# LeetCode 403

https://leetcode.com/problems/frog-jump/description/

Yifeng Zeng

# Description

403. Frog Jump

# Idea Report

The first stone's position is always 0, so the first jump is always 1. We can have a map to save the next jump distance that can be made for the current stone. Map key is the current stone location, map value is a set of jumps that can be make based on the current stone. So on current stone, if we jump to the last stone, we can directly return true. If not, we will have nextStone = currentLocation + jump. We add the jump-1, jump, and jump+1 to the set of the map.get(nextStone). If we never jumps to the last element of the stones, we can't jump to the end and return false.

Code

```
class Solution {
    public boolean canCross(int[] stones) {
        HashMap<Integer, HashSet<Integer>> map = new HashMap<>();
        for (int i = 0; i < stones.length; i++) {
            map.put(stones[i], new HashSet<Integer>());
        }
        map.get(0).add(1);

        for (int i = 0; i < stones.length; i++) {
            int stone = stones[i];
            Set<Integer> nextJumps = map.get(stone);
            for (int jump : nextJumps) {
                int goesTo = stone + jump;
                if (goesTo == stones[stones.length - 1]) {
                    return true;
                }
                if (map.containsKey(goesTo)) {
                    if (jump - 1 > 0) {
                        map.get(goesTo).add(jump - 1);
                    }
                    map.get(goesTo).add(jump);
                    map.get(goesTo).add(jump + 1);
                }
            }
        }
        return false;
    }
}
```

Code

```
class Solution {
    // AC
    public boolean canCross(int[] stones) {
        for (int i = 0; i < stones.length - 1; i++) {
            if (stones[i] * 2 + 1 < stones[i+1]) {
                return false;
            }
        }
        Set<Integer> set =
            Arrays.stream(stones).boxed().collect(Collectors.toSet());
        return helper(stones, set, 0, 0);
    }

    private boolean helper(int[] stones, Set<Integer> set,
                           int curStone, int lastStep) {
        if (curStone == stones[stones.length - 1]) {
            return true;
        }
        if (!set.contains(curStone)) {
            return false;
        }

        boolean canJump = false;
        for (int nextStep = Math.min(stones[stones.length - 1], lastStep + 1);
             nextStep >= Math.max(1, lastStep - 1); nextStep--) {
            if (set.contains(curStone + nextStep)) {
                canJump = canJump ||
                    helper(stones, set, curStone + nextStep, nextStep);
            }
        }
        return canJump;
    }
}
```

# Summary

- Anaylize next situation based on current situation.