CloudFront Caching

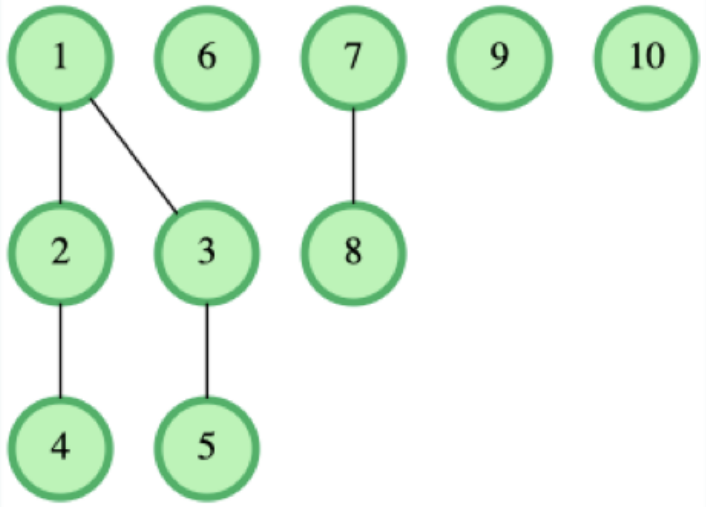
1. Cloudfront Caching

AWS CloudFront wants to build an algorithm to measure the efficiency of its caching network. The network is represented as a number of nodes and a list of connected pairs. The efficiency of this network can be estimated by first summing the cost of each isolated set of nodes where each individual node has a cost of 1. To account for the increase in efficiency as more nodes are connected, update the cost of each isolated set to be the ceiling of the square root of the original cost and return the final sum of all costs.

Example

n = 10 nodes

edges = [[1 2], [1 3], [2 4], [3 5], [7 8]]



There are *2* isolated sets with more than one node, *{1, 2, 3, 4, 5}* and *{7, 8}*. The ceilings of their square roots are $5^{1/2} \approx 2.236$ and *ceil(2.236) = 3*, $2^{1/2} \approx 1.414$ and *ceil(1.414) = 2*. The other three isolated nodes are separate and the square root of their weights is $1^{1/2} = 1$ respectively. The sum is *3 + 2 + (3 * 1) = 8*.

Function Description

Complete the function *connectedSum* in the editor below.

connectedSum has the following parameter(s):

int n: the number of nodes

str edges[m]: an array of strings that consist of a space-separated integer pair that denotes two connected nodes, *p* and *q*

Returns:

int an integer that denotes the sum of the values calculated

思路

Union-find: 先把每个edge上的两个nodes两两union起来.

Code

```python
from typing import List
import math


def connectedSum(n: int, edges: List[str]) -> int:
    # n + 1: 因为是从1开始计数的.
    rank_dic = [1 for _ in range(n + 1)]
    parent = [i for i in range(n + 1)]

    def find(child) -> int:
        while parent[child] != child:
            parent[child] = parent[parent[child]]
            child = parent[child]
        return child

    def union(node1: int, node2: int) -> None:
        root_node1 = find(node1)
        root_node2 = find(node2)

        if root_node1 == root_node2:
            return False
        else:
            root1_rank = rank_dic[root_node1]
            root2_rank = rank_dic[root_node2]
            if root1_rank > root2_rank:
                parent[root_node2] = root_node1
                rank_dic[root_node1] += root2_rank
            elif root1_rank < root2_rank:
                parent[root_node1] = root_node2
                rank_dic[root_node2] += root1_rank
            else:
                parent[root_node2] = root_node1
```

```
                rank_dic[root_node1] += root2_rank
            return True

        for edge in edges:
            tmp = edge.split(" ")
            node1 = int(tmp[0])
            node2 = int(tmp[1])
            union(node1, node2)
        res = 0
        for index in range(len(parent)):
            cur_parent = parent[index]
            cur_rank = rank_dic[index]
            if cur_parent == index:
                res += math.ceil(math.sqrt(cur_rank))
        return res - 1
```