<https://leetcode.com/problems/node-with-highest-edge-score/description/>

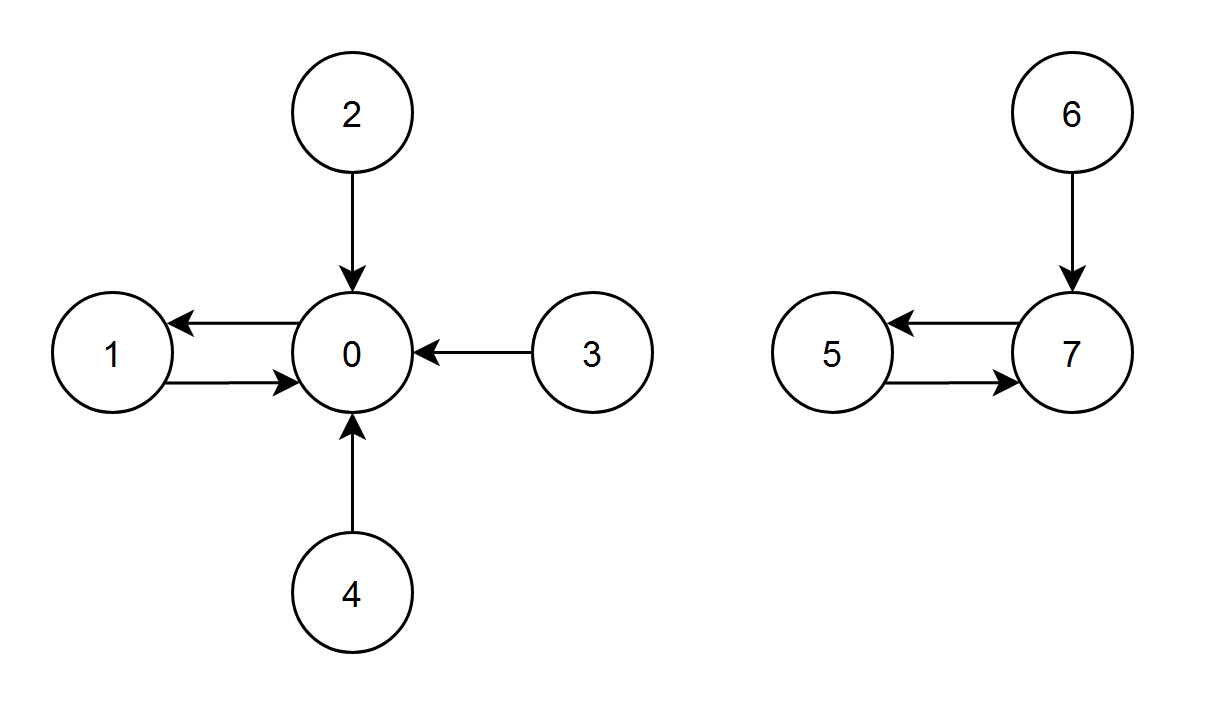
You are given a directed graph with n nodes labeled from 0 to n - 1, where each node has **exactly one** outgoing edge.

The graph is represented by a given **0-indexed** integer array edges of length n, where edges[i] indicates that there is a **directed** edge from node i to node edges[i].

The **edge score** of a node i is defined as the sum of the **labels** of all the nodes that have an edge pointing to i.

Return **the node with the highest edge score**. If multiple nodes have the same **edge score**, return the node with the **smallest** index.

**Example 1:**



**Input:** edges = [1,0,0,0,0,7,7,5]

**Output:** 7

**Explanation:**

- The nodes 1, 2, 3 and 4 have an edge pointing to node 0. The edge score of node 0 is 1 + 2 + 3 + 4 = 10.

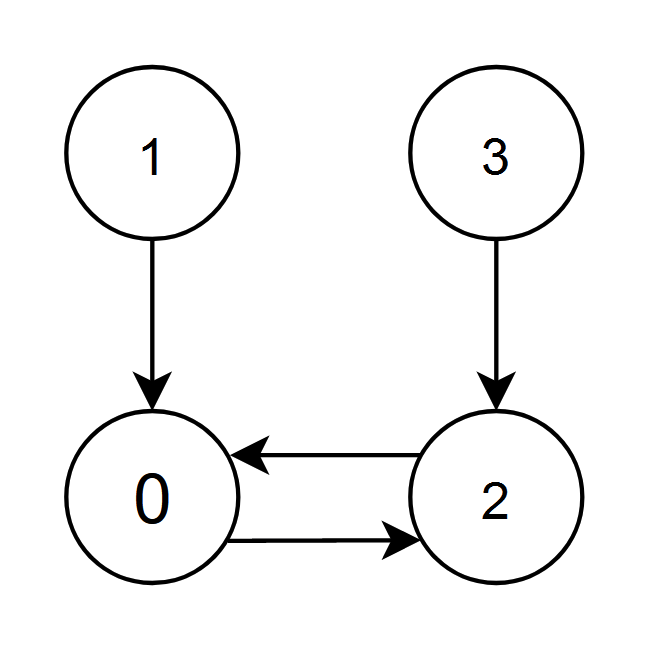
- The node 0 has an edge pointing to node 1. The edge score of node 1 is 0.

- The node 7 has an edge pointing to node 5. The edge score of node 5 is 7.

- The nodes 5 and 6 have an edge pointing to node 7. The edge score of node 7 is 5 + 6 = 11.

Node 7 has the highest edge score so return 7.

**Example 2:**



**Input:** edges = [2,0,0,2]

**Output:** 0

**Explanation:**

- The nodes 1 and 2 have an edge pointing to node 0. The edge score of node 0 is 1 + 2 = 3.

- The nodes 0 and 3 have an edge pointing to node 2. The edge score of node 2 is 0 + 3 = 3.

Nodes 0 and 2 both have an edge score of 3. Since node 0 has a smaller index, we return 0.

**Constraints:**

n == edges.length

2 <= n <= 10^5

0 <= edges[i] < n

edges[i] != i

**Attempt 1: 2024-01-26**

**Solution 1: Hash Table (10 min)**

**Wrong Solution (114/118)**

**Test out by extremely long input [1,0,1,1,1.....,1,1] as 66002 length, 1's 0 and remain 66001's 1, expect 1, output 0**

class Solution {

    public int edgeScore(int[] edges) {

        // For all edges[i] have same value -> find whose index sum is max

        // And use TreeMap to satisfy: If multiple nodes have the same edge

        // score, return the node with the smallest index.

        TreeMap<Integer, Integer> map = new TreeMap<>();

        for(int i = 0; i < edges.length; i++) {

            map.put(edges[i], map.getOrDefault(edges[i], 0) + i);

        }

        int result = -1;

        int max = -1;

        for(Map.Entry<Integer, Integer> e : map.entrySet()) {

            if(e.getValue() > max) {

                max = e.getValue();

                result = e.getKey();

            }

        }

        return result;

    }

}

**Correct Solution**

**The indexes sum will be Long type**

class Solution {

    public int edgeScore(int[] edges) {

        // For all edges[i] have same value -> find whose index sum is max

        // And use TreeMap to satisfy: If multiple nodes have the same edge

        // score, return the node with the smallest index.

        TreeMap<Integer, Long> map = new TreeMap<>();

        for(int i = 0; i < edges.length; i++) {

            map.put(edges[i], map.getOrDefault(edges[i], (long)0) + (long)i);

        }

        int result = -1;

        long max = -1;

        for(Map.Entry<Integer, Long> e : map.entrySet()) {

            if(e.getValue() > max) {

                max = e.getValue();

                result = e.getKey();

            }

        }

        return result;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/2374>

**Problem Description**

In this problem, we're given a directed graph consisting of n nodes. These nodes are uniquely labeled from 0 to n - 1. The special thing about this graph is that each node has exactly one outgoing edge, which means that every node points to exactly one other node. This setup creates a series of chains and possibly cycles within the graph.

The graph is described using an array edges of length n, where the value edges[i] represents the node that node i points to. The problem asks us to compute the "edge score" for each node. The edge score of a node is the sum of the labels of all nodes that point to it.

Our task is to identify the node with the highest edge score. If there happens to be a tie where multiple nodes have the same highest edge score, we should return the node with the smallest index among them.

To summarize, we are to calculate the edge scores, find the maximum, and return the node that obtains it, resolving ties by choosing the lowest-indexed node.

**Intuition**

To solve this problem, our approach involves two main steps:

Calculating Edge Scores:

 To calculate the edge score of each node efficiently, we can traverse the graph once and increment the edge score for the target nodes. For instance, if there is an edge from node i to node j, we add i (the label of the starting node) to the edge score of node j (the destination node). We can keep a count of these edge scores using a data structure such as a Counter from Python's collections module, where the keys correspond to node indices and the values to their edge scores.

Finding the Node with the Highest Edge Score:

 After we have all the edge scores, we iterate through them to find the maximum score. While doing this, we must also keep track of the node index because if multiple nodes have the same edge score, we need to return the one with the smallest index. An efficient way to tackle this is to initialize a variable (let's say ans) to store the index of the node with the current highest edge score. We iterate through all possible node indices, compare their edge scores with the current highest, and update ans when we find a higher score or if the score is the same but the node index is smaller.

This approach ensures we traverse the graph only once to compute the scores and a second time to find the maximum score with the smallest node index—both operations having linear time complexity, which is efficient.

**Solution Approach**

To implement the solution as described in the intuition, we're using a counter and a for-loop construct. Here's the step-by-step breakdown of the implementation:

**Initializing the Counter:** We start by initializing a Counter, which is a special dictionary that lets us keep a tally for each element. It is a part of Python's built-in collections module. In our case, each element corresponds to a node and its tally to the node's edge score.

cnt = Counter()

**Calculating Edge Scores:** We loop over each edge in the edges list with its index. At each step, we increment the edge score of the node pointed to by the current index. The index indicates the starting node (contributing to the edge score) and edges[i] the destination node. The score of the destination node is increased by the label of the starting node (which is its index i).

for i, v in enumerate(edges):

cnt[v] += i

**Finding the Node with the Highest Edge Score:** We initialize a variable ans to keep track of the index of the node with the highest edge score found so far, starting with the first node (0).

Then, we iterate over the range of node indices, using another for-loop. For each index, we compare its edge score (cnt[i]) with the edge score of the current answer (cnt[ans]). If we find a higher edge score, or if the edge scores are equal and the current index is less than ans (implying a smaller node index), we update ans.

ans = 0

for i in range(len(edges)):

if cnt[ans] < cnt[i]:

ans = i

**Return the Result:** Finally, after finishing the iteration, the variable ans holds the index of the node with the highest edge score. We return ans as the final result.

return ans

The combination of Counter to tally the score and a for-loop to determine the maximum ensures a straightforward and efficient implementation. It uses O(n) time for computing the edge scores and O(n) time to identify the node with the highest edge score, leading to an overall linear time complexity, where n is the number of nodes. The space complexity is also O(n) due to the storage needed for the Counter. This solution leverages the characteristics of our graph (each node pointing to exactly one other node) to maintain simplicity and efficiency.

**Example Walkthrough**

Let’s walk through a small example to illustrate the solution approach.

Imagine a graph represented by the edges array: [1, 2, 3, 4, 0]. This array means:

Node 0 points to node 1

Node 1 points to node 2

Node 2 points to node 3

Node 3 points to node 4

Node 4 points to node 0

Now, following the step-by-step solution approach:

**Initializing the Counter:**

cnt = Counter()

**Calculating Edge Scores:**

As per our edges array:

cnt[1] += 0 # node 0 points to node 1

cnt[2] += 1 # node 1 points to node 2

cnt[3] += 2 # node 2 points to node 3

cnt[4] += 3 # node 3 points to node 4

cnt[0] += 4 # node 4 points to node 0

After this loop:

cnt = {1: 0, 2: 1, 3: 2, 4: 3, 0: 4}

**Finding the Node with the Highest Edge Score:**

Initialize ans = 0. Then we compare:

cnt[0] is 4. ans remains 0.

cnt[1] is 0. No change since cnt[0] > cnt[1].

cnt[2] is 1. No change since cnt[0] > cnt[2].

cnt[3] is 2. No change since cnt[0] > cnt[3].

cnt[4] is 3. No change since cnt[0] > cnt[4].

After comparing all, ans is still 0 as it has the highest score 4.

**Return the Result:**

return ans # returns 0

With this example, we can see how the Counter was used to calculate edge scores by aggregating contributions from nodes that point to a specific node. Afterward, we iterated through the node indices, keeping track of the node with the current highest score and returned the one with the smallest index in case of a tie. The node with index 0 has the highest edge score of 4 in this example, so it is the answer.

**Java Solution**

class Solution {

    public int edgeScore(int[] edges) {

        // Get the number of nodes in the graph.

        int numNodes = edges.length;

        // Create an array to keep track of the cumulative edge scores for each node.

        long[] edgeScores = new long[numNodes];

        // Iterate over the array and accumulate edge scores.

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

            // Increment the edge score of the destination node by the index of the current node.

            edgeScores[edges[i]] += i;

        }

        // Initialize 'answer' to the first node's index (0) by default.

        int answer = 0;

        // Iterate over edgeScores to find the node with the highest edge score.

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

            // If the current node's edge score is higher than the score of the answer node,

            // then update the answer to the current node's index.

            if (edgeScores[answer] < edgeScores[i]) {

                answer = i;

            }

        }

        // Return the node with the highest edge score.

        return answer;

    }

}

**Time and Space Complexity**

**Time complexity**

The given code consists primarily of two parts: a loop to accumulate the edge scores and another loop to find the node with the highest edge score. Let's analyze each part to determine the overall time complexity:

The for i, v in enumerate(edges): iterates through the edges list once. The length of this list is n, where n is the number of nodes in the graph. Inside this loop, each iteration performs an O(1) operation, where it updates the Counter object. Therefore, this loop has a time complexity of O(n).

The second loop for i in range(len(edges)): is also iterating n times for each node in the graph. For each iteration, it performs a comparison operation which is O(1). Hence, the time complexity of this loop is also O(n).

Combining both parts, the overall time complexity for the entire function is O(n) + O(n) which simplifies to O(n).

**Space complexity**

The space complexity is determined by the data structures used in the algorithm:

The cnt variable is a Counter object which, in the worst case, will contain an entry for each unique node in edges. This means its size grows linearly with the number of nodes, contributing a space complexity of O(n).

The ans variable is an integer, which occupies O(1) space.

As such, the total space complexity of the algorithm is O(n) for the Counter object plus O(1) for the integer, which results in an overall space complexity of O(n).

**Refer to**

[L451.P14.5.Sort Characters By Frequency (Ref.L347)](note://512A6121D907491D8CF8FDD9119E5CC3)