as it's unnecessary to jump from the last position.
        for index, value in enumerate(nums[:-1]):
            # Update the maximum reach with the furthest position we can get to from the current index.
            max_reach = max(max_reach, index + value)

            # If we have reached the furthest point to which we had jumped previously,
            # increment the jump count and update the last reached position to the current max_reach.
            if last_reach == index:
                jump_count += 1
                last_reach = max_reach

        # Return the minimum number of jumps needed to reach the end of the list.
        return jump_count
```

Java Solution

```java
class Solution {
    public int jump(int[] nums) {
        int steps = 0;          // Initialize a step counter to keep track of the number of jumps made
        int maxReach = 0;       // Initialize the maximum reach from the current position
        int lastJumpMaxReach = 0;  // Initialize the maximum reach of the last jump

        // Iterate through the array except for the last element,
        // because we want to reach the last index, not jump beyond it
        for (int i = 0; i < nums.length - 1; ++i) {
            // Update the maximum reach by taking the maximum between the current maxReach
            // and the position we could reach from the current index (i + nums[i])
            maxReach = Math.max(maxReach, i + nums[i]);

            // If the current index reaches the last jump's maximum reach,
            // it means we have to make another jump to proceed further
            if (lastJumpMaxReach == i) {
                steps++;            // Increment the step counter because we're making another jump

                lastJumpMaxReach = maxReach;  // Update the last jump's max reach to the current maxReach

                // There's no need to continue if the maximum reach is already beyond the last index,
                // as we are guaranteed to end the loop
                if (maxReach >= nums.length - 1) {
                    break;
                }
            }
        }

        // Return the minimum number of jumps needed to reach the last index
        return steps;
    }
}
```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to return the minimum number of jumps to reach the end of the array.
    int jump(vector<int>& nums) {
        int numJumps = 0;          // Number of jumps made.
        int currentMaxReach = 0;   // Max index we can reach with the current number of jumps.
        int lastMaxReach = 0;      // Max index we can reach from the last jump.

        // Iterate over each element in the array except the last one.
        for (int i = 0; i < nums.size() - 1; ++i) {
            // Update the furthest index we can reach from here.
            currentMaxReach = max(currentMaxReach, i + nums[i]);

            // If we're at the last index reached by the last jump,
            // it's time to make another jump.
            if (lastMaxReach == i) {
                numJumps++;
                lastMaxReach = currentMaxReach;  // Update the max reach for the new jump.
            }
        }

        return numJumps;  // Return the total number of jumps made to reach the end.
    }
};
```

Typescript Solution

```typescript
function jump(nums: number[]): number {
    // Initialize the variables:
    // jumps to keep track of the minimum number of jumps needed
    // maxReach: the max index that can be reached
    // currentEnd: the last index of the current jump
    let [jumps, maxReach, currentEnd] = [0, 0, 0];

    // Iterate over the array excluding the last element because
    // no jump is needed from the last element
    for (let index = 0; index < nums.length - 1; ++index) {
        // Update maxReach to be the furthest index reachable from here
        maxReach = Math.max(maxReach, index + nums[index]);

        // When we reach the currentEnd, it means we've made a jump
        if (currentEnd === index) {
            // Increment the jump count
            ++jumps;
            // Move the currentEnd to the furthest index reachable
            currentEnd = maxReach;
        }
    }

    // Return the minimum number of jumps needed to reach the end of the array
    return jumps;
}
```

Time and Space Complexity

The given Python code implements a greedy algorithm to solve the jump game problem, where you're required to find the minimum number of jumps needed to reach the last index in the array.

The time complexity is `O(n)` because there is a single for loop that goes through the `nums` array once. Even though there is a maximum calculation within the loop, this does not change the overall linear time complexity, as it only takes constant time to calculate the maximum of two numbers.

The space complexity is `O(1)` because the algorithm only uses a fixed number of variables (`ans`, `mx`, `last`) and does not allocate any additional data structures that grow with input size. This means that the space required by the algorithm does not increase with the size of the input array `nums`.