1340. Jump Game V

Hard Array Dynamic Programming Sorting

Problem Description

The problem presents an array of integers called `arr` alongside an integer `d`. You are tasked with finding the maximum number of indices you can visit starting from any index in the array. From a given index `i`, you can jump to an index `i + x` or `i - x` under two conditions:

1. The value of `x` is between 1 and `d`, and the jump does not take you outside the bounds of the array.
2. You can only make the jump if the number at the starting index `arr[i]` is greater than `arr[j]`, where `j` is the index you want to jump to, and `arr[i]` is also greater than all the numbers between indices `i` and `j`.

You must navigate through the array by following these rules, with the goal being to maximize the number of indices visited.

Intuition

The intuition for solving this problem hinges on dynamic programming, which keeps track of the optimal sub-structure: the maximum number of jumps that can be made from each index. The process can be visualized as follows:

1. We understand that each index is a potential starting point, and from each index, we need to consider jumps in both directions within the allowed range.
2. If we are at index `i`, to determine the optimal number of jumps (`f[i]`) from that index, we need to look at all indices `j` we could jump to within the specified `d` range. If a jump to `j` is permissible, we update `f[i]` to be a maximum between its current value and `f[j] + 1` which accounts for the jump just made.
3. Since we need to ensure we are always jumping to a lower value, we sort the indices of the array based on the values at those indices. This ensures that when we are considering jumps from a particular index `i`, we have already computed the optimal jumps from lower valued indices.
4. In this manner, we build up an array `f` where `f[i]` represents the maximum number of indices that can be visited starting from index `i`. The end goal is to return the maximum value from this `f` array, representing the maximum number of indices visited from the best starting location.

By building up a table of optimal solutions to sub-problems (`f` array) and using the sorted values to iterate in a controlled manner, we can use dynamic programming to compute the solution in an efficient way, without re-computing results. This makes the program more efficient compared to brute-force approaches, which would try to make a jump from every index in an uncontrolled manner and encounter considerable overlap and repeated calculation of the same scenarios.

Solution Approach

The solution for this problem involves a dynamic programming approach. We create a table `f` where each index `i` has an initial value of 1, representing the minimum number of indices that can be visited (the index itself). We'll walk through the implementation approach step by step:

- The problem is framed as a dynamic programming one, which commonly involves filling out a table or array of values. In this case, we're filling out the array `f` with the maximum number of indices that can be visited starting from each index.
- Initially, all entries in `f` are set to 1, but as we explore jumps from each index, these entries may increase.
- We sort a list of pairs, where each pair consists of the value at each index `arr[i]` and the index itself `i`. This ensures we always work from lower to higher values, respecting the jump rule that you can only jump to a smaller value.
- We iterate over the sorted list, and for each index `i`, we examine the possible indices we can jump to within the range `d` in both directions (to the left and to the right).
- For each possible jump from `i` to `j`, we check two conditions: whether the jump is within the distance `d`, and if `arr[j]` is less than `arr[i]`. If both conditions are met, it means a jump from `i` to `j` is valid.
- Upon finding a valid jump, we update `f[i]` to be the maximum of its current value and `f[j] + 1`, reflecting the number of indices visited from `j` plus the jump just made from `i` to `j`.
- This updating of `f[i]` happens for all `j` within the jumpable range from `i` that match the criteria of lower value and within the `d` distance. This ensures that `f[i]` represents the optimal (maximum) number of indices visited starting from `i`.
- Since `f` has been filled from the lowest value of `arr[i]` upwards, by the time we reach higher values, we've already calculated the optimal number of indices that can be visited from all lower values; this property of the solution eliminates unnecessary recalculations and ensures the solution is efficient.
- After we finish updating `f[i]` for all indices, the last step is to return `max(f)`, the maximum value within `f`, which represents the maximum number of indices we can visit starting from the best possible index.

This process structurally follows a bottom-up dynamic programming approach, going from smaller sub-problems to larger sub-problems and using previous results to construct an optimal final answer.

Example Walkthrough

Let's assume we have an array `arr = [6, 4, 14, 6, 8]` and `d = 2`.

Following the solution approach:

1. We create our table `f` with the same length as `arr` and initialize all values to 1, because each index can visit at least itself.

   `f = [1, 1, 1, 1, 1]`

2. We create a list of pairs to sort by value in `arr`: `[(6, 0), (4, 1), (14, 2), (6, 3), (8, 4)]`.
3. After sorting by the first element in the pairs: `[(4, 1), (6, 0), (6, 3), (8, 4), (14, 2)]`.
4. We proceed to iterate through our sorted list and consider jumps.
   - Starting with the pair `(4, 1)`. There are no indices with smaller values than 4 within the range `d`, so we skip to the next pair.
   - For the pair `(6, 0)`, we can jump to index 1 which has the value 4 (and is within `d` distance). We update `f[0]` to be the maximum of `f[0]` and `f[1] + 1`, so:

     `f = [2, 1, 1, 1, 1]`

   - Considering pair `(6, 3)`, we can jump to index 1 and index 4. We update `f[3]` to be the maximum of `f[3]` and `f[1] + 1`, and `f[3]` and `f[4] + 1`:

     `f = [2, 1, 1, 2, 1]`

   - With pair `(8, 4)`, we examine the indices within range `d` (indices 2 and 3), but we can only jump to index 3, updating `f[4]`:

     `f = [2, 1, 1, 2, 3]`

   - Lastly, the pair `(14, 2)`. This value lets us jump to any lower index within `d` (0, 1, 3, and 4). We update `f[2]` for each of these:

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

5. Now we have our completed `f` table indicating the maximum number of indices that can be visited starting from each index. To get the answer, we just take the maximum value in `f`:

   `max(f) = 4`

So the maximum number of indices we can visit starting from the best index is 4, which starts from index 2 in the example array.

Solution Implementation

Python

```python
class Solution:
    def maxJumps(self, arr: List[int], max_distance: int) -> int:
        # Determine the total number of elements in the array
        num_elements = len(arr)

        # Initial jump counts set to 1 for each position, as each position can jump at least once
        jumps = [1] * num_elements

        # Process positions in ascending order of their heights, along with their indices
        for height, position in sorted(zip(arr, range(num_elements))):
            # Check for jumps to the left
            for left in range(position - 1, -1, -1):
                # Stop if out of jump distance or if an equal or higher bar is met
                if position - left > max_distance or arr[left] >= height:
                    break
                # Update the jump count if a new max is found
                jumps[position] = max(jumps[position], 1 + jumps[left])

            # Check for jumps to the right
            for right in range(position + 1, num_elements):
                # Stop if out of jump distance or if an equal or higher bar is met
                if right - position > max_distance or arr[right] >= height:
                    break
                # Update the jump count if a new max is found
                jumps[position] = max(jumps[position], 1 + jumps[right])

        # Return the maximum jump count found across all positions
        return max(jumps)
```

Java

```java
class Solution {
    public int maxJumps(int[] heights, int maxDistance) {
        int n = heights.length;

        // Create an array of indices from the input array
        Integer[] indices = new Integer[n];
        for (int i = 0; i < n; ++i) {
            indices[i] = i;
        }

        // Sort the indices based on the heights, if the same height maintain the order of indices
        Arrays.sort(indices, (i, j) -> Integer.compare(heights[i], heights[j]));

        int[] dp = new int[n]; // Dynamic programming array to store the max number of jumps
        Arrays.fill(dp, 1); // Initialize with 1 since the min jumps for each position is 1 (stand still)

        int maxJumps = 0; // Variable to keep track of the maximum number of jumps

        // Calculate the maximum number of jumps you can do from each position
        for (int currentIndex : indices) {
            // Look left of the current index
            for (int leftIndex = currentIndex - 1; leftIndex >= 0; --leftIndex) {
                // If it's too far or the height at leftIndex is higher or equal, we can't jump from there
                if (currentIndex - leftIndex > maxDistance || heights[leftIndex] >= heights[currentIndex]) {
                    break;
                }
                // Update the dp value if we find a better path
                dp[currentIndex] = Math.max(dp[currentIndex], 1 + dp[leftIndex]);
            }

            // Look right of the current index
            for (int rightIndex = currentIndex + 1; rightIndex < n; ++rightIndex) {
                // If it's too far or the height at rightIndex is higher or equal, we can't jump from there
                if (rightIndex - currentIndex > maxDistance || heights[rightIndex] >= heights[currentIndex]) {
                    break;
                }
                // Update the dp value if we find a better path
                dp[currentIndex] = Math.max(dp[currentIndex], 1 + dp[rightIndex]);
            }

            // Update the result with the maximum dp value found so far
            maxJumps = Math.max(maxJumps, dp[currentIndex]);
        }

        return maxJumps; // Return the maximum number of jumps possible
    }
}
```

C++

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

class Solution {
public:
    // Function to find the maximum number of jumps you can make in the array
    int maxJumps(vector<int>& heights, int maxDistance) {
        int n = heights.size(); // Size of the array

        // Initialize an index vector with values from 0 to n-1
        vector<int> indices(n);
        iota(indices.begin(), indices.end(), 0);

        // Sort the indices based on the corresponding values in 'heights'
        sort(indices.begin(), indices.end(), [&](int a, int b) { return heights[a] < heights[b]; });

        // Initialize a vector to store the furthest jump length from every position
        vector<int> dp(n, 1); // All elements are initialized to 1 as the minimum jump is always 1 (itself)

        // Loop through the sorted indices to fill dp in increasing order of heights
        for (int index : indices) {
            // Look to the left of the current index
            for (int left = index - 1; left >= 0; --left) {
                // If the left element is out of bounds or taller, stop checking further
                if (index - left > maxDistance || heights[left] >= heights[index]) {
                    break;
                }
                // Update dp array for index with the optimal previous jump count
                dp[index] = max(dp[index], 1 + dp[left]);
            }
            // Look to the right of the current index
            for (int right = index + 1; right < n; ++right) {
                // If the right element is out of bounds or taller, stop checking further
                if (right - index > maxDistance || heights[right] >= heights[index]) {
                    break;
                }
                // Update dp array for index with the optimal previous jump count
                dp[index] = max(dp[index], 1 + dp[right]);
            }
        }
        // Return the maximum jumps that can be made, found in the dp array
        return *max_element(dp.begin(), dp.end());
    }
};
```

TypeScript

```typescript
// Importing necessary functionalities from arrays and algorithms
import { max } from 'lodash';

// Function to find the maximum number of jumps you can make in the array
function maxJumps(heights: number[], maxDistance: number): number {
    const n: number = heights.length; // Size of the array

    // Initialize an index array with values from 0 to n-1
    const indices: number[] = Array.from({ length: n }, (_, i) => i);

    // Sort the indices based on the corresponding values in 'heights'
    indices.sort((a, b) => heights[a] - heights[b]);

    // Initialize an array to store the furthest jump length from every position
    const dp: number[] = new Array(n).fill(1); // All elements are initialized to 1 as the minimum jump is always 1 (itself)

    // Loop through the sorted indices to fill dp in increasing order of heights
    indices.forEach((index) => {
        // Look to the left of the current index
        for (let left = index - 1; left >= 0; --left) {
            // If the left element is out of maxDistance or taller, stop checking further
            if (index - left > maxDistance || heights[left] >= heights[index]) {
                break;
            }
            // Update dp array for index with the optimal previous jump count
            dp[index] = Math.max(dp[index], 1 + dp[left]);
        }
        // Look to the right of the current index
        for (let right = index + 1; right < n; ++right) {
            // If the right element is out of maxDistance or taller, stop checking further
            if (right - index > maxDistance || heights[right] >= heights[index]) {
                break;
            }
            // Update dp array for index with the optimal previous jump count
            dp[index] = Math.max(dp[index], 1 + dp[right]);
        }
    });

    // Return the maximum jumps that can be made, found in the dp array
    return Math.max(...dp);
}

// Example usage:
// If you want to call the function, you would pass in the heights array and maxDistance like so:
// maxJumps([4, 2, 7, 6, 9, 14, 12], 2);
```

```python
class Solution:
    def maxJumps(self, arr: List[int], max_distance: int) -> int:
        # Determine the total number of elements in the array
        num_elements = len(arr)

        # Initial jump counts set to 1 for each position, as each position can jump at least once
        jumps = [1] * num_elements

        # Process positions in ascending order of their heights, along with their indices
        for height, position in sorted(zip(arr, range(num_elements))):
            # Check for jumps to the left
            for left in range(position - 1, -1, -1):
                # Stop if out of jump distance or if an equal or higher bar is met
                if position - left > max_distance or arr[left] >= height:
                    break
                # Update the jump count if a new max is found
                jumps[position] = max(jumps[position], 1 + jumps[left])

            # Check for jumps to the right
            for right in range(position + 1, num_elements):
                # Stop if out of jump distance or if an equal or higher bar is met
                if right - position > max_distance or arr[right] >= height:
                    break
                # Update the jump count if a new max is found
                jumps[position] = max(jumps[position], 1 + jumps[right])

        # Return the maximum jump count found across all positions
        return max(jumps)
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed based on the following:

1. Sorting the array `arr` alongside the indices has a complexity of `O(n log n)` where `n` is the length of the array `arr`.
2. The double nested loops contribute to the time complexity in the following way:
   - The outer loop runs `n` times since it iterates over the sorted arrays of heights and indices.
   - Each inner loop performs a linear scan in both directions, but due to the constraint limiting the jumps to distance `d`, the inner loops each run at most `d` times.
   - Therefore, the inner loops collectively contribute at most `O(n * 2d) = O(nd)` time complexity.

Combining these, the total time complexity of the code is `O(n log n + nd)`.

Space Complexity

The space complexity of the given code can be analyzed based on the following:

1. An auxiliary array `f` of size `n` is used to store the maximum number of jumps from each position, contributing an `O(n)` space complexity.
2. Since there are no recursive calls or additional data structures that grow with the input size, and the sorting operation can be assumed to use `O(1)` space given that most sorting algorithms can be done in-place (like Timsort in Python), the additional space complexity imposed by the stack frames during the for loops and sorting is constant.

Therefore, the total space complexity of the code is `O(n)`.