# Daily Coding Problem #279

## Problem

This problem was asked by Twitter.

A classroom consists of `N` students, whose friendships can be represented in an adjacency list. For example, the following descibes a situation where `0` is friends with `1` and `2`, `3` is friends with `6`, and so on.

```
{0: [1, 2],
 1: [0, 5],
 2: [0],
 3: [6],
 4: [],
 5: [1],
 6: [3]}
```

Each student can be placed in a friend group, which can be defined as the transitive closure of that student's friendship relations. In other words, this is the smallest set such that no student in the group has any friends outside this group. For the example above, the friend groups would be `{0, 1, 2, 5}, {3, 6}, {4}`.

Given a friendship list such as the one above, determine the number of friend groups in the class.

# Solution

This problem is a classic motivating example for the use of a disjoint-set data structure.

To implement this data structure, we must create two main methods: `union` and `find`. Initially, each student will be in a friend group consisting of only him- or herself. For each friendship in our input, we will call our `union` method to place the two students in the same set. To perform this, we must call `find` to discover which friend group each student is in, and, if they are not the same, assign one student to the friend group of the other.

It may be the case that, after a few `union` calls, friend `N` may be placed in set `N - 1`, friend `N - 1` may be placed in set `N - 2`, and so on. As a result, our `find` operation must follow the chain of friend sets until reaching a student who has not been reassigned, which will properly identify the group.

Because the chain we must follow for each `find` call can be `N` students long, both methods run in `O(N)` time. However, with just two minor changes we can cut this runtime down to `O(1)` (technically, `O(α(N))`, where α is the inverse Ackermann function).

The first is called path compression. After we call `find`, we know exactly which group the student belongs to, so we can reassign this student directly to that group. Second, when we unite two students, instead of assigning based on value, we can always assign the student belonging to the smaller set to the larger set. Taken together, these optimizations drastically cut down the time it takes to perform both operations.

```
class DisjointSet:
    def __init__(self, n):
        self.sets = list(range(n))
        self.sizes = [1] * n
        self.count = n


    def union(self, x, y):
        x, y = self.find(x), self.find(y)
        if x != y:
            if self.sizes[x] < self.sizes[y]: # union by size
                x, y = y, x
            self.sets[y] = x
            self.sizes[x] += self.sizes[y]
            self.count -= 1
```

```
    def find(self, x):
        group = self.sets[x]

        while group != self.sets[group]:
            group = self.sets[group]
        self.sets[x] = group # path compression

        return group
```

With this data structure in place, our solution will be to go through the list of friendships, calling `union` on each of them. Each time we reassign a student to a different group, we decrement a counter for the number of friend groups, which starts at `N`. Finally, we return the value of this counter.

```
def friend_groups(students):
    groups = DisjointSet(len(students))

    for student, friends in students.items():
        for friend in friends:
            groups.union(student, friend)

    return groups.count
```

Since `union` and `find` operations are both `O(1)`, the time complexity of this solution is `O(E)`, where `E` is the number of edges represented in the adjacency list. We will also use `O(N)` space to store the list of assigned friend groups.