<https://leetcode.com/problems/minimum-height-trees/>

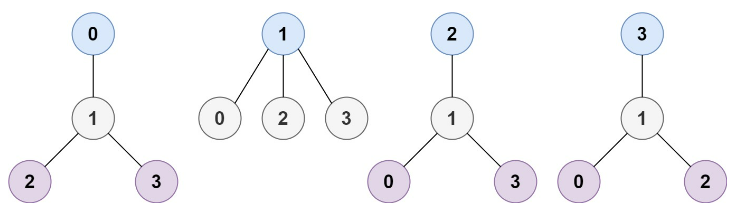
A tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h))  are called **minimum height trees** (MHTs).

Return *a list of all* ***MHTs'*** *root labels*. You can return the answer in **any order**.

The **height** of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

**Example 1:**

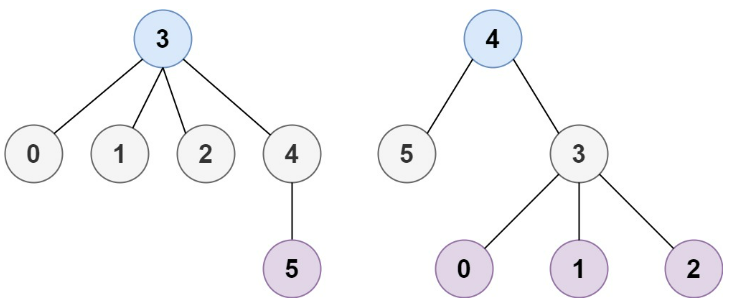


Input: n = 4, edges = [[1,0],[1,2],[1,3]]

Output: [1]

Explanation: As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.

**Example 2:**



Input: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]

Output: [3,4]

**Constraints:**

1 <= n <= 2 \* 10^4

edges.length == n - 1

0 <= ai, bi < n

ai != bi

All the pairs (ai, bi) are distinct.

The given input is **guaranteed** to be a tree and there will be **no repeated** edges.

**Attempt 1: 2022-12-21**

**Solution 1: Recursive traversal (60 min)**

**Style 1: Use Build graph with map to increase store and retrieve speed as O(1)**

class Solution {

    public List<Integer> findMinHeightTrees(int n, int[][] edges) {

        // Corner case

        // This is needed...since when there is only 1 vertex... the incident of

        // it will be 0..this case is not included in the following discussion...

        List<Integer> list = new ArrayList<Integer>();

        if(n == 1) {

            list.add(0);

            return list;

        }

        List<Integer> result = new ArrayList<Integer>();

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < n; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(int[] edge : edges) {

            graph.get(edge[0]).add(edge[1]);

            graph.get(edge[1]).add(edge[0]);

        }

        List<Integer> leaves = new ArrayList<Integer>();

        for(int i = 0; i < n; i++) {

            if(graph.get(i).size() == 1) {

                leaves.add(i);

            }

        }

        while(n > 2) {

            n -= leaves.size();

            List<Integer> newLeaves = new ArrayList<Integer>();

            for(int i = 0; i < leaves.size(); i++) {

                int leaf = leaves.get(i);

                // Since its a leaf node, because incident=1 it has only one adjacent

                // node stored in its adjacent list, to get this adjacent node just

                // pick index=0 element in the adjacent list, no need iterator().next()

                /\*\*

                  \* How graph.get(leaf).iterator().next() works ?

                  \* Since current node is leave node, it only have 1 connection node

                  \* e.g      4

                  \*        / | \

                  \*        1  2  3

                  \* For nodes 1, 2, 3, they are all leave node, and connection node as

                  \* only 4, to find node 1, 2, 3's connection node 4 is able to get by

                  \* Set's iterator, as only 1 item 4 will be in 1, 2, 3's adjacence set

                  \* so, we can say

                  \* adj.get(1).iterator().next() = 4

                  \* adj.get(2).iterator().next() = 4

                  \* adj.get(3).iterator().next() = 4

                  \*/

                int leaf\_adj = graph.get(leaf).get(0);

                // Note: remove(leaf) is wrong method since remove(int index):

                // Removes the element at the specified position in this list.

                // remove(Object o):

                // Removes the first occurrence of the specified element from this list,

                // if it is present.

                graph.get(leaf\_adj).remove(Integer.valueOf(leaf));

                if(graph.get(leaf\_adj).size() == 1) {

                    newLeaves.add(leaf\_adj);

                }

            }

            leaves = newLeaves;

        }

        return leaves;

    }

}

**Style 2: Use indegree + BFS, similar to Course Schedule**

class Solution {

    public List<Integer> findMinHeightTrees(int n, int[][] edges) {

        // Corner case

        // This is needed...since when there is only 1 vertex... the incident of

        // it will be 0..this case is not included in the following discussion...

        List<Integer> list = new ArrayList<Integer>();

        if(n == 1) {

            list.add(0);

            return list;

        }

        List<Integer> result = new ArrayList<Integer>();

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < n; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(int[] edge : edges) {

            graph.get(edge[0]).add(edge[1]);

            graph.get(edge[1]).add(edge[0]);

        }

        Queue<Integer> q = new LinkedList<Integer>();

        for(int i = 0; i < n; i++) {

            if(graph.get(i).size() == 1) {

                q.offer(i);

            }

        }

        while(n > 2) {

            int size = q.size();

            n -= size;

            for(int i = 0; i < size; i++) {

                int leaf = q.poll();

                int leaf\_adj = graph.get(leaf).get(0);

                // Note: remove(leaf) is wrong method since remove(int index):

                // Removes the element at the specified position in this list.

                // remove(Object o):

                // Removes the first occurrence of the specified element from this list,

                // if it is present.

                graph.get(leaf\_adj).remove(Integer.valueOf(leaf));

                // After remove current leaf from leaf\_adj's adjacent list,

                // check adjacent list size if equal to 1 or not (1 for incident=1),

                // no need separately build indegree array

                if(graph.get(leaf\_adj).size() == 1) {

                    q.offer(leaf\_adj);

                }

            }

        }

        result.addAll(q);

        return result;

    }

}

===============================================================

Below is the conventional version still use indegree array

class Solution {

    public List<Integer> findMinHeightTrees(int n, int[][] edges) {

        // This is needed...since when there is only 1 vertex... the incident of

        // it will be 0..this case is not included in the following discussion...

        if(n == 1) {

            List<Integer> result = new ArrayList<Integer>();

            result.add(0);

            return result;

        }

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < n; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(int[] edge : edges) {

            graph.get(edge[0]).add(edge[1]);

            graph.get(edge[1]).add(edge[0]);

        }

        Queue<Integer> q = new LinkedList<Integer>();

        int[] indegree = new int[n];

        for(int i = 0; i < n; i++) {

            indegree[i] = graph.get(i).size();

            if(indegree[i] == 1) {

                q.offer(i);

            }

        }

        List<Integer> result = new ArrayList<Integer>();

        while(n > 2) {

            int size = q.size();

            n -= size;

            for(int i = 0; i < size; i++) {

                int cur = q.poll();

                List<Integer> neighbors = graph.get(cur);

                for(int neighbor : neighbors) {

                    indegree[neighbor]--;

                    if(indegree[neighbor] == 1) {

                        q.offer(neighbor);

                    }

                }

            }

        }

        result.addAll(q);

        return result;

    }

}

**Refer to**

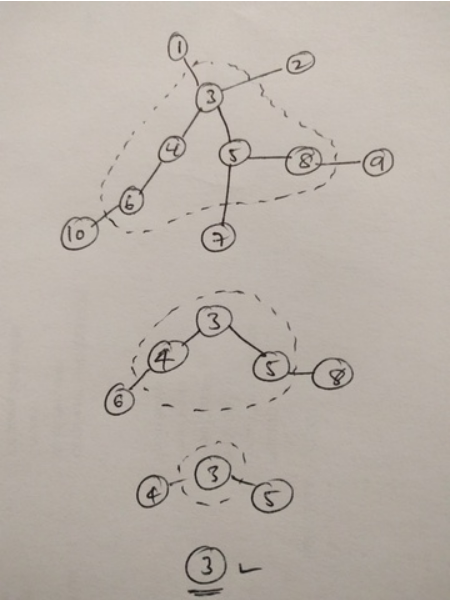
<http://www.cnblogs.com/grandyang/p/5000291.html>

这道题虽然是树的题目, 但是跟其最接近的题目是Course Schedule 课程清单和Course Schedule II 课程清单之二。由于LeetCode中的树的题目主要都是针对于二叉树的, 而这道题虽说是树但其实本质是想考察图的知识，这道题刚开始在拿到的时候，我最先想到的解法是遍历的点，以每个点都当做根节点，算出高度，然后找出最小的，但是一时半会又写不出程序来，于是上网看看大家的解法，**发现大家推崇的方法是一个类似剥洋葱的方法，就是一层一层的褪去叶节点，最后剩下的一个或两个节点就是我们要求的最小高度树的根节点**，这种思路非常的巧妙，而且实现起来也不难，跟之前那到课程清单的题一样, 我们需要建立一个图g, 是一个二维数组, 其中g[i]是一个一维数组, 保存了i节点可以到达的所有节点。**我们开始将所有只有一个连接边的节点(叶节点)都存入到一个队列queue中，然后我们遍历每一个叶节点，通过图来找到和其相连的节点，并且在其相连节点的集合中将该叶节点删去，如果删完后此节点也也变成一个叶节点了，加入队列中，再下一轮删除。那么我们删到什么时候呢，当节点数小于等于2时候停止，此时剩下的一个或两个节点就是我们要求的最小高度树的根节点啦**

**Refer to**

[https://leetcode.com/problems/minimum-height-trees/solutions/76055/](https://leetcode.com/problems/minimum-height-trees/solutions/76055/share-some-thoughts/?orderBy=most_votes)

Basically, the idea is to eat up all the leaves at the same time, until one/two leaves are left.



First let's review some statement for tree in graph theory:

(1) A tree is an undirected graph in which any two vertices are connected by exactly one path.

(2) Any connected graph who has n nodes with n-1 edges is a tree.

(3) The degree of a vertex of a graph is the number of edges incident to the vertex.

(4) A leaf is a vertex of degree 1. An internal vertex is a vertex of degree at least 2.

(5) A path graph is a tree with two or more vertices that is not branched at all.

(6) A tree is called a rooted tree if one vertex has been designated the root.

(7) The height of a rooted tree is the number of edges on the longest downward path between root and a leaf.

OK. Let's stop here and look at our problem.

Our problem want us to find the minimum height trees and return their root labels. First we can think about a simple case -- a path graph.

For a path graph of n nodes, find the minimum height trees is trivial. Just designate the middle point(s) as roots.

Despite its triviality, let design a algorithm to find them.

Suppose we don't know n, nor do we have random access of the nodes. We have to traversal. It is very easy to get the idea of two pointers. One from each end and move at the same speed. When they meet or they are one step away, (depends on the parity of n), we have the roots we want.

This gives us a lot of useful ideas to crack our real problem.

For a tree we can do some thing similar. We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

It is easy to see that the last two pointers are from the two ends of the longest path in the graph.

The actual implementation is similar to the BFS topological sort. Remove the leaves, update the degrees of inner vertexes. Then remove the new leaves. Doing so level by level until there are 2 or 1 nodes left. What's left is our answer!

The time complexity and space complexity are both O(n).

Note that for a tree we always have V = n, E = n-1.

**Java**

public List<Integer> findMinHeightTrees(int n, int[][] edges) {

    if (n == 1) return Collections.singletonList(0);

    List<Set<Integer>> adj = new ArrayList<>(n);

    for (int i = 0; i < n; ++i) adj.add(new HashSet<>());

    for (int[] edge : edges) {

        adj.get(edge[0]).add(edge[1]);

        adj.get(edge[1]).add(edge[0]);

    }

    List<Integer> leaves = new ArrayList<>();

    for (int i = 0; i < n; ++i)

        if (adj.get(i).size() == 1) leaves.add(i);

    while (n > 2) {

        n -= leaves.size();

        List<Integer> newLeaves = new ArrayList<>();

        for (int i : leaves) {

            int j = adj.get(i).iterator().next();

            adj.get(j).remove(i);

            if (adj.get(j).size() == 1) newLeaves.add(j);

        }

        leaves = newLeaves;

    }

    return leaves;

}

**Refer to**

<https://leetcode.com/problems/minimum-height-trees/solutions/900035/minimum-height-trees>

**Overview**

As the hints suggest, this problem is related to the [graph](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)) data structure. Moreover, it is closely related to the problems of [Course Schedule](https://leetcode.com/problems/course-schedule/) and [Course Schedule II](https://leetcode.com/problems/course-schedule-ii/). This relationship is not evident, yet it is the key to solve the problem, as one will see later.

First of all, as a ***straight-forward*** way to solve the problem, we can simply follow the requirements of the problem, as follows:

Starting from each node in the graph, we treat it as a ***root*** to build a tree. Furthermore, we would like to know the distance between this root node and the rest of the nodes. The maximum of the distance would be the ***height*** of this tree.

Then according to the definition of **Minimum Height Tree**(MHT), we simply filter out the roots that have the minimal height among all the trees.

The first step we describe above is actually the problem of [Maximum Depth of N-ary Tree](https://leetcode.com/problems/maximum-depth-of-n-ary-tree/), which is to find the maximum distance from the root to the leaf nodes. For this, we can either apply the [Depth-First Search](https://leetcode.com/explore/learn/card/queue-stack/232/practical-application-stack/)(**DFS**) or [Breadth-First Search](https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/)(**BFS**) algorithms.

Without a rigid proof, we can see that the above straight-forward solution is *correct*, and it would work for most of the test cases.

However, this solution is not efficient, whose time complexity would be O(N^2) where N is the number of nodes in the tree. As one can imagine, it will result in ***Time Limit Exceeded*** exception in the online judge.

As a spoiler alert, in this article, we will present a ***topological sorting***alike algorithm with time complexity of O(N), which is also the algorithm to solve the well-known course schedule problems.

**Approach 1: Topological Sorting**

**Intuition**

First of all, let us clarify some concepts.

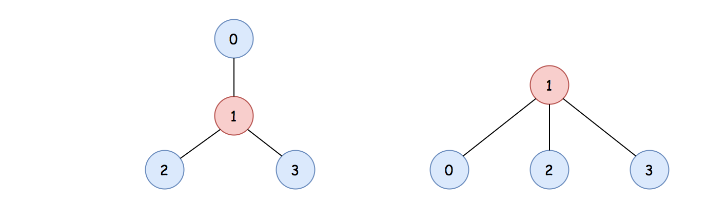
The ***distance*** between two nodes is the number of edges that connect the two nodes.

Note, normally there could be multiple paths to connect nodes in a graph. In our case though, since the input graph can form a tree from any node, as specified in the problem, there could only be ***one path*** between any two nodes. In addition, there would be no cycle in the graph. As a result, there would be no ambiguity in the above definition of *distance*.

The ***height*** of a tree can be defined as the *maximum distance* between the root and all its leaf nodes.

With the above definitions, we can rephrase the problem as finding out the nodes that are *overall* close to all other nodes, especially the leaf nodes.

If we view the graph as an *area of circle*, and the leaf nodes as the *peripheral* of the circle, then what we are looking for are actually the ***centroids*** of the circle, *i.e.* nodes that is close to all the peripheral nodes (leaf nodes).



For instance, in the above graph, it is clear that the node with the value 1 is the *centroid* of the graph. If we pick the node 1 as the root to form a tree, we would obtain a tree with the *minimum height*, compared to other trees that are formed with any other nodes.

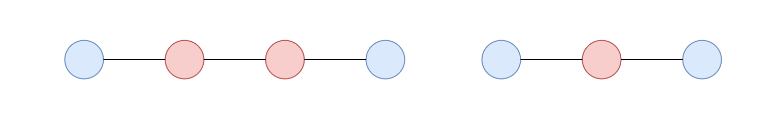
Before we proceed, here we make one assertion which is essential to the algorithm.

For the tree-alike graph, the number of centroids is no more than 2.

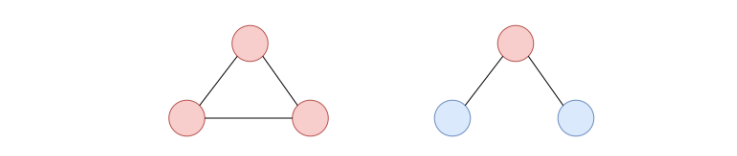
If the nodes form a chain, it is intuitive to see that the above statement holds, which can be broken into the following two cases:

If the number of nodes is even, then there would be two centroids.

If the number of nodes is odd, then there would be only one centroid.



For the rest of cases, we could prove by ***contradiction***. Suppose that we have 3 centroids in the graph, if we remove all the non-centroid nodes in the graph, then the 3 centroids nodes must form a *triangle* shape, as follows:



Because these centroids are equally important to each other, and they should equally close to each other as well. If any of the edges that is missing from the triangle, then the 3 centroids would be reduced down to a single centroid.

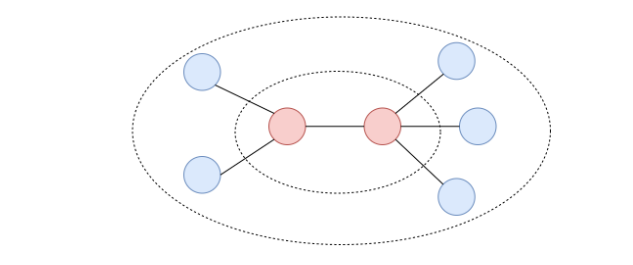
However, the triangle shape forms a *cycle* which is ***contradicted*** to the condition that there is no cycle in our tree-alike graph. Similarly, for any of the cases that have more than 2 centroids, they must form a cycle among the centroids, which is contradicted to our condition.

Therefore, there cannot be more than 2 centroids in a tree-alike graph.

**Algorithm**

Given the above intuition, the problem is now reduced down to looking for all the ***centroid*** nodes in a tree-alike graph, which in addition are no more than two.

The idea is that we *trim* out the leaf nodes layer by layer, until we reach the *core* of the graph, which are the centroids nodes.



Once we trim out the first layer of the leaf nodes (nodes that have only one connection), some of the non-leaf nodes would become leaf nodes.

The trimming process continues until there are only two nodes left in the graph, which are the *centroids* that we are looking for.

The above algorithm resembles the *topological sorting* algorithm which generates the order of objects based on their dependencies. For instance, in the scenario of course scheduling, the courses that have the least dependency would appear first in the order.

In our case, we trim out the leaf nodes first, which are the **farther** away from the centroids. At each step, the nodes we trim out are closer to the centroids than the nodes in the previous step. At the end, the trimming process terminates at the **centroids** nodes.

**Implementation**

Given the above algorithm, we could implement it via the *Breadth First Search* (BFS) strategy, to trim the leaf nodes layer by layer (*i.e.* level by level).

Initially, we would build a graph with the *adjacency list* from the input.

We then create a queue which would be used to hold the leaf nodes.

At the beginning, we put all the current leaf nodes into the queue.

We then run a loop until there is only two nodes left in the graph.

At each iteration, we remove the current leaf nodes from the queue. While removing the nodes, we also remove the edges that are linked to the nodes. As a consequence, some of the non-leaf nodes would become leaf nodes. And these are the nodes that would be trimmed out in the next iteration.

The iteration terminates when there are no more than two nodes left in the graph, which are the desired *centroids* nodes.

Here are some sample implementations that are inspired from the post of [dietpepsi](https://leetcode.com/problems/minimum-height-trees/discuss/76055/Share-some-thoughts) in the discussion forum.

class Solution {

    public List<Integer> findMinHeightTrees(int n, int[][] edges) {

        // edge cases

        if (n < 2) {

            ArrayList<Integer> centroids = new ArrayList<>();

            for (int i = 0; i < n; i++)

                centroids.add(i);

            return centroids;

        }

        // Build the graph with the adjacency list

        ArrayList<Set<Integer>> neighbors = new ArrayList<>();

        for (int i = 0; i < n; i++)

            neighbors.add(new HashSet<Integer>());

        for (int[] edge : edges) {

            Integer start = edge[0], end = edge[1];

            neighbors.get(start).add(end);

            neighbors.get(end).add(start);

        }

        // Initialize the first layer of leaves

        ArrayList<Integer> leaves = new ArrayList<>();

        for (int i = 0; i < n; i++)

            if (neighbors.get(i).size() == 1)

                leaves.add(i);

        // Trim the leaves until reaching the centroids

        int remainingNodes = n;

        while (remainingNodes > 2) {

            remainingNodes -= leaves.size();

            ArrayList<Integer> newLeaves = new ArrayList<>();

            // remove the current leaves along with the edges

            for (Integer leaf : leaves) {

                // the only neighbor left for the leaf node

                Integer neighbor = neighbors.get(leaf).iterator().next();

                // remove the edge along with the leaf node

                neighbors.get(neighbor).remove(leaf);

                if (neighbors.get(neighbor).size() == 1)

                    newLeaves.add(neighbor);

            }

            // prepare for the next round

            leaves = newLeaves;

        }

        // The remaining nodes are the centroids of the graph

        return leaves;

    }

}

**Complexity Analysis**

Let ∣V∣be the number of nodes in the graph, then the number of edges would be ∣V∣−1| as specified in the problem.

Time Complexity: O(∣V∣)

First, it takes ∣V∣−1 iterations for us to construct a graph, given the edges.

With the constructed graph, we retrieve the initial leaf nodes, which takes ∣V∣steps.

During the BFS trimming process, we will trim out *almost* all the nodes (∣V∣) and edges (∣V∣−1) from the edges. Therefore, it would take us around ∣V∣+∣V∣−1 operations to reach the centroids.

To sum up, the overall time complexity of the algorithm is O(∣V∣).

Space Complexity: O(∣V∣)

We construct the graph with adjacency list, which has ∣V∣ nodes and ∣V∣−1 edges. Therefore, we would need ∣V∣+∣V∣−1 space for the representation of the graph.

In addition, we use a queue to keep track of the leaf nodes. In the worst case, the nodes form a star shape, with one centroid and the rest of the nodes as leaf nodes. In this case, we would need ∣V∣−1 space for the queue.

To sum up, the overall space complexity of the algorithm is also O(∣V∣).

Refer to

[L207.P17.2.Course Schedule (Ref.L210,L261)](note://C74CA92E9FAA407FAC5C404EC10B0262)

[L210.P17.3.Course Schedule II (Ref.L207,L261)](note://457E51C3DDB149349C49DC3D661615A9)

[L261.Lint178.Graph Valid Tree (Ref.L841)](note://2DB801CF2BAB4DDE9E74C819757178B5)

[Directed and Undirected Graph Cycle Detection in DFS and BFS](note://8D51E6EB87834766A8F4688C191B70AC)