1101. The Earliest Moment When Everyone Become Friends

Leetcode Link

There are n people in a social group labeled from 0 to $n - 1$. You are given an array `logs` where `logs[i]` = $[\text{timestamp}_i, x_i, y_i]$ indicates that x_i and y_i will be friends at the time timestamp_i.

Friendship is **symmetric**. That means if a is friends with b, then b is friends with a. Also, person a is acquainted with a person b if a is friends with b, or a is a friend of someone acquainted with b.

Return the earliest time for which every person became acquainted with every other person. If there is no such earliest time, return -1.

Example 1:

Input: `logs = [[20190101,0,1],[20190104,3,4],[20190107,2,3],[20190211,1,5],[20190224,2,4],[20190301,0,3],[20190312,1,2],[20190322,4,5]], n = 6`
Output: 20190301
Explanation:
The first event occurs at `timestamp = 20190101` and after 0 and 1 become friends we have the following friendship groups `[0,1], [2], [3], [4], [5]`. The second event occurs at `timestamp = 20190104` and after 3 and 4 become friends we have the following friendship groups `[0,1], [2], [3,4], [5]`. The third event occurs at `timestamp = 20190107` and after 2 and 3 become friends we have the following friendship groups `[0,1], [2,3,4], [5]`. The fourth event occurs at `timestamp = 20190211` and after 1 and 5 become friends we have the following friendship groups `[0,1,5], [2,3,4]`. The fifth event occurs at `timestamp = 20190224` and as 2 and 4 are already friends anything happens. The sixth event occurs at `timestamp = 20190301` and after 0 and 3 become friends we have that all become friends.

Example 2:

Input: `logs = [[0,2,0],[1,0,1],[3,0,3],[4,1,2],[7,3,1]],n = 4`
Output: 3

Constraints:

- $2 \leq n \leq 100$
- $1 \leq$ `logs.length` $\leq 10^4$
- `logs[i].length == 3`
- $0 \leq \text{timestamp}_i \leq 10^9$
- $0 \leq x_i, y_i \leq n - 1$
- $x_i \neq y_i$
- All the values timestamp_i are unique.
- All the pairs (x_i, y_i) occur at most one time in the input.

Solution

Brute Force

When we see problems related to connectivity, we should think of applying DSU. This problem asks us to find the first instance where the graph formed by friendships is connected. To accomplish this, we'll first sort the friendships by timestamp_j. Then, we'll iterate through friendships from the least to greatest timestamp_j and merge nodes connected by a friendship until the graph is connected.

After each iteration, we'll check if the graph is connected and return the timestamp value if it is. An easy way to check if the graph is connected is to check if node 1 is connected to all other $n - 1$ nodes. If the graph isn't connected after processing all the friendships, we'll return -1.

Let's denote the number of friendships(edges) as M.

Since checking the connectivity of the graph is $\mathcal{O}(N \log N)$ and we do this $\mathcal{O}(M)$ times, this solution runs in $\mathcal{O}(MN \log N)$.

Full Solution

Since checking the connectivity of the graph is too inefficient, we'll maintain the number of components in the graph as we include more and more friendships. We can make the observation that every time we merge two disjoint sets, the number of components decreases by 1. This is true as this operation turns 2 disjoint sets into 1 disjoint set without disturbing any other disjoint sets. We initially start with N components (one for each person) and once we reach one component, we'll return the respective timestamp value. -1 will be returned if we never reach one component.

Time Complexity

Sorting takes $\mathcal{O}(M \log M)$ and the main algorithm runs in $\mathcal{O}(M \log N)$. Thus our time complexity is $\mathcal{O}(M \log M + M \log N)$.

Time Complexity: $\mathcal{O}(M \log M + M \log N)$

Bonus: We can also use union by rank mentioned here to improve the time complexity of DSU operations from $\mathcal{O}(\log N)$ to $\mathcal{O}(\alpha(N))$.

Space Complexity

Our DSU uses $\mathcal{O}(N)$ memory.

Space Complexity: $\mathcal{O}(N)$.

C++ Solution

```cpp
class Solution {
  public:
    vector<int> parent;
    int find(int x) {  // finds the id/leader of a node
        if (parent[x] == x) {
            return x;
        }
        parent[x] = find(parent[x]);
        return parent[x];
    }
    void Union(int x, int y) {  // merges two disjoint sets into one set
        x = find(x);
        y = find(y);
        parent[x] = y;
    }
    static bool comp(vector<int>& a, vector<int>& b) {  // sorting comparator
        return a[0] < b[0];
    }
    int earliestAcq(vector<vector<int>>& logs, int n) {
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        sort(logs.begin(), logs.end(), comp);  // sorts friendships by timestamp
        int components = n;
        for (vector<int> friendship : logs) {
            int timestamp = friendship[0];
            int x = friendship[1];
            int y = friendship[2];
            if (find(x) != find(y)) {  // merge two disjoint sets
                Union(x, y);
                components--;
            }
            if (components == 1) {  // reached connected graph
                return timestamp;
            }
        }
        return -1;
    }
};
```

Java Solution

```java
class Solution {
    private int find(int x, int[] parent) { // finds the id/leader of a node
        if (parent[x] == x) {
            return x;
        }
        parent[x] = find(parent[x], parent);
        return parent[x];
    }
    private void Union(int x, int y, int[] parent) { // merges two disjoint sets into one set
        x = find(x, parent);
        y = find(y, parent);
        parent[x] = y;
    }

    public int earliestAcq(int[][] logs, int n) {
        int[] parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        Arrays.sort(logs, (a, b) -> a[0] - b[0]); // sorts friendships by timestamp
        int components = n;
        for (int[] friendship : logs) {
            int timestamp = friendship[0];
            int x = friendship[1];
            int y = friendship[2];
            if (find(x, parent) != find(y, parent)) { // merge two disjoint sets
                Union(x, y, parent);
                components--;
            }
            if (components == 1) { // reached connected graph
                return timestamp;
            }
        }
        return -1;
    }
}
```

Python Solution

```python
class Solution:
    def earliestAcq(self, logs: List[List[int]], n: int) -> int:
        parent = [i for i in range(n)]

        def find(x):  # finds the id/leader of a node
            if parent[x] == x:
                return x
            parent[x] = find(parent[x])
            return parent[x]

        def Union(x, y):  # merges two disjoint sets into one set
            x = find(x)
            y = find(y)
            parent[x] = y

        logs.sort(key=lambda x: x[0])  # sorts friendships by timestamp
        components = n
        for friendship in logs:
            timestamp = friendship[0]
            x = friendship[1]
            y = friendship[2]
            if find(x) != find(y):  # merge two disjoint sets
                Union(x, y)
                components -= 1
            if components == 1:  # reached connected graph
                return timestamp
        return -1

```

Got a question? Ask the Teaching Assistant anything you don't understand.