403. Frog Jump

Hard · Array · Dynamic Programming

Problem Description

In this problem, we are tasked with determining whether a frog is able to cross a river by hopping from stone to stone. The river is made up of several units, and at each unit, there may be a stone present. The frog can only land on stones and is not allowed to land in the water.

The stones are depicted as a list of integer positions, in sorted ascending order, representing their place in the river. The challenge is to figure out if the frog can make it to the last stone, starting from the first stone. The frog's first jump is always 1 unit. If the frog's last jump covered `k` units, the next jump can either be `k - 1`, `k`, or `k + 1` units. It's important to note that the frog can only move forward, not backward.

The main objective is to navigate through the given list of stones and find a sequence of jumps that will lead the frog to the final stone without falling into the water at any point.

Intuition

To solve the problem, we have to account for multiple possibilities of jump sequences that the frog can take, which suggests a need for exploring various jump lengths dynamically as the frog makes its way from the first to the last stone. This kind of problem is well-suited for depth-first search (DFS) or dynamic programming (DP) approaches.

DFS Approach: With the DFS solution, we explore all the possible jump lengths from each stone. We start from the first stone and recursively try every possible next jump length (k-1, k, k+1). If at any point we can jump to a stone that allows us to eventually reach the last stone, we return true. We employ memoization to store the results of subproblems (i.e., if a certain jump from a particular stone is possible or not), which prevents us from recalculating the same scenarios multiple times, thus optimizing efficiency.

Dynamic Programming Approach: The DP solution sets up a table where each entry `f[i][k]` represents whether it's possible to reach stone `i` with a last jump of `k` units. We initialize the first stone's entry as true because the frog starts there. Then we systematically fill out the table: for each stone and each potential jump length, we check if we could have gotten there from an earlier stone with a valid jump length. The tabulated results are reused to infer about the subsequent stones.

Both approaches fundamentally examine which stones are reachable from any given stone and under what conditions, ultimately producing a result indicating whether the last stone can be reached or not.

Solution Approach

The solution we are looking at employs two different strategies: Hash Table with Memoization and Dynamic Programming (DP). Each has its own set of data structures and patterns.

Hash Table + Memoization:

This approach utilizes a recursive Depth-First Search (DFS) method enhanced by a memorization technique to efficiently solve the problem.

- **Algorithm:**
  - We use a hash table, denoted as `pos`, to map each stone's value to its index, which allows us to quickly check if a stone exists at a certain position.
  - We implement a recursive function `dfs(i, k)` that returns true if the frog is capable of reaching the end of the stones from the current position `i` with the last jump of `k` units.
  - In the function `dfs(i, k)`, if the current index `i` is equal to the last index `n-1` of stones, we return true indicating that the frog has successfully reached the end.
  - Otherwise, we explore all feasible jumps from the current stone - the frog can jump `k-1`, `k`, or `k+1` units forward.
  - We check each of these jumps to see if a stone exists at the new position - if so, we make a recursive call to `dfs` with the new position index and the jump distance.
  - The answer from subsequent `dfs` calls is stored using memoization to avoid re-computation, thereby optimizing the solution.
- **Data Structures:**
  - A hash table (dictionary in Python) is used to quickly determine if a stone exists at a particular jump distance.
  - A memoization cache, which the `@cache` decorator in Python provides, is used to store the outcomes of the recursive `dfs` calls.
- **Patterns:**
  - Memoization pattern is applied here to store already computed results of certain subproblems, avoiding redundant computation and reducing time complexity.

Dynamic Programming:

The Dynamic Programming approach constructs a DP table where each cell `f[i][k]` indicates the possibility of reaching the `i-th` stone using a jump of `k` units.

- **Algorithm:**
  - Initialize a 2D table `f` for the DP, where most of the elements are false except `f[0][0]` which should be true, as the frog starts at the first stone with a jump of 0 units.
  - Iterate through all stones and possible jump lengths to populate this table.
  - For each stone `i` and each jump length `k`, we check if there is a way to reach `i` from a previous stone with the previous jump lengths `k-1`, `k`, or `k+1`. If such a way exists, `f[i][k]` is set to true.
  - Finally, check whether any jump length at the last stone entry in the DP table is true, which indicates that the river can be crossed.
- **Data Structures:**
  - A 2D array or list of lists in Python is used to keep track of the dynamic programming state.
- **Patterns:**
  - Dynamic Programming pattern is employed to break down the problem into smaller overlapping subproblems and build up the solution incrementally.

The time complexity for both approaches is `O(n^2)`, and the space complexity is also `O(n^2)`, where `n` is the number of stones. For larger values of `n`, these approaches might be computationally intense due to the square nature of their complexity.

Example Walkthrough

Let's consider an example where the stones are placed at the following positions in the river: `[0, 1, 3, 5, 6, 8, 12, 17]`.

Remember, the frog can start at the first stone and the first jump is always 1 unit.

Using the Hash Table + Memoization approach:

1. We create a hash table as described in the algorithm to map stone positions to their index: `pos = {0:0, 1:1, 3:2, 5:3, 6:4, 8:5, 12:6, 17:7}`.
2. We then implement the `dfs(i, k)` function, starting at the first stone with index `i = 0` and initial jump length `k = 0` (this is because the first jump will always be 1, so the last jump length `k` that brought us to the start is 0).
3. Checking at `i = 0`, `dfs(0, 0)` is called, and we look for stones that can be reached with a jump of `1` (k + 1 since `k = 0` initially).
4. Our next stone is at position 1, and there's indeed a stone there, so we call `dfs(1, 1)`.
5. At `i = 1`, with `k = 1`, we have possible jump lengths of `0` (k-1), `1` (k), and `2` (k+1). Only a jump of 2 leads to the next stone at position index `3`.
6. We repeat the process, calling `dfs(3, 2)`. The recursion continues to explore jumps, storing results in the memoization cache to prevent recalculating scenarios for each stone position and jump length combination.
7. The process continues until we reach the last stone or determine that the frog cannot reach the end. In this case, by following the jumps from position indices `0 -> 1 -> 3 -> 4 -> 5 -> 6 -> 7`, the frog can successfully make it across to the last stone.

Using the Dynamic Programming approach:

1. We set up a 2D DP table `f` with dimensions `n x n` initialized with `False` values, except for `f[0][0]` which is set to `True`.
2. We iterate over each stone position `i` and consider all possible `k` values that could reach this stone, checking whether a valid jump from the previous stone can be made with a range of `k-1`, `k`, or `k+1`.
3. As we fill the table, we find that the frog can reach the stone at index `1` with a jump of `1`. Moving on, we find the frog can reach the stone at index `3` with a jump of `2`, and the process continues in that manner.
4. We keep building the table based on previous jumps that can reach the current stone until we've considered all stones.
5. At the end, we check `f[n-1][k]` for any `k` to see if the frog can reach the last stone with `k` jumps. For the last stone at index `7`, if any `f[7][k]` is `True` for any `k`, the frog can cross the river. In our example, the frog can indeed cross the river as there will be `True` entries indicating valid jumps to the last stone.

In this small example, both approaches demonstrate the frog's ability to cross the river by carefully examining each possible jump and either using a hash table with memoization to store results of recursive DFS calls or using a DP table to iteratively track the stone reachable states with diverse jump lengths.

Python Solution

```python
# Import the required lru_cache decorator from functools to cache expensive recursive calls
from functools import lru_cache

class Solution:
    def canCross(self, stones):
        # A helper function is defined for Depth-First Search (DFS) to determine if the frog can cross
        # using dynamic programming to avoid recalculations.
        # This function is cached to store the results of subproblems.
        @lru_cache(maxsize=None)
        def dfs(index, jump_size):
            # If the current index is the last stone, then the frog can cross
            if index == last_stone_index:
                return True
            # Loop through the potential next jump sizes: jump_size-1, jump_size, jump_size+1
            for next_jump in range(jump_size - 1, jump_size + 2):
                # Check if the next jump is possible (jump size must be greater than 0)
                # and the stone to jump on exists in the stone positions dictionary.
                # If a valid stone is found, recurse from the position of that stone with the size of the next jump.
                if next_jump > 0 and index + next_jump in stone_positions:
                    if dfs(stone_positions[index + next_jump], next_jump):
                        return True
            # If none of the possible jumps lead to a crossing, return False.
            return False

        # The length of the stones list is stored for easy access.
        last_stone_index = len(stones) - 1
        # A dictionary (hash map) to store the position of each stone for O(1) look-up.
        stone_positions = {stone: idx for idx, stone in enumerate(stones)}
        # Kickstart the recursive DFS with the first stone and initial jump size 0.
        return dfs(0, 0)

# Example usage:
# sol = Solution()
# can_cross = sol.canCross([0,1,3,5,6,8,12,17])
# print(can_cross)  # Outputs: True or False based on whether the frog can cross or not
```

Java Solution

```java
class Solution {
    private Boolean[][] memo; // memoization array to store results of subproblems
    private Map<Integer, Integer> positionMap = new HashMap<>(); // maps stone's value to its index
    private int[] stones; // array of stones
    private int numOfStones; // number of stones

    // Determines if you can cross the river by jumping on stones
    public boolean canCross(int[] stones) {
        numOfStones = stones.length;
        memo = new Boolean[numOfStones][numOfStones];
        this.stones = stones;
        // populate the position map
        for (int i = 0; i < numOfStones; ++i) {
            positionMap.put(stones[i], i);
        }
        // start DFS from stone 0 with jump size 0
        return dfs(0, 0);
    }

    // Uses DFS to see if a crossing is possible from the current stone with a given jump size
    private boolean dfs(int index, int lastJumpSize) {
        // if we are at the last stone, crossing is successful
        if (index == numOfStones - 1) {
            return true;
        }
        // if we have visited this state, return the stored result
        if (memo[index][lastJumpSize] != null) {
            return memo[index][lastJumpSize];
        }
        // try to jump to the next stone with jump sizes: k-1, k, or k+1
        for (int currentJumpSize = lastJumpSize - 1; currentJumpSize <= lastJumpSize + 1; ++currentJumpSize) {
            if (currentJumpSize > 0) { // cannot jump back or stay
                int nextPosition = stones[index] + currentJumpSize; // next stone's position
                // check if there is a stone at the next position
                if (positionMap.containsKey(nextPosition) && dfs(positionMap.get(nextPosition), currentJumpSize)) {
                    // if we can cross starting from the next stone, mark the current state as true
                    return memo[index][lastJumpSize] = true;
                }
            }
        }
        // if none of the jumps work, mark the current state as false
        return memo[index][lastJumpSize] = false;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <functional>
#include <cstring>

class Solution {
public:
    // Determines if it is possible to cross the river by jumping on the given stones
    bool canCross(vector<int>& stones) {
        int n = stones.size();
        // f is the memoization table where f[i][j] represents the possibility to reach stone i with the last jump of j units
        vector<vector<int>> memo(n, vector<int>(n, -1));
        // Maps stone positions to their indices for quick lookup
        unordered_map<int, int> positionIndexMap;
        for (int i = 0; i < n; ++i) {
            positionIndexMap[stones[i]] = i;
        }

        // Depth-first search function that uses memoization
        function<bool(int, int)> dfs = [&](int currentIndex, int lastJump) -> bool {
            // Base case: if the current index is the last one in stones array, the frog can cross
            if (currentIndex == n - 1) {
                return true;
            }
            // If the current state has already been computed, return the stored value
            if (memo[currentIndex][lastJump] != -1) {
                return memo[currentIndex][lastJump];
            }
            // Attempt to jump to the next stone with the last jump minus one, the same, or plus one
            for (int nextJump = lastJump - 1; nextJump <= lastJump + 1; ++nextJump) {
                if (nextJump > 0 && positionIndexMap.count(stones[currentIndex] + nextJump)) {
                    if (dfs(positionIndexMap[stones[currentIndex] + nextJump], nextJump)) {
                        // If a valid jump is found, store the result and return true
                        return memo[currentIndex][lastJump] = true;
                    }
                }
            }
            // If no valid jump can be found, store the result and return false
            return memo[currentIndex][lastJump] = false;
        };

        // Start DFS from the first stone with an initial jump size of 0
        return dfs(0, 0);
    }
};
```

Typescript Solution

```typescript
function canCross(stones: number[]): boolean {
    const totalStones = stones.length;
    const positionToIndex: Map<number, number> = new Map();

    // Map each stone's position to its index for quick access
    for (let i = 0; i < totalStones; ++i) {
        positionToIndex.set(stones[i], i);
    }

    // Memoization table, -1 indicates state has not been computed yet
    const memo: number[][] = new Array(totalStones).fill(0).map(() => new Array(totalStones).fill(-1));

    // Helper DFS function to determine if the frog can reach the end
    const dfs = (index: number, lastJump: number): boolean => {
        // Reached the last stone
        if (index === totalStones - 1) {
            return true;
        }

        // Return cached result if this state has been computed
        if (memo[index][lastJump] !== -1) {
            return memo[index][lastJump] === 1;
        }

        // Iterate through all possible next jumps (k - 1, k, k + 1)
        for (let jumpDistance = lastJump - 1; jumpDistance <= lastJump + 1; ++jumpDistance) {
            if (jumpDistance > 0 && positionToIndex.has(stones[index] + jumpDistance)) {
                const nextIndex = positionToIndex.get(stones[index] + jumpDistance)!;
                if (dfs(nextIndex, jumpDistance)) {
                    memo[index][lastJump] = 1;
                    return true;
                }
            }
        }

        // Cache the result (0 for false)
        memo[index][lastJump] = 0;
        return false;
    };

    // Start DFS from the first stone with 0 as the last jump distance
    return dfs(0, 0);
}
```

Time and Space Complexity

The time complexity of the given code is `O(n^2)`. This is because, in the worst case, the `dfs` function is called for every stone, and at each stone, it attempts three different jump distances (k - 1, k, k + 1). The `@cache` decorator ensures that each combination of `i` (stone index) and `k` (last jump distance) is calculated only once. There are at most `n` different `i` values and potentially `n` different `k` values (since the furthest a frog can jump from stone `i` is to stone `i + k`), which results in a total of `n * n` unique states.

The space complexity is also `O(n^2)` because of the caching. For each unique state (`i`, `k`), the result of the computation is stored to prevent re-computation. Considering that there can be up to `n` unique values for `i` and `n` unique values for `k`, the space required for caching these results can take up to `n * n` space in the worst-case scenario.