【第十五课】广搜

彩蛋入口题目：开课吧OJ-115

993. 二叉树的堂兄弟节点

1.要想判断两个节点 x 和 y 是否为堂兄弟节点，我们就需要求出这两个节点分别的「深度」以及「父节点」。2.

2.因此，我们可以从根节点开始，对树进行一次遍历，在遍历的过程中维护「深度」以及「父节点」这两个信息。

3.当我们遍历到 x 或 y 节点时，就将信息记录下来；当这两个节点都遍历完成了以后，我们就可以退出遍历的过程，判断它们是否为堂兄弟节点了。

4.我们只需要在深度优先搜索的递归函数中增加表示「深度」以及「父节点」的两个参数即可。

```
/**
 * @param {TreeNode} root
 * @param {number} x
 * @param {number} y
 * @return {boolean}
 */
var isCousins = function(root, x, y) {
    // x 的信息
    let x_parent = null, x_depth = null, x_found = false;
    // y 的信息
    let y_parent = null, y_depth = null, y_found = false;

    const dfs = (node, depth, parent) => {
        if (!node) {
            return;
        }
        if (node.val === x) {
            [x_parent, x_depth, x_found] = [parent, depth, true];
        } else if (node.val === y) {
            [y_parent, y_depth, y_found] = [parent, depth, true];
        }

        // 如果两个节点都找到了，就可以提前退出遍历
        // 即使不提前退出，对最坏情况下的时间复杂度也不会有影响
        if (x_found && y_found) {
            return;
        }

        dfs(node.left, depth + 1, node);

        if (x_found && y_found) {
            return;
        }

        dfs(node.right, depth + 1, node);
    }
    dfs(root, 0, null);
```

```
    return x_depth === y_depth && x_parent !== y_parent;
};
```

542. 01 矩阵

本题可以转化成对于从所有值为0的点同时向外扩散的时间（速度为 1 格/次），求当扩散到矩阵中所有点的时间。

```
/**
 * @param {number[][]} matrix
 * @return {number[][]}
 */
//  bfs最优解
// 本题可以转化成对于从所有值为0的点同时向外扩散的时间（速度为 1 格/次），求当扩散到矩阵中所有点的时间。
var updateMatrix = function(matrix) {
    if(!matrix.length || !matrix[0].length) return null;

    let n = matrix.length;
    let m = matrix[0].length;
    let ans = new Array(n);
    for(let i = 0; i < n; i++) ans[i] = new Array(m).fill(-1);

    let queue = [];
    for(let i = 0; i < n; i++)
        for(let j = 0; j < m; j++) {
            if(matrix[i][j] === 0) {
                ans[i][j] = 0;
                queue.push([i, j]);
            }
        }

    let dist = 0;
    while(queue.length) {
        let len = queue.length;
        dist++;
        for(let i = 0; i < len; i++) {
            let top = queue.shift();

            if(top[0] + 1 < n && ans[top[0]+1][top[1]] === -1) {
                queue.push([top[0]+1, top[1]]);
                ans[top[0]+1][top[1]] = dist;
            }
            if(top[0] - 1 >= 0 && ans[top[0]-1][top[1]] === -1) {
                queue.push([top[0]-1, top[1]]);
                ans[top[0]-1][top[1]] = dist;
            }
            if(top[1] + 1 < m && ans[top[0]][top[1]+1] === -1) {
                queue.push([top[0], top[1]+1]);
                ans[top[0]][top[1]+1] = dist;
            }
            if(top[1] - 1 >= 0 && ans[top[0]][top[1]-1] === -1) {
                queue.push([top[0], top[1]-1]);
                ans[top[0]][top[1]-1] = dist;
            }
        }
```

```
    }
    return ans;

};
```

1091. 二进制矩阵中的最短路径

1.从任意一个位置，都有8个方向可以走，每次走一步。但其中上、左、左上可以不用走，因为该题并不存在遇到障碍物后需要往回走绕过的场景，只需要不断向右下方走即可。
2.使用BFS，每次循环都以当前一层节点为起点，向右下方扩散出下一层节点。每次走一层步数（即路径长度）加1。
3.当遇到终点时，表示找到了最短路径，此时的层数就是路径长度

```
/**
 * @param {number[][]} grid
 * @return {number}
 */
var shortestPathBinaryMatrix = function (grid) {
  // 缓存矩阵的终点位置
  const m = grid.length - 1;
  const n = grid[0].length - 1;

  // 当起点和终点为1时，必然无法到达终点
  if (grid[0][0] === 1 || grid[m][n] === 1) {
    return -1;
  }

  // 如果矩阵只有1个点，且为0，路径为1
  if (m === 0 && n === 0 && grid[0][0] === 0) {
    return 1;
  }

  let queue = [[0, 0]]; // 使用队列进行BFS搜索
  let level = 1; // 缓存路径长度，起点的长度为1
  // 可以向四周所有方向行走，缓存8个方向
  const direction = [
    [-1, 1], // 右上
    [0, 1], // 右
    [1, 1], // 右下
    [1, 0], // 下
    [1, -1], // 左下
    [-1, 0], // 上
    [0, -1], // 左
    [-1, -1], // 左上
  ];

  // 如果队列中有值，则继续搜索
  while (queue.length) {
    // 缓存当前层的节点数量
    let queueLength = queue.length;

    // 每次只遍历当前一层的节点
    while (--queueLength >= 0) {
      // 出队一个坐标，计算它可以行走的下一步位置
      const [x, y] = queue.shift();
```

```
      for (let i = 0; i < direction.length; i++) {
        // 下一步可以向四周行走，计算出相应新坐标
        const newX = x + direction[i][0];
        const newY = y + direction[i][1];

        // 如果新坐标超出网格，或者被标记为1，表示无法行走，则跳过
        if (
          newX < 0 ||
          newY < 0 ||
          newX > m ||
          newY > m ||
          grid[newX][newY] === 1
        ) {
          continue;
        }

        // 如果新坐标是终点，表示找到路径，返回长度即可
        if (newX === m && newY === n) {
          return level + 1;
        }
        // 将走过的位置标记为1，避免重复行走
        grid[newX][newY] = 1;
        // 将下一步的坐标存入队列，用于下一层循环
        queue.push([newX, newY]);
      }
    }

    level++; // 每向前走一层，将步数加1
  }

  return -1;
};
```

752. 打开转盘锁

1.一个锁拨动一次，只有两种选择：向上，向下（要注意0、9边界性）。

2.当前有四个锁，所以选择一个锁拨动一次时，有 2 * 4 种可能性。

3.在这次选择的基础上，进行下一次选择，又有8种可能性，依次类推 。

```
/**
 * @param {string[]} deadends
 * @param {string} target
 * @return {number}
 */
var openLock = function(deadends, target) {
    let path = '0000';
    const queue = [path];
    const visited = {}; // 已访问节点
    const dead = {}; // deadends死节点
    let step = 0; // 最少拨动次数
    const swipeUp = (path, index) => { // 向上拨动一个数字
        const arr = path.split('');
        if (arr[index] === '9') {
            arr[index] = '0';
```

```
        } else {
            arr[index]++;
        }
       return arr.join('');
    };
    const swipeDown = (path, index) => { // 向下拨动一个数字
        const arr = path.split('');
        if (arr[index] === '0') {
            arr[index] = '9';
        } else {
            arr[index]--;
        }
        return arr.join('');
    };
    for (let num of deadends) {
        dead[num] = true;
    }

    while (queue.length) {
        const len = queue.length;
        for (let i = 0; i < len; i++) {
            const password = queue.shift();
            if (dead[password]) continue; // 剪枝
            if (password === target) return step; // 找到答案
            for (let j = 0; j < 4; j++) {
                const upPassword = swipeUp(password, j);
                const downPassword = swipeDown(password, j);
                !visited[upPassword] && queue.push(upPassword) &&
(visited[upPassword] = true); // 剪枝
                !visited[downPassword] && queue.push(downPassword) &&
(visited[downPassword] = true); // 剪枝
            }
        }

        step++;
    }
    return -1;
};
```

剑指 Offer 13. 机器人的运动范围

1.将行坐标和列坐标数位之和大于 `k` 的格子看作障碍物，然后用广度优先搜索它。

2.我们只需要对数 x 每次对 10 取余，就能知道数 x 的个位数是多少，然后再将 x 除 10，这个操作等价于将 x 的十进制数向右移一位，删除个位数（类似于二进制中的 >> 右移运算符），不断重复直到 x 为 0 时结束。

```
/**
 * @param {number} m
 * @param {number} n
 * @param {number} k
 * @return {number}
 */
var movingCount = function(m, n, k) {
    // 位数和
    function getSum(num) {
```

```
        let answer = 0;

        while(num) {
            answer += num % 10;
            num = Math.floor(num / 10);
        }

        return answer;
    }
    // 方向数组
    const directionAry = [
        [-1, 0], // 上
        [0, 1], // 右
        [1, 0], // 下
        [0, -1] // 左
    ];

    // 已经走过的坐标
    let set = new Set(['0,0']);

    // 将遍历的坐标队列，题意要求从[0,0]开始走
    let queue = [[0, 0]];

    // 遍历队列中的坐标
    while(queue.length) {
        // 移除队首坐标
        let [x, y] = queue.shift();

        // 遍历方向
        for(let i = 0; i < 4; i++) {
            let offsetX = x + directionAry[i][0];
            let offsetY = y + directionAry[i][1];


            // 临界值判断
            if(offsetX < 0 || offsetX >= m || offsetY < 0 || offsetY >= n ||
getSum(offsetX) + getSum(offsetY) > k || set.has(`${offsetX},${offsetY}`)) {
                continue;
            }

            // 走过的格子就不再纳入统计
            set.add(`${offsetX},${offsetY}`);

            // 将该坐标加入队列（因为这个坐标的四周没有走过，需要纳入下次的遍历）
            queue.push([offsetX, offsetY]);
        }
    }

    // 走过坐标的个数就是可以到达的格子数
    return set.size;
};
```

开课吧