55. Jump Game

Medium Greedy Array Dynamic Programming

Problem Description

You are given an array `nums`, where each element represents the maximum number of steps you can jump forward from that position. Your goal is to determine if you can reach the last element of the array starting from the first element. You start at the first index of the array, and at each step, you can jump forward to any position within your current element's range. If you can make it to any index that has enough range to reach the end, or if the end is within your reach from where you are, the answer is `true`. Otherwise, it's `false`.

Intuition

The key intuition behind the solution is to take a greedy approach. This means we want to make the locally optimal choice at each step in the hope that this will lead to the globally optimal solution. In this case, as we traverse the array, we continuously keep track of the furthest we can reach (`mx`).

We start at the first index and iterate through the array. For each index, we update `mx` to be the maximum of its current value or the furthest we can get from this index (which is the current index `i` plus the jump length `nums[i]`). As we do this, if we find `mx` is ever less than `i`, we cannot reach position `i` or beyond, which means we cannot reach the last index, and return `false`. If we can always reach the position we're iterating over, by the end of the iteration, we can reach the end, and we return `true`.

Solution Approach

The solution uses a simple but effective **greedy algorithm**. The implementation of this solution does not require any complex data structures, relying only on a single integer variable `mx` which keeps track of the maximum index that can be reached at each step.

The following steps summarize the implementation of the solution:

1. We initialize a variable `mx` to 0. `mx` represents the maximum index that can be reached so far.
2. We iterate through each index `i` of the `nums` array using a `for` loop. The `enumerate` function is handy here as it gives us both the index `i` and the value `x` at that index.
3. During each iteration, we first check if `mx` is less than `i`. If this is the case, we return `false` because it indicates that we can't reach the current index (or any index beyond it).
4. If `mx` is not less than `i`, it means we can reach this position, and therefore we update `mx` to be the maximum of its current value and `i + x`, where `x` is the maximum jump length from the current position. The expression `max(mx, i + x)` efficiently accomplishes this.
5. After the loop, if we never encountered a situation where `mx < i`, we have been able to reach or surpass each index, including the last index. Thus, we return `true`.

The code or mathematical formulas are encapsulated using backticks, for example, `mx = max(mx, i + nums[i])`.

By always keeping track of the furthest reachable index and bailing out early if we find an unreachable index (`mx < i`), we ensure an efficient solution with a time complexity that is linear with respect to the length of the `nums` array.

Example Walkthrough

Let's walk through a small example using the greedy solution approach outlined above. Suppose we are given the following array `nums`:

```
nums = [2, 3, 1, 1, 4]
```

Our task is to determine if we can reach the last element, which in this case is `4`. We will use the steps of the solution to illustrate how we can arrive at the answer:

1. Initialize `mx` to `0`. This means before we start, the furthest index we can reach is `0`.
2. Start iterating:
   - At index `0`, the value is `2`. We can jump to index `1` or `2`. Update `mx` to `max(0, 0 + 2)` which is `2`.
   - Move to index `1`, with a value of `3`. Now we can reach as far as `1 + 3 = 4`, which is the end of the array. Update `mx` to `max(2, 1 + 3)` which is `4`.
   - At this point, we can already reach the last element, so we could stop and return `true`. Nonetheless, for completeness:
   - At index `2`, the value is `1`. From here, `max(4, 2 + 1)` is still `4`.
   - At index `3`, the value is `1`. From here, `max(4, 3 + 1)` is still `4`.
3. Throughout the iteration, `mx` was always greater than or equal to `i`, so we never returned `false`.
4. After the loop, since there were no indices that we could not reach, and we were always able to update `mx` to reach or go beyond the next index, we can return `true`.

So, in our example, it is indeed possible to reach the last element starting from the first element following the greedy approach.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def canJump(self, nums: List[int]) -> bool:
        # Initialize the variable `max_reach` which represents the maximum
        # index we can reach so far.
        max_reach = 0

        # Enumerate through the list, with `index` representing the current
        # position and `jump_length` the maximum jump length from that position.
        for index, jump_length in enumerate(nums):
            # If our current `max_reach` is less than the `index`, we can't
            # jump to `index` or beyond, hence we return False.
            if max_reach < index:
                return False

            # Update `max_reach` by the farthest we can get from here, which
            # is either the current `max_reach` or `index + jump_length`.
            max_reach = max(max_reach, index + jump_length)

        # If we've gone through all elements without returning False, it means
        # we can reach the end of the list, so we return True.
        return True
```

Java

```java
class Solution {
    public boolean canJump(int[] nums) {
        int maxReachable = 0; // Initialize the maximum reachable index to 0

        // Iterate over each index in the array
        for (int i = 0; i < nums.length; ++i) {
            // If the current index is greater than the maximum reachable index,
            // it means we cannot proceed further, so return false.
            if (maxReachable < i) {
                return false;
            }

            // Update the maximum reachable index if the reachable index
            // from the current position is greater than the previous max.
            maxReachable = Math.max(maxReachable, i + nums[i]);
        }

        // If the loop completes without returning false, it means we can
        // reach the last index, so return true.
        return true;
    }
}
```

C++

```cpp
#include <vector> // Include the necessary header for vector
using namespace std;

class Solution {
public:
    // Function to check if we can jump to the last index of the vector 'nums'
    bool canJump(vector<int>& nums) {
        int maxReachable = 0; // Variable to keep track of the maximum reachable index so far

        // Iterate through each element of the vector
        for (int i = 0; i < nums.size(); ++i) {
            // If the current index is greater than the maximum reachable index, we can't proceed, return false
            if (maxReachable < i) {
                return false;
            }

            // Update maxReachable to the maximum of the current maxReachable and the current index plus its jump length
            maxReachable = max(maxReachable, i + nums[i]);
        }

        // If we are able to iterate through the entire vector, return true
        return true;
    }
};
```

TypeScript

```typescript
// Determines if it is possible to reach the last index of the array
// nums[i] represents the maximum jump length from that position.
function canJump(nums: number[]): boolean {
    let maxReach: number = 0; // Variable to keep track of the maximum reachable index
    for (let currentIndex = 0; currentIndex < nums.length; ++currentIndex) {
        // Check if the current index is beyond the maximum reachable index
        if (maxReach < currentIndex) {
            // If we cannot reach this index, return false
            return false;
        }
        // Update the maximum reachable index
        maxReach = Math.max(maxReach, currentIndex + nums[currentIndex]);
    }
    // If we can reach beyond the last index, return true
    return true;
}
```

```python
from typing import List

class Solution:
    def canJump(self, nums: List[int]) -> bool:
        # Initialize the variable `max_reach` which represents the maximum
        # index we can reach so far.
        max_reach = 0

        # Enumerate through the list, with `index` representing the current
        # position and `jump_length` the maximum jump length from that position.
        for index, jump_length in enumerate(nums):
            # If our current `max_reach` is less than the `index`, we can't
            # jump to `index` or beyond, hence we return False.
            if max_reach < index:
                return False

            # Update `max_reach` by the farthest we can get from here, which
            # is either the current `max_reach` or `index + jump_length`.
            max_reach = max(max_reach, index + jump_length)

        # If we've gone through all elements without returning False, it means
        # we can reach the end of the list, so we return True.
        return True
```

Time and Space Complexity

The given Python code aims to determine whether it is possible to jump to the last index of the given list `nums`. The function `canJump` works by iterating through each element in the list, calculating the maximum distance that can be reached from the current position, and checking whether that distance is sufficient to continue progressing through the array.

- **Time Complexity**: The time complexity of the code is `O(n)`, where `n` is the length of the array `nums`. This is because the function involves a single loop that goes through the array once, making a constant-time check and update at each step.
- **Space Complexity**: The space complexity of the code is `O(1)`. The algorithm uses a fixed amount of additional space (the variable `mx`), regardless of the input size, so the space used does not grow with the size of the input array.