403. Frog Jump

Jan. 6, 2017 | 31.6K views

Average Rating: 4.14 (21 votes)

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's last jump was *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 < 1,100.
- Each stone's position will be a non-negative integer < 2^{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.
```

Summary

Given a sorted stone array containing the positions at which there are stones in a river. We need to determine whether it is possible or not for a frog to cross the river by stepping over these stones, provided that the frog starts at position 0, and at every step the frog can make a jump of size $k-1$, k or $k+1$ if the previous jump is of size k.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

In the brute force approach, we make use of a recursive function $canCross$ which takes the given stone array, the current position and the current $jumpsize$ as input arguments. We start with $currentPosition = 0$ and $jumpsize = 0$. Then for every function call, we start from the $currentPosition$ and check if there lies a stone at $(currentPostion + newjumpsize)$, where, the $newjumpsize$ could be $jumpsize$, $jumpsize + 1$ or $jumpsize - 1$. In order to check whether a stone exists at the specified positions, we check the elements of the array in a linear manner. If a stone exists at any of these positions, we call the recursive function again with the same stone array, the $currentPosition$ and the $newjumpsize$ as the parameters. If we are able to reach the end of the stone array through any of these calls, we return $true$ to indicate the possibility of reaching the end.

Java Copy

```java
public class Solution {
    public boolean canCross(int[] stones) {
        return can_Cross(stones, 0, 0);
    }
    public boolean can_Cross(int[] stones, int ind, int jumpsize) {
        for (int i = ind + 1; i < stones.length; i++) {
            int gap = stones[i] - stones[ind];
            if (gap >= jumpsize - 1 && gap <= jumpsize + 1) {
                if (can_Cross(stones, i, gap)) {
                    return true;
                }
            }
        }
        return ind == stones.length - 1;
    }
}
```

Complexity Analysis

- Time complexity : $O(3^n)$. Recursion tree can grow upto 3^n.
- Space complexity : $O(n)$. Recursion of depth n is used.

Approach #2 Better Brute Force[Time Limit Exceeded]

Algorithm

In the previous brute force approach, we need to find if a stone exists at $(currentPosition + newjumpsize)$, where $newjumpsize$ could be either of $jumpsize - 1$, $jumpsize$ or $jumpsize + 1$. But in order to check if a stone exists at the specified location, we searched the given array in linearly. To optimize this, we can use binary search to look for the element in the given array since it is sorted. Rest of the method remains the same.

Java Copy

```java

public class Solution {
    public boolean canCross(int[] stones) {
        return can_Cross(stones, 0, 0);
    }
    public boolean can_Cross(int[] stones, int ind, int jumpsize) {
        if (ind == stones.length - 1) {
            return true;
        }
        int ind1 = Arrays.binarySearch(stones, ind + 1, stones.length, stones[ind] + jumpsize);
        if (ind1 >= 0 && can_Cross(stones, ind1, jumpsize)) {
            return true;
        }
        int ind2 = Arrays.binarySearch(stones, ind + 1, stones.length, stones[ind] + jumpsize - 1);
        if (ind2 >= 0 && can_Cross(stones, ind2, jumpsize - 1)) {
            return true;
        }
        int ind3 = Arrays.binarySearch(stones, ind + 1, stones.length, stones[ind] + jumpsize + 1);
        if (ind3 >= 0 && can_Cross(stones, ind3, jumpsize + 1)) {
            return true;
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(3^n)$. Recursion tree can grow upto 3^n.
- Space complexity : $O(n)$. Recursion of depth n is used.

Approach #3 Using Memorization [Accepted]

Algorithm

Another problem with above approaches is that we can make the same function calls coming through different paths e.g. For a given $currentIndex$, we can call the recursive function $canCross$ with the $jumpsize$, say n. This n could be resulting from previous $jumpsize$ being $n-1$,n or $n+1$. Thus, many redundant function calls could be made prolonging the running time. This redundancy can be removed by making use of memorization. We make use of a 2-d $memo$ array, initialized by -1s, to store the result returned from a function call for a particular $currentIndex$ and $jumpsize$. If the same $currentIndex$ and $jumpsize$ happens is encountered again, we can return the result directly using the $memo$ array. This helps to prune the search tree to a great extent.

Java Copy

```java
public class Solution {
    public boolean canCross(int[] stones) {
        int[][] memo = new int[stones.length][stones.length];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return can_Cross(stones, 0, 0, memo) == 1;
    }
    public int can_Cross(int[] stones, int ind, int jumpsize, int[][] memo) {
        if (memo[ind][jumpsize] >= 0) {
            return memo[ind][jumpsize];
        }
        for (int i = ind + 1; i < stones.length; i++) {
            int gap = stones[i] - stones[ind];
            if (gap >= jumpsize - 1 && gap <= jumpsize + 1) {
                if (can_Cross(stones, i, gap, memo) == 1) {
                    memo[ind][gap] = 1;
                    return 1;
                }
            }
        }
        memo[ind][jumpsize] = (ind == stones.length - 1) ? 1 : 0;
        return memo[ind][jumpsize];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$. Memorization will reduce time complexity to $O(n^3)$.
- Space complexity : $O(n^2)$. $memo$ matrix of size n^2 is used.

Approach #4 Using Memorization with Binary Search [Accepted]

Algorithm

We can optimize the above memorization approach, if we make use of Binary Search to find if a stone exists at $currentPostion + newjumpsize$ instead of searching linearly.

Java Copy

```java
public class Solution {
    public boolean canCross(int[] stones) {
        int[][] memo = new int[stones.length][stones.length];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return can_Cross(stones, 0, 0, memo) == 1;
    }
    public int can_Cross(int[] stones, int ind, int jumpsize, int[][] memo) {
        if (memo[ind][jumpsize] >= 0) {
            return memo[ind][jumpsize];
        }
        int ind1 = Arrays.binarySearch(stones, ind + 1, stones.length, stones[ind] + jumpsize);
        if (ind1 >= 0 && can_Cross(stones, ind1, jumpsize, memo) == 1) {
            memo[ind][jumpsize] = 1;
            return 1;
        }
        int ind2 = Arrays.binarySearch(stones, ind + 1, stones.length, stones[ind] + jumpsize - 1);
        if (ind2 >= 0 && can_Cross(stones, ind2, jumpsize - 1, memo) == 1) {
            memo[ind][jumpsize - 1] = 1;
            return 1;
        }
        int ind3 = Arrays.binarySearch(stones, ind + 1, stones.length, stones[ind] + jumpsize + 1);
        if (ind3 >= 0 && can_Cross(stones, ind3, jumpsize + 1, memo) == 1) {
            memo[ind][jumpsize + 1] = 1;
            return 1;
        }
        memo[ind][jumpsize] = ((ind == stones.length - 1) ? 1 : 0);
```

Complexity Analysis

- Time complexity : $O\big(n^2 log(n)\big)$. We traverse the complete dp matrix once $(O(n^2))$. For every entry we take atmost n numbers as pivot.
- Space complexity : $O(n^2)$. dp matrix of size n^2 is used.

Approach #5 Using Dynamic Programming[Accepted]

Algorithm

In the DP Approach, we make use of a hashmap map which contains $key : value$ pairs such that key refers to the position at which a stone is present and $value$ is a set containing the $jumpsize$ which can lead to the current stone position. We start by making a hashmap whose $keys$ are all the positions at which a stone is present and the $values$ are all empty except position 0 whose value contains 0. Then, we start traversing the elements(positions) of the given stone array in sequential order. For the $currentPosition$, for every possible $jumpsize$ in the $value$ set, we check if $currentPosition + newjumpsize$ exists in the map, where $newjumpsize$ can be either $jumpsize - 1$, $jumpsize$, $jumpsize + 1$. If so, we append the corresponding $value$ set with $newjumpsize$. We continue in the same manner. If at the end, the $value$ set corresponding to the last position is non-empty, we conclude that reaching the end is possible, otherwise, it isn't.

For more understanding see this animation-

Java Copy

```java
public class Solution {
    public boolean canCross(int[] stones) {
        HashMap<Integer, Set<Integer>> map = new HashMap<>();
        for (int i = 0; i < stones.length; i++) {
            map.put(stones[i], new HashSet<Integer>());
        }
        map.get(0).add(0);
        for (int i = 0; i < stones.length; i++) {
            for (int k : map.get(stones[i])) {
                for (int step = k - 1; step <= k + 1; step++) {
                    if (step > 0 && map.containsKey(stones[i] + step)) {
                        map.get(stones[i] + step).add(step);
                    }
                }
            }
        }
        return map.get(stones[stones.length - 1]).size() > 0;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two nested loops are there.
- Space complexity : $O(n^2)$. $hashmap$ size can grow upto n^2 .

Rate this article:

Previous Next

Comments: 32 Sort By

Type comment here... (Markdown is supported)

Preview Post

sean46 106 January 17, 2017 11:48 PM

Can someone explain why approach 3 is O(n^3)?

15 Share Reply

SHOW 3 REPLIES

thibauds 18 August 14, 2018 11:13 PM

I believe approach 5 is not O(n^2) but O(n * sqrt(n)). (Didnt check other approach)

The length of possible "jumpsize" set (which is the second loop we iterate on), is not is O(n).
Worst case is when we have one stone at each step (creating a lot of possible jumpsize), but to get a jump K, we need at least the stones 0-1-3-6-...-x-x+K-1, meaning to have a set of length K possible

Read More

12 Share Reply

SHOW 2 REPLIES

calvinchankf 2988 May 23, 2019 4:45 PM

Actually, It can be regarded as a tree problem, each node has 3 children. The goal is to check if we can reach to the end along the edges. We can do it with a Depth First Search with a Hashtable(to avoid redundant calculation)

Here is my 20 lines approach in python which beats 93.81%

Read More

11 Share Reply

Brookeran 10 October 18, 2017 2:24 AM

Seems the time complexity analysis here is quite suspicious, for approach #3, time complexity is O(n^2) to me, how can you get a O(n^3) when you fill in a 2-D memo matrix?

8 Share Reply

SHOW 3 REPLIES

lc19890306 42 October 15, 2019 8:29 AM

In Approach #3, the for loop should break when gap is greater than jumpsize + 1. Then the time complexity will be O(n^2) since we only visit three effective stones each iteration.

3 Share Reply

dag88ind 25 June 24, 2020 6:51 AM

Since the input is a vector and from each node we can only go to 3 neighboring nodes at best, DFS or BFS with memoization is really fast for this. It is not a full 3^n tree.

Time complexity analysis
|V| = N

Read More

1 Share Reply

ssmoon 16 May 9, 2020 2:23 AM

It's MEMOization, not memorization

1 Share Reply

Hammer001 235 May 5, 2019 1:17 PM

(Updates): Seems I am not the only one feeling confused about the Approach 3, based on the comments before....

I am confused about the memoization solution (Approach 3). I do not think that implementation is the optimal. I think we can definitely leverage the hash-table to store the stone position -> index and

Read More

1 Share Reply

happyzone8 16 April 16, 2018 10:20 AM

Thanks for sharing!

1 Share Reply

zzhai 1000 January 8, 2017 3:43 AM

memo[ind][jumpsize] = ((ind == stones.length - 1) ? 1 : 0);

Properly using parentheses makes expressions clear, but the outer parentheses in this statement look a bit over-conservative.
http://introcs.cs.princeton.edu/java/11precedence/

1 Share Reply

‹ 1 2 3 4 ›