Problem 913: Cat and Mouse

Problem Information

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

Problem Description

A game on an

undirected

graph is played by two players, Mouse and Cat, who alternate turns.

The graph is given as follows:

graph[a]

is a list of all nodes

b

such that

ab

is an edge of the graph.

The mouse starts at node

1

and goes first, the cat starts at node

2

and goes second, and there is a hole at node

0

.

During each player's turn, they

must

travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it

must

travel to any node in

graph[1]

.

Additionally, it is not allowed for the Cat to travel to the Hole (node

0

).

Then, the game can end in three ways:

If ever the Cat occupies the same node as the Mouse, the Cat wins.

If ever the Mouse reaches the Hole, the Mouse wins.

If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.

Given a

graph

, and assuming both players play optimally, return

1

if the mouse wins the game,

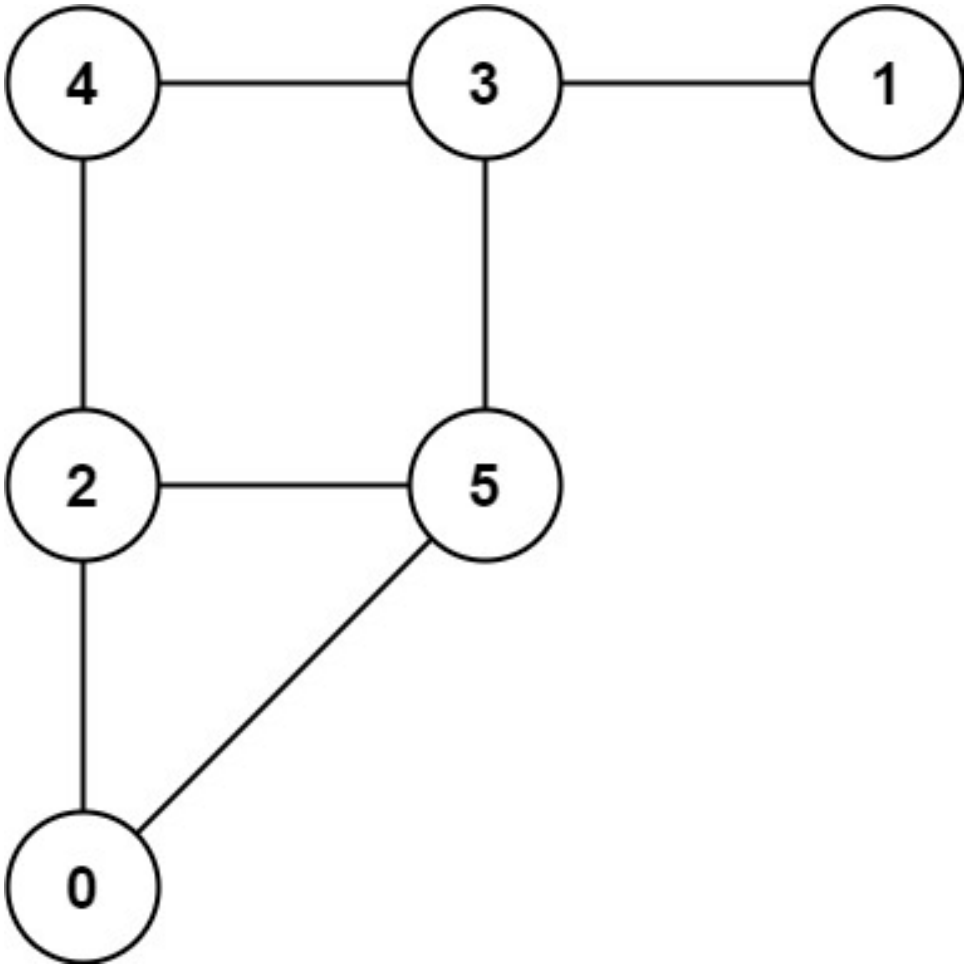
2

if the cat wins the game, or

0

if the game is a draw.

Example 1:



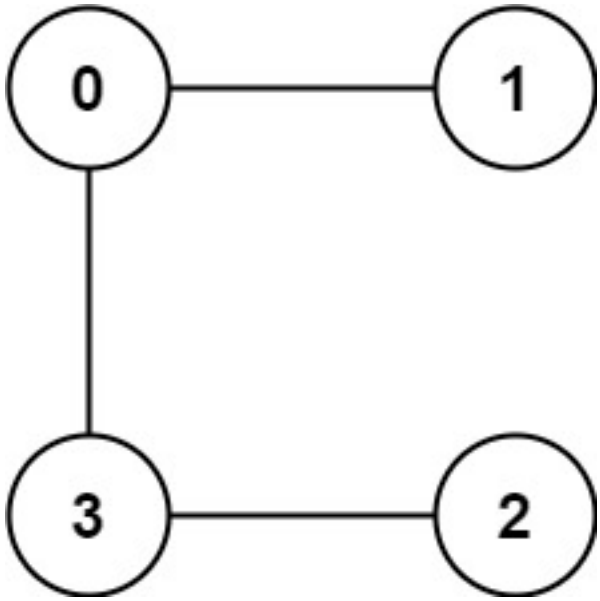
Input:

graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

Output:

0

Example 2:



Input:

graph = [[1,3],[0],[3],[0,2]]

Output:

1

Constraints:

3 <= graph.length <= 50

1 <= graph[i].length < graph.length

0 <= graph[i][j] < graph.length

graph[i][j] != i

graph[i]

is unique.

The mouse and the cat can always move.

Code Snippets

C++:

```
class Solution {
public:
int catMouseGame(vector<vector<int>>& graph) {

}
};
```

Java:

```
class Solution {
public int catMouseGame(int[][] graph) {

}
}
```

Python3:

```
class Solution:
def catMouseGame(self, graph: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def catMouseGame(self, graph):
"""
:type graph: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} graph
* @return {number}
*/
var catMouseGame = function(graph) {

};
```

TypeScript:

```
function catMouseGame(graph: number[][]): number {

};
```

C#:

```
public class Solution {
public int CatMouseGame(int[][] graph) {

}
}
```

C:

```
int catMouseGame(int** graph, int graphSize, int* graphColSize) {

}
```

Go:

```
func catMouseGame(graph [][]int) int {

}
```

Kotlin:

```
class Solution {
fun catMouseGame(graph: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func catMouseGame(_ graph: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn cat_mouse_game(graph: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} graph
# @return {Integer}
def cat_mouse_game(graph)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $graph
* @return Integer
*/
function catMouseGame($graph) {

}
}
```

Dart:

```
class Solution {
int catMouseGame(List<List<int>> graph) {

}
```

```
}
```

Scala:

```
object Solution {
def catMouseGame(graph: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec cat_mouse_game(graph :: [[integer]]) :: integer
def cat_mouse_game(graph) do

end
end
```

Erlang:

```
-spec cat_mouse_game(Graph :: [[integer()]]) -> integer().
cat_mouse_game(Graph) ->
.
```

Racket:

```
(define/contract (cat-mouse-game graph)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Cat and Mouse
* Difficulty: Hard
* Tags: graph, dp, math, sort
*
```

```
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int catMouseGame(vector<vector<int>>& graph) {

}
};
```

Java Solution:

```
/**
* Problem: Cat and Mouse
* Difficulty: Hard
* Tags: graph, dp, math, sort
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int catMouseGame(int[][] graph) {

}
}
```

Python3 Solution:

```
"""
Problem: Cat and Mouse
Difficulty: Hard
Tags: graph, dp, math, sort

Approach: Dynamic programming with memoization or tabulation
Time Complexity: O(n * m) where n and m are problem dimensions
Space Complexity: O(n) or O(n * m) for DP table
"""
```

```
class Solution:
def catMouseGame(self, graph: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def catMouseGame(self, graph):
"""
:type graph: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Cat and Mouse
* Difficulty: Hard
* Tags: graph, dp, math, sort
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[][]} graph
* @return {number}
*/
var catMouseGame = function(graph) {

};
```

TypeScript Solution:

```
/**
* Problem: Cat and Mouse
* Difficulty: Hard
* Tags: graph, dp, math, sort
```

```
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

function catMouseGame(graph: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Cat and Mouse
* Difficulty: Hard
* Tags: graph, dp, math, sort
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int CatMouseGame(int[][] graph) {

}
}
```

C Solution:

```
/*
* Problem: Cat and Mouse
* Difficulty: Hard
* Tags: graph, dp, math, sort
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

int catMouseGame(int** graph, int graphSize, int* graphColSize) {
```

```
}
```

Go Solution:

```
// Problem: Cat and Mouse
// Difficulty: Hard
// Tags: graph, dp, math, sort
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
// Space Complexity: O(n) or O(n * m) for DP table

func catMouseGame(graph [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun catMouseGame(graph: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func catMouseGame(_ graph: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Cat and Mouse
// Difficulty: Hard
// Tags: graph, dp, math, sort
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
```

```
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn cat_mouse_game(graph: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} graph
# @return {Integer}
def cat_mouse_game(graph)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $graph
* @return Integer
*/
function catMouseGame($graph) {

}
}
```

Dart Solution:

```
class Solution {
int catMouseGame(List<List<int>> graph) {

}
}
```

Scala Solution:

```
object Solution {
def catMouseGame(graph: Array[Array[Int]]): Int = {
```

```
}
}
```

Elixir Solution:

```
defmodule Solution do
@spec cat_mouse_game(graph :: [[integer]]) :: integer
def cat_mouse_game(graph) do

end
end
```

Erlang Solution:

```
-spec cat_mouse_game(Graph :: [[integer()]]) -> integer().
cat_mouse_game(Graph) ->
.
```

Racket Solution:

```
(define/contract (cat-mouse-game graph)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```