1260. 二维网格迁移

1. 题目描述

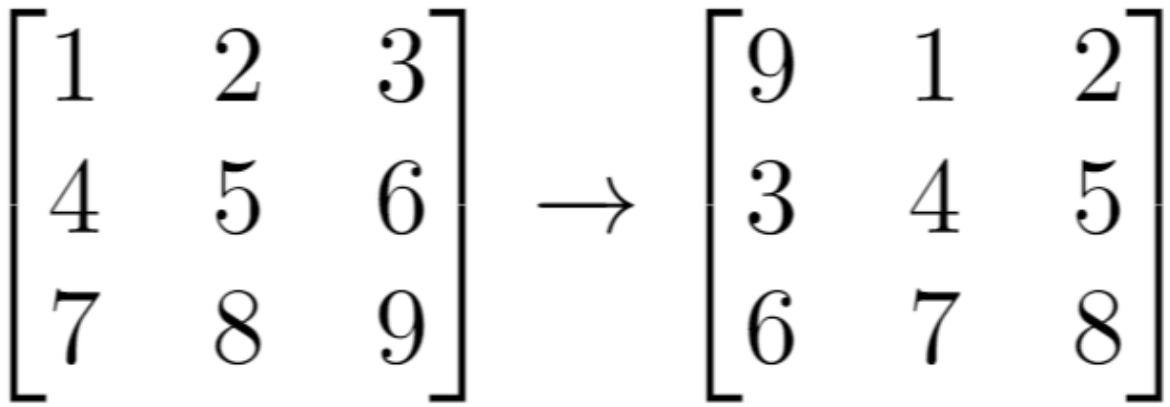
给你一个 `n` 行 `m` 列的二维网格 `grid` 和一个整数 `k`。你需要将 `grid` 迁移 `k` 次。

每次「迁移」操作将会引发下述活动：

- 位于 `grid[i][j]` 的元素将会移动到 `grid[i][j + 1]`。
- 位于 `grid[i][m - 1]` 的元素将会移动到 `grid[i + 1][0]`。
- 位于 `grid[n - 1][m - 1]` 的元素将会移动到 `grid[0][0]`。

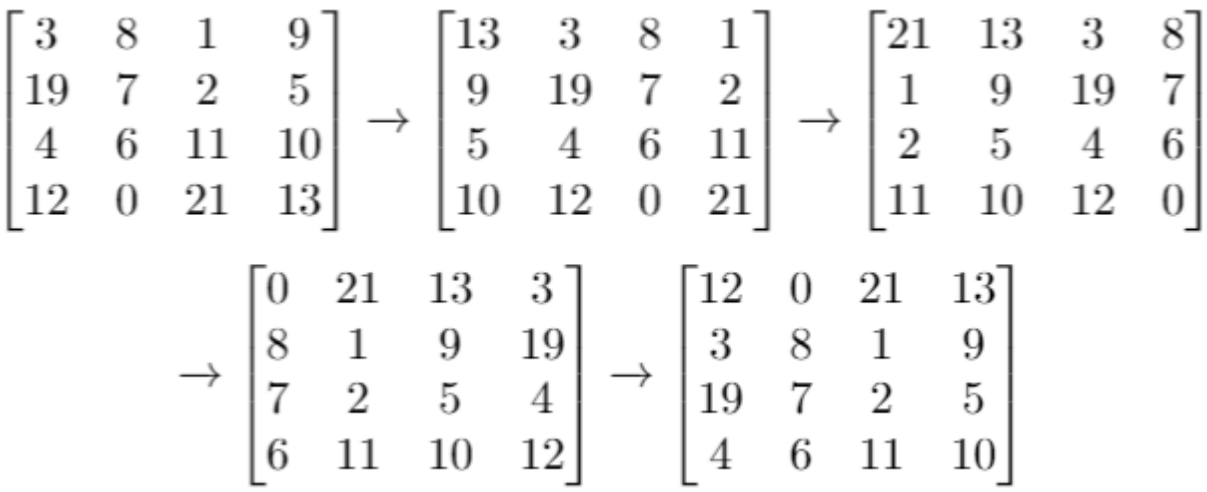
请你返回 `k` 次迁移操作后最终得到的 **二维网格**。

示例 1：



```
输入：grid = [[1,2,3],[4,5,6],[7,8,9]], k = 1
输出：[[9,1,2],[3,4,5],[6,7,8]]
```

示例 2：



```
输入：grid = [[3,8,1,9],[19,7,2,5],[4,6,11,10],[12,0,21,13]], k = 4
输出：[[12,0,21,13],[3,8,1,9],[19,7,2,5],[4,6,11,10]]
```

示例 3：

```
输入：grid = [[1,2,3],[4,5,6],[7,8,9]], k = 9
输出：[[1,2,3],[4,5,6],[7,8,9]]
```

提示：

- `1 <= grid.length <= 50`
- `1 <= grid[i].length <= 50`
- `-1000 <= grid[i][j] <= 1000`
- `0 <= k <= 100`

2. 比赛时实现

总结数学规律

```cpp
class Solution {
public:
    vector<vector<int>> shiftGrid(vector<vector<int>>& grid, int k) {
        int n = grid.size();
        int m = grid[0].size();
        k %= m*n;//每m*n次迁移为一个循环，恢复为原数组
        if(k == 0) return grid;
        int row = k / m;//每m次迁移意味着grid的每一行循环下移一行,此处即求得变换后首元素的行位移
        int col = k % m;//同理求得列位移
        vector<vector<int>> ans = vector<vector<int>>(n, vector<int>(m));
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++){//将原数组从首元素开始按行优先存入ans中
                ans[row][col] = grid[i][j];
                col++;
                if(col == m){//列满了，转到下一行首
                    col = 0;
                    row++;
                    if(row == n)//行满了，转到首行
                        row = 0;
                }
            }
        return ans;
    }
};
```

3. 其他方法——双向队列

双向队列

1. 先将二维网格线性成一维的双向队列
2. 执行k次操作：将双向队列的最后一个数字压入到队列的前面，弹出队列内的最后一个数字；
3. 最后将一维的双向队列重新转换成二维的网格即可。

```
class Solution {
public:
        vector<vector<int>> shiftGrid(vector<vector<int>>& grid, int k) {
                deque<int> dequeGrid;
                for (size_t i = 0; i < grid.size(); ++i) {
                        for (size_t j = 0; j < grid[i].size(); ++j) {
                                dequeGrid.push_back(grid[i][j]);
                        }
                }
                for (int i = 0; i < k; ++i) {
                        int nBack = dequeGrid.back();
                        dequeGrid.pop_back();
                        dequeGrid.push_front(nBack);
                }

                for (size_t i = 0, k = 0; i < grid.size(); ++i) {
                        for (size_t j = 0; j < grid[i].size(); ++j) {
                                grid[i][j] = dequeGrid.at(k++);
                        }
                }

                return grid;
        }
};
```

1261.在受污染的二叉树中查找元素

1. 题目描述

给出一个满足下述规则的二叉树：

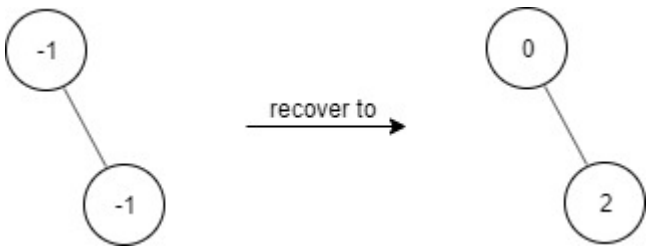
1. `root.val == 0`
2. 如果 `treeNode.val == x` 且 `treeNode.left != null`，那么 `treeNode.left.val == 2 * x + 1`
3. 如果 `treeNode.val == x` 且 `treeNode.right != null`，那么 `treeNode.right.val == 2 * x + 2`

现在这个二叉树受到「污染」，所有的 `treeNode.val` 都变成了 `-1`。

请你先还原二叉树，然后实现 `FindElements` 类：

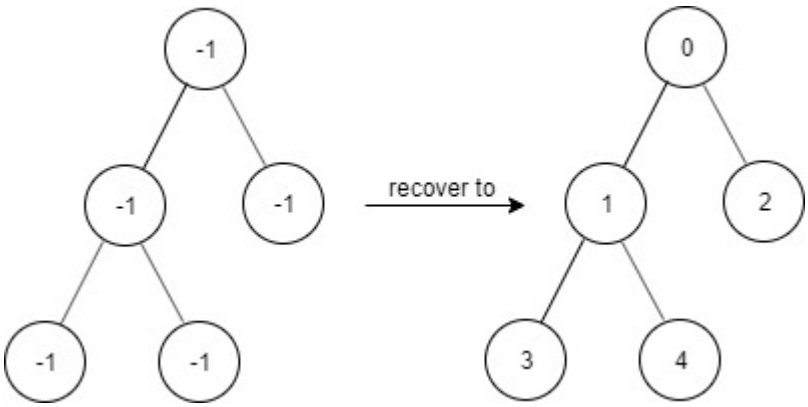
- `FindElements(TreeNode* root)` 用受污染的二叉树初始化对象，你需要先把它还原。
- `bool find(int target)` 判断目标值 `target` 是否存在于还原后的二叉树中并返回结果。

示例 1：



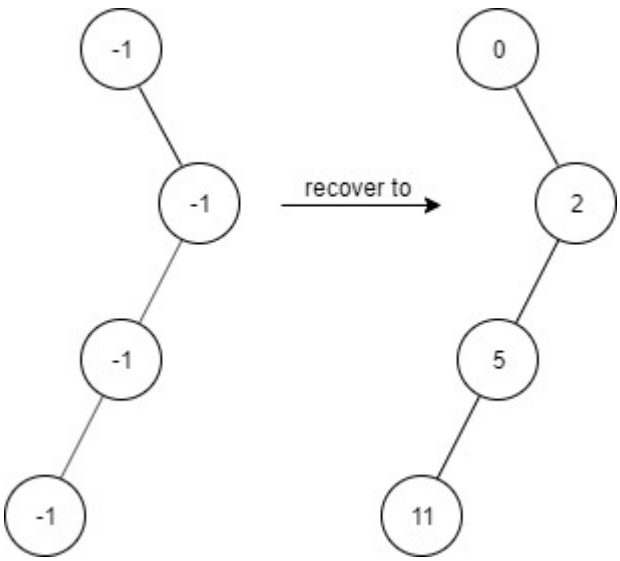
```
输入:
["FindElements","find","find"]
[[[-1,null,-1]],[1],[2]]
输出:
[null,false,true]
解释:
FindElements findElements = new FindElements([-1,null,-1]);
findElements.find(1); // return False
findElements.find(2); // return True
```

示例 2:




```
输入:
["FindElements","find","find","find"]
[[[-1,-1,-1,-1,-1]],[1],[3],[5]]
输出:
[null,true,true,false]
解释:
FindElements findElements = new FindElements([-1,-1,-1,-1,-1]);
findElements.find(1); // return True
findElements.find(3); // return True
findElements.find(5); // return False
```

示例 3:



```
输入:
["FindElements","find","find","find","find"]
[[[-1,null,-1,-1,null,-1]],[2],[3],[4],[5]]
输出:
[null,true,false,false,true]
解释:
FindElements findElements = new FindElements([-1,null,-1,-1,null,-1]);
findElements.find(2); // return True
findElements.find(3); // return False
findElements.find(4); // return False
findElements.find(5); // return True
```

提示：

- `TreeNode.val == -1`
- 二叉树的高度不超过 `20`
- 节点的总数在 `[1, 10^4]` 之间
- 调用 `find()` 的总次数在 `[1, 10^4]` 之间
- `0 <= target <= 10^6`

2. 比赛时实现

用递归恢复值，用哈希表保存所有值，find时直接查哈希表，因为节点数众多，所以空间消耗很大

```
class FindElements {
public:
    TreeNode* root;
    unordered_set<int> s;
    void recover(TreeNode* root, int val){
        if(!root) return;
        root->val = val;
        s.insert(val);
        recover(root->left, 2*val+1);
        recover(root->right, 2*val+2);
    }
    FindElements(TreeNode* root) {
        recover(root, 0);
    }
    bool find(int target) {
        if(s.count(target) <= 0)
            return false;
        else
            return true;
    }
};
```

3. 自我改进

用层序遍历的方法恢复，则可以同时得到从小到大排序的节点值数组，在查找时使用二分查找，因为没用递归，所以时间和空间都节省了近一半

```
class FindElements {
public:
```

```
    TreeNode* root = NULL;
    vector<int> v;
    FindElements(TreeNode* root) {
        if(!root) return;
        root->val = 0;
        v.push_back(0);
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            TreeNode* cur = q.front();
            q.pop();
            if(cur->left){
                cur->left->val = 2 * cur->val + 1;
                v.push_back(cur->left->val);
                q.push(cur->left);
            }
            if(cur->right){
                cur->right->val = 2 * cur->val + 2;
                v.push_back(cur->right->val);
                q.push(cur->right);
            }
        }
        this->root = root;
    }

    bool find(int target) {
        int l = 0;
        int r = v.size()-1;
        if(target < v[l] || target > v[r])
            return false;
        while(l <= r){
            int mid = l + (r - l) / 2;
            if(v[mid] == target)
                return true;
            else if(v[mid] < target)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return false;
    }
};
```

1262.可被三整除的最大和

1. 题目描述

   给你一个整数数组 `nums`，请你找出并返回能被三整除的元素最大和。

   示例 1：

```
输入：nums = [3,6,5,1,8]
输出：18
解释：选出数字 3，6，1 和 8，它们的和是 18（可被 3 整除的最大和）。
```

示例 2：

```
输入：nums = [4]
输出：0
解释：4 不能被 3 整除，所以无法选出数字，返回 0。
```

示例 3：

```
输入：nums = [1,2,3,4,4]
输出：12
解释：选出数字 1，3，4 以及 4，它们的和是 12（可被 3 整除的最大和）。
```

提示：

- `1 <= nums.length <= 4 * 10^4`
- `1 <= nums[i] <= 10^4`

2. 最优解法
   - 暴力法是找出所有和的情况，然后遍历一遍，取%3==0的最大值
   - 优化思路是：我们只需要动态更新遍历过的元素之和中 n%3==0，n%3==1，n%3==2的三个最大值即可舍弃其它和的情况

```
class Solution {

public:
    int maxSumDivThree(vector<int>& nums) {
    vector<int> ans(3, 0);//ans[0],ans[1],ans[2]分别保存遍历过的元素的和中%3==0，1，2的
最大值
    vector<int>temp(3, 0);
    for (auto num : nums) { //遍历nums
        for (auto a : ans) {  //将ans中每个元素与num求和
            if ((num + a) % 3 == 0) temp[0] = max(num + a, temp[0]); //如果当前和
%3==0  且大于之前的temp[0]则更新temp[0]
            else if ((num + a) % 3 == 1)temp[1] = max(num + a, temp[1]);//同上
            else if ((num + a) % 3 == 2)temp[2] = max(num + a, temp[2]);//同上
        }
        ans = temp;//将修正过的temp赋给ans
    }
    return ans[0];//完成遍历返回a[0]即可
   }
};
```

1263.推箱子

1. 题目描述

推箱子」是一款风靡全球的益智小游戏，玩家需要将箱子推到仓库中的目标位置。

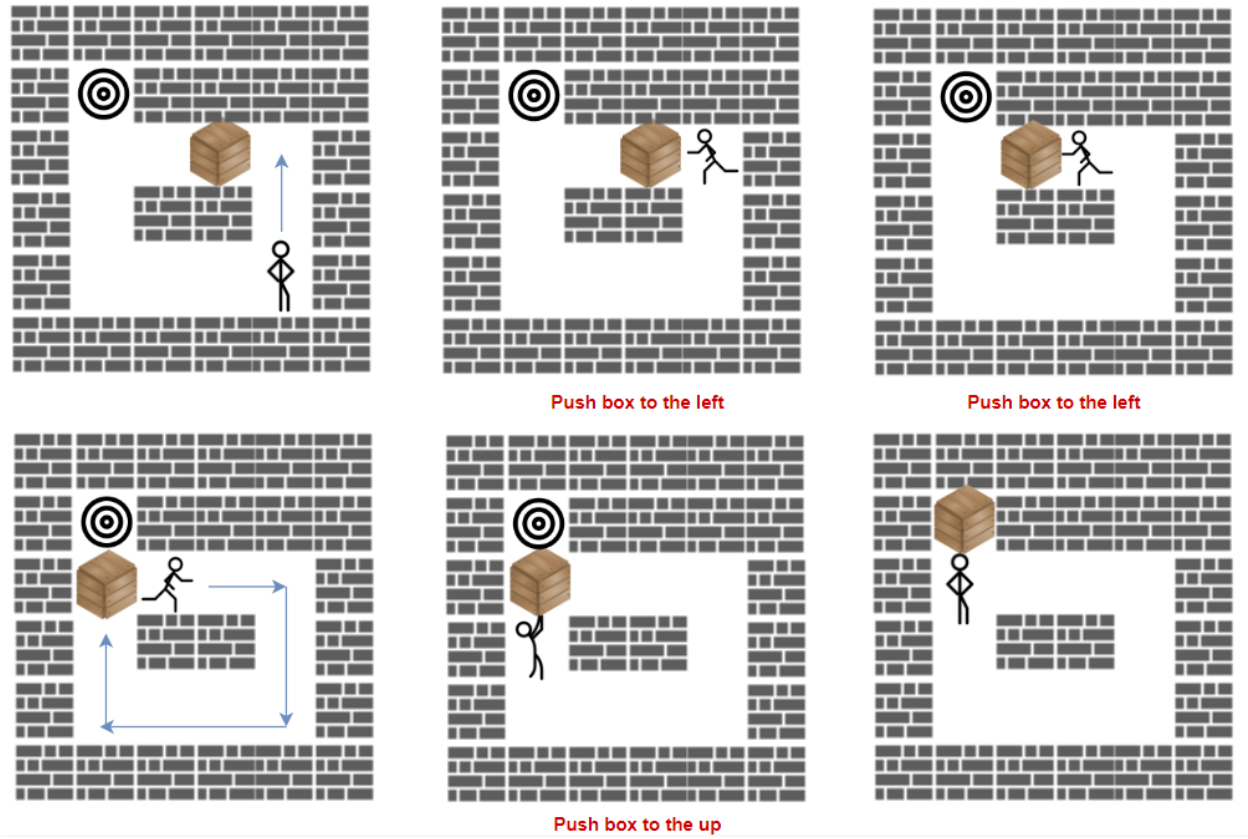
游戏地图用大小为 `n * m` 的网格 `grid` 表示，其中每个元素可以是墙、地板或者是箱子。

现在你将作为玩家参与游戏，按规则将箱子 `'B'` 移动到目标位置 `'T'` ：

- 玩家用字符 `'S'` 表示，只要他在地板上，就可以在网格中向上、下、左、右四个方向移动。
- 地板用字符 `'.'` 表示，意味着可以自由行走。
- 墙用字符 `'#'` 表示，意味着障碍物，不能通行。
- 箱子仅有一个，用字符 `'B'` 表示。相应地，网格上有一个目标位置 `'T'`。
- 玩家需要站在箱子旁边，然后沿着箱子的方向进行移动，此时箱子会被移动到相邻的地板单元格。记作一次「推动」。
- 玩家无法越过箱子。

返回将箱子推到目标位置的最小 **推动** 次数，如果无法做到，请返回 `-1`。

示例 1：




```
输入：grid = [["#","#","#","#","#","#"],
              ["#","T","#","#","#","#"],
              ["#",".",".","B",".","#"],
              ["#",".","#","#",".","#"],
              ["#",".",".",".","S","#"],
              ["#","#","#","#","#","#"]]
输出：3
解释：我们只需要返回推箱子的次数。
```

示例 2：

```
输入：grid = [["#","#","#","#","#","#"],
              ["#","T","#","#","#","#"],
              ["#",".",".","B",".","#"],
              ["#","#","#","#",".","#"],
              ["#",".",".",".","S","#"],
              ["#","#","#","#","#","#"]]
输出：-1
```

示例 3：

```
输入：grid = [["#","#","#","#","#","#"],
              ["#","T",".",".","#","#"],
              ["#",".","#","B",".","#"],
              ["#",".",".",".",".","#"],
              ["#",".",".",".","S","#"],
              ["#","#","#","#","#","#"]]
输出：5
解释：向下、向左、向左、向上再向上。
```

示例 4：

```
输入：grid = [["#","#","#","#","#","#","#"],
              ["#","S","#",".","B","T","#"],
              ["#","#","#","#","#","#","#"]]
输出：-1
```

提示：

- `1 <= grid.length <= 20`
- `1 <= grid[i].length <= 20`
- `grid` 仅包含字符 `'.'`，`'#'`，`'S'` ，`'T'`，以及 `'B'`。
- `grid` 中 `'S'`，`'B'` 和 `'T'` 各只能出现一个。

2. 比赛时实现

   只实现了人可以跨越箱子的一重BFS（箱子），貌似用两重BFS（一重箱子一重人）也可以不超时

3. 最优解法

   对于二维矩阵中求最短路的问题，我们一般可以使用广度优先搜索 + 队列的方法解决。在本题中，如果没有玩家，而是箱子每一步可以自行向四个方向移动一格，那么我们就可以从箱子的初始位置开始，使用二元组 (x, y)，即箱子的坐标表示状态，通过广度优先搜索，直至搜索到目标位置。

   然而本题需要玩家推动箱子，因此只用二元组 (x, y) 表示一个状态是不够的，因为玩家的可移动范围是随着箱子位置的变化而变化的。因此我们可以考虑用四元组 (bx, by, mx, my) 表示一个状态，其中 (bx, by) 表示箱子的位置，(mx, my) 表示玩家的位置。

   对于当前的状态 (bx, by, mx, my)，它可以向最多四个新状态进行搜索，即将玩家 (mx, my) 向四个方向移动一格。假设移动的方向为 (dx, dy)，那么玩家的新位置为 (mx + dx, my + dy)。如果该位置为地板且箱子不在此处，那么根据题目要求，玩家移动到新位置不计入推动次数。如果该位置为箱子 (bx, by)，那么箱子可能的新位置为 (bx + dx, bx + dy)，如果该位置为地板，那么箱子会被推动，根据题目要求，计入一次推动次数。当箱子到达了目标位置，我们就得到了最小推动次数。

注意到上面的方法存在一个小问题，假设状态 S 可以向两个状态 S1 和 S2 进行搜索，S1 中箱子被推动，S2 中箱子未被推动。由于 S1 相较于 S2 先进入队列，因此我们并没有按照广度优先搜索的要求，先搜索小状态，后搜索大状态。因**此当我们有搜索队列 q 时，对于 q 中的每一个状态 S 可以得到的新状态 Sx，如果 Sx 中箱子未被推动，那么可以直接将 Sx 加入队列末尾；如果 Sx 中箱子被推动，那么需要将 Sx 加入一个新的队列 nq 中。**可以发现，**q 中所有的状态都有着相同的推动次数 k，而 nq 中所有的状态都有着相同的推动次数 k + 1。**在 q 中所有状态都搜索完毕，即 q 为空时，我们将 nq 赋予 q，再开始新的一轮广度优先搜索。这样我们就保证了先搜索小状态，后搜索大状态的策略。

```
struct Dwell {//记录当前状态，包括人和箱子的位置
    int box_x, box_y;
    int man_x, man_y;
    Dwell(int _bx, int _by, int _mx, int _my): box_x(_bx), box_y(_by), man_x(_mx),
man_y(_my) {}
};

class Solution {
private:
    static constexpr int dirs[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

public:
    int minPushBox(vector<vector<char>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        bool dist[m][n][m][n];//该状态是否被访问
        memset(dist, false, sizeof(dist));

        int box_x, box_y;
        int start_x, start_y;
        int end_x, end_y;
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == 'B') {
                    box_x = i;
                    box_y = j;
                    grid[i][j] = '.';
                }
                else if (grid[i][j] == 'S') {
                    start_x = i;
                    start_y = j;
                    grid[i][j] = '.';
                }
                else if (grid[i][j] == 'T') {
                    end_x = i;
                    end_y = j;
                    grid[i][j] = '.';
                }
            }
        }

        queue<Dwell> q;
        q.emplace(box_x, box_y, start_x, start_y);//比push节省内存
        dist[box_x][box_y][start_x][start_y] = true;
```

```
        int ans = 0;
        while (!q.empty()) {
            ans++;
            queue<Dwell> nq;//下一轮BFS队列
            while (!q.empty()) {
                Dwell cur = q.front();
                q.pop();
                for (int i = 0; i < 4; ++i) {//人往四个方向走
                    int nxt_x = cur.man_x + dirs[i][0];
                    int nxt_y = cur.man_y + dirs[i][1];
                    if (nxt_x >= 0 && nxt_x < m && nxt_y >= 0 && nxt_y < n) {
                        if (cur.box_x == nxt_x && cur.box_y == nxt_y) {
                            //人走到箱子上，判断能否推动
                            int nxt_box_x = cur.box_x + dirs[i][0];
                            int nxt_box_y = cur.box_y + dirs[i][1];
                            if (nxt_box_x >= 0 && nxt_box_x < m
                                && nxt_box_y >= 0 && nxt_box_y < n) {
                                if (grid[nxt_box_x][nxt_box_y] == '.' &&
                                    dist[nxt_box_x][nxt_box_y][nxt_x][nxt_y] ==
false) {

                                    //箱子可以被推动且推动后达到的状态未被访问
                                    //新状态入下一轮BFS队列
                                    nq.emplace(nxt_box_x, nxt_box_y, nxt_x, nxt_y);
                                    dist[nxt_box_x][nxt_box_y][nxt_x][nxt_y] =
true;

                                    if (nxt_box_x == end_x && nxt_box_y == end_y)

//到达目标

                                        return ans;
                                }
                            }
                        }
                        else {//人未走到箱子上
                            if (grid[nxt_x][nxt_y] == '.' &&
                                dist[cur.box_x][cur.box_y][nxt_x][nxt_y] == false)
{

                                //人走到平地且移动后状态未访问，新状态入本轮BFS队列
                                q.emplace(cur.box_x, cur.box_y, nxt_x, nxt_y);
                                dist[cur.box_x][cur.box_y][nxt_x][nxt_y] = true;
                            }
                        }
                    }
                }
            }
            q = nq;//更新队列
        }
    }
};
```