数组和字符串

238.除自身以外数组的乘积（中等）

1. 题目描述

   给定长度为 *n* 的整数数组 `nums`，其中 *n* > 1，返回输出数组 `output` ，其中 `output[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

   示例:

   ```
   输入: [1,2,3,4]
   输出: [24,12,8,6]
   ```

   说明: 请**不要使用除法，**且在 O(*n*) 时间复杂度内完成此题。

   进阶：你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组**不被视为**额外空间。）

2. 简单实现

   head[i]表示nums[0]到nums[i]的乘积。tail[i]表示nums[i]到nums[nums.size()-1]的乘积，时间复杂度O(n)，空间复杂度O(n)

   ```
   class Solution {
   public:
       vector<int> productExceptSelf(vector<int>& nums) {
           int len = nums.size();
           vector<int> head = vector<int>(len, nums[0]);
           vector<int> tail = vector<int>(len, nums[len-1]);
           for(int i = 1; i < len; i++)
               head[i] = head[i-1]*nums[i];
           for(int i = len-2; i >= 0; i--)
               tail[i] = tail[i+1]*nums[i];

           for(int i = len-1; i >= 0; i--)//从后向前修改head，做为答案返回
               if(i == 0)
                   head[i] = tail[i+1];
               else if(i == len-1)
                   head[i] = head[i-1];
               else
                   head[i] = head[i-1] * tail[i+1];
           return head;
       }
   };
   ```

3. 进阶实现

   稍微再改一改

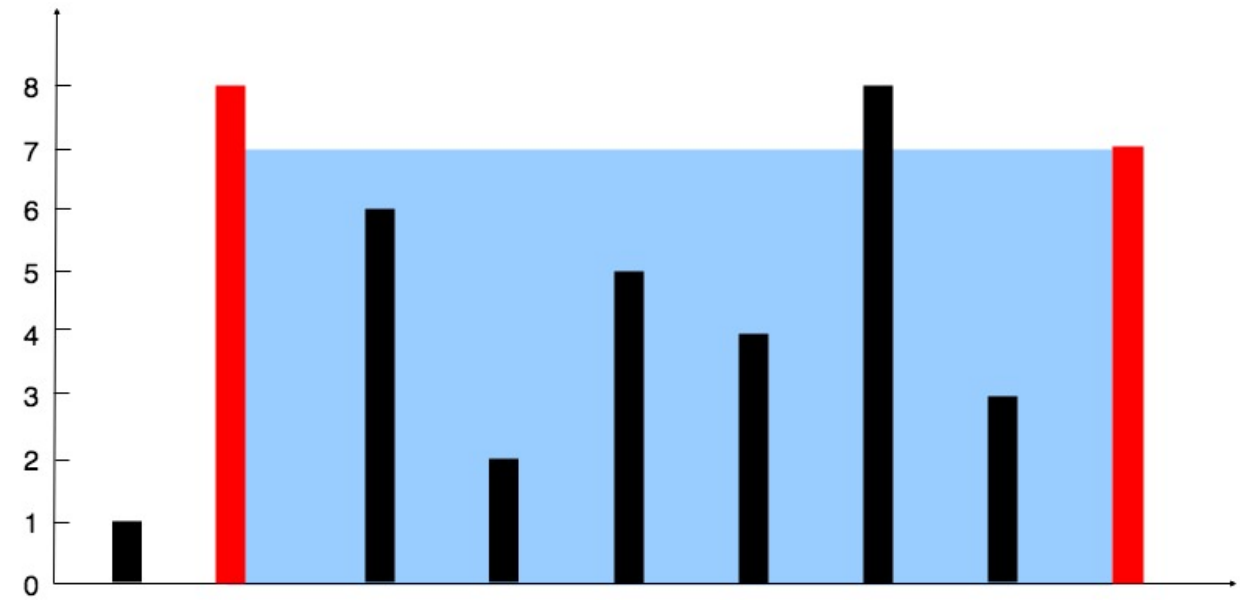
```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int len = nums.size();
        vector<int> ans = vector<int>(len, 1);
        int k = 1;
        for(int i = 0; i < len; i++){
            ans[i] = k;//  k为该数左边的乘积，此时数组存储的是除去当前元素左边的元素乘积
            k *= nums[i];
        }
        k = 1;
        for(int i = len - 1; i >= 0; i--){
            ans[i] *= k; // k为该数右边的乘积，此时数组等于左边的 * 该数右边的。
            k *= nums[i];
        }
        return ans;
    }
};
```

11.盛最多水的容器（中等）

1. 题目描述

给定 *n* 个非负整数 *a*1，*a*2，...，*a*n，每个数代表坐标中的一个点 (*i*, *ai*) 。在坐标内画 *n* 条垂直线，垂直线 *i* 的两个端点分别为 (*i*, *ai*) 和 (*i*, 0)。找出其中的两条线，使得它们与 *x* 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器，且 *n* 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例:

```
输入: [1,8,6,2,5,4,8,3,7]
输出: 49
```

2. 简单实现

双指针

- 线段之间形成的区域总是会受到其中较短那条长度的限制。此外，两线段距离越远，得到的面积就越大。
- 我们在由线段长度构成的数组中使用两个指针，一个放在开始，一个置于末尾。 此外，我们会使用变量ans来持续存储到目前为止所获得的最大面积
- 如果我们试图将指向较长线段的指针向内侧移动，矩形区域的面积将受限于较短的线段而不会获得任何增加。但是，在同样的条件下，移动指向较短线段的指针尽管造成了矩形宽度的减小，但却可能会有助于面积的增大。因为移动较短线段的指针会得到一条相对较长的线段，这可以克服由宽度减小而引起的面积减小。

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int ans = -1;
        int l = 0;
        int r = height.size()-1;
        while(l < r){
            int temp = (r - l) * min(height[l], height[r]);
            if(temp > ans) ans = temp;

            if(height[l] <= height[r])
                l++;
            else
                r--;
        }
        return ans;
    }
};
```

289.生命游戏（中等）

1. 题目描述

根据百度百科，生命游戏，简称为生命，是英国数学家约翰·何顿·康威在1970年发明的细胞自动机。

给定一个包含 m × n 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞具有一个初始状态 *live*（1）即为活细胞， 或 *dead*（0）即为死细胞。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

1. 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；
2. 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；
3. 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；
4. 如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

根据当前状态，写一个函数来计算面板上细胞的下一个（一次更新后的）状态。下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。

示例:

```
输入：
[
  [0,1,0],
  [0,0,1],
  [1,1,1],
  [0,0,0]
]
输出：
[
  [0,0,0],
  [1,0,1],
  [0,1,1],
  [0,1,0]
]
```

进阶:

- 你可以使用原地算法解决本题吗？请注意，面板上所有格子需要同时被更新：你不能先更新某些格子，然后使用它们的更新后的值再更新其他格子。
- 本题中，我们使用二维数组来表示面板。原则上，面板是无限的，但当活细胞侵占了面板边界时会造成问题。你将如何解决这些问题？

2. 简单实现

用正数存储活细胞周围的活细胞数，用负数存储死细胞周围的活细胞数

TIP：注意到周围活细胞为0时，用0存储会对当前细胞"提前致死"导致周围细胞统计出错，因此统计时对所有的细胞计数绝对值+1，判断时再减掉

```
class Solution {
public:
    int m;
    int n;

    void countLive(vector<vector<int>>& board, int x, int y){
        int cnt = 1;//绝对值加1，初始值从1开始
        for(int i = x-1; i <= x+1; i++)
            for(int j = y-1; j <= y+1; j++)
                if(!(i == x && j == y) && (i >= 0 && i < m) && (j >= 0 && j < n) &&
(board[i][j] > 0))
                    cnt++;//周围活细胞计数
        if(board[x][y] <= 0)//当前是死细胞，用负数计数
            cnt = -cnt;
        board[x][y] = cnt;
    }
    void gameOfLife(vector<vector<int>>& board) {
        m = board.size();
        if(m <= 0) return;
        n = board[0].size();
        if(n <= 0) return;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                countLive(board, i, j);//对每个位置周围的活细胞计数
        for(int i = 0; i < m; i++)
```

```
            for(int j = 0; j < n; j++){//根据计数判断死活
                int cur = board[i][j];
                if(cur > 0){//活细胞
                    cur--;//恢复真实计数值
                    if(cur < 2 || cur > 3) board[i][j] = 0;
                    else board[i][j] = 1;
                }
                else{//死细胞
                    cur++;//恢复真实计数值
                    if(cur == -3) board[i][j] = 1;
                    else board[i][j] = 0;
                }
        }
    }
};
```

41.第一个缺失的正数（困难）

1. 题目描述

   给定一个未排序的整数数组，找出其中没有出现的最小的正整数。

   示例 1:

   ```
   输入: [1,2,0]
   输出: 3
   ```

   示例 2:

   ```
   输入: [3,4,-1,1]
   输出: 2
   ```

   示例 3:

   ```
   输入: [7,8,9,11,12]
   输出: 1
   ```

   说明:

   你的算法的时间复杂度应为O(*n*)，并且只能使用常数级别的空间。

2. 最优实现

   - 首先，常数级别空间不能开辅助空间，**但是原数组是可以修改的！！！**
   - 假设数组长度为len，则考虑的范围只在1...len之前，大于len的数字出现也没用，前面必然有缺失的正整数
   - 抽屉原理，各归各位，第一个位置对应值不对的就是缺的

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int len = nums.size();
```

```
        if(len == 0) return 1;
        for(int i = 0; i < len; i++){
            while(nums[i] != i+1){//该位置值不对
                if(nums[i] > len || nums[i] <= 0)//不在考虑范围内
                    break;
                if(nums[i] == nums[nums[i]-1])//正确的位置也有正确的值，只好跳过
                    break;
                swap(nums[i], nums[nums[i]-1]);//把nums[i]换到正确的位置去
            }
        }
        for(int i = 0; i < len; i++)
            if(nums[i] != i+1)
                return i+1;
        return len+1;
    }
};
```

128.最长连续序列（困难）

1. 题目描述

   给定一个未排序的整数数组，找出最长连续序列的长度。

   要求算法的时间复杂度为 *O(n)*。

   示例:

```
输入: [100, 4, 200, 1, 3, 2]
输出: 4
解释: 最长连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

2. 简单实现

   - 排序的方法是O(nlogn)的
   - 考虑暴力解法：依次以每个nums作为序列起始值，考察所能构成的最长序列——O(n^2)
   - 优化：利用哈希表存放所有数字，对每个num，只在num-1不存在时才开始遍历（若num-1存在，必然是以num-1为起始值的更大），这样看似是O(n^2)，但整个数组内每个元素至多访问三次（构造哈希表，查看num-1，查找连续序列），因此是O(n)的

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> s;
        for(int i = 0; i < nums.size(); i++)
            s.insert(nums[i]);
        int ans = 0;
        for(auto it = s.begin(); it != s.end(); it++){
            int cur = *it;
            if(s.count(cur - 1) <= 0){
                while(s.count(cur + 1))
                    cur++;
                ans = max(ans, cur - *it + 1);
```

```
            }
        }
        return ans;
    }
};
```

287.寻找重复数（中等）

1. 题目描述

   给定一个包含 *n* + 1 个整数的数组 *nums*，其数字都在 1 到 *n* 之间（包括 1 和 *n*），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

   示例 1:

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

   示例 2:

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

   说明：

   1. **不能**更改原数组（假设数组是只读的）。
   2. 只能使用额外的 *O*(1) 的空间。
   3. 时间复杂度小于 *O*(n^2) 。
   4. 数组中只有一个重复的数字，但它可能不止重复出现一次。
2. 简单实现
   - 由于题目的限制，使用排序、鸽巢原理、哈希表的方法都不满足要求
   - 尝试二分法：数据范围是1-n，对数组中小于等于mid的值进行计数
     - 如果计数值小于等于mid，那么重复值一定大于mid
     - 否则重复值小于等于mid
     - 时间复杂度为O(nlogn)

```
class Solution {
public:
    int count(vector<int>& nums, int n){
        int ans = 0;
        for(int i = 0; i < nums.size(); i++)
            if(nums[i] <= n)
                ans++;
        return ans;
    }
    int findDuplicate(vector<int>& nums) {
        int l = 0, r =  nums.size() - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
```

```
            if(count(nums, mid) <= mid)
                l = mid + 1;
            else
                r = mid;
        }
        return l;
    }
};
```

3. 最优解法——弗洛伊德的乌龟和兔子，以前在二分法那章记过的、、、

227基本计算器 II（中等）

1. 题目描述

   实现一个基本的计算器来计算一个简单的字符串表达式的值。

   字符串表达式仅包含非负整数，`+`， `-`，`*`，`/` 四种运算符和空格 `` 。 整数除法仅保留整数部分。

   示例 1:

   ```
   输入: "3+2*2"
   输出: 7
   ```

   示例 2:

   ```
   输入: " 3/2 "
   输出: 1
   ```

   示例 3:

   ```
   输入: " 3+5 / 2 "
   输出: 5
   ```

   说明:

   - 你可以假设所给定的表达式都是有效的
   - 请**不要**使用内置的库函数 `eval`

2. 简单实现

   题目简单，难点在于保证代码无bug

   思路

   - 用一个stack nums记录操作数，用stack op记录操作符。
   - 顺序遍历，按顺序读取每个char。
     - 空格再见。
     - 如果是数字，记录一下。
     - 如果是+ - * / ，就要开始计算了。原则是这样的。
       - 只要上次的操作符是 * /，那么上个操作数和当前刚拿到的第二操作数都在，直接计算这一步的结果，然后把他们出栈。

        - 如果这一次的操作符是 * /，那么不管上个操作数是+还是-，操作数和操作符都要入栈。因为优先级高。
        - 当然如果你上个操作数是空，也要入栈。
        - 剩下唯一一种情况，就是上个操作符是+-了，计算结果入栈，新操作符也入栈。
    - 因为遇到操作符才会计算，为了在循环里一同处理，在字符串后面加个操作符。
    - 遍历完毕的时候，nums顶为计算结果。

```
class Solution {
public:
    int calculate(string s) {
        stack<int> nums;
        stack<char> op;
        int val = 0;
        s.append("+");
        for (auto c : s)
            if (isspace(c)) continue;
            if (isdigit(c)) {
                val = val * 10 + (c - '0');
                continue;
            }
            if (!nums.empty() && !op.empty() && (op.top() == '*' || op.top() == '/'))
{
                val = (op.top() == '*') ? nums.top() * val: nums.top() / val;
                nums.pop();
                op.pop();
            }
            if (nums.empty() || op.empty() || c == '*' || c == '/') {
                nums.push(val);
                op.push(c);
            }
            else {
                nums.top() = (op.top() == '+') ? nums.top() + val : nums.top() - val;
                op.top() = c;
            }
            val = 0;
        }
        return nums.top();
    }
};
```

239.滑动窗口最大值（困难）

1. 题目描述

   给定一个数组 *nums*，有一个大小为 *k* 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 *k* 个数字。滑动窗口每次只向右移动一位。

   返回滑动窗口中的最大值。

   示例:

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

提示：你可以假设 *k* 总是有效的，在输入数组不为空的情况下，1 ≤ k ≤ 输入数组的大小。

进阶：你能在线性时间复杂度内解决此题吗?

2. 简单实现

近似线性的解法，不过和只记录最大值索引貌似没什么区别。。。

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> ans;
        if(nums.size() == 0) return ans;
        int a = 0;//窗口内最大值索引
        int b = 1;//窗口内第二大值索引，且满足b > a（用来在a移出滑窗时快速更新a，故b<a时无意义）
        //初始化窗口
        for(int i = 1; i < k; i++)
            if(nums[i] >= nums[a]){
                a = i;
                if(a >= b) b = a+1;
            }
            else if(nums[i] >= nums[b])
                b = i;
        ans.push_back(nums[a]);
        //滑动窗口
        for(int i = k; i < nums.size(); i++){
            if(i-a < k){//当前最大值还在窗口内
                if(nums[i] >= nums[a]){//更新最大值
                    a = i;
                    if(a >= b) b = a+1;
                }
                else if(nums[i] >= nums[b])
                    b = i;
            }
            else{//最大值出窗口
                if(nums[i] >= nums[b]){//新滑入的值成为最大值
                    a = i;
                    b = a + 1;
                }
                else{
                    a = b;//新的最大值为原来的第二大值
```

```
                    b = a+1;//找到新的第二大值
                    for(int j = b+1; j <= i; j++)
                        if(nums[j] >= nums[b])
                            b = j;
                }
            }
            ans.push_back(nums[a]);
        }
        return ans;
    }
};
```

3. 双向队列

想法是用一个数据结构保存各个阶段可能的最大值，使用双向队列：

遍历整个数组。在每一步清理双向队列：

- 只保留当前滑动窗口中有的元素的索引。
- 移除比当前元素小的所有元素，它们在现阶段不可能是最大的了
- 将当前元素添加到双向队列中。
- 将 队列头元素添加到输出中。

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> ans;
        if(nums.size() == 0) return ans;
        deque<int> q;
        for(int i = 0; i < nums.size(); i++){
            while(!q.empty() && nums[q.back()] <= nums[i])
                q.pop_back();
            q.push_back(i);
            if(i-q.front() > k-1)//移除滑动窗口外元素
                q.pop_front();
            if(i >= k-1)
                ans.push_back(nums[q.front()]);
        }
        return ans;
    }
};
```

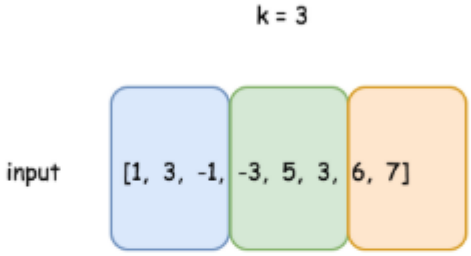
4. 动态规划

方法三: 动态规划

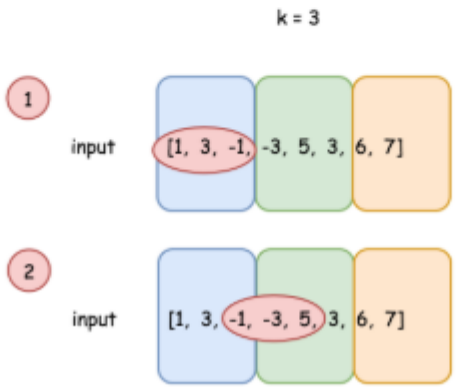
直觉

这是另一个 $O(N)$ 的算法。本算法的优点是不需要使用 `数组 / 列表` 之外的任何数据结构。

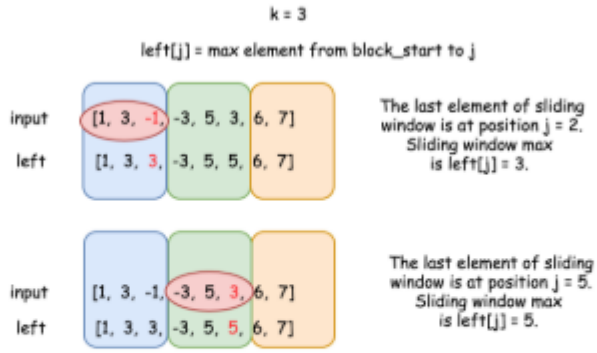
算法的思想是将输入数组分割成有 `k` 个元素的块。
若 `n % k != 0`，则最后一块的元素个数可能更少。



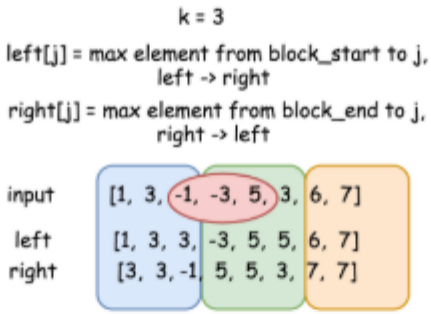
开头元素为 `i`，结尾元素为 `j` 的当前滑动窗口可能在一个块内，也可能在两个块中。



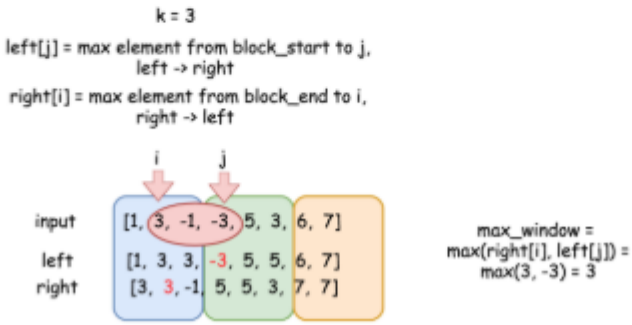
情况 `1` 比较简单。建立数组 `left`，其中 `left[j]` 是从块的开始到下标 `j` 最大的元素，方向 `左->右`。



为了处理更复杂的情况 `2`，我们需要数组 `right`，其中 `right[j]` 是从块的结尾到下标 `j` 最大的元素，方向 `右->左`。`right` 数组和 `left` 除了方向不同以外基本一致。



两数组一起可以提供两个块内元素的全部信息。考虑从下标 `i` 到下标 `j` 的滑动窗口。根据定义，`right[i]` 是左侧块内的最大元素，`left[j]` 是右侧块内的最大元素。因此滑动窗口中的最大元素为 `max(right[i], left[j])`。



76.最小覆盖子串（困难）

1. 题目描述

   给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

   示例：

   ```
   输入：S = "ADOBECODEBANC", T = "ABC"
   输出："BANC"
   ```

   说明：

- 如果 S 中不存这样的子串，则返回空字符串 `""`。
- 如果 S 中存在这样的子串，我们保证它是唯一的答案。

2. 错解题意

理解的题意是即使T中有重复字符，也只要包含一个该字母即可。。。代码如下：

```
class Solution {
public:
    string minWindow(string s, string t) {
        if(t.size() <= 0) return "";
        //统计s中每个字符出现的所有位置
        unordered_map<char, vector<int>> m1;
        for(int i = 0; i < s.size(); i++)
            if(m1.count(s[i]) <= 0)
                m1[s[i]] = {i};
            else
                m1[s[i]].push_back(i);
        //找到t中所有字符在s中出现的位置，由于前面遍历索引是从小到大的，因此这里的索引满足升序排列
        unordered_map<char, vector<int>> m2;
        for(int i = 0; i < t.size(); i++)
            if(m1.count(t[i]) <= 0)//s中无该字符，返回”“
                return "";
            else
                m2[t[i]] = m1[t[i]];//是升序的
        //找到最小串
        string ans = s;//最大就是s
        while(1){
            auto min = m2.begin();//当前t中各字符对应的索引最小值
            auto max = m2.begin();//当前t中各字符对应的索引最大值
            for(auto it = m2.begin(); it != m2.end(); it++){//找到最小最大值
                if(it->second[0] < min->second[0])
                    min = it;
                if(it->second[0] > max->second[0])
                    max = it;
            }
            if(ans.size() > max->second[0] - min->second[0]+1)//两者之间为一个满足条件的
子串
                ans = s.substr(min->second[0], max->second[0] - min->second[0]+1);
            //删去最小索引， 只有这样才有可能缩小子串
            min->second.erase(min->second.begin());
            if(min->second.size() == 0)//其中一个空了，再遍历只可能取得更大的子串，因此停止
                break;
        }

        return ans;
    }
};
```

3. 调整修改

```
class Solution {
public:
    string minWindow(string s, string t) {
```

```
        if(t.size() <= 0) return "";
        if(s.size() < t.size()) return "";
        //统计s中每个字符出现的所有位置
        unordered_map<char, vector<int>> m1;
        for(int i = 0; i < s.size(); i++)
            if(m1.count(s[i]) <= 0)
                m1[s[i]] = {i};
            else
                m1[s[i]].push_back(i);
        //找到t中所有字符在s中出现的位置，由于前面遍历索引是从小到大的，因此这里的索引满足升序排列
        unordered_map<char, vector<int>> m2;
        for(int i = 0; i < t.size(); i++)
            if(m1.count(t[i]) <= 0)//s中无该字符，返回""
                return "";
            else
                m2[t[i]] = m1[t[i]];//是升序的

        priority_queue<int, vector<int>, greater<int>> less;//t中所有字符当前对应的最小索
引
        int max_idx = -1;//t中所有字符当前对应的最大索引
        for(int i = 0; i < t.size(); i++){//先为t中每个字符找到一个最小的索引
            if(m2.count(t[i]) <= 0)//找不到，说明无解
                return "";
            less.push(m2[t[i]][0]);//加入最小堆
            max_idx = max(max_idx, m2[t[i]][0]);
            m2[t[i]].erase(m2[t[i]].begin());//考察过的都删掉
            if(m2[t[i]].size() <= 0)//后面没有索引了，删掉
                m2.erase(t[i]);
        }
        string ans = s.substr(less.top(), max_idx-less.top()+1);
        while(m2.count(s[less.top()]) > 0){
            int cur = less.top();//替换掉最小索引
            less.pop();
            less.push(m2[s[cur]][0]);//替换为其对应字符的下一个最小索引
            max_idx = max(max_idx, m2[s[cur]][0]);
            if(m2[s[cur]].size() > 1)
                m2[s[cur]].erase(m2[s[cur]].begin());
            else
                m2.erase(s[cur]);
            if(ans.size() > max_idx-less.top()+1)
                ans = s.substr(less.top(), max_idx-less.top()+1);
        }
        return ans;
    }
};
```

链表

23.合并K个元素的有序链表（困难）

1. 题目描述

   合并 *k* 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

   示例:

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

2. 简单实现

   优先队列保存当前所有链表的最小值，O(nlogk)

```
class Solution {
public:
    struct cmp{
        bool operator() (ListNode* a, ListNode* b){
            return a->val > b->val;
        }
    };
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if(lists.size() == 0) return NULL;
        if(lists.size() == 1) return lists[0];
        priority_queue<ListNode*, vector<ListNode*>, cmp> q;
        for(int i = 0; i < lists.size(); i++){//所有链表最小元素入队列
            if(lists[i])//防止输入空链表
                q.push(lists[i]);
        }
        ListNode* head = new ListNode(-1);
        ListNode* cur = head;
        while(q.size() > 1) {//为1时表示只剩下一个有序链表
            cur->next = q.top();//最小元素入结果链表
            cur = cur->next;
            q.pop();
            if(cur->next != NULL)//该元素在链表中的下一个元素入队列
                q.push(cur->next);
        }
        if(q.size() == 1)
                cur->next = q.top();
        return head->next;
    }
};
```

148.链表排序（中等）

1. 题目描述

   在 *O*(*n* log *n*) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

```
输入: 4->2->1->3
输出: 1->2->3->4
```

示例 2:

```
输入: -1->5->3->4->0
输出: -1->0->3->4->5
```

2. 简单实现

模拟快排，但是使用了递归，空间貌似不太符合要求

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if(!head) return NULL;
        if(!head->next) return head;
        //以链表头节点做partition
        ListNode* head_cur = head;//值等于链表头节点的值串在head上
        ListNode* left = new ListNode(-1);//值小于链表头节点的值串在left上
        ListNode* left_cur = left;
        ListNode* right = new ListNode(-1);//值大于链表头节点的值串在right上
        ListNode* right_cur = right;
        //遍历partition
        ListNode* cur = head->next;
        while(cur){
            if(cur->val < head_cur->val){
                left_cur->next = cur;
                left_cur = left_cur->next;
            }
            else if(cur->val > head_cur->val){
                right_cur->next = cur;
                right_cur = right_cur->next;
            }
            else{
                head_cur->next = cur;
                head_cur = head_cur->next;
            }
            cur = cur->next;
        }
        left_cur->next = NULL;//必须置NULL拆分开
        right_cur->next = NULL;//必须置NULL拆分开
        head_cur->next = NULL;//必须置NULL拆分开
        ListNode* ans;
        if(left->next){//左链表不空
            left = sortList(left->next);//排序左链表
            left_cur = left;
            while(left_cur->next)//找到其尾部
                left_cur = left_cur->next;
            ans = left;
```

```
            left_cur->next = head;//尾部连接head链表
        }
        else
            ans = head;
        if(right->next){//右链表不空
            right = sortList(right->next);
            head_cur->next = right;//head链表尾部连接right头节点
        }
        return ans;
    }
};
```

3. 最优解法——自底向上归并
    - bottom-to-up 的归并思路是这样的：先两个两个的 merge，完成一趟后，再 4 个4个的 merge，直到结束
    - 链表里操作最难掌握的应该就是各种断链啊，然后再挂接啊。在这里，我们主要用到链表操作的两个技术：
        - merge(l1, l2)，双路归并，我相信这个操作大家已经非常熟练的，就不做介绍了
        - cut(l, n)，可能有些同学没有听说过，它其实就是一种 split 操作，即断链操作。不过我感觉使用 cut 更准确一些，它表示，将链表 l 切掉前 n 个节点，并返回后半部分的链表头
        - 额外再补充一个 dummyHead 大法（虚拟头结点），已经讲过无数次了，仔细体会吧。

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        ListNode dummyHead(0);
        dummyHead.next = head;
        auto p = head;
        int length = 0;
        while (p) {
            ++length;
            p = p->next;
        }
        for (int size = 1; size < length; size <<= 1) {
            auto cur = dummyHead.next;
            auto tail = &dummyHead;
            while (cur) {
                auto left = cur;
                auto right = cut(left, size); // left->@->@ right->@->@->@...
                cur = cut(right, size); // left->@->@ right->@->@  cur->@->...
                tail->next = merge(left, right);
                while (tail->next)
                    tail = tail->next;
            }
        }
        return dummyHead.next;
    }
    ListNode* cut(ListNode* head, int n) {
        auto p = head;
        while (--n && p)
            p = p->next;
```

```
        if (!p) return nullptr;
        auto next = p->next;
        p->next = nullptr;
        return next;
    }
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode dummyHead(0);
        auto p = &dummyHead;
        while (l1 && l2) {
            if (l1->val < l2->val) {
                p->next = l1;
                p = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                p = l2;
                l2 = l2->next;
            }
        }
        p->next = l1 ? l1 : l2;
        return dummyHead.next;
    }
};
```

树和图

127.单词接龙（中等）

1. 题目描述

   给定两个单词（*beginWord* 和 *endWord*）和一个字典，找到从 *beginWord* 到 *endWord* 的最短转换序列的长度。转换需遵循如下规则：

   1. 每次转换只能改变一个字母。
   2. 转换过程中的中间单词必须是字典中的单词。

   说明:

   - 如果不存在这样的转换序列，返回 0。
   - 所有单词具有相同的长度。
   - 所有单词只由小写字母组成。
   - 字典中不存在重复的单词。
   - 你可以假设 *beginWord* 和 *endWord* 是非空的，且二者不相同。

   示例 1:

```
输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",
     返回它的长度 5。
```

示例 2:

```
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: 0

解释: endWord "cog" 不在字典中，所以无法进行转换。
```

2. 简单实现

可以看做图的广度优先遍历问题，每个字符串为图中一个节点，若两个节点表示的字符串只有一位不同，则它们之间有一条边

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        if(wordList.size() <= 0) return 0;
        unordered_map<string, vector<string>> m;//用map存储图，<节点,该节点的所有相邻节点>
        int len = wordList[0].size();
        for(int i = 0; i < wordList.size()-1; i++)
            for(int j = i + 1; j < wordList.size(); j++){//遍历构造图
                int cnt = 0;//不同字符数
                for(int k = 0; k < len && cnt <= 1; k++){
                    if(wordList[i][k] != wordList[j][k])
                        cnt++;
                }
                if(cnt == 1){//相邻节点
                    if(m.count(wordList[i]) <= 0)
                        m[wordList[i]] = {wordList[j]};
                    else
                        m[wordList[i]].push_back(wordList[j]);
                    if(m.count(wordList[j]) <= 0)
                        m[wordList[j]] = {wordList[i]};
                    else
                        m[wordList[j]].push_back(wordList[i]);
                }
            }
        if(m.count(beginWord) <= 0){//起始字符串可以不在字典中，因此加入图中
            for(int i = 0; i < wordList.size(); i++){
```

```
                int cnt = 0;
                for(int k = 0; k < len && cnt <= 1; k++){
                    if(wordList[i][k] != beginWord[k])
                        cnt++;
                }
                if(cnt == 1){
                    if(m.count(beginWord) <= 0)
                        m[beginWord] = {wordList[i]};
                    else
                        m[beginWord].push_back(wordList[i]);
                    if(m.count(wordList[i]) <= 0)
                        m[wordList[i]] = {beginWord};
                    else
                        m[wordList[i]].push_back(beginWord);
                }
            }
        }
        if(m.count(endWord) <= 0) return 0;//endWord不在图中（不存在该节点或该节点无相邻节点）
        //BFS
        int ans = 0;
        queue<string> q;
        q.push(beginWord);
        unordered_set<string> visited;
        visited.insert(beginWord);
        while(!q.empty()){
            ans++;
            int size = q.size();
            for(int i = 0; i < size; i++){
                string cur = q.front();
                q.pop();
                for(auto it = m[cur].begin(); it != m[cur].end(); it++){
                    if(*it == endWord) return ans+1;//注意返回序列长度
                    else if(visited.count(*it) <= 0){
                        q.push(*it);
                        visited.insert(*it);
                    }
                }
            }
        }
        return 0;
    }
};
```

3. 改进一——双向BFS

基本思想就是BFS遍历是一个金字塔型，越往后遍历，塔基越大。而在众多塔基上只有一点（end）是解，其回溯的路径也很少。所以逆向思维自底向上BFS，就有可能减少时空消耗。而双向BFS更加精妙，从begin->end遍历和从end->begin同时进行，当待遍历的队列哪个短先遍历哪个。结束条件是，双向BFS遍历碰头。

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
```

```
        if(wordList.size() <= 0) return 0;
        unordered_map<string, vector<string>> m;//用map存储图，<节点,该节点的所有相邻节点>
        int len = wordList[0].size();
        for(int i = 0; i < wordList.size()-1; i++)
            for(int j = i + 1; j < wordList.size(); j++){//遍历构造图
                int cnt = 0;//不同字符数
                for(int k = 0; k < len && cnt <= 1; k++){
                    if(wordList[i][k] != wordList[j][k])
                        cnt++;
                }
                if(cnt == 1){//相邻节点
                    if(m.count(wordList[i]) <= 0)
                        m[wordList[i]] = {wordList[j]};
                    else
                        m[wordList[i]].push_back(wordList[j]);
                    if(m.count(wordList[j]) <= 0)
                        m[wordList[j]] = {wordList[i]};
                    else
                        m[wordList[j]].push_back(wordList[i]);
                }
            }
        if(m.count(beginWord) <= 0){//起始字符串可以不在字典中，因此加入图中
            for(int i = 0; i < wordList.size(); i++){
                int cnt = 0;
                for(int k = 0; k < len && cnt <= 1; k++){
                    if(wordList[i][k] != beginWord[k])
                        cnt++;
                }
                if(cnt == 1){
                    if(m.count(beginWord) <= 0)
                        m[beginWord] = {wordList[i]};
                    else
                        m[beginWord].push_back(wordList[i]);
                    if(m.count(wordList[i]) <= 0)
                        m[wordList[i]] = {beginWord};
                    else
                        m[wordList[i]].push_back(beginWord);
                }
            }
        }
        if(m.count(endWord) <= 0) return 0;//endWord不在图中（不存在该节点或该节点无相邻节
点)
        //双向BFS
        int ans = 0;
        //begin->end
        queue<string> lq;
        lq.push(beginWord);
        unordered_set<string> lvisited;
        lvisited.insert(beginWord);
        unordered_set<string> l_cur;//当前步数下从beginWord开始所有可能遍历到的点
        l_cur.insert(beginWord);
        //end->begin
        queue<string> rq;
```

```
        rq.push(endWord);
        unordered_set<string> rvisited;
        rvisited.insert(endWord);
        unordered_set<string> r_cur;//当前步数下从endWord开始所有可能遍历到的点
        r_cur.insert(endWord);
        while(!lq.empty() && !rq.empty()){
            //begin->end
            ans++;
            int size = lq.size();
            l_cur.clear();
            for(int i = 0; i < size; i++){
                string cur = lq.front();
                lq.pop();
                for(auto it = m[cur].begin(); it != m[cur].end(); it++){
                    if(r_cur.count(*it) > 0) return ans+1;//注意返回序列长度
                    else if(lvisited.count(*it) <= 0){
                        lq.push(*it);
                        lvisited.insert(*it);
                        l_cur.insert(*it);
                    }
                }
            }
            //end->begin
            ans++;
            size = rq.size();
            r_cur.clear();
            for(int i = 0; i < size; i++){
                string cur = rq.front();
                rq.pop();
                for(auto it = m[cur].begin(); it != m[cur].end(); it++){
                    if(l_cur.count(*it) > 0) return ans+1;//注意返回序列长度
                    else if(rvisited.count(*it) <= 0){
                        rq.push(*it);
                        rvisited.insert(*it);
                        r_cur.insert(*it);
                    }
                }
            }
        }
        return 0;
    }
};
```

4. 改进二——去掉初始化

   初始化耗时`O(n^2*字符串长度)`，但实际所有可能的字符只有26个小写字母，因此可以直接构造哈希表存储所有的字符串，在BFS时依次修改字符串每一位为a-z并判断，耗时`O(n*26*字符串长度)`，对海量数据的情况，这样反而更简单

5. 改进三——简化初始化

   用通用状态做键值，例如dog可以衍生出`*og, d*g, do*`三种键值，BFS时依次遍历将字符串各位通用键值对应的字符串列表即可

130. 被围绕的区域（中等）

1. 题目描述

   给定一个二维的矩阵，包含 `'X'` 和 `'O'`（**字母 O**）。找到所有被 `'X'` 围绕的区域，并将这些区域里所有的 `'O'` 用 `'X'` 填充。

   示例:

   ```
   X X X X
   X O O X
   X X O X
   X O X X
   ```

   运行你的函数后，矩阵变为：

   ```
   X X X X
   X X X X
   X X X X
   X O X X
   ```

   解释:被围绕的区间不会存在于边界上，换句话说，任何边界上的 `'O'` 都不会被填充为 `'X'`。 任何不在边界上，或不与边界上的 `'O'` 相连的 `'O'` 最终都会被填充为 `'X'`。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

2. 简单实现

   题目中的解释给了很强的暗示：从边界的O做BFS，能遍历到的就不变，遍历不到的就变X

   ```
   class Solution {
   public:
       void solve(vector<vector<char>>& board) {
           int m = board.size();
           if(m <= 2) return;
           int n = board[0].size();
           if(n <= 2) return;
           queue<pair<int, int>> q;
           vector<vector<bool>> visited = vector<vector<bool>>(m, vector<bool>(n,
   false));
           //找到边界的O作为起点
           for(int i = 0; i < m; i++){
               if(board[i][0] == 'O'){
                   visited[i][0] = true;
                   q.push(make_pair(i,0));
               }
               if(board[i][n-1] == 'O'){
                   visited[i][n-1] = true;
                   q.push(make_pair(i,n-1));

               }
           }
           for(int j = 0; j < n; j++){
   ```

```
            if(board[0][j] == 'O'){
                visited[0][j] = true;
                q.push(make_pair(0,j));
            }
            if(board[m-1][j] == 'O'){
                visited[m-1][j] = true;
                q.push(make_pair(m-1,j));
            }
        }
        //BFS
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                int x = q.front().first;
                int y = q.front().second;
                q.pop();
                if(x-1>=0 && !visited[x-1][y] && board[x-1][y]=='O'){
                    visited[x-1][y] = true;
                    q.push(make_pair(x-1,y));
                }
                if(x+1<m && !visited[x+1][y] && board[x+1][y]=='O'){
                    visited[x+1][y] = true;
                    q.push(make_pair(x+1,y));
                }
                 if(y-1>=0 && !visited[x][y-1] && board[x][y-1]=='O'){
                    visited[x][y-1] = true;
                    q.push(make_pair(x,y-1));
                }
                if(y+1<n && !visited[x][y+1] && board[x][y+1]=='O'){
                    visited[x][y+1] = true;
                    q.push(make_pair(x,y+1));
                }
            }
        }
        //未访问过的O变X
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(board[i][j] == 'O' && visited[i][j] == false)
                    board[i][j] = 'X';
    }
};
```

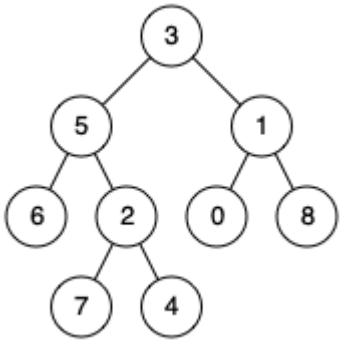
TIP：也可以不用visited，在BFS时把O变成其他字符以作区别即可

236.二叉树的最近公共祖先（中等）

1. 题目描述

   给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

   例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。
```

示例 2:

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

2. 简单实现

用哈希映射记录每个节点在树的中序遍历数组中的索引，再从根节点开始根据索引的分布判断，该算法不管节点值是否唯一都可以正常工作

```
class Solution {
public:
    void init(TreeNode* root, unordered_map<TreeNode*, int>& m, int& idx){
        if(!root) return;
        init(root->left, m, idx);
        m[root] = idx++;
        init(root->right, m, idx);
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        unordered_map<TreeNode*, int> m;//<节点，中序遍历索引>
        int idx = 0;
        init(root, m, idx);//构造m
        TreeNode* cur = root;
        while(cur){
            if(m[cur] == m[p] || m[cur] == m[q])//当前节点为p和q中的一个
                return cur;
            else if(m[cur] < m[p] && m[cur] < m[q])//p,q都在当前节点的右子树上
                cur = cur->right;
            else if(m[cur] > m[p] && m[cur] > m[q])//p,q都在当前节点的左子树上
                cur = cur->left;
```

```
            else//p,q在当前节点的两则
                return cur;
        }
        return NULL;
    }
};
```

3. 自我改进

上面的算法遍历了整棵树，但实际上我们关心的只有p,q两个节点和它们的相对位置，因此：

- 当遍历到p（q同理）时，就不必再遍历其右子树，这样可能导致的情况是
  - 节点q在右子树上，这样最终的哈希映射无q的值，因此返回p
  - 节点q不在右子树上，则去除右子树也不会对p和q在中序遍历中的相对位置产生影响，判断规则依然成立

```
class Solution {
public:
    void init(TreeNode* root, unordered_map<TreeNode*, int>& m, int& idx){
        if(!root) return;
        init(root->left, m, idx);
        m[root] = idx++;
        if(root == p || root == q)//添加判断语句
            return;
        else
            init(root->right, m, idx, p, q);
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        unordered_map<TreeNode*, int> m;//<节点，中序遍历索引>
        int idx = 0;
        init(root, m, idx);//构造m
        TreeNode* cur = root;
        while(cur){
            if(m[cur] == m[p] || m[cur] == m[q])//当前节点为p和q中的一个
                return cur;
            else if(m[cur] < m[p] && m[cur] < m[q])//p,q都在当前节点的右子树上
                cur = cur->right;
            else if(m[cur] > m[p] && m[cur] > m[q])//p,q都在当前节点的左子树上
                cur = cur->left;
            else//p,q在当前节点的两则
                return cur;
        }
        return NULL;
    }
};
```

4. 最简代码——递归

在左、右子树中分别查找是否包含p或q：

- 如果以下两种情况（左子树包含p，右子树包含q/左子树包含q，右子树包含p），那么此时的根节点就是最近公共祖先
- 如果左子树包含p和q，那么到root->left中继续查找，最近公共祖先在左子树里面
- 如果右子树包含p和q，那么到root->right中继续查找，最近公共祖先在右子树里面

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == nullptr || root == p || root == q){return root; }
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        return left == nullptr? right : (right == nullptr? left : root);
    }
```

5. 官方题解——构造父节点字典
    - 根节点开始遍历树。
    - 在找到 p 和 q 之前，将父指针存储在字典中。
    - 一旦我们找到了 p 和 q，我们就可以使用父亲字典获得 p 的所有祖先，并添加到一个称为祖先的集合中。
    - 同样，我们遍历节点 q 的祖先。如果祖先存在于为 p 设置的祖先中，这意味着这是 p 和 q 之间的第一个共同祖先（同时向上遍历），因此这是 LCA 节点。

124.二叉树中的最大路径和（困难）

1. 题目描述

    给定一个**非空**二叉树，返回其最大路径和。

    本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径**至少包含一个**节点，且不一定经过根节点。

    示例 1:

```
输入: [1,2,3]

       1
      / \
     2   3

输出: 6
```

    示例 2:

```
输入: [-10,9,20,null,null,15,7]

   -10
   / \
  9  20
    /  \
   15   7

输出: 42
```

2. 递归实现

解题思路：

二叉树 abc，a 是根结点（递归中的 root），bc 是左右子结点（代表其递归后的最优解）。
最大的路径，可能的路径情况：

```
   a
  / \
 b   c
```

1. b + a + c。
2. b + a + a 的父结点。
3. a + c + a 的父结点。

其中情况 1，表示如果不联络父结点的情况，或本身是根结点的情况。
这种情况是没法递归的，但是结果有可能是全局最大路径和。
情况 2 和 3，递归时计算 `a+b` 和 `a+c`，选择一个更优的方案返回，也就是上面说的递归后的最优解啦。

另外结点有可能是负值，最大和肯定就要想办法舍弃负值（$max(0, x)$）。
但是上面 3 种情况，无论哪种，a 作为联络点，都不能够舍弃。

代码中使用 val 来记录全局最大路径和。
`ret` 是情况 2 和 3。
`lmr` 是情况 1。

所要做的就是递归，递归时记录好全局最大和，返回联络最大和。

```
class Solution {
public:
    int maxPathSum(TreeNode* root, int &val) {
        if (root == nullptr) return 0;
        int left = maxPathSum(root->left, val);
        int right = maxPathSum(root->right, val);
        int lmr = root->val + max(0, left) + max(0, right);
        int ret = root->val + max(0, max(left, right));
        val = max(val, max(lmr, ret));
        return ret;
    }
    int maxPathSum(TreeNode* root) {
        int val = INT_MIN;
        maxPathSum(root, val);
        return val;
    }
};
```

547.朋友圈（中等）

---

1. 题目描述

班上有 **N** 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 **N * N** 的矩阵 **M**，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生**互为**朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

示例 1:

```
输入:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
输出: 2
说明：已知学生0和学生1互为朋友，他们在一个朋友圈。
第2个学生自己在一个朋友圈。所以返回2。
```

示例 2:

```
输入:
[[1,1,0],
 [1,1,1],
 [0,1,1]]
输出: 1
说明：已知学生0和学生1互为朋友，学生1和学生2互为朋友，所以学生0和学生2也是朋友，所以他们三个在一个朋友圈，返回1。
```

注意:

1. N 在[1,200]的范围内。
2. 对于所有学生，有M[i][i] = 1。
3. 如果有M[i][j] = 1，则有M[j][i] = 1。

2. 简单实现

并查集

```
class UnionFind{
private:
    vector<int> father;
    int size;
public:
    UnionFind(int n){
        father = vector<int>(n);
        size = n;
        for(int i = 0; i < n; i++)
            father[i] = i;
    }
    int get(int x){
        if(father[x] == x)
            return x;
        return father[x] = get(father[x]);//路径压缩
    }
    void merge(int x, int y){
        x = get(x);
```

```
        y = get(y);
        if(x != y){
            father[y] = x;
            size--;
        }
    }
    int getSize() {return size;}
};
class Solution {
public:
    int findCircleNum(vector<vector<int>>& M) {
        int size = M.size();
        UnionFind u(size);
        for(int i = 0; i < size-1; i++)
            for(int j = i + 1; j < size; j++)
                if(M[i][j] == 1)
                    u.merge(i, j);
        return u.getSize();
    }
};
```

3. 其他解法

   把矩阵看作是图的邻接矩阵，用BFS/DFS计算图的连通块数

207.课程表（中等）

1. 题目描述

   现在你总共有 *n* 门课需要选，记为 `0` 到 `n-1`。

   在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: `[0,1]`

   给定课程总量以及它们的先决条件，判断是否可能完成所有课程的学习?

   示例 1:

```
输入: 2, [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。
```

   示例 2:

```
输入: 2, [[1,0],[0,1]]
输出: false
解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。
```

   说明:

   1. 输入的先决条件是由**边缘列表**表示的图形，而不是邻接矩阵。详情请参见图的表示法。
   2. 你可以假定输入的先决条件中没有重复的边。

提示:

1. 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2. 通过 DFS 进行拓扑排序 - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3. 拓扑排序也可以通过 BFS 完成。

2. 简单实现

拓扑排序

```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        unordered_map<int, vector<int>> m;//构造邻接表法表示的图
        vector<int> incnt = vector<int>(numCourses, 0);//记录每个节点的入度
        //初始化
        for(int i = 0; i < prerequisites.size(); i++){
            incnt[prerequisites[i][0]]++;
            if(m.count(prerequisites[i][1]) <= 0)
                m[prerequisites[i][1]] = {prerequisites[i][0]};
            else
                m[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }
        //拓扑排序
        queue<int> q;
        int done = 0;//记录可以修的课程数
        for(int i = 0; i < numCourses; i++)
            if(incnt[i] == 0){//0入度的进队列
                q.push(i);
                        done++;
            }
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                int cur = q.front();
                q.pop();
                for(int j = 0; j < m[cur].size(); j++){
                    if(--incnt[m[cur][j]] == 0){//所有下一个节点入度-1后为0则入队列
                        q.push(m[cur][j]);
                        done++;
                        }
                }
            }
        }
        if(done == numCourses)
            return true;
        else
            return false;
    }
};
```

210.课程表 II（中等）

1. 题目描述

现在你总共有 *n* 门课需要选，记为 `0` 到 `n-1`。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: `[0,1]`

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

```
输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。
```

示例 2:

```
输入: 4, [[1,0],[2,0],[3,1],[3,2]]
输出: [0,1,2,3] or [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
     因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。
```

说明:

1. 输入的先决条件是由**边缘列表**表示的图形，而不是邻接矩阵。详情请参见图的表示法。
2. 你可以假定输入的先决条件中没有重复的边。

提示:

1. 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2. 通过 DFS 进行拓扑排序 - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3. 拓扑排序也可以通过 BFS 完成。

2. 简单实现

在上题基础上稍作修改即可，用ans数组记录每次修的课程

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        unordered_map<int, vector<int>> m;//构造邻接表法表示的图
        vector<int> incnt = vector<int>(numCourses, 0);//记录每个节点的入度
        //初始化
        for(int i = 0; i < prerequisites.size(); i++){
            incnt[prerequisites[i][0]]++;
            if(m.count(prerequisites[i][1]) <= 0)
                m[prerequisites[i][1]] = {prerequisites[i][0]};
            else
                m[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }
        //拓扑排序
        queue<int> q;
        vector<int> ans;//记录修完的课程
```

```
        for(int i = 0; i < numCourses; i++)
            if(incnt[i] == 0){//0入度的进队列
                q.push(i);
                        ans.push_back(i);
            }
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                int cur = q.front();
                q.pop();
                for(int j = 0; j < m[cur].size(); j++){
                    if(--incnt[m[cur][j]] == 0){//所有下一个节点入度-1后为0则入队列
                        q.push(m[cur][j]);
                        ans.push_back(m[cur][j]);
                        }
                }
            }
        }
        if(ans.size() == numCourses)
            return ans;
        else
            return vector<int>();
    }
};
```

329.矩阵中的最长递增路径（困难）

1. 题目描述

   给定一个整数矩阵，找出最长递增路径的长度。

   对于每个单元格，你可以往上，下，左，右四个方向移动。 你不能在对角线方向上移动或移动到边界外（即不允许环绕）。

   示例 1:

```
输入: nums =
[
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
输出: 4
解释: 最长递增路径为 [1, 2, 6, 9]。
```

   示例 2:

```
输入: nums =
[
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
输出: 4
解释: 最长递增路径是 [3, 4, 5, 6]。注意不允许在对角线方向上移动。
```

2. 简单实现

dfs+记忆化，时空复杂度都是O(mn)

```
class Solution {
public:
    vector<vector<int>> maxlen;//记录以每个节点出发的最长升序序列
    vector<vector<int>> dirs = {{-1,0}, {1,0}, {0,-1}, {0,1}};
    int dfs(vector<vector<int>>& matrix, int x, int y){
        if(maxlen[x][y] != 0)//已有答案直接返回
            return maxlen[x][y];
        for(int i = 0; i < 4; i++){//找到上下左右四个方向相邻节点的最长升序列
            int m = x + dirs[i][0];
            int n = y + dirs[i][1];
            if(m >= 0 && m < matrix.size()
               && n >= 0 && n < matrix[0].size()
               && matrix[x][y] > matrix[m][n])
                maxlen[x][y] = max(maxlen[x][y], dfs(matrix, m, n));
        }
        return ++maxlen[x][y];//还要加上本节点的长度
    }
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if(m == 0) return 0;
        int n = matrix[0].size();
        if(n == 0) return 0;
        maxlen = vector<vector<int>>(m, vector<int>(n, 0));
        int ans = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                ans = max(ans, dfs(matrix, i,j));
        return ans;
    }
};
```

3. 剥洋葱动态规划

方法三：“剥洋葱”（动态规划） 【通过】

直觉

每个细胞的结果只与相邻的结果相关，能否使用动态规划？

算法

如果我们定义从单元格 (i, j) 开始的最长递增路径为函数

$$f(i, j)$$

则可以写出状态转移函数

$$f(i,j) = max\{f(x,y) | (x,y) \text{ is a nei} * * * \text{ or of} (i,j) \text{ and matrix}[x][y] > \text{matrix}[i][j]\} + 1$$

此公式与以前方法中使用的公式相同。有了状态转移函数，你可能会觉得可以使用动态规划来推导出所有结果，去他的深度优先搜索!

这听起来很美好，可惜你忽略了一件事：我们没有依赖列表。

想要让动态规划有效，如果问题 B 依赖于问题 A 的结果，就必须确保问题 A 比问题 B先计算。这样的依赖顺序对许多问题十分简单自然。如著名的斐波那契数列：

$$F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

子问题 $F(n)$ 依赖于 $F(n-1)$ 和 $F(n-2)$。因此，自然顺序就是正确的计算顺序。被依赖者总会先被计算。

这种依赖顺序的术语是“拓扑顺序”或“拓扑排序”：

> 对有向无环图的拓扑排序是顶点的一个线性排序，使得对于任何有向边 (u, v)，顶点 u 都在 顶点 v 的前面。

在本问题中，拓扑顺序并不简单自然。没有矩阵的值，我们无法知道两个邻居 A 和 B 的依赖关系。作为预处理，我们必须显式执行拓扑排序。之后，我们可以按照存储的拓扑顺序使用状态转移函数动态地解决问题。

有多种实现拓扑排序的方法。这里我们使用的是一种被称为“剥洋葱”的方法。其思路是在一个有向无环图中，会有一些不依赖于其他顶点的顶点，称为“叶子”。我们将这些叶子放在一个列表中（他们的内部排序不重要），然后将他们从图中移除。移除之后，会产生新的“叶子”。重复以上过程，就像一层一层一层地拨开洋葱的心。最后，列表中就会存储有效的拓扑排序。

在本问题中，因为我们想要求出在整个图中最长的路径，也就是“洋葱”的层总数。因此，我们可以在“剥离”的期间计算层数，在不调用动态规划的情况下返回计数。

4. 低->高动态规划

1. 将所有的坐标按照对应位置的值升序排序
2. 依次处理上一步中得到的坐标列表,从附近比当前点矮的点DP值取最大加1得到当前点的DP值、
   第一步的时间复杂度是 O(log(mn))
   第二步的时间复杂度是 O(mn)

```
class Point {
public:
    int x, y, v;
    Point(int ix, int iy, int iv) {
        x = ix;
        y = iy;
        v = iv;
```

```
    }
};

class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        if (matrix.size() == 0)
            return 0;
        const int rows = matrix.size();
        const int cols = matrix[0].size();
        vector<vector<int>> dp(rows, vector<int>(cols, 1));
        vector<Point> vp;
        vp.reserve(rows * cols);
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++) {
                vp.push_back(Point(i, j, matrix[i][j]));
            }
        // 对坐标排序
        sort(vp.begin(), vp.end(), [](const Point &lhs, const Point &rhs){
            return lhs.v < rhs.v;
        });
        int ret = 1;
        for (const auto &p : vp) {
            auto x = p.x;
            auto y = p.y;
            auto v = p.v;
            // 寻找附近比当前矮的点的最高DP值
            if (x > 0 && matrix[x - 1][y] < v && dp[x - 1][y] + 1 > dp[x][y])
                dp[x][y] = dp[x - 1][y] + 1;
            if (x < rows - 1 && matrix[x + 1][y] < v && dp[x + 1][y] + 1 > dp[x]
[y])
                dp[x][y] = dp[x + 1][y] + 1;
            if (y > 0 && matrix[x][y - 1] < v && dp[x][y - 1] + 1 > dp[x][y])
                dp[x][y] = dp[x][y - 1] + 1;
            if (y < cols - 1 && matrix[x][y + 1] < v && dp[x][y + 1] + 1 > dp[x]
[y])
                dp[x][y] = dp[x][y + 1] + 1;
            if (dp[x][y] > ret)
                ret = dp[x][y];
        }
        return ret;
    }
};
```

315.计算右侧小于当前元素的个数（困难）

1. 题目描述

   给定一个整数数组 *nums*，按要求返回一个新数组 *counts*。数组 *counts* 有该性质： `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

   示例:

```
输入: [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1).
2 的右侧仅有 1 个更小的元素 (1).
6 的右侧有 1 个更小的元素 (1).
1 的右侧有 0 个更小的元素.
```

2. 简单实现

从右到左维护一个有序数列，每次将nums[i]插入该序列中，其索引值就是其右侧小于自己的元素个数，用二分法插入，整体时间复杂度O(n(n +logn))（插入的复杂度是o(n），空间复杂度O(n)

```
class Solution {
public:
    int insert(vector<int>& nums, int n){//将n插入有序数组nums中，并返回其在数组中的索引值
        int l = 0;
        int r = nums.size() - 1;
        while(l <= r){
            int mid = l + (r - l) / 2;
            if(nums[mid] < n)//如果存在相等的值要放在最左边，这里不取等号
                l = mid + 1;
            else
                r = mid - 1;
        }
        nums.insert(nums.begin()+l, n);
        return l;
    }
    vector<int> countSmaller(vector<int>& nums) {
        int len = nums.size();
        vector<int> ans = vector<int>(len, 0);
        if(len <= 1) //长度为1则答案已有
            return ans;
        vector<int> temp = {nums[len-1]};//最后一个数放入
        for(int i = len-2; i >= 0; i--)
            ans[i] = insert(temp, nums[i]);
        return ans;
    }
};
```

其他诸如归并排序（与求逆序对差不多），二叉搜索树等的算法在时间上可以达到o(nlogn)的