Frog Jump

A frog is crossing a river. The river is divided into x 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 is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

If the frog has just jumped k units, then its next jump must be either k - 1, k, or k + 1 units. Note that the frog can only jump in the forward direction.

Note:

- The number of stones is ≥ 2 and is
- Each stone's position will be a non-negative integer 31.
- The first stone's position is always 0.

Example 1:

```
[0,1,3,5,6,8,12,17]

There are a total of 8 stones.
The first stone at the 0th unit, second stone at the 1st unit,
third stone at the 3rd unit, and so on...
The last stone at the 17th unit.

Return true. 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:

```
[0,1,2,3,4,8,9,11]

Return false. There is no way to jump to the last stone as
the gap between the 5th and 6th stone is too large.
```

Solution 1

Search for the last stone in a depth-first way, prune those exceeding the [k-1,k+1] range. Well, I think the code is simple enough and need no more explanation.

```
bool canCross(vector<int>& stones, int pos = 0, int k = 0) {
    for (int i = pos + 1; i < stones.size(); i++) {
        int gap = stones[i] - stones[pos];
        if (gap < k - 1) continue;
        if (gap > k + 1) return false;
        if (canCross(stones, i, gap)) return true;
    }
    return pos == stones.size() - 1;
}
```

This can pass OJ at 9ms but is inefficient for extreme cases. (update: new test cases are added and the solution above no longer passes OJ, please see the solution below which takes 62ms) We can memorize the returns with minimum effort:

```
unordered_map<int, bool> dp;

bool canCross(vector<int>& stones, int pos = 0, int k = 0) {
    int key = pos | k << 11;

    if (dp.count(key) > 0)
        return dp[key];

    for (int i = pos + 1; i < stones.size(); i++) {
        int gap = stones[i] - stones[pos];
        if (gap < k - 1)
            continue;
        if (gap > k + 1)
            return dp[key] = false;
        if (canCross(stones, i, gap))
            return dp[key] = true;
    }

    return dp[key] = (pos == stones.size() - 1);
}
```

The number of stones is less than 1100 so**pos** will always be less than 2^11 (2048). Stone positions could be theoretically up to 2^31 but **k** is practically not possible to be that big for the parameter as the steps must start from 0 and 1 and at the 1100th step the greatest valid k would be 1100. So combining **pos** and **k** is safe here.

written by mzchen original link here

Solution 2

For each stone, we write down a set of jump distances taken from previous stones to reach this stone (for example, if the stones were 3, 5, 7, then for stone 7 we write down 7 - 3 = 4 and 7 - 5 = 2, assuming they're both valid moves). From the distance set we can find a set of "imaginary" stones reachable from this stone, so all we have to do is to figure out (through hash table) which of those target stones actually exists, and propagate the distance to their distance sets.

Finally, we check if the last stone was reachable by checking if its distance set wasn't empty.

```java
public class Solution {
    public boolean canCross(int[] stones) {
        final int l = stones != null ? stones.length : 0;
        if (l < 1 || stones[0] != 0) return false;
        final Map<Integer, Set<Integer>> map = new HashMap<>();
        for (int s : stones) map.put(s, new HashSet<Integer>());
        for (int s : stones) {
            Set<Integer> jSet = map.get(s);
            // Initial condition
            if (s == 0) {
                jSet.add(0);
                if (map.containsKey(1)) map.get(1).add(1);
                continue;
            }
            // For other stones
            for (int j : jSet) {
                int jj = j - 1;
                int ss = s + jj;
                // Previous jump - 1
                if (ss != s && map.containsKey(ss)) map.get(ss).add(jj);
                // Previous jump
                jj++; ss++;
                if (ss != s && map.containsKey(ss)) map.get(ss).add(jj);
                // Previous jump + 1
                jj++; ss++;
                if (ss != s && map.containsKey(ss)) map.get(ss).add(jj);
            }
        }
        return !map.get(stones[l - 1]).isEmpty();
    }
}
```

written by Rabby250 original link here

Solution 3

```java
public boolean canCross(int[] stones) {
        if(stones[1] > 1) return false;
        if(stones.length == 2) return true;
        return helper(stones, 1, 1);
    }
    private boolean helper(int[] arr, int i, int step){
        boolean pass = false;
        if(i == arr.length-1) return true;
        for(int j = i+1; j < arr.length; j++){
            if(arr[j] <= arr[i] + step + 1 && arr[j] >= arr[i]+step-1){
                pass = pass || helper(arr, j, arr[j] - arr[i]);
            }
        }
        return pass;
    }
```

written by 类与对象Re original link here