2770. Maximum Number of Jumps to Reach the Last Index

Medium Array Dynamic Programming Leetcode Link

Problem Description

The given problem presents us with a particular type of jumping puzzle. You have an array called `nums` which consists of `n` integers, indexed from `0` to `n-1`. You also have a value called `target`. Starting at the first element of the array (index `0`), you can jump to any later element in the array (from index `i` to index `j`, where `i < j`) as long as the absolute difference between the values of `nums[i]` and `nums[j]` is less than or equal to `target`. The question is to determine the maximum number of jumps that can be made to reach the last element of the array (`n-1` index).

If, at any point, there are no legal jumps to make, which means you cannot reach the last index of the array, the function should return `-1`.

In essence, the problem is asking for the furthest reach in the array through a series of legal jumps, where each jump abides by the rule concerning the allowable difference defined by the `target` value.

Intuition

The solution approach relies on the idea of recursion and dynamic programming. We can define a recursive function, say `dfs(i)`, that computes the maximum number of jumps needed to reach the end of the array starting from the current index `i`.

Initially, we might consider simply iterating through the array, and at each step, trying each possible legal jump. However, this brute-force approach can be highly inefficient as it involves many repeated calculations.

Instead, we can use a technique called memoization, which is a strategy to store the results of expensive function calls and return the cached result when the same inputs occur again. Thus, for each index `i`, we remember the maximum number of jumps we can make. If we revisit the same index `i`, we don't recalculate; we simply use the stored value.

Here's how the thought process goes for the given solution:

1. If we are at the last index (`n - 1`), return `0` because we don't need any more jumps;
2. If we are at any other index `i`, look at all potential jumps, i.e., loop from `i + 1` to `n - 1` and find indexes `j` to jump to where `|nums[i] - nums[j]| <= target`;
3. Use our recursive function `dfs(j)` to compute the maximum number of jumps from index `j` to the end. The answer for our current position `i` would then be the maximum value of `1 + dfs(j)` over all legal `j`'s plus one (for the jump we are currently considering);
4. If we cannot jump anywhere from `i`, we use negative infinity to mark that we can't reach the end from this index;
5. We use memoization (`@cache` decorator) to avoid re-computation for indexes we have already visited.
6. Finally, we initiate our recursive calls with `dfs(0)` to start the process from the first index. The result is either the maximum number of jumps or `-1` if the end is unreachable (`ans < 0`).

This dynamic programming approach, combined with memoization (caching), provides an efficient solution to what could otherwise be a very time-consuming problem if solved with plain recursion or brute-forcing.

Solution Approach

The implementation uses a recursive depth-first search (DFS) approach in combination with memoization. The core of this approach is the `dfs` function, which solves the problem for a given index `i` and provides the maximum number of jumps that can be made from that index to the end. To prevent re-computation of results for each index `i`, memoization is used via the `@cache` decorator provided by Python's standard library. Here's an outline of how the algorithm works:

- A helper function `dfs(i)` is defined:
  - If `i` is the last index (`n - 1`), return `0` because no more jumps are necessary.
  - If not, initialize a variable `ans` to negative infinity, symbolizing that the end is not yet reachable from `i`.
  - Loop through each potential target index `j` (where `j` goes from `i + 1` to `n - 1`), and for each target index, check if the jump from `i` to `j` is legal; that is, if `abs(nums[i] - nums[j]) <= target`.
  - For every legal jump, use the `dfs` function to compute the maximum number of jumps from index `j` to the end. The value of `ans` is updated to be the maximum of the current `ans` and `1 + dfs(j)`, which represents making one jump to index `j` plus the maximum jumps from `j` to the end.
- The global scope begins by determining `n`, the length of the `nums` array.
- It then calls the helper function `dfs(0)` to initiate the recursive jumping process from the first element of the array.
- Lastly, the function returns `-1` if `ans < 0` indicating that the last index is unreachable or `ans` if the end can be reached, providing the maximum number of jumps to reach the end.

Data structures used:

- The `nums` array holds the input sequence of integers.
- An implicit call stack for recursive function calls.
- An internal cache for memoization, which is abstracted away by the `@cache` decorator but essentially behaves as a hash map storing computed results keyed by function arguments.

Algorithms and patterns used:

- **DFS (Depth-First Search):** This recursive strategy is used to explore all possible jumping paths.
- **Memoization:** This pattern avoids recalculating `dfs(i)` for any index by caching the result. When `dfs(i)` is called multiple times with the same `i`, it returns the cached result instead of recalculating.
- **Dynamic Programming (Top-Down Approach):** The problem is broken down into smaller subproblems (`dfs(j)` for each `j`) and solved in a recursive manner. The overlapping subproblems are handled efficiently using memoization.

In summary, the solution employs a combination of recursive DFS and memoization in a top-down dynamic programming framework to efficiently compute the maximum number of jumps to the end of the array.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following input:

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

We want to find the maximum number of jumps to reach the end of the array.

1. We call `dfs(0)`, starting at index `0`, where the value is `2`. Our first action is to look for all indices we can jump to from `0`.
2. We can only jump to an index `i` if `abs(nums[0] - nums[i]) <= target`. In this case, we can jump from index `0` (`nums[0] = 2`) to index `1` (`nums[1] = 3`) because `abs(2 - 3) = 1 <= target`.
3. Now at index `1`, we call `dfs(1)` and look for a jump. We can jump from `1` to `2` and `1` to `3` since both satisfy the condition (`abs(3 - 1) <= 1` and `abs(3 - 1) <= 1`).
4. At index `2`, calling `dfs(2)` returns `0`, as we can jump directly to the end (index `4`) from here (`abs(1 - 4) <= 1`).
5. At index `3`, calling `dfs(3)` also returns `0`, for the same reason as `dfs(2)`.
6. Backtracking, we can see that from `dfs(1)`, we have two possible destinations to consider: index `2` and index `3`. Choosing index `2` lets us reach the end in one jump (the same for index `3`), so we update our maximum jumps from `1` to `2`.
7. Finally, considering jumps from `dfs(0)` to `dfs(1)`, we have now found that from `dfs(1)` we can reach the end with `2` jumps, hence we add one more jump (the jump from `0` to `1`) and update our answer to `3`.
8. The recursive function will use memoization to ensure that if we call `dfs(2)` or `dfs(3)` again during exploration, it will return the cached result without recalculating it.
9. If there were no possible jumps at any point, `dfs(i)` would return negative infinity to indicate that it's impossible to proceed. But in this case, we can reach the end, and the maximum number of jumps is `3`.

Hence, `dfs(0)` will finally return `3` as the maximum number of jumps needed to reach the end of the array from the first element.

Using this approach, the solution capitalizes on the efficiency of memoization to avoid re-computing the same values, thereby reducing the time complexity significantly from what would be seen in a brute-force approach.

Python Solution

```python
from functools import lru_cache
from math import inf

class Solution:
    def maximum_jumps(self, nums: List[int], target: int) -> int:
        # This function seeks the maximum number of jumps
        # satisfying the constraint where the absolute difference
        # between the values at the current index and the chosen jump index
        # is less than or equal to the target value.

        # Cache the results of the dfs function to avoid recomputation
        @lru_cache(maxsize=None)
        def dfs(current_index: int) -> int:
            # Base case: If we're at the last index, no more jumps are possible
            if current_index == num_elements - 1:
                return 0

            # Initialize the maximum jumps from the current index to negative infinity
            max_jumps_from_current = -inf

            # Iterate over possible jumps from the current index
            for next_index in range(current_index + 1, num_elements):
                # If the jump is valid (difference within the target)
                if abs(nums[current_index] - nums[next_index]) <= target:
                    # Update max jumps from the current index by trying the jump
                    # and adding 1 to the result of subsequent jumps
                    max_jumps_from_current = max(max_jumps_from_current, 1 + dfs(next_index))

            # Return the computed maximum jumps from the current index
            return max_jumps_from_current

        # Find out the number of elements in nums list
        num_elements = len(nums)

        # Start the depth-first search from the first index
        max_jumps = dfs(0)

        # If max jumps is negative, return -1 indicating no valid jumps sequence exists
        # Otherwise, return the computed max jumps
        return -1 if max_jumps < 0 else max_jumps
```

Note that `lru_cache` from Python's `functools` is used to cache results, replacing the `@cache` decorator, which is not standard in Python 3 before version 3.9. Also, the variable `ans` is renamed to `max_jumps_from_current` to be more descriptive, and `n` is renamed to `num_elements` to better communicate its meaning.

Import statements for `List` and `inf` should also be added:

```python
from typing import List  # Import List type for type hinting

```

Java Solution

```java
class Solution {
    private Integer[] memo;  // Memoization array to store the maximum jumps from each position
    private int[] nums;      // Array of numbers representing positions
    private int n;           // Total number of positions
    private int target;      // The maximum allowed difference between positions for a valid jump

    // Method to calculate the maximum number of jumps to reach the end, starting from the first position
    public int maximumJumps(int[] nums, int target) {
        n = nums.length;            // Initialize the length of the nums array
        this.target = target;       // Initialize the target difference
        this.nums = nums;           // Assign the input array nums to the instance variable nums
        memo = new Integer[n];      // Initialize the memoization array
        int ans = dfs(0);           // Start a depth-first search from position 0
        return ans < 0 ? -1 : ans; // If the result is negative, no valid sequence of jumps is possible, hence return -1; otherwise
    }

    // Helper method to perform a depth-first search, which calculates the maximum jumps from the current position 'i' to the end
    private int dfs(int i) {
        if (i == n - 1) {    // If the current position is the last one, no jumps are needed so return 0
            return 0;
        }
        if (memo[i] != null) { // If we have already computed the number of jumps from this position, return it
            return memo[i];
        }
        int ans = -(1 << 30);  // Initialize ans with a large negative number to ensure that any positive number will be greater du
        for (int j = i + 1; j < n; ++j) { // Explore all the positions that can be jumped to from position 'i'
            if (Math.abs(nums[i] - nums[j]) <= target) { // If the difference between positions 'i' and 'j' is within the target ra
                ans = Math.max(ans, 1 + dfs(j)); // Update ans with the maximum jumps by considering the jump from 'i' to 'j' and t
            }
        }
        return memo[i] = ans; // Memoize and return the maximum jumps from the current position 'i'
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>  // To use std::function
using namespace std;

class Solution {
public:
    int maximumJumps(vector<int>& nums, int target) {
        int n = nums.size();

        // f will store the maximum jumps from index i to the end
        int max_jumps[n];
        memset(max_jumps, -1, sizeof(max_jumps));  // Initialize with -1

        // Depth-First Search (DFS) to calculate maximum jumps using memoization
        function<int(int)> dfs = [&](int i) {
            if (i == n - 1) {  // Base case: when we reach the last element
                return 0;
            }
            if (max_jumps[i] != -1) {  // Check for already computed result
                return max_jumps[i];
            }
            max_jumps[i] = INT_MIN;  // Initialize this state as minimum value
            for (int j = i + 1; j < n; ++j) {  // Try to jump to every possible next step
                if (abs(nums[i] - nums[j]) <= target) {  // Check if the jump is within the target difference
                    max_jumps[i] = max(max_jumps[i], 1 + dfs(j));  // Recursively find the max jumps from the next index
                }
            }
            return max_jumps[i];  // Return the maximum jumps from index i
        };

        // Start the process from index 0
        int result = dfs(0);

        // If result is negative, it means it's not possible to jump to the end
        return result < 0 ? -1 : result;
    }
};

```

Typescript Solution

```typescript
// Define the maximumJumps function which calculates the maximum number of jumps needed to reach the last index
function maximumJumps(nums: number[], target: number): number {
    // Store the length of the nums array
    const length = nums.length;
    // Initialize an array to store the memoized results of subproblems
    const memo: number[] = Array(length).fill(-1);

    // Define the depth-first search (dfs) helper function
    const dfs = (index: number): number => {
        // If the current index is the last index of the array, no more jumps are needed
        if (index === length - 1) {
            return 0;
        }
        // If this index has been visited before, return the stored result
        if (memo[index] !== -1) {
            return memo[index];
        }
        // Set the current index's assumed minimum jumps to negative infinity
        memo[index] = -(1 << 30);
        // Iterate through the array starting from the current index plus one
        for (let nextIndex = index + 1; nextIndex < length; ++nextIndex) {
            // If the difference between the current index's value and the next index's value is within the target
            if (Math.abs(nums[index] - nums[nextIndex]) <= target) {
                // Update the current index's maximum jumps with the maximum of its current value and one plus the result of dfs at t
                memo[index] = Math.max(memo[index], 1 + dfs(nextIndex));
            }
        }
        // Return the computed number of jumps for the current index
        return memo[index];
    };

    // Start the dfs from the first index to calculate the maximum number of jumps
    const maxJumps = dfs(0);
    // Return the result or -1 if it’s less than 0, indicating that the end is not reachable
    return maxJumps < 0 ? -1 : maxJumps;
}

```

Time and Space Complexity

The given Python code defines a function `maximumJumps` that calculates the maximum number of jumps you can make in a list of integers, where a jump from index `i` to index `j` is valid if the absolute difference between `nums[i]` and `nums[j]` is less than or equal to a given `target`. The analysis of the time complexity and space complexity is as follows:

The time complexity of the `maximumJumps` function is `O(n^2)` where `n` is the length of the input array `nums`. This quadratic time complexity arises because, in the worst case, the recursive function `dfs` is called for every pair of indices `i` and `j` in the array. The outer loop runs once for each of the `n` elements, and the inner loop runs up to `n-1` times for each iteration of the outer loop, hence the `n * (n - 1)` which simplifies to `O(n^2)`.

The space complexity of the function is `O(n)` which is attributable to two factors:

- The recursion depth can go up to `n` in the worst case, where each function call adds a new frame to the call stack.
- The use of the `@cache` decorator on the `dfs` function adds memoization, storing the results of subproblems to prevent re-computation. As there are `n` possible starting positions for jumps, the cache could potentially hold `n` entries, one for each subproblem.

Therefore, the space used by the call stack and memoization dictates the space complexity of `O(n)`.