LeetCode 1380. 矩阵中的幸运数

1380. 矩阵中的幸运数

难度 简单 6

给你一个 `m * n` 的矩阵，矩阵中的数字 **各不相同** 。请你按 **任意** 顺序返回矩阵中的所有幸运数。

幸运数是指矩阵中满足同时下列两个条件的元素：

- 在同一行的所有元素中最小
- 在同一列的所有元素中最大

示例 1：

```
输入：matrix = [[3,7,8],[9,11,13],[15,16,17]]
输出：[15]
解释：15 是唯一的幸运数，因为它是其所在行中的最小值，也是所在列中的最大值。
```

示例 2：

```
输入：matrix = [[1,10,4,2],[9,3,8,7],[15,16,17,12]]
输出：[12]
解释：12 是唯一的幸运数，因为它是其所在行中的最小值，也是所在列中的最大值。
```

示例 3：

```
输入：matrix = [[7,8],[1,2]]
输出：[7]
```

```cpp
class Solution {
public:
    vector<int> luckyNumbers (vector<vector<int>>& matrix) {
        int n = matrix.size(),m = matrix[0].size();
        vector<int> res;

        for(int i = 0;i < n;i ++)
        {
            for(int j = 0;j < m;j ++)
            {
                bool is_lucky = true;
                for(int k = 0; k < m;k ++)
                {
                    if(matrix[i][j] > matrix[i][k])
                    {
                        is_lucky = false;
                        break;
```

```
                    }

                }
                if(is_lucky)
                {
                    for(int k = 0; k < n;k ++)
                        if(matrix[i][j] < matrix[k][j])
                        {
                            is_lucky = false;
                            break;
                        }
                }
                if(is_lucky) res.push_back(matrix[i][j]);
            }
        }
        return res;
    }
};
```

LeetCode 1381. 设计一个支持增量操作的栈

1381. 设计一个支持增量操作的栈

难度 中等 12

请你设计一个支持下述操作的栈。

实现自定义栈类 `CustomStack` ：

- `CustomStack(int maxSize)` ：用 `maxSize` 初始化对象，`maxSize` 是栈中最多能容纳的元素数量，栈在增长到 `maxSize` 之后则不支持 `push` 操作。
- `void push(int x)` ：如果栈还未增长到 `maxSize` ，就将 `x` 添加到栈顶。
- `int pop()` ：返回栈顶的值，或栈为空时返回 **-1** 。
- `void inc(int k, int val)` ：栈底的 `k` 个元素的值都增加 `val` 。如果栈中元素总数小于 `k` ，则栈中的所有元素都增加 `val` 。

示例:

```
输入：
["CustomStack","push","push","pop","push","push","push","i
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],
[]]
输出：
[null,null,null,2,null,null,null,null,null,103,202,201,-1]
解释：
CustomStack customStack = new CustomStack(3); // 栈是空
的 []
customStack.push(1);                          // 栈变为
[1]
customStack.push(2);                          // 栈变为
[1, 2]
customStack.pop();                            // 返回 2
--> 返回栈顶值 2，栈变为 [1]
customStack.push(2);                          // 栈变为
[1, 2]
customStack.push(3);                          // 栈变为
[1, 2, 3]
customStack.push(4);                          // 栈仍然
```

```cpp
class CustomStack {
public:

    vector<int> stk;
    int top;

    CustomStack(int maxSize) {
        stk = vector<int>(maxSize);
        top = 0;
    }

    void push(int x) {
```

```
        if(top == stk.size()) return;
        stk[top ++ ] = x;
    }

    int pop() {
        if(!top) return -1;
        return stk[-- top];
    }

    void increment(int k, int val) {
        for(int i = 0;i < k && i < top;i ++)
            stk[i] += val;
    }
};

/**
 * Your CustomStack object will be instantiated and called as such:
 * CustomStack* obj = new CustomStack(maxSize);
 * obj->push(x);
 * int param_2 = obj->pop();
 * obj->increment(k,val);
 */
```

LeetCode 1382. 将二叉搜索树变平衡

1382. 将二叉搜索树变平衡

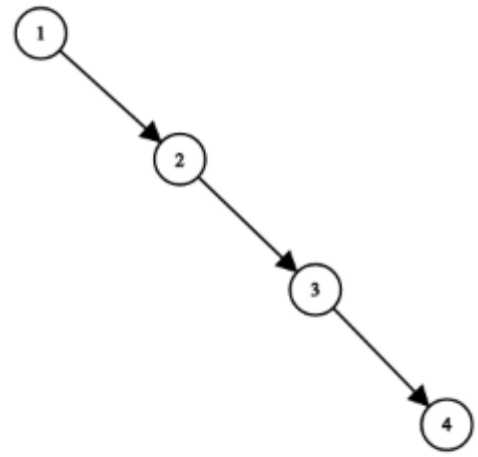
难度 中等 10

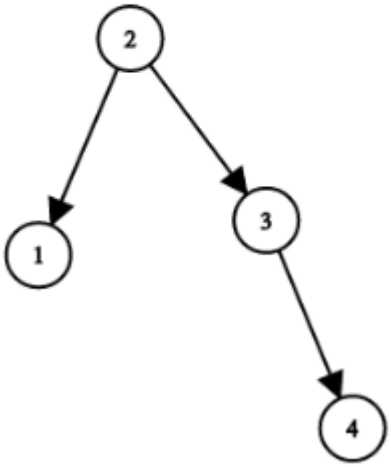
给你一棵二叉搜索树，请你返回一棵 **平衡后** 的二叉搜索树，新生成的树应该与原来的树有着相同的节点值。

如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过 1 ，我们就称这棵二叉搜索树是 **平衡的** 。

如果有多种构造方法，请你返回任意一种。

示例：






```
 /*
 中序遍历--->有序数组--->平衡二叉树
 中间节点是根节点，左边递归创建左子树，右边递归创建右子树
节点个数m是奇数时，左右都为(m - 1)/2个，差为0
节点个数m是偶数时，左(m - 1)/2 - 1 右都为(m - 1)/2
 */
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
```

```
 */
class Solution {
public:
    vector<TreeNode*> node;
    TreeNode* balanceBST(TreeNode* root) {
        dfs(root);

        return build(0,node.size() - 1);
    }

    TreeNode * build(int l,int r)
    {
        if(l > r) return NULL;
        int mid = l + r >> 1;
        node[mid]->left = build(l,mid - 1);
        node[mid]->right = build(mid + 1,r);
        return node[mid];
    }
    void dfs(TreeNode* root)
    {
        if(!root) return;
        dfs(root->left);
        node.push_back(root);
        dfs(root->right);
    }
};
```