70. Climbing Stairs

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

思路：
这是一个动态规划问题，假如我们知道了爬 n 阶楼梯和爬 n+1 阶楼梯的种类数，分别设为 dp(n)和 dp(n+1)，那么爬 n+2 阶楼梯的时候，要么从第 n 阶爬 2 阶到达，要么从第 n+1 阶爬 1 阶到达。因此，公式为 dp(n+2) = dp(n) + dp(n+1)。

代码：

```
class Solution {
public:
    int climbStairs(int n) {
        if (n == 0) return 0;
        vector<int> dp;
        dp.push_back(1);
        dp.push_back(2);
        for (int i = 2;i < n;i++)
            dp.push_back(dp[i - 1] + dp[i - 2]);
        return dp[n - 1];
    }
};
```

73. Set Matrix Zeroes

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Follow up:

Did you use extra space?

A straight forward solution using O(*mn*) space is probably a bad idea.

A simple improvement uses O(*m* + *n*) space, but still not the best solution.

Could you devise a constant space solution?

思路：
O(*mn*)的方法是建立一个新的 m*n 的矩阵，从原矩阵中粘贴数值。

O(*m* + *n*)在上面基础上做了优化，用 m 大小的空间记录 m 行中哪行为 0，用 n 大小的空间记录 n 列中哪列为 0.
而如果用常数空间，可根据 O(m+n)的思路，依然是用 O(m+n)的空间记录哪行哪列为 0，但是使用的是 m*n 矩阵里的空间。因此，可以占用第 0 行和第 0 列的空间来保存。其中有个问题是，matrix[0][0]位置，如果保存的是第 0 行是否有 0，那么就需要额外的一个空间记录第 0 列是否有 0，因此实际消耗的空间为 1.

代码：

```
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {
        if(matrix.empty()) return;
        int col0 = 1;
        int rows = matrix.size();
        int cols = matrix[0].size();

        for (int i = 0; i < rows; i++) {
            if (matrix[i][0] == 0) col0 = 0;
            for (int j = 1; j < cols; j++)
                if (matrix[i][j] == 0)
                    matrix[i][0] = matrix[0][j] = 0;
        }

        for (int i = rows - 1; i >= 0; i--) {
            for (int j = cols - 1; j >= 1; j--)
                if (matrix[i][0] == 0 || matrix[0][j] == 0)
                    matrix[i][j] = 0;
            if (col0 == 0) matrix[i][0] = 0;
        }
    }
};
```

86. Partition List

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given `1->4->3->2->5->2` and *x* = 3,

return `1->2->2->4->3->5`.

思路：
本题为链表操作的题目。建立两个带表头的空链表，从原链表的起始位置一直走到尾，将小于 x 的节点接入链表 1，将大于等于 x 的节点接入链表 2，然后将链表 1 的尾部连接链表 2 的首部，链表 2 的尾部设为 NULL，链表处理完毕。

代码：

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        if (!head) return head;
        ListNode *head1 = new ListNode(0), *head2 = new ListNode(0);
        ListNode *tmp1 = head1, *tmp2 = head2;
        ListNode *tmp = head;
        while (tmp != NULL) {
            if (tmp->val < x) {
                tmp1->next = tmp;
                tmp1 = tmp1->next;
            }
            else {
                tmp2->next = tmp;
                tmp2 = tmp2->next;
            }
            tmp = tmp->next;
        }
        tmp2->next = NULL;
        tmp1->next = head2->next;
        return head1->next;
    }
};
```

93. Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given `"25525511135"`,

return `["255.255.11.135", "255.255.111.35"]`. (Order does not matter)

思路：

该题目是典型的 DFS 类型。终止情况为四段地址已经解析完或者剩余的数字长度不符合要求，即长度大于 3*residual 或者长度小于 1*residual。当解析完某段 ip，则将剩余的段数以及剩余的字符串传入下一层。

代码：

```
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        string ip;
        dfs(s,0,0,ip,result);  //paras:string s,start index of s,step(from0-3),intermediate ip,final result
        return result;
    }
    void dfs(string s,int start,int step,string ip,vector<string>& result){
        if(start==s.size()&&step==4){
            ip.erase(ip.end()-1); //remove the last '.' from the last decimal number
            result.push_back(ip);
            return;
        }
        if(s.size()-start>(4-step)*3) return;
        if(s.size()-start<(4-step)) return;
        int num=0;
        for(int i=start;i<start+3;i++){
            num=num*10+(s[i]-'0');
            if(num<=255){
                ip+=s[i];
                dfs(s,i+1,step+1,ip+'.',result);
            }
            if(num==0) break;
        }
    }
};
```

98. Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

思路：
本题考察的是对树的遍历。如果是采用前序遍历，只能保证某节点的左孩子和右孩子满足条件，但不能保证节点左孩子的右孩子或者节点右孩子的左孩子满足条件。同理，后序遍历时，也不能保证左子树的最大值小于根节点，或者右子树的最小值大于根节点。因此，只有中序遍历可以验证 BST，因为中序遍历时访问根节点前的最后一个节点一定是左子树中的最大值。

代码：

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, NULL, NULL);
    }

    bool isValidBST(TreeNode* root, TreeNode* minNode, TreeNode* maxNode) {
        if(!root) return true;
        if(minNode && root->val <= minNode->val || maxNode && root->val >= maxNode->val)
            return false;
        return isValidBST(root->left, minNode, root) && isValidBST(root->right, root, maxNode);
    }
};
```