MCS – 253P ADVANCED PROGRAMMING AND PROBLEM SOLVING

HOMEWORK –8 (Frog Jump)

Aswin Sampath
saswin@uci.edu

(53844684)

Description | Editorial | Solutions | Submissions | Accepted ×

403. Frog Jump

Solved

Hard | Topics | Companies

A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of `stones` positions (in units) in sorted **ascending order**, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be `1` unit.

If the frog's last jump was `k` units, its next jump must be either `k - 1`, `k`, or `k + 1` units. The frog can only jump in the forward direction.

Example 1:

```
Input: stones = [0,1,3,5,6,8,12,17]
Output: true
Explanation: The frog can jump to the last stone by jumping 1 unit to the 2nd stone, then 2 units to the 3rd stone, then 2 units to the 4th
stone, then 3 units to the 6th stone, 4 units to the 7th stone, and 5 units to the 8th stone.
```

Example 2:

```
Input: stones = [0,1,2,3,4,8,9,11]
Output: false
Explanation: There is no way to jump to the last stone as the gap between the 5th and 6th stone is too large.
```

Constraints:

- `2 <= stones.length <= 2000`

Understanding the Problem

The problem involves a frog trying to cross a river by jumping on stones placed at various positions in ascending order across the river. The frog starts at the first stone (position 0) and aims to reach the last stone. The frog's jumps must follow specific rules:

If the frog's last jump was k units, its next jump must be either k - 1, k, or k + 1 units.

The frog can only jump in the forward direction.

The task is to determine if the frog can reach the last stone following the given rules.

Identifying Edge Cases

- An empty input: When the input list of stones is empty, the frog cannot move, so it cannot reach the last stone.
- A single stone: When there's only one stone, if it's not positioned at 0, the frog cannot reach it. If it's positioned at 0, the frog is already there.

Effective Test Cases

A list of stones with a clear path to the last stone:

Input: stones = [0, 1, 3, 5, 6, 8, 12, 17]

Expected Output: true

A list of stones with no possible path to the last stone due to a large gap:

Input: stones = [0, 1, 2, 3, 4, 8, 9, 11]

Expected Output: false

An array with only two stones:

Input: stones = [0, 1]

Expected Output: true

Algorithmic Solution

The provided C++ code solves the problem using dynamic programming and backtracking:

- It creates a map (mp) to store the positions of stones.
- Initializes a 2D DP array (dp) to store results of recursive calls to avoid redundant calculations.
- Implements a recursive function solve to check if the frog can reach the last stone from a specific stone position num by considering all possible valid jumps.
- The solve function performs backtracking, checking if jumps of k-1, k, or k+1 units from the current stone position can lead to the last stone.

Time and Space Complexity Analysis

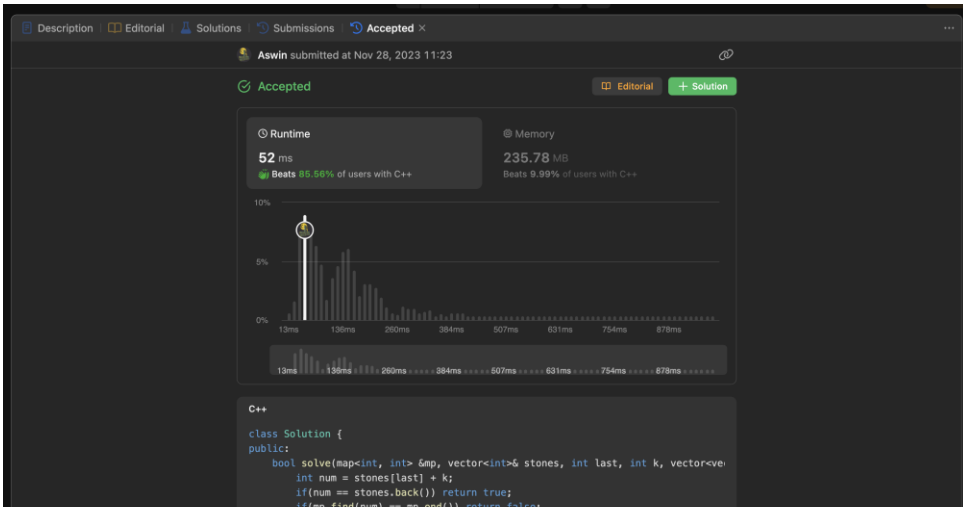
The time complexity for this algorithm is O(n^3), where n is the number of stones. This arises from the three nested loops in the backtracking function, each running up to n times.

The space complexity is O(n^2) due to the usage of the DP array, where n is the number of stones.

Code:

```cpp
class Solution {
public:
    bool solve(map<int, int> &mp, vector<int>& stones, int last, int k, vector<vector<int>> &dp){
        int num = stones[last] + k;
        if(num == stones.back()) return true;
        if(mp.find(num) == mp.end()) return false;

        if(dp[last][k] != -1) return dp[last][k];

        int idx = mp[num];
        bool ans;

        ans = solve(mp, stones, idx, k+1, dp);
        ans = ans || solve(mp, stones, idx, k, dp);
        if(k > 1){
            ans = ans || solve(mp, stones, idx, k-1, dp);
        }

        return dp[last][k] = ans;
    }

    bool canCross(vector<int>& stones) {
        map<int, int> mp;
        for(int i=0; i<stones.size(); i++){
            mp[stones[i]] = i;
        }

        vector<vector<int>> dp(stones.size(), vector<int>(stones.size(), -1));
        return solve(mp, stones, 0, 1, dp);
    }
};
```

<u>Output:</u>


Description
Editorial
Solutions
Submissions
Accepted
Aswin submitted at Nov 28, 2023 11:23
Accepted
Editorial
Solution
Runtime
52 ms
Beats 85.56% of users with C++
Memory
235.78 MB
Beats 9.99% of users with C++
10%
5%
0%
13ms
136ms
260ms
384ms
507ms
631ms
754ms
878ms
C++
class Solution {
public:
bool solve(map<int, int> &mp, vector<int>& stones, int last, int k, vector<ve
int num = stones[last] + k;
if(num == stones.back()) return true;