01. 并查集知识

ITCharge 大约 25 分钟

1. 并查集简介

1.1 并查集的定义

> **并查集（Union Find）**：一种树型的数据结构，用于处理一些不交集（Disjoint Sets）的合并及查询问题。不交集指的是一系列没有重复元素的集合。
> 并查集主要支持两种操作：
>
> - **合并（Union）**：将两个集合合并成一个集合。
> - **查找（Find）**：确定某个元素属于哪个集合。通常是返回集合内的一个「代表元素」。

简单来说，并查集就是用来处理集合的合并和集合的查询。

- 并查集中的「集」指的就是我们初[illegible]学的集合概念，在这里指的是不相交的集合，即一系列没有重复元素的集合。
- 并查集中的「并」指的就是集合的并集操作，将两个集合合并之后就变成一个集合。合并操作如下所示：

```py
{1, 3, 5, 7} U {2, 4, 6, 8} = {1, 2, 3, 4, 5, 6, 7, 8}
```

- 并查集中的「查」是对于集合中存放的元素来说的，通常我们需要查询两个元素是否属于同一个集合。

如果我们只是想知道一个元素是否在集合中，可以通过 Python 或其他语言中的 `set` 集合来解决。而如果我们想知道两个元素是否属于同一个集合，则仅用一个 `set` 集合就很难做到了。这就需要用到我们接下来要讲解的「并查集」结构。

根据上文描述，我们就可以定义一下「并查集」结构所支持的操作接口：

- **合并** `union(x, y)`：将集合 x 和集合 y 合并成一个集合。
- **查找** `find(x)`：查找元素 x 属于哪个集合。
- **查找** `is_connected(x, y)`：查询元素 x 和 y 是否在同一个集合中。

1.2 并查集的两种实现思路

下面我们来讲解一下并查集的两种实现思路：一种是使用「快速查询」思路、基于数组结构实现的并查集；另一种是使用「快速合并」思路、基于森林实现的并查集。

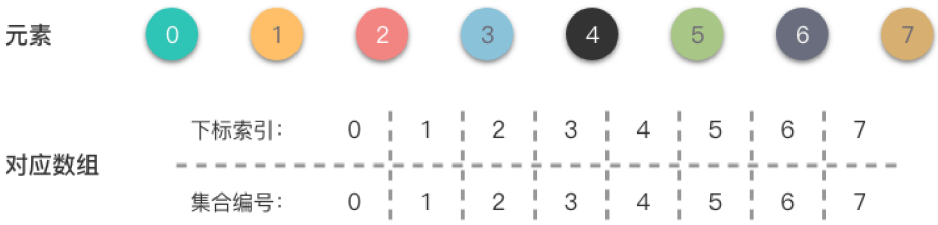
1.2.1 快速查询：基于数组实现

如果我们希望并查集的查询效率高一些，那么我们就可以侧重于查询操作。

在使用「快速查询」思路实现并查集时，我们可以使用一个「数组结构」来表示集合中的元素。数组元素和集合元素是一一对应的，我们可以将数组的索引值作为每个元素的集合编号，称为 id。然后可以对数组进行以下操作来实现并查集：

- **当初始化时**：将数组下标索引值作为每个元素的集合编号。所有元素的 id 都是唯一的，代表着每个元素单独属于一个集合。
- **合并操作时**：需要将其中一个集合中的所有元素 id 更改为另一个集合中的 id，这样能够保证在合并后一个集合中所有元素的 id 均相同。
- **查找操作时**：如果两个元素的 id 一样，则说明它们属于同一个集合；如果两个元素的 id 不一样，则说明它们不属于同一个 。

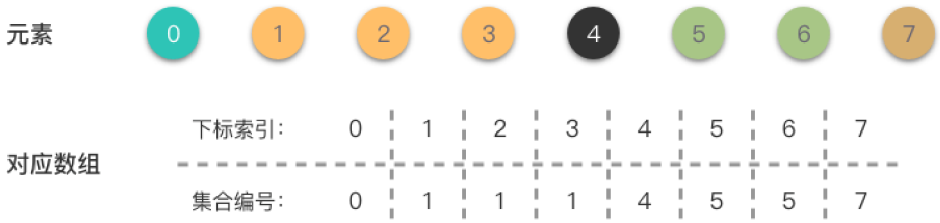
举个例子来说明一下，我们使用数组来表示一系列集合元素 $\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\}$，初始化时如下图所示。



基于数组实现：初始化操作

从上图中可以看出：数组的每个下标索引值对应一个元素的集合编号，代表着每个元素单独属于一个集合。

当我们进行一系列的合并操作后，比如合并后变为 $\{0\},\{1,2,3\},\{4\},\{5,6\},\{7\}$，合并操作的结果如下图所示。



基于数组实现：合并操作

从上图中可以看出，在进行一系列合并操作后，下标为 1、2、3 的元素集合编号是一致的，说明这 3 个元素同属于一个集合。同理下标为 5 和 6 的元素则同属于另一个集合。

在快速查询的实现思路中，单次查询操作的时间复杂度是 $O(1)$，而单次合并操作的时间复杂度为 $O(n)$（每次合并操作需要遍历数组）。两者的时间复杂度相差得比较大，完全牺牲了合并操作的性能。因此，这种并查集的实现思路并不常用。

- 使用「快速查询」思路实现并查集代码如下所示：

```py
class UnionFind:
    def __init__(self, n):                          # 初始化：将每个元素的集合编号初始化为数组下标索引
        self.ids = [i for i in range(n)]

    def find(self, x):                              # 查找元素所属集合编号内部实现方法
        return self.ids[x]

    def union(self, x, y):                          # 合并操作：将集合 x 和集合 y 合并成一个集合
        x_id = self.find(x)
        y_id = self.find(y)

        if x_id == y_id:                            # x 和 y 已经同属于一个集合
            return False

        for i in range(len(self.ids)):              # 将两个集合的集合编号改为一致
            if self.ids[i] == y_id:
                self.ids[i] = x_id
```

```
        return True

    def is_connected(self, x, y):                    # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)
```

1.2.2 快速合并：基于森林实现

因为快速查询的实现思路中，合并操作的效率比较低。所以我们现在的重点是提高合并操作的效率。

在使用「快速合并」思路实现并查集时，我们可以使用「一个森林（若干棵树）」来存储所有集合。每一棵树代表一个集合，树上的每个节点都是一个元素，树根节点为这个集合的代表元素。

> **注意**：与普通的树形结构（父节点指向子节点）不同的是，基于森林实现的并查集中，树中的子节点是指向父节点的。

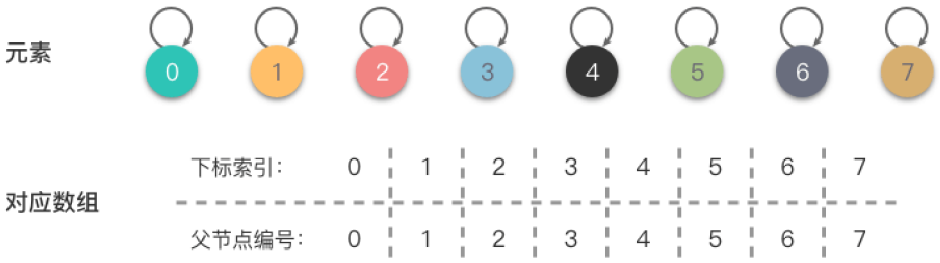
此时，我们仍然可以使用一个数组 fa 来记录这个森林。我们用 $fa[x]$ 来保存 x 的父节点的集合编号，代表着元素节点 x 指向父节点 $fa[x]$。

当初始化时，$fa[x]$ 值赋值为下标索引 在进行合并操作时，只需要将两个元素的树根节点相连接（ `fa[root1] = root2` ）即可。而在进行查询操作时，只需要查看两个元素的树根节点是否一致，就能知道两个元素是否属于同一个集合。

总结一下，我们可以对数组 fa 进行以下操作来实现并查集：

- **当初始化时**：将数组 fa 的下标索引作为每个元素的集合编号。所有元素的根节点的集合编号都不一样，代表着每个元素单独属于一个集合。
- **合并操作时**：需要将两个集合的树根节点相连接。即令其中一个集合的树根节点指向另一个集合的树根节点（ `fa[root1] = root2` ），这样合并后当前集合中的所有元素的树根节点均为同一个。
- **查找操作时**：分别从两个元素开始，通过数组 fa 存储的值，不断递归访问元素的父节点，直到到达树根节点。如果两个元素的树根节点一样，则说明它们属于同一个集合；如果两个元素的树根节点不一样，则说明它们不属于同一个集合。

举个例子来说明一下，我们使用数组来表示一系列集合元素 $\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\}$，初始化时如下图所示。



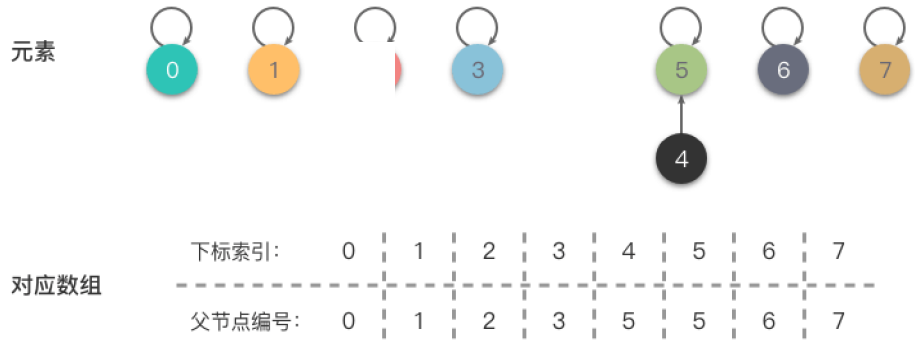
基于森林实现：初始化操作

从上图中可以看出：fa 数组的每个下标索引值对应一个元素的集合编号，代表着每个元素属于一个集合。

当我们进行一系列的合并操作后，比如 `union(4, 5)` 、 `union(6, 7)` 、 `union(4, 7)` 操作后变为 $\{0\},\{1\},\{2\},\{3\},\{4,5,6,7\}$，合并操作的步骤及结果如下图所示。

<1>

- 合并 $(4,5)$：令 4 的根节点指向 5，即将 $fa[4]$ 更改为 5。



基于森林实现：合并操作 1

<2>

- 合并 $(6,7)$：令 6 的根节点指向 7，即将 $fa[6]$ 更改为 7。

基于森林实现：合并操作 2

<3>

- 合并 $(4,7)$：令 4 的的根节点指向 7，即将 $fa[fa[4]]$（也就是 $fa[5]$）更改为 7。

基于森林实现：合并操作 3

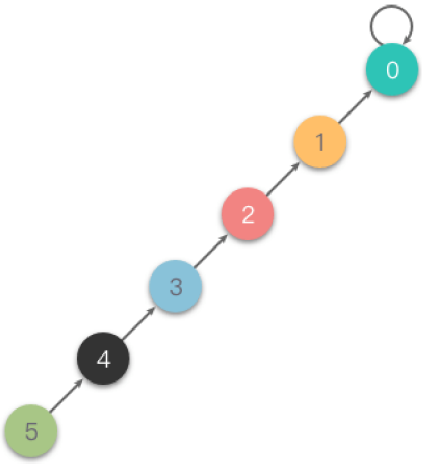
从上图中可以看出，在进行一系列合并操作后， `fa[fa[4]] == fa[5] == fa[6] == f[7]` ，即 4、5、6、7 的元素根节点编号都是 4，说明这 4 个元素同属于一个集合。

- 使用「快速合并」思路实现并查集代码如下所示：

```py
class UnionFind:
    def __init__(self, n):                          # 初始化：将每个元素的集合编号初始化为数组 fa 的下标索引
        self.fa = [i for i in range(n)]

    def find(self, x):                              # 查找元素根节点的集合编号内部实现方法
        while self.fa[x] != x:                      # 递归查找元素的父节点，直到根节点
            x = self.fa[x]
        return x                                    # 返回元素根节点的集合编号

    def union(self, x, y):                          # 合并操作：令其中一个集合的树根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                        # x 和 y 的根节点集合编号相同，说明 x 和 y 已经同属于一个集合
            return False
        self.fa[root_x] = root_y                    # x 的根节点连接到 y 的根节点上，成为 y 的根节点的子节点
        return True

    def is_connected(self, x, y):                   # 查询操作：判断 x 和 y 是否同属于一个集合
        return self.find(x) == self.find(y)
```

2. 路径压缩

在集合很大或者树很不平衡时，使用上述「快速合并」思路实现并查集的代码效率很差，最坏情况下，树会退化成一条链，单次查询的时间复杂度高达 $O(n)$。并查集的最坏情况如下图所示。



并查集最坏情况

为了避免出现最坏情况，一个常见的优化方式是「路径压缩」。

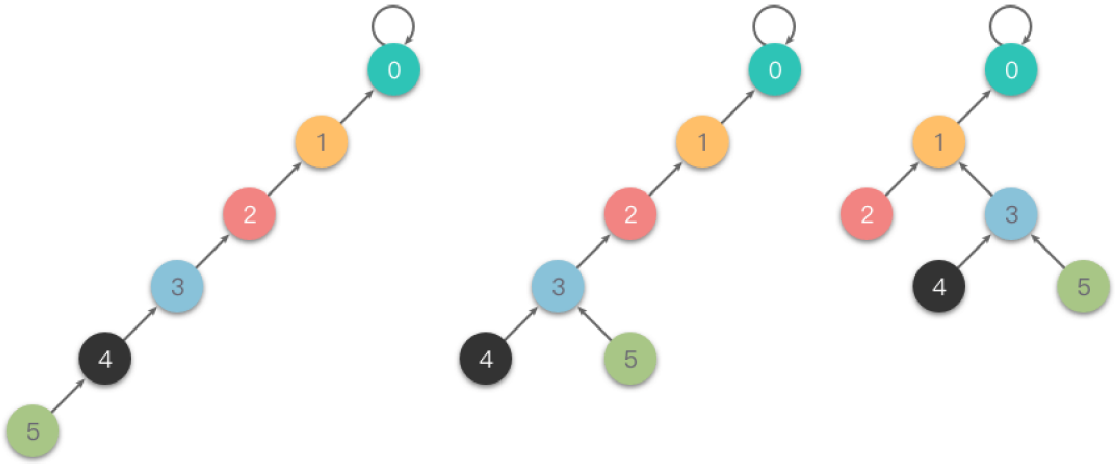
> **路径压缩（Path Compression）**：在从底向上查找根节点过程中，如果此时访问的节点不是根节点，则我们可以把这个节点尽量向上移动一下，从而减少树的层树。这个过程就叫做路径压缩。

路径压缩有两种方式：一种叫做「隔代压缩」；另一种叫做「完全压缩」。

2.1 隔代压缩

> **隔代压缩**：在查询时，两步一压缩，一直循环执行「把当前节点指向它的父亲节点的父亲节点」这样的操作，从而减小树的深度。

下面是一个「隔代压缩」的例子。



路径压缩：隔代压缩

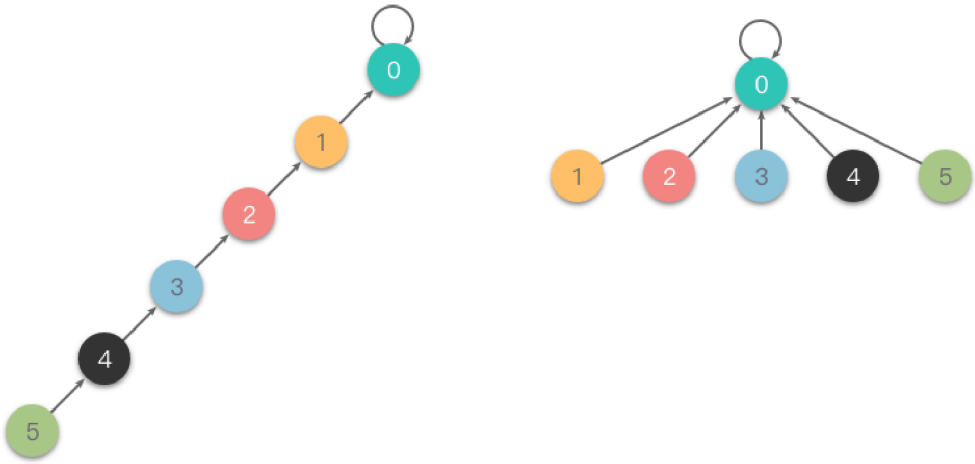
- 隔代压缩的查找代码如下：

```py
def find(self, x):                          # 查找元素根节点的集合编号内部实现方法
    while self.fa[x] != x:                  # 递归查找元素的父节点，直到根节点
        self.fa[x] = self.fa[self.fa[x]]    # 隔代压缩
        x = self.fa[x]
    return x                                # 返回元素根节点的集合编号
```

2.2 完全压缩

> **完全压缩**：在查询时，把被查询的节点到根节点的路径上的所有节点的父节点设置为根节点，从而减小树的深度。也就是说，在向上查询的同时，把在路径上的每个节点都直接连接到根上，以后查询时就能直接查询到根节点。

相比较于「隔代压缩」，「完全压缩」压缩的更加彻底。下面是一个「完全压缩」的例子。



路径压缩：完全压缩

- 完全压缩的查找代码如下：

```py
def find(self, x):                          # 查找元素根节点的集合编号内部实现方法
    if self.fa[x] != x:                     # 递归查找元素的父节点，直到根节点
        self.fa[x] = self.find(self.fa[x])  # 完全压缩优化
    return self.fa[x]
```

3. 按秩合并

因为路径压缩只在查询时进行，并且只压缩一棵树上的路径，所以并查集最终的结构仍然可能是比较复杂的。为了避免这种情况，另一个优化方式是「按秩合并」。

> **按秩合并（Union By Rank）**：指的是在每次合并操作时，都把「秩」较小的树根节点指向「秩」较大的树根节点。

这里的「秩」有两种定义，一种定义指的是树的深度；另一种定义指的是树的大小（即集合节点个数）。无论采用哪种定义，集合的秩都记录在树的根节点上。

按秩合并也有两种方式：一种叫做「按深度合并」；另一种叫做「按大小合并」。

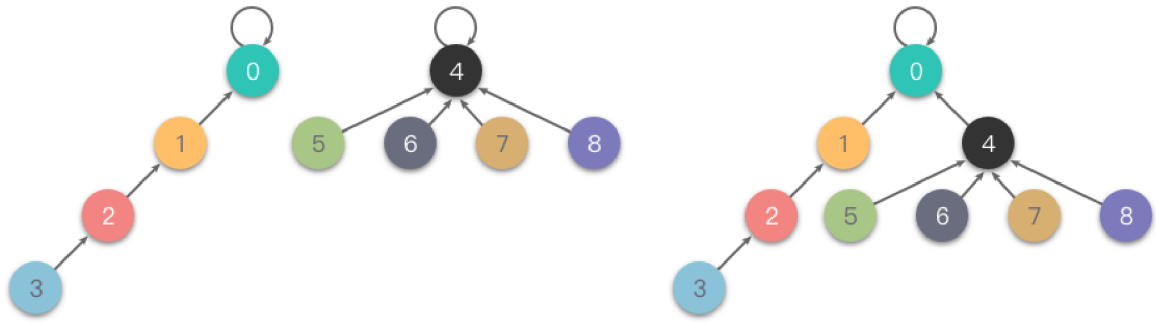
3.1 按深度合并

> **按深度合并（Unoin By Rank）**：在每次合并操作时，都把「深度」较小的树根节点指向「深度」较大的树根节点。

我们用一个数组 $rank$ 记录每个根节点对应的树的深度（如果不是根节点，其 $rank$ 值相当于以它作为根节点的子树的深度）。

初始化时，将所有元素的 $rank$ 值设为 1。在合并操作时，比较两个根节点，把 $rank$ 值较小的根节点指向 $rank$ 值较大的根节点上合并。

下面是一个「按深度合并」的例子。



按秩合并：按深度合并

- 按深度合并的实现代码如下：

```py
class UnionFind:
    def __init__(self, n):                          # 初始化
        self.fa = [i for i in range(n)]             # 每个元素的集合编号初始化为数组 fa 的下标索引
        self.rank = [1 for i in range(n)]           # 每个元素的深度初始化为 1

    def find(self, x):                              # 查找元素根节点的集合编号内部实现方法
        while self.fa[x] != x:                      # 递归查找元素的父节点，直到根节点
            self.fa[x] = self.fa[self.fa[x]]        # 隔代压缩
            x = self.fa[x]
        return x                                    # 返回元素根节点的集合编号
```

```
    def union(self, x, y):                          # 合并操作：令其中一个集合的树
根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                        # x 和 y 的根节点集合编号相
同，说明 x 和 y 已经同属于一个集合
            return False

        if self.rank[root_x] < self.rank[root_y]:   # x 的根节点对应的树的深度 小
于 y 的根节点对应的树的深度
            self.fa[root_x] = root_y                # x 的根节点连接到 y 的根节点
上，成为 y 的根节点的子节点
        elif self.rank[root_y] > self.rank[root_y]: # x 的根节点对应的树的深度 大
于 y 的根节点对应的树的深度
            self.fa[root_y] = root_x                # y 的根节点连接到 x 的根节点
上，成为 x 的根节点的子节点
        else:                                       # x 的根节点对应的树的深度 等
于 y 的根节点对应的树的深度
            self.fa[root_x] = root_y                # 向任意一方合并即可
            self.rank[root_y] += 1                  # 因为层数相同，被合并的树必然
层数会 +1
        return True

    def is_connected(self, x, y):                   # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)
```

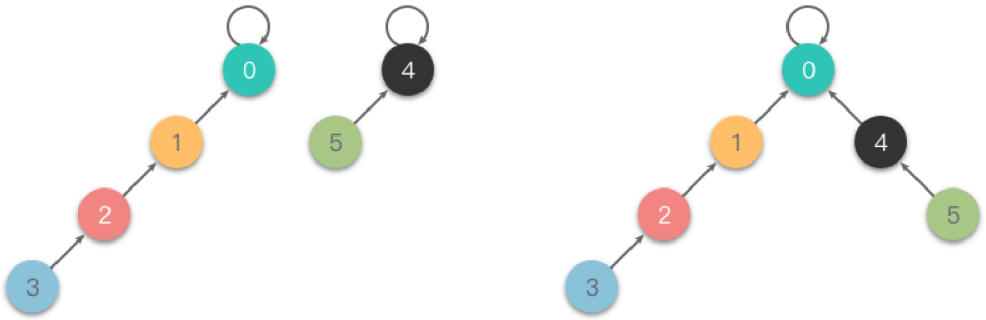
3.2 按大小合并

> **按大小合并（Unoin By Size）**：这里的大小指的是集合节点个数。在每次合并操作时，都把「集合节点个数」较少的树根节点指向「集合节点个数」较大的树根节点。

我们用一个数组 $size$ 记录每个根节点对应的集合节点个数（如果不是根节点，其 $size$ 值相当于以它作为根节点的子树的集合节点个数）。

初始化时，将所有元素的 $size$ 值设为 1。在合并操作时，比较两个根节点，把 $size$ 值较小的根节点指向 $size$ 值较大的根节点上合并。

下面是一个「按大小合并」的例子。



按秩合并：按大小合并

- 按大小合并的实现代码如下：

```py
class UnionFind:
    def __init__(self, n):                          # 初始化
        self.fa = [i for i in range(n)]             # 每个元素的集合编号初始化为数组 fa 的下标索引
        self.size = [1 for i in range(n)]           # 每个元素的集合个数初始化为 1

    def find(self, x):                              # 查找元素根节点的集合编号内部实现方法
        while self.fa[x] != x:                      # 递归查找元素的父节点，直到根节点
            self.fa[x] = self.fa[self.fa[x]]        # 隔代压缩优化
            x = self.fa[x]
        return x                                    # 返回元素根节点的集合编号

    def union(self, x, y):                          # 合并操作：令其中一个集合的树根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                        # x 和 y 的根节点集合编号相同，说明 x 和 y 已经同属于一个集合
            return False

        if self.size[root_x] < self.size[root_y]:   # x 对应的集合元素个数 小于 y 对应的集合元素个数
            self.fa[root_x] = root_y                # x 的根节点连接到 y 的根节点上，成为 y 的根节点的子节点
            self.size[root_y] += self.size[root_x]  # y 的根节点对应的集合元素个数累加上 x 的根节点对应的集合元素个数
        elif self.size[root_x] > self.size[root_y]: # x 对应的集合元素个数 大于 y 对应的集合元素个数
```

```
            self.fa[root_y] = root_x                # y 的根节点连接到 x 的根节点
上，成为 x 的根节点的子节点
            self.size[root_x] += self.size[root_y]  # x 的根节点对应的集合元素个数
累加上 y 的根节点对应的集合元素个数
        else:                                       # x 对应的集合元素个数 小于 y
对应的集合元素个数
            self.fa[root_x] = root_y                # 向任意一方合并即可
            self.size[root_y] += self.size[root_x]

        return True

    def is_connected(self, x, y):                   # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)
```

3.3 按秩合并的注意点

看过「按深度合并」和「按大小合并」的实现代码后，大家可能会产生一个疑问：为什么在路径压缩的过程中不用更新 $rank$ 值或者 $size$ 值呢？

其实，代码中的 $rank$ 值或者 $size$ 值并不完全是树中真实的深度或者集合元素个数。

这是因为当我们在代码中引入路径压　　后，维护真实的深度或者集合元素个数就会变得比较难。此时我们使用的 $rank$ 值或者 $size$ 值更像是用于当前节点排名的一个标志数字，只在合并操作的过程中，用于比较两棵树的权值大小。

换句话说，我们完全可以不知道每个节点的具体深度或者集合元素个数，只要能够保证每两个节点之间的深度或者集合元素个数关系可以通过 $rank$ 值或者 $size$ 值正确的表达即可。

而根据路径压缩的过程，$rank$ 值或者 $size$ 值只会不断的升高，而不可能降低到比原先深度更小的节点或者集合元素个数更少的节点还要小。所以，$rank$ 值或者 $size$ 值足够用于比较两个节点的权值，进而选择合适的方式进行合并操作。

4. 并查集的算法分析

首先我们来分析一下并查集的空间复杂度。在代码中，我们主要使用了数组 fa 来存储集合中的元素。如果使用了「按秩合并」的优化方式，还会使用数组 $rank$ 或者数组 $size$ 来存放权值。因为空间复杂度取决于元素个数，不难得出空间复杂度为 $O(n)$。

在同时使用了「路径压缩」和「按秩合并」的情况下，并查集的合并操作和查找操作的时间复杂度可以接近于 $O(1)$。最坏情况下的时间复杂度是 $O(m \times \alpha(n))$。这里的 m 是合并操作和查找操作的次数，$\alpha(n)$ 是 Ackerman 函数的某个反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数。

总结一下：

- 并查集的空间复杂度：$O(n)$。
- 并查集的时间复杂度：$O(m \times \alpha(n))$。

5. 并查集的最终实现代码

根据我自己的做题经验和网上大佬的经验，我使用并查集的策略（仅供参考）是这样：使用「隔代压缩」，一般不使用「按秩合并」。

这样选择的原因是既能保证代码简单易写，又能得到不错的性能。如果这样写的性能还不够好的话，再考虑使用「按秩合并」。

在有些题目中，还会遇到需要查询集合的个数或者集合中元素个数的情况，可以根据题目具体要求再做相应的更改。

- 使用「隔代压缩」，不使用「按秩 ˆ ``:」的并查集最终实现代码：

```py
class UnionFind:
    def __init__(self, n):                          # 初始化
        self.fa = [i for i in range(n)]             # 每个元素的集合编号初始化为数组 fa 的下标索引

    def find(self, x):                              # 查找元素根节点的集合编号内部实现方法
        while self.fa[x] != x:                      # 递归查找元素的父节点，直到根节点
            self.fa[x] = self.fa[self.fa[x]]        # 隔代压缩优化
            x = self.fa[x]
        return x                                    # 返回元素根节点的集合编号

    def union(self, x, y):                          # 合并操作：令其中一个集合的树根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                        # x 和 y 的根节点集合编号相同，说明 x 和 y 已经同属于一个集合
```

```
            return False

        self.fa[root_x] = root_y                    # x 的根节点连接到 y 的根节点
上，成为 y 的根节点的子节点
        return True

    def is_connected(self, x, y):                   # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)
```

- 使用「隔代压缩」，使用「按秩合并」的并查集最终实现代码：

```py
class UnionFind:
    def __init__(self, n):                          # 初始化
        self.fa = [i for i in range(n)]             # 每个元素的集合编号初始化为数
组 fa 的下标索引
        self.rank = [1 for i in range(n)]           # 每个元素的深度初始化为 1

    def find(self, x):                              # 查找元素根节点的集合编号内部
实现方法
        while self.fa[x] != x:                      # 递归查找元素的父节点，直到根
节点
            self.fa[x] = self.fa    f.fa[x]]        # 隔代压缩优化
            x = self.fa[x]
        return x                                    # 返回元素根节点的集合编号

    def union(self, x, y):                          # 合并操作：令其中一个集合的树
根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                        # x 和 y 的根节点集合编号相
同，说明 x 和 y 已经同属于一个集合
            return False

        if self.rank[root_x] < self.rank[root_y]:   # x 的根节点对应的树的深度 小
于 y 的根节点对应的树的深度
            self.fa[root_x] = root_y                # x 的根节点连接到 y 的根节点
上，成为 y 的根节点的子节点
        elif self.rank[root_x] > self.rank[root_y]: # x 的根节点对应的树的深度 大
于 y 的根节点对应的树的深度
            self.fa[root_y] = root_x                # y 的根节点连接到 x 的根节点
上，成为 x 的根节点的子节点
        else:                                       # x 的根节点对应的树的深度 等
```

```
于 y 的根节点对应的树的深度
            self.fa[root_x] = root_y                    # 向任意一方合并即可
            self.rank[root_y] += 1                      # 因为层数相同，被合并的树必然
层数会 +1
        return True

    def is_connected(self, x, y):                       # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)
```

6. 并查集的应用

并查集通常用来求解不同元素之间的关系问题，比如判断两个人是否是亲戚关系、两个点之间时候存在至少一条路径连接。或者用来求解集合的个数、集合中元素的个数等等。

6.1 等式方程的可满足性

6.1.1 题目链接

- 990. 等式方程的可满足性 - 力扣（ Code）

6.1.2 题目大意

描述：给定一个由字符串方程组成的数组 $equations$，每个字符串方程 $equations[i]$ 的长度为 4，有以下两种形式组成： `a==b` 或 `a!=b` 。a 和 b 是小写字母，表示单字母变量名。

要求：判断所有的字符串方程是否能同时满足，如果能同时满足，返回 $True$，否则返回 $False$。

说明：

- $1 \leq equations.length \leq 500$。
- $equations[i].length == 4$。
- $equations[i][0]$ 和 $equations[i][3]$ 是小写字母。
- $equations[i][1]$ 要么是 `'='` ，要么是 `'!'` 。
- $equations[i][2]$ 是 `'='` 。

示例：

```py
输入    ["a==b","b!=a"]
输出    False
解释    如果我们指定，a = 1 且 b = 1，那么可以满足第一个方程，但无法满足第二个方程。没有办法分配变量同时满足这两个方程。
```

6.1.3 解题思路

字符串方程只有 `==` 或者 `!=`，可以考虑将相等的遍历划分到相同集合中，然后再遍历所有不等式方程，看方程的两个变量是否在之前划分的相同集合中，如果在则说明不满足。

这就需要用到并查集，具体操作如下：

- 遍历所有等式方程，将等式两边的单字母变量顶点进行合并。
- 遍历所有不等式方程，检查不等式两边的单字母遍历是不是在一个连通分量中，如果在则返回 $False$，否则继续扫描。如果所有不等式检查都没有矛盾，则返回 $True$。

6.1.4 代码

```py
class UnionFind:
    def __init__(self, n):                          # 初始化
        self.fa = [i for i in range(n)]             # 每个元素的集合编号初始化为数组 fa 的下标索引

    def find(self, x):                              # 查找元素根节点的集合编号内部实现方法
        while self.fa[x] != x:                      # 递归查找元素的父节点，直到根节点
            self.fa[x] = self.fa[self.fa[x]]        # 隔代压缩优化
            x = self.fa[x]
        return x                                    # 返回元素根节点的集合编号

    def union(self, x, y):                          # 合并操作：令其中一个集合的树根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                        # x 和 y 的根节点集合编号相同，说明 x 和 y 已经同属于一个集合
            return False

        self.fa[root_x] = root_y                    # x 的根节点连接到 y 的根节点
```

```
上，成为 y 的根节点的子节点
        return True

    def is_connected(self, x, y):                    # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)

class Solution:
    def equationsPossible(self, equations: List[str]) -> bool:
        union_find = UnionFind(26)
        for eqation in equations:
            if eqation[1] == "=":
                index1 = ord(eqation[0]) - 97
                index2 = ord(eqation[3]) - 97
                union_find.union(index1, index2)

        for eqation in equations:
            if eqation[1] == "!":
                index1 = ord(eqation[0]) - 97
                index2 = ord(eqation[3]) - 97
                if union_find.is_connected(index1, index2):
                    return False
        return True
```

6.2 省份数量

6.2.1 题目链接

- 547. 省份数量 - 力扣（LeetCode）

6.2.2 题目大意

描述：有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。

「省份」是由一组直接或间接链接的城市组成，组内不含有其他没有相连的城市。

现在给定一个 $n \times n$ 的矩阵 $isConnected$ 表示城市的链接关系。其中 `isConnected[i][j] = 1` 表示第 i 个城市和第 j 个城市直接相连，`isConnected[i][j] = 0` 表示第 i 个城市和第 j 个城市没有相连。

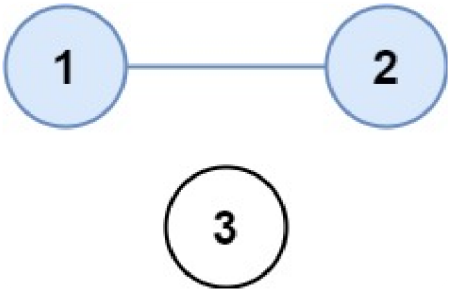
要求：根据给定的城市关系，返回「省份」的数量。

说明：

- $1 \leq n \leq 200$。
- $n == isConnected.length$。
- $n == isConnected[i].length$。
- $isConnected[i][j]$ 为 1 或 0。
- $isConnected[i][i] == 1$。
- $isConnected[i][j] == isConnected[j][i]$。

示例：

- 如图所示：




```py
输入    isConnected = [[1,1,0],[    ],[0,0,1]]
输出    2
```

6.2.3 解题思路

具体做法如下：

- 遍历矩阵 $isConnected$。如果 `isConnected[i][j] = 1` ，将 i 节点和 j 节点相连。
- 然后判断每个城市节点的根节点，然后统计不重复的根节点有多少个，也就是集合个数，即为「省份」的数量。

6.2.4 代码

```py
class UnionFind:
    def __init__(self, n):                          # 初始化
        self.fa = [i for i in range(n)]             # 每个元素的集合编号初始化为数组 fa 的下标索引
```

```
    def find(self, x):                                  # 查找元素根节点的集合编号内部
实现方法
        while self.fa[x] != x:                          # 递归查找元素的父节点，直到根
节点
            self.fa[x] = self.fa[self.fa[x]]            # 隔代压缩优化
            x = self.fa[x]
        return x                                        # 返回元素根节点的集合编号

    def union(self, x, y):                              # 合并操作：令其中一个集合的树
根节点指向另一个集合的树根节点
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:                            # x 和 y 的根节点集合编号相
同，说明 x 和 y 已经同属于一个集合
            return False
        self.fa[root_x] = root_y                        # x 的根节点连接到 y 的根节点
上，成为 y 的根节点的子节点
        return True

    def is_connected(self, x, y):                       # 查询操作：判断 x 和 y 是否同
属于一个集合
        return self.find(x) == self.find(y)

class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        size = len(isConnected)
        union_find = UnionFind(size)
        for i in range(size):
            for j in range(i + 1, size):
                if isConnected[i][j] == 1:
                    union_find.union(i, j)

        res = set()
        for i in range(size):
            res.add(union_find.find(i))
        return len(res)
```

参考资料

- 【博文】并查集 - OI Wiki
- 【博文】并查集 - LeetBook - 力扣
- 【博文】并查集概念及用法分析 - 掘金

- 【博文】数据结构之并查集 - 端碗吹水的技术博客
- 【博文】并查集复杂度 - OI Wiki
- 【题解】使用并查集处理不相交集合问题（Java、Python） - 等式方程的可满足性 - 力扣

- 【书籍】算法训练营 - 陈小玉 著
- 【书籍】算法 第 4 版 - 谢路云 译
- 【书籍】算法竞赛进阶指南 - 李煜东 著
- 【书籍】算法竞赛入门经典：训练指南 - 刘汝佳，陈锋 著