DFS & BFS

岛屿类题目

200. Number of Islands（模板题）

```
class Solution:
    def numIslands(self, grid: [[str]]) -> int:
        def dfs(grid, i, j):
            if not 0 <= i < len(grid) or not 0 <= j < len(grid[0]) or grid[i][j] == '0':
return
            grid[i][j] = '0' # 淹没 这里直接在原数组上修改 很多时候会使用visited数组
            '''
            #写法一
            dfs(grid, i + 1, j)
            dfs(grid, i, j + 1)
            dfs(grid, i - 1, j)
            dfs(grid, i, j - 1)
            '''

            #写法二
            directions = [(1,0),(-1,0),(0,1),(0,-1)]
            for d in directions:
                cur_i = i + d[0]
                cur_j = j + d[1]
                dfs(grid, cur_i, cur_j)

        count = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    dfs(grid, i, j)
                    count += 1
        return count
```

463. Island Perimeter

```
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int:
        # 求周长

        def dfs(grid,i,j):
```

```
            #本题特殊判断

            # 从一个岛屿方格走向网格边界，周长加 1
            if not 0<=i<len(grid) or not 0<=j<len(grid[0]):
                return 1

            #从一个岛屿方格走向水域方格，周长加 1
            if grid[i][j] == 0:
                return 1

            #遇到遍历过的，周长不变
            if grid[i][j] == 2:
                return 0

        #这里开始后面都是模版
            grid[i][j] = 2
            ans = 0
            directions = [(1,0),(-1,0),(0,1),(0,-1)]
            for d in directions:
                cur_i = i + d[0]
                cur_j = j + d[1]
                ans += dfs(grid, cur_i, cur_j)
            return ans

        ans = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == 1:
                    ans += dfs(grid,i,j)
        return ans
```

695. Max Area of Island

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        # 同200 只是需要记录最大岛屿面积

        #dfs作用：走过的全部等于0，然后统计走过的路径长度
        def dfs(grid,i,j):
            if not 0 <= i < len(grid) or not 0 <= j < len(grid[0]):
                return 0
            if grid[i][j] == 0:
                return 0
```

```
            grid[i][j] = 0

            ans = 1 # 为什么要等于1 因为是从岛开始搜，而不是从水开始搜
            directions = [(1,0),(-1,0),(0,1),(0,-1)]
            for d in directions:
                cur_i = i + d[0]
                cur_j = j + d[1]
                ans += dfs(grid, cur_i, cur_j)
            return ans

        ans = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == 1:
                    ans = max(ans,dfs(grid,i,j))
        return ans
```

1020. Number of Enclaves

```
class Solution:
    def numEnclaves(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])
        #dfs函数的目的：将走过的标记成0
        def dfs(grid,i,j):
            if not 0 <= i < m or not 0 <= j < n:
                return
            if grid[i][j] == 0:
                return

            grid[i][j] = 0

            directions = [(1,0),(-1,0),(0,1),(0,-1)]
            for d in directions:
                cur_i = i + d[0]
                cur_j = j + d[1]
                dfs(grid, cur_i, cur_j)


        #遍历所有边界为1的结点，找到他们联通的所有路径，全部变成0，那么剩下为1的结点就是需要的数量
        for i in range(m):
            if grid[i][0] == 1:
                dfs(grid,i,0)
            if grid[i][n-1] == 1:
```

```
                dfs(grid,i,n-1)

        for j in range(n):
            if grid[0][j] == 1:
                dfs(grid,0,j)
            if grid[m-1][j] == 1:
                dfs(grid,m-1,j)

        #统计非0的数量
        ans = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    # 这里是统计1的数量
                    ans += 1
        return ans
```

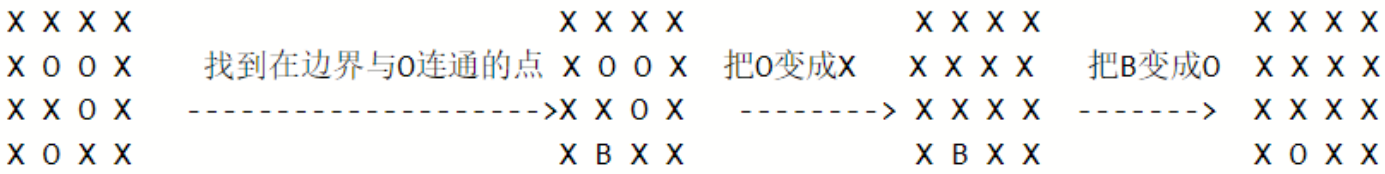
1254. Number of Closed Islands

类似题目：200,1020

```
class Solution:
    def closedIsland(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])
        #dfs函数的目的：将走过的标记成1
        def dfs(grid,i,j):
            if not 0 <= i < m or not 0 <= j < n:
                return
            if grid[i][j] == 1:
                return

            grid[i][j] = 1

            directions = [(1,0),(-1,0),(0,1),(0,-1)]
            for d in directions:
                cur_i = i + d[0]
                cur_j = j + d[1]
                dfs(grid, cur_i, cur_j)


        #遍历所有边界为0的结点，找到他们联通的所有路径，全部变成1，那么剩下为0的结点就是需要的数量
        for i in range(m):
```

```
            if grid[i][0] == 0:
                dfs(grid,i,0)
            if grid[i][n-1] == 0:
                dfs(grid,i,n-1)

        for j in range(n):
            if grid[0][j] == 0:
                dfs(grid,0,j)
            if grid[m-1][j] == 0:
                dfs(grid,m-1,j)

        ans = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 0:
                    ans += 1
                    #最大一个区别在这里 1020需要统计的是0的个数，而这里不是，这里需要的是0群的个数，同200.
                    dfs(grid, i, j)
        return ans
```

130. Surrounded Regions




```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        m = len(board)
        n = len(board[0])
        def dfs(board,i,j):
            if not 0 <= i < m or not 0 <= j < n:
                return
            if board[i][j] != 'O': #这里要使用不等于 因为出现了第三个字母
                return
            board[i][j] = '#'

            directions = [(1,0),(-1,0),(0,1),(0,-1)]
```

```
            for d in directions:
                cur_i = i + d[0]
                cur_j = j + d[1]
                dfs(board, cur_i, cur_j)

        for i in range(m):
            if board[i][0] == 'O':
                dfs(board,i,0)
            if board[i][n-1] == 'O':
                dfs(board,i,n-1)

        for j in range(n):
            if board[0][j] =='O':
                dfs(board,0,j)
            if board[m-1][j] == 'O':
                dfs(board,m-1,j)

        for i in range(m):
            for j in range(n):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                if board[i][j] == '#':
                    board[i][j] = 'O'
```

1905. Count Sub Islands

其他

329.Longest Increasing Path in a Matrix

```
class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        m, n = len(matrix), len(matrix[0])
        flag = [[-1] * n for _ in range(m)] #存储从（i, j）出发的最长递归路径

        def dfs(i, j):
            if flag[i][j] != -1: # 记忆化搜索，避免重复的计算
                return flag[i][j]
            else:
```

```
                d = 1
                for (x, y) in [[-1, 0], [1, 0], [0, 1], [0, -1]]:
                    x, y = i + x, j + y
                    #在里面判断
                    if 0 <= x < m and 0 <= y < n and matrix[x][y] > matrix[i][j]:
                        d = max(d, dfs(x, y) + 1) # 取四个邻接点的最长
                flag[i][j] = d
                return d

        res = 0
        for i in range(m): # 遍历矩阵计算最长路径
            for j in range(n):
                if flag[i][j] == -1:
                    res = max(res, dfs(i, j))
        return res
```

79. Word Search

```
class Solution(object):

    # 定义上下左右四个行走方向
    directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    def exist(self, board, word):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """
        m = len(board)
        if m == 0:
            return False
        n = len(board[0])
        mark = [[0 for _ in range(n)] for _ in range(m)]

        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] == word[0]:
                    # 将该元素标记为已使用
                    mark[i][j] = 1
                    if self.backtrack(i, j, mark, board, word[1:]) == True:
                        return True
                    else:
                        # 回溯
```

```
                        mark[i][j] = 0
        return False


    def backtrack(self, i, j, mark, board, word):
        if len(word) == 0:
            return True

        #回溯模版
        #如何遍历上下左右的做法需要学习
        for direct in self.directs:
            cur_i = i + direct[0]
            cur_j = j + direct[1]

            if cur_i >= 0 and cur_i < len(board) and cur_j >= 0 and cur_j < len(board[0])
and board[cur_i][cur_j] == word[0]:
                # 如果是已经使用过的元素，忽略
                if mark[cur_i][cur_j] == 1:
                    continue
                # 将该元素标记为已使用
                mark[cur_i][cur_j] = 1
                if self.backtrack(cur_i, cur_j, mark, board, word[1:]) == True:
                    return True
                else:
                    # 回溯
                    mark[cur_i][cur_j] = 0
        return False
```