# Contents



# 1 字符串

## 1.1 回文数

题目：判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

示例 1:

输入: 121
输出: true
示例 2:

输入: -121
输出: false
解释: 从左向右读，为 -121 。 从右向左读，为 121- 。因此它不是一个回文数。
示例 3:

输入: 10
输出: false
解释: 从右向左读，为 01 。因此它不是一个回文数。

解答

为了避免数字反转可能导致的溢出问题，为什么不考虑只反转数字的一半？毕竟，如果该数字是回文，其后半部分反转后应该与原始数字的前半部分相同。

例如，输入 1221，我们可以将数字 “1221” 的后半部分从 “21” 反转为 “12”，并将其与前半部分 “12” 进行比较，因为二者相同，我们得知数字 1221 是回文。

- 特判所有负数都不可能是回文，例如：-123 不是回文，因为 - 不等于 3
- 反转

对于数字 1221，如果执行 1221 % 10，我们将得到最后一位数字 1，要得到倒数第二位数字，我们可以先通过除以 10 把最后一位数字从 1221 中移除，1221 / 10 = 122，再求出上一步结果除以 10 的余数，122 % 10 = 2，就可以得到倒数第二位数字。如果我们把最后一位数字乘以 10，再加上倒数第二位数字，1 * 10 + 2 = 12，就得到了我们想要的反转后的数字。如果继续这个过程，我们将得到更多位数的反转数字。

所以，每次把上一次的数字 *10，加上这一次的最后一位数字，然后 x/=10

现在的问题是，我们如何知道反转数字的位数已经达到原始数字位数的一半？

- 终止

我们将原始数字除以 10，然后给反转后的数字乘上 10，所以，当除完的原始数字小于反转后的数字时，就意味着我们已经处理了一半位数的数字。

当数字长度为奇数时，我们可以通过 revertedNumber/10 去除处于中位的数字。例如，当输入为 12321 时，在 while 循环的末尾我们可以得到 x = 12，revertedNumber = 123，由于处于中位的数字不影响回文（它总是与自己相等），所以我们可以简单地将其去除。

```
class Solution {
public:
    bool isPalindrome(int x) {
        if(x < 0 || (x % 10 == 0 && x != 0)) {
            return false;
        }
        int revertedNumber = 0;
        while(x > revertedNumber) {
            revertedNumber = revertedNumber * 10 + x % 10;
            x /= 10;
        }
        return (x == revertedNumber || x == revertedNumber / 10);
    }
};
```

## 1.2 反转整数

题目:

给定一个 32 位有符号整数，将整数中的数字进行反转。

示例：

```
示例 1:

输入: 123
输出: 321
 示例 2:

输入: -123
输出: -321
示例 3:
```

输入： 120
输出： 21

注意:

假设我们的环境只能存储 32 位有符号整数，其数值范围是 [−231, 231 − 1]。根据这个假设，如果反转后的整数溢出，则返回 0。

解答

写一个 valid 函数，记得参数变成 long，然后去看这个 long 是不是在 int32 的范围里

```
class Solution {
public:
    bool valid(long x) { // 注意，这里要是 long
        if (x > 0) {
            if (x >= pow(2, 31) -1)
                return false;
        }
        if (x < 0) {
            if (x <= -pow(2, 31)) {
                return false;
            }
        }
        return true;
    }

    int reverse(int x) {
        long tmp = 0;
        if (!valid(x)) {
            return 0;
        }
        bool flag = true;
        if (x < 0) {
            x = -x;
            flag = false;
        }
        while (x != 0) {
            tmp *= 10;
            tmp += x % 10;
            x /= 10;
        }
        if (flag == false) {
            tmp = -tmp;
        }
        if (valid(tmp)) {
            return tmp;
        }
        return 0;
    }
};
```

### 1.3 罗马数字转整数

题目罗马数字包含以下七种字符: I，V，X，L，C，D 和 M。

| 字符 | 数值 |
| --- | --- |
| I | 1 |
| V | 5 |
| X | 10 |
| L | 50 |
| C | 100 |
| D | 500 |
| M | 1000 |

例如，罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

```
示例 1:

输入: "III"
输出: 3
示例 2:

输入: "IV"
输出: 4
示例 3:

输入: "IX"
输出: 9
示例 4:

输入: "LVIII"
输出: 58
解释: L = 50, V= 5, III = 3.
示例 5:

输入: "MCMXCIV"
输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

解答：

- 第一，如果当前数字是最后一个数字，或者之后的数字比它小的话，则加上当前数字
- 第二，其他情况则减去这个数字 (例如，IV，看到 I 的时候就是减去 I，然后到 V 就是加 V; XL，看到 X 的时候就是-X，然后到 L 就是加 L)

```
class Solution {
public:
    int romanToInt(string s) {
        unordered_map<char, int> x_map;
        x_map.insert(std::make_pair('I', 1));
        x_map.insert(std::make_pair('V', 5));
        x_map.insert(std::make_pair('X', 10));
        x_map.insert(std::make_pair('L', 50));
        x_map.insert(std::make_pair('C', 100));
```

```
        x_map.insert(std::make_pair('D', 500));
        x_map.insert(std::make_pair('M', 1000));
        int res = 0;
        for (int i = 0; i < s.size(); ++i) {
            cout << i << s[i] << endl;
            int val = x_map[s[i]];
            if (i == s.size() - 1 || x_map[s[i+1]] <= x_map[s[i]]) {
                res += val;
            } else {
                res -= val;
            }
        }
        return res;
    }
};
```

### 1.4 有效的括号

题目：给定一个只包括 ‘(’，‘)’，‘{’，‘}’，‘[’，’]’ 的字符串，判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。
- 注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入: "{[]}"
输出: true
```

解答：注意一定要先判断 st.size()>0 再取 top，不然会出错，其他的是常规操作

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        unordered_map<char, char> mp;
        mp.insert(std::make_pair('}', '{'));
        mp.insert(std::make_pair(']', '['));
```

```
        mp.insert(std::make_pair(')', '('));
        for (int i = 0; i < s.size(); i++) {
            if (mp.find(s[i]) != mp.end() &&
                st.size() > 0 &&
                mp[s[i]] == st.top()) {
                st.pop();
            } else {
                st.push(s[i]);
            }
        }
        if (st.size() == 0) return true;
        return false;
    }
};
```

# 2 数组

## 2.1 移除元素

题目：给定一个数组 nums 和一个值 val，你需要原地移除所有数值等于 val 的元素，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

```
给定 nums = [3,2,2,3], val = 3,

函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,1,2,2,3,0,4,2], val = 2,

函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。

注意这五个元素可为任意顺序。

你不需要考虑数组中超出新长度后面的元素。
```

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
```

}

解答：

关键：保留两个指针 **i** 和 **j**，其中 **i** 是慢指针，**j** 是快指针。用 j 来遍历数组，当 nums[j]!=val 时，把 num[j] 赋值给 num[i]，然后移动 i

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int i = 0, j = 0;
        for (;j < nums.size(); ++j) {
            if (nums[j] != val) {
                nums[i] = nums[j];
                ++i;
            }
        }
        return i;
    }
};
```

## 2.2 两数之和

题目：

给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。

你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。

示例**:**

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9 所以返回 [0, 1]

解法：

用一个 map，key 是元素值，value 是 idx 看新来的这个元素的目标值（tgt - nums[i]）在不在 map 里，在的话把它的 value 拿出来就行了。。

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> res;
        unordered_map<int, int> map;
        for (int i = 0; i < nums.size(); ++i) {
            const int& tgt_val = target - nums[i];
            if (map.find(tgt_val) != map.end()) {
                res.push_back(map[tgt_val]);
                res.push_back(i);
                return res;
            } else {
                map.insert(std::make_pair(nums[i], i));
            }
        }
    }
};
```

## 2.3 无重复字符的最长子串

题目：给定一个字符串，找出不含有重复字符的最长子串的长度。

```
示例 1:
输入: "abcabcbb"
输出: 3
解释: 无重复字符的最长子串是 "abc"，其长度为 3。

示例 2:
输入: "bbbbb"
输出: 1
解释: 无重复字符的最长子串是 "b"，其长度为 1。

示例 3:
输入: "pwwkew"
输出: 3
```

解释: 无重复字符的最长子串是 “wke”，其长度为 3。请注意，答案必须是一个子串，“pwke” 是一个子序列而不是子串。

解答：

- 用滑动窗口，最终要返回的 size 就是这个窗口的大小 ( j-i+1 )，而 set 只是为了存储这个窗口里的元素
- 对于 j，如果在 set 中没找到这个元素 s[j]，更新 res，并把 s[j] 扔进 set，再 ++j；
- 如果找到了，那么，说明 s[j] 在 set 里了，这时候需要把开头的元素 s[i] 从 set 里删了，把 i++，窗口的开始往右移一格

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        set<char> set_char;
        int res = 0;
        int tmp = 0;
        for (int i = 0, j = 0; i < s.size() && j < s.size(); ) {
            if (set_char.find(s[j]) != set_char.end()) {
                set_char.erase(s[i]);
                ++i;
            } else {
                if (j - i + 1 > res) {
                    res = j - i + 1;
                }
                set_char.insert(s[j]);
                ++j;
            }
        }

        return res;
    }
};
```

## 2.4 两个排序数组的中位数

题目：

给定两个大小为 m 和 n 的有序数组 nums1 和 nums2 。

请找出这两个有序数组的中位数。要求算法的时间复杂度为 O(log (m+n)) 。

你可以假设 nums1 和 nums2 不同时为空。

示例 1:

```
nums1 = [1, 3]
nums2 = [2]

中位数是 2.0
```

示例 2:

```
nums1 = [1, 2]
nums2 = [3, 4]

中位数是 (2 + 3)/2 = 2.5
```

解答：

方法 1：

先归并两个数组，再取中点，归并的复杂度是 O(m+n)，参考第 88 题https://leetcode-cn.com/problems/merge-sorted-array/description/

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        vector<int> tmp;
        int m = nums1.size();
        int n = nums2.size();
        int total_size = n + m;
        tmp.resize(total_size);
        int j = n - 1;
        int i = m - 1;
        while (j >= 0) {
            if (i < 0) {
                // 如果 i 数组遍历完了，要把 j 数据剩下的全部拷过来，记住是<j+1
                for(int k = 0; k < j + 1; ++k) {
                    tmp[k] = nums2[k];
                }
                break;
            }
            if (nums2[j] > nums1[i]) {
                tmp[i + j + 1] = nums2[j];
                j--;
            } else {
                tmp[i + j + 1] = nums1[i];
                i--;
            }
        }
        if (j < 0) {
            for(int k = 0; k < i + 1; ++k) {
                tmp[k] = nums1[k];
            }
        }
        // 以上是归并两个数组的方法
        if (total_size % 2 != 0) {
```

```
            return tmp[total_size / 2];
        } else {
            return (tmp[total_size / 2 - 1] + tmp[total_size / 2]) *1.0 / 2;
        }
    }
};
```

方法 2：递归

参考https://leetcode-cn.com/problems/median-of-two-sorted-arrays/solution/

## 2.5 合并两个有序数组

题目：

给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使得 num1 成为一个有序数组。

说明:

- 初始化 nums1 和 nums2 的元素数量分别为 m 和 n。
- 你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

示例:

```
输入：
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

输出：[1,2,2,3,5,6]
```

解答：

提示中已经给出，假设 array1 有足够的空间了，于是我们不需要额外构造一个数组，并且可以从后面不断地比较元素进行合并。

- 比较 array2 与 array1 中最后面的那个元素，把最大的插入第 m+n 位
- 改变数组的索引，再次进行上面的比较，把最大的元素插入到 array1 中的第 m+n-1 位。
- 循环一直到结束。循环结束条件：当 index1 或 index2 有一个小于 0 时，此时就可以结束循环了。如果 index2 小于 0，说明目的达到了。如果 index1 小于 0，就把 array2 中剩下的前面的元素都复制到 array1 中去就行。

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int j = n - 1;
        int i = m - 1;
        while (j >= 0) {
            if (i < 0) {
                // 如果 i 数组遍历完了，要把 j 数据剩下的全部拷过来，记住是<j+1
                for(int k = 0; k < j + 1; ++k) {
                    nums1[k] = nums2[k];
                }
                break;
            }
            if (nums2[j] > nums1[i]) {
                nums1[i + j + 1] = nums2[j];
                j--;
            } else {
                nums1[i + j + 1] = nums1[i];
                i--;
            }
```

```
        }
    }
};
```

# 3 链表

## 3.1 删除排序链表中的重复元素

题目：

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

输入: 1->1->2
输出: 1->2
示例 2:

输入: 1->1->2->3->3
输出: 1->2->3

解答：

方法一：**set+** 双指针双指针，记得理清楚什么时候两个指针后移就行，记得用 new ListNode(xx)，还有 insert 的时间点

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == NULL) {
            return NULL;
        }
        ListNode* res = new ListNode(head->val);
        ListNode* j = head->next;
        ListNode* i = res;
        set<int> set_v;
        set_v.insert(head->val);
        for (; j != NULL; j = j->next) {
            cout << j->val << "xx" << endl;
            if (set_v.find(j->val) == set_v.end()) {
                cout << j->val << "xx2" << endl;
                i->next = new ListNode(j->val);
                i = i->next;
                set_v.insert(j->val);
            }
        }
        return res;
```

```
    }
};
```

方法二：

考虑到这题限制了是排序数组，所以其实可以不用 set，重复的数字是连续出现的。。。而且，连 new 都不用，直接在原链表上改 next 就行，如果 next= 当前值，就把 next 指向 next 的 next

注意：只有当 next!= 当前值时，才 i = i->next；不然超过两个连续的重复就干不掉了，例如 [1,1,1,1,2]

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* i = head;
        while (i != NULL && i->next != NULL) {
            if(i->val == i->next->val) {
                i->next = i->next->next;
            } else {
                i = i->next;
            }
        }
        return head;
    }
};
```

### 3.2 两数相加

题目：给定两个非空链表来表示两个非负整数。位数按照逆序方式存储，它们的每个节点只存储单个数字。将两数相加返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例：

```
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

解答：

- 搞一个 dummyhead，然后每一次 while 的时候，填充他的 next，最后返回出是 dummyhead 的 next
- 要 x->next 之前先判断 x!=NULL(不是判断 x->next!=NULL)
- while 的条件是或，处理两个链表不等长的情况
- while 外面，如果还有 carry，要再 new 一个

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
```

```
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        int carry = 0;
        ListNode* dummy_head = new ListNode(0);
        ListNode* tmp = dummy_head;
        ListNode* ptr1 = l1;
        ListNode* ptr2 = l2;
        while (ptr1 != NULL || ptr2 != NULL) {
            int val1 = ptr1 != NULL? ptr1->val: 0;
            int val2 = ptr2 != NULL? ptr2->val: 0;
            int sum = val1 + val2 + carry;
            //cout << sum << " " << carry << " " << val1 << " " << val2 << endl;
            carry = sum / 10;
            int remain = sum % 10;
            tmp->next = new ListNode(remain);
            ptr1 = (NULL == ptr1? NULL: ptr1->next);
            ptr2 = (NULL == ptr2? NULL: ptr2->next);
            tmp = tmp->next;
        }
        if (carry > 0) {
            tmp->next = new ListNode(carry);
        }
        return dummy_head->next;
    }
};
```

# 4 树

## 4.1 对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

解答：

方法一：递归

如果同时满足下面的条件，两个树互为镜像：

- 它们的两个根结点具有相同的值。
- 每个树的右子树都与另一个树的左子树镜像对称。

```
/**
 * 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:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (t1 == NULL && t2 == NULL) return true;
        // 如果两个不都是 NULL，这个时候，如果一个是 NULL，另一个不是，那么肯定不镜像！！
        // 如果两个都不是 NULL，那还有可能，可以等下一次递归
        if (t1 == NULL || t2 == NULL) return false;
        return (t1->val == t2->val &&
               isMirror(t1->left, t2->right) &&
               isMirror(t1->right, t2->left));
    }
    bool isSymmetric(TreeNode* root) {
        return isMirror(root, root);
    }
};
```

方法二：迭代

利用队列进行迭代

队列中每两个连续的结点应该是相等的，而且它们的子树互为镜像。最初，队列中包含的是 root 以及 root。该算法的工作原理类似于 BFS，但存在一些关键差异。每次提取两个结点并比较它们的值。然后，将两个结点的左右子结点按相反的顺序插入队列中 (即 t1->left, t2->right, t1->right, t2->left)。

当队列为空时，或者我们检测到树不对称（即从队列中取出两个不相等的连续结点）时，该算法结束。

注意，c++ 的 queue 的 front 不会 pop，要手动 pop

```
/**
 * 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:
    bool isSymmetric(TreeNode* root) {
        queue<TreeNode*> q;
        q.push(root);
        q.push(root);
        while (!q.empty()) {
```

```
            TreeNode* t1 = q.front();
            q.pop();
            TreeNode* t2 = q.front();
            q.pop();
            if (t1 == NULL && t2 == NULL) continue;
            if (t1 == NULL || t2 == NULL) return false;
            if (t1->val != t2->val) return false;
            q.push(t1->left);
            q.push(t2->right);
            q.push(t1->right);
            q.push(t2->left);
        }
        return true;
    }
};
```

## 4.2 二叉树的最大深度

题目：

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例：给定二叉树 [3,9,20,null,null,15,7]，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

解答：

方法一：递归

每往下一层就加 1，体现在走完左右子树的时候，return 的时候加 1，只有这样，最终的深度才会体现出来。。只有一个节点的时候，深度是 1。

时间复杂度 O(N), N 为节点总数，因为需要遍历所有节点

空间复杂度：在最糟糕的情况下，树是完全不平衡的，例如每个结点只剩下左子结点，递归将会被调用 N 次，因此保持调用栈的存储将是 O(N)。但在最好的情况下（树是完全平衡的），树的高度将是 log(N)。因此，在这种情况下的空间复杂度将是 O(log(N))。

```
/**
 * 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:
```

```
    int maxDepth(TreeNode* root) {
        if (root == NULL) {
            return 0;
        } else {
            int left_height = maxDepth(root->left);
            int right_height = maxDepth(root->right);
            return std::max(left_height, right_height) + 1; // 每往下一层就要加 1
        }
    }
};
```

方法 2：迭代（栈）

使用 **DFS** 策略访问每个结点，同时在每次访问时更新最大深度。

从包含根结点且相应深度为 1 的栈开始。然后我们继续迭代：将当前结点弹出栈并推入子结点。每一步都会更新深度。

注意，push 的时候是 cur_depth + 1，而非 depth+1

```
/**
 * 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:

    int maxDepth(TreeNode* root) {
        stack<std::pair<TreeNode*, int> > st;
        if (root != NULL) {
            st.push(std::make_pair(root, 1));
        }
        int depth = 0;
        while (!st.empty()) {
            std::pair<TreeNode*, int> a = st.top();
            TreeNode* cur_root = a.first;
            int cur_depth = a.second;
            st.pop();
            if (cur_root != NULL) {
                depth = std::max(depth, cur_depth);
                st.push(std::make_pair(cur_root->left, cur_depth + 1));
                st.push(std::make_pair(cur_root->right, cur_depth + 1));
            }
        }
        return depth;
    }
};
```

# 5 图

# 6 动态规划

# 7 模拟题

## 7.1 杨辉三角

题目：

在杨辉三角中，每个数是它左上方和右上方的数的和。

示例：

```
输入：5
输出：
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

解答：

1. i 从 1 开始，到 <numRows +1
2. j 从 0 开始，到 <i
3. j=0|| i-1 时，直接放 1
4. else，放 `res[i-2][j-1]+res[i-2][j]`

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> res;

        for (size_t i = 1; i < numRows + 1; ++i) {
            vector<int> sub_res;
            size_t j = 0;
            while (j < i) {
                //cout << i << "xxxx" << j << endl;
                if (j == 0 || j == i - 1) {
                    sub_res.emplace_back(1);
                } else {
                    //cout << i << "x" << j << endl;
                    sub_res.emplace_back(res[i - 2][j - 1] + res[i - 2][j]);
                }
                ++j;
            }
            res.emplace_back(sub_res);
        }
        return res;
    }
};
```

## 7.2 买卖股票的最佳时机

题目：

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

解答：

我们需要找到最小的谷之后的最大的峰。我们可以维持两个变量

- minprice ：迄今为止所得到的最小的谷值。初始化为 int_max，如果当前价格有比它小，那就更新它为当前价格
- maxprofit，迄今为止所得到的最大的利润（卖出价格与最低价格之间的最大差值）。如果当前价格与 minprice 的差比它大，那就更新它

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minprice = INT_MAX;
        int maxprofit = 0;
        for (int i = 0; i < prices.size(); ++i) {
            if (prices[i] < minprice) {
                minprice = prices[i];
            } else if (prices[i] - minprice > maxprofit) {
                maxprofit = prices[i] - minprice;
            }
        }
        return maxprofit;
    }
};
```

## 7.3 买卖股票的最佳时机 II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 7
```

解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，
这笔交易所能获得利润 = 5-1 = 4 。
随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，
这笔交易所能获得利润 = 6-3 = 3 。
示例 2:

输入：[1,2,3,4,5]
输出：4
解释：在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，
这笔交易所能获得利润 = 5-1 = 4 。
注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
示例 3:

输入：[7,6,4,3,1]
输出：0
解释：在这种情况下，没有交易完成，所以最大利润为 0。

解答：

方法一：

我们的兴趣点是连续的峰和谷。

关键是我们需要考虑到紧跟谷的每一个峰值以最大化利润。如果我们试图跳过其中一个峰值来获取更多利润，那么我们最终将失去其中一笔交易中获得的利润，从而导致总利润的降低。

$$totalprofit = \sum_{i} height(peak_i) - height(valley_i)$$

例如，在上述情况下，如果我们跳过 $peak_i$ 和 $valley_j$。试图通过考虑差异较大的点以获取更多的利润，获得的净利润总是会小与包含它们而获得的静利润，因为 C 总是小于 A+B。

时间复杂度：O(n)。遍历一次。

空间复杂度：O(1)。需要常量的空间。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int i = 0;
        if (prices.size() == 0) return 0;
        int valley = prices[0];
        int peak = prices[0];
        int maxprofit = 0;
        while (i < prices.size() - 1) {
            while (i < prices.size() - 1 && prices[i] >= prices[i + 1])
                i++;
            valley = prices[i];
            while (i < prices.size() - 1 && prices[i] <= prices[i + 1])
                i++;
            peak = prices[i];
            maxprofit += peak - valley;
        }
        return maxprofit;
    }
};
```

方法二：

该解决方案遵循方法一的本身使用的逻辑，但有一些轻微的变化。在这种情况下，我们可以简单地继续在斜坡上爬升并持续增加从连续交易中获得的利润，而不是在谷之后寻找每个峰值。最后，我们将有效地使用峰值和谷值，但我们不需要跟踪峰值和谷值对应的成本以及最大利润，但我们可以直接继续增加加数组的连续数字之间的差值，如果第二个数字大于第一个数字，我们获得的总和将是最大利润。

时间复杂度：O(n)。遍历一次。

空间复杂度：O(1)。需要常量的空间。

```
class Solution {
public:
     int maxProfit(vector<int>& prices) {
        int maxprofit = 0;
        for (int i = 1; i < prices.size(); i++) {
            if (prices[i] > prices[i - 1])
                maxprofit += prices[i] - prices[i - 1];
        }
        return maxprofit;
    }
};
```