

1 平时刷

1.1 414 第三大的数

[第三大的数](#)

给你一个非空数组，返回此数组中 **第三大的数**。如果不存在，则返回数组中最大的数。

示例 1:

```
1 输入: [3, 2, 1]
2 输出: 1
3 解释: 第三大的数是 1。
```

示例 2:

```
1 输入: [1, 2]
2 输出: 2
3 解释: 第三大的数不存在，所以返回最大的数 2。
```

示例 3:

```
1 输入: [2, 2, 3, 1]
2 输出: 1
3 解释: 注意，要求返回第三大的数，是指在所有不同数字中排第三大的数。
4 此例中存在两个值为 2 的数，它们都排第二。在所有不同数字中排第三大的数为 1。
```

题解:

可以创建三个变量，用来储存第一、第二、第三大的数，遍历一次数组 $O(n)$ 即可实现:

```
1 class Solution {
2 public:
3     int thirdMax(vector<int>& nums) {
4         long long first = LONG_MIN, second = LONG_MIN, third = LONG_MIN;
5         for (auto &a : nums){ //遍历数组
6             if(a > first){ //最大的数
7                 third = second;
8                 second = first;
9                 first = a;
10            }
11            else if(a < first && a > second){ //第二大的数
12                third = second;
13                second = a;
14            }
15            if(a > third && a < second){ //第三大的数
16                third = a;
17            }
18        }
19        if(third == LONG_MIN)
20            return first;
21        else
22            return third;
23    }
24 };
```

1.2 581最短无序连续子数组

[581. 最短无序连续子数组 - 力扣 \(LeetCode\)](#)

给你一个整数数组 `nums`，你需要找出一个 **连续子数组**，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

请你找出符合题意的 **最短** 子数组，并输出它的长度。

示例 1:

```
1 输入: nums = [2,6,4,8,10,9,15]
2 输出: 5
3 解释: 你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。
```

示例 2:

```
1 输入: nums = [1,2,3,4]
2 输出: 0
```

示例 3:

```
1 输入: nums = [1]
2 输出: 0
```

题解:

首先创建两个变量`left`和`right`分别来表示符合题意子数组的左右边界，两个变量`max`和`min`来表示遍历数组得到的最大值和最小值，设`max`初始值为`nums[0]`，`min`初始值为`nums[len-1]`。

从`nums[i]` ($i = 1$)开始遍历数组，如果`nums[i] > max`说明`nums[0~i]`是升序，将`max`的值更新成`nums[i]`，如果`nums[i] < max`，说明`nums[0~(i-1)]`是升序，可以不管前面那部分，第 i 项数组变成了无序，所以令`right = i`，继续遍历数组，重复以上操作。在同一个`for`循环内可以同时实现从右往左的遍历，逻辑差不多，最后得到`left`和`right`后判断`right`是否大于`left`，不大于说明不需要改动，输出0，大于则输出`right - left + 1`

```
1  class Solution {
2  public:
3      int findUnsortedSubarray(vector<int>& nums) {
4          if(nums.size() <= 1)
5              return 0;
6          int len = nums.size(), low = 0, high = len - 1, maxNum = nums[0], minNum =
nums[len-1];
7          for(int i = 1; i < len; i++){
8              maxNum = max(maxNum, nums[i]);
9              minNum = min(minNum, nums[len - 1 - i]);
10             if(nums[i] < maxNum)
11                 low = i;
12             if(nums[len - 1 - i] > minNum)
13                 high = len - 1 - i;
14         }
15         return low > high ? low - high + 1 : 0;
16     }
17 };
```

1.3 605 种花问题

[605. 种花问题 - 力扣 \(LeetCode\)](#)

假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给你一个整数数组 `flowerbed` 表示花坛，由若干 `0` 和 `1` 组成，其中 `0` 表示没种植花，`1` 表示种植了花。另有一个数 `n`，能否在不打破种植规则的情况下种入 `n` 朵花？能则返回 `true`，不能则返回 `false`。

示例 1:

```
1 输入: flowerbed = [1,0,0,0,1], n = 1
2 输出: true
```

示例 2:

```
1 输入: flowerbed = [1,0,0,0,1], n = 2
2 输出: false
```

题解:

防御式编程思想：在 `flowerbed` 数组两端各增加一个 0，这样处理的好处在于不用考虑边界条件，任意位置处只要连续出现三个 0 就可以栽上一棵花。

```
1  class Solution {
2  public:
3      bool canPlaceFlowers(vector<int>& flowerbed, int n) {
4
5
6          int len = 1, ans = 0;                //认为左边界提供1个0
7          for (auto &i : flowerbed) {
8              if (i) { //为1,遇到1了
9                  ans += (len - 1) / 2;        //len个0可以种这么多花
10                 len = 0;
11             }
12             else { //为0
13                 ++len;
14             }
15         }
16         ans += (len) / 2;                    //处理0尾, 认为右边界提供一个0
17         return ans >= n;
18     }
19 }
20 };
```

1.4 628 三个数的最大乘积

[628. 三个数的最大乘积 - 力扣 \(LeetCode\)](#)

给你一个整型数组 `nums`，在数组中找出由三个数组成的最大乘积，并输出这个乘积。

示例 1:

```
1 输入: nums = [1,2,3]
2 输出: 6
```

示例 2:

```
1 输入: nums = [1,2,3,4]
2 输出: 24
```

示例 3:

```
1 输入: nums = [-1,-2,-3]
2 输出: -6
```

题解:

创建五个变量分别存储最大值、第二大值、第三大值、最小值、第二小值

① 若thirdMax > 0, 毫无疑问最大的结果为firstMax * secondMax * thirdMax

② 若只有thirdMax <= 0, 则上述结果不一定最大, 可与firstMin * secondMin * firstMax比较

③ 若只有firstMax > 0, 则上述两个结果也无法确定, 可计算出结果比较

综上, 我们可以将两个结果算出来直接选最大的

```
1  class Solution {
2  public:
3      int maximumProduct(vector<int>& nums) {
4          int min1= INT_MAX, min2 = INT_MAX;
5          int max1 = INT_MIN, max2 = INT_MIN, max3 = INT_MIN;
6          for(int x : nums){
7              if(x < min1){
8                  min2 = min1;
9                  min1 = x;
10             }
11             else if(x < min2)
12                 min2 = x;
13             if(x > max1){
14                 max3 = max2;
15                 max2 = max1;
16                 max1 =x;
17             }
18             else if(x > max2){
19                 max3 = max2;
20                 max2 = x;
21             }
22             else if(x > max3)
23                 max3 = x;
24         }
25
26
27         return max(min1*min2*max1, max1*max2*max3);
28     }
29 };
```

1.5 643 子数组最大平均数I

[643. 子数组最大平均数 I - 力扣 \(LeetCode\)](#)

给你一个由 `n` 个元素组成的整数数组 `nums` 和一个整数 `k`。

请你找出平均数最大且 长度为 `k` 的连续子数组, 并输出该最大平均数。

任何误差小于 `10-5` 的答案都将被视为正确答案。

示例 1:

```
1 输入: nums = [1,12,-5,-6,50,3], k = 4
2 输出: 12.75
3 解释: 最大平均数 (12-5-6+50)/4 = 51/4 = 12.75
```

示例 2:

```
1 输入: nums = [5], k = 1
2 输出: 5.00000
```

题解:

创建一个变量max用来保存长度为K的连续子数组的元素的和的最大值，然后一直遍历就OK了

```
1  class Solution {
2  public:
3      double findMaxAverage(vector<int>& nums, int k) {
4          int sum=0, i=0;
5          int n = nums.size();
6          for(i = 0; i < k; i++){
7              sum += nums[i];
8          }
9          int maxSum = sum;
10         for(i = k; i < n; i++){
11             sum = sum + nums[i] - nums[i-k];
12             maxSum = max(maxSum, sum);
13         }
14         return static_cast<double>(maxSum)/k;
15     }
16 };
```

1.6 665 非递减数列

[665. 非递减数列 - 力扣 \(LeetCode\)](#)

给你一个长度为 `n` 的整数数组 `nums`，请你判断在 **最多** 改变 `1` 个元素的情况下，该数组能否变成一个非递减数列。

我们是这样定义一个非递减数列的：对于数组中任意的 `i` ($0 \leq i \leq n-2$)，总满足 `nums[i] <= nums[i + 1]`。

示例 1:

```
1 输入: nums = [4,2,3]
2 输出: true
3 解释: 你可以通过把第一个 4 变成 1 来使得它成为一个非递减数列。
```

示例 2:

```
1 输入: nums = [4,2,1]
2 输出: false
3 解释: 你不能在只改变一个元素的情况下将其变为非递减数列。
```

题解:

假设一个数组有两对 $\text{nums}[i] > \text{nums}[i + 1]$ ，那这样结果一定为false

例如：[3, 2, 1]、[4, 2, 1, 3]

假设一个数组只有一堆 $\text{nums}[i] > \text{nums}[i + 1]$ ，结果也不一定为true

例如：[3, 4, 1, 2]

创建一个计数器cnt来存有几对

可见修改值之后还得再判断是否是非递减数列，但是老是遍历判断太费时间了，那还可以咋做呢？

假设现在 $\text{nums}[i] > \text{nums}[i + 1]$ ，则有两种改法， $\text{nums}[i] = \text{nums}[i + 1]$ 或者 $\text{nums}[i + 1] = \text{nums}[i]$

对于前者，改完必须还要保证 $\text{nums}[i] \geq \text{nums}[i - 1]$ ，既 $\text{nums}[i + 1] \geq \text{nums}[i - 1]$ ，要不然就只能改 $\text{nums}[i + 1]$ ，

而对于前者改完我们也判断不到了，不影响整体判断，因为for循环i++，i已经到后面去了，所以我们不用处理前面的情况，只要处理一下改 $\text{nums}[i + 1]$ 的情况，毕竟改完了我们还会遍历到i+1这个元素，看看改之后是否符合要求。在每次cnt++之后看看是否大于1，

```
1  class Solution {
2  public:
3      bool checkPossibility(vector<int>& nums) {
4          int n = nums.size(), cnt = 0;
5          for(int i = 0; i < n-1; i++){
6              int x = nums[i], y = nums[i + 1];
7              if(x > y){
8                  cnt++;
9                  if(cnt > 1){
10                     return false;
11                 }
12                 if(i > 0 && y < nums[i - 1]){
13                     nums[i + 1] = x;
14                 }
15             }
16         }
17         return true;
18     }
19 };
```

1.7 674 最长连续递增序列

[674. 最长连续递增序列 - 力扣 \(LeetCode\)](#)

给定一个未经排序的整数数组，找到最长且 **连续递增的子序列**，并返回该序列的长度。

连续递增的子序列 可以由两个下标 l 和 r ($l < r$) 确定，如果对于每个 $l \leq i < r$ ，都有 $\text{nums}[i] < \text{nums}[i + 1]$ ，那么子序列 $[\text{nums}[l], \text{nums}[l + 1], \dots, \text{nums}[r - 1], \text{nums}[r]]$ 就是连续递增子序列。

示例 1:

```
1  输入: nums = [1,3,5,4,7]
2  输出: 3
3  解释: 最长连续递增序列是 [1,3,5]，长度为3。
4  尽管 [1,3,5,7] 也是升序的子序列，但它不是连续的，因为 5 和 7 在原数组里被 4 隔开。
```

示例 2:

```
1 输入: nums = [2,2,2,2,2]
2 输出: 1
3 解释: 最长连续递增序列是 [2], 长度为1。
```

题解:

遍历, 把符合条件的子数组长度求出来取最大值就OK

```
1  class Solution {
2  public:
3      int findLengthOfLCIS(vector<int>& nums) {
4          int l = 0, r = 0, n = nums.size(), maxSize = 1, maxNum = INT_MIN;
5          for(int i = 0; i < n ; i++){
6              r = i;
7              if(maxNum >= nums[i]){
8                  maxSize = max(maxSize, r - l);
9                  l = r;
10             }
11             maxNum = nums[i];
12         }
13         maxSize = max(maxSize, r - l + 1 );
14         return maxSize;
15     }
16 };
```

1.8 697 数组的度

[697. 数组的度 - 力扣 \(LeetCode\)](#)

给定一个非空且只包含非负数的整数数组 `nums`, 数组的 **度** 的定义是指数组里任一元素出现频数的最大值。

你的任务是在 `nums` 中找到与 `nums` 拥有相同大小的度的最短连续子数组, 返回其长度。

示例 1:

```
1 输入: nums = [1,2,2,3,1]
2 输出: 2
3 解释:
4 输入数组的度是 2 , 因为元素 1 和 2 的出现频数最大, 均为 2 。
5 连续子数组里面拥有相同度的有如下所示:
6 [1, 2, 2, 3, 1], [1, 2, 2, 3], [2, 2, 3, 1], [1, 2, 2], [2, 2, 3], [2, 2]
7 最短连续子数组 [2, 2] 的长度为 2 , 所以返回 2 。
```

示例 2:

```
1 输入: nums = [1,2,2,3,1,4,2]
2 输出: 6
3 解释:
4 数组的度是 3 , 因为元素 2 重复出现 3 次。
5 所以 [2,2,3,1,4,2] 是最短子数组, 因此返回 6 。
```

题解:

哈希表, 存储每个值出现的次数、第一次出现的位置、最后一次出现的位置

```
1  class Solution {
2  public:
3      int findShortestSubArray(vector<int>& nums) {
```

```

4         unordered_map<int, vector<int >> mp;
5         int n = nums.size();
6         for(int i = 0; i < n; i++){
7             if(mp.count(nums[i])){
8                 mp[nums[i]][0]++;
9                 mp[nums[i]][2] = i;
10            }
11            else{
12                mp[nums[i]] = {1, i, i};
13            }
14        }
15        int maxNum = 0, minLen = 0;
16        for(auto&[_ , vec] : mp){
17            if(maxNum < vec[0]){
18                maxNum = vec[0];
19                minLen = vec[2] - vec[1] + 1;
20            }
21            else if(maxNum == vec[0]){
22                if(minLen > vec[2] - vec[1] + 1){
23                    minLen = vec[2] - vec[1] + 1;
24                }
25            }
26        }
27        return minLen;
28    }
29 };

```

1.9 717 1比特与2比特字符

[717. 1 比特与 2 比特字符 - 力扣 \(LeetCode\)](#)

有两种特殊字符：

- 第一种字符可以用一比特 `0` 表示
- 第二种字符可以用两比特 (`10` 或 `11`) 表示

给你一个以 `0` 结尾的二进制数组 `bits`，如果最后一个字符必须是一个一比特字符，则返回 `true`。

示例 1:

```

1  输入: bits = [1, 0, 0]
2  输出: true
3  解释: 唯一的解码方式是将其解析为一个两比特字符和一个一比特字符。
4  所以最后一个字符是一比特字符。

```

示例 2:

```

1  输入: bits = [1,1,1,0]
2  输出: false
3  解释: 唯一的解码方式是将其解析为两比特字符和两比特字符。
4  所以最后一个字符不是一比特字符。

```

题解:

已知结尾一定是0，则可以从后面入手，找到结尾前一个0的位置，假设其位置分别为n, i (n为结尾)，则这两个0中间一定有n - i - 2个1，判断是不是偶数就可以得到结果，因为n-i-2和n-i奇偶性相同，所以判断n-i是不是偶数即可


```

1  class Solution {
2  public:
3      bool isOneBitCharacter(vector<int> &bits) {
4          int n = bits.size(), i = n - 2;
5          while (i >= 0 and bits[i]) {
6              --i;
7          }
8          return (n - i) % 2 == 0;
9      }
10 };

```

1.10 724 寻找数组的中心下标

[724. 寻找数组的中心下标 - 力扣 \(LeetCode\)](#)

给你一个整数数组 `nums`，请计算数组的 **中心下标**。

数组 **中心下标** 是数组的一个下标，其左侧所有元素相加的和等于右侧所有元素相加的和。

如果中心下标位于数组最左端，那么左侧数之和视为 `0`，因为在下标的左侧不存在元素。这一点对于中心下标位于数组最右端同样适用。

如果数组有多个中心下标，应该返回 **最靠近左边** 的那一个。如果数组不存在中心下标，返回 `-1`。

示例 1:

```

1  输入: nums = [1, 7, 3, 6, 5, 6]
2  输出: 3
3  解释:
4  中心下标是 3 。
5  左侧数之和 sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11 ,
6  右侧数之和 sum = nums[4] + nums[5] = 5 + 6 = 11 , 二者相等。

```

示例 2:

```

1  输入: nums = [1, 2, 3]
2  输出: -1
3  解释:
4  数组中不存在满足此条件的中心下标。

```

示例 3:

```

1  输入: nums = [2, 1, -1]
2  输出: 0
3  解释:
4  中心下标是 0 。
5  左侧数之和 sum = 0 , (下标 0 左侧不存在元素) ,
6  右侧数之和 sum = nums[1] + nums[2] = 1 + -1 = 0 。

```

题解:

因为左右两边相等，假设左边和为 n ，则 $2n + \text{nums}[i] = \text{nums总和}$ ，求出数组所有元素和放在 sum 中，遍历数组，直到 $2 \times \text{temp} = 2 \times (\text{nums}[0] + \dots + \text{nums}[i-1]) = 2n - \text{nums}[i]$

```

1  class Solution {
2  public:
3      int pivotIndex(vector<int>& nums) {
4          int sum = accumulate(nums.begin(), nums.end(), 0);

```

```

5         int temp = 0;
6         for (int i = 0; i < nums.size(); i++){
7             if(temp * 2 == sum - nums[i]){
8                 return i;
9             }
10            temp += nums[i];
11        }
12        return -1;
13    }
14 };

```

1.11 747 至少是其他数字两倍的最大数

[747. 至少是其他数字两倍的最大数 - 力扣 \(LeetCode\)](#)

给你一个整数数组 `nums`，其中总是存在 **唯一的** 一个最大整数。

请你找出数组中的最大元素并检查它是否 **至少是数组中每个其他数字的两倍**。如果是，则返回 **最大元素的下标**，否则返回 `-1`。

示例 1:

```

1  输入: nums = [3,6,1,0]
2  输出: 1
3  解释: 6 是最大的整数，对于数组中的其他整数，6 至少是数组中其他元素的两倍。6 的下标是 1，所以返回 1。

```

示例 2:

```

1  输入: nums = [1,2,3,4]
2  输出: -1
3  解释: 4 没有超过 3 的两倍大，所以返回 -1。

```

题解:

没啥好说的，遍历找到最大和次大，判断是否两倍关系

```

1  class Solution {
2  public:
3      int dominantIndex(vector<int>& nums) {
4          int maxNum = INT_MIN, lastMax = INT_MIN;
5          int sub = 0;
6          for (int i = 0; i < nums.size(); i++){
7              if (maxNum < nums[i]){
8                  lastMax = maxNum;
9                  maxNum = nums[i];
10                 sub = i;
11             }
12             else if (lastMax < nums[i]){
13                 lastMax = nums[i];
14             }
15         }
16         return maxNum >= 2 * lastMax ? sub : -1;
17     }
18 };

```

1.12 2 两数相加

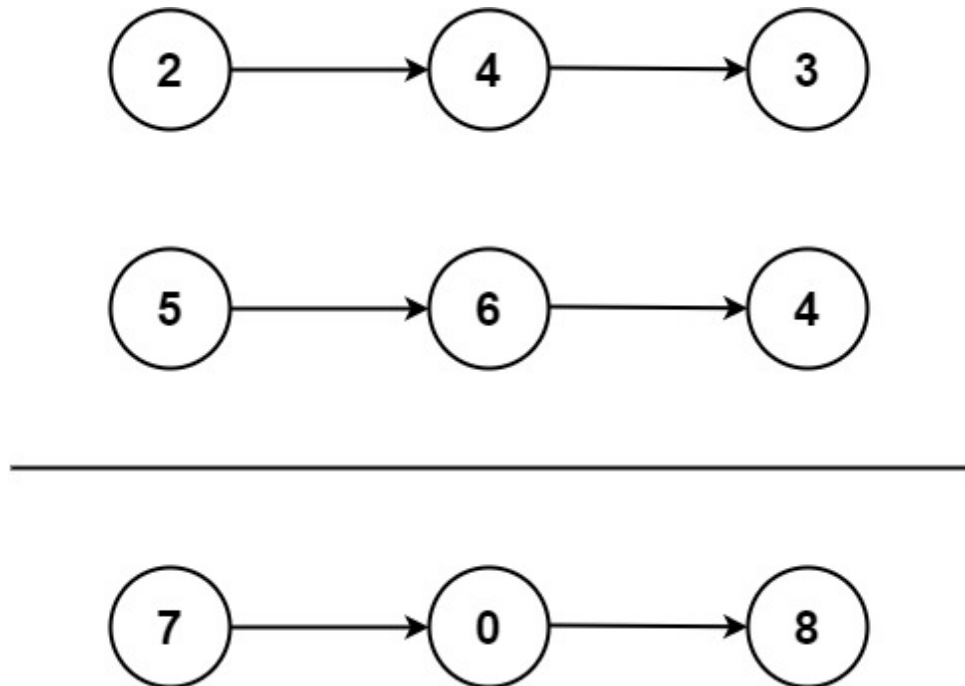
2. 两数相加

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1:



```
1 | 输入: l1 = [2,4,3], l2 = [5,6,4]
2 | 输出: [7,0,8]
3 | 解释: 342 + 465 = 807.
```

示例 2:

```
1 | 输入: l1 = [0], l2 = [0]
2 | 输出: [0]
```

示例 3:

```
1 | 输入: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
2 | 输出: [8,9,9,9,0,0,0,1]
```

提示:

- 每个链表中的节点数在范围 `[1, 100]` 内
- `0 <= Node.val <= 9`
- 题目数据保证列表表示的数字不含前导零

题解:

第一版直接暴力求解，把链表的数字表示出来，溢出了 🤖🤖🤖

```

1  class Solution {
2  public:
3      ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
4          if (l1 == nullptr || l2 == nullptr)
5              return nullptr;
6          int num1 = 0, num2 = 0;
7          int d = 0;
8          //求l1的数值
9          if (l1->val != 0){
10             while (l1 != nullptr) {
11                 num1 += l1->val * pow(10, d++);
12                 l1 = l1->next;
13             }
14         }
15
16         //求l2的数值
17         d = 0;
18         if (l2->val != 0){
19             while (l2 != nullptr) {
20                 num2 += l2->val * pow(10, d++);
21                 l2 = l2->next;
22             }
23         }
24
25         //求结果
26         ListNode *pre = new ListNode(0);
27         int sum = num1 + num2;
28         ListNode *cur = pre;
29         while (sum > 0) {
30             ListNode *next = new ListNode(sum % 10);
31             cur->next = next;
32             cur = cur->next;
33             sum /= 10;
34         }
35
36         return pre->next == nullptr ? pre : pre->next;
37     }
38 };

```

执行出错

Line 27: Char 22: runtime error: 1e+10 is outside the range of representable values of type 'int' (solution.cpp)

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior prog_joined.cpp:36:22

最后执行的输入

```

1  l1 =[9]
2
3  l2 =[1,9,9,9,9,9,9,9,9,9]

```

第二版看了题解，妙啊，对于按位运算可以这样做：将进位（carry）和求和（sum）分开考虑，sum每次加上两个数和进位，然后判断是否大于10，大于则sum %10，最后一位加完若进位存在，则最高位在添个1

```

1  class Solution {
2  public:
3      ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
4          if (l1 == nullptr || l2 == nullptr)

```

```

5         return nullptr;
6     ListNode *pre = new ListNode(0);
7     int carry = 0;    //进位
8     ListNode *cur = pre;
9     while (l1 != nullptr || l2 != nullptr) {
10        //每一位求和
11        int x = l1 == nullptr ? 0 : l1->val;
12        int y = l2 == nullptr ? 0 : l2->val;
13        int sum = x + y + carry;
14
15        carry = sum / 10;    //求进位
16        sum %= 10;    //sum一定小于10
17        cur->next = new ListNode(sum);
18        cur = cur->next;
19        if (l1 != nullptr)
20            l1 = l1->next;
21        if (l2 != nullptr)
22            l2 = l2->next;
23    }
24    if (carry > 0)
25        cur->next = new ListNode(carry);
26
27    return pre->next;
28 }
29 };

```

1.13 每日一题 2024/4/1

[故障键盘](#)

你的笔记本键盘存在故障，每当你在上面输入字符 `'i'` 时，它会反转你所写的字符串。而输入其他字符则可以正常工作。

给你一个下标从 0 开始的字符串 `s`，请你用故障键盘依次输入每个字符。

返回最终笔记本屏幕上输出的字符串。

示例 1:

```

1  输入: s = "string"
2  输出: "rtsng"
3  解释:
4  输入第 1 个字符后，屏幕上的文本是："s" 。
5  输入第 2 个字符后，屏幕上的文本是："st" 。
6  输入第 3 个字符后，屏幕上的文本是："str" 。
7  因为第 4 个字符是 'i' ，屏幕上的文本被反转，变成 "rts" 。
8  输入第 5 个字符后，屏幕上的文本是："rtsn" 。
9  输入第 6 个字符后，屏幕上的文本是："rtsng" 。
10 因此，返回 "rtsng" 。

```

示例 2:

```

1  输入: s = "poiinter"
2  输出: "ponter"
3  解释:
4  输入第 1 个字符后, 屏幕上的文本是: "p" 。
5  输入第 2 个字符后, 屏幕上的文本是: "po" 。
6  因为第 3 个字符是 'i' , 屏幕上的文本被反转, 变成 "op" 。
7  因为第 4 个字符是 'i' , 屏幕上的文本被反转, 变成 "po" 。
8  输入第 5 个字符后, 屏幕上的文本是: "pon" 。
9  输入第 6 个字符后, 屏幕上的文本是: "pont" 。
10 输入第 7 个字符后, 屏幕上的文本是: "ponte" 。
11 输入第 8 个字符后, 屏幕上的文本是: "ponter" 。
12 因此, 返回 "ponter" 。

```

提示:

- `1 <= s.length <= 100`
- `s` 由小写英文字母组成
- `s[0] != 'i'`

题解:

用双端队列反向模拟

```

1  class Solution {
2  public:
3      string finalString(string s) {
4          deque<char> q;
5          bool back = false;
6          for (auto ch : s) {
7              if (ch != 'i') {
8                  if (back)
9                      q.push_back(ch);
10                 else
11                     q.push_front(ch);
12             }
13             else
14                 back = !back;
15         }
16         string res = (back ? string{q.begin(), q.end()} : string{q.rbegin(), q.rend()});
17         return res;
18     }
19 };

```

1.14 5 最长回文子串

5. 最长回文子串

给你一个字符串 `s`, 找到 `s` 中最长的回文子串。

如果字符串的反序与原始字符串相同, 则该字符串称为回文字符串。

示例 1:

```

1  输入: s = "babad"
2  输出: "bab"
3  解释: "aba" 同样是符合题意的答案。

```

示例 2:

```
1 输入: s = "cbbd"
2 输出: "bb"
```

提示:

- `1 <= s.length <= 1000`
- `s` 仅由数字和英文字母组成

题解:

动态规划，得多练练

```
1 class Solution {
2 public:
3     string longestPalindrome(string s) {
4         if (s.size() == 0)
5             return "";
6         int start = 0, end = 0;
7         for (int i = 0; i < s.size(); i++) {
8             auto [left1, right1] = expandAroundCenter(s, i, i);
9             auto [left2, right2] = expandAroundCenter(s, i, i + 1);
10            if (right1 - left1 > end - start){
11                start = left1;
12                end = right1;
13            }
14            if (right2 - left2 > end - start) {
15                start = left2;
16                end = right2;
17            }
18        }
19        return s.substr(start, end - start + 1);
20    }
21
22    pair<int, int> expandAroundCenter(string &s, int i, int j){
23        int left = i, right = j;
24        while (left >= 0 && right <= s.size() && s[left] == s[right]) {
25            left--;
26            right++;
27        }
28        return {left + 1, right - 1};
29    }
30 }
31 };
```

1.15 每日一题 2024/4/2

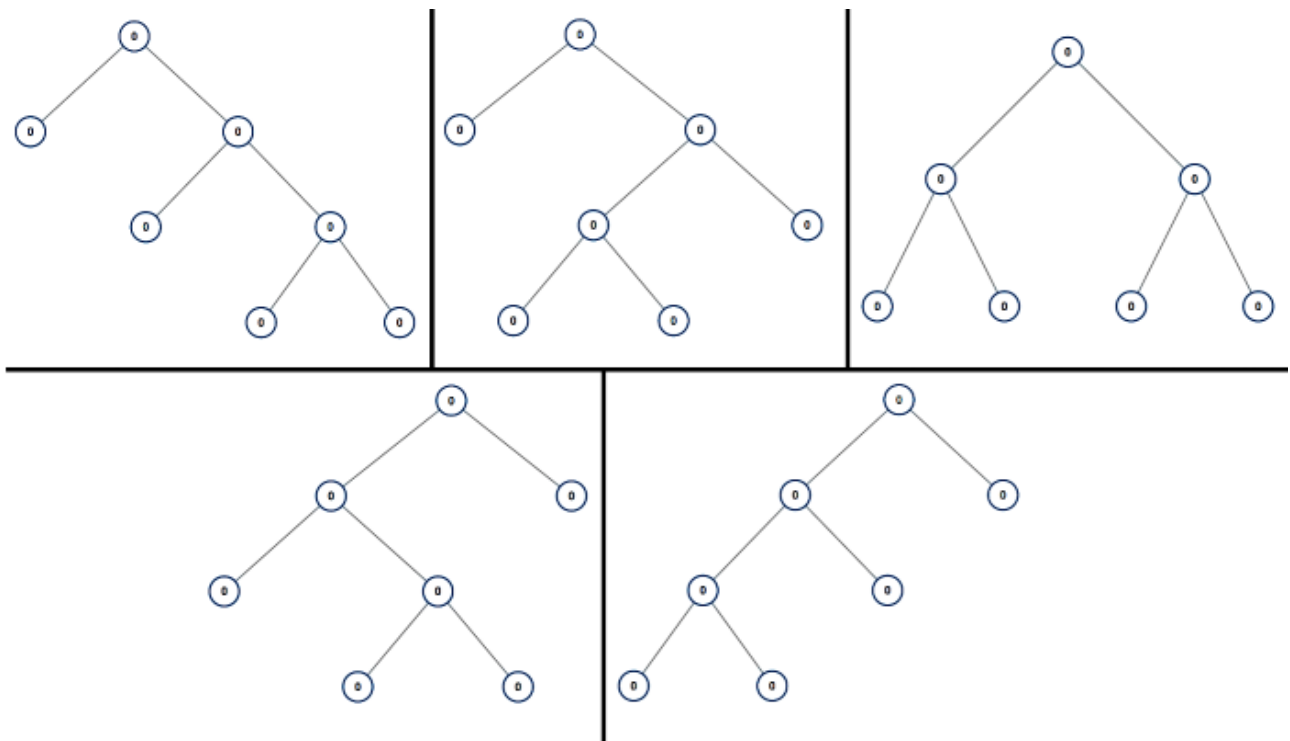
[所有可能的真二叉树](#)

给你一个整数 `n`，请你找出所有可能含 `n` 个节点的真二叉树，并以列表形式返回。答案中每棵树的每个节点都必须符合 `Node.val == 0`。

答案的每个元素都是一棵真二叉树的根节点。你可以按任意顺序返回最终的真二叉树列表。

真二叉树 是一类二叉树，树中每个节点恰好有 0 或 2 个子节点。

示例 1:



1 输入: $n = 7$
2 输出: $[[0,0,0,null,null,0,0,null,null,0,0],[0,0,0,null,null,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,null,null,null,null,0,0],[0,0,0,0,0,0,null,null,0,0]]$

示例 2:

1 输入: $n = 3$
2 输出: $[[0,0,0]]$

提示:

- $1 \leq n \leq 20$

题解:

分治思想, 左右子树节点为 $i, n - 1 - i$

```
1 class Solution {
2 public:
3     vector<TreeNode*> allPossibleFBT(int n) {
4         vector<TreeNode*> res;
5         if (n % 2 == 0)
6             return res;
7         if (n == 1){
8             res = {new TreeNode(0)};
9             return res;
10        }
11        for (int i = 1; i < n; i++) {
12            vector<TreeNode*> leftTree = allPossibleFBT(i);
13            vector<TreeNode*> rightTree = allPossibleFBT(n - 1 - i);
14            for (auto a : leftTree){
15                for (auto b : rightTree) {
16                    TreeNode *root = new TreeNode(0, a, b);
17                    res.push_back(root);
18                }
19            }
20        }
21        return res;
22    }
23 }
```



```

18         }
19     }
20 }
21 return res;
22 }
23 };

```

2 HOT100

2.1 49 字母异位词分组

[49. 字母异位词分组](#)

给你一个字符串数组，请你将 **字母异位词** 组合在一起。可以按任意顺序返回结果列表。

字母异位词 是由重新排列源单词的所有字母得到的一个新单词。

示例 1:

```

1 输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
2 输出: [["bat"],["nat","tan"],["ate","eat","tea"]]

```

示例 2:

```

1 输入: strs = [""]
2 输出: [[""]]

```

示例 3:

```

1 输入: strs = ["a"]
2 输出: [["a"]]

```

提示:

- `1 <= strs.length <= 104`
- `0 <= strs[i].length <= 100`
- `strs[i]` 仅包含小写字母

题解:

一开始没搞懂题目意思，还以为不同单词可能有重复的位置的字母，结果没有，

用哈希表，key为排序后的str中的每个string（唯一的）

```

1  class Solution {
2  public:
3      vector<vector<string>> groupAnagrams(vector<string>& strs) {
4          unordered_map<string, vector<string>> mp;
5          for (auto &s : strs) {
6              string key = s;
7              sort(key.begin(), key.end());    //对key排序
8              mp[key].emplace_back(s);
9          }
10         vector<vector<string>> res;
11         for (auto it = mp.begin(); it != mp.end(); ++it) {
12             res.emplace_back(it->second);
13         }

```

```
14     return res;
15 }
16 };
```

2.2 128 最长连续序列

[128. 最长连续序列](#)

给定一个未排序的整数数组 `nums`，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 $O(n)$ 的算法解决此问题。

示例 1:

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

示例 2:

```
1 输入: nums = [0,3,7,2,5,8,4,6,0,1]
2 输出: 9
```

提示:

- `0 <= nums.length <= 105`
- `-109 <= nums[i] <= 109`

题解:

第一版，利用哈希表存储，key为数组中的元素（其实可以用set），根据key查找对应的key+1在不在哈希表中

```
1  class Solution {
2  public:
3      int longestConsecutive(vector<int>& nums) {
4          if (nums.size() == 0)
5              return 0;
6          map<int, int> mp;
7          for (auto a : nums)
8              mp[a]++;
9          int maxLen = 1;
10         int maxNum = 1;
11         for (auto it = mp.begin(); it != mp.end(); it++) {
12             if (mp.count(1+it->first))
13                 maxNum++;
14             else{
15                 maxLen = max(maxNum, maxLen);
16                 maxNum = 1;
17             }
18         }
19         return maxLen;
20     }
21 };
```

2.3 283 移动零

283. 移动零

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意，必须在不复制数组的情况下原地对数组进行操作。

示例 1:

```
1 输入: nums = [0,1,0,3,12]
2 输出: [1,3,12,0,0]
```

示例 2:

```
1 输入: nums = [0]
2 输出: [0]
```

提示:

- `1 <= nums.length <= 104`
- `-231 <= nums[i] <= 231 - 1`

进阶：你能尽量减少完成的操作次数吗？

题解：

笑死了，一开始想用erase删除0，然后尾插0，但是逻辑出了点问题，会导致遍历到后面插入的几个0的时候死循环😂😂

还是用双指针好，右指针遍历，左指针指向非零数序列尾，右指针只要找到非零数就交换左右指针的数

```
1 class Solution {
2 public:
3     void moveZeroes(vector<int>& nums) {
4         int n = nums.size(), right = 0, left = 0;
5         while (right < n) {
6             if (nums[right]) {
7                 swap(nums[left], nums[right]);
8                 left++;
9             }
10            right++;
11        }
12    }
13};
```

2.4 11 盛最多水的容器

11. 盛最多水的容器

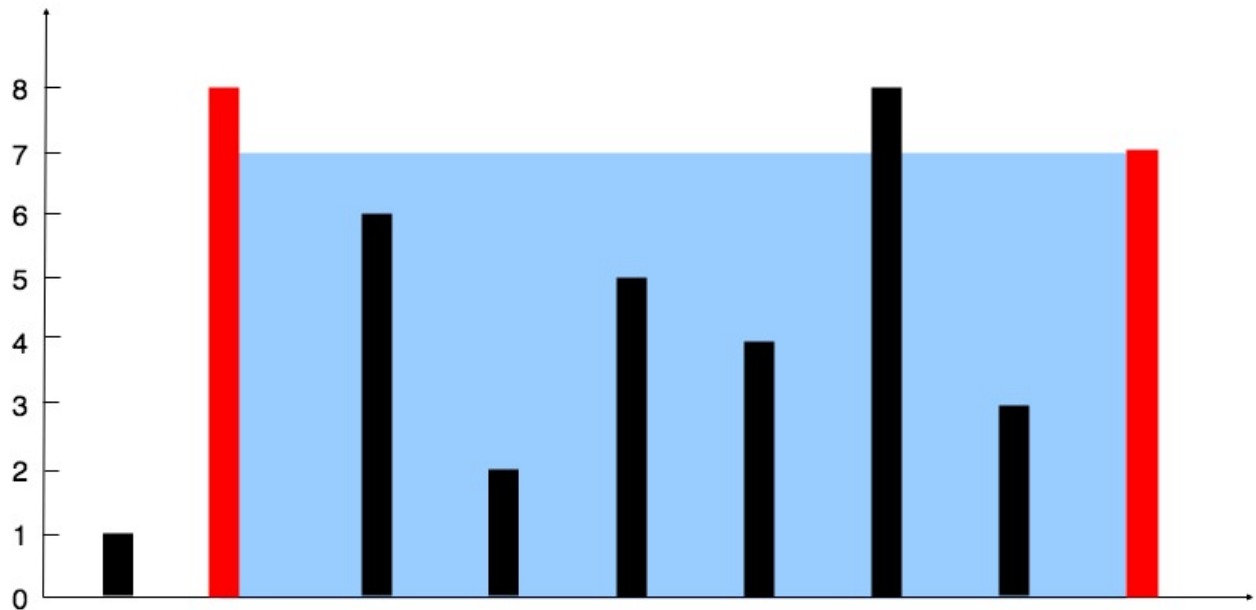
给定一个长度为 `n` 的整数数组 `height`。有 `n` 条垂线，第 `i` 条线的两个端点是 `(i, 0)` 和 `(i, height[i])`。

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。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1:



1 输入: [1,8,6,2,5,4,8,3,7]

2 输出: 49

3 解释: 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 2:

1 输入: height = [1,1]

2 输出: 1

提示:

- `n == height.length`
- `2 <= n <= 105`
- `0 <= height[i] <= 104`

题解:

双指针, 从左右两端开始往中间靠, 每次扔掉“短板”并维护最大容量

```
1 class Solution {
2 public:
3     int maxArea(vector<int>& height) {
4         int left = 0, right = height.size() - 1;
5         int maxRes = 0;
6         while (left < right) {
7             int h = height[left] <= height[right] ? height[left] : height[right]; //不能在这里left++或right--, 因为后面res计算用到了
8             int res = h * (right - left);
9             maxRes = max(maxRes, res);
10            height[left] <= height[right] ? left++ : right--;
11        }
12        return maxRes;
13    }
14};
```

2.5 15 三数之和

15. 三数之和

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k`，同时还满足 `nums[i] + nums[j] + nums[k] == 0`。请

你返回所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

```
1 输入：nums = [-1,0,1,2,-1,-4]
2 输出：[[-1,-1,2],[-1,0,1]]
3 解释：
4 nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。
5 nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。
6 nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。
7 不同的三元组是 [-1,0,1] 和 [-1,-1,2] 。
8 注意，输出的顺序和三元组的顺序并不重要。
```

示例 2：

```
1 输入：nums = [0,1,1]
2 输出：[]
3 解释：唯一可能的三元组和不为 0 。
```

示例 3：

```
1 输入：nums = [0,0,0]
2 输出：[[0,0,0]]
3 解释：唯一可能的三元组和为 0 。
```

提示：

- `3 <= nums.length <= 3000`
- `-105 <= nums[i] <= 105`

题解：

已经是二刷了，不过还是得加深印象啊，也是双指针类似问题。

只需要两层遍历，第一层找基数，第二层找和为0 - 基数的两个数

```
1  class Solution {
2  public:
3      vector<vector<int>> threeSum(vector<int>& nums) {
4          vector<vector<int>> res;
5          if (nums.size() < 3)
6              return res;
7          sort(nums.begin(), nums.end());    //排序很重要，根据排序判断大小呢得
8          int n = nums.size();
9          for (int first = 0; first < n; first++) {
10             if (first > 0 && nums[first] == nums[first - 1])
11                 continue;
12             int third = n - 1;
```

```

13         int target = -nums[first];
14         for (int second = first + 1; second < n; second++){
15             if (second > first + 1 && nums[second] == nums[second - 1])
16                 continue;
17             while (second < third && nums[second] + nums[third] > target)
18                 third--;
19             if (second == third)
20                 break;
21             if (nums[second] + nums[third] == target)
22                 res.push_back({nums[first], nums[second], nums[third]});
23         }
24     }
25     return res;
26 }
27 };

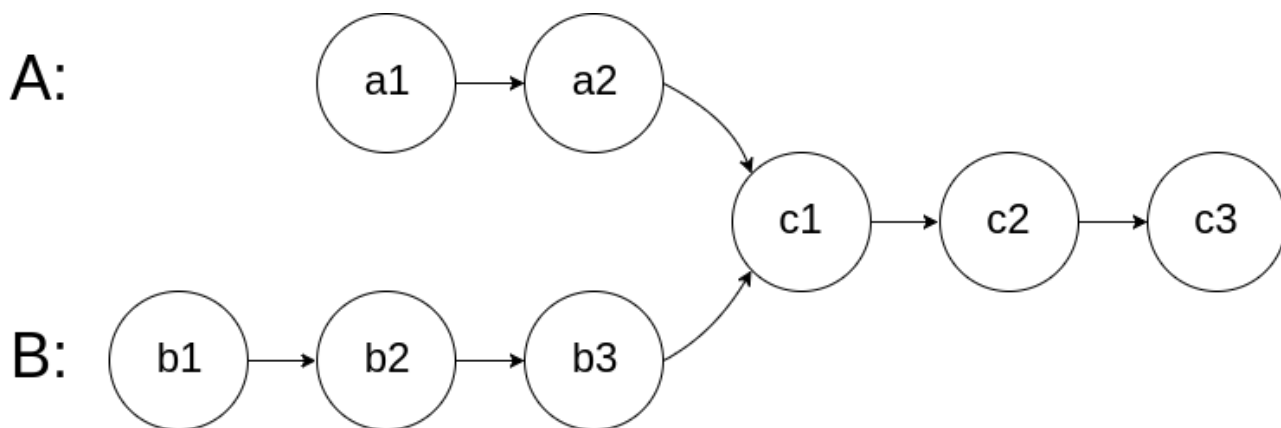
```

2.6 160 相交链表

160. 相交链表

给你两个单链表的头节点 `headA` 和 `headB`，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 `null`。

图示两个链表在节点 `c1` 开始相交：



题目数据 **保证** 整个链式结构中不存在环。

注意，函数返回结果后，链表必须 **保持其原始结构**。

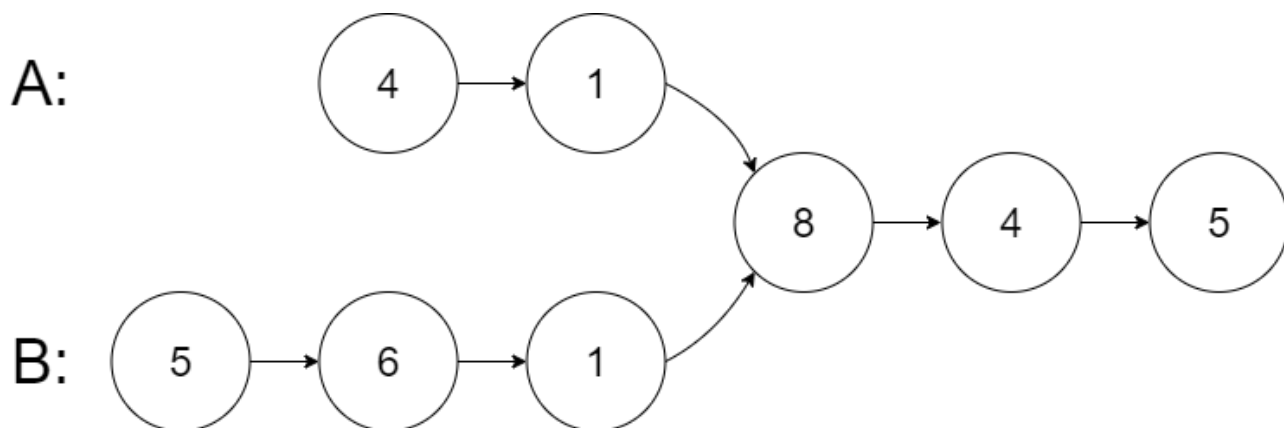
自定义评测：

评测系统的输入如下（你设计的程序 **不适用** 此输入）：

- `intersectVal` - 相交的起始节点的值。如果不存在相交节点，这一值为 `0`
- `listA` - 第一个链表
- `listB` - 第二个链表
- `skipA` - 在 `listA` 中（从头节点开始）跳到交叉节点的节点数
- `skipB` - 在 `listB` 中（从头节点开始）跳到交叉节点的节点数

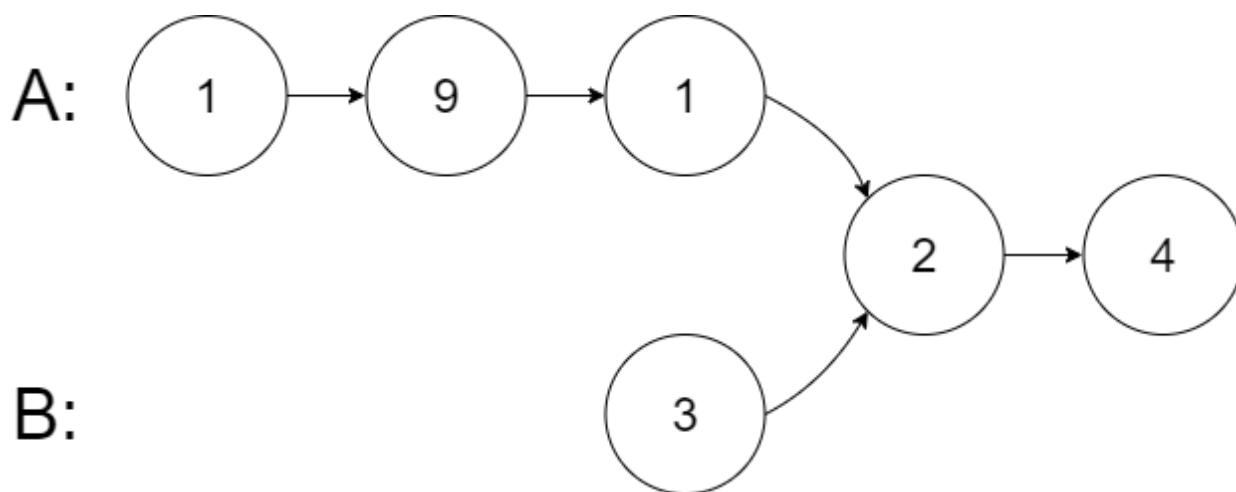
评测系统将根据这些输入创建链式数据结构，并将两个头节点 `headA` 和 `headB` 传递给你的程序。如果程序能够正确返回相交节点，那么你的解决方案将被 **视作正确答案**。

示例 1:



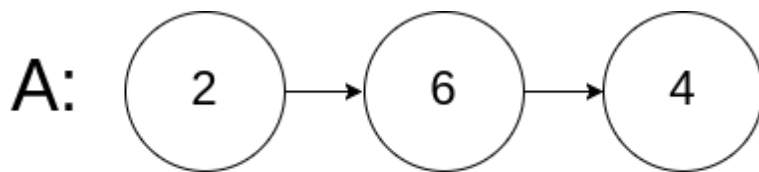
```
1 输入: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3
2 输出: Intersected at '8'
3 解释: 相交节点的值为 8 (注意, 如果两个链表相交则不能为 0)。
4 从各自的表头开始算起, 链表 A 为 [4,1,8,4,5], 链表 B 为 [5,6,1,8,4,5]。
5 在 A 中, 相交节点前有 2 个节点; 在 B 中, 相交节点前有 3 个节点。
6 - 请注意相交节点的值不为 1, 因为在链表 A 和链表 B 之中值为 1 的节点 (A 中第二个节点和 B 中第三个节点) 是不同的节点。换句话说, 它们在内存中指向两个不同的位置, 而链表 A 和链表 B 中值为 8 的节点 (A 中第三个节点, B 中第四个节点) 在内存中指向相同的位置。
```

示例 2:



```
1 输入: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
2 输出: Intersected at '2'
3 解释: 相交节点的值为 2 (注意, 如果两个链表相交则不能为 0)。
4 从各自的表头开始算起, 链表 A 为 [1,9,1,2,4], 链表 B 为 [3,2,4]。
5 在 A 中, 相交节点前有 3 个节点; 在 B 中, 相交节点前有 1 个节点。
```

示例 3:



```

1  输入: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
2  输出: null
3  解释: 从各自的表头开始算起, 链表 A 为 [2,6,4], 链表 B 为 [1,5]。
4  由于这两个链表不相交, 所以 intersectVal 必须为 0, 而 skipA 和 skipB 可以是任意值。
5  这两个链表不相交, 因此返回 null 。
  
```

提示:

- `listA` 中节点数目为 `m`
- `listB` 中节点数目为 `n`
- `1 <= m, n <= 3 * 104`
- `1 <= Node.val <= 105`
- `0 <= skipA <= m`
- `0 <= skipB <= n`
- 如果 `listA` 和 `listB` 没有交点, `intersectVal` 为 `0`
- 如果 `listA` 和 `listB` 有交点, `intersectVal == listA[skipA] == listB[skipB]`

进阶: 你能否设计一个时间复杂度 `O(m + n)`、仅用 `O(1)` 内存的解决方案?

题解:

双指针, 记住, 不管有无交点, 相交前的总和是一样的, 无交点的情况可以看做交点为 `nullptr`

```

1  ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
2      if (!headA || !headB)
3          return nullptr;
4      ListNode *p1 = headA;
5      ListNode *p2 = headB;
6      while (p1 != p2) {
7          p1 = (p1 == nullptr ? headB : p1->next);
8          p2 = (p2 == nullptr ? headA : p2->next);
9      }
10     return p1;
11 }
  
```


2.7 42 接雨水

42. 接雨水

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1:



```
1 输入: height = [0,1,0,2,1,0,1,3,2,1,2,1]
2 输出: 6
3 解释: 上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。
```

示例 2:

```
1 输入: height = [4,2,0,3,2,5]
2 输出: 9
```

提示:

- `n == height.length`
- `1 <= n <= 2 * 10^4`
- `0 <= height[i] <= 10^5`

题解:

大名鼎鼎的接雨水，终于刷了哈，动态规划，第一版：（空间复杂度为 $O(n)$ ）

```
1  class Solution {
2  public:
3      int trap(vector<int>& height) {
4          if (height.size() == 0)
5              return 0;
6          int n = height.size() - 1;
7          vector<int> leftMax(height.size());
8          vector<int> rightMax(height.size());
9          leftMax[0] = height[0];
10         rightMax[n] = height[n];
11         for (int i = 1; i <= n; i++)
12             leftMax[i] = max(leftMax[i - 1], height[i]);
13         for (int i = n - 1; i >= 0; i--)
14             rightMax[i] = max(rightMax[i + 1], height[i]);
15         int ret = 0;
16         for (int i = 0; i <= n; i++)
17             ret += min(leftMax[i], rightMax[i]) - height[i];
18     }
```

```

19         return ret;
20     }
21 };

```

第二版，双指针，重要的就是理解，leftMax和rightMax。如果当前的数小于右边的数，则我这个点撑死了，最多接的水也就leftMax-本身，就算我这个点相邻好几个都是0，那也是左边那么多啊，因为右边大了呀

```

1  class Solution {
2  public:
3      int trap(vector<int>& height) {
4          if (height.size() == 0)
5              return 0;
6          int left = 0, right = height.size() - 1;
7          int leftMax = 0, rightMax = 0;
8          int res = 0;
9          while (left < right) {
10             leftMax = max(leftMax, height[left]);
11             rightMax = max(rightMax, height[right]);
12             if (height[left] < height[right])
13                 res += leftMax - height[left++];
14             else
15                 res += rightMax - height[right--];
16         }
17         return res;
18     }
19 };

```

2.8 3 无重复字符的最长子串

[3. 无重复字符的最长子串](#)

给定一个字符串 `s`，请你找出其中不含有重复字符的**最长子串**的长度。

示例 1:

```

1  输入: s = "abcabcbb"
2  输出: 3
3  解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

```

示例 2:

```

1  输入: s = "bbbbbb"
2  输出: 1
3  解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

```

示例 3:

```

1  输入: s = "pwwkew"
2  输出: 3
3  解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
4      请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

```

提示:

- `0 <= s.length <= 5 * 104`
- `s` 由英文字母、数字、符号和空格组成

题解:

滑动窗口, 值得看看, 感觉滑动窗口这方面挺欠缺

```
1   int lengthOfLongestSubstring(string s) {
2       if (s.size() == 0)
3           return 0;
4       unordered_set<char> st;
5       int r = -1;
6       int maxLen = 0;
7       for (int i = 0; i < s.size(); i++) {
8           if (i != 0)
9               st.erase(s[i - 1]);
10          while (r + 1 < s.size() && !st.count(s[r + 1])) {
11              st.insert(s[r + 1]);
12              r++;    //右边可以滑, 只要没有重复的
13          }
14          maxLen = max(maxLen, r - i + 1);
15      }
16      return maxLen;
17  }
```

2.9 438 找到字符串中所有字母异位词

[438. 找到字符串中所有字母异位词](#)

给定两个字符串 `s` 和 `p`, 找到 `s` 中所有 `p` 的 **异位词** 的子串, 返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串 (包括相同的字符串)。

示例 1:

```
1  输入: s = "cbaebabacd", p = "abc"
2  输出: [0,6]
3  解释:
4  起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
5  起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

示例 2:

```
1  输入: s = "abab", p = "ab"
2  输出: [0,1,2]
3  解释:
4  起始索引等于 0 的子串是 "ab", 它是 "ab" 的异位词。
5  起始索引等于 1 的子串是 "ba", 它是 "ab" 的异位词。
6  起始索引等于 2 的子串是 "ab", 它是 "ab" 的异位词。
```

提示:

- `1 <= s.length, p.length <= 3 * 104`
- `s` 和 `p` 仅包含小写字母

题解:

滑动窗口, 妙啊, 每次记录不相同的个数, 然后根据这一个变量取舍就好了

```

1  class Solution {
2  public:
3      vector<int> findAnagrams(string s, string p) {
4          int slen = s.size(), plen = p.size();
5          if (slen < plen)
6              return vector<int> ();
7          vector<int> ans;
8          vector<int> count(26);
9          int differ = 0;
10         for (int i = 0; i < plen; i++) {
11             ++count[s[i] - 'a'];
12             --count[p[i] - 'a'];
13         }
14         for (int i = 0; i < 26; i++) {
15             if (count[i] != 0)
16                 ++differ;
17         }
18         if (differ == 0)
19             ans.emplace_back(0);
20         for (int i = 0; i < slen - plen; i++) {
21             if (count[s[i] - 'a'] == 0)    //当前字母s[i]的个数已经相同，结果往前滑又给他扔
了，得，不同了
22                 ++differ;
23             else if (count[s[i] - 'a'] == 1)    //当前字母s[i]个数不同，且差值为1，相当于多了
1个，OK往前滑扔掉了，相同了
24                 --differ;
25             --count[s[i] - 'a'];
26             //同理
27             if (count[s[i + plen] - 'a'] == 0)
28                 ++differ;
29             else if (count[s[i + plen] - 'a'] == -1)
30                 --differ;
31             ++count[s[i + plen] - 'a'];
32
33             if (differ == 0)
34                 ans.emplace_back(i + 1);
35         }
36         return ans;
37     }
38 };

```

2.10 560 和为K的子数组

[560. 和为 K 的子数组](#)

给你一个整数数组 `nums` 和一个整数 `k`，请你统计并返回 该数组中和为 `k` 的子数组的个数。

子数组是数组中元素的连续非空序列。

示例 1:

```

1  输入: nums = [1,1,1], k = 2
2  输出: 2

```

示例 2:

1 输入: nums = [1,2,3], k = 3
2 输出: 2

提示:

- `1 <= nums.length <= 2 * 104`
- `-1000 <= nums[i] <= 1000`
- `-107 <= k <= 107`

题解:

一开始想用滑动窗口, 不知道为什么不太好实现, 看了看题解, 用到了哈希表记录前缀和

```
1 class Solution {
2 public:
3     int subarraySum(vector<int>& nums, int k) {
4         unordered_map<int, int> mp;
5         mp[0] = 1;
6         int pre = 0, count = 0;
7         for (int a : nums) {
8             pre += a;
9             if (mp.find(pre - k) != mp.end())
10                 count += mp[pre - k];
11             mp[pre]++;
12         }
13         return count;
14     }
15 };
```

2.11 239 滑动窗口最大值

[239. 滑动窗口最大值](#)

给你一个整数数组 `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 1:

```
1 输入: nums = [1,3,-1,-3,5,3,6,7], k = 3
2 输出: [3,3,5,5,6,7]
3 解释:
4 滑动窗口的位置          最大值
5 -----
6 [1  3  -1] -3  5  3  6  7      3
7  1 [3  -1  -3] 5  3  6  7      3
8  1  3 [-1  -3  5] 3  6  7      5
9  1  3  -1 [-3  5  3] 6  7      5
10 1  3  -1  -3 [5  3  6] 7      6
11 1  3  -1  -3  5 [3  6  7]     7
```

示例 2:

```
1 输入: nums = [1], k = 1
2 输出: [1]
```

提示:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`
- `1 <= k <= nums.length`

题解:

第一版, 优先队列, 不是最优, 但是起码能解了

```
1      vector<int> maxSlidingWindow(vector<int>& nums, int k) {
2          int n = nums.size();
3          if (n < k)
4              return vector<int> ();
5          priority_queue<pair<int, int>> que;
6          for (int i = 0; i < k; i++)
7              que.emplace(nums[i], i);
8          vector<int> ans = {que.top().first};
9          for (int i = k; i < n; i++) {
10             que.emplace(nums[i], i);
11             while (que.top().second <= i - k) {
12                 que.pop();
13             }
14             ans.push_back(que.top().first);
15         }
16         return ans;
17     }
```

第二版, 单调队列, 时间复杂度和空间复杂度更低了

```
1      vector<int> maxSlidingWindow(vector<int>& nums, int k) {
2          int n = nums.size();
3          if (n < k)
4              return vector<int> ();
5          deque<int> q;
6          for (int i = 0; i < k; i++) {
7              while (!q.empty() && nums[i] >= nums[q.back()])
8                  q.pop_back();
9              q.push_back(i);
10         }
11         vector<int> ans = {nums[q.front()]};
12         for (int i = k; i < n; i++) {
13             while (!q.empty() && nums[i] >= nums[q.back()])
14                 q.pop_back();
15             q.push_back(i);
16             while (q.front() <= i - k) {
17                 q.pop_front();
18             }
19             ans.push_back(nums[q.front()]);
20         }
21         return ans;
22     }
```

2.12 76 最小覆盖子串

76. 最小覆盖子串

给你一个字符串 `s`、一个字符串 `t`。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""`。

注意：

- 对于 `t` 中重复字符，我们寻找的子字符串中该字符数量必须不少于 `t` 中该字符数量。
- 如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

```
1 输入：s = "ADOBECODEBANC", t = "ABC"
2 输出："BANC"
3 解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。
```

示例 2：

```
1 输入：s = "a", t = "a"
2 输出："a"
3 解释：整个字符串 s 是最小覆盖子串。
```

示例 3：

```
1 输入：s = "a", t = "aa"
2 输出：" "
3 解释：t 中两个字符 'a' 均应包含在 s 的子串中，
4 因此没有符合条件的子字符串，返回空字符串。
```

提示：

- `m == s.length`
- `n == t.length`
- `1 <= m, n <= 105`
- `s` 和 `t` 由英文字母组成

题解：

大致思虑就是滑动窗口，看看当前窗口中包含了t中的所有字符了吗？具体用哈希表实现

```
1 unordered_map<char, int> ori, cnt;
2
3 bool check() {
4     for (const auto &p : ori) {
5         if (cnt[p.first] < p.second)
6             return false;
7     }
8     return true;
9 }
10
11 string minWindow(string s, string t) {
12     int sLen = s.size(), tLen = t.size();
13     if (sLen < tLen)
```

```

14         return "";
15     int l = 0, r = -1;
16     int ansL = -1, len = INT_MAX;
17     for (const auto &c : t)
18         ori[c]++;
19     while (r < sLen) {
20         if (ori.find(s[++r]) != ori.end())
21             ++cnt[s[r]];
22         while (check() && l <= r) {
23             if (r - l + 1 < len) {
24                 len = r - l + 1;
25                 ansL = l;
26             }
27             if (ori.find(s[l]) != ori.end())
28                 --cnt[s[l]];
29             ++l;
30         }
31     }
32     return ansL == -1 ? string() : s.substr(ansL, len);
33 }

```

2.13 53 最大子数组和

53. 最大子数组和

给你一个整数数组 `nums`，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

子数组

是数组中的一个连续部分。

示例 1:

```

1 输入: nums = [-2,1,-3,4,-1,2,1,-5,4]
2 输出: 6
3 解释: 连续子数组 [4,-1,2,1] 的和最大, 为 6。

```

示例 2:

```

1 输入: nums = [1]
2 输出: 1

```

示例 3:

```

1 输入: nums = [5,4,-1,7,8]
2 输出: 23

```

提示:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`

进阶: 如果你已经实现复杂度为 `O(n)` 的解法，尝试使用更为精妙的 **分治法** 求解。

题解:

动态规划，理解理解，有兴趣可以尝试他说的分治


```

1 | int maxSubArray(vector<int>& nums) {
2 |     int pre = 0, maxAns = nums[0];
3 |     for (const auto &a : nums) {
4 |         pre = max(pre + a, a);
5 |         maxAns = max(pre, maxAns);
6 |     }
7 |     return maxAns;
8 | }

```

2.14 56 合并区间

56. 合并区间

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`。请你合并所有重叠的区间，并返回 一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1:

```

1 | 输入: intervals = [[1,3],[2,6],[8,10],[15,18]]
2 | 输出: [[1,6],[8,10],[15,18]]
3 | 解释: 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6]。

```

示例 2:

```

1 | 输入: intervals = [[1,4],[4,5]]
2 | 输出: [[1,5]]
3 | 解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。

```

提示:

- `1 <= intervals.length <= 104`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 104`

题解:

对区间左侧元素排序，然后就好求了

```

1 | vector<vector<int>> merge(vector<vector<int>>& intervals) {
2 |     if (intervals.size() == 0)
3 |         return {};
4 |     sort(intervals.begin(), intervals.end());
5 |     vector<vector<int>> merged;
6 |     for (int i = 0; i < intervals.size(); i++) {
7 |         int L = intervals[i][0], R = intervals[i][1];
8 |         if (!merged.size() || merged.back()[1] < L)
9 |             merged.push_back({L, R});
10 |        else
11 |            merged.back()[1] = max(merged.back()[1], R);
12 |    }
13 |    return merged;
14 | }

```

2.15 189 轮转数组

[189. 轮转数组](#)

给定一个整数数组 `nums`，将数组中的元素向右轮转 `k` 个位置，其中 `k` 是非负数。

示例 1:

```
1 输入: nums = [1,2,3,4,5,6,7], k = 3
2 输出: [5,6,7,1,2,3,4]
3 解释:
4 向右轮转 1 步: [7,1,2,3,4,5,6]
5 向右轮转 2 步: [6,7,1,2,3,4,5]
6 向右轮转 3 步: [5,6,7,1,2,3,4]
```

示例 2:

```
1 输入: nums = [-1,-100,3,99], k = 2
2 输出: [3,99,-1,-100]
3 解释:
4 向右轮转 1 步: [99,-1,-100,3]
5 向右轮转 2 步: [3,99,-1,-100]
```

提示:

- `1 <= nums.length <= 105`
- `-231 <= nums[i] <= 231 - 1`
- `0 <= k <= 105`

进阶:

- 尽可能想出更多的解决方案，至少有 **三种** 不同的方法可以解决这个问题。
- 你可以使用空间复杂度为 `O(1)` 的 **原地** 算法解决这个问题吗？

题解:

不会，看了题解，看的我一脸懵逼，数学上的理解，反复多看看吧

```
1  void rotate(vector<int>& nums, int k) {
2      int n = nums.size();
3      k = k % n;
4      int count = gcd(k, n);
5      for (int start = 0; start < count; ++start) {
6          int current = start;
7          int prev = nums[start];
8          do {
9              int next = (current + k) % n;
10             swap(nums[next], prev);
11             current = next;
12         } while (start != current);
13     }
14 }
```

2.16 238 除自身以外数组的乘积

238. 除自身以外数组的乘积

给你一个整数数组 `nums`，返回 数组 `answer`，其中 `answer[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

题目数据 保证 数组 `nums` 之中任意元素的全部前缀元素和后缀的乘积都在 32 位 整数范围内。

请 不要使用除法，且在 `O(*n*)` 时间复杂度内完成此题。

示例 1:

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

示例 2:

```
1 输入: nums = [-1,1,0,-3,3]
2 输出: [0,0,9,0,0]
```

提示:

- `2 <= nums.length <= 105`
- `-30 <= nums[i] <= 30`
- 保证 数组 `nums` 之中任意元素的全部前缀元素和后缀的乘积都在 32 位 整数范围内

进阶：你可以在 `O(1)` 的额外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。）

题解:

解题思路:

本题的难点在于 不能使用除法，即需要 只用乘法 生成数组 `ans`。根据题目对 `ans[i]` 的定义，可列出下图所示的表格。

根据表格的主对角线（全为 1），可将表格分为 上三角 和 下三角 两部分。分别迭代计算下三角和上三角两部分的乘积，即可 不使用除法 就获得结果。

```
1  vector<int> productExceptSelf(vector<int>& nums) {
2      vector<int> ans(nums.size(), 1);
3      int n = nums.size();
4      if (n < 1)
5          return ans;
6      ans[0] = 1;
7      int temp = 1;
8      for (int i = 1; i < n; i++) {
9          ans[i] *= ans[i - 1] * nums[i - 1];
10     }
11     for (int i = n - 2; i >= 0; i--) {
12         temp *= nums[i + 1];
13         ans[i] *= temp;
14     }
15
16     return ans;
17 }
```

2.17 41 缺失的第一个正数

41. 缺失的第一个正数

给你一个未排序的整数数组 `nums`，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为 $O(n)$ 并且只使用常数级别额外空间的解决方案。

示例 1:

```
1 输入: nums = [1,2,0]
2 输出: 3
3 解释: 范围 [1,2] 中的数字都在数组中。
```

示例 2:

```
1 输入: nums = [3,4,-1,1]
2 输出: 2
3 解释: 1 在数组中, 但 2 没有。
```

示例 3:

```
1 输入: nums = [7,8,9,11,12]
2 输出: 1
3 解释: 最小的正数 1 没有出现。
```

提示:

- `1 <= nums.length <= 105`
- `-231 <= nums[i] <= 231 - 1`

题解:

妙啊，利用原数组构造一个“哈希表”，重点就是理解一个长度为 n 的数组，最小未出现的整数只会在 $[1, n+1]$ 的范围。为什么呢？因为只要数组元素任意一个大于 n ，则肯定有个漏了，且一定是在 $[1, n+1]$ 的范围上，然后利用下标来当这个判断条件，构造出只要出现过的就给对应值下标表示的元素设置成负数

```
1  int firstMissingPositive(vector<int>& nums) {
2      if (nums.size() == 0)
3          return 0;
4      int n = nums.size();
5      /*先把负数全部改成n+1*/
6      for (int &num : nums) {
7          if (num <= 0)
8              num = n + 1;
9      }
10     /*标记过程*/
11     for (int i = 0; i < n; i++) {
12         int num = abs(nums[i]);
13         if (num <= n)
14             nums[num - 1] = -abs(nums[num - 1]);
15     }
16     /*根据标记判断*/
17     for (int i = 0; i < n; i++) {
18         if (nums[i] > 0)
19             return i + 1;
20     }
```

```

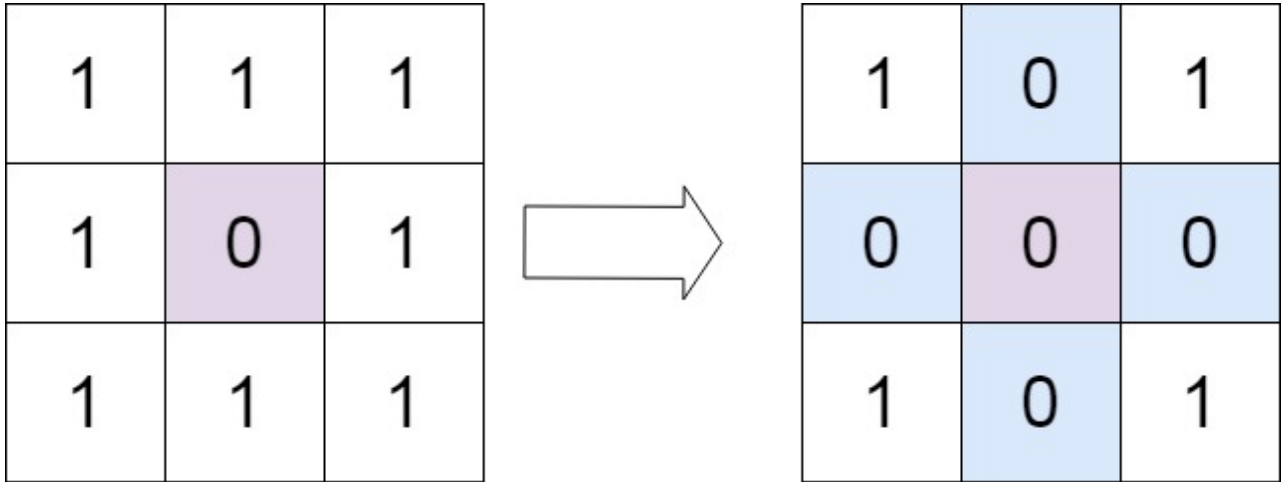
21 |         return n + 1;
22 |     }

```

2.18 73 矩阵置零

73. 矩阵置零

示例 1:

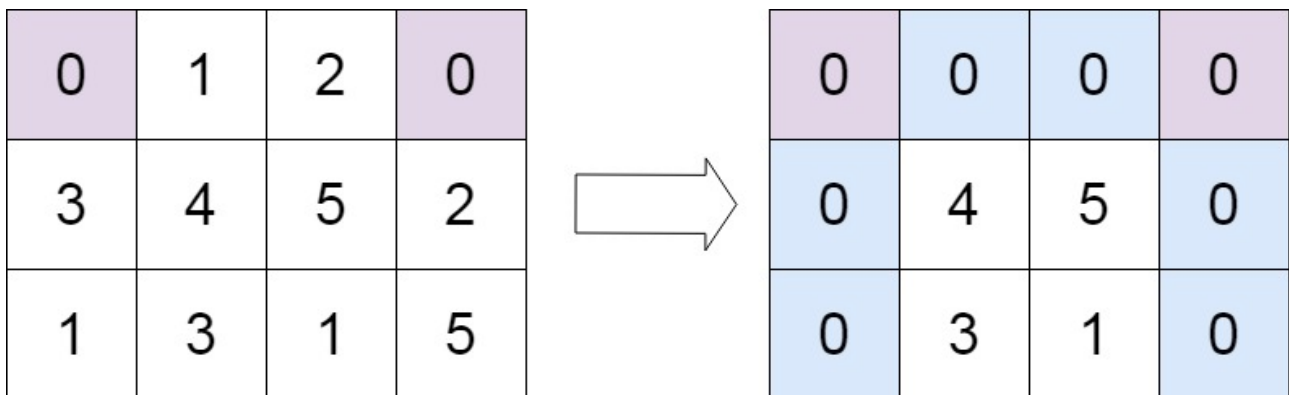


```

1 | 输入: matrix = [[1,1,1],[1,0,1],[1,1,1]]
2 | 输出: [[1,0,1],[0,0,0],[1,0,1]]

```

示例 2:



```

1 | 输入: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
2 | 输出: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

```

提示:

- `m == matrix.length`
- `n == matrix[0].length`
- `1 <= m, n <= 200`
- `-231 <= matrix[i][j] <= 231 - 1`

题解:

可以用辅助空间，用来表示某行某列是否需要置零，当然了官方题解给出了其他回答，可以看看（不过好像可读性不好）

```

1      void setZeroes(vector<vector<int>>& matrix) {
2          int m = matrix.size();
3          int n = matrix[0].size();
4          vector<bool> rows(m), cols(n);
5          for (int i = 0; i < m; i++) {
6              for (int j = 0; j < n; j++) {
7                  if (matrix[i][j] == 0)
8                      rows[i] = cols[j] = true;
9              }
10         }
11         for (int i = 0; i < m; i++) {
12             for (int j = 0; j < n; j++) {
13                 if (rows[i] || cols[j])
14                     matrix[i][j] = 0;
15             }
16         }
17     }

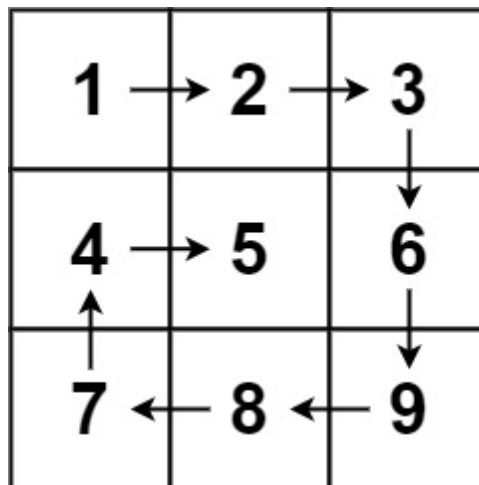
```

2.19 54 螺旋矩阵

[54. 螺旋矩阵](#)

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:

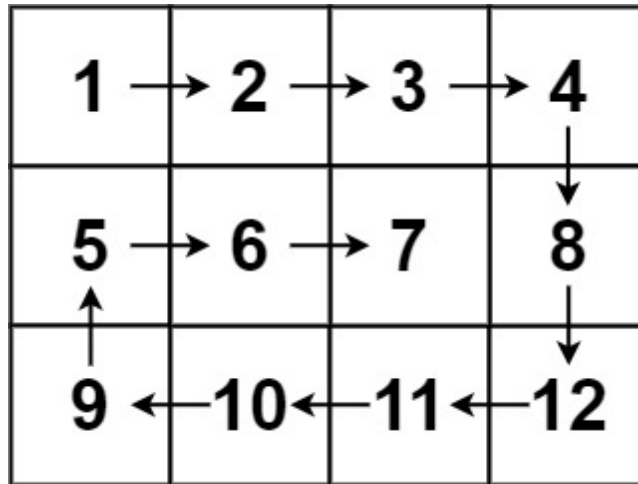


```

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

```

示例 2:



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

题解:

1. 首先设定上下左右边界
2. 其次向右移动到最右, 此时第一行因为已经使用过了, 可以将其从图中删去, 体现在代码中就是重新定义上边界
3. 判断若重新定义后, 上下边界交错, 表明螺旋矩阵遍历结束, 跳出循环, 返回答案
4. 若上下边界不交错, 则遍历还未结束, 接着向下向左向上移动, 操作过程与第一, 二步同理
5. 不断循环以上步骤, 直到某两条边界交错, 跳出循环, 返回答案

```

1  class Solution {
2  public:
3      vector<int> spiralOrder(vector<vector<int>>& matrix) {
4          vector<int> ans;
5          if (matrix.empty()) return ans;
6          int u = 0;
7          int d = matrix.size() - 1;
8          int l = 0;
9          int r = matrix[0].size() - 1;
10         while (true) {
11             for (int i = l; i <= r; i++) ans.push_back(matrix[u][i]);
12             if (++u > d) break;
13             for (int i = u; i <= d; i++) ans.push_back(matrix[i][r]);
14             if (--r < l) break;
15             for (int i = r; i >= l; i--) ans.push_back(matrix[d][i]);
16             if (--d < u) break;
17             for (int i = d; i >= u; i--) ans.push_back(matrix[i][l]);

```

```

18         if (++l > r) break;
19     }
20     return ans;
21 }
22 };

```

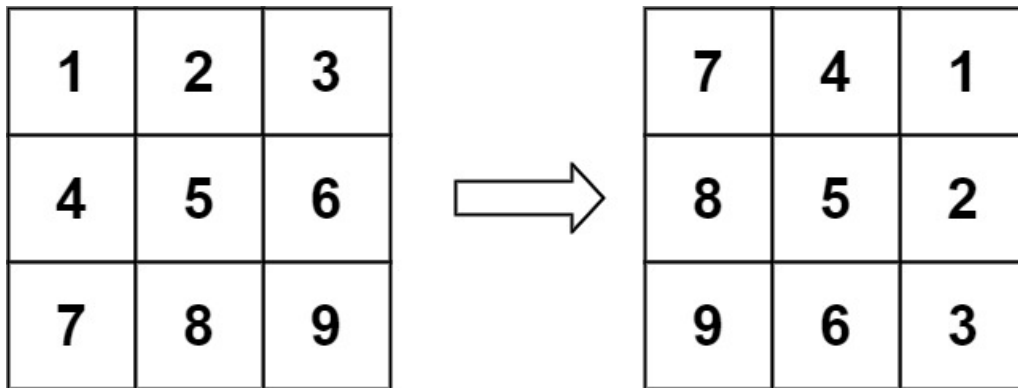
2.20 48 旋转图像

48. 旋转图像

给定一个 $n \times n$ 的二维矩阵 `matrix` 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请**不要** 使用另一个矩阵来旋转图像。

示例 1:

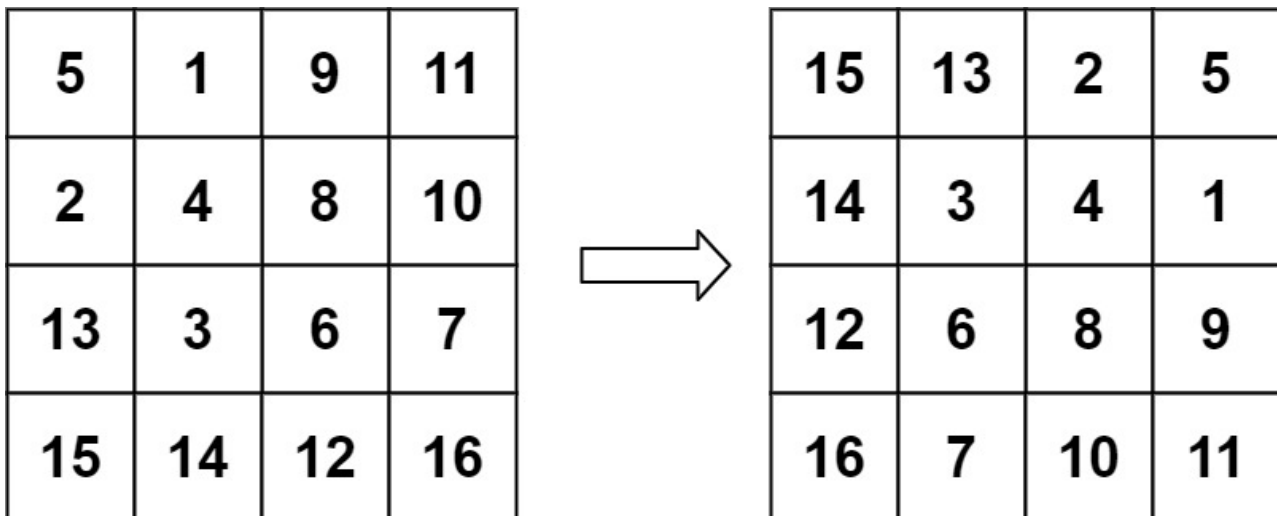


```

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

```

示例 2:



```

1 输入: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
2 输出: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

```

题解:

找规律，有点意思了哈，和数学有关（让我想起了图形学）


```

1 void rotate(vector<vector<int>>& matrix) {
2     if (matrix.empty()) return;
3     int n = matrix.size();
4     for (int i = 0; i < n / 2; ++i) {
5         for (int j = 0; j < (n + 1) / 2; ++j) {
6             int temp = matrix[i][j];
7             matrix[i][j] = matrix[n - j - 1][i];    //这算是一个公式
8             matrix[n - j - 1][i] = matrix[n - i - 1][n - j - 1];
9             matrix[n - i - 1][n - j - 1] = matrix[j][n - i - 1];
10            matrix[j][n - i - 1] = temp;
11        }
12    }
13 }

```

2.21 240 搜索二维矩阵II

[240. 搜索二维矩阵 II](#)

编写一个高效的算法来搜索 $m \times n$ 矩阵 `matrix` 中的一个目标值 `target`。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例 1：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```

1 输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
2                [18,21,23,26,30]], target = 5
   输出：true

```

示例 2：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```

1  输入: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
    [18,21,23,26,30]], target = 20
2  输出: false

```

题解:

有意思了哈，从最右上方开始排除，如果小了说明这一行都小了，往下走，如果大了说明该位置所在的列往下都大了，往左走，知道找到或者越界了

```

1  bool searchMatrix(vector<vector<int>>& matrix, int target) {
2      int m = matrix.size(), n = matrix[0].size();
3      int x = 0, y = n - 1;
4      while (x < m && y >= 0) {
5          if (matrix[x][y] == target)
6              return true;
7          else if (matrix[x][y] > target)
8              --y;
9          else
10             ++x;
11     }
12     return false;
13 }

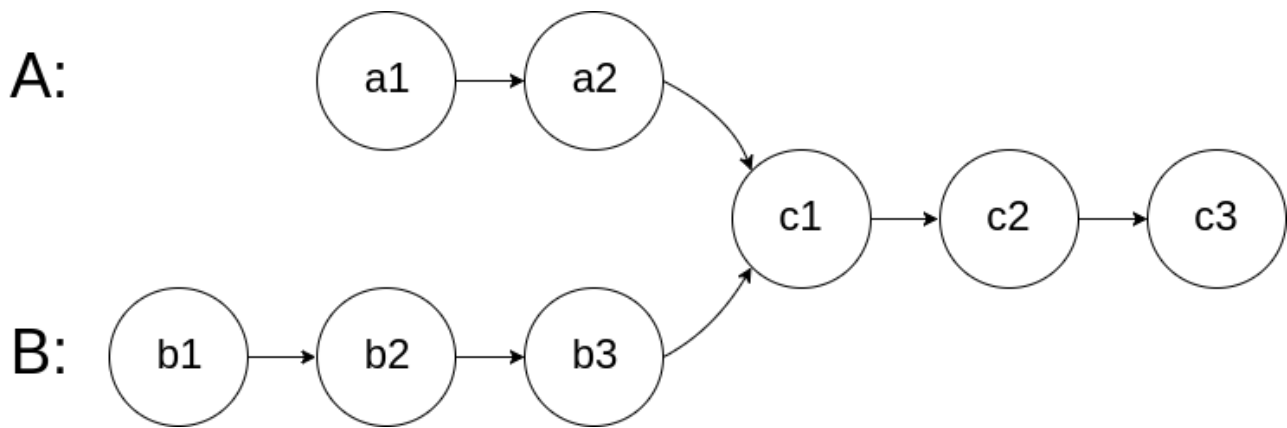
```

2.22 160 相交链表

[160. 相交链表](#)

给你两个单链表的头节点 `headA` 和 `headB`，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 `null`。

图示两个链表在节点 `c1` 开始相交：



题目数据 **保证** 整个链式结构中不存在环。

注意，函数返回结果后，链表必须 **保持其原始结构**。

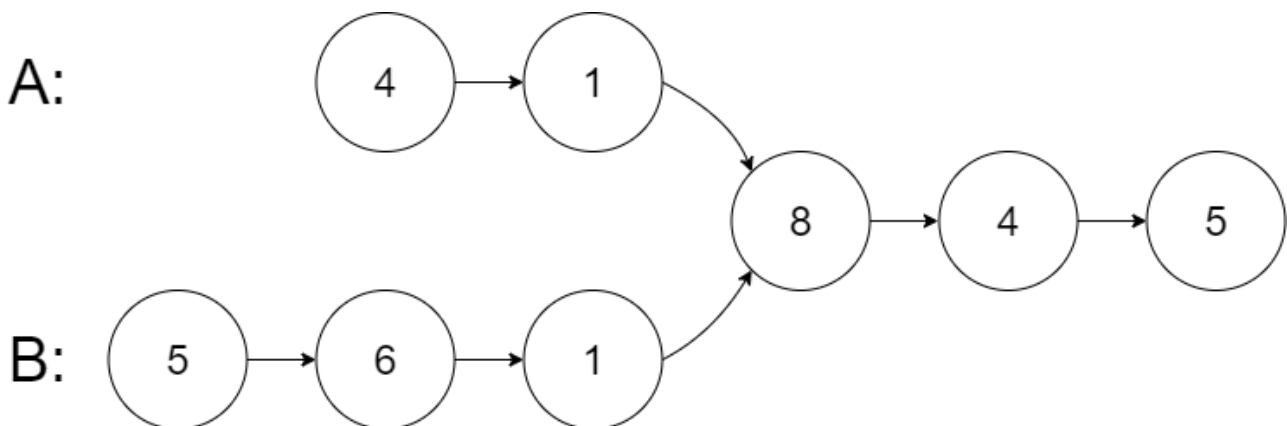
自定义评测：

评测系统 的输入如下（你设计的程序 **不适用** 此输入）：

- `intersectVal` - 相交的起始节点的值。如果不存在相交节点，这一值为 `0`
- `listA` - 第一个链表
- `listB` - 第二个链表
- `skipA` - 在 `listA` 中（从头节点开始）跳到交叉节点的节点数
- `skipB` - 在 `listB` 中（从头节点开始）跳到交叉节点的节点数

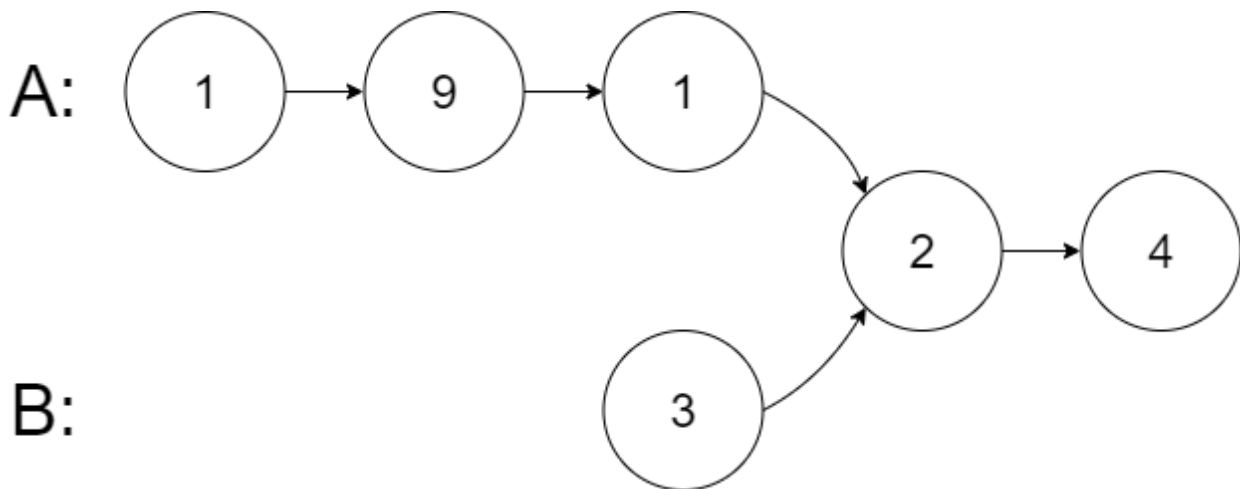
评测系统将根据这些输入创建链式数据结构，并将两个头节点 `headA` 和 `headB` 传递给你的程序。如果程序能够正确返回相交节点，那么你的解决方案将被 **视作正确答案**。

示例 1：



- 1 输入： `intersectVal = 8`, `listA = [4,1,8,4,5]`, `listB = [5,6,1,8,4,5]`, `skipA = 2`, `skipB = 3`
- 2 输出： `Intersected at '8'`
- 3 解释：相交节点的值 为 8（注意，如果两个链表相交则不能为 0）。
- 4 从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,6,1,8,4,5]。
- 5 在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
- 6 - 请注意相交节点的值不为 1，因为在链表 A 和链表 B 之中值为 1 的节点（A 中第二个节点和 B 中第三个节点）是不同的节点。换句话说，它们在内存中指向两个不同的位置，而链表 A 和链表 B 中值为 8 的节点（A 中第三个节点，B 中第四个节点）在内存中指向相同的位置。

示例 2：

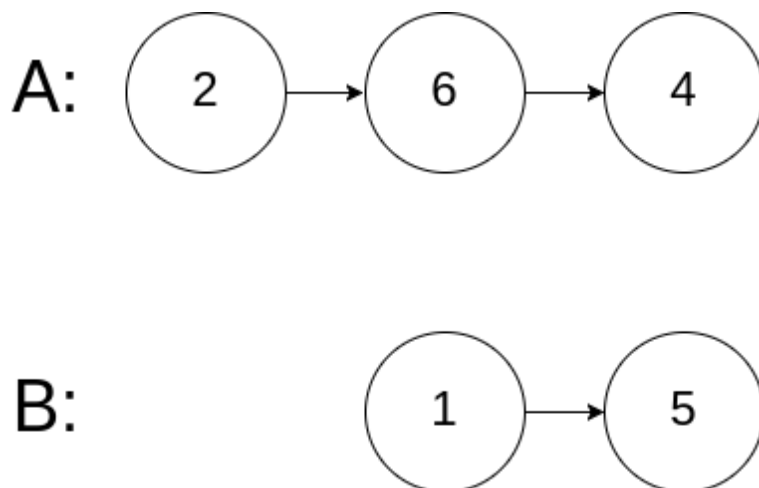


```

1 输入：intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
2 输出：Intersected at '2'
3 解释：相交节点的值为 2（注意，如果两个链表相交则不能为 0）。
4 从各自的表头开始算起，链表 A 为 [1,9,1,2,4]，链表 B 为 [3,2,4]。
5 在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

```

示例 3:



```

1 输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
2 输出：null
3 解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。
4 由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
5 这两个链表不相交，因此返回 null。

```

题解:

其实已经是很多次做这个题了，记住while循环里要判断的是节点本身是否为空，如果判断next则这辈子都出不去循环（不可能相等，我老是判断next）

```

1  ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
2      if (headA == nullptr || headB == nullptr)
3          return nullptr;
4      ListNode *p1 = headA;
5      ListNode *p2 = headB;
6      while (p1 != p2) {
7          p1 = p1 ? p1->next : headB;
8          p2 = p2 ? p2->next : headA;
9      }
10     return p1;
11 }

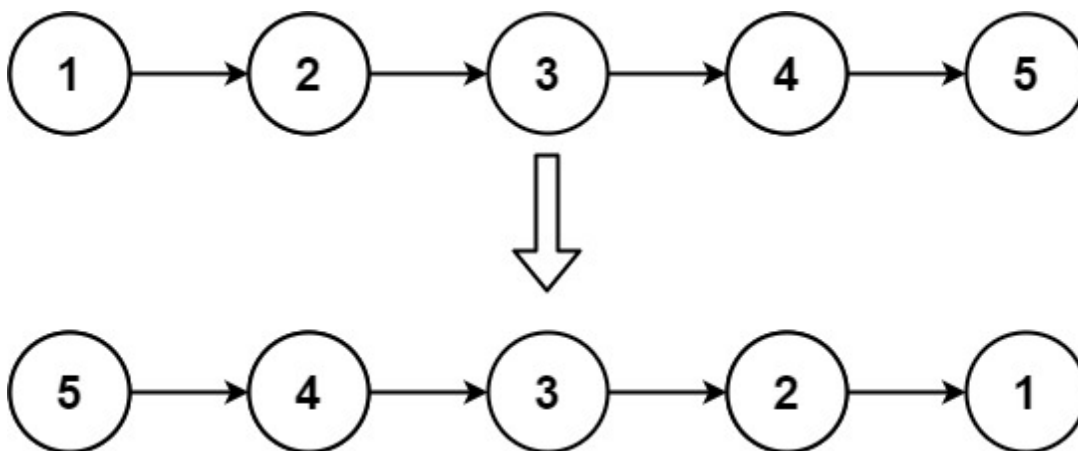
```

2.23 206 反转链表

206. 反转链表

给你单链表的头节点 `head`，请你反转链表，并返回反转后的链表。

示例 1:

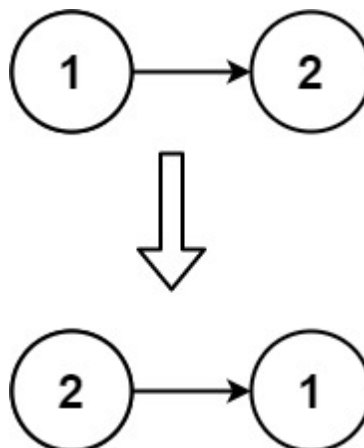


```

1 输入: head = [1,2,3,4,5]
2 输出: [5,4,3,2,1]

```

示例 2:



```

1 输入: head = [1,2]
2 输出: [2,1]

```

示例 3:

1 输入: head = []
2 输出: []

题解:

经典题目了, 先来个前序节点就OK

```
1  ListNode* reverseList(ListNode* head) {  
2      if (head == nullptr)  
3          return nullptr;  
4      ListNode *pre = nullptr;  
5      ListNode *cur = head;  
6      while (cur) {  
7          ListNode *next = cur->next;  
8          cur->next = pre;  
9          pre = cur;  
10         cur = next;  
11     }  
12     return pre;  
13 }
```

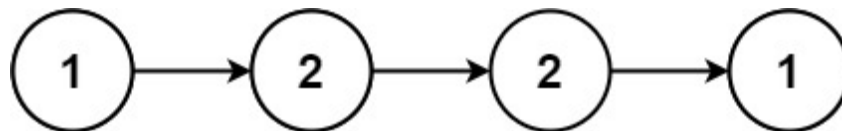
2.24 234 回文链表

[234. 回文链表](#)

给你一个单链表的头节点 `head` , 请你判断该链表是否为

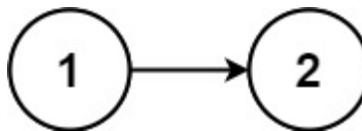
回文链表。如果是, 返回 `true` ; 否则, 返回 `false` 。

示例 1:



1 输入: head = [1,2,2,1]
2 输出: true

示例 2:



1 输入: head = [1,2]
2 输出: false

提示:

- 链表中节点数目在范围 `[1, 105]` 内
- `0 <= Node.val <= 9`

进阶: 你能否用 `O(n)` 时间复杂度和 `O(1)` 空间复杂度解决此题?

题解:

原地构造, 使用快慢指针, 将后面一般的节点逆序, 然后两边一起遍历

```

1  bool isPalindrome(ListNode* head) {
2      ListNode *fast = head;
3      ListNode *slow = head;
4      while (fast != nullptr && fast->next != nullptr) {
5          fast = fast->next->next;
6          slow = slow->next;
7      }
8      fast = head;
9      ListNode *pre = nullptr;
10     while (slow) {
11         ListNode *next = slow->next;
12         slow->next = pre;
13         pre = slow;
14         slow = next;
15     }
16     while (pre != nullptr && pre != fast) {
17         if (pre->val != fast->val)
18             return false;
19         pre = pre->next;
20         fast = fast->next;
21     }
22     return true;
23 }

```

2.25 141 环形链表

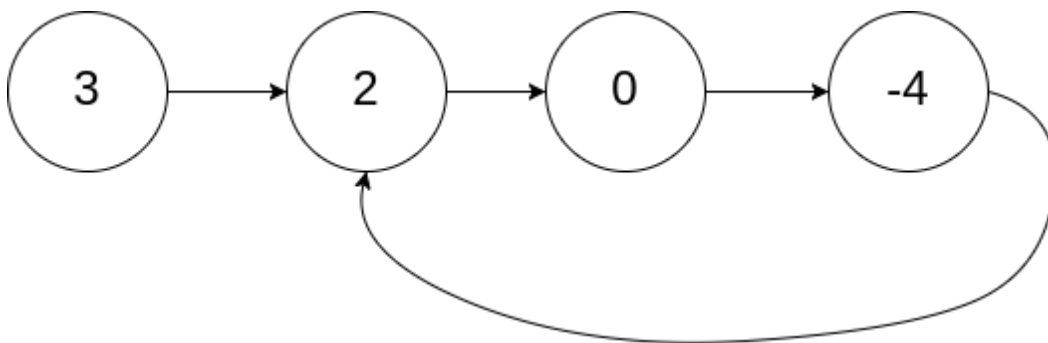
[141. 环形链表](#)

给你一个链表的头节点 `head`，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。 为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。注意：`pos` 不作为参数进行传递。仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true`。否则，返回 `false`。

示例 1:

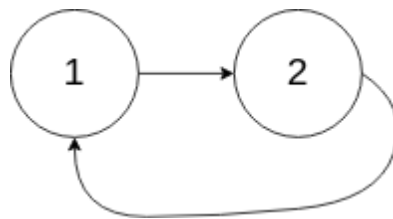


```

1  输入: head = [3,2,0,-4], pos = 1
2  输出: true
3  解释: 链表中有一个环，其尾部连接到第二个节点。

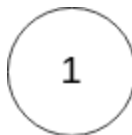
```

示例 2:



1 输入: head = [1,2], pos = 0
 2 输出: true
 3 解释: 链表中有一个环, 其尾部连接到第一个节点。

示例 3:



1 输入: head = [1], pos = -1
 2 输出: false
 3 解释: 链表中没有环。

题解:

也是很经典的题了, 双指针, 快慢指针如果相遇的话一定有环, 如果快指针会走空的话一定无环, 不过有个小细节: 程序开始判断一下单个元素和没有元素的情况, 加快一丢丢时间

```

1  bool hasCycle(ListNode *head) {
2      if (head == nullptr || head->next == nullptr)
3          return false;
4      ListNode *fast = head;
5      ListNode *slow = head;
6      while (fast != nullptr && fast->next != nullptr) {
7          fast = fast->next->next;
8          slow = slow->next;
9          if (fast == slow)
10             return true;
11     }
12     return false;
13 }
  
```

2.26 142 环形链表II

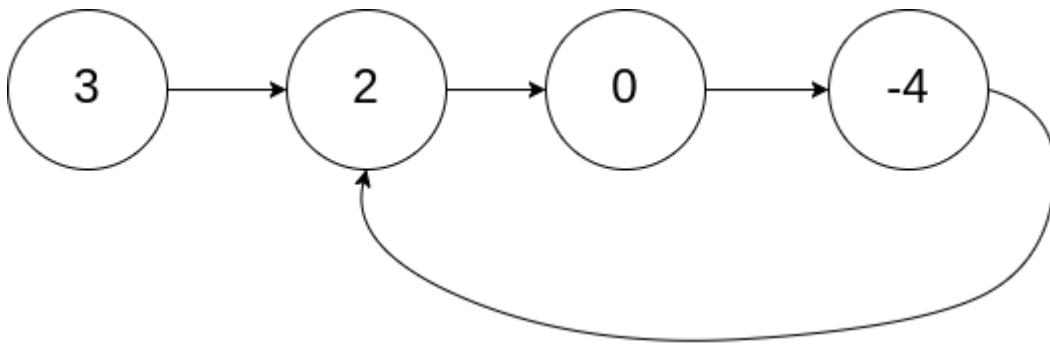
[142. 环形链表 II](#)

给定一个链表的头节点 `head`, 返回链表开始入环的第一个节点。如果链表无环, 则返回 `null`。

如果链表中有某个节点, 可以通过连续跟踪 `next` 指针再次到达, 则链表中存在环。 为了表示给定链表中的环, 评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置 (索引从 0 开始)。如果 `pos` 是 `-1`, 则在该链表中没有环。注意: `pos` 不作为参数进行传递, 仅仅是为了标识链表的实际情况。

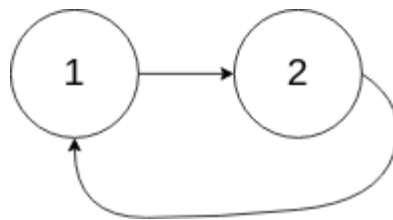
不允许修改 链表。

示例 1:



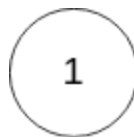
- 1 输入: head = [3,2,0,-4], pos = 1
- 2 输出: 返回索引为 1 的链表节点
- 3 解释: 链表中有一个环, 其尾部连接到第二个节点。

示例 2:



- 1 输入: head = [1,2], pos = 0
- 2 输出: 返回索引为 0 的链表节点
- 3 解释: 链表中有一个环, 其尾部连接到第一个节点。

示例 3:



- 1 输入: head = [1], pos = -1
- 2 输出: 返回 null
- 3 解释: 链表中没有环。

题解:

也是经典的题了, 双指针很轻松解决, 不过原理还是值得搞清楚一下

```

1  ListNode *detectCycle(ListNode *head) {
2      if (head == nullptr || head->next == nullptr)
3          return nullptr;
4      ListNode *fast = head->next->next;
5      ListNode *slow = head->next;
6      while (fast != slow) {
7          if (fast == nullptr || fast->next == nullptr)
8              return nullptr;
9          fast = fast->next->next;
10         slow = slow->next;
11     }
12     fast = head;
13     while (fast != slow) {
14         fast = fast->next;
15         slow = slow->next;
16     }
  
```

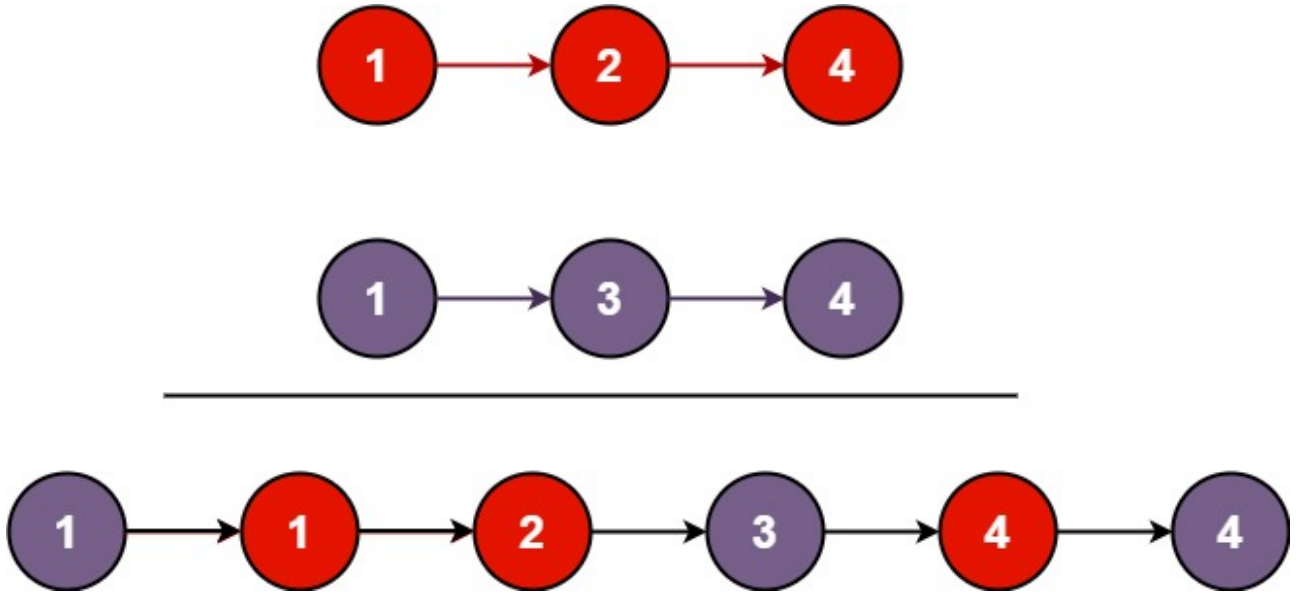
```
17 |         return slow;
18 |     }
```

2.27 21 合并两个有序链表

[21. 合并两个有序链表](#)

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1:



```
1 | 输入: l1 = [1,2,4], l2 = [1,3,4]
2 | 输出: [1,1,2,3,4,4]
```

示例 2:

```
1 | 输入: l1 = [], l2 = []
2 | 输出: []
```

示例 3:

```
1 | 输入: l1 = [], l2 = [0]
2 | 输出: [0]
```

题解:

也是很经典的链表题目了，练熟（细节就是返回的一定是构造出来的“哨兵”的下一个节点）

```
1 | ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
2 |     ListNode *pre = new ListNode(0);
3 |     ListNode *head = pre;
4 |     while (list1 != nullptr && list2 != nullptr) {
5 |         if (list1->val < list2->val) {
6 |             pre->next = list1;
7 |             list1 = list1->next;
8 |         }
9 |         else {
10 |             pre->next = list2;
11 |             list2 = list2->next;
12 |         }
13 |         pre = pre->next;

```

```

14 |
15 |     }
16 |     pre->next = list1 ? list1 : list2;    //别忘记收尾
17 |     return head->next;
18 | }

```

2.28 2 两数相加

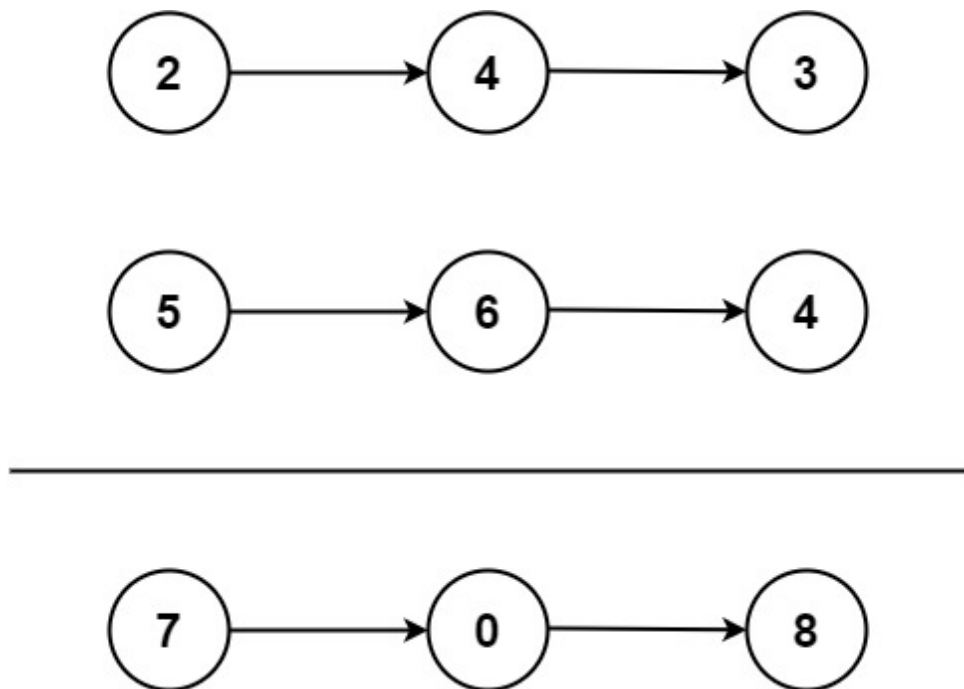
[2. 两数相加](#)

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1:



```

1 | 输入: l1 = [2,4,3], l2 = [5,6,4]
2 | 输出: [7,0,8]
3 | 解释: 342 + 465 = 807.

```

示例 2:

```

1 | 输入: l1 = [0], l2 = [0]
2 | 输出: [0]

```

示例 3:

```

1 | 输入: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
2 | 输出: [8,9,9,9,0,0,0,1]

```

题解:

主要的一个问题是节点两个值会有进位的问题，处理好就可以了[^ note]

```

1 |     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
2 |         if (l1 == nullptr || l2 == nullptr)

```

```

3         return nullptr;
4     int carry = 0, sum = 0;
5     ListNode *pre = new ListNode(0);
6     ListNode *cur = pre;
7     while (l1 != nullptr || l2 != nullptr) {
8         int x = l1 ? l1->val : 0;
9         int y = l2 ? l2->val : 0;
10        sum = x + y + carry;
11        carry = sum / 10;
12        sum = sum % 10;
13        cur->next = new ListNode(sum);
14        cur = cur->next;
15        if (l1) l1 = l1->next;
16        if (l2) l2 = l2->next;
17    }
18    if (carry)
19        cur->next = new ListNode(carry);
20    return pre->next;
21 }

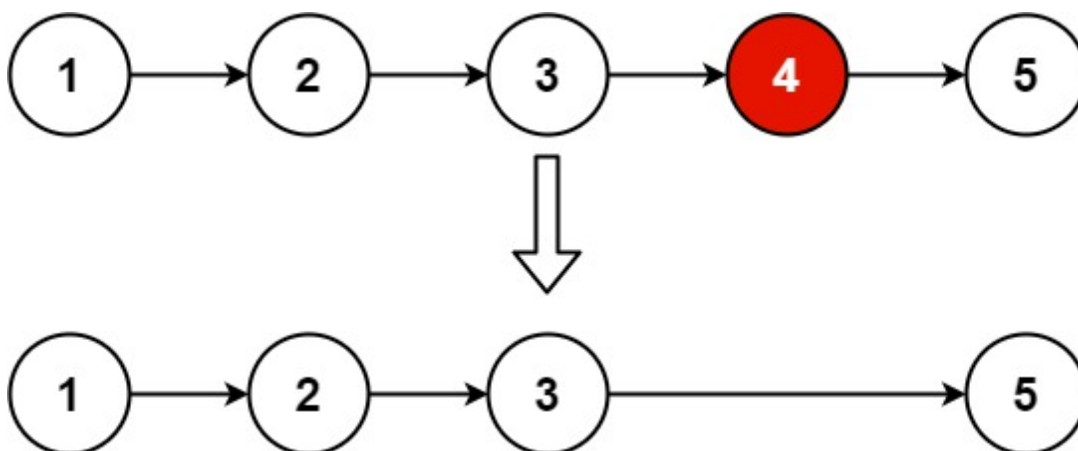
```

2.29 19 删除链表的倒数第N个结点

[19. 删除链表的倒数第 N 个结点](#)

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

示例 1:



```

1 输入: head = [1,2,3,4,5], n = 2
2 输出: [1,2,3,5]

```

示例 2:

```

1 输入: head = [1], n = 1
2 输出: []

```

示例 3:

```

1 输入: head = [1,2], n = 1
2 输出: [1]

```

题解:

想法是没问题的，不过没Coding出来，再好好看看（一开始用了while循环，不太好搞啊）

```

1  ListNode* removeNthFromEnd(ListNode* head, int n) {
2      int len = 0;
3      ListNode *cur = head;
4      ListNode *pre = new ListNode(0);
5      pre->next = cur;
6      while (cur != nullptr) {
7          ++len;
8          cur = cur->next;
9      }
10     cur = pre;
11     int cnt = len - n + 1;
12     for (int i = 1; i < len - n + 1; ++i)
13         cur = cur->next;
14     cur->next = cur->next->next;
15     ListNode *ans = pre->next;
16     delete pre;
17     return ans;
18 }

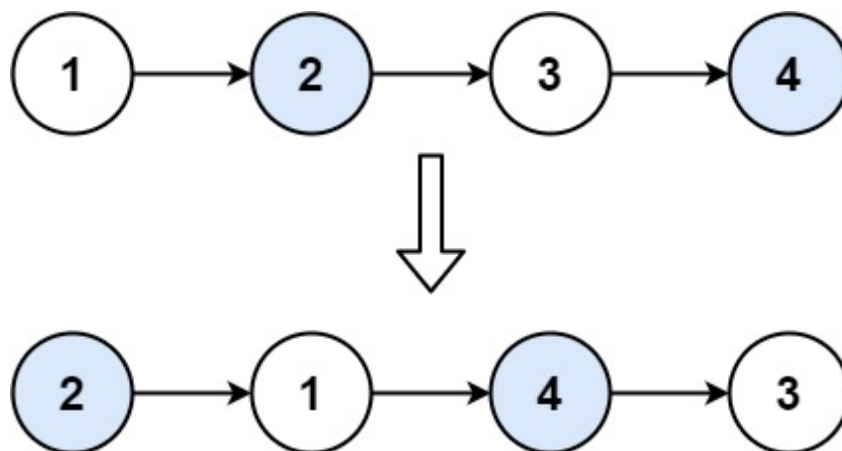
```

2.30 24 两两交换链表中的节点

[24. 两两交换链表中的节点](#)

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在`不修改节点内部的值`的情况下完成本题（即，只能进行节点交换）。

示例 1:



```

1  输入: head = [1,2,3,4]
2  输出: [2,1,4,3]

```

示例 2:

```

1  输入: head = []
2  输出: []

```

示例 3:

```

1  输入: head = [1]
2  输出: [1]

```

题解:

思路也没问题，不过又没Coding出来，注意好好利用pre指针，不要想着cur了容易出错呢

```

1  ListNode* swapPairs(ListNode* head) {
2      ListNode *newHead = new ListNode(0);
3      newHead->next = head;
4      ListNode *temp = newHead;    //善用哨兵
5      while (temp->next && temp->next->next) {
6          ListNode *node1 = temp->next;
7          ListNode *node2 = temp->next->next;
8          temp->next = node2;
9          node1->next = node2->next;
10         node2->next = node1;
11         temp = node1;
12     }
13     ListNode *res = newHead->next;
14     delete newHead;
15     return res;
16 }

```

2.31 25 K个一组翻转链表

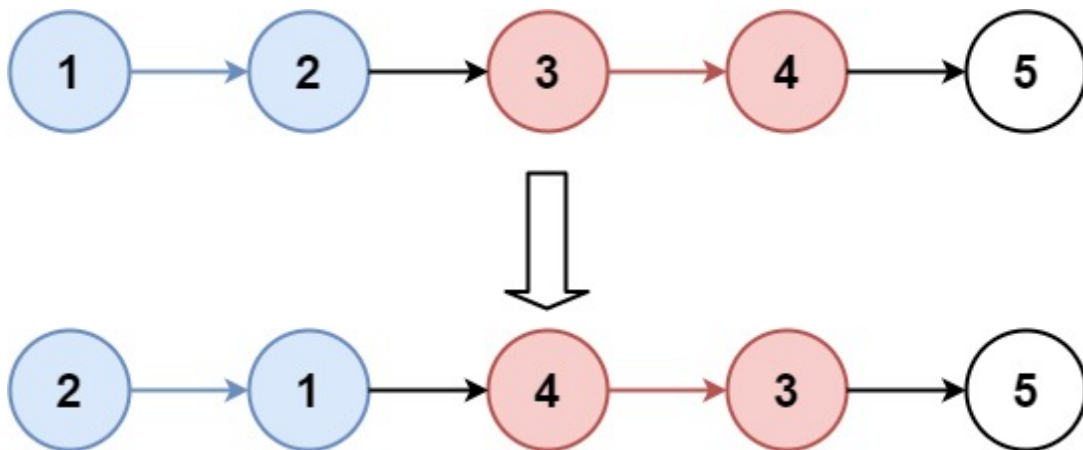
[25. K 个一组翻转链表](#)

给你链表的头节点 `head`，每 `k` 个节点一组进行翻转，请你返回修改后的链表。

`k` 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 `k` 的整数倍，那么请将最后剩余的节点保持原有顺序。

你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1:

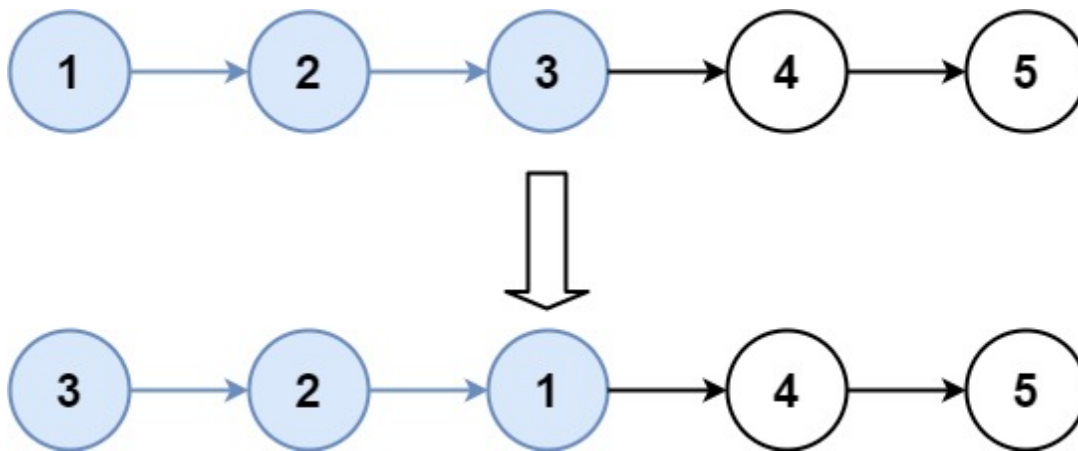


```

1  输入: head = [1,2,3,4,5], k = 2
2  输出: [2,1,4,3,5]

```

示例 2:



1 输入: head = [1,2,3,4,5], k = 3
2 输出: [3,2,1,4,5]

题解:

思路是没问题的, 用一个处理函数返回处理后的头和尾, 不过注意一下如何Coding

```
1  pair<ListNode*, ListNode*> myReverse(ListNode *head, ListNode *end) {
2      ListNode *pre = end->next;
3      ListNode *cur = head;
4      while (pre != end) {
5          ListNode *nex = cur->next;
6          cur->next = pre;
7          pre = cur;
8          cur = nex;
9      }
10     return {end, head};
11 }
12 ListNode* reverseKGroup(ListNode* head, int k) {
13     ListNode *res = new ListNode(-1);
14     res->next = head;
15     ListNode *pre = res;
16
17     while (head) {
18         ListNode *end = pre;
19         for (int i = 0; i < k; ++i) {
20             end = end->next;
21             if (!end)
22                 return res->next;
23         }
24         ListNode *nex = end->next;
25         pair<ListNode*, ListNode*> temp = myReverse(head, end);
26         head = temp.first, end = temp.second;
27         pre->next = head;
28         end->next = nex;
29         pre = end;
30         head = nex;
31     }
32     return res->next;
33 }
```

2.32 138 随机链表的复制

138. 随机链表的复制

给你一个长度为 `n` 的链表，每个节点包含一个额外增加的随机指针 `random`，该指针可以指向链表中的任何节点或空节点。

构造这个链表的 **深拷贝**。深拷贝应该正好由 `n` 个 **全新** 节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 `next` 指针和 `random` 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。**复制链表中的指针都不应指向原链表中的节点**。

例如，如果原链表中有 `X` 和 `Y` 两个节点，其中 `X.random --> Y`。那么在复制链表中对应的两个节点 `x` 和 `y`，同样有 `x.random --> y`。

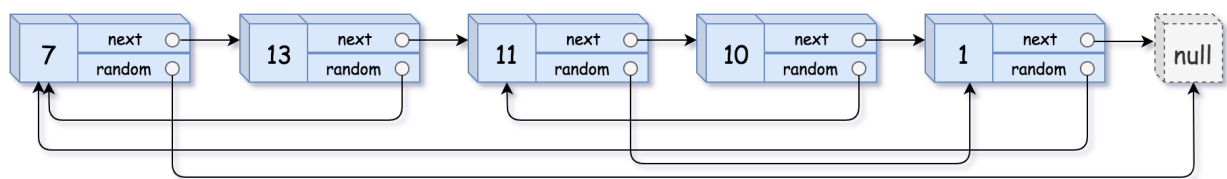
返回复制链表的头节点。

用一个由 `n` 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 `[val, random_index]` 表示：

- `val`：一个表示 `Node.val` 的整数。
- `random_index`：随机指针指向的节点索引（范围从 `0` 到 `n-1`）；如果不指向任何节点，则为 `null`。

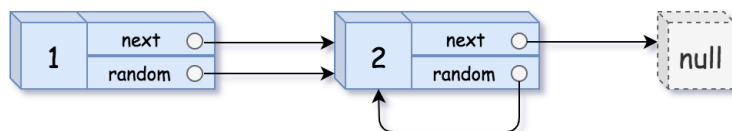
你的代码 **只** 接受原链表的头节点 `head` 作为传入参数。

示例 1：



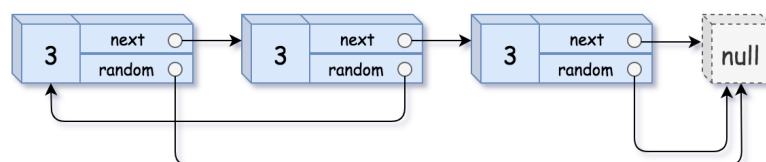
```
1 输入: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
2 输出: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

示例 2：



```
1 输入: head = [[1,1],[2,1]]
2 输出: [[1,1],[2,1]]
```

示例 3：




```
1 输入: head = [[3,null],[3,0],[3,null]]
2 输出: [[3,null],[3,0],[3,null]]
```

题解:

这其实是第二次做这个题，还是有些不熟练啊

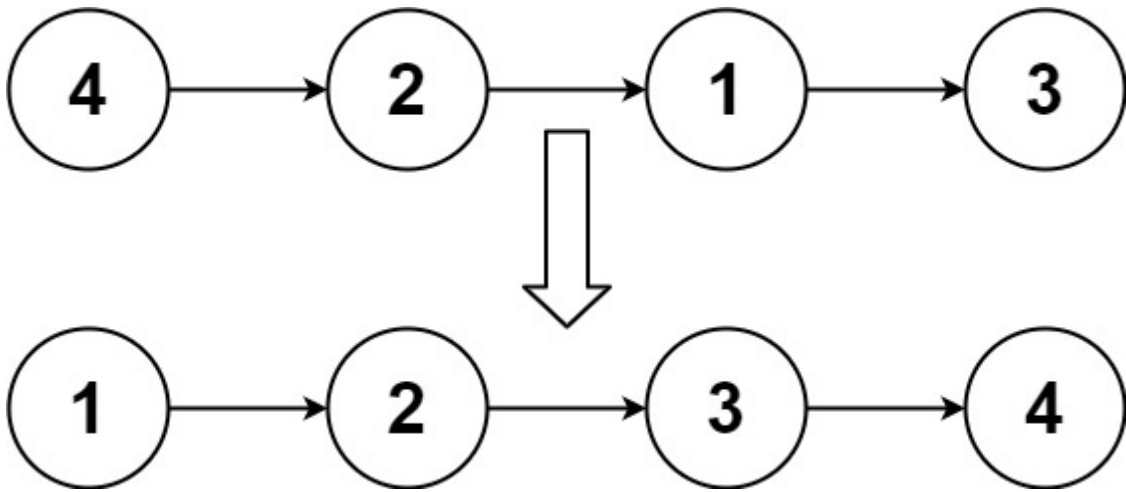
```
1  Node* copyRandomList(Node* head) {
2      if (head == nullptr)
3          return nullptr;
4      Node *cur = head;
5      Node *next = nullptr;
6      while (cur != nullptr) {
7          next = cur->next;
8          Node *copyNode = new Node(cur->val);
9          copyNode->next = next;
10         cur->next = copyNode;
11         cur = next;
12     }
13     Node *copyNode = nullptr;
14     cur = head;
15     while (cur != nullptr) {
16         next = cur->next->next;
17         copyNode = cur->next;
18         copyNode->random = (cur->random == nullptr) ? nullptr : cur->random->next;
19         cur = next;
20     }
21     Node *res = head->next;
22     cur = head;
23     while (cur != nullptr) {
24         next = cur->next->next;
25         copyNode = cur->next;
26         cur->next = next;
27         copyNode->next = (next == nullptr) ? nullptr : next->next;
28         cur = cur->next;
29     }
30     return res;
31 }
```

2.33 148 排序链表

[148. 排序链表](#)

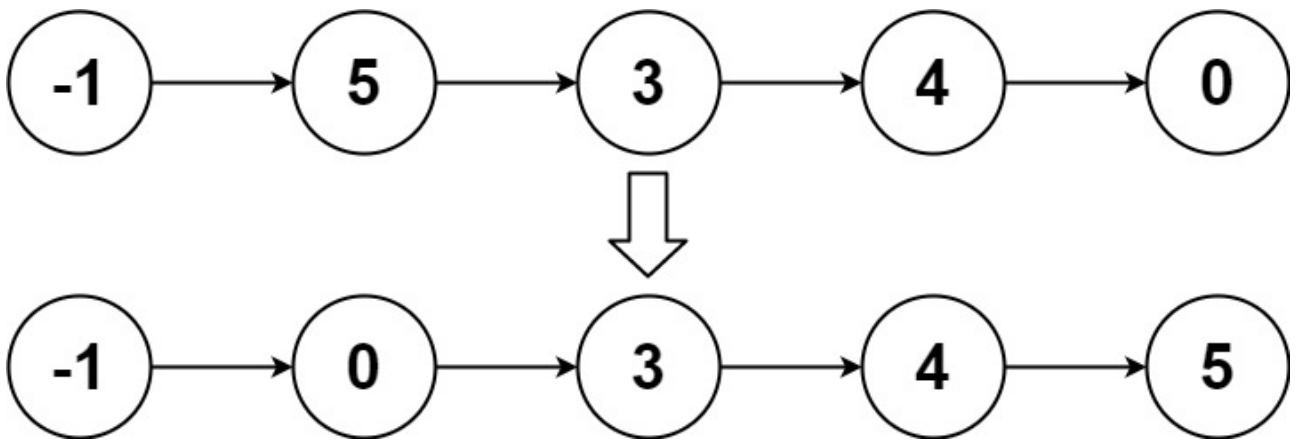
给你链表的头结点 `head`，请将其按 **升序** 排列并返回 **排序后的链表**。

示例 1:



1 输入: head = [4,2,1,3]
2 输出: [1,2,3,4]

示例 2:



1 输入: head = [-1,5,3,4,0]
2 输出: [-1,0,3,4,5]

示例 3:

1 输入: head = []
2 输出: []

题解:

归并排序思想, 记得对链表拆分

```

1  ListNode* sortList(ListNode* head) {
2      if (head == nullptr)
3          return nullptr;
4      int length = 0;
5      ListNode *node = head;
6      while (node != nullptr) {
7          ++length;
8          node = node->next;
9      }
10     ListNode *help = new ListNode(0, head);
11     for (int subLength = 1; subLength < length; subLength <= 1) {
12         ListNode *pre = help, *cur = help->next;
13         while (cur != nullptr) {
14             ListNode *head1 = cur;
  
```

```

15         for(int i = 1; i < subLength && cur->next != nullptr; ++i)
16             cur = cur->next;
17         ListNode *head2 = cur->next;
18         cur->next = nullptr;
19         cur = head2;
20         for (int i = 1; i < subLength && cur != nullptr && cur->next != nullptr;
++i)
21             cur = cur->next;
22         ListNode *next = nullptr;
23         if (cur != nullptr) {
24             next = cur->next;
25             cur->next = nullptr;
26         }
27         ListNode *merged = merge(head1, head2);
28         pre->next = merged;
29         while (pre->next != nullptr)
30             pre = pre->next;
31         cur = next;
32     }
33 }
34 return help->next;
35 }
36
37 ListNode *merge(ListNode *head1, ListNode *head2) {
38     ListNode *newHead = new ListNode(0);
39     ListNode *temp = newHead, *temp1 = head1, *temp2 = head2;
40     while (temp1 != nullptr && temp2 != nullptr) {
41         if (temp1->val <= temp2->val) {
42             temp->next = temp1;
43             temp1 = temp1->next;
44         }
45         else {
46             temp->next = temp2;
47             temp2 = temp2->next;
48         }
49         temp = temp->next;
50     }
51     temp->next = temp1 ? temp1 : temp2;
52     return newHead->next;
53 }

```

2.34 合并K个升序链表

[23. 合并 K 个升序链表](#)

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1:

```
1  输入: lists = [[1,4,5],[1,3,4],[2,6]]
2  输出: [1,1,2,3,4,4,5,6]
3  解释: 链表数组如下:
4  [
5      1->4->5,
6      1->3->4,
7      2->6
8  ]
9  将它们合并到一个有序链表中得到。
10 1->1->2->3->4->4->5->6
```

示例 2:

```
1  输入: lists = []
2  输出: []
```

示例 3:

```
1  输入: lists = [[]]
2  输出: []
```

题解:

归并排序思想, 把vector中的链表两两合并就OK (好好看看)

```
1  ListNode* mergeKLists(vector<ListNode*>& lists) {
2      if (lists.size() == 0)
3          return nullptr;
4      return mergeSort(lists, 0, lists.size() - 1);
5  }
6  ListNode *mergeSort(vector<ListNode *> &lists, int l, int r) {
7      if (l > r)
8          return nullptr;
9      if (l == r)
10         return lists[l];
11     int mid = l + ((r - l) >> 1);
12     return merge(mergeSort(lists, l, mid), mergeSort(lists, mid + 1, r));
13 }
14
15 ListNode *merge(ListNode *node1, ListNode *node2) {
16     ListNode *pre = new ListNode(0);
17     ListNode *cur = pre;
18     while (node1 && node2) {
19         if (node1->val <= node2->val) {
20             cur->next = node1;
21             node1 = node1->next;
22         }
23         else {
24             cur->next = node2;
25             node2 = node2->next;
26         }
27         cur = cur->next;
28     }
29     if (node1)
30         cur->next = node1;
31     if (node2)
32         cur->next = node2;
```

```

33 |
34 |         return pre->next;
35 |     }

```

2.35 146 LRU缓存

146. LRU 缓存

请你设计并实现一个满足 [LRU \(最近最少使用\) 缓存](#) 约束的数据结构。

实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以 **正整数** 作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1`。
- `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值 `value`；如果不存在，则向缓存中插入该组 `key-value`。如果插入操作导致关键字数量超过 `capacity`，则应该 **逐出** 最久未使用的关键字。

函数 `get` 和 `put` 必须以 `O(1)` 的平均时间复杂度运行。

示例：

```

1  输入
2  ["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
3  [[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
4  输出
5  [null, null, null, 1, null, -1, null, -1, 3, 4]
6
7  解释
8  LRUCache lRUcache = new LRUCache(2);
9  lRUcache.put(1, 1); // 缓存是 {1=1}
10 lRUcache.put(2, 2); // 缓存是 {1=1, 2=2}
11 lRUcache.get(1);    // 返回 1
12 lRUcache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
13 lRUcache.get(2);    // 返回 -1 (未找到)
14 lRUcache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
15 lRUcache.get(1);    // 返回 -1 (未找到)
16 lRUcache.get(3);    // 返回 3
17 lRUcache.get(4);    // 返回 4

```

题解：

挺六的，题目要手搓一个双向链表并且用到了一个哈希map辅助查询

```

1  struct DLinkedNode {    //双向链表（表头表示最近使用过，表尾为最近最久未使用）
2      int key, value;
3      DLinkedNode *prev;
4      DLinkedNode *next;
5      DLinkedNode(): key(0), value(0), prev(nullptr), next(nullptr) {}
6      DLinkedNode(int _key, int _value): key(_key), value(_value), prev(nullptr),
next(nullptr) {}
7  };
8
9  class LRUCache {
10 private:

```

```

11 unordered_map<int, DLinkedNode*> cache;
12 DLinkedNode* head;
13 DLinkedNode* tail;
14 int size;
15 int capacity;
16
17 public:
18 LRUCache(int _capacity): capacity(_capacity), size(0) {
19     head = new DLinkedNode();
20     tail = new DLinkedNode();
21     head->next = tail;
22     tail->prev = head;
23 }
24
25 int get(int key) {
26     if (!cache.count(key))
27         return -1;
28     DLinkedNode *node = cache[key];
29     moveToHead(node);
30     return node->value;
31 }
32
33 void put(int key, int value) {
34     if (!cache.count(key)) {
35         DLinkedNode *node = new DLinkedNode(key, value);
36         cache[key] = node;
37         addToHead(node);
38         ++size;
39         if (size > capacity) {
40             DLinkedNode *removed = removeTail();
41             cache.erase(removed->key);
42             delete removed;
43             --size;
44         }
45     }
46     else {
47         DLinkedNode *node = cache[key];
48         node->value = value;
49         moveToHead(node);
50     }
51 }
52
53 void addToHead(DLinkedNode *node) {
54     node->prev = head;
55     node->next = head->next;
56     head->next->prev = node;
57     head->next = node;
58 }
59
60 void removeNode(DLinkedNode *node) {
61     node->prev->next = node->next;
62     node->next->prev = node->prev;
63 }
64
65 void moveToHead(DLinkedNode *node) {
66     removeNode(node);

```

```

67         addToHead(node);
68     }
69     DLinkedList *removeTail() {
70         DLinkedList *node = tail->prev;
71         removeNode(node);
72         return node;
73     }
74 };

```

2.36 543 二叉树的路径

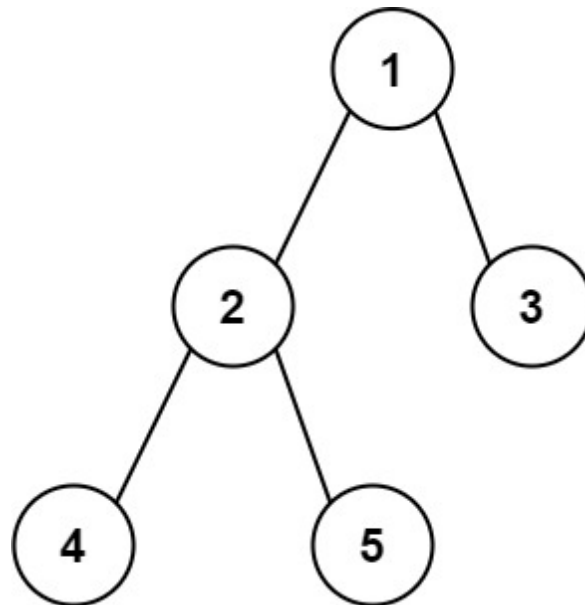
543. 二叉树的直径

给你一棵二叉树的根节点，返回该树的 **直径**。

二叉树的 **直径** 是指树中任意两个节点之间最长路径的 **长度**。这条路径可能经过也可能不经过根节点 `root`。

两节点之间路径的 **长度** 由它们之间边数表示。

示例 1:



```

1 输入: root = [1,2,3,4,5]
2 输出: 3
3 解释: 3 , 取路径 [4,2,1,3] 或 [5,2,1,3] 的长度。

```

示例 2:

```

1 输入: root = [1,2]
2 输出: 1

```

题解:

其实就是套路的一个变体，向左右孩子要信息（最大的节点数，因为路劲=节点数-1）

```

1 | int maxRes = 1;
2 | int diameterOfBinaryTree(TreeNode* root) {
3 |     depth(root);
4 |     return maxRes - 1;
5 | }
6 | int depth(TreeNode *root) {
7 |     if (root == nullptr)
8 |         return 0;
9 |     int l = depth(root->left);
10 |    int r = depth(root->right);
11 |    maxRes = max(maxRes, l + r + 1);
12 |    return max(l, r) + 1;
13 | }

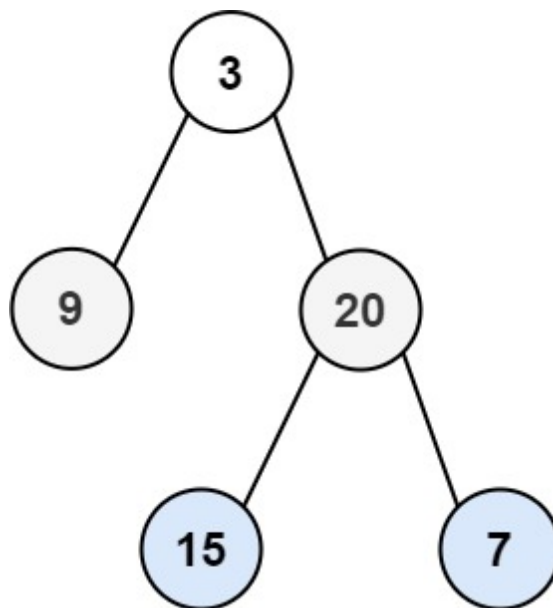
```

2.37 102 二叉树的层序遍历

[102. 二叉树的层序遍历](#)

给你二叉树的根节点 `root`，返回其节点值的 **层序遍历**。（即逐层地，从左到右访问所有节点）。

示例 1:



```

1 | 输入: root = [3,9,20,null,null,15,7]
2 | 输出: [[3],[9,20],[15,7]]

```

示例 2:

```

1 | 输入: root = [1]
2 | 输出: [[1]]

```

示例 3:

```

1 | 输入: root = []
2 | 输出: []

```

题解:

用到了队列来辅助完成

```

1 | vector<vector<int>> levelOrder(TreeNode* root) {
2 |     vector<vector<int>> ans;

```



```

3         if (root == nullptr)
4             return ans;
5         queue<TreeNode*> q;
6         vector<int> help;
7         q.push(root);
8         while (!q.empty()) {
9             int temp = q.size();
10            for (int i = 0; i < temp; ++i) {
11                TreeNode *temp = q.front();
12                help.push_back(temp->val);
13                if (temp->left)
14                    q.push(temp->left);
15                if (temp->right)
16                    q.push(temp->right);
17                q.pop();
18            }
19            ans.push_back(help);
20            help.erase(help.begin(), help.end());
21        }
22        return ans;
23    }

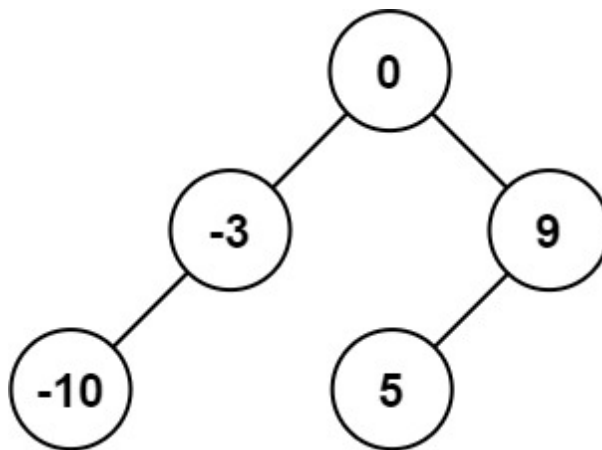
```

2.38 108 将有序数组转换为二叉搜索树

[108. 将有序数组转换为二叉搜索树](#)

给你一个整数数组 `nums`，其中元素已经按 **升序** 排列，请你将其转换为一棵 **平衡** 二叉搜索树。

示例 1:

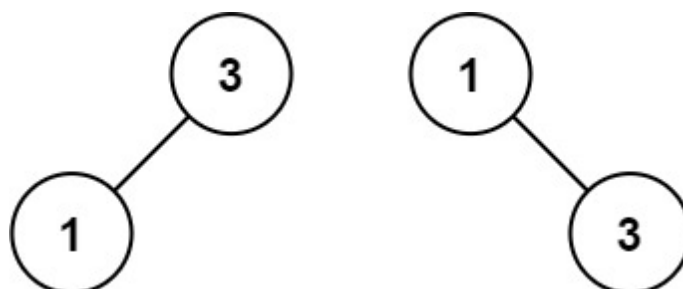


```

1  输入: nums = [-10,-3,0,5,9]
2  输出: [0,-3,9,-10,null,5]
3  解释: [0,-10,5,null,-3,null,9] 也将被视为正确答案:

```

示例 2:



```
1 输入: nums = [1,3]
2 输出: [3,1]
3 解释: [1,null,3] 和 [3,1] 都是高度平衡二叉搜索树。
```

题解:

二叉树中序递归，思路得打开啊

```
1  TreeNode* sortedArrayToBST(vector<int>& nums) {
2      return buildTree(nums, 0, nums.size() - 1);
3  }
4
5  TreeNode *buildTree(vector<int> &nums, int l, int r) {
6      if (l > r)
7          return nullptr;
8      int mid = l + ((r - l) >> 1);
9      TreeNode *root = new TreeNode(nums[mid]);
10     root->left = buildTree(nums, l, mid - 1);
11     root->right = buildTree(nums, mid + 1, r);
12     return root;
13 }
```

2.39 98 验证二叉搜索树

[98. 验证二叉搜索树](#)

给你一个二叉树的根节点 `root`，判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

- 节点的左子树只包含小于 当前节点的数。
- 节点的右子树只包含 大于 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
1 输入: root = [2,1,3]
2 输出: true
```

示例 2:

```
1 输入: root = [5,1,4,null,null,3,6]
2 输出: false
3 解释: 根节点的值是 5，但是右子节点的值是 4。
```

题解:

二叉树递归，主要就是看看判断逻辑怎么处理

```

1      bool helper(TreeNode *root, long long lower, long long upper) {
2          if (root == nullptr)
3              return true;
4          if (root->val <= lower || root->val >= upper)
5              return false;
6          return helper(root->left, lower, root->val) && helper(root->right, root->val,
upper);
7      }
8      bool isValidBST(TreeNode* root) {
9          return helper(root, LONG_MIN, LONG_MAX);
10     }

```

2.40 98 验证二叉搜索树

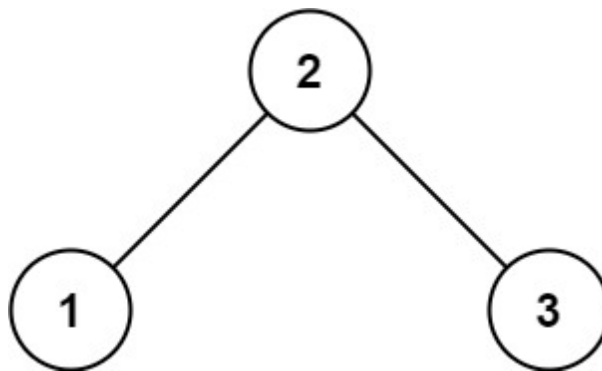
[98. 验证二叉搜索树](#)

给你一个二叉树的根节点 `root`，判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含 **大于** 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1：

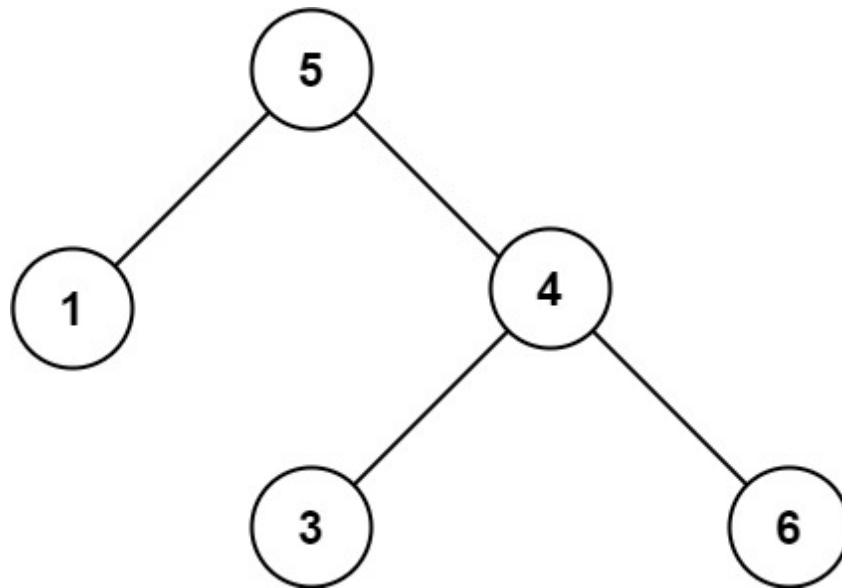


```

1  输入：root = [2,1,3]
2  输出：true

```

示例 2：



```
1 输入: root = [5,1,4,null,null,3,6]
2 输出: false
3 解释: 根节点的值是 5 , 但是右子节点的值是 4 。
```

题解:

判断当前节点是否属于左右节点的值区间范围内 (开区间) 即:

$root \rightarrow value \in (root \rightarrow left \rightarrow value, root \rightarrow right \rightarrow value)$, 递归判断当前节点左右孩子是否满足, 如果为空节点返回true

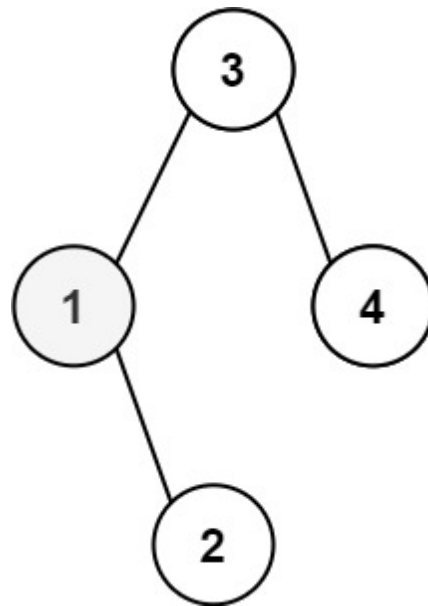
```
1  bool helper(TreeNode *root, long long lower, long long upper) {
2      if (root == nullptr)
3          return true;
4      if (root->val <= lower || root->val >= upper)
5          return false;
6      return helper(root->left, lower, root->val) && helper(root->right, root->val,
upper);
7  }
8  bool isValidBST(TreeNode* root) {
9      return helper(root, LONG_MIN, LONG_MAX);
10 }
```

2.41 230 二叉搜索树中第K小的元素

[230. 二叉搜索树中第K小的元素](#)

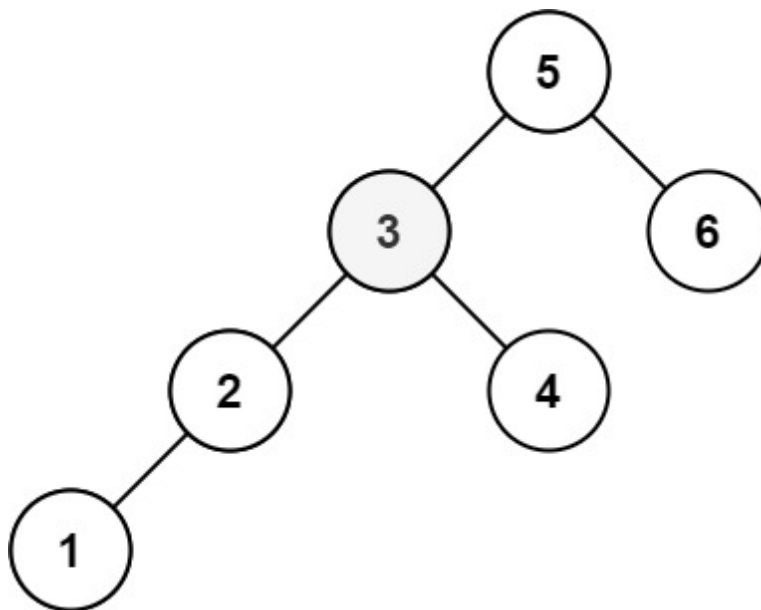
给定一个二叉搜索树的根节点 `root` , 和一个整数 `k` , 请你设计一个算法查找其中第 `k` 个最小元素 (从 1 开始计数)。

示例 1:



1 | 输入: root = [3,1,4,null,2], k = 1
2 | 输出: 1

示例 2:



1 | 输入: root = [5,3,6,2,4,null,null,1], k = 3
2 | 输出: 3

题解:

最简单的中序遍历递归方法: (当然有更优解法, 不过有兴趣再看看吧)

```

1 | int cnt = 0;
2 | int kthSmallest(TreeNode* root, int k) {
3 |     if (root == nullptr)
4 |         return 0;
5 |     int leftval = kthSmallest(root->left, k);
6 |     ++cnt;
7 |     if (cnt == k)
8 |         return root->val;
9 |     int rightval = kthSmallest(root->right, k);
10 |    return leftval ? leftval : rightval;
11 | }

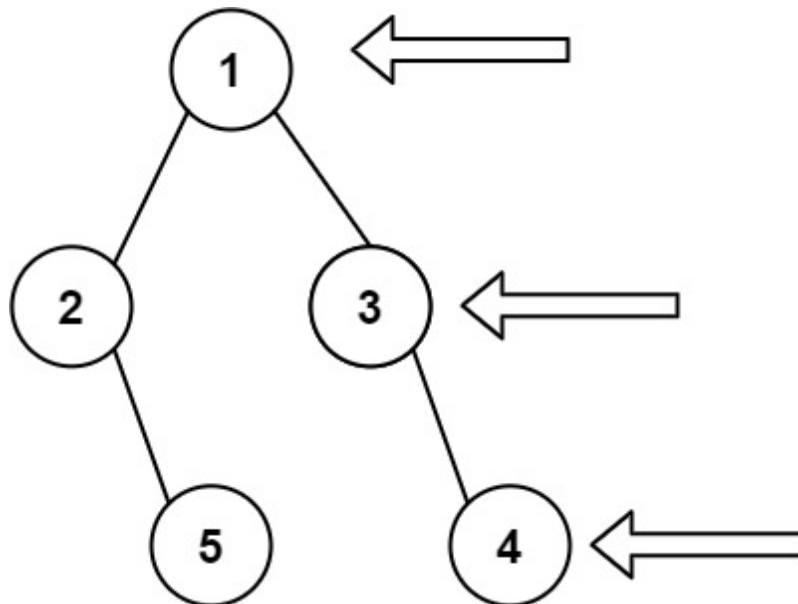
```

2.42 199 二叉树的右视图

[199. 二叉树的右视图](#)

给定一个二叉树的 **根节点** `root`，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例 1:



```

1 | 输入: [1,2,3,null,5,null,4]
2 | 输出: [1,3,4]

```

示例 2:

```

1 | 输入: [1,null,3]
2 | 输出: [1,3]

```

示例 3:

```

1 | 输入: []
2 | 输出: []

```

题解:

用一个队列实现层序遍历，可以再考虑考虑用常数实现，到时候去看看之前看过的一个视频

```

1 | vector<int> rightSideView(TreeNode* root) {
2 |     vector<int> ans;
3 |     if (root == nullptr)

```

```

4         return ans;
5     queue<TreeNode*> que;
6     que.push(root);
7     while (!que.empty()) {
8         int temp = que.size();
9         while (true) {
10            TreeNode *cur = que.front();
11            que.pop();
12            --temp;
13            if (cur->left) que.push(cur->left);
14            if (cur->right) que.push(cur->right);
15            if (temp == 0) {
16                ans.push_back(cur->val);
17                break;
18            }
19        }
20    }
21    return ans;
22 }

```

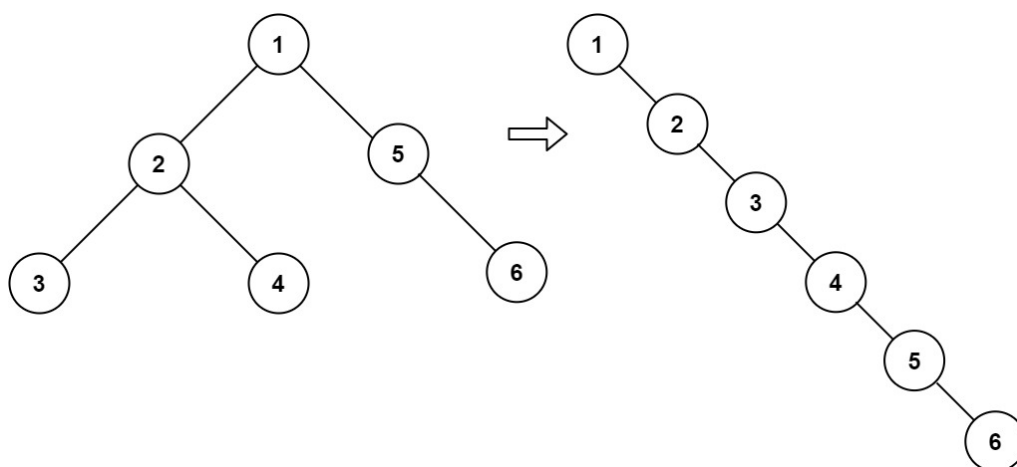
2.43 114 二叉树展开为链表

[114. 二叉树展开为链表](#)

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。
- 展开后的单链表应该与二叉树 [先序遍历](#) 顺序相同。

示例 1：



```

1 输入：root = [1,2,5,3,4,null,6]
2 输出：[1,null,2,null,3,null,4,null,5,null,6]

```

示例 2：

```

1 输入：root = []
2 输出：[]

```

示例 3：

```
1 输入: root = [0]
2 输出: [0]
```

题解:

很帅，中心思想就是把当前节点的右节点接到左子树最右边的节点的右节点后面

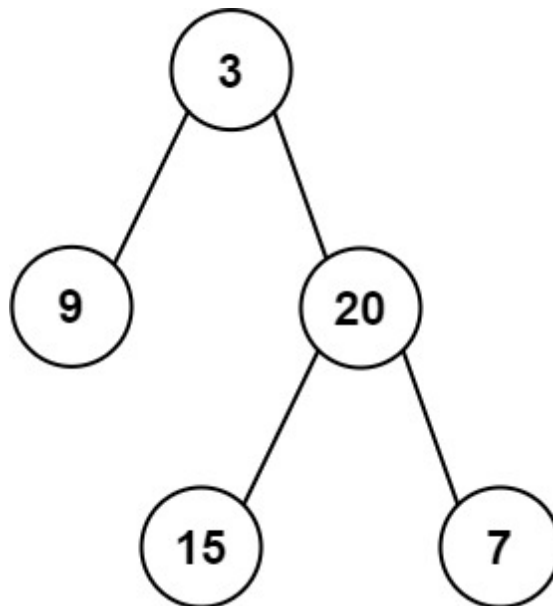
```
1 void flatten(TreeNode* root) {
2     TreeNode *cur = root;
3     while (cur != nullptr) {
4         if (cur->left != nullptr) {
5             TreeNode *save = cur->left;
6             TreeNode *pre = save;
7             while (pre->right != nullptr)
8                 pre = pre->right;
9             pre->right = cur->right;
10            cur->left = nullptr;
11            cur->right = save;
12        }
13        cur = cur->right;
14    }
15 }
```

2.44 105 从前序与中序遍历序列构造二叉树

[105. 从前序与中序遍历序列构造二叉树](#)

给定两个整数数组 `preorder` 和 `inorder`，其中 `preorder` 是二叉树的先序遍历，`inorder` 是同一棵树的中序遍历，请构造二叉树并返回其根节点。

示例 1:



```
1 输入: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
2 输出: [3,9,20,null,null,15,7]
```

示例 2:

```
1 输入: preorder = [-1], inorder = [-1]
2 输出: [-1]
```


题解:

老题了，好好看看

```
1  TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
2      TreeNode *root = buildTree(preorder, 0, preorder.size()-1, inorder, 0,
inorder.size()-1);
3      return root;
4  }
5  TreeNode *buildTree(vector<int> &preorder, int l1, int r1, vector<int> &inorder, int
l2, int r2) {
6      if (l1 > r1 || l2 > r2)
7          return nullptr;
8      TreeNode *root = new TreeNode(preorder[l1]);
9      int index = l2;
10     while (index <= r2) {
11         if (preorder[l1] == inorder[index])
12             break;
13         ++index;
14     }
15     int nodes = index - l2;
16     root->left = buildTree(preorder, l1 + 1, l1 + nodes, inorder, l2, index - 1);
17     root->right = buildTree(preorder, l1 + nodes + 1, r1, inorder, index + 1, r2);
18     return root;
19 }
```

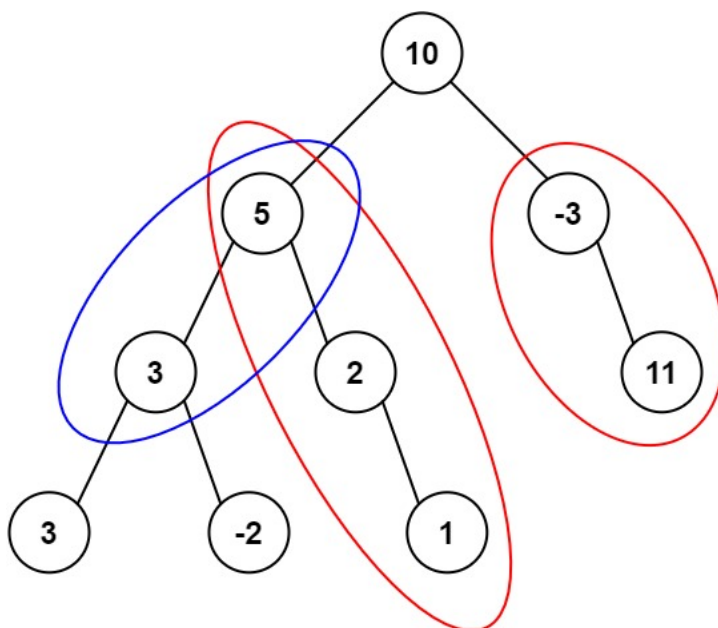
2.45 437 路径总和

[437. 路径总和 III](#)

给定一个二叉树的根节点 `root`，和一个整数 `targetSum`，求该二叉树里节点值之和等于 `targetSum` 的路径的数目。

路径 不需要从根节点开始，也不需要叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

示例 1:



1 输入: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
2 输出: 3
3 解释: 和等于 8 的路径有 3 条, 如图所示。

示例 2:

1 输入: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
2 输出: 3

题解:

第二次做这个题了, 第一次是在牛客的剑指Offer, 主要就用到了map保存前缀和来避免多余的重复计算:

```
1 unordered_map<long long, int> map;  
2 int pathSum(TreeNode* root, int targetSum) {  
3     map[0] = 1;  
4     return findPath(root, targetSum, 0);  
5 }  
6 int findPath(TreeNode *root, int targetSum, long long last)  
7 {  
8     if (root == nullptr)  
9         return 0;  
10    int res = 0;  
11    last += root->val;  
12    if (map.find(last - targetSum) != map.end()) {  
13        res = map[last - targetSum];  
14    }  
15    ++map[last];  
16    res += findPath(root->left, targetSum, last);  
17    res += findPath(root->right, targetSum, last);  
18    --map[last];  
19    return res;  
20 }
```

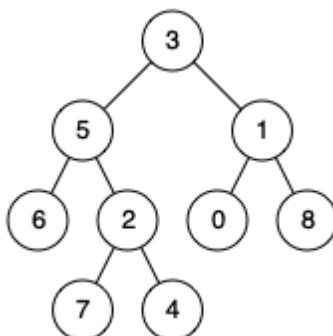
2.46 236 二叉树的最近公共祖先

[236. 二叉树的最近公共祖先](#)

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。

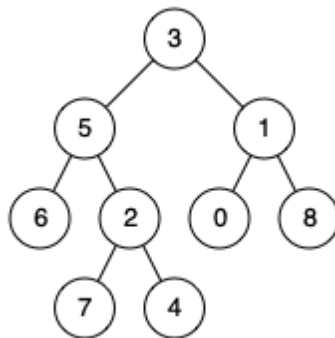
[百度百科](#)中最近公共祖先的定义为: “对于有根树 T 的两个节点 p、q, 最近公共祖先表示为一个节点 x, 满足 x 是 p、q 的祖先且 x 的深度尽可能大 (一个节点也可以是它自己的祖先)。”

示例 1:



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

示例 2:



1 输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
2 输出: 5
3 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。

示例 3:

1 输入: root = [1,2], p = 1, q = 2
2 输出: 1

题解:

好几次做这个题了，重点就是二叉树dp，向左右子树要信息（有没有p或者q）

```
1  TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
2      if (root == nullptr || root == p || root == q)
3          return root;
4      TreeNode *leftdata = lowestCommonAncestor(root->left, p, q);
5      TreeNode *rightdata = lowestCommonAncestor(root->right, p, q);
6      if (leftdata && rightdata)
7          return root;
8      return leftdata ? leftdata : rightdata;
9  }
```

2.47 124 二叉树中的最大路径和

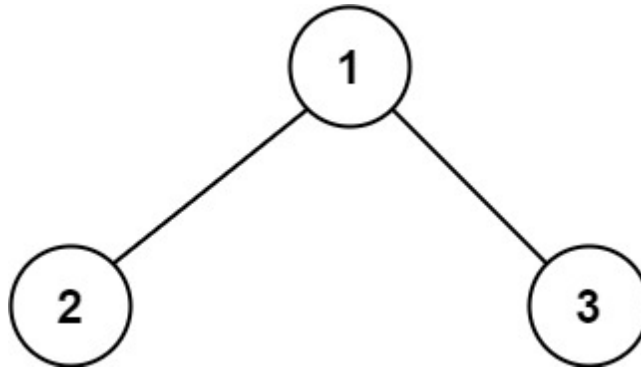
[124. 二叉树中的最大路径和](#)

二叉树中的 **路径** 被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。同一个节点在一条路径序列中 **至多出现一次**。该路径 **至少包含一个** 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

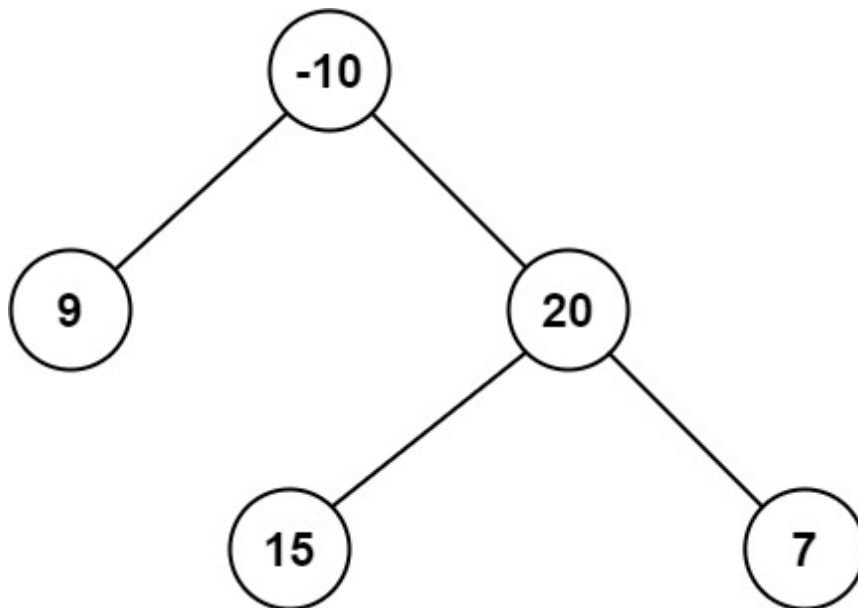
给你一个二叉树的根节点 `root`，返回其 **最大路径和**。

示例 1:



1 输入: root = [1,2,3]
2 输出: 6
3 解释: 最优路径是 2 -> 1 -> 3 , 路径和为 2 + 1 + 3 = 6

示例 2:



1 输入: root = [-10,9,20,null,null,15,7]
2 输出: 42
3 解释: 最优路径是 15 -> 20 -> 7 , 路径和为 15 + 20 + 7 = 42

题解:

其实还是树形DP，向左右子树要信息（左右能提供的最大和），然后加上自身，得到经过当前节点的一条最大路径和，维护一下maxSum

```
1  int maxSum = INT_MIN;
2  int maxPathSum(TreeNode* root) {
3      help(root);
4      return maxSum;
5  }
6  int help(TreeNode *root) {
7      if (root == nullptr)
8          return 0;
9      int leftData = max(help(root->left), 0);
10     int rightData = max(help(root->right), 0);
11
12     int temp = root->val + leftData + rightData;
13     maxSum = max(temp, maxSum);
```

```

14 |
15 |         return root->val + max(leftData, rightData);
16 |     }

```

2.48 200 岛屿数量

[200. 岛屿数量](#)

给你一个由 `'1'`（陆地）和 `'0'`（水）组成的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1:

```

1 | 输入: grid = [
2 |   ["1","1","1","1","0"],
3 |   ["1","1","0","1","0"],
4 |   ["1","1","0","0","0"],
5 |   ["0","0","0","0","0"]
6 | ]
7 | 输出: 1

```

示例 2:

```

1 | 输入: grid = [
2 |   ["1","1","0","0","0"],
3 |   ["1","1","0","0","0"],
4 |   ["0","0","1","0","0"],
5 |   ["0","0","0","1","1"]
6 | ]
7 | 输出: 3

```

题解:

看了题解区大佬的精华题解，妙啊，学到了，图的dfs套路和二叉树的挺像

```

1 |     void dfs(vector<vector<char>> &grid, int x, int y) {
2 |         if (!inGrid(grid, x, y)) //base case
3 |             return;
4 |         if (grid[x][y] == '1') { //递归
5 |             grid[x][y] = '2';
6 |             dfs(grid, x - 1, y);
7 |             dfs(grid, x + 1, y);
8 |             dfs(grid, x, y - 1);
9 |             dfs(grid, x, y + 1);
10 |        }
11 |    }
12 |    int numIslands(vector<vector<char>>& grid) {
13 |        int ans = 0;
14 |        int n = grid.size();
15 |        int m = grid[0].size();
16 |        for (int i = 0; i < n; ++i) {
17 |            for (int j = 0; j < m; ++j) {
18 |                if (grid[i][j] == '1') {
19 |                    ++ans;
20 |                    dfs(grid, i, j);

```

```

21         }
22     }
23 }
24 return ans;
25 }
26 bool inGrid(vector<vector<char>> &grid, int x, int y) {
27     int m = grid.size();
28     int n = grid[0].size();
29     if (x < 0 || y < 0 || x >= m || y >= n)
30         return false;
31     return true;
32 }

```

2.49 994 腐烂的橘子

994. 腐烂的橘子

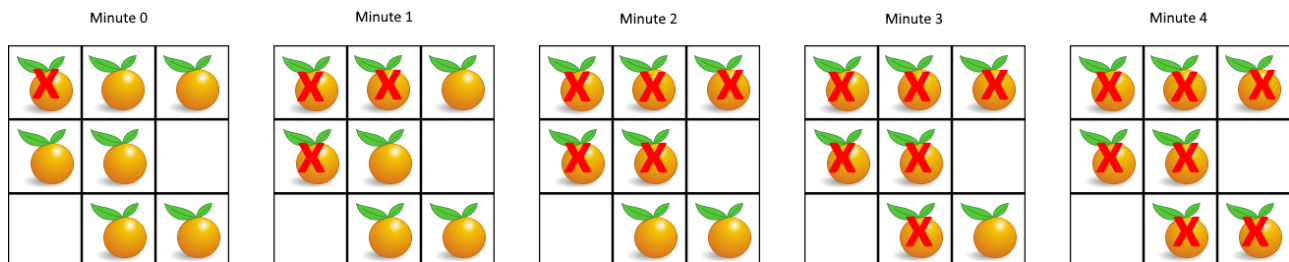
在给定的 $m \times n$ 网格 `grid` 中，每个单元格可以有以下三个值之一：

- 值 `0` 代表空单元格；
- 值 `1` 代表新鲜橘子；
- 值 `2` 代表腐烂的橘子。

每分钟，腐烂的橘子 **周围 4 个方向上相邻** 的新鲜橘子都会腐烂。

返回 直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 `-1`。

示例 1：



```

1 输入: grid = [[2,1,1],[1,1,0],[0,1,1]]
2 输出: 4

```

示例 2：

```

1 输入: grid = [[2,1,1],[0,1,1],[1,0,1]]
2 输出: -1
3 解释: 左下角的橘子 (第 2 行, 第 0 列) 永远不会腐烂, 因为腐烂只会发生在 4 个方向上。

```

示例 3：

```

1 输入: grid = [[0,2]]
2 输出: 0
3 解释: 因为 0 分钟时已经没有新鲜橘子了, 所以答案就是 0。

```

题解：

大概意思就是广度优先搜索，首先用一个辅助数组记录一下每个橘子腐烂时间，一开始把所有设置为-1，然后遍历一遍矩阵，如果遇到腐烂的橘子就把相应位置的腐烂时间设置为0（这个才是初始值，其实用-1为了区分腐烂区和新鲜区），并将该位置加入队列（用于广度优先搜索），最后队列非空就一直搜索下去，只要记录的新鲜橘子数大于0则返回-1

```
1   int badTime[10][10];
2   int cnt = 0;
3   int dir_x[4] = {0, 0, 1, -1}; //行
4   int dir_y[4] = {1, -1, 0, 0}; //列
5
6   int orangesRotting(vector<vector<int>>& grid) {
7       int m = grid.size(); //行
8       int n = grid[0].size(); //列
9       memset(badTime, -1, sizeof(badTime));
10      queue<pair<int, int>> badOranges;
11
12      for (int i = 0; i < m; ++i) {
13          for (int j = 0; j < n; ++j) {
14              if (grid[i][j] == 2) {
15                  badOranges.emplace(i, j);
16                  badTime[i][j] = 0;
17              }
18              else if (grid[i][j] == 1) {
19                  ++cnt;
20              }
21          }
22      }
23      int ans = 0;
24      while (!badOranges.empty()) {
25          auto [row, col] = badOranges.front();
26          badOranges.pop();
27
28          for (int i = 0; i < 4; ++i) {
29              int dx = row + dir_x[i];
30              int dy = col + dir_y[i];
31
32              if (dx < 0 || dy < 0 || dx >= m || dy >= n ||
33                  badTime[dx][dy] != -1 || grid[dx][dy] == 0)
34                  continue;
35              badTime[dx][dy] = badTime[row][col] + 1;
36              badOranges.emplace(dx, dy);
37              if (grid[dx][dy] == 1) {
38                  --cnt;
39                  ans = badTime[dx][dy];
40                  if (cnt == 0) break;
41              }
42          }
43      }
44      return cnt == 0 ? ans : -1;
45  }
```

2.50 46 全排列

[46. 全排列](#)

给定一个不含重复数字的数组 `nums`，返回其 所有可能的全排列。你可以 **按任意顺序** 返回答案。

示例 1:

```
1 输入: nums = [1,2,3]
2 输出: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

示例 2:

```
1 输入: nums = [0,1]
2 输出: [[0,1],[1,0]]
```

示例 3:

```
1 输入: nums = [1]
2 输出: [[1]]
```

题解:

递归回溯

```
1  vector<vector<int>> permute(vector<int>& nums) {
2      vector<vector<int>> ans;
3      if (nums.size() == 0)
4          return ans;
5      sort(nums.begin(), nums.end());
6      findAns(nums, ans, 0);
7      return ans;
8  }
9
10 void findAns(vector<int> &nums, vector<vector<int>> &ans, int index) {
11     if (index == nums.size())
12         ans.push_back(nums);
13     for (int i = index; i < nums.size(); ++i) {
14         swap(nums[i], nums[index]);
15         findAns(nums, ans, index + 1);
16         swap(nums[i], nums[index]);
17     }
18 }
```

2.51 35 搜索插入位置

[35. 搜索插入位置](#)

已解答

简单

相关标签

相关企业

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请务必使用时间复杂度为 $O(\log n)$ 的算法。

示例 1:

```
1 输入: nums = [1,3,5,6], target = 5
2 输出: 2
```

示例 2:

```
1 输入: nums = [1,3,5,6], target = 2
2 输出: 1
```

示例 3:

```
1 输入: nums = [1,3,5,6], target = 7
2 输出: 4
```

题解

主要就是二分法，得好好练练，这道题实际上也是可以转换为二分的

```
1  int searchInsert(vector<int>& nums, int target) {
2      int n = nums.size();
3      int l = 0, r = n - 1, ans = n;
4      while (l <= r) {
5          int mid = ((r - l) >> 1) + 1;
6          if (target <= nums[mid]) {
7              ans = mid;
8              r = mid - 1;
9          }
10         else
11             l = mid + 1;
12     }
13
14     return ans;
15 }
```

2.52 74 搜索二维矩阵

[74. 搜索二维矩阵](#)

给你一个满足下述两条属性的 $m \times n$ 整数矩阵：

- 每行中的整数从左到右按非严格递增顺序排列。
- 每行的第一个整数大于前一行的最后一个整数。

给你一个整数 `target`，如果 `target` 在矩阵中，返回 `true`；否则，返回 `false`。

输入: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

输出: true

输入: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

输出: false

题解:

可以把二维矩阵展开成一维矩阵，只需要管理好映射关系即可

```
1 | bool searchMatrix(vector<vector<int>>& matrix, int target) {
2 |     if (matrix.size() == 0)
3 |         return false;
4 |     int m = matrix.size(), n = matrix[0].size();
5 |     int low = 0, high = m * n - 1;
6 |     while (low <= high) {
7 |         int mid = ((high - low) >> 1) + low;
8 |         int x = matrix[mid / n][mid % n];
9 |         if (x < target)
10 |             low = mid + 1;
11 |         else if (x > target)
12 |             high = mid - 1;
13 |         else
14 |             return true;
15 |     }
16 |     return false;
17 | }
```

2.53 78 子集

[78. 子集](#)

给你一个整数数组 `nums`，数组中的元素 **互不相同**。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1:

```
1 | 输入: nums = [1,2,3]
2 | 输出: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

示例 2:

```
1 | 输入: nums = [0]
2 | 输出: [[],[0]]
```

题解:

有点暴力哈，枚举，用每一位是否在当前的自己来枚举

```
1 | vector<int> t;
2 | vector<vector<int>> ans;
3 | vector<vector<int>> subsets(vector<int>& nums) {
4 |     int n = nums.size();
5 |     for (int mask = 0; mask < (1 << n); ++mask) {
6 |         t.clear();
7 |         for (int i = 0; i < n; ++i) {
8 |             if (mask & (1 << i))
9 |                 t.push_back(nums[i]);
```

```

10         }
11         ans.push_back(t);
12     }
13
14     return ans;
15 }

```

2.54 17 电话号码的字母组合

[17. 电话号码的字母组合](#)

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



示例 1:

```

1 输入: digits = "23"
2 输出: ["ad","ae","af","bd","be","bf","cd","ce","cf"]

```

示例 2:

```

1 输入: digits = ""
2 输出: []

```

示例 3:

```

1 输入: digits = "2"
2 输出: ["a","b","c"]

```

题解:

回溯相关的知识点还是太欠缺了啊，还得好好练练

```

1     vector<string> letterCombinations(string digits) {
2         vector<string> combinations;
3         if (digits.empty())
4             return combinations;
5         unordered_map<char, string> phoneMap {
6             {'2', "abc"},
7             {'3', "def"},
8             {'4', "ghi"},
9             {'5', "jkl"},

```

```

10         {'6', "mno"},
11         {'7', "pqrs"},
12         {'8', "tuv"},
13         {'9', "wxyz"}
14     };
15
16     string combination;
17     backtrack(combinations, phoneMap, digits, 0, combination);
18     return combinations;
19 }
20
21 void backtrack(vector<string> &combinations, unordered_map<char, string> &phoneMap,
string &digits, int index, string &combination) {
22     if (index == digits.size())
23         combinations.push_back(combination);
24     else {
25         char digit = digits[index];
26         string &letters = phoneMap.at(digit);
27         for (const char &letter : letters) {
28             combination.push_back(letter);
29             backtrack(combinations, phoneMap, digits, index + 1, combination);
30             combination.pop_back();
31         }
32     }
33 }

```

2.55 39 组合总和

39. 组合总和

给你一个 **无重复元素** 的整数数组 `candidates` 和一个目标整数 `target`，找出 `candidates` 中可以使数字和为目标数 `target` 的所有 **不同组合**，并以列表形式返回。你可以按 **任意顺序** 返回这些组合。

`candidates` 中的 **同一个** 数字可以 **无限制重复被选取**。如果至少一个数字的被选数量不同，则两种组合是不同的。

对于给定的输入，保证和为 `target` 的不同组合数少于 `150` 个。

示例 1:

```

1 输入: candidates = [2,3,6,7], target = 7
2 输出: [[2,2,3],[7]]
3 解释:
4 2 和 3 可以形成一组候选, 2 + 2 + 3 = 7 。注意 2 可以使用多次。
5 7 也是一个候选, 7 = 7 。
6 仅有这两种组合。

```

示例 2:

```

1 输入: candidates = [2,3,5], target = 8
2 输出: [[2,2,2,2],[2,3,3],[3,5]]

```

示例 3:

```
1 输入: candidates = [2], target = 1
2 输出: []
```

题解:

依旧是回溯法, 不过又来了个什么剪枝? 得补习补习了

```
1  vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
2      vector<vector<int>> ans;
3      vector<int> combine;
4      dfs(candidates, target, ans, combine, 0);
5      return ans;
6  }
7
8  void dfs(vector<int> &candidates, int target, vector<vector<int>> &ans, vector<int>
&combine, int index) {
9      if (index == candidates.size())
10         return;
11     if (target == 0) {
12         ans.emplace_back(combine);
13         return;
14     }
15
16     dfs(candidates, target, ans, combine, index + 1);
17
18     if (target - candidates[index] >= 0) {
19         combine.emplace_back(candidates[index]);
20         dfs(candidates, target - candidates[index], ans, combine, index);
21         combine.pop_back();
22     }
23 }
```

2.56 22 括号生成

[22. 括号生成](#)

数字 `n` 代表生成括号的对数, 请你设计一个函数, 用于能够生成所有可能的并且 **有效的** 括号组合。

示例 1:

```
1 输入: n = 3
2 输出: ["((()))", "(()())", "(())()", "()(())", "()()()"]
```

示例 2:

```
1 输入: n = 1
2 输出: ["()"]
```

题解:

回溯真的是硬伤啊!!! 🤯

```
1  vector<string> generateParenthesis(int n) {
2      vector<string> result;
3      string current;
```

```

4         backtrack(result, current, 0, 0, n);
5         return result;
6     }
7
8     void backtrack(vector<string> &ans, string &cur, int open, int close, int n) {
9         if (cur.size() == n * 2) {
10             ans.push_back(cur);
11             return;
12         }
13         if (open < n) {
14             cur.push_back('(');
15             backtrack(ans, cur, open + 1, close, n);
16             cur.pop_back();
17         }
18         if (close < open) {
19             cur.push_back(')');
20             backtrack(ans, cur, open, close + 1, n);
21             cur.pop_back();
22         }
23     }

```

2.57 79 单词搜索

[79. 单词搜索](#)

给定一个 `m x n` 二维字符网格 `board` 和一个字符串单词 `word`。如果 `word` 存在于网格中，返回 `true`；否则，返回 `false`。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1:

A	B	C	E
S	F	C	S
A	D	E	E

```

1  输入: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
2  输出: true

```

示例 2:

A	B	C	E
S	F	C	S
A	D	E	E

```

1  输入: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
2  输出: true

```

示例 3:

A	B	C	E
S	F	C	S
A	D	E	E

```

1  输入: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
2  输出: false

```

题解:

用了牛客剑指Offer的一道类似的题的思路, 做是做出来了, 不过击败10%, 太捞了, 第一版一个普通递归回溯

```

1      bool exist(vector<vector<char>>& board, string word) {
2          if (board.size() == 0)
3              return false;
4          int row = board.size();
5          int col = board[0].size();
6          vector<vector<bool>> flags(row, vector<bool>(col, false));
7          for (int i = 0; i < row; ++i) {
8              for (int j = 0; j < col; ++j) {
9                  if (exist(board, word, row, col, i, j, 0, flags))
10                     return true;
11              }
12          }
13          return false;
14      }
15
16      bool exist(vector<vector<char>>& board, string word, int row, int col, int i, int j,
17                  int k, vector<vector<bool>> &flags) {

```

```

17         if (i < 0 || j < 0 || i >= row || j >= col || (board[i][j] != word[k]) || flags[i]
[j] == true)
18             return false;
19         if (k == word.size() - 1)
20             return true;
21         flags[i][j] = true;
22         if (exist(board, word, row, col, i + 1, j, k + 1, flags) ||
23             exist(board, word, row, col, i - 1, j, k + 1, flags) ||
24             exist(board, word, row, col, i, j + 1, k + 1, flags) ||
25             exist(board, word, row, col, i, j - 1, k + 1, flags))
26             return true;
27         flags[i][j] = false;
28         return false;
29     }

```

2.58 分割回文串

[131. 分割回文串](#)

给你一个字符串 `s`，请你将 `s` 分割成一些子串，使每个子串都是回文串。返回 `s` 所有可能的分割方案。

示例 1:

```

1 输入: s = "aab"
2 输出: [["a","a","b"],["aa","b"]]

```

示例 2:

```

1 输入: s = "a"
2 输出: [["a"]]

```

题解:

差不多的，就是多了dfs的结合

```

1  vector<vector<int>> flags;
2  vector<vector<string>> ret;
3  vector<string> ans;
4  int n;
5  public:
6      vector<vector<string>> partition(string s) {
7          n = s.size();
8          flags.assign(n, vector<int>(n, true));
9
10         for (int i = n - 1; i >= 0; --i) {
11             for (int j = i + 1; j < n; ++j) {
12                 flags[i][j] = (s[i] == s[j]) && flags[i + 1][j - 1];
13             }
14         }
15     }

```



```

16     dfs(s, 0);
17     return ret;
18 }
19
20 void dfs(const string &s, int i) {
21     if (i == n) {
22         ret.push_back(ans);
23         return;
24     }
25     for (int j = i; j < n; ++j) {
26         if (flags[i][j]) {
27             ans.push_back(s.substr(i, j - i + 1));
28             dfs(s, j + 1);
29             ans.pop_back();
30         }
31     }
32 }

```

2.59 N皇后

[51. N 皇后](#)

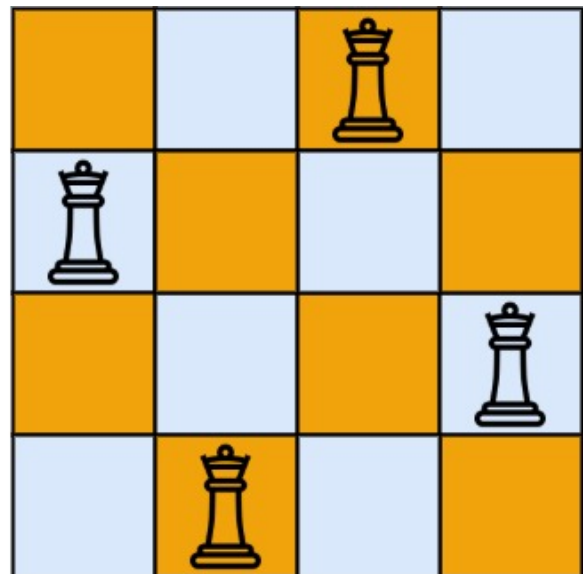
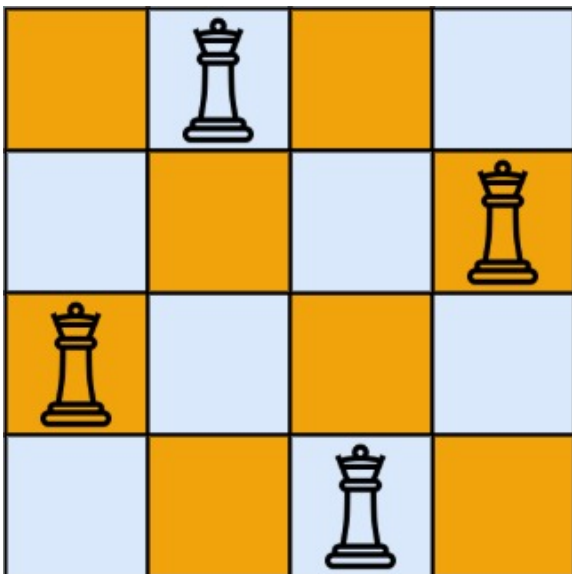
按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

n 皇后问题 研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n ，返回所有不同的 **n 皇后问题** 的解决方案。

每一种解法包含一个不同的 **n 皇后问题** 的棋子放置方案，该方案中 `'Q'` 和 `'.'` 分别代表了皇后和空位。

示例 1:



```

1  输入: n = 4
2  输出: [[".Q..", "...Q", "Q...", "...Q."], ["..Q.", "Q...", "...Q", ".Q.."]]
3  解释: 如上图所示, 4 皇后问题存在两个不同的解法。

```

示例 2:

1 输入: n = 1
2 输出: [["Q"]]

题解:

时隔这么久,我终于又有感觉了,一道难题,也是对回溯法的一个锻炼提升

```
1  vector<vector<string>> solveNQueens(int n) {
2      vector<vector<string>> ans;
3      //标记列、斜对角是否有皇后
4      unordered_set<int> posCol = unordered_set<int>();
5      unordered_set<int> pos1 = unordered_set<int>();
6      unordered_set<int> pos2 = unordered_set<int>();
7      //用下标对应行, 值对应列
8      vector<int> queens(n, -1);
9      //vector<string> ret;
10
11     backtrace(ans, posCol, pos1, pos2, queens, n, 0);
12
13     return ans;
14 }
15 void backtrace(vector<vector<string>> &ans, unordered_set<int> &posCol,
16 unordered_set<int> &pos1, unordered_set<int> &pos2, vector<int> &queens, int n, int row) {
17     if (row == n) {
18         vector<string> board = generate(queens, n);
19         ans.push_back(board);
20     }
21     else {
22         for (int i = 0; i < n; ++i) {
23             if (posCol.find(i) != posCol.end())
24                 continue;
25             int delta1 = row + i;
26             if (pos1.find(delta1) != pos1.end())
27                 continue;
28             int delta2 = row - i;
29             if (pos2.find(delta2) != pos2.end())
30                 continue;
31             queens[row] = i;
32             posCol.insert(i);
33             pos1.insert(delta1);
34             pos2.insert(delta2);
35             backtrace(ans, posCol, pos1, pos2, queens, n, row + 1);
36             queens[row] = -1;
37             posCol.erase(i);
38             pos1.erase(delta1);
39             pos2.erase(delta2);
40         }
41     }
42 }
43 vector<string> generate(vector<int> &queens, int n) {
44     vector<string> board = vector<string>();
45     for (int i = 0; i < n; ++i) {
46         string row(n, '.');
```

```

47         row[queens[i]] = 'Q';
48         board.push_back(row);
49     }
50     return board;
51 }

```

2.60 35 搜索插入位置

[35. 搜索插入位置](#)

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请必须使用时间复杂度为 $O(\log n)$ 的算法。

示例 1:

```

1  输入: nums = [1,3,5,6], target = 5
2  输出: 2

```

示例 2:

```

1  输入: nums = [1,3,5,6], target = 2
2  输出: 1

```

示例 3:

```

1  输入: nums = [1,3,5,6], target = 7
2  输出: 4

```

题解:

下面就开始二分咯

```

1  int searchInsert(vector<int>& nums, int target) {
2      int n = nums.size();
3      int l = 0, r = n - 1, ans = n;
4      while (l <= r) {
5          int mid = ((r - l) >> 1) + 1;
6          if (target <= nums[mid]) {
7              ans = mid;
8              r = mid - 1;
9          }
10         else
11             l = mid + 1;
12     }
13
14     return ans;
15 }

```

::感觉自己的思维真的不够打的开啊，局限性还存在

2.61 34 在排序数组中查找元素的第一个和最后一个位置

34. 在排序数组中查找元素的第一个和最后一个位置

给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

你必须设计并实现时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1:

```
1 输入: nums = [5,7,7,8,8,10], target = 8
2 输出: [3,4]
```

示例 2:

```
1 输入: nums = [5,7,7,8,8,10], target = 6
2 输出: [-1,-1]
```

示例 3:

```
1 输入: nums = [], target = 0
2 输出: [-1,-1]
```

题解:

二分法，巧妙利用一个flag区分了找左右边界的条件

```
1  vector<int> searchRange(vector<int>& nums, int target) {
2      int leftIndex = binarySearch(nums, target, true);
3      int rightIndex = binarySearch(nums, target, false) - 1;
4      if (leftIndex <= rightIndex && rightIndex < nums.size() && nums[leftIndex] ==
target && nums[rightIndex] == target)
5          return vector<int>{leftIndex, rightIndex};
6      return vector<int>{-1, -1};
7  }
8
9  int binarySearch(vector<int> &nums, int target, bool flag) {
10     int left = 0, right = nums.size() - 1, ans = nums.size();
11     while (left <= right) {
12         int mid = left + ((right - left) >> 1);
13         if (nums[mid] > target || (flag && nums[mid] >= target)) {
14             right = mid - 1;
15             ans = mid;
16         }
17         else {
18             left = mid + 1;
19         }
20     }
21     return ans;
22 }
```

2.62 33 搜索旋转排序数组

33. 搜索旋转排序数组

整数数组 `nums` 按升序排列，数组中的值 **互不相同**。

在传递给函数之前，`nums` 在预先未知的某个下标 `k` ($0 \leq k < \text{nums.length}$) 上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标从 0 开始计数）。例如，`[0,1,2,4,5,6,7]` 在下标 3 处经旋转后可能变为 `[4,5,6,7,0,1,2]`。

给你 **旋转后** 的数组 `nums` 和一个整数 `target`，如果 `nums` 中存在这个目标值 `target`，则返回它的下标，否则返回 `-1`。

你必须设计一个时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1:

```
1 输入: nums = [4,5,6,7,0,1,2], target = 0
2 输出: 4
```

示例 2:

```
1 输入: nums = [4,5,6,7,0,1,2], target = 3
2 输出: -1
```

示例 3:

```
1 输入: nums = [1], target = 0
2 输出: -1
```

题解:

二分查找，不过这里有很多细节，得注意下里面的判断条件：

```
1  int search(vector<int>& nums, int target) {
2      int left = 0, right = nums.size() - 1;
3      if (nums.size() == 1) {
4          return nums[0] == target ? 0 : -1;
5      }
6      //二分查找
7      while (left <= right) {
8          int mid = left + ((right - left) >> 1);
9          if (nums[mid] == target)
10             return mid;
11         if (nums[0] <= nums[mid]) {
12             if (nums[0] <= target && target < nums[mid])
13                 right = mid - 1;
14             else
15                 left = mid + 1;
16         }
17         else {
```

```

18         if (nums[mid] < target && target <= nums[nums.size() - 1]) //尤其是这里的
    条件, 好好想想
19             left = mid + 1;
20         else
21             right = mid - 1;
22     }
23 }
24 return -1;
25 }

```

2.63 153 寻找旋转排序数组中的最小值

[153. 寻找旋转排序数组中的最小值](#)

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 **旋转** 后，得到输入数组。例如，原数组 `nums = [0,1,2,4,5,6,7]` 在变化后可能得到：

- 若旋转 4 次，则可以得到 `[4,5,6,7,0,1,2]`
- 若旋转 7 次，则可以得到 `[0,1,2,4,5,6,7]`

注意，数组 `[a[0], a[1], a[2], ..., a[n-1]]` **旋转一次** 的结果为数组 `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`。

给你一个元素值 **互不相同** 的数组 `nums`，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 **最小元素**。

你必须设计一个时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1:

```

1  输入: nums = [3,4,5,1,2]
2  输出: 1
3  解释: 原数组为 [1,2,3,4,5]，旋转 3 次得到输入数组。

```

示例 2:

```

1  输入: nums = [4,5,6,7,0,1,2]
2  输出: 0
3  解释: 原数组为 [0,1,2,4,5,6,7]，旋转 3 次得到输入数组。

```

示例 3:

```

1  输入: nums = [11,13,15,17]
2  输出: 11
3  解释: 原数组为 [11,13,15,17]，旋转 4 次得到输入数组。

```

题解:

二分查找一个小变化，理解本质了并不复杂：

```

1  int findMin(vector<int>& nums) {
2      if (nums.size() == 0) {
3          return -1;

```

```

4         }
5         int l = 0, r = nums.size() - 1;
6         while (l < r) {
7             int mid = l + ((r - l) >> 1);
8             if (nums[mid] > nums[r])
9                 l = mid + 1;
10            else if (nums[mid] == nums[r])
11                --r;
12            else
13                r = mid;
14        }
15        return nums[l];
16    }

```

2.64 4 寻找两个正序数组的中位数

[4. 寻找两个正序数组的中位数](#)

给定两个大小分别为 `m` 和 `n` 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 **中位数**。

算法的时间复杂度应该为 $O(\log(m+n))$ 。

示例 1:

```

1  输入: nums1 = [1,3], nums2 = [2]
2  输出: 2.00000
3  解释: 合并数组 = [1,2,3] , 中位数 2

```

示例 2:

```

1  输入: nums1 = [1,2], nums2 = [3,4]
2  输出: 2.50000
3  解释: 合并数组 = [1,2,3,4] , 中位数 (2 + 3) / 2 = 2.5

```

题解:

这题都能二分我是真没想到，主要关键点就是理解在两个数组上划分范围，以及各种特殊情况

```

1      double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
2          if (nums1.size() > nums2.size())
3              return findMedianSortedArrays(nums2, nums1);
4          int m = nums1.size(), n = nums2.size();
5          int left = 0, right = m;
6
7          int preMin = 0, blackMax = 0;
8          while (left <= right) {
9              int mid1 = left + (right - left) / 2;
10             int mid2 = (m + n + 1) / 2 - mid1;
11
12             int mid1_left = (mid1 == 0 ? INT_MIN : nums1[mid1 - 1]);
13             int mid1_right = (mid1 == m ? INT_MAX : nums1[mid1]);

```

```

14         int mid2_left = (mid2 == 0 ? INT_MIN : nums2[mid2 - 1]);
15         int mid2_right = (mid2 == n ? INT_MAX : nums2[mid2]);
16
17         if (mid1_left <= mid2_right) {
18             preMin = max(mid1_left, mid2_left);
19             blackMax = min(mid1_right, mid2_right);
20             left = mid1 + 1;
21         }
22         else
23             right = mid1 - 1;
24     }
25
26     return (m + n) % 2 == 0 ? (preMin + blackMax) / 2.0 : preMin;
27 }

```

2.65 394

[394. 字符串解码](#)

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: `k[encoded_string]`，表示其中方括号内部的 `encoded_string` 正好重复 `k` 次。注意 `k` 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 `k`，例如不会出现像 `3a` 或 `2[4]` 的输入。

示例 1:

```

1 输入: s = "3[a]2[bc]"
2 输出: "aaabcbc"

```

示例 2:

```

1 输入: s = "3[a2[c]]"
2 输出: "accaccacc"

```

示例 3:

```

1 输入: s = "2[abc]3[cd]ef"
2 输出: "abccbccdcdcdcd"

```

示例 4:

```

1 输入: s = "abc3[cd]xyz"
2 输出: "abcccdcdcdxyz"

```

题解:

注意在栈中获取字符串时，不能简单粗暴的直接把字符拿到，因为每个字符串中的字母顺序要保证好，直接拿的话后面在reverse的时候会被打乱

```

1 class Solution {

```



```

2 public:
3     string getDigits(string &s, int &index) {
4         string ret = "";
5         while (isdigit(s[index])) {
6             ret += s[index++];
7         }
8         return ret;
9     }
10
11     string conString(string s, int num) {
12         string ret = "";
13         while (num-->0) {
14             ret += s;
15         }
16         return ret;
17     }
18     string getString(vector<string> &str) {
19         string ret = "";
20         for (const string &s : str) {
21             ret += s;
22         }
23         return ret;
24     }
25
26     string decodeString(string s) {
27         vector<string> stk; //模拟栈
28         string ret = "";
29         int index = 0;
30         while (index < s.size()) {
31             char cur = s[index];
32             if (isdigit(cur)) {
33                 string digits = getDigits(s, index);
34                 stk.push_back(digits);
35             }
36             else if (isalpha(cur) || cur == '[') {
37                 stk.push_back(string(1, s[index++]));
38             }
39             else {
40                 ++index;
41                 vector<string> sub_str;
42                 while (stk.back() != "[") {
43                     sub_str.push_back(stk.back());
44                     stk.pop_back();
45                 }
46                 reverse(sub_str.begin(), sub_str.end());
47                 string t = getString(sub_str);
48                 stk.pop_back();
49                 int times = stoi(stk.back());
50                 stk.pop_back();
51                 string res;
52                 res = conString(t, times);
53                 stk.push_back(res);
54             }
55         }
56         for (const string &s : stk) {
57             ret += s;

```

```

58     }
59     return ret;
60 }
61 };

```

2.66 739 每日温度

[739. 每日温度](#)

给定一个整数数组 `temperatures`，表示每天的温度，返回一个数组 `answer`，其中 `answer[i]` 是指对于第 `i` 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

示例 1:

```

1 输入: temperatures = [73,74,75,71,69,72,76,73]
2 输出: [1,1,4,2,1,1,0,0]

```

示例 2:

```

1 输入: temperatures = [30,40,50,60]
2 输出: [1,1,1,0]

```

示例 3:

```

1 输入: temperatures = [30,60,90]
2 输出: [1,1,0]

```

题解:

用单调栈，栈存放下标，只要下标在栈内说明该下标的温度还没有找到比它大的下一温度

```

1  vector<int> dailyTemperatures(vector<int>& temperatures) {
2      int n = temperatures.size();
3      vector<int> ans(n, 0);
4      stack<int> stk;
5      if (n < 2) {
6          return ans;
7      }
8      for (int i = 0; i < n; ++i) {
9          while (!stk.empty() && temperatures[i] > temperatures[stk.top()]) {
10             int index = stk.top();
11             ans[index] = i - index;
12             stk.pop();
13         }
14         stk.push(i);
15     }
16     return ans;
17 }

```

2.67 198 打家劫舍

[198. 打家劫舍](#)

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下**，一夜之内能够偷窃到的最高金额。

示例 1:

```
1 输入: [1,2,3,1]
2 输出: 4
3 解释: 偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3)。
4      偷窃到的最高金额 = 1 + 3 = 4 。
```

示例 2:

```
1 输入: [2,7,9,3,1]
2 输出: 12
3 解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。
4      偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

题解:

动态规划，主要就是观察状态转换方程，找到了就很好求解了

```
1  // 状态转换方程为 dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])
2  // 注意边界条件即可
3  class Solution {
4  public:
5      int rob(vector<int>& nums) {
6          if (nums.empty()) {
7              return 0;
8          }
9          int size = nums.size();
10         if (size == 1) {
11             return nums[0];
12         }
13         int first = nums[0], second = max(nums[1], nums[0]);    // 边界条件
14         for (int i = 2; i < size; ++i) {
15             int temp = second;
16             second = max(first + nums[i], second);
17             first = temp;
18         }
19         return second;
20     }
21 };
```

2.68 279 完全平方数

[279. 完全平方数](#)

给你一个整数 n ，返回和为 n 的完全平方数的最少数量。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如， 1 、 4 、 9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1:

```
1 输入: n = 12
2 输出: 3
3 解释: 12 = 4 + 4 + 4
```

示例 2:

```
1 输入: n = 13
2 输出: 2
3 解释: 13 = 4 + 9
```

题解:

动态规划版: 找到其状态方程, 主要就是理解 $F[n]$ 等于啥

```
1 class Solution {
2 public:
3     int numSquares(int n) {
4         vector<int> f(n + 1, 0);
5         for (int i = 1; i <= n; ++i) {
6             int minn = INT_MAX;
7             for (int j = 1; j * j <= i; ++j) {
8                 minn = min(minn, f[i - j * j]);
9             }
10            f[i] = minn + 1;
11        }
12        return f[n];
13    }
14};
```

数学版: 多的不说, 看官方题解就OK了

```
1 class Solution {
2 public:
3     bool isPerfectSquare(int n) {
4         int x = sqrt(n);
5         return x * x == n;
6     }
7
8     bool checkAnswer4(int n) {
9         while (n % 4 == 0) {
10            n /= 4;
11        }
12        return n % 8 == 7;
13    }
14
15    int numSquares(int n) {
16        if (isPerfectSquare(n)) {
17            return 1;
18        }
19        if (checkAnswer4(n)) {
20            return 4;
21        }
22        for (int i = 1; i * i <= n; ++i) {
23            int j = n - i * i;
24            if (isPerfectSquare(j)) {
```

```

25         return 2;
26     }
27 }
28     return 3;
29 }
30 };

```

2.69 322 零钱兑换

[322. 零钱兑换](#)

给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。

计算并返回可以凑成总金额所需的 **最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

你可以认为每种硬币的数量是无限的。

示例 1:

```

1 输入: coins = [1, 2, 5], amount = 11
2 输出: 3
3 解释: 11 = 5 + 5 + 1

```

示例 2:

```

1 输入: coins = [2], amount = 3
2 输出: -1

```

示例 3:

```

1 输入: coins = [1], amount = 0
2 输出: 0

```

题解:

这道题和上面的完全平方数简直一模一样，只需要找到状态转移方程即可

```

1  class Solution {
2  public:
3      int coinChange(vector<int>& coins, int amount) {
4          int maxNum = amount + 1;
5          vector<int> f(maxNum, maxNum);
6          f[0] = 0;
7          for (int i = 1; i <= amount; ++i) {
8              for (int j = 0; j < coins.size(); ++j) {
9                  if (coins[j] <= i) {
10                     f[i] = min(f[i], f[i - coins[j]] + 1);
11                 }
12             }
13         }
14         return f[amount] > amount ? -1 : f[amount];
15     }
16 };

```