Contents

1 c++ 基础

1.1 实现单例模式

参考https://www.cnblogs.com/qiaoconglovelife/p/5851163.html

1.1.1 线程不安全版本的经典实现 (懒汉实现)

- **构造函数**声明为 **private 或 protect** 防止被外部函数实例化
- 内部保存一个 **private static 的类指针 p** 保存唯一的实例
- 由一个 **public** 的类方法 **(instance 函数)** 返回单例类唯一的 **static 实例指针**
- 全局范围内给 p 赋**初始值 NULL**
- insance 函数里判断指针 p 是否为 NULL，如果是就 new 一个，反之直接 return p

如果两个线程同时首次调用 instance 方法且**同时检测到 p 是 NULL 值**，则两个线程会**同时构造一个实例给 p**，因此是线程不安全的!!

```
class singleton
{
protected:
    singleton(){}
private:
    static singleton* p;
public:
    static singleton* instance();
};
singleton* singleton::p = NULL;
singleton* singleton::instance()
{
    if (p == NULL)
        p = new singleton();
    return p;
}
```

单例大约有两种实现方法：懒汉与饿汉。

- **懒汉**：故名思义，**不到万不得已就不会去实例化类**，也就是说在**第一次用到类实例的时候才会去实例化**，所以上边的经典方法被归为懒汉实现；
- **饿汉**：饿了肯定要饥不择食。所以在**单例类定义的时候就进行实例化。** 特点与选择：
- 由于要进行**线程同步**，所以在**访问量比较大，或者可能访问的线程比较多时**，采用**饿汉**实现，可以实现更好的性能。这是**以空间换时间。**
- 在**访问量较小时**，采用**懒汉**实现。这是**以时间换空间。**

1.1.2 加锁的经典版本懒汉实现

```
class singleton
{
protected:
    singleton()
    {
        pthread_mutex_init(&mutex);
    }
private:
```

```
    static singleton* p;
public:
    static pthread_mutex_t mutex;
    static singleton* initance();
};

pthread_mutex_t singleton::mutex;
singleton* singleton::p = NULL;
singleton* singleton::initance()
{
    if (p == NULL)
    {
        pthread_mutex_lock(&mutex);
        if (p == NULL)
            p = new singleton();
        pthread_mutex_unlock(&mutex);
    }
    return p;
}
```

1.1.3 加锁的懒汉实现

- 定义一个**静态**的 pthread_mutex_t 类型的**类变量 mutex**
- **构造函数**中对这个 mutex 进行 **pthread_mutex_init**
- **instance 函数里**
  - 先加锁 pthread_mutex_lock
  - 定义一个**静态的实例**
  - 释放锁 pthread_mutex_unlock
  - **返回其静态实例的地址**

```
class singleton
{
protected:
    singleton()
    {
        pthread_mutex_init(&mutex);
    }
public:
    static pthread_mutex_t mutex;
    static singleton* initance();
    int a;
};

pthread_mutex_t singleton::mutex;
singleton* singleton::initance()
{
    pthread_mutex_lock(&mutex);
    static singleton obj;
    pthread_mutex_unlock(&mutex);
    return &obj;
}
```

1.1.4 不用加锁的饿汉版本实现

- **构造函数**声明为 **private 或 protect** 防止被外部函数实例化
- 内部保存一个 **private static 的类指针 p** 保存唯一的实例
- 由一个 **public** 的类方法 **(instance 函数)** 返回单例类唯一的 **static 实例指针**，实现时**直接返回 p**
- 全局范围内 **new 一个对象给 p**

```
class singleton
{
protected:
    singleton()
    {}
private:
    static singleton* p;
public:
    static singleton* initance();
};
singleton* singleton::p = new singleton;
singleton* singleton::initance()
{
    return p;
}
```

2 字符串

2.1 替换空格 (offerNo2)

请实现一个函数，将一个字符串中的每个空格替换成 “%20”。例如，当字符串为 We Are Happy. 则经过替换之后的字符串为 We%20Are%20Happy。

解答:

- 先**从前往后**，计算有**多少个空格**，然后预分配原始长度 +2 倍的空格数这么长
- 然后使用**两个指针**，**从后往前**处理:
  - 第一个指针初始指向原字符串的末尾。
  - 第二个指针初始指向新的末尾。
  - 如果第一个指针指向的是空格，那么第二个指针往前移三格，并依次置 0、2、%，并前移第一个指针
  - 如果第一个指针指向的不是空格，那么第二个指针把第一个指针的值拷过来，并前移两个指针
  - 当两个指针相遇时结束

char*str，直接用下标来取就行 str[xxlength]，所以上面说的两个指针，其实就是两个下标 ~!

```
class Solution {
public:
    void replaceSpace(char *str,int length) {
        if (str == NULL || length <= 0) {
            return;
        }
        int old_len = 0;
        int new_len = 0;
        int space_cnt = 0;
        while (str[old_len] != '\0') {
            if (str[old_len] == ' ') space_cnt++;
            old_len++;
        }
```

```
        new_len = old_len + 2 * space_cnt;
        if (new_len > length) {
            return;
        }
        int pOldlength = old_len; //注意不要减一因为隐藏个‘\0’也要算里
        int pNewlength = new_len;
        while (pOldlength >= 0 && pNewlength > pOldlength) {
             if (str[pOldlength] == ' ') {
                 str[pNewlength--]='0';
                 str[pNewlength--]='2';
                 str[pNewlength--]='%';
                 } else {//不是空格就把 pOldlength 指向的字符装入 pNewlength 指向的位置
                     str[pNewlength--]=str[pOldlength];
                 }
                 pOldlength--; //不管是 if 还是 else 都要把 pOldlength 前移
          }

    }
};
```

2.2 亲密字符串

给定两个由小写字母构成的字符串 A 和 B ，只要我们可以通过交换 A 中的两个字母得到与 B 相等的结果，就返回 true ；否则返回 false 。

例如:

示例 1:

```
输入： A = "ab", B = "ba"
输出： true
```

示例 2:

```
输入： A = "ab", B = "ab"
输出： false
```

示例 3:

```
输入： A = "aa", B = "aa"
输出： true
```

示例 4:

```
输入： A = "aaaaaaabc", B = "aaaaaaacb"
输出： true
```

示例 5:

```
输入： A = "", B = "aa"
输出： false
```

限制:

- 0 <= A.length <= 20000
- 0 <= B.length <= 20000
- A 和 B 仅由小写字母构成。

分析:

因为题目要求只有两个元素交换一次，所以

- 如果 AB 不等长，不可能
- 如果 AB 完全相等，看看有没有重复数字，有的话，随便交换两个就能达到要求，否则不可能
- else，只要遍历数组，找到两个位置 i 和 j，A[i]!=B[i]，且 A[j]!=B[j]，且 A[i]=B[j]，且 A[j]=B[i]

```
class Solution {
public:
    bool buddyStrings(string A, string B) {
        if (A.size() != B.size()) {
            return false;
        }
        if (A == B) {
            // 这种情况下，如果 A 里面字母都不重复，那就不符合要求
            int* counts = new int[26];
            for (auto& i: A) {
                counts[i - 'a']++;
            }
            for (int i = 0; i < 26; ++i) {
                if (counts[i] > 1) {
                    return true;
                }
            }
            return false;
        } else {
            int first = -1, second = -1;
            for (int i = 0; i < A.size(); ++i) {
                if (A[i] != B[i]) {
                    // 第一次不相等，给 first 赋值
                    if (first == -1) {
                        first = i;
                    } else if (second == -1) {
                        // 第二次不相等，给 second 赋值
                        second = i;
                    } else {
                        // 如果有第 3 次不相等，就不行了
                        return false;
                    }
                }
            }
            // 如果只有一个不相等是不行的，如果有两个，那就是前面提到的判断条件
            return (second != -1 && A[first] == B[second] && A[second] ==B[first]);
        }
    }
};
```

2.3 回文数

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

尾数能被 10 整除，即尾数是 0 的也不行，因为首位不是 0

- 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**，**把这次的尾数扔掉**

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，当除完的原始数字不大于反转后的数字时，就意味着我们已经处理了一半位数的数字。

例如，原数字是 4123，反转到 321>41 的时候，就到一半了；如果原数字是 412，反转到 21>4 的时候也到一半了。也就是反转的位数比剩下的多，肯定到一半了。或者，原数字是 1234，反转到 34>12

举个是回文数的例子，原数字是 3223，32==32，break 了；原数字 121，12>1，break 掉

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。所以对于奇数位，就是判断 x==revertedNumber/10

```
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);
    }
};
```

2.4 反转整数

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;
    }
};
```

2.5 罗马数字转整数

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;
    }
};
```

2.6 有效的括号

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;
    }
};
```

3 数组、栈、队列

3.1 旋转数组的最小数字（offerNo6）

把一个数组**最开始的若干个元素搬到数组的末尾**，我们称之为数组的旋转。输入一个**非减排序**的数组的一个旋转，输出旋转数组的最小元素。例如数组 {3,4,5,1,2} 为 {1,2,3,4,5} 的一个旋转，该数组的最小值为 1。NOTE：给出的所有元素都大于 0，若数组大小为 0，请返回 0。

旋转之后的数组实际上可以划分成**两个有序的子数组**：**前面子数组**的大小**都大于后面子数组**中的元素

实际上**最小的元素就是两个子数组的分界线。**

方法一：直接找到后一个数比上一个小的位置，那这后一个数就是我们要的了：

```
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        if (rotateArray.size() == 0) return 0;
        int tmp = rotateArray[0];
        for (int i = 0; i < rotateArray.size(); ++i) {
            if (tmp > rotateArray[i]) {
                return rotateArray[i];
            }
            tmp = rotateArray[i];
        }
        return tmp;
```

```
    }
};
```

方法二：二分法：

数组一定程度上是排序的，因此我们试着用**二分查找法**寻找这个最小的元素。拿 mid 和 high 做比较，考虑以下三种情况：

- array[mid] > array[high]:

出现这种情况的 array 类似 [3,4,5,6,0,1,2]，此时最小数字**一定在 mid 的右边**。

```
low = mid + 1
```

- array[mid] == array[high]:

出现这种情况的 array 类似 [1,0,1,1,1] 或者 [1,1,1,0,1]，此时最小数字不好判断在 mid 左边还是右边, 这时只好一个一个试，所以 **high 左移一格**

```
high = high - 1
```

- array[mid] < array[high]:

出现这种情况的 array 类似 [2,2,3,4,5,6,6], 此时最小数字一定就**是 array[mid]** 或者**在 mid 的左边**。因为右边必然都是递增的。所以**往左半边找**。

```
high = mid
```

注意这里有个坑：如果待查询的范围**最后只剩两个数**，那么 mid 一定会指向下标靠前的数字比如 array = [4,6] array[low] = 4 ;array[mid] = 4 ; array[high] = 6 ; **如果 high = mid - 1，就会产生错误，因此 high = mid**

但情形 (1) 中 low = mid + 1 就不会错误

- 最终返回 array[low] 或者 array[high] 都行

```
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        int low = 0;
        int high = rotateArray.size() - 1;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (rotateArray[mid] > rotateArray[high]) {
                low = mid + 1;
            } else if (rotateArray[mid] == rotateArray[high]) {
                high = high - 1;
            } else if (rotateArray[mid] < rotateArray[high]) {
                high = mid;
            }
        }
        return rotateArray[high];
    }
};
```

3.2 用两个栈来实现一个队列（offerNo5）

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型。

- 入队：将元素进栈 A
- 出队：判断栈 B 是否为空，
  - 如果为空，则将栈 A 中所有元素 pop，并 push 进栈 B，栈 B 出栈；
  - 如果不为空，栈 B 直接出栈。

```
class Solution
{
public:
    void push(int node) {
       stack1.push(node);
    }

    int pop() {
        int a;
        if (stack2.empty()) {
            while (!stack1.empty()) {
                a = stack1.top();
                stack2.push(a);
                stack1.pop();
            }
        }
        a = stack2.top();
        stack2.pop();
        return a;
    }

private:
    stack<int> stack1;
    stack<int> stack2;
};
```

变形：用两个队列实现一个栈的功能

- 入栈：将元素进队列 A
- 出栈：判断队列 A 中元素的个数是否为 1，
  - 如果等于 1，则出队列，
  - 否则将队列 A 中的元素依次出队列并放入队列 B，直到队列 A 中的元素留下一个，然后队列 A 出队列，再把队列 B 中的元素出队列以此放入队列 A 中。

3.3 二维数组中的查找 (offerNo1)

在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

解法：

- 矩阵是有序的，从**左下角**来看，向上数字递减，向右数字递增，
- 因此从左下角开始查找，当要查找数字比左下角数字大时。右移
- 要查找数字比左下角数字小时，上移

```
class Solution {
public:
    bool Find(int target, vector<vector<int> > array) {
        int row_num = array.size();
        int col_num = array[0].size();
        // start from left corner
        for (int i = row_num - 1, j=0; i >= 0 && j < col_num; ) {
            if (array[i][j] > target) {
                i--;
            } else if(array[i][j] < target){
                j++;
```

```
            } else {
                return true;
            }
        }
        return false;
    }
};
```

注意：也可以从右上角开始，但不能从左上角或者右下角开始（因为这样就无法缩小查找范围了）

3.4 转置矩阵

给定一个矩阵 A，返回 A 的转置矩阵。

矩阵的转置是指将矩阵的主对角线翻转，交换矩阵的行索引与列索引。

示例 1：

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

示例 2：

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

限制：

- 1 <= A.length <= 1000
- 1 <= A[0].length <= 1000

解答：

其实就是把按行遍历改成按列遍历输出，拿笔画一下就知道，例如有一个 2x3 的，push 的顺序就是 `a[0][0]`、`a[1][0]`、`a[2][0]`、`a[0][1]`、`...`

```
class Solution {
public:
    vector<vector<int>> transpose(vector<vector<int>>& A) {
        vector<vector<int> > res;
        int row_size = A.size();
        int col_size = A[0].size();
        for (int i = 0; i < col_size; ++i) {
            vector<int> tmp_vec;
            for (int j = 0; j < row_size; ++j) {
                tmp_vec.push_back(A[j][i]);
            }
            res.push_back(tmp_vec);
        }
        return res;
    }
};
```

3.5 移除元素

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;
    }
};
```

3.6 两数之和

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));
            }
        }
    }
};
```

3.7 无重复字符的最长子串

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;
    }
};
```

3.8 两个排序数组的中位数

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/

3.9 合并两个有序数组

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--;
            }
        }
    }
};
```

4 链表

4.1 从头到尾打印一个链表 (offerNo3)

输入一个链表，按链表值从尾到头的顺序返回一个 ArrayList。

解析：

直接按顺序扔到一个 vec 里，然后返回前调用 `std::reverse` 把这个 vec 反转一下

```
/**
*  struct ListNode {
*        int val;
*        struct ListNode *next;
*        ListNode(int x) :
*              val(x), next(NULL) {
*        }
*  };
*/
class Solution {
public:
    vector<int> printListFromTailToHead(ListNode* head) {
        vector<int> vec;
        while (head != NULL) {
            vec.emplace_back(head->val);
            head = head->next;
        }
```

```
        std::reverse(vec.begin(), vec.end());
        return vec;
    }
};
```

4.2 环形链表

给定一个链表，判断链表中是否有环。

分析:

方法一：哈希表

检查一个结点此前是否被访问过来判断链表。常用的方法是使用哈希表。

我们遍历所有结点并在哈希表中存储每个结点的引用（或内存地址）。如果当前结点为空结点 null（即已检测到链表尾部的下一个结点），那么我们已经遍历完整个链表，并且该链表不是环形链表。如果当前结点的引用已经存在于哈希表中，那么返回 true（即该链表为环形链表）。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        set<ListNode*> set_link;
        ListNode* p = head;
        while (p != NULL) {
            if (set_link.find(p) != set_link.end()) {
                return true;
            }
            set_link.insert(p);
            p = p->next;
        }
        return false;
    }
};
```

方法二：双指针

使用具有不同速度的快、慢两个指针遍历链表，空间复杂度可以被降低至 O(1)O(1)。慢指针每次移动一步，而快指针每次移动两步。

如果列表中不存在环，最终快指针将会最先到达尾部，此时我们可以返回 false

时间复杂度的分析见https://leetcode-cn.com/problems/linked-list-cycle/solution/

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
```

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (head == NULL || head->next == NULL) {
            return false;
        }
        ListNode* fast = head->next;// fast 初始化为 next，起点就比 slow 快了
        ListNode* slow = head;
        while(fast != slow) {
            if (fast == NULL || fast->next == NULL) {
                // 如果 fast 到终点了，或者 fast 的下一个节点是终点，说明 slow 肯定追不上来了
                return false;
            }
            slow = slow->next;
            fast = fast->next->next;
        }
        return true;
    }
};
```

4.3 删除链表中的节点

请编写一个函数，使其可以删除某个链表中**给定的（非末尾）**节点，输入的是要求被删除的节点的值。

示例 1:

```
输入：head = [4,5,1,9], node = 5
输出：[4,1,9]
解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入：head = [4,5,1,9], node = 1
输出：[4,5,9]
解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

解答:

正常的是例如 1->2->3->4->5，想把 3 给删了，那么我们可以让 2 指向 4。

但这题传入的就是 3，我们没法拿到 2。所以要稍微改一下:

把 4 的值赋给 3，然后让 3 指向 5。

这样就相当于把 4 前移了到 3 的位置，然后把 4 删了。。成立的前提是，我们要删除的节点不是末尾。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
```

```
 * };
 */
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
    }
};
```

4.4 删除排序链表中的重复元素

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;
    }
};
```

4.5 两数相加

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;
    }
};
```

5 树

5.1 前序遍历

根-》左-》右这样遍历

递归：

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> res;
    void helper(TreeNode* root) {
        if (root == NULL) return ;
        res.push_back(root->val);
```

```
        helper(root->left);
        helper(root->right);
    }

    vector<int> preorderTraversal(TreeNode *root) {
        helper(root);
        return res;
    }
};
```

非递归:

使用栈:

方法 1:

注意，先扔右子树再扔左子树，因为栈是后进先出，前序是先左再右。但是这种方法好像会爆内存...

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> res;
        stack<TreeNode *> s;
        if (root == NULL){
            return res;
        }
        s.push(root);
        while (!s.empty()){
            TreeNode *cur = s.top();
            s.pop();
            res.push_back(cur->val);
            if (cur->right!=NULL)
                s.push(cur->right);
            if (cur->left!=NULL)
                s.push(cur->left);
        }
        return res;
    }
};
```

方法 2:

不是非常非常懂。。先记着

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> preorder;
        stack<TreeNode*> st;
```

```
        TreeNode *p = root;
        while (p || !st.empty()) {
            if (p) {
                preorder.push_back(p->val);
                st.push(p);
                p=p->left;
            } else {
                p=st.top();
                st.pop();
                p=p->right;
            }
        }
        return preorder;
    }
};
```

5.2 后序遍历

左-》右-》根

递归：

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    void postOrder(TreeNode *root, vector<int>& vec){
        if (root != NULL) {
            postOrder(root->left, vec);
            postOrder(root->right, vec);
            vec.push_back(root->val);
        }
    }
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> vec;
        postOrder(root, vec);
        return vec;
    }
};
```

非递归：

方法 1：

参考非递归的前序遍历，然后做如下改动：

前序遍历根-> 左-> 右变成根-> 右-> 左结果再 reverse 一下

```
/**
 * Definition for binary tree
```

```
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
       vector<int> res;
       if(!root)
           return res;
       stack<TreeNode*> st;
       st.push(root);
       TreeNode *temp;
       while( st.size()) {
           temp = st.top();
           st.pop();
           res.push_back(temp->val);
           if (temp->left) {
               st.push(temp->left);
           }
           if (temp->right) {
               st.push(temp->right);
           }
       }
       std::reverse(res.begin(),res.end());
       return res;
    }
};
```

方法 2:

先从根往左一直入栈，直到为空，然后判断栈顶元素的右孩子，如果不为空且未被访问过，则从它开始重复左孩子入栈的过程；否则说明此时栈顶为要访问的节点（因为左右孩子都是要么为空要么已访问过了），出栈然后访问即可，接下来再判断栈顶元素的右孩子...直到栈空。

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> postorder;
        stack<TreeNode*> st;
        TreeNode *p = root;
        TreeNode *r = NULL;// r 用来记录上一次访问的节点
        while (p || !st.empty()) {
            if (p) { //左孩子一直入栈，直到左孩子为
```

```
                st.push(p);
                p=p->left;
            } else {
                p=st.top();
                p = p->right;
                if (p!=NULL && p != r) {
                    //如果栈顶元素的右孩子不为空，且未被访问
                    //则右孩子进栈，
                    //然后重复左孩子一直进栈直到为空的过程
                    st.push(p);
                    p = p->left;
                } else {
                    //否则取出栈顶元素，放到结果数组中，
                    //然后 pop，
                    //r 记录刚刚访问的节点
                    p = st.top();
                    postorder.push_back(p->val);
                    st.pop();
                    r = p;
                    p = NULL;
                }
            }
        }
        return postorder;
    }
};
```

5.3 重建二叉树 (offerNo4)

输入某二叉树的**前序遍历**和**中序遍历**的结果，请**重建**出该二叉树。假设输入的前序遍历和中序遍历的结果中都**不含重复的数字**。例如输入前序遍历序列 {1,2,4,7,3,5,6,8} 和中序遍历序列 {4,7,2,1,5,3,8,6}，则重建二叉树并返回。

思路：

- 创建根节点，根节点肯定是前序遍历的第一个数，new 一个 head 节点，值是根
- 把根节点在中序遍历结果的『第几位』存放于变量 root 中
- 对于中序遍历，根节点左边的节点位于二叉树的左边，根节点右边的节点位于二叉树的右边。所以
  - 把根节点左边的元素（i->root-1）依次扔到 left_in 数组中，作为左子树的中序遍历结果；
  - 把 (i+1->root-1) 的元素扔到 left_pre 这个数组中，当做左子树的前序遍历结果
- 同样地：
  - 把根节点右边的元素（root+1->inlen）依次扔到 right_in 数组中，作为右子树的中序遍历结果；
  - 把 (i->inlen) 的元素扔到 right_pre 这个数组中，当做右子树的前序遍历结果
- head->left 就是递归 left_pre,left_in 的返回结果
- head->right 就是递归 rightt_pre,right_in 的返回结果
- 返回 head

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

```
class Solution {
public:
    TreeNode* reConstructBinaryTree(vector<int> pre,vector<int> vin) {
        int inlen = vin.size();
        if(inlen == 0)
            return NULL;
        vector<int> left_pre, right_pre, left_in, right_in;
        // 创建根节点，根节点肯定是前序遍历的第一个数
        TreeNode* head = new TreeNode(pre[0]);
        // 找到中序遍历根节点所在位置，存放于变量 root 中
        int root = 0;
        for(int i=0; i < inlen; i++) {
            if (vin[i] == pre[0]) {
                root = i;
                break;
            }
        }
        // 对于中序遍历，根节点左边的节点位于二叉树的左边，根节点右边的节点位于二叉树的右边
        // 利用上述这点，对二叉树节点进行归并
        for(int i = 0; i < root; i++) {
            left_in.push_back(vin[i]);
            left_pre.push_back(pre[i + 1]);//前序第一个为根节点
        }
        for(int i = root + 1; i < inlen; i++) {
            right_in.push_back(vin[i]);
            right_pre.push_back(pre[i]);
        }
        //和 shell 排序的思想类似，取出前序和中序遍历根节点左边和右边的子树
        //递归，再对其进行上述所有步骤，即再区分子树的左、右子子数，直到叶节点
        head->left = reConstructBinaryTree(left_pre, left_in);
        head->right = reConstructBinaryTree(right_pre, right_in);
        return head;
    }
};
```

5.4 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;
    }
};
```

5.5 二叉树的最大深度

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;
    }
};
```

5.6 二叉树的最近公共祖先

参考https://blog.csdn.net/wangls56/article/details/88783783

递归

```
TreeNode findAncestor(TreeNode root, TreeNode node1, TreeNode node2) {
    if (root == null) return null;
    if (root == node1 || root == node2) return root;
    TreeNode left  = findAncestor(root.left,  node1, node2);
    TreeNode right = findAncestor(root.right, node1, node2);
    if (left != null && right != null) return root;
    return right == null ? left : right;
}
```

如果有指向父节点的指针，可以转化成两个链表求第一个交点的问题。

6 图

7 动态规划

8 模拟题

8.1 杨辉三角

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;
}
};
```

8.2 买卖股票的最佳时机

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;
    }
};
```

8.3 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;
    }
};
```

8.4 棒球比赛

给定一个字符串列表，每个字符串可以是以下四种类型之一：

1. 整数（一轮的得分）：直接表示您在本轮中获得的积分数。
2. “+”（一轮的得分）：表示本轮获得的得分是**前两轮有效**回合得分的总和。
3. “D”（一轮的得分）：表示本轮获得的得分是**前一轮有效**回合得分的**两倍**。
4. “C”（一个操作，这不是一个回合的分数）：表示您获得的**最后一个有效回合的分数是无效的**，应该被**移除**。

每一轮的操作都是永久性的，可能会对前一轮和后一轮产生影响。你需要返回你在所有回合中得分的总和。

示例：

```
输入：["5","-2","4","C","D","9","+","+"]
输出：27
解释：
第1轮：你可以得到5分。总和是：5。
第2轮：你可以得到-2分。总数是：3。
第3轮：你可以得到4分。总和是：7。
操作1：第3轮的数据无效。总数是：3。
第4轮：你可以得到-4分（第三轮的数据已被删除）。总和是：-1。
第5轮：你可以得到9分。总数是：8。
第6轮：你可以得到-4 + 9 = 5分。总数是13。
第7轮：你可以得到9 + 5 = 14分。总数是27。
```

注意：

- 输入列表的大小将介于 1 和 1000 之间。
- 列表中的每个整数都将介于-30000 和 30000 之间。

解法：

使用**栈**！！！因为我们只处理涉及最后或倒数第二轮的操作。

1. c++11 里的 string 和各种数值类型的互转：http://www.cnblogs.com/gtarcoder/p/4925592.html
2. 如果是数字，就 push 进去
3. 如果是 C，就 pop 掉 top，因为这个 stack 只存『有效的』
4. 如果是 D，就把 top 乘以 2，push 进去（因为这个 stack 只存『有效的』）
5. 如果是 +，计算 top 两个元素的和，当做这轮的得分，扔进去（注意，前面几轮的得分不能变，而因为是栈，所以我们先把 top 给 pop 出来，算原来两个 top 的和得到新的 top，然后把原来的 top 扔回去，再把新的 top 扔进去。。。）
6. stack 没有迭代器，不能被遍历。。。所以只能一个个 pop 出来

```
class Solution {
public:
    int calPoints(vector<string>& ops) {

        stack<int> st;
        for (auto& i: ops) {
            if (i == "+") {
                int top = st.top();
                st.pop();
                int new_top = top + st.top();
                st.push(top);
                st.push(new_top);
            } else if (i == "D") {
                st.push(2 * st.top());
            } else if (i == "C") {
                st.pop();
            } else {
                int tmp = std::stoi(i);
                st.push(tmp);
            }
        }
        int res = 0;
        while (st.size() > 0) {
            res += st.top();
            st.pop();
        }
        return res;
    }
};
```

8.5 柠檬水找零

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。

顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

示例 1：

输入：[5,5,5,10,20]
输出：true
解释：
前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
由于所有客户都得到了正确的找零，所以我们输出 true。
示例 2：

输入：[5,5,10]
输出：true
示例 3：

输入：[10,10]
输出：false
示例 4：

输入：[5,5,10,10,20]
输出：false
解释：
前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
由于不是每位顾客都得到了正确的找零，所以答案是 false。

限制：

- 0 <= bills.length <= 10000
- bills[i] 不是 5 就是 10 或是 20

解答：

用两个变量，分别表示 5 块和 10 块的个数。因为 20 块没用。。找不出去

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int five = 0;
        int ten = 0;
        for (auto& i: bills) {
            if (i == 5) {
                five++;
            } else if (i == 10) {
                if (five > 0) {
                    five--;
                    ten++;
                } else {
                    return false;
                }
            } else {
                if (five > 0 && ten > 0) {
                    five--;
                    ten--;
```

```
                } else if (five >=3 ) {
                    five -=3;
                } else {
                    return false;
                }
            }
        }
        return true;
    }
};
```