给你一个由 `'1'`（陆地）和 `'0'`（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

输入：
[
['1','1','1','1','0'],
['1','1','0','1','0'],
['1','1','0','0','0'],
['0','0','0','0','0']
]
输出： 1

示例 2：

输入：
[
['1','1','0','0','0'],
['1','1','0','0','0'],
['0','0','1','0','0'],
['0','0','0','1','1']
]
输出： 3
解释： 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。

解法：
DFS：
思想是，遍历长宽，如果遇到1，就把岛屿数量+1，
然后递归将其周围的数都为 0

```
    def dfs(self, grid, r, c):
        grid[r][c] = 0
        nr, nc = len(grid), len(grid[0])
        for x, y in [(r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)]:
            if 0 <= x < nr and 0 <= y < nc and grid[x][y] == "1":
                self.dfs(grid, x, y)
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
         :rtype: int
         """
         # 宽
         nr = len(grid)
         if nr == 0:
             return 0
         # 长
         nc = len(grid[0])

         # 设置岛屿数量为 0
         num_islands = 0
         # 遇到1，就把岛屿数量 +1
```

```
        # 然后遍历它的周围，都弄成0
        for r in range(nr):
            for c in range(nc):
                if grid[r][c] == "1":
                    num_islands += 1
                    self.dfs(grid, r, c)

        return num_islands
```

时间复杂度：O(MN),M N分别是行数和列数

空间复杂度：O(min(M,N))，在最坏的情况下，整个网格均为陆地，深度优先搜索的深度达到MN

BFS

```
def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        nr = len(grid)
        if nr == 0:
            return 0
        nc = len(grid[0])

        num_islands = 0
        # 遍历grid，遇到1，岛屿数量 + 1 ，并将 遍历的位置加1
        for r in range(nr):
            for c in range(nc):
                if grid[r][c] == "1":
                    num_islands += 1
                    grid[r][c] = "0"
                    neighbors = collections.deque([(r, c)])
                    while neighbors:
                        row, col = neighbors.popleft()
                        for x, y in [(row - 1, col), (row + 1, col), (row, col - 1), (row, co
                            if 0 <= x < nr and 0 <= y < nc and grid[x][y] == "1":
                                neighbors.append((x, y))
                                grid[x][y] = "0"

        return num_islands

```

时间复杂度：O(MN),M N分别是行数和列数

空间复杂度：O(min(M,N))，在最坏的情况下，整个网格均为陆地，队列大小为min(M,N)

并查集

```
class UnionFind:
    # 初始化的过程
    def __init__(self, grid):
        m, n = len(grid), len(grid[0])
        # 记录的是岛屿数量
        self.count = 0
        self.parent = [-1] * (m * n)
        self.rank = [0] * (m * n)
        for i in range(m):
```

```
            for j in range(n):
                if grid[i][j] == "1":
                    self.parent[i*n +j] = i * n + j
                    self.count += 1
    # 查找
    def find(self, i):
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    # 合并
    def union(self, x, y):
        rootx = self.find(x)
        rooty = self.find(y)
        if rootx != rooty:
            if self.rank[rootx] < self.rank[rooty]:
                rootx, rooty  = rooty, rootx
            self.parent[rooty] = rootx
            if self.rank[rootx] == self.rank[rooty]:
                self.rank[rootx] += 1
            self.count -=1

    def getCount(self):
        return self.count

class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        nr = len(grid)
        if nr == 0:
            return 0
        nc = len(grid[0])

        uf = UnionFind(grid)
        num_island = 0
        for r in range(nr):
            for c in range(nc):
                if grid[r][c] == "1":
                    grid[r][c] = "0"
                    for x, y in [(r-1, c), (r+1, c), (r, c-1), (r, c+1)]:
                        if 0 <= x <nr and 0 <= y < nc and grid[x][y] == "1":
                            uf.union(r * nc + c, x * nc + y) # 合并点(r,c)和其上下左右为1的点

        return uf.getCount
```

时间复杂度：O(MN *α(*MN*))， M，N是行数和列数

当使用路径压缩（find）和按秩合并（rank）现并查集时，单次操作的时间复杂度为α(MN)，其中α(x) 为反阿克曼函数，当自变量 xx 的值在人类可观测的范围内（宇宙中粒子的数量）时，函数α(x) 的值不会超过 5，因此也可以看成是常数时间复杂度。

空间复杂度：O(MN)

按秩合并：

给每个点一个秩，其实就是树高，每次合并的时候都用秩小的指向秩大的，可以保证树高最高为$\log_2(n)$。操作的时候，一开始所有点的秩都为1

在一次合并后，假设是点x和点y，x的秩小，当然x和y都是原来x和y所在区间的顶点，设点x秩为rank[x]，将fa[x]指向y，然后将rank[y]的与rank[x+1]取max。因为rank[x]为区间x的高，将它连向y之后，y的树高就会是x的树高+1，当然也可能y在另一边树高更高，所以取max