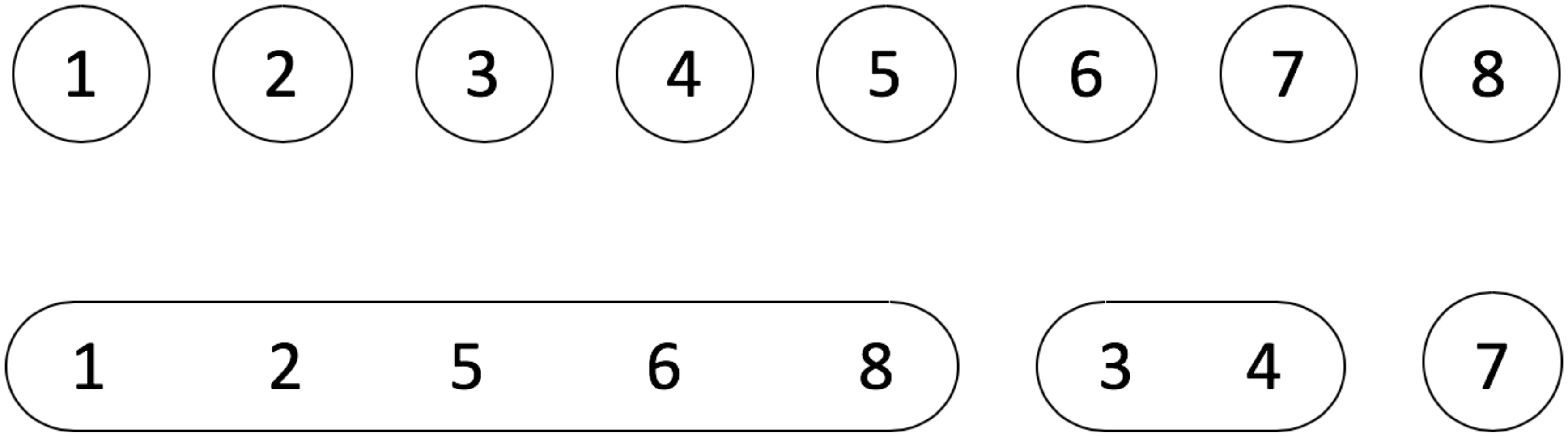
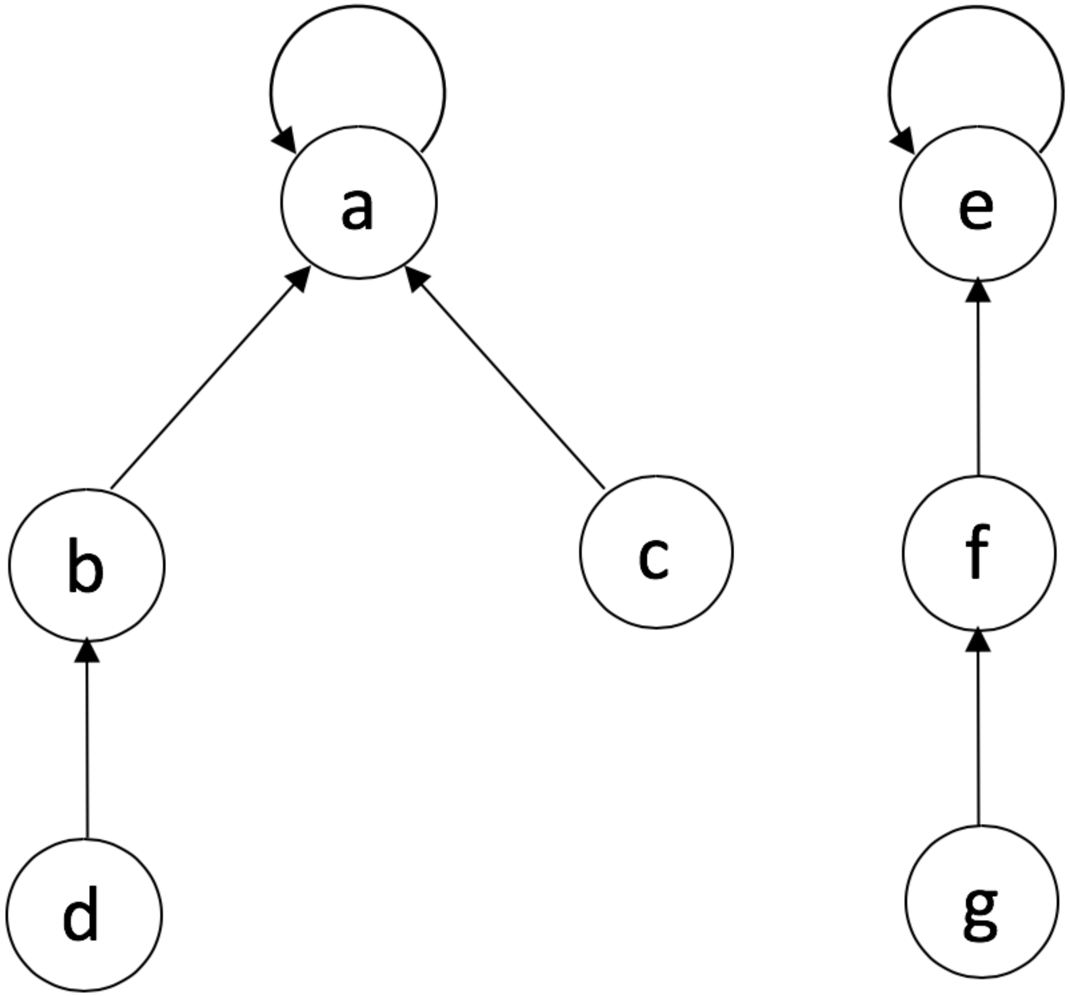
并查集

并查集 (union & find) 是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。

Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。

Union：将两个子集合并成同一个集合。


1
2
3
4
5
6
7
8
1 2 5 6 8
3 4
7

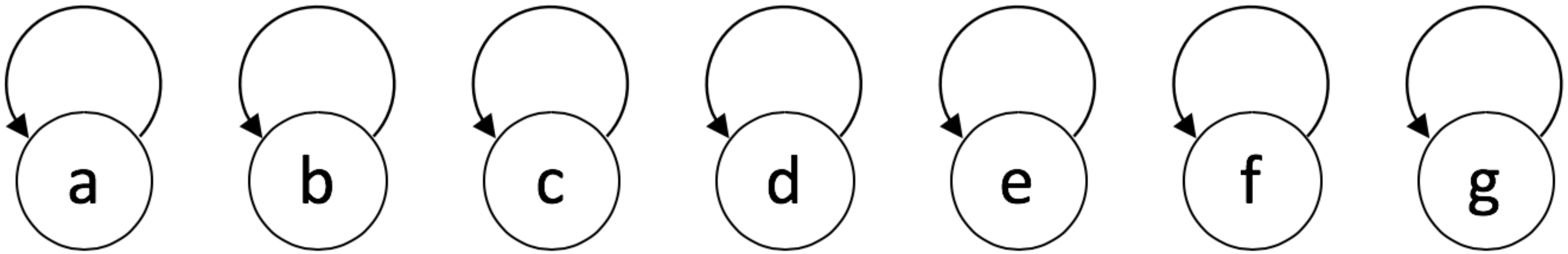

a
b
c
d
e
f
g

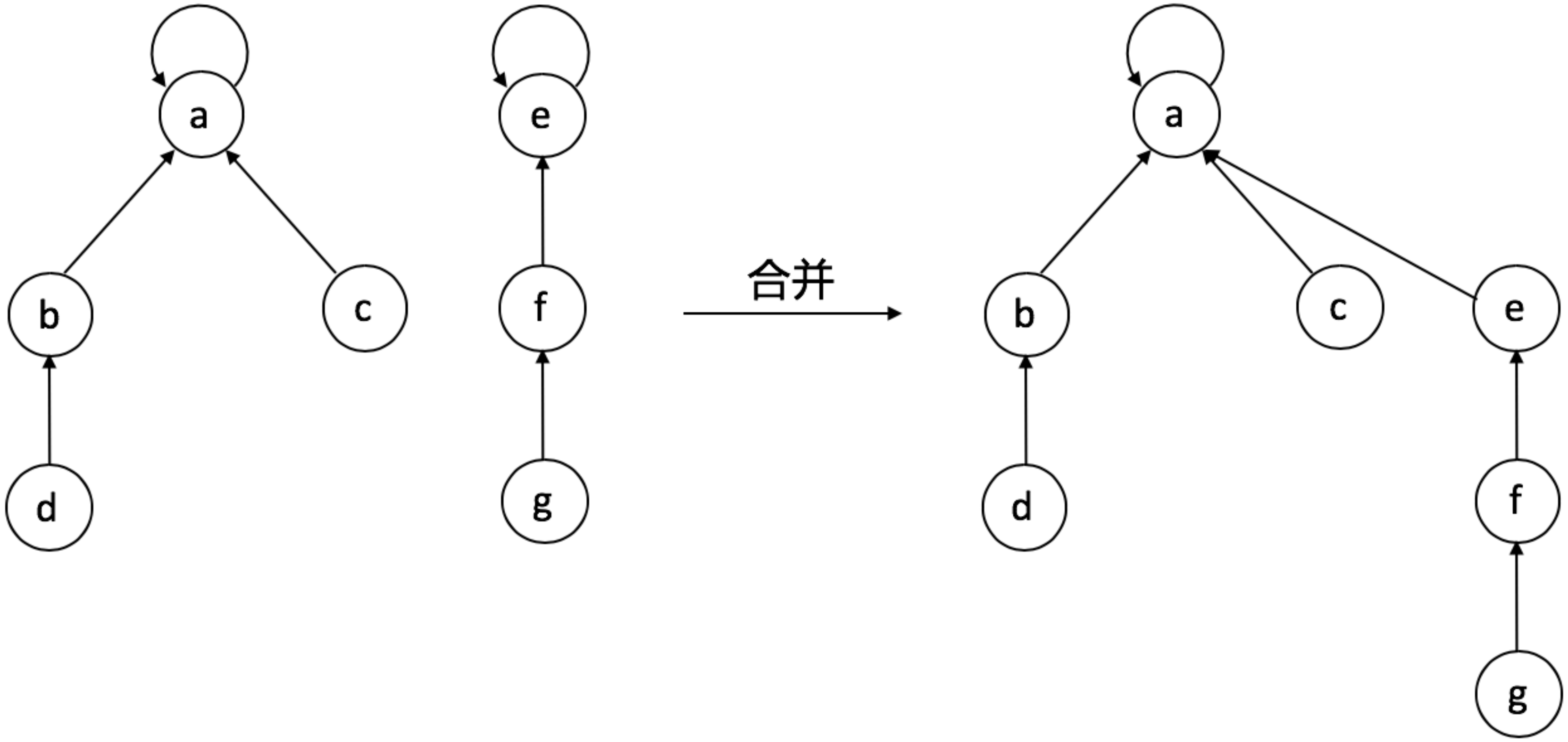
在生活中的例子

1. 小弟 –> 老大

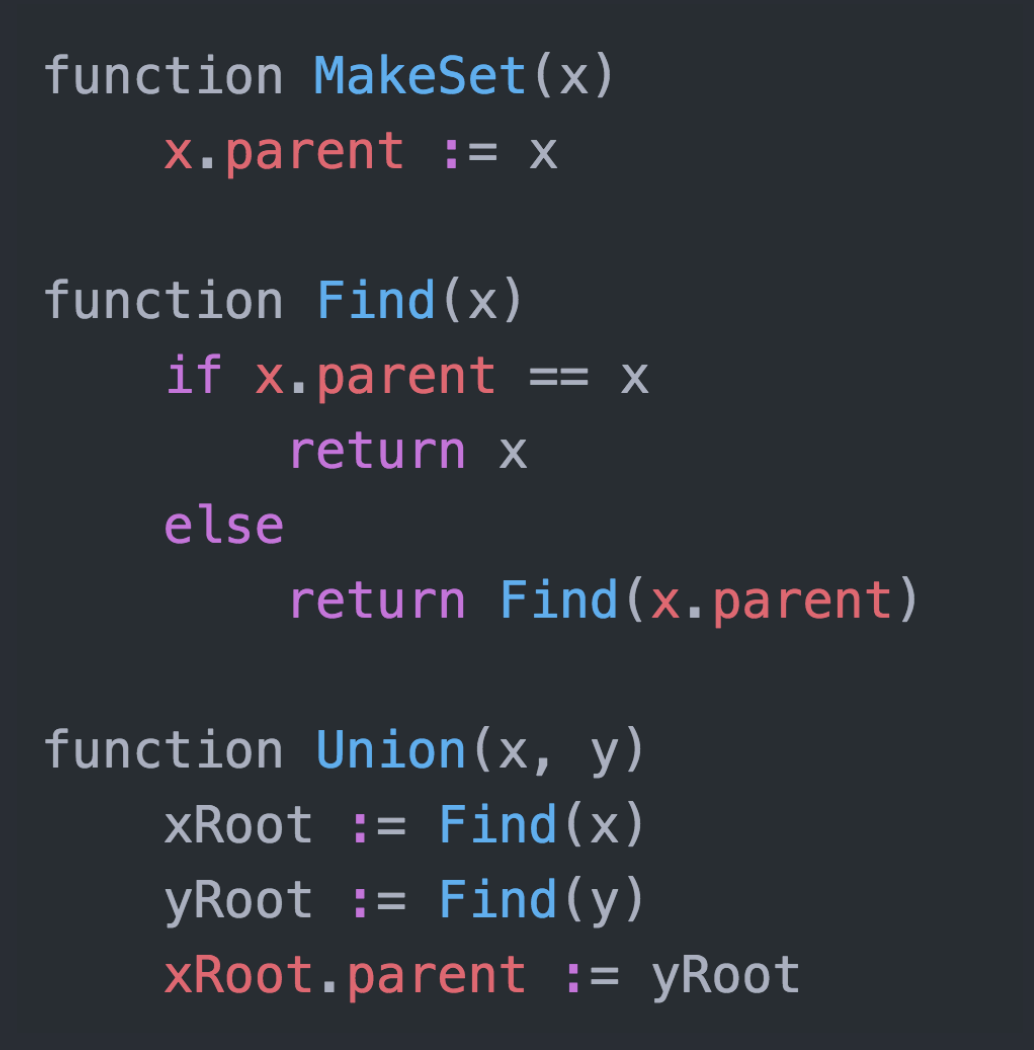
2. 帮派识别

3. 两种优化方式


a
b
c
d
e
f
g

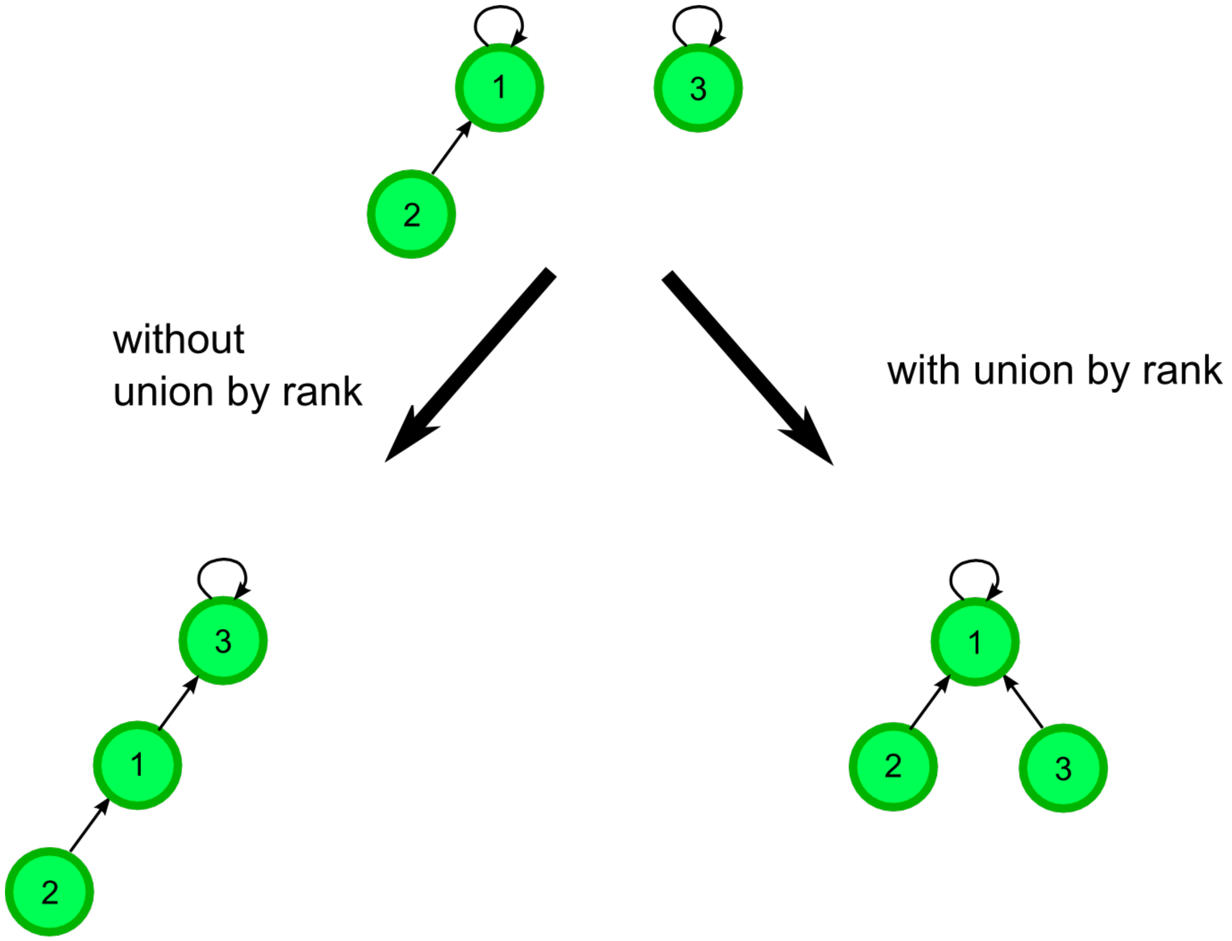

a
e
b
c
f
合并
a
d
g
b
c
e
d
f
g

并查集代码



```
function MakeSet(x)
    x.parent := x

function Find(x)
    if x.parent == x
        return x
    else
        return Find(x.parent)

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

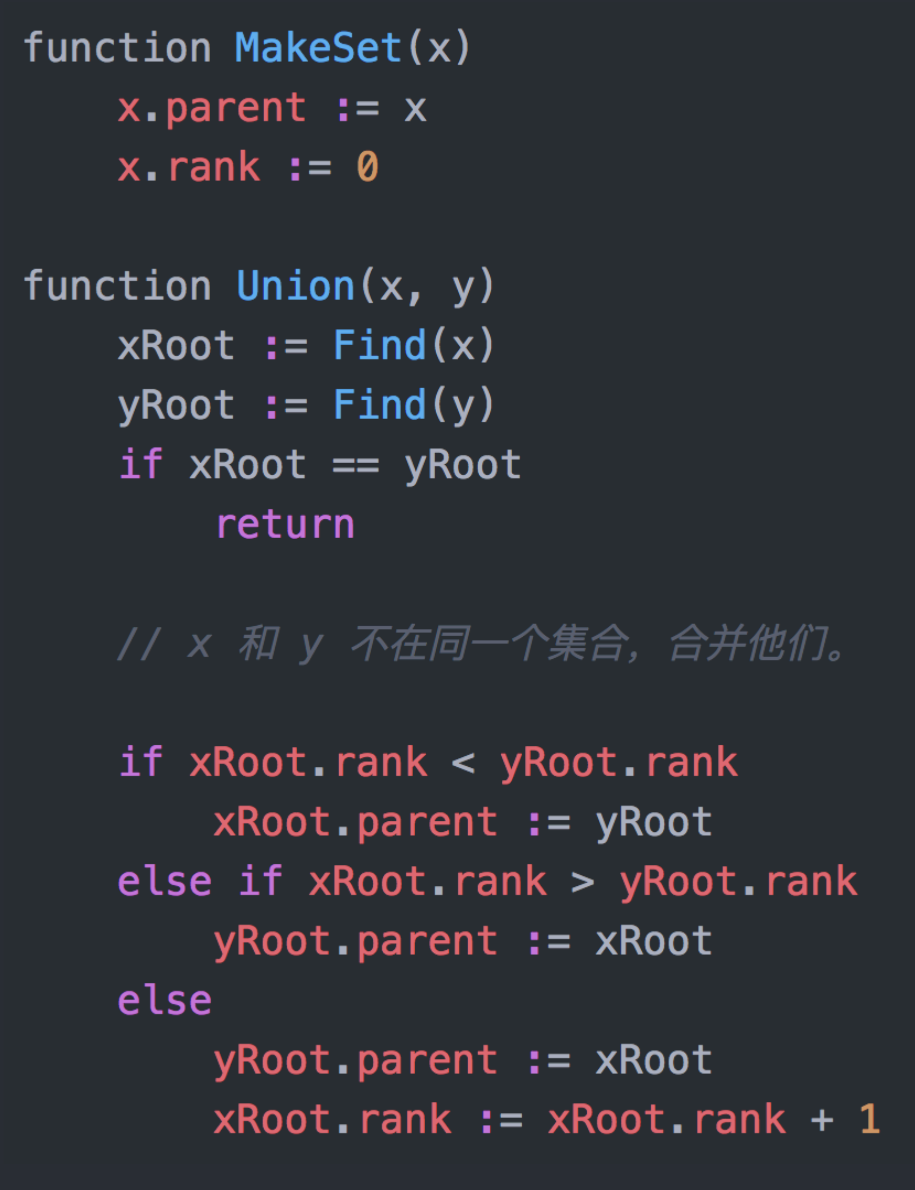
并查集优化一



without union by rank

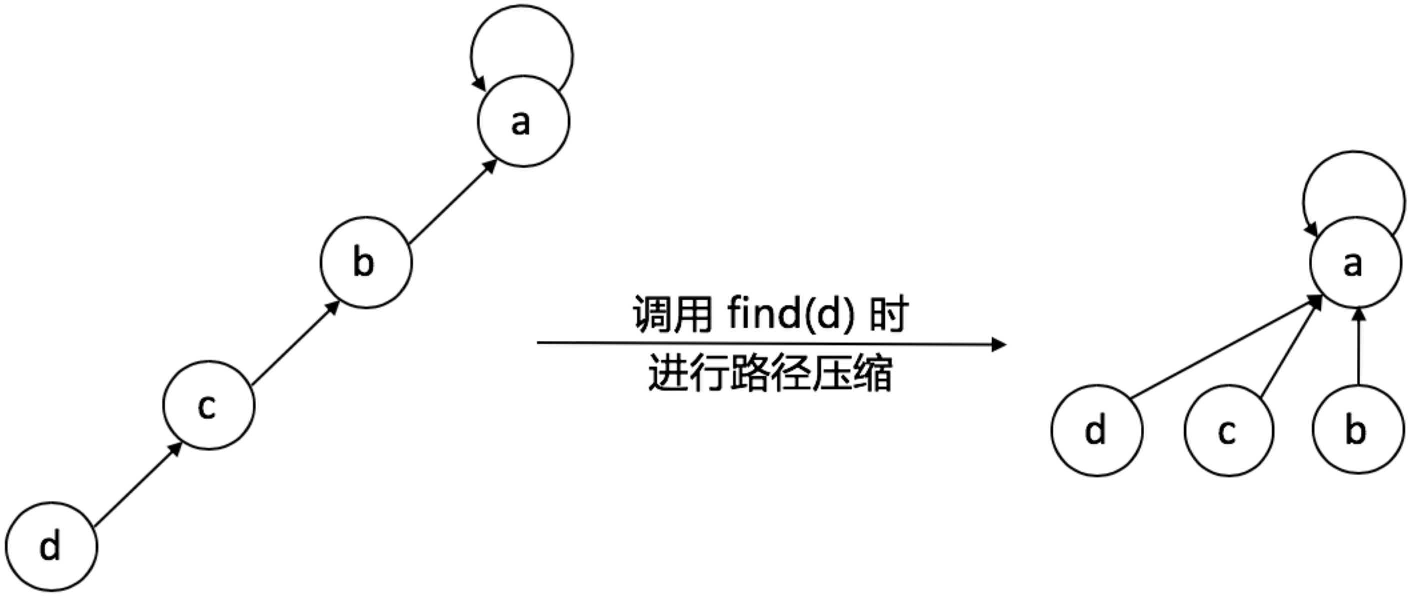
with union by rank

并查集优化一



```
function MakeSet(x)
    x.parent := x
    x.rank := 0

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot == yRoot
        return

    // x 和 y 不在同一个集合，合并他们。

    if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
    else if xRoot.rank > yRoot.rank
        yRoot.parent := xRoot
    else
        yRoot.parent := xRoot
        xRoot.rank := xRoot.rank + 1
```

并查集优化二



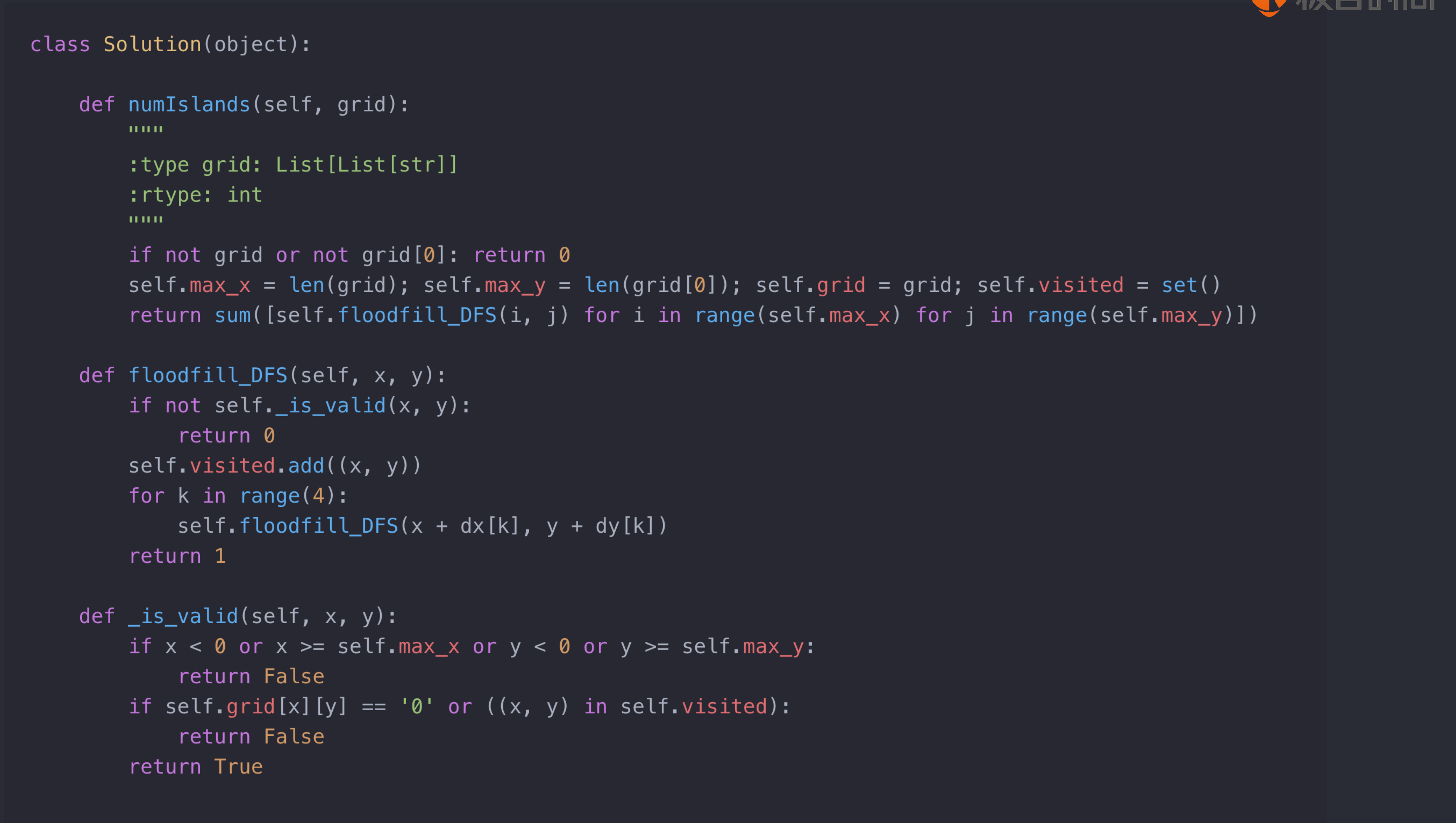
Java 实现
路径压缩

```java
public class QuickUnionUF {
  private int[] roots;

  public QuickUnionUF(int N) {
    roots = new int[N];
    for (int i = 0; i < N; i++) {
      roots[i] = i;
    }
  }

  private int findRoot(int i) {
    int root = i;
    while (root != roots[root])
      root = roots[root];
    while (i != roots[i]) {
      int tmp = roots[i]; roots[i] = root; i = tmp;
    }
    return root;
  }

  public boolean connected(int p, int q) {
    return findRoot(p) == findRoot(q);
  }

  public void union(int p, int q) {
    int qroot = findRoot(q);
    int proot = findRoot(p);
    roots[proot] = qroot;
  }
}
```

实战题目

1. https://leetcode.com/problems/number-of-islands/

2. https://leetcode.com/problems/friend-circles/



```python
class Solution(object):

    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        if not grid or not grid[0]: return 0
        self.max_x = len(grid); self.max_y = len(grid[0]); self.grid = grid; self.visited = set()
        return sum([self.floodfill_DFS(i, j) for i in range(self.max_x) for j in range(self.max_y)])

    def floodfill_DFS(self, x, y):
        if not self._is_valid(x, y):
            return 0
        self.visited.add((x, y))
        for k in range(4):
            self.floodfill_DFS(x + dx[k], y + dy[k])
        return 1

    def _is_valid(self, x, y):
        if x < 0 or x >= self.max_x or y < 0 or y >= self.max_y:
            return False
        if self.grid[x][y] == '0' or ((x, y) in self.visited):
            return False
        return True
```

```
def floodfill_BFS(self, x, y):
    if not self._is_valid(x, y):
        return

    self.visited.add((x, y))
    queue = collections.deque()
    queue.append((x, y))

    while queue:
        cur_x, cur_y = queue.popleft()
        for i in range(4):
            new_x, new_y = cur_x + dx[i], cur_y + dy[i]
            if self._is_valid(new_x, new_y):
                self.visited.add((new_x, new_y))
                queue.append((new_x, new_y))
```