模拟算法刷题路线

力扣

Fizz Buzz

题目描述： 给你一个整数 `n` ，返回一个字符串数组 `answer` （**下标从 1 开始**），其中：

- `answer[i] == "FizzBuzz"` 如果 `i` 同时是 `3` 和 `5` 的倍数。
- `answer[i] == "Fizz"` 如果 `i` 是 `3` 的倍数。
- `answer[i] == "Buzz"` 如果 `i` 是 `5` 的倍数。
- `answer[i] == i` （以字符串形式）如果上述条件全不满足。

示例 1：

> **输入：** n = 3
> **输出：** ["1","2","Fizz"]

示例 2：

> **输入：** n = 5
> **输出：** ["1","2","Fizz","4","Buzz"]

示例 3：

> **输入：** n = 15
> **输出：** ["1","2","Fizz","4","Buzz","Fizz","7","8","Fizz","Buzz","11","Fizz","13","14","FizzBuzz"]

提示：

- `1 <= n <= 10^4`

```cpp
class Solution {
public:
    vector<string> fizzBuzz(int n) {
        vector<string> answer;  // 存储结果

        for(int i = 1; i <= n; i++){  // 遍历1到n
            if(i % 3 == 0 && i % 5 == 0){  // 同时是3和5的倍数
                answer.push_back("FizzBuzz");
            }else if(i % 3 == 0){  // 只是3的倍数
                answer.push_back("Fizz");
            }else if(i % 5 == 0){  // 只是5的倍数
                answer.push_back("Buzz");
            }else {  // 其他情况
                answer.push_back(to_string(i));  // 将数字转为字符串
            }
        }
        return answer;  // 返回结果数组
    }
};
```

> 轮到你报 1：你就说 "1"。

> 轮到你报 2：你就说 "2"。
> 轮到你报 3：因为 3 是 3的倍数，你不能说 "3"，你要喊 "Fizz"。
> 轮到你报 4：你就说 "4"。
> 轮到你报 5：因为 5 是 5的倍数，你不能说 "5"，你要喊 "Buzz"。
> ...
> 一直报到 15：因为 15 既是 3的倍数 又是 5的倍数，你要喊 "FizzBuzz"。

机器人能否返回原点

题目描述： 在二维平面上，有一个机器人从原点 `(0, 0)` 开始。给出它的移动顺序，判断这个机器人在完成移动后是否在 `(0, 0)` 处结束。

移动顺序由字符串 `moves` 表示。字符 `moves[i]` 表示其第 `i` 次移动。机器人的有效动作有 `R`（右），`L`（左），`U`（上）和 `D`（下）。

如果机器人在完成所有动作后返回原点，则返回 `true`。否则，返回 `false`。

注意： 机器人“面朝”的方向无关紧要。 `“R”` 将始终使机器人向右移动一次，`“L”` 将始终向左移动等。此外，假设每次移动机器人的移动幅度相同。

示例 1:

> **输入:** moves = "UD"
> **输出:** true
> **解释:** 机器人向上移动一次，然后向下移动一次。所有动作都具有相同的幅度，因此它最终回到它开始的原点。因此，我们返回 true。

示例 2:

> **输入:** moves = "LL"
> **输出:** false
> **解释:** 机器人向左移动两次。它最终位于原点的左侧，距原点有两次 “移动” 的距离。我们返回 false，因为它在移动结束时没有返回原点。

提示:

- `1 <= moves.length <= 2 * 10^4`
- `moves` 只包含字符 `'U'`, `'D'`, `'L'` 和 `'R'`

```
class Solution {
public:
    bool judgeCircle(string moves) {
        int x = 0;  // 水平方向坐标
        int y = 0;  // 垂直方向坐标

        for(char c : moves){  // 遍历每个移动指令
            if(c == 'U'){      // 向上移动
                y++;
            }else if(c == 'D'){ // 向下移动
                y--;
            }else if(c == 'R'){ // 向右移动
                x++;
            }else {             // 向左移动（指令为'L'）
                x--;
            }
        }
        return x == 0 && y == 0;  // 判断是否回到原点
    }
```

```
};
```

加一

题目描述： 给定一个由 **整数** 组成的 **非空** 数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1：

> **输入：** digits = [1,2,3]
> **输出：** [1,2,4]
> **解释：** 输入数组表示数字 123。 加 1 得到 123 + 1 = 124。 因此，结果应该是 [1,2,4]。

示例 2：

> **输入：** digits = [4,3,2,1]
> **输出：** [4,3,2,2]
> **解释：** 输入数组表示数字 4321。 加 1 得到 4321 + 1 = 4322。 因此，结果应该是 [4,3,2,2]。

示例 3：

> **输入：** digits = [0] **输出：** [1]

提示：

- `1 <= digits.length <= 100`
- `0 <= digits[i] <= 9`

```cpp
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        int n = digits.size();  // 获取数字的位数

        // 从最低位开始处理
        for(int i = n - 1; i >= 0; i--) {
            if(digits[i] < 9) {  // 当前位小于9，直接加一
                digits[i]++;
                return digits;    // 返回结果
            }
            digits[i] = 0;  // 当前位为9，加一后变为0
        }

        // 处理全部位都是9的情况（例如999→1000）
        digits.insert(digits.begin(), 1);  // 在最前面插入1
        return digits;
    }
};
```

IP 地址无效化

题目描述： 给你一个有效的 IPv4 地址 `address`，返回这个 IP 地址的无效化版本。

所谓无效化 IP 地址，其实就是用 `"[.]"` 代替了每个 `"."`。

示例 1：

> **输入：** address = "1.1.1.1"
> **输出：** "1[.]1[.]1[.]1"

示例 2：

> **输入：** address = "255.100.50.0"
> **输出：** "255[.]100[.]50[.]0"

提示：

- 给出的 `address` 是一个有效的 IPv4 地址。

```
class Solution {
public:
    string defangIPaddr(string address) {
        string ans = "";  // 存储结果的字符串

        for(char c : address){  // 遍历原地址的每个字符
            if(c == '.'){       // 遇到点号时
                ans += "[.]";   // 添加"[.]"进行替换
            }else{              // 其他字符保持不变
                ans += c;
            }
        }
        return ans;  // 返回修改后的地址
    }
};
```

最富有客户的资产总量

给你一个 `m x n` 的整数网格 `accounts` ，其中 `accounts[i][j]` 是第 `i` 位客户在第 `j` 家银行托管的资产数量。返回最富有客户所拥有的 **资产总量** 。

客户的 **资产总量** 就是他们在各家银行托管的资产数量之和。最富有客户就是 **资产总量** 最大的客户。

示例 1：

```
输入：accounts = [[1,2,3],[3,2,1]]
输出：6
解释：
第 1 位客户的资产总量 = 1 + 2 + 3 = 6
第 2 位客户的资产总量 = 3 + 2 + 1 = 6
两位客户都是最富有的，资产总量都是 6 ，所以返回 6 。
```

示例 2：

```
输入：accounts = [[1,5],[7,3],[3,5]]
输出：10
解释：
第 1 位客户的资产总量 = 6
第 2 位客户的资产总量 = 10
第 3 位客户的资产总量 = 8
第 2 位客户是最富有的，资产总量是 10
```

示例 3：

```
输入：accounts = [[2,8,7],[7,1,3],[1,9,5]]
输出：17
```

提示：

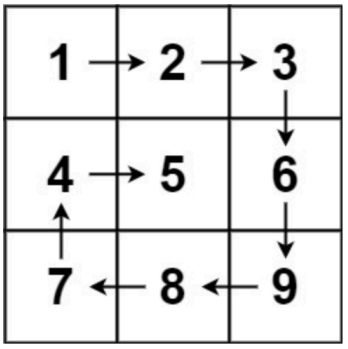
- `m == accounts.length`
- `n == accounts[i].length`
- `1 <= m, n <= 50`
- `1 <= accounts[i][j] <= 100`

```cpp
class Solution {
public:
    int maximumWealth(vector<vector<int>>& accounts) {
        int maxWealth = 0;  // 记录最大资产

        for(int i = 0; i < accounts.size(); i++) {  // 遍历每个客户
            int countWealth = 0;  // 当前客户的资产总和
            for(int j = 0; j < accounts[i].size(); j++) {  // 遍历客户的每个账户
                countWealth += accounts[i][j];  // 累加资产
            }
            maxWealth = max(maxWealth, countWealth);  // 更新最大资产
        }
        return maxWealth;  // 返回最大资产值
    }
};
```

螺旋矩阵

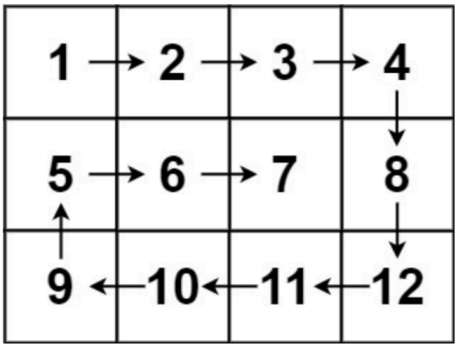
题目描述： 给你一个 `m` 行 `n` 列的矩阵 `matrix` ，请按照 **顺时针螺旋顺序** ，返回矩阵中的所有元素。

示例 1：



> **输入：** matrix = [[1,2,3],[4,5,6],[7,8,9]]
> **输出：** [1,2,3,6,9,8,7,4,5]

示例 2：



> **输入：** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
> **输出：** [1,2,3,4,8,12,11,10,9,5,6,7]

提示：

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 10`
- `-100 <= matrix[i][j] <= 100`

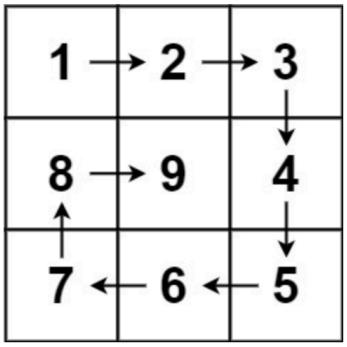
```cpp
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> ans;  // 存储结果

        if(matrix.empty()) return ans;  // 处理空矩阵的情况

        int top = 0;                          // 上边界
        int bottom = matrix.size() - 1;       // 下边界
        int left = 0;                         // 左边界
        int right = matrix[0].size() - 1;     // 右边界

        while(left <= right && top <= bottom) {  // 继续遍历的条件
            // 从左到右遍历上边界
            for(int j = left; j <= right; j++) {
                ans.push_back(matrix[top][j]);
            }
            top++;  // 上边界下移

            // 从上到下遍历右边界
            for(int i = top; i <= bottom; i++) {
                ans.push_back(matrix[i][right]);
            }
            right--;  // 右边界左移

            // 检查是否还有未遍历的行列
            if(left > right || top > bottom) break;

            // 从右到左遍历下边界
            for(int j = right; j >= left; j--) {
                ans.push_back(matrix[bottom][j]);
            }
            bottom--;  // 下边界上移

            // 从下到上遍历左边界
            for(int i = bottom; i >= top; i--) {
                ans.push_back(matrix[i][left]);
            }
            left++;  // 左边界右移
```

```
        }
        return ans;  // 返回螺旋遍历结果
    }
};
```

螺旋矩阵 II

题目描述： 给你一个正整数 `n` ，生成一个包含 `1` 到 `n^2` 所有元素，且元素按顺时针顺序螺旋排列的 `n x n` 正方形矩阵 `matrix` 。

示例 1：



> **输入：** n = 3
> **输出：** [[1,2,3],[8,9,4],[7,6,5]]

示例 2：

> **输入：** n = 1
> **输出：** [[1]]

提示：

- `1 <= n <= 20`

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        // 创建n×n的矩阵
        vector<vector<int>> matrix(n, vector<int>(n));

        // 定义四个边界
        int top = 0;
        int bottom = n - 1;
        int left = 0;
        int right = n - 1;

        int num = 1;        // 当前要填充的数字
        int sum = n * n;    // 总数字个数

        // 当还有数字要填充时继续循环
        while(num <= sum) {
            // 从左到右填充上边界
            for(int j = left; j <= right; j++) {
                matrix[top][j] = num;
                num++;
            }
            top++;  // 上边界下移
```

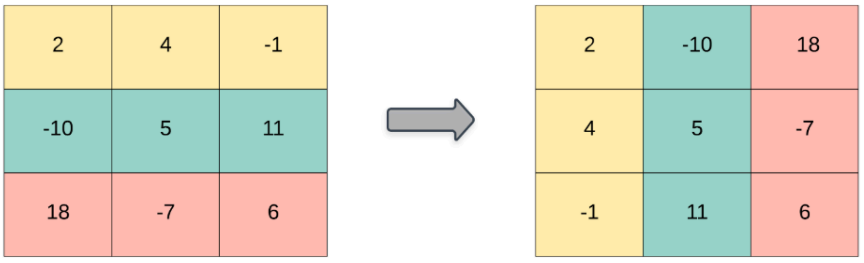
```
            // 从上到下填充右边界
            for(int i = top; i <= bottom; i++) {
                matrix[i][right] = num;
                num++;
            }
            right--;  // 右边界左移

            // 从右到左填充下边界
            for(int j = right; j >= left; j--) {
                matrix[bottom][j] = num;
                num++;
            }
            bottom--;  // 下边界上移

            // 从下到上填充左边界
            for(int i = bottom; i >= top; i--) {
                matrix[i][left] = num;
                num++;
            }
            left++;  // 左边界右移
        }
        return matrix;  // 返回生成的螺旋矩阵
    }
};
```

转置矩阵

题目描述： 给你一个二维整数数组 `matrix`， 返回 `matrix` 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：** matrix = [[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：** matrix = [[1,2,3],[4,5,6]]
> **输出：** [[1,4],[2,5],[3,6]]

提示：

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 1000`
- `1 <= m * n <= 10^5`
- `-10^9 <= matrix[i][j] <= 10^9`

```cpp
class Solution {
public:
    vector<vector<int>> transpose(vector<vector<int>>& matrix) {
        int m = matrix.size();      // 原矩阵的行数
        int n = matrix[0].size();   // 原矩阵的列数

        // 创建转置矩阵，行数变为原列数，列数变为原行数
        vector<vector<int>> res(n, vector<int>(m));

        // 遍历原矩阵，将元素 (i,j) 放到转置矩阵的 (j,i) 位置
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                res[j][i] = matrix[i][j];  // 行列互换
            }
        }
        return res;  // 返回转置后的矩阵
    }
};
```

> 公式如下

$$B[j][i] = A[i][j]$$

棒球比赛

你现在是一场采用特殊赛制棒球比赛的记录员。这场比赛由若干回合组成，过去几回合的得分可能会影响以后几回合的得分。

比赛开始时，记录是空白的。你会得到一个记录操作的字符串列表 `ops`，其中 `ops[i]` 是你需要记录的第 `i` 项操作，`ops` 遵循下述规则：

1. 整数 `x` - 表示本回合新获得分数 `x`
2. `"+"` - 表示本回合新获得的得分是前两次得分的总和。题目数据保证记录此操作时前面总是存在两个有效的分数。
3. `"D"` - 表示本回合新获得的得分是前一次得分的两倍。题目数据保证记录此操作时前面总是存在一个有效的分数。
4. `"C"` - 表示前一次得分无效，将其从记录中移除。题目数据保证记录此操作时前面总是存在一个有效的分数。

请你返回记录中所有得分的总和。

示例 1：

> **输入：** ops = ["5","2","C","D","+"] **输出：** 30
> **解释：** "5" - 记录加 5，记录现在是 `[5]` "2" - 记录加 2，记录现在是 `[5, 2]` "C" - 使前一次得分的记录无效并将其移除，记录现在是 `[5]` "D" - 记录加 2 * 5 = 10，记录现在是 `[5, 10]` "+" - 记录加 5 + 10 = 15，记录现在是 `[5, 10, 15]` 所有得分的总和 5 + 10 + 15 = 30

示例 2：

> **输入：** ops = ["5","-2","4","C","D","9","+","+"]
> **输出：** 27
> **解释：** "5" - 记录加 5，记录现在是 `[5]` "-2" - 记录加 -2，记录现在是 `[5, -2]` "4" - 记录加 4，记录现在是 `[5, -2, 4]` "C" - 使前一次得分的记录无效并将其移除，记录现在是 `[5, -2]` "D" - 记录加 2 * -2 = -4，记录现在是 `[5, -2, -4]` "9" - 记录加 9，记录现在是 `[5, -2, -4, 9]` "+" - 记录加 -4 + 9 = 5，记录现在是 `[5, -2, -4, 9, 5]` "+" - 记录加 9 + 5 = 14，记录现在是 `[5, -2, -4, 9, 5, 14]` 所有得分的总和 5 + -2 + -4 + 9 + 5 + 14 = 27

示例 3：

> **输入：** ops = ["1"]
> **输出：** 1

提示：

- `1 <= ops.length <= 1000`
- `ops[i]` 为 `"C"`、`"D"`、`"+"`，或者一个表示整数的字符串。整数范围是 `[-3 * 10^4, 3 * 10^4]`
- 对于 `"+"` 操作，题目数据保证记录此操作时前面总是存在两个有效的分数
- 对于 `"C"` 和 `"D"` 操作，题目数据保证记录此操作时前面总是存在一个有效的分数

```cpp
class Solution {
public:
    int calPoints(vector<string>& operations) {
        vector<int> res;  // 存储历史有效得分

        for(string op : operations) {  // 遍历每个操作指令
            if(op == "+") {  // 本回合得分为前两次有效得分之和
                int n = res.size();
                int newsum = res[n-1] + res[n-2];  // 计算前两次得分和
                res.push_back(newsum);
            } else if(op == "D") {  // 本回合得分为上一次有效得分的两倍
                int newsum = 2 * res.back();
                res.push_back(newsum);
            } else if(op == "C") {  // 撤销上一次有效得分
                res.pop_back();
            } else {  // 普通数字得分
                res.push_back(stoi(op));  // 字符串转整数并记录
            }
        }

        int sum = 0;  // 总得分
        for(int score : res) {
            sum += score;  // 累加所有有效得分
        }
        return sum;
    }
};
```

字符串相乘

题目描述： 给定两个以字符串形式表示的非负整数 `num1` 和 `num2`，返回 `num1` 和 `num2` 的乘积，它们的乘积也表示为字符串形式。

注意： 不能使用任何内置的 BigInteger 库或直接将输入转换为整数。

示例 1:

> **输入:** num1 = "2", num2 = "3"
> **输出:** "6"

示例 2:

> **输入:** num1 = "123", num2 = "456"
> **输出:** "56088"

提示：

- `1 <= num1.length, num2.length <= 200`

- `num1` 和 `num2` 只能由数字组成。
- `num1` 和 `num2` 都不包含任何前导零，除了数字0本身。

```cpp
class Solution {
public:
    string multiply(string num1, string num2) {
        // 如果任一乘数为0，结果直接返回"0"
        if(num1 == "0" || num2 == "0"){
            return "0";
        }

        int m = num1.size();   // 第一个数的位数
        int n = num2.size();   // 第二个数的位数

        // 结果最多有 m+n 位，初始化为0
        vector<int> res(m + n, 0);

        // 从最低位开始模拟竖式乘法
        for(int i = m - 1; i >= 0; i--) {       // 遍历 num1 的每一位（从个位开始）
            for(int j = n - 1; j >= 0; j--) {   // 遍历 num2 的每一位（从个位开始）
                int mul = (num1[i] - '0') * (num2[j] - '0');  // 当前两位的乘积

                int p1 = i + j;       // 乘积的高位索引（进位位置）
                int p2 = i + j + 1;   // 乘积的低位索引（当前位置）

                int sum = mul + res[p2];  // 当前位乘积加上该位置原有的值

                res[p2] = sum % 10;       // 更新当前位的值
                res[p1] += sum / 10;      // 进位累加到高位
            }
        }

        // 将结果数组转换成字符串，跳过前导零
        string ans = "";
        for(int num : res) {
            if(ans.empty() && num == 0) {  // 跳过结果最前面的0
                continue;
            }
            ans += to_string(num);
        }
        return ans;
    }
};
```

字符串转换整数 (atoi)

题目描述：

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数。

函数 `myAtoi(string s)` 的算法如下：

1. **空格**：读入字符串并丢弃无用的前导空格（`" "`）
2. **符号**：检查下一个字符（假设还未到字符末尾）为 `'-'` 还是 `'+'`。如果两者都不存在，则假定结果为正。
3. **转换**：通过跳过前置零来读取该整数，直到遇到非数字字符或到达字符串的结尾。如果没有读取数字，则结果为0。
4. **舍入**：如果整数数超过 32 位有符号整数范围 $[-2^{31}, 2^{31} - 1]$，需要截断这个整数，使其保持在这个范围内。具体来说，小于 -2^{31} 的整数应该被舍入为 -2^{31}，大于 $2^{31} - 1$ 的整数应该被舍入为 $2^{31} - 1$。

返回整数作为最终结果。

示例 1：

输入： s = "42"
输出： 42
解释： 加粗的字符串为已经读入的字符，插入符号是当前读取的字符。
带下划线的字符是所读的内容，插入符号是当前读入位置。
第 1 步："42"（当前没有读入字符，因为没有前导空格）
^
第 2 步："42"（当前没有读入字符，因为这里不存在 '-' 或 '+'）
^
第 3 步："42"（读入 "42"）
^

示例 2：

输入： s = " -042"
输出： -42
解释：
第 1 步：" -042"（读入前导空格，但忽视掉）
^
第 2 步：" -042"（读入 '-' 字符，所以结果应该是负数）
^
第 3 步：" -042"（读入 "042"，在结果中忽略前导零）
^

示例 3：

输入： s = "1337c0d3"
输出： 1337
解释：
第 1 步："1337c0d3"（当前没有读入字符，因为没有前导空格）
^
第 2 步："1337c0d3"（当前没有读入字符，因为这里不存在 '-' 或 '+'）
^
第 3 步："1337c0d3"（读入 "1337"；由于下一个字符不是一个数字，所以读入停止）
^

示例 4：

输入： s = "0-1"
输出： 0
解释：
第 1 步："0-1"（当前没有读入字符，因为没有前导空格）
^
第 2 步："0-1"（当前没有读入字符，因为这里不存在 '-' 或 '+'）
^
第 3 步："0-1"（读入 "0"；由于下一个字符不是一个数字，所以读入停止）
^

示例 5：

输入： s = "words and 987"
输出： 0
解释：

读取在第一个非数字字符“w”处停止。

提示：

- `0 <= s.length <= 200`
- `s` 由英文字母（大写和小写）、数字（`0-9`）、`' '`、`'+'`、`'-'` 和 `'.'` 组成

```cpp
class Solution {
public:
    int myAtoi(string s) {
        int i = 0;
        int n = s.size();

        // 1. 跳过前导空格
        while(i < n && s[i] == ' '){
            i++;
        }

        int sign = 1;  // 符号位，默认为正

        // 2. 处理正负号
        if (i < n) {
            if (s[i] == '-') {
                sign = -1;  // 负号
                i++;
            } else if (s[i] == '+') {
                sign = 1;   // 正号（可省略）
                i++;
            }
        }

        long res = 0;  // 用 long 存储结果，便于判断溢出
        // 3. 转换数字字符为整数
        while(i < n && isdigit(s[i])){
            int digit = s[i] - '0';  // 当前数字
            res = res * 10 + digit;  // 累加

            // 4. 溢出处理：超出 int 范围时直接返回边界值
            if (res > INT_MAX) {
                return sign == 1 ? INT_MAX : INT_MIN;
            }
            i++;
        }

        return sign * res;  // 返回带符号的结果
    }
};
```

四个严格顺序：
第一步：吃掉空格，只要是空格 `' '`，就忽略；一旦遇到非空格的东西，马上进入第二步。
第二步：判断正负号，如果是 `'-'`：记下来，结果要是负数。如果是 `'+'`：记下来，结果是正数。如果既不是加也不是减（比如是数字 `5`）：默认为正数。
第三步：疯狂读取数字，只要当前字符是 `'0'` 到 `'9'` 之间的数字，就把它拼接到结果里。一旦遇到任何非数字字符（字母、符号、空格、小数点），立刻停止！
第四步：防止爆表，C++ 里的 `int` 类型是有范围的（-2^{31} 到 $2^{31}-1$）。

案例 A："words and 987"

- 去空格：没空格。
- 看符号：第一个是 w，不是符号。
- 读数字：第一个是 w，不是数字！立刻停止！

- 输出：0（因为一个数字都没读到）。

案例 C：" -42"
- 去空格：跳过前三个空格。
- 看符号：看到了 -，记录 sign = -1。
- 读数字：读到 4，结果变 4；读到 2，结果变 42。
- 结束：没字符了。
- 输出：42 * -1 = -42。

整数转罗马数字

题目描述：

七个不同的符号代表罗马数字，其值如下：

符号	值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

罗马数字是通过添加从最高到最低的小数位值的转换而形成的。将小数位值转换为罗马数字有以下规则：

- 如果该值不是以 4 或 9 开头，请选择可以从输入中减去的最大值的符号，将该符号附加到结果，减去其值，然后将其余部分转换为罗马数字。
- 如果该值以 4 或 9 开头，使用 **减法形式**，表示从以下符号中减去一个符号，例如 4 是 5 (`V`) 减 1 (`I`)： `IV` ，9 是 10 (`X`) 减 1 (`I`)： `IX` 。仅使用以下减法形式：4 (`IV`)，9 (`IX`)，40 (`XL`)，90 (`XC`)，400 (`CD`) 和 900 (`CM`)。
- 只有 10 的次方（ `I` , `X` , `C` , `M` ）最多可以连续附加 3 次以代表 10 的倍数。你不能多次附加 5 (`V`)，50 (`L`) 或 500 (`D`)。如果需要将符号附加 4 次，请使用 **减法形式**。

给定一个整数，将其转换为罗马数字。

示例 1：

输入： num = 3749
输出： "MMMDCCXLIX"
解释：
3000 = MMM 由于 1000 (M) + 1000 (M) + 1000 (M)
700 = DCC 由于 500 (D) + 100 (C) + 100 (C)
40 = XL 由于 50 (L) 减 10 (X)
9 = IX 由于 10 (X) 减 1 (I)
注意：49 不是 50 (L) 减 1 (I) 因为转换是基于小数位

示例 2：

输入： num = 58

> **输出：** "LVIII"
> **解释：**
> 50 = L
> 8 = VIII

示例 3：

> **输入：** num = 1994
> **输出：** "MCMXCIV"
> **解释：**
> 1000 = M
> 900 = CM
> 90 = XC
> 4 = IV

提示：

- `1 <= num <= 3999`

```
class Solution {
public:
    string intToRoman(int num) {
        // 从大到小排列的罗马数字对应数值
        int values[] = {1000,900,500,400,100,90,50,40,10,9,5,4,1};
        // 对应的罗马数字符号
        string symbols[] = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
        string res = "";  // 存储转换结果

        // 遍历所有数值组合
        for(int i = 0; i < 13; i++) {
            // 尽可能多地使用当前最大符号
            while(num >= values[i]) {
                num -= values[i];     // 减去已表示的数值
                res += symbols[i];    // 添加对应的罗马符号
            }
        }
        return res;  // 返回罗马数字字符串
    }
};
```

Z 字形变换

题目描述： 将一个给定字符串 `s` 根据给定的行数 `numRows` ，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 `"PAYPALISHIRING"` 行数为 `3` 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如：`"PAHNAPLSIIGYIR"`。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1：

> **输入：** s = "PAYPALISHIRING", numRows = 3
> **输出：** "PAHNAPLSIIGYIR"

示例 2：

> **输入：** s = "PAYPALISHIRING", numRows = 4
> **输出：** "PINALSIGYAHRPI"
> **解释：**

```
P     I    N
A   L S  I G
Y A   H R
P     I
```

示例 3：

> **输入：** s = "A", numRows = 1
> **输出：** "A"

提示：

- `1 <= s.length <= 1000`
- `s` 由英文字母（小写和大写）、`,` 和 `.` 组成
- `1 <= numRows <= 1000`

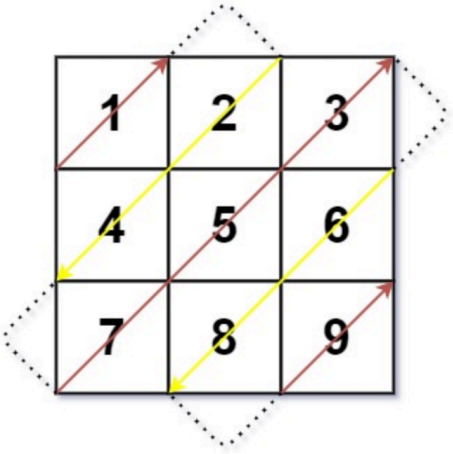
```cpp
class Solution {
public:
    string convert(string s, int numRows) {
        // 特殊情况：单行或行数足够多时，直接返回原字符串
        if(numRows == 1 || numRows >= s.size()){
            return s;
        }

        // 存储每一行字符的数组
        vector<string> rows(numRows);

        int currow = 0;      // 当前行索引
        bool goingDown = false; // 是否向下移动

        // 遍历原字符串每个字符
        for(char c : s){
            rows[currow] += c;   // 将字符加入当前行

            // 到达第一行或最后一行，改变方向
            if(currow == 0 || currow == numRows - 1){
                goingDown = !goingDown;
            }

            // 向下移动则行数+1，向上移动则行数-1
            currow += goingDown ? 1 : -1;
        }

        // 合并所有行的字符串
        string res = "";
        for(string row : rows){
            res += row;
```

```
        }
        return res;
    }
};
```

对角线遍历

题目描述： 给你一个大小为 `m x n` 的矩阵 `mat` ，请以对角线遍历的顺序，用一个数组返回这个矩阵中的所有元素。

示例 1：



> **输入：** mat = [[1,2,3],[4,5,6],[7,8,9]]
> **输出：** [1,2,4,7,5,3,6,8,9]

示例 2：

> **输入：** mat = [[1,2],[3,4]]
> **输出：** [1,2,3,4]

提示：

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n <= 10^4`
- `1 <= m * n <= 10^4`
- `-10^5 <= mat[i][j] <= 10^5`

```
class Solution {
public:
    vector<int> findDiagonalOrder(vector<vector<int>>& mat) {
        int m = mat.size();       // 矩阵行数
        int n = mat[0].size();    // 矩阵列数

        vector<int> res;  // 存储遍历结果

        int x = 0, y = 0;    // 当前位置坐标（行，列）
        bool up = true;      // 移动方向，true为右上，false为左下

        for(int i = 0; i < m * n; i++) {  // 遍历所有元素
            res.push_back(mat[x][y]);     // 将当前元素加入结果

            if(up) {  // 向右上方向移动
```

```
                if(y == n - 1) {      // 到达最右列
                    x++;              // 向下移动一行
                    up = false;       // 切换方向
                } else if(x == 0) {   // 到达第一行
                    y++;              // 向右移动一列
                    up = false;       // 切换方向
                } else {              // 正常右上移动
                    x--;
                    y++;
                }
            } else {  // 向左下方向移动
                if(x == m - 1) {      // 到达最后一行
                    y++;              // 向右移动一列
                    up = true;        // 切换方向
                } else if(y == 0) {   // 到达第一列
                    x++;              // 向下移动一行
                    up = true;        // 切换方向
                } else {              // 正常左下移动
                    x++;
                    y--;
                }
            }
        }
        return res;  // 返回对角线遍历结果
    }
};
```