Problem 2192: All Ancestors of a Node in a Directed Acyclic Graph

Problem Information

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

Problem Description

You are given a positive integer

n

representing the number of nodes of a

Directed Acyclic Graph

(DAG). The nodes are numbered from

0

to

n - 1

(

inclusive

).

You are also given a 2D integer array

edges

, where

edges[i] = [from

i

, to

i

]

denotes that there is a

unidirectional

edge from

from

i

to

to

i

in the graph.

Return

a list

answer

, where

answer[i]

is the

list of ancestors

of the

i

th

node, sorted in

ascending order

.

A node

u

is an

ancestor

of another node

v

if

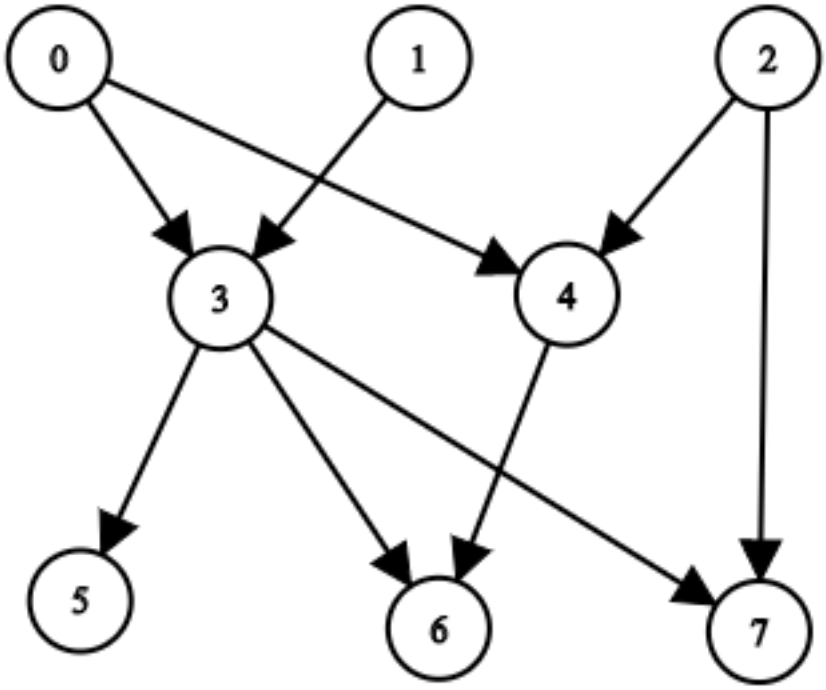
u

can reach

v

via a set of edges.

Example 1:



Input:

n = 8, edgeList = [[0,3],[0,4],[1,3],[2,4],[2,7],[3,5],[3,6],[3,7],[4,6]]

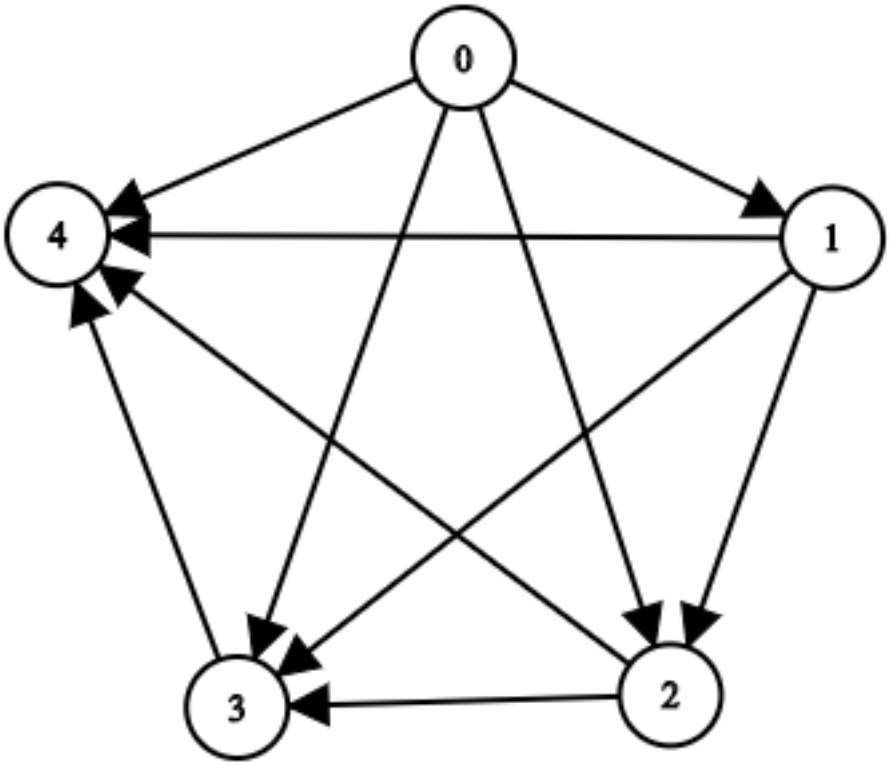
Output:

[[],[],[],[0,1],[0,2],[0,1,3],[0,1,2,3,4],[0,1,2,3]]

Explanation:

The above diagram represents the input graph. - Nodes 0, 1, and 2 do not have any ancestors. - Node 3 has two ancestors 0 and 1. - Node 4 has two ancestors 0 and 2. - Node 5 has three ancestors 0, 1, and 3. - Node 6 has five ancestors 0, 1, 2, 3, and 4. - Node 7 has four ancestors 0, 1, 2, and 3.

Example 2:



Input:

n = 5, edgeList = [[0,1],[0,2],[0,3],[0,4],[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]

Output:

[[],[0],[0,1],[0,1,2],[0,1,2,3]]

Explanation:

The above diagram represents the input graph. - Node 0 does not have any ancestor. - Node 1 has one ancestor 0. - Node 2 has two ancestors 0 and 1. - Node 3 has three ancestors 0, 1, and 2. - Node 4 has four ancestors 0, 1, 2, and 3.

Constraints:

1 <= n <= 1000

0 <= edges.length <= min(2000, n * (n - 1) / 2)

edges[i].length == 2

0 <= from

i

, to

i

<= n - 1

from

i

!= to

i

There are no duplicate edges.

The graph is

directed

and

acyclic

.

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> getAncestors(int n, vector<vector<int>>& edges) {
```

```
}
};
```

Java:

```
class Solution {
public List<List<Integer>> getAncestors(int n, int[][] edges) {

}
}
```

Python3:

```
class Solution:
def getAncestors(self, n: int, edges: List[List[int]]) -> List[List[int]]:
```

Python:

```
class Solution(object):
def getAncestors(self, n, edges):
"""
:type n: int
:type edges: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript:

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

};
```

TypeScript:

```
function getAncestors(n: number, edges: number[][]): number[][] {

};
```

C#:

```
public class Solution {
public IList<IList<int>> GetAncestors(int n, int[][] edges) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** getAncestors(int n, int** edges, int edgesSize, int* edgesColSize, int*
returnSize, int** returnColumnSizes) {

}
```

Go:

```
func getAncestors(n int, edges [][]int) [][]int {

}
```

Kotlin:

```
class Solution {
fun getAncestors(n: Int, edges: Array<IntArray>): List<List<Int>> {

}
}
```

Swift:

```
class Solution {
func getAncestors(_ n: Int, _ edges: [[Int]]) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn get_ancestors(n: i32, edges: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @return {Integer[][]}
def get_ancestors(n, edges)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @return Integer[][]
*/
function getAncestors($n, $edges) {

}
}
```

Dart:

```
class Solution {
List<List<int>> getAncestors(int n, List<List<int>> edges) {

}
}
```

Scala:

```
object Solution {
def getAncestors(n: Int, edges: Array[Array[Int]]): List[List[Int]] = {
```

```
}
}
```

Elixir:

```
defmodule Solution do
@spec get_ancestors(n :: integer, edges :: [[integer]]) :: [[integer]]
def get_ancestors(n, edges) do

end
end
```

Erlang:

```
-spec get_ancestors(N :: integer(), Edges :: [[integer()]]) -> [[integer()]].
get_ancestors(N, Edges) ->
.
```

Racket:

```
(define/contract (get-ancestors n edges)
(-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: All Ancestors of a Node in a Directed Acyclic Graph
* Difficulty: Medium
* Tags: array, graph, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class Solution {
public:
vector<vector<int>> getAncestors(int n, vector<vector<int>>& edges) {

}
};
```

Java Solution:

```
/**
* Problem: All Ancestors of a Node in a Directed Acyclic Graph
* Difficulty: Medium
* Tags: array, graph, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public List<List<Integer>> getAncestors(int n, int[][] edges) {

}
}
```

Python3 Solution:

```
"""
Problem: All Ancestors of a Node in a Directed Acyclic Graph
Difficulty: Medium
Tags: array, graph, sort, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

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

Python Solution:

```
class Solution(object):
def getAncestors(self, n, edges):
"""
:type n: int
:type edges: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: All Ancestors of a Node in a Directed Acyclic Graph
* Difficulty: Medium
* Tags: array, graph, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number} n
* @param {number[][]} edges
* @return {number[][]}
*/
var getAncestors = function(n, edges) {

};
```

TypeScript Solution:

```
/**
* Problem: All Ancestors of a Node in a Directed Acyclic Graph
* Difficulty: Medium
* Tags: array, graph, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
function getAncestors(n: number, edges: number[][]): number[][] {

};
```

C# Solution:

```
/*
* Problem: All Ancestors of a Node in a Directed Acyclic Graph
* Difficulty: Medium
* Tags: array, graph, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public IList<IList<int>> GetAncestors(int n, int[][] edges) {

}
}
```

C Solution:

```
/*
* Problem: All Ancestors of a Node in a Directed Acyclic Graph
* Difficulty: Medium
* Tags: array, graph, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
```

```
int** getAncestors(int n, int** edges, int edgesSize, int* edgesColSize, int*
returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: All Ancestors of a Node in a Directed Acyclic Graph
// Difficulty: Medium
// Tags: array, graph, sort, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func getAncestors(n int, edges [][]int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun getAncestors(n: Int, edges: Array<IntArray>): List<List<Int>> {

}
}
```

Swift Solution:

```
class Solution {
func getAncestors(_ n: Int, _ edges: [[Int]]) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: All Ancestors of a Node in a Directed Acyclic Graph
// Difficulty: Medium
// Tags: array, graph, sort, search
//
```

```
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn get_ancestors(n: i32, edges: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} edges
# @return {Integer[][]}
def get_ancestors(n, edges)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $edges
* @return Integer[][]
*/
function getAncestors($n, $edges) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> getAncestors(int n, List<List<int>> edges) {

}
}
```

Scala Solution:

```
object Solution {
def getAncestors(n: Int, edges: Array[Array[Int]]): List[List[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec get_ancestors(n :: integer, edges :: [[integer]]) :: [[integer]]
def get_ancestors(n, edges) do

end
end
```

Erlang Solution:

```
-spec get_ancestors(N :: integer(), Edges :: [[integer()]]) -> [[integer()]].
get_ancestors(N, Edges) ->
.
```

Racket Solution:

```
(define/contract (get-ancestors n edges)
(-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)))
)
```