Problem 1377: Frog Position After T Seconds

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given an undirected tree consisting of

n

vertices numbered from

1

to

n

. A frog starts jumping from

vertex 1

. In one second, the frog jumps from its current vertex to another

unvisited

vertex if they are directly connected. The frog can not jump back to a visited vertex. In case the frog can jump to several vertices, it jumps randomly to one of them with the same probability. Otherwise, when the frog can not jump to any unvisited vertex, it jumps forever on the same vertex.

The edges of the undirected tree are given in the array

edges

, where

edges[i] = [a

i

, b

i

]

means that exists an edge connecting the vertices

a

i

and

b

i

.

Return the probability that after

t

seconds the frog is on the vertex

target

.

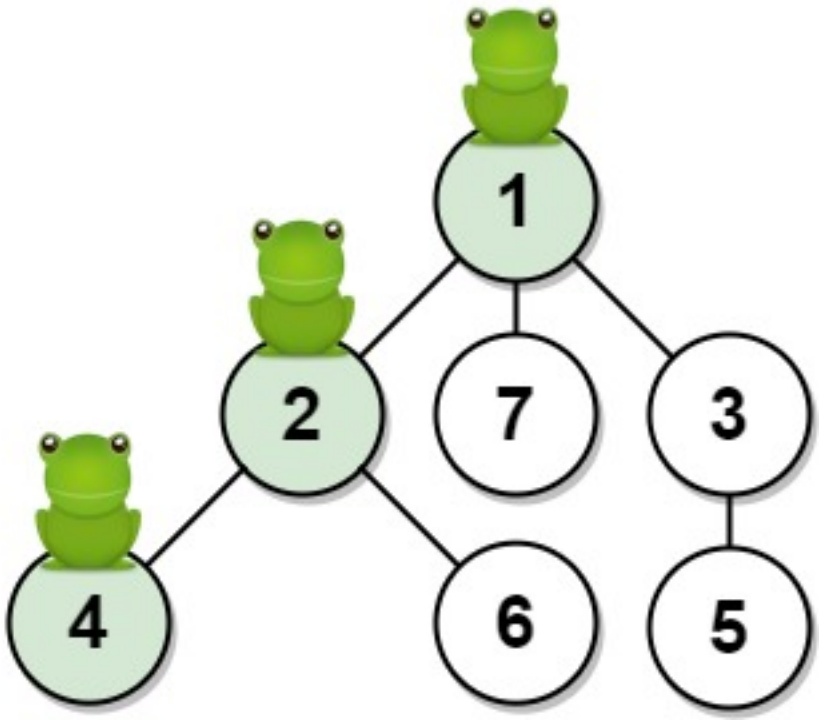
Answers within

10

-5

of the actual answer will be accepted.

Example 1:



Input:

n = 7, edges = [[1,2],[1,3],[1,7],[2,4],[2,6],[3,5]], t = 2, target = 4

Output:

0.16666666666666666

Explanation:

The figure above shows the given graph. The frog starts at vertex 1, jumping with 1/3 probability to the vertex 2 after

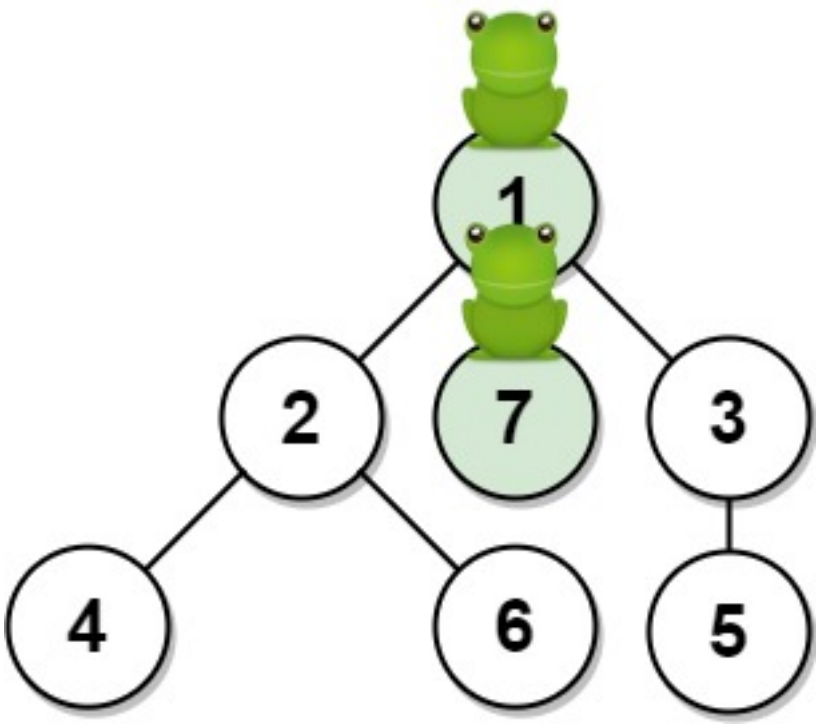
second 1

and then jumping with 1/2 probability to vertex 4 after

second 2

. Thus the probability for the frog is on the vertex 4 after 2 seconds is 1/3 * 1/2 = 1/6 = 0.16666666666666666.

Example 2:



Input:

n = 7, edges = [[1,2],[1,3],[1,7],[2,4],[2,6],[3,5]], t = 1, target = 7

Output:

0.3333333333333333

Explanation:

The figure above shows the given graph. The frog starts at vertex 1, jumping with 1/3 = 0.3333333333333333 probability to the vertex 7 after

second 1

.

Constraints:

1 <= n <= 100

edges.length == n - 1

edges[i].length == 2

1 <= a

i

, b

i

<= n

1 <= t <= 50

1 <= target <= n

Code Snippets

C++:

```
class Solution {
public:
double frogPosition(int n, vector<vector<int>>& edges, int t, int target) {
```

```
}
};
```

Java:

```
class Solution {
public double frogPosition(int n, int[][] edges, int t, int target) {

}
}
```

Python3:

```
class Solution:
def frogPosition(self, n: int, edges: List[List[int]], t: int, target: int)
-> float:
```

Python:

```
class Solution(object):
def frogPosition(self, n, edges, t, target):
"""
:type n: int
:type edges: List[List[int]]
:type t: int
:type target: int
:rtype: float
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[][]} edges
* @param {number} t
* @param {number} target
* @return {number}
*/
var frogPosition = function(n, edges, t, target) {

};
```

TypeScript:

```
function frogPosition(n: number, edges: number[][], t: number, target:
number): number {

};
```

C#:

```
public class Solution {
public double FrogPosition(int n, int[][] edges, int t, int target) {

}
}
```

C:

```
double frogPosition(int n, int** edges, int edgesSize, int* edgesColSize, int
t, int target) {

}
```

Go:

```
func frogPosition(n int, edges [][]int, t int, target int) float64 {

}
```

Kotlin:

```
class Solution {
fun frogPosition(n: Int, edges: Array<IntArray>, t: Int, target: Int): Double
{

}
}
```

Swift:

```
class Solution {
func frogPosition(_ n: Int, _ edges: [[Int]], _ t: Int, _ target: Int) ->
Double {
```

```
}
}
```

Rust:

```
impl Solution {
pub fn frog_position(n: i32, edges: Vec<Vec<i32>>, t: i32, target: i32) ->
f64 {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @param {Integer} t
# @param {Integer} target
# @return {Float}
def frog_position(n, edges, t, target)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @param Integer $t
* @param Integer $target
* @return Float
*/
function frogPosition($n, $edges, $t, $target) {

}
}
```

Dart:

```
class Solution {
double frogPosition(int n, List<List<int>> edges, int t, int target) {

}
}
```

Scala:

```
object Solution {
def frogPosition(n: Int, edges: Array[Array[Int]], t: Int, target: Int):
Double = {

}
}
```

Elixir:

```
defmodule Solution do
@spec frog_position(n :: integer, edges :: [[integer]], t :: integer, target
:: integer) :: float
def frog_position(n, edges, t, target) do

end
end
```

Erlang:

```
-spec frog_position(N :: integer(), Edges :: [[integer()]], T :: integer(),
Target :: integer()) -> float().
frog_position(N, Edges, T, Target) ->
.
```

Racket:

```
(define/contract (frog-position n edges t target)
(-> exact-integer? (listof (listof exact-integer?)) exact-integer?
exact-integer? flonum?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Frog Position After T Seconds
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
double frogPosition(int n, vector<vector<int>>& edges, int t, int target) {

}
};
```

Java Solution:

```
/**
* Problem: Frog Position After T Seconds
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public double frogPosition(int n, int[][] edges, int t, int target) {

}
}
```

Python3 Solution:

```
"""
Problem: Frog Position After T Seconds
Difficulty: Hard
Tags: array, tree, graph, search
```

```
Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def frogPosition(self, n: int, edges: List[List[int]], t: int, target: int)
-> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def frogPosition(self, n, edges, t, target):
"""
:type n: int
:type edges: List[List[int]]
:type t: int
:type target: int
:rtype: float
"""
```

JavaScript Solution:

```
/**
* Problem: Frog Position After T Seconds
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {number} n
* @param {number[][]} edges
* @param {number} t
* @param {number} target
```

```
* @return {number}
*/
var frogPosition = function(n, edges, t, target) {

};
```

TypeScript Solution:

```
/**
* Problem: Frog Position After T Seconds
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function frogPosition(n: number, edges: number[][], t: number, target:
number): number {

};
```

C# Solution:

```
/*
* Problem: Frog Position After T Seconds
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public double FrogPosition(int n, int[][] edges, int t, int target) {

}
}
```

C Solution:

```
/*
* Problem: Frog Position After T Seconds
* Difficulty: Hard
* Tags: array, tree, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

double frogPosition(int n, int** edges, int edgesSize, int* edgesColSize, int
t, int target) {

}
```

Go Solution:

```
// Problem: Frog Position After T Seconds
// Difficulty: Hard
// Tags: array, tree, graph, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func frogPosition(n int, edges [][]int, t int, target int) float64 {

}
```

Kotlin Solution:

```
class Solution {
fun frogPosition(n: Int, edges: Array<IntArray>, t: Int, target: Int): Double
{

}
}
```

Swift Solution:

```
class Solution {
func frogPosition(_ n: Int, _ edges: [[Int]], _ t: Int, _ target: Int) ->
Double {

}
}
```

Rust Solution:

```
// Problem: Frog Position After T Seconds
// Difficulty: Hard
// Tags: array, tree, graph, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn frog_position(n: i32, edges: Vec<Vec<i32>>, t: i32, target: i32) ->
f64 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @param {Integer} t
# @param {Integer} target
# @return {Float}
def frog_position(n, edges, t, target)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
```

```
* @param Integer $t
* @param Integer $target
* @return Float
*/
function frogPosition($n, $edges, $t, $target) {

}
}
```

Dart Solution:

```
class Solution {
double frogPosition(int n, List<List<int>> edges, int t, int target) {

}
}
```

Scala Solution:

```
object Solution {
def frogPosition(n: Int, edges: Array[Array[Int]], t: Int, target: Int):
Double = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec frog_position(n :: integer, edges :: [[integer]], t :: integer, target
:: integer) :: float
def frog_position(n, edges, t, target) do

end
end
```

Erlang Solution:

```
-spec frog_position(N :: integer(), Edges :: [[integer()]], T :: integer(),
Target :: integer()) -> float().
frog_position(N, Edges, T, Target) ->
```

```
.
```

Racket Solution:

```
(define/contract (frog-position n edges t target)
(-> exact-integer? (listof (listof exact-integer?)) exact-integer?
exact-integer? flonum?)
)
```