剑指Offer总结

数组与矩阵

剑指 Offer 03. 数组中重复的数字

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

```
输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3
```

评价:

一看还以为送分题，结果题解一看才知道是要求候选人先问清楚边界条件的，如果要求时间优先就用哈希表计数，如果要求原地处理就是鸽巢理论，即遍历元素，将元素按其元素值x放入对应的下标x中(不用担心越界，因为规定元素范围为 0 ~ n - 1)，如果发现对应下标x已有相同元素，说明找到重复元素。

鸽巢法代码如下：

```
int findRepeatNumber(vector<int>& nums) {
        int i = 0;
        while(i < nums.size()){
            if(nums[i] == i){
                i++;
                continue;
            }
            if(nums[nums[i]] == nums[i])
                return nums[i];
            swap(nums[i], nums[nums[i]]);
        }

        return -1;
    }
```

这题还有个范围改为 1 - n ， 长度为n + 1的变式，要求原地且不允许修改数组， 枯了，希望面试官别问到这个。

剑指 Offer 04. 二维数组中的查找

在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

示例:

现有矩阵 matrix 如下:

```
[
  [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]
]
```

给定 target = 5，返回 true。

给定 target = 20，返回 false。

限制:

0 <= n <= 1000

0 <= m <= 1000

思路:

以右上角为根节点可以把数组看成一个BST, 可以以此来模拟查找。

```
class Solution {
public:
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
        if(matrix.size() == 0)
            return false;
        int row = 0, col = matrix[0].size() - 1;
        while(row < matrix.size() && col >= 0){
            if(matrix[row][col] < target){
                row++;
            }
            else if(matrix[row][col] > target){
                col--;
            }
            else{
                return true;
            }
        }

        return false;
    }
};
```

剑指 Offer 05. 替换空格

请实现一个函数，把字符串 s 中的每个空格替换成"%20"。

示例 1：

输入：s = "We are happy."
输出："We%20are%20happy."

思路：先确定所有空格被替换后新字符串的长度，然后用快慢指针分别对应老字符串闭尾和新字符串闭尾，逆序遍历构造新字符（为什么不顺序的原因是防止新字符串覆盖老字符串还未用于构造新字符串的对应部分）

```
class Solution {
public:
    string replaceSpace(string s) {
        int cnt = 0;
        for(int i = 0; i < s.size(); i++){
            if(s[i] == ' ')
                cnt++;
        }
        int oldSize = s.size();
        s.resize(s.size() + cnt * 2);
        int newSize = s.size();
        for(int slow = oldSize - 1, fast = newSize - 1; slow >= 0; slow--){
            if(s[slow] != ' '){
                s[fast--] = s[slow];
            }
            else{
                s[fast] = '0';
                s[fast - 1] = '2';
                s[fast - 2] = '%';
                fast -= 3;
            }
        }
        return s;
    }
};
```

剑指 Offer 29. 顺时针打印矩阵

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

示例 1：

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

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

限制：

0 <= matrix.length <= 100
0 <= matrix[i].length <= 100

思路：

确定四个边界top,left,right,bottom, 让指针在[left,right]和[top,bottom]内遍历，遍历完一个方向就收缩边界，直到边界相撞（对于闭区间倒不如说是相交）后终止。

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if(matrix.size() == 0)
            return {};
        int m = matrix.size(), n = matrix[0].size();
        int top = 0, left = 0, right = n - 1, bottom = m - 1;
        int cnt = 1;
        vector<int> res;
        while(top <= bottom && left <= right){
            for(int j = left; j <= right; j++)
                res.push_back(matrix[top][j]);
            top++;
            if(top > bottom)
                break;
            for(int i = top; i <= bottom; i++)
                res.push_back(matrix[i][right]);
            right--;
            if(left > right)
                break;
            for(int j = right; j >= left; j--)
                res.push_back(matrix[bottom][j]);
            bottom--;
            if(top > bottom)
                break;
            for(int i = bottom; i >= top; i--)
                res.push_back(matrix[i][left]);
            left++;
            if(left > right)
                break;
        }

        return res;
    }
```

剑指 Offer 39. 数组中出现次数超过一半的数字

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

输入: [1, 2, 3, 2, 2, 2, 5, 4, 2]
输出: 2

解:

这题就是求众数，可以直接用哈希表记录各元素数量找出目标数，时间和空间复杂度都是O(N)，显然不是最优解法。

以下引入一种可以实现原地的解法：摩尔投票法，其思想是不同值的元素可以一一抵消，遍历完最后剩下的最多的元素便是目标元素。

```
int majorityElement(vector<int>& nums) {
        int mode = nums[0];
        int cnt = 1;
        for(int i = 1; i < nums.size(); i++){
            if(cnt == 0){
                mode = nums[i];
                cnt = 1;
                continue;
            }
            if(nums[i] == mode)
                cnt++;
            else{
                cnt--;
                if(cnt == 0){
                    mode = nums[i];
                }
            }
        }
        return mode;
    }
```

剑指 Offer 61. 扑克牌中的顺子

从**若干副扑克牌**中随机抽 `5` 张牌，判断是不是一个顺子，即这5张牌是不是连续的。2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为 0 ，可以看成任意数字。A 不能视为 14。

示例 1:

```
输入: [1,2,3,4,5]
输出: True
```

示例 2:

```
输入: [0,0,1,2,5]
输出: True
```

限制:

数组长度为 5

数组的数取值为 [0, 13] .

思路:

注意审题:

数组长度限定为5，顺子即5个连续的数字 -> 数字间必须连续（不管递增递减），且出现2 - 10这些数字本身出现重复的情况直接判错

大小王为0且可以是任意数 -> 遍历遇到0直接跳过，可以拿0作为不连续的两数间的替补

讨论如下情况:

数组内无0的情况，这时候欲满足条件必须有 max - min = 4

数组内有0的情况，由于0可以是任意数，可以进一步把条件放宽为max - min <= 4

```
bool isStraight(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        int max_ = INT_MIN, min_ = INT_MAX;
        for(int i = 0; i < nums.size(); i++){
            if(nums[i] == 0)
                continue;
            if(i > 0 && nums[i - 1] == nums[i])
                return false;
            max_ = max(max_,nums[i]);
            min_ = min(min_,nums[i]);
        }

        return max_ - min_ < 5;
    }
```

剑指 Offer 66. 构建乘积数组(前缀和)

给定一个数组 A[0,1,…,n-1]，请构建一个数组 B[0,1,…,n-1]，其中 B[i] 的值是数组 A 中除了下标 i 以外的元素的积, 即 B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]。不能使用除法。

示例:

输入: [1,2,3,4,5]
输出: [120,60,40,30,24]

提示：

所有元素乘积之和不会溢出 32 位整数
a.length <= 100000

思路：前缀和类型的一道题，题解思想好理解，实现颇花了点脑子。

注：一定要列表格！

```
vector<int> constructArr(vector<int>& a) {
        int n = a.size();
        if(n == 0)
            return {};
        vector<int> b(a);
        //对于每一个下标i，求出a[0]到a[i - 1]的前缀积(上三角)
        for(int i = 1; i < n; i++)
            a[i] *= a[i - 1];

        //对于每一个下标i，求出a[n - 1]到a[i + 1]的前缀积(下三角)
        for(int i = n - 2; i >= 0; i--)
            b[i] *= b[i + 1];


        vector<int> c(n);
        c[0] = b[1];
        c[n - 1] = a[n - 2];

        //对于每一个下标i，将a[0, i - 1]的累积和a[i + 1, n]的累积相积
        for(int i = 1; i <= n - 2; i++){
            c[i] = a[i - 1] * b[i + 1];
        }
        return c;
    }
```

由于结果数组的递推过程中`b[i]`只计算一次，因此可以直接将b作为结果数组，简化空间复杂度。

```
class Solution {
public:
    vector<int> constructArr(vector<int>& a) {
        int n = a.size();
        if(n == 0)
            return {};
        vector<int> b(a);
        //对于每一个下标i，求出a[0]到a[i - 1]的前缀积
        for(int i = 1; i < n; i++)
            a[i] *= a[i - 1];

        //对于每一个下标i，求出a[n - 1]到a[i + 1]的前缀积
        for(int i = n - 2; i >= 0; i--)
            b[i] *= b[i + 1];

        b[0] = b[1];
```

```
        //对于每一个下标i，将a[0, i - 1]的累积和a[i + 1, n]的累积相积
        for(int i = 1; i <= n - 2; i++){
            b[i] = a[i - 1] * b[i + 1];
        }
        b[n - 1] = a[n - 2];
        return b;
    }
};
```

栈，队列，堆

剑指 Offer 09. 用两个栈实现队列

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1)

示例 1：

输入：
["CQueue","appendTail","deleteHead","deleteHead"]
[[],[3],[],[]]
输出：[null,null,3,-1]
示例 2：

输入：
["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"]
[[],[],[5],[2],[],[]]
输出：[null,-1,null,null,5,2]
提示：

1 <= values <= 10000

思路：

两个栈分别用于代表队列的入和出，难点在于出队操作，要讨论辅助栈是否为空的情况，如果为空就要把主栈（用于入队）的元素都pop到辅助栈中，这样辅助栈最顶端的元素就是要出队的元素。

```
class CQueue {
public:
    CQueue() {

    }

    void appendTail(int value) {
        in.push(value);
    }

    int deleteHead() {
        //00
```

```
        if(in.empty() && out.empty())
            return -1;
        //10
        if(out.empty()){
            while(!in.empty()){
                out.push(in.top());
                in.pop();
            }
        }
        //10 01
        int ret = out.top();
        out.pop();
        return ret;
    }
    stack<int> in,out;
};
```

```
class MinStack {
public:
    stack<int> st1;
    stack<int> st2;
    int minval = INT_MAX;
    MinStack() {

    }
    void push(int x) {
        st1.push(x);
        minval = std::min(x,minval);
        st2.push(minval);
        // printf("push: %d, now min: %d\n",x,minval);
    }

    void pop() {
        int ret = st1.top();
        st1.pop();
        st2.pop();
        //最小值更新,注意pop后栈空的情况
        if(st2.empty())
            minval = INT_MAX;
        else
            minval = st2.top();
        // printf("pop: %d, now min: %d\n",ret,minval);
    }

    int top() {
        return st1.top();
    }

    int min() {
        return st2.top();
    }
```

```
};
```

剑指 Offer 30. 包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

示例:

MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min();   --> 返回 -3.
minStack.pop();
minStack.top();   --> 返回 0.
minStack.min();   --> 返回 -2.

辅助栈解法

```
class MinStack {
public:
    stack<int> st1;
    stack<int> st2;
    int minval = INT_MAX;
    MinStack() {

    }
    void push(int x) {
        st1.push(x);
        minval = std::min(x,minval);
        st2.push(minval);
        // printf("push: %d, now min: %d\n",x,minval);
    }

    void pop() {
        int ret = st1.top();
        st1.pop();
        st2.pop();
        //最小值更新,注意pop后栈空的情况
        if(st2.empty())
            minval = INT_MAX;
        else
            minval = st2.top();
        // printf("pop: %d, now min: %d\n",ret,minval);
    }

    int top() {
        return st1.top();
    }

    int min() {
```

```
        return st2.top();
    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->min();
 */
```

一个栈解法(空间O(1))

思路和上面大致相同，最小值在下，新值在上。

```
class MinStack {
public:
    stack<int> sta;
    int minval = INT_MAX;
    /** initialize your data structure here. */
    MinStack() {
    }

    void push(int x) {
        minval = std::min(minval,x);
        sta.push(minval);
        sta.push(x);
    }

    void pop() {
        int x = sta.top();
        sta.pop();
        sta.pop();
        //这部分反复拷贝的开销可以被什么优化？
        if(!sta.empty()){
            int tmp = sta.top();
            sta.pop();
            minval = sta.top();
            sta.push(tmp);
        }
        else{
            minval = INT_MAX;
        }
    }

    int top() {
        return sta.top();
    }

    int min() {
        return minval;
    }
```

```
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->min();
 */
```

再优化

参考大佬insomine的解法。

原先push的逻辑是：更新最小值->放入更新后的最小值和新元素

现在改为：放入先前最小值和新元素->更新最小值

将更新最小值的逻辑推后，这样新的元素更小时，它的前一个元素将成为次小值，`pop()`操作时只要`pop()`一次就能取到更新后的最小值了。

```
class MinStack {
public:
    /** initialize your data structure here. */
    stack<int> st;
    int minval = INT_MAX;
    MinStack() {

    }

    void push(int x) {
        st.push(minval);
        printf("push: %d\n",minval);
        minval = std::min(x,minval);
        st.push(x);
        printf("push: %d\n",x);
    }

    void pop() {
        printf("pop: %d\n",st.top());
        st.pop();
        minval = st.top();
        printf("pop: %d\n",st.top());
        st.pop();
    }

    int top() {
        return st.top();
```

```
    }

    int min() {
        return minval;
    }
};
```

再不懂两个程序的过程打印一下再手写一遍就知道了。

剑指 Offer 31. 栈的压入、弹出序列

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,5,3,2,1} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

示例 1：

输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
示例 2：

输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。

提示：

0 <= pushed.length == popped.length <= 1000
0 <= pushed[i], popped[i] < 1000
pushed 是 popped 的排列。

模拟

push数组对应快指针，popped数组对应慢指针，难点在于知道什么时候该弹出元素，处理逻辑是在发现当前放入的元素就是当前要popped的元素后就可以开始pop元素，而且由于push的序列和pop的序列可能有镜像对应的关系（比如push = [1,2,3] pop = [3,2,1]），

因此不一定只pop一次，得用while循环不断更新pop数组的指针到不能再pop了为止。

```
bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        stack<int> sta;
        int j = 0;
        for(int i = 0; i < pushed.size(); i++){
            sta.push(pushed[i]);
            while(!sta.empty() && sta.top() == popped[j] && j < popped.size()){
                sta.pop();
                j++;
            }
        }

        return sta.empty();
    }
```

进阶：空间复杂度O(1)

可以将pushed数组直接用作栈

```
 bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        int top = -1, out = 0, n = pushed.size();
        for(int i = 0; i < n; i++){
            pushed[++top] = pushed[i];
            while(top >= 0 && out < popped.size() && pushed[top] == popped[out])
{
                top--;
                out++;
            }

        }

        return out == popped.size();
    }
```

剑指 Offer 41. 数据流中的中位数 💀

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

void addNum(int num) - 从数据流中添加一个整数到数据结构中。
double findMedian() - 返回目前所有元素的中位数。
示例 1：

输入：
["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]
[[],[1],[2],[],[3],[]]
输出：[null,null,null,1.50000,null,2.00000]
示例 2：

输入：
["MedianFinder","addNum","findMedian","addNum","findMedian"]
[[],[2],[],[3],[]]
输出：[null,null,2.00000,null,2.50000]

限制：

最多会对 addNum、findMedian 进行 50000 次调用。

思路：

这题最初想法是直接手动数组模拟一个优先队列然后下标取中作答案，不知道为什么TLE了。

对顶堆解法：

维护两个堆 `queLeft` 和 `queRight` 分别装较小的一半元素和较大的一半元素，其中 `queLeft` 是大顶堆，`queRight` 是小顶堆，这样取中位数直接就能在两个堆中 `O(1)` 取到，数据流总数为偶数时，中位数是两个堆堆顶的元素和除以2.0, 为奇数时，规定 `queLeft` 元素比 `queRight` 多一位，这样中位数是 `queLeft` 堆顶的元素。

难点在于 `addNum` 的逻辑，需要保证如下循环不变量：

```
数据流总量为偶数时，queLeft容量等于queRight容量
数据流总量为奇数时，queLeft容量比queRight容量多1
```

为此需要考虑两个条件，分四种情况讨论：

`queLeft` 是否比 `queRight` 大1(决定堆间是否要进行元素交换)

```
`nums`是否小于`queLeft`顶部（决定nums该插入哪个堆）
```

情况00：`queLeft` 和 `queRight` 容量相等且 `nums` 大于等于 `queLeft` 顶部，这时候 `num` 应插入 `queRight` 中并从 `queRight` 取一位给 `queLeft`；

```
//00演示
初态：  num = 6
+-------    -------+
|1 2 3 4    5 6 7 8|
+-------    -------+
 queLeft     queRight

中间态：
+-------    ---------+
|1 2 3 4    5 6 6 7 8|
```

```
+--------    ----------+
queLeft     queRight

末态：
+----------    --------+
|1 2 3 4 5    6 6 7 8|
+----------    --------+
 queLeft     queRight
```

情况01：`queLeft`和`queRight`容量相等且`nums`小于`queLeft`顶部，这时候`num`可直接插入`queLeft`中；

```
//01演示
初态：  num = 3
+--------    --------+
|1 2 3 4    5 6 7 8|
+--------    --------+
 queLeft     queRight

末态：
+----------    --------+
|1 2 3 3 4    6 6 7 8|
+----------    --------+
 queLeft     queRight
```

代码：

```
if(queLeft.size() == queRight.size()){
    if(num >= queLeft.top()){
        queRight.push(num);
        queLeft.push(queRight.top());
        queRight.pop();
    }
    else{
        queLeft.push(num);
    }
}
```

情况10: `queLeft`比`queRight`大1且`nums`大于等于`queLeft`顶部，这时候`nums`能直接插入到`queRight`中

```
//10演示
初态：  num = 6
+---------    -------+
|1 2 3 3 4    5 6 7 8|
+---------    -------+
 queLeft     queRight

末态：
+---------    ---------+
|1 2 3 3 4    5 6 6 7 8|
+---------    ---------+
 queLeft     queRight
```

情况11: `queLeft` 比 `queRight` 大1且 `nums` 小于 `queRight` 顶部，这时候 `nums` 要插入到 `queLeft` 中, 同时弹出一位给 `queRight` 。

```
//11演示
初态：  num = 2
+---------    -------+
|1 2 3 3 4    5 6 7 8|
+---------    -------+
 queLeft      queRight

中间态：
+----------    -------+
|1 2 2 3 3 4    5 6 7 8|
+----------    -------+
 queLeft        queRight

末态：
+----------    ---------+
|1 2 2 3 3     4 5 6 7 8|
+----------    ---------+
 queLeft        queRight
```

代码:

```
else if(queLeft.size() - 1 == queRight.size()){
    if(num < queLeft.top()){
        queLeft.push(num);
        queRight.push(queLeft.top());
        queLeft.pop();
    }
    else{
        queRight.push(num);
    }
}
```

分好情况之后，就可以写代码了：

```
class MedianFinder {
```

```
public:
    priority_queue<int,vector<int>, less<int>> queLeft;
    priority_queue<int, vector<int>,greater<int>> queRight;

    /** initialize your data structure here. */
    MedianFinder() {

    }

    void addNum(int num) {
        if(queLeft.empty()){
            queLeft.push(num);
            return;
        }
         //用于调试
        // printf("nums = %d\n",num);

        //循环不变量：
        // 当数据流总量为偶数时，queLeft容量等于queRight容量
        // 当数据流总量为奇数时，queLeft容量比queRight容量多1
        if(queLeft.size() == queRight.size()){
            if(num >= queLeft.top()){
                queRight.push(num);
                queLeft.push(queRight.top());
                queRight.pop();
            }
            else{
                queLeft.push(num);
            }
        }
        else if(queLeft.size() - 1 == queRight.size()){
            if(num < queLeft.top()){
                queLeft.push(num);
                queRight.push(queLeft.top());
                queLeft.pop();
            }
            else{
                queRight.push(num);
            }
        }
         //用于调试
        // printf("left:\n");
        // printPq(queLeft);
        // printf("right\n");
        // printPq(queRight);
    }
    //用于调试
    template <typename T>
    void printPq(priority_queue<int,vector<int>, T> pq){
        vector<int> res;
        while(!pq.empty()){
            res.insert(res.begin(),pq.top());
            pq.pop();
        }
```

```
        for(int i : res){
            cout << i << " ";
        }
        cout << endl;
    }

    double findMedian() {
        if(queLeft.size() == queRight.size()){
            return (queLeft.top() + queRight.top()) / 2.0;
        }
        else{
            return queLeft.top();
        }
    }

};

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num);
 * double param_2 = obj->findMedian();
 */
```

addNum逻辑优化

分析上面逻辑，发现两个堆总有如下关系：如果`queLeft.size() - 1 == queRight.size()`，则无论`num`小于还是大于等于`queLeft`顶部，`queRight`都要被push一次。区别在于`queRight`是否会排序调整。

如果`queLeft.size() == queRight.size()`，则无论`num`小于还是大于等于`queLeft`顶部，大于等于queLeft顶部，queLeft都要被push一次。区别在于`queLeft`是否会排序调整。

如果在不需要堆交换的情况下，不直接把`num`放在应该放的堆，而是交由另一个堆交给它们呢？

```
        if(queLeft.size() == queRight.size()){
            if(num >= queLeft.top()){
                queRight.push(num);
                queLeft.push(queRight.top());
                queRight.pop();
            }
            else{
                queRight.push(num);
                queLeft.push(queRight.top());
                queRight.pop();
            }
        }
        else if(queLeft.size() - 1 == queRight.size()){
            if(num < queLeft.top()){
                queLeft.push(num);
                queRight.push(queLeft.top());
                queLeft.pop();
```

```
            }
            else{
                queLeft.push(num);
                queRight.push(queLeft.top());
                queLeft.pop();
            }
        }
```

会发现逻辑并不会变，只不过被push的栈不会发生排序调整，因为`num`这时就是极值。也就是说我们只需要关注两个堆的容量是否满足`queLeft.size() 至多比 queRight.size() 大1`这一条件即可。

综上可以对代码做进一步简化：

```
        if(queLeft.size() == queRight.size()){
                queRight.push(num);
                queLeft.push(queRight.top());
                queRight.pop();

        }
        else if(queLeft.size() - 1 == queRight.size()){
                queLeft.push(num);
                queRight.push(queLeft.top());
                queLeft.pop();
            }
        }
```

[剑指 Offer 59 - I. 滑动窗口的最大值]

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：

输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置　　最大值

[1 3 -1] -3 5 3 6 7　　3
1 [3 -1 -3] 5 3 6 7　　3
1 3 [-1 -3 5] 3 6 7　　5
1 3 -1 [-3 5 3] 6 7　　5
1 3 -1 -3 [5 3 6] 7　　6
1 3 -1 -3 5 [3 6 7]　　7

示例 2：

输入：nums = [1], k = 1
输出：[1]

提示：

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

思路：

单调队列法

维护一个单调队列，保证队头的元素永远是**窗口内的**最大值，队尾的元素永远是当前窗口右指针指向的值，且队内元素单调递减

```
class mono_queue{
    public:
        void pop(int val){
            if(que.front() == val)
                que.pop_front();
        }
        void push(int val){
            while(!que.empty() && que.back() < val)
                que.pop_back();
            que.push_back(val);

        }
        int front(){
            return que.front();
        }

        deque<int> que;
};
vector<int> maxSlidingwindow(vector<int>& nums, int k) {
    vector<int> res;
    mono_queue que;
    for(int i = 0; i < k; i++)
        que.push(nums[i]);
    res.push_back(que.front());
    for(int i = k; i < nums.size(); i++){
        que.push(nums[i]);
        que.pop(nums[i - k]);
        res.push_back(que.front());
    }
    return res;
}
```

面试题59 - II. 队列的最大值

请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back 和 pop_front 的均摊时间复杂度都是O(1)。

若队列为空，pop_front 和 max_value 需要返回 -1

示例 1：

输入:
["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
[[],[1],[2],[],[],[]]
输出: [null,null,null,2,1,2]
示例 2：

输入:
["MaxQueue","pop_front","max_value"]
[[],[],[]]
输出: [null,-1,-1]

限制：

1 <= push_back,pop_front,max_value的总操作数 <= 10000
1 <= value <= 10^5

思路：要求push_back为O(1)时间，首先排除优先队列。

由于之前做过单调队列相关题，自然想到单调队列， 但是对于处理pop顺序的问题一直想不通。

大佬的解法是，除了用单调队列外，另设一个辅助队列存放真实存放的元素，这样要求最大值就把单调队列内的元素给露出来，要pop就把辅助队列的元素弹出去并更新单调队列队头只需要保证两个队列队尾都在数据窗口的最右侧即可。

演示: push序列{1,2,3,4,5}

```
v_que[5]
r_que[1,2,3,4,5]
```

```
class mono_queue{
    public:
        int front(){
            return que.front();
        }
        void pop(int val){
            if(que.front() == val)
                que.pop_front();
        }
        void push(int val){
            while(!que.empty() && que.back() < val)
                que.pop_back();
            que.push_back(val);
        }
        deque<int> que;
};
class MaxQueue {
```

```
public:
    //单调队列作虚队列，负责维护最大值，另有辅助队列负责pop值的处理
    mono_queue v_que;
    queue<int> r_que;
    MaxQueue() {

    }

    int max_value() {
        if(r_que.empty())
            return -1;
        return v_que.front();
    }

    void push_back(int value) {
        r_que.push(value);
        v_que.push(value);
    }

    int pop_front() {
        if(r_que.empty())
            return -1;
        int ret = r_que.front();
        v_que.pop(ret);
        r_que.pop();
        return ret;
    }
};

/**
 * Your MaxQueue object will be instantiated and called as such:
 * MaxQueue* obj = new MaxQueue();
 * int param_1 = obj->max_value();
 * obj->push_back(value);
 * int param_3 = obj->pop_front();
 */
```

双指针

剑指 Offer 21. 调整数组顺序使奇数位于偶数前面

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数在数组的前半部分，所有偶数在数组的后半部分。

示例：

输入：nums = [1,2,3,4]
输出：[1,3,2,4]
注：[3,1,2,4] 也是正确的答案之一。

思路：典型双指针，用slow维护已满足条件的数组，fast遍历，遍历完后就是结果数组

```
vector<int> exchange(vector<int>& nums) {
        int slow = 0, fast = 0;
        int n = nums.size();
        while(fast < n){
            if(nums[slow] % 2)
                slow++;
            else if(nums[slow] % 2 == 0 && nums[fast] % 2){
                swap(nums[slow],nums[fast]);
                slow++;
            }
            fast++;
        }
        return nums;
    }
```

同样的问题：

80. 删除有序数组中的重复项 II

283. 移动零

剑指 Offer 57. 和为s的两个数字

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们的和正好是s。如果有多对数字的和等于s，则输出任意一对即可。

示例 1：

输入：nums = [2,7,11,15], target = 9
输出：[2,7] 或者 [7,2]
示例 2：

输入：nums = [10,26,30,31,47,60], target = 40
输出：[10,30] 或者 [30,10]

限制：

1 <= nums.length <= 10^5
1 <= nums[i] <= 10^6

思路：数组已经排好序，明显提示二分。

```
    vector<int> twoSum(vector<int>& nums, int target) {
        for(int l = 0; l < nums.size(); l++){
            int r = bi_search(nums, l + 1, target - nums[l]);
            if(r != -1)
                return {nums[l],nums[r]};
        }
        return {-1,-1};
    }
    int bi_search(vector<int>& nums, int l, int tar){
        int r = nums.size() -1;
```

```
        while(l <= r){
            int mid = l + (r - l) / 2;
            if(nums[mid] < tar)
                l = mid + 1;
            else if(nums[mid] > tar)
                r = mid - 1;
            else
                return mid;
        }
        return -1;
    }
```

剑指 Offer 58 - I. 翻转单词顺序

输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串"I am a student. "，则输出"student. a am I"。

示例 1：

输入: "the sky is blue"
输出: "blue is sky the"
示例 2：

输入: " hello world! "
输出: "world! hello"
解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
示例 3：

输入: "a good   example"
输出: "example good a"
解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

说明：

无空格字符构成一个单词。
输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。
注意：本题与主站 151 题相同：https://leetcode-cn.com/problems/reverse-words-in-a-string/

注意：此题对比原题有改动

```
string reverseWords(string s) {
        remove_extra_space(s);
        reverse(s.begin(),s.end());
```

```
        int l = 0;
        for(int r = 1; r <= s.size(); r++){
            if(r == s.size() || s[r] == ' '){
                reverse(s.begin() + l, s.begin() + r);
                l = r + 1;
            }
        }
        return s;
    }
    void remove_extra_space(string &s){
        int slow = 0, fast = 0;
        //去前导空格
        while(s[fast] == ' '){
            fast++;
        }
        while(fast < s.size()){
            if(s[fast] == ' ' && s[fast - 1] == ' '){
                fast++;
            }
            else{
                s[slow++] = s[fast++];
            }
        }
        //检测尾部空格
        if(slow - 1 > 0 && s[slow - 1] == ' ')
            s = s.substr(0, slow - 1);
        else
            s = s.substr(0, slow);
    }
```

滑动窗口

剑指 Offer 48. 最长不含重复字符的子字符串

请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

示例 1:

```
输入: "abcabcbb"
输出: 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:

```
输入: "bbbbb"
输出: 1
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:

```
输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
      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：

- `s.length <= 40000`

思路：类似于76. 最小覆盖子串，不过变成了求最长连续不重复子串。

求连续子串，考虑用滑动窗口，如果窗口纳入的子串发生了重复，就拨动左指针到子串不重复为止。

可以直接用维护一个结果字符串在循环时不断更新，但是substr也是O(N)的，导致最后时间复杂度会是O(N^2), 因此改为维护一对双指针表示结果字符串的左右边界，最后只会调用一次substr， 时间复杂度优化为O(N)。

```
int lengthOfLongestSubstring(string str) {
        if(str.size() == 0){
            cout << "" << endl;
            return 0;
        }

        int len = 0, maxlen = 1, maxleft = 0, maxright = 0;
        int l = 0;
        unordered_map<char,int> cnt_map;
        int r = 0;
        for(;r < str.size(); r++){
            cnt_map[str[r]]++;
            if(cnt_map[str[r]] > 1){
                while(cnt_map[str[r]] > 1){
                    cnt_map[str[l]]--;
                    l++;
                }
            }
            else{
                if(r - l + 1 > maxlen){
                    maxlen = r - l + 1;
                    maxright = r;
                    maxleft = l;
                }
            }
        }

        return maxright - maxleft + 1;
    }
```

剑指 Offer 57 - II. 和为s的连续正数序列

输入一个正整数 target ，输出所有和为 target 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例 1：

输入：target = 9
输出：[[2,3,4],[4,5]]
示例 2：

输入：target = 15
输出：[[1,2,3,4,5],[4,5,6],[7,8]]

限制：

1 <= target <= 10^5

思路：一开始以为是回溯暴力，看题解才发现是滑动窗口和等差数列公式的综合运用。。。

时间复杂度O(N), 空间复杂度O(1)

```
vector<vector<int>> findContinuousSequence(int target) {
        if(target < 3)
            return {{}};
        vector<vector<int>> res;
        int l = 1, r = 2;


        //r最多滑动(target + 1) / 2, 举例5 + 5 = 10 -> 5, 4 + 5 = 9 -> 5
        while(r <= (target + 1) / 2){
            if(target == (r - l + 1) * (l + r) / 2){
                vector<int> path;
                for(int i = l; i <= r; i++){
                    path.push_back(i);
                }
                res.push_back(path);
                l++;
            }
            else if(target > (r - l + 1) * (l + r) / 2){
                r++;
            }
            else{
                l++;

            }

        }
        return res;
    }
```

剑指 Offer 58 - II. 左旋转字符串

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

示例 1：

输入: s = "abcdefg", k = 2
输出: "cdefgab"
示例 2：

输入: s = "lrloseumgh", k = 6
输出: "umghlrlose"

限制：

1 <= k < s.length <= 10000

思路：

解法1，拼接子串，时间复杂度和空间复杂度都是O(N)

```
string reverseLeftWords(string s, int k) {
        return s.substr(k, s.size() - k) + s.substr(0, k);
    }
```

解法2，原地反转，比较考验找规律，能把空间复杂度优化到O(1)

```
 string reverseLeftWords(string s, int k) {
        reverse(s.begin(), s.begin() + k);
        reverse(s.begin() + k, s.end());
        reverse(s.begin(), s.end());
        return s;
    }
```

链表

剑指 Offer 06. 从尾到头打印链表

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

输入：head = [1,3,2]
输出：[2,3,1]

限制：

0 <= 链表长度 <= 10000

来源：力扣（LeetCode）
链接：https://leetcode.cn/problems/cong-wei-dao-tou-da-yin-lian-biao-lcof

思路：链表题，一定要画图！

头插法迭代

```
    vector<int> reversePrint(ListNode* head) {
        vector<int> res;
        if(head == nullptr)
            return res;
        ListNode *dummyHead = new ListNode(-1);
        dummyHead->next = head;
        ListNode *pre = dummyHead, *cur = head;
        while(cur->next){
            ListNode *tmp = cur->next;
            cur->next = tmp->next;
            tmp->next = pre->next;
            pre->next = tmp;
        }

        cur = dummyHead->next;
        while(cur){
            res.push_back(cur->val);
            cur = cur->next;
        }

        return res;
    }
```

剑指 Offer 18. 删除链表的节点

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

注意：此题对比原题有改动

示例 1:

输入: head = [4,5,1,9], val = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
示例 2:

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

说明：

题目保证链表中节点的值互不相同
若使用 C 或 C++ 语言，你不需要 free 或 delete 被删除的节点

```
ListNode* deleteNode(ListNode* head, int val) {
        ListNode *dummyHead = new ListNode(-1);
        dummyHead->next = head;
        ListNode *pre = dummyHead;
        while(pre->next->val != val){
            pre = pre->next;
        }
        pre->next = pre->next->next;
        return dummyHead->next;
    }
```

剑指 Offer 22. 链表中倒数第k个节点

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。

例如，一个链表有 6 个节点，从头节点开始，它们的值依次是 1、2、3、4、5、6。这个链表的倒数第 3 个节点是值为 4 的节点。

示例：

给定一个链表: 1->2->3->4->5, 和 k = 2.

返回链表 4->5.

思路：

类似19. 删除链表的倒数第 N 个结点，求倒数第k个节点本质是求第 `链表长度 - k` 个节点，可以用快慢指针，从起点开始，让快指针先走k步，然后快慢指针一起运动到快指针越界为止，这时候慢指针站的位置就是倒数第k个节点。

```
 */
class Solution {
public:
    ListNode* getKthFromEnd(ListNode* head, int k) {
        ListNode *slow = head, *fast = head;
        while(k--){
            fast = fast->next;
        }
        while(fast){
            slow = slow->next;
            fast = fast->next;
        }
```

```
        return slow;
    }
};
```

剑指 Offer 24. 反转链表

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例:

输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL

限制:

0 <= 节点个数 <= 5000

来源：力扣（LeetCode）
链接：https://leetcode.cn/problems/fan-zhuan-lian-biao-lcof
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

迭代

假头+头插法

```
ListNode* reverseList(ListNode* head) {
        ListNode *dummyHead = new ListNode(-1);
        dummyHead->next = head;
        ListNode *pre = dummyHead, *cur = head;
        while(cur && cur->next){
            ListNode *tmp = cur->next;
            cur->next = tmp->next;
            tmp->next = pre->next;
            pre->next = tmp;
        }

        return dummyHead->next;
    }
```

递归

现在看着仍旧觉得晦涩。。。。不想了

```
ListNode* reverseList(ListNode* head) {
        if(!head || !head->next)
            return head;
        auto ret = reverseList(head->next);
        head->next->next = head;
        head->next = nullptr;
        return ret;
    }
```

剑指 Offer 25. 合并两个排序的链表

输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。

示例1：

输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
限制：

0 <= 链表长度 <= 1000

迭代

当时还因为返回哪个头懵想了半天，后来看题解才发现也可以用假头法。。。枯了

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummyHead = new ListNode(-1);
        ListNode* cur = dummyHead;
        while(l1 && l2){
            if(l1->val <= l2->val){
                cur->next = l1;
                cur = cur->next;
                l1 = l1->next;
            }
            else{
                cur->next = l2;
                cur = cur->next;
                l2 = l2->next;
            }
        }

        if(l1){
            cur->next = l1;
        }
        else{
            cur->next = l2;
        }

        return dummyHead->next;
```

```
    }
};
```

递归

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(!l1)
            return l2;
        else if(!l2)
            return l1;

        if(l1->val <= l2->val){
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        else{
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

剑指 Offer 35. 复杂链表的复制

请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。

示例 1：

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

示例 2：

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

输入：head = [[3,null],[3,0],[3,null]]
输出：[[3,null],[3,0],[3,null]]
示例 4：

输入：head = []
输出：[]
解释：给定的链表为空（空指针），因此返回 null。

提示：

-10000 <= Node.val <= 10000
Node.random 为空（null）或指向链表中的节点。
节点数目不超过 1000 。

问题分析：

1.要构造链表的next

2.要构造random，难点在这，因为难以定位新链表random将指向的节点

第一种解法是用哈希表法，好理解，就是用哈希表将原有节点和拷贝节点做一一映射，这样next拉完之后，就能通过哈希表查到拷贝节点random要指向的节点了。我没弄，因为并不是最优解法。这个方法时间复杂度和空间复杂度都是O(N);

第二章解法是拼接链表，将每个拷贝节点都固定到原有节点的next， 遍历梳理完random后将拷贝链表分离出去。这个方法由于是原地，故空间复杂度能优化到O(1)。

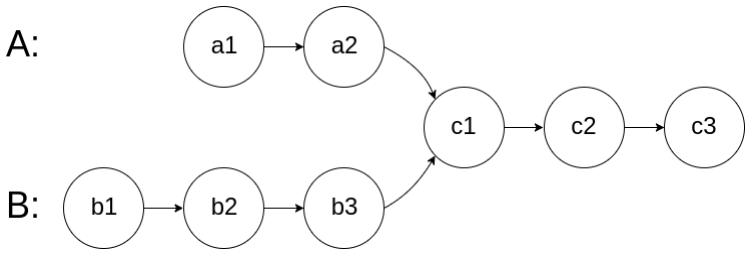
```
Node* copyRandomList(Node* head) {
        if(!head)
            return nullptr;
        Node *cur = head, *cpycur;
        //拼接链表，拉next
        while(cur){
            cpycur = new Node(cur->val);
            cpycur->next = cur->next;
            cur->next = cpycur;
            cur = cur->next->next;
        }

        cur = head;

        //拉random
        while(cur && cur->next){
            cpycur = cur->next;
            if(cur->random != nullptr)
                cpycur->random = cur->random->next;
            cur = cur->next->next;
        }

        //将拷贝链表分离出去
        cur = head;
        cpycur = cur->next;
        Node *res = cur->next;
        while(cpycur && cpycur->next){
            cur->next = cur->next->next;
            cpycur->next = cpycur->next->next;

            cur = cur->next;
            cpycur = cpycur->next;
        }
        cur->next = nullptr;
```

```
        return res;
    }
```

剑指 Offer 52. 两个链表的第一个公共节点

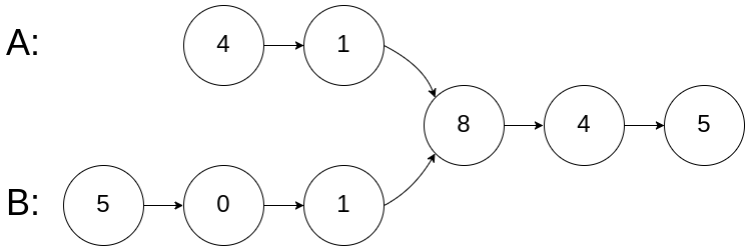
输入两个链表，找出它们的第一个公共节点。

如下面的两个链表：



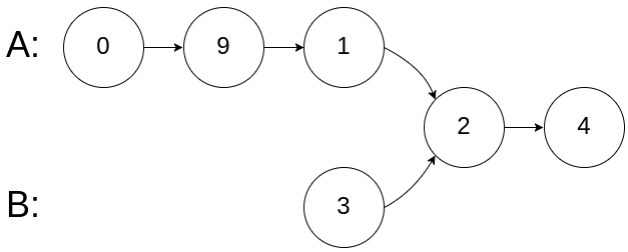
在节点 c1 开始相交。

示例 1：



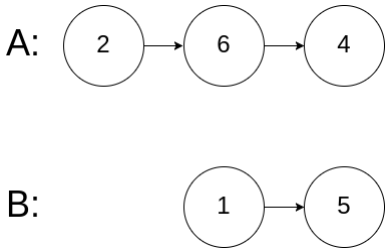

```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

示例 2：




```
输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。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：




```
输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。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。
解释：这两个链表不相交，因此返回 null。
```

注意：

- 如果两个链表没有交点，返回 `null`.
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(*n*) 时间复杂度，且仅用 O(*1*) 内存。
- 本题与主站 160 题相同：https://leetcode-cn.com/problems/intersection-of-two-linked-lists/

思路：写了N遍了，自然想到双指针法。

首先两个指针从各自链表头开始遍历，分别求出各自的链表的长度，再减去长度差，让两个指针从头开始，让长链表的指针先移动和长度差相等的步数，之后两个指针就是同一起点了，一起移动就能相遇。

如果不熟悉的话应该会想到哈希表先遍历A遍历记录节点再遍历B链表逐个检查，不知道面试该不该先抛出来。

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode *curA = headA, *curB = headB;
        int lenA = 0, lenB = 0;
        while(curA){
            curA = curA->next;
            lenA++;
        }

        while(curB){
            curB = curB->next;
            lenB++;
        }

        //默认长链表为B 短链表为A
        if(lenA > lenB){
            std::swap(lenA,lenB);
            std::swap(headA,headB);
        }

        curA = headA, curB = headB;
        int gap = lenB - lenA;
```

```
        while(gap--){
            curB = curB->next;
        }
        while(curA && curB && curA != curB){
            curA = curA->next;
            curB = curB->next;
        }

        return curA;
    }
```

二叉树

剑指 Offer 07. 重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请构建该二叉树并返回其根节点。

假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

示例 1:

Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
示例 2:

Input: preorder = [-1], inorder = [-1]
Output: [-1]

限制：

0 <= 节点个数 <= 5000

思路：分治，每次递归preorder数组指针+1，指向下一个根节点，按先序遍历的顺序构造树。难点在于中序遍历数组的切分。

```
 TreeNode* build(vector<int>& preorder, int ps, int pe, vector<int>& inorder,
int is, int ie){
        if(ps == pe){
            return nullptr;
        }
        TreeNode *root = new TreeNode(preorder[ps]);
        if(pe - ps == 1){
            return root;
        }
        int mid = 0;
        for(;mid < inorder.size(); mid++){
```

```
            if(inorder[mid] == root->val)
                break;
        }
        int pre_left_beg = ps + 1;
        int pre_left_end = ps + 1 + (mid - is);
        int pre_right_beg = ps + 1 + (mid - is);
        int pre_right_end = pe;
        root->left = build(preorder, pre_left_beg, pre_left_end, inorder, is,
mid - 1);
        root->right = build(preorder, pre_right_beg, pre_right_end, inorder, mid
+ 1, ie);

        return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        TreeNode *root = build(preorder,0 ,preorder.size(), inorder, 0,
inorder.size());
        return root;
    }
```

剑指 Offer 26. 树的子结构

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构， 即 A中有出现和B相同的结构和节点值。

例如:
给定的树 A:

```
     3
    / \
   4   5
  / \
 1   2
```

给定的树 B：

```
   4
  /
 1
```

返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。

示例 1：

输入：A = [1,2,3], B = [3,1]
输出：false
示例 2：

输入：A = [3,4,5,1,2], B = [4,1]
输出：true
限制：

0 <= 节点个数 <= 10000

思路：

572. 另一棵树的子树的变种, 由于约定空树不是任意一个树的子结构，故无A和无B都不能匹配，可以先排除掉，然后讨论三种情况：A和B根节点相等，A的左子树与B，A的右子树与B

```
 bool compare(TreeNode *A, TreeNode *B){
        if(!B) return true;
        if(!A && B) return false;
        if(A->val != B->val) return false;
        bool left = compare(A->left, B->left);
        bool right = compare(A->right, B->right);
        return left && right;
    }
    bool isSubStructure(TreeNode* A, TreeNode* B) {
        if(!A || !B)
            return false;
        return compare(A,B) || isSubStructure(A->left,B) || isSubStructure(A-
>right,B);
    }
```

另附572. 另一棵树的子树代码

```
   bool isSameTree(TreeNode* p, TreeNode* q){
        if(!p && !q)
            return true;
        if(!p && q || p && !q)
            return false;
        if(p->val != q->val)
            return false;
        bool isLeftVaild = isSameTree(p->left,q->left);
        bool isRightVaild = isSameTree(p->right,q->right);
        return isLeftVaild && isRightVaild;
    }
    bool isSubtree(TreeNode* root, TreeNode* subRoot) {
        if(!root)
            return false;
        return isSameTree(root,subRoot) || isSubtree(root->left,subRoot) ||
isSubtree(root->right,subRoot);
    }
```

剑指 Offer 27. 二叉树的镜像

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

例如输入：

```
  4
```

```
 / \
 2   7
/\  /\
1  3 6  9
```

镜像输出：

```
  4
```

```
 / \
 7   2
/\  /\
9  6 3  1
```

示例 1：

输入：root = [4,2,7,1,3,6,9]
输出：[4,7,2,9,6,3,1]

限制：

0 <= 节点个数 <= 1000

思路：就是翻转二叉树

递归：

```
    TreeNode* mirrorTree(TreeNode* root) {
        if(!root)
            return nullptr;
        std::swap(root->left,root->right);
        root->left = mirrorTree(root->left);
        root->right = mirrorTree(root->right);
        return root;
    }
```

迭代：

```
   TreeNode* mirrorTree(TreeNode* root) {
        stack<TreeNode*> sta;
        TreeNode *cur = root;
        while(!sta.empty() || cur != nullptr){
            if(cur != nullptr){
                sta.push(cur);
                std::swap(cur->left,cur->right);
```

```
                cur = cur->left;
            }
            else{
                cur = sta.top();
                sta.pop();
                cur = cur->right;
            }
        }

        return root;
    }
```

剑指 Offer 28. 对称的二叉树

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```
1
```

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

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

```
1
```

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

示例 1：

输入：root = [1,2,2,3,4,4,3]
输出：true
示例 2：

输入：root = [1,2,2,null,3,null,3]
输出：false

限制：

来源：力扣（LeetCode）
链接：https://leetcode.cn/problems/dui-cheng-de-er-cha-shu-lcof

递归:

```
class Solution {
public:
    bool compare(TreeNode *p, TreeNode *q){
        if(!p && !q) return true;
        if(!p && q) return false;
        if(p && !q) return false;
        if(p->val != q->val)  return false;

        bool outward = compare(p->left, q->right);
        bool inward = compare(p->right, q->left);

        return outward && inward;
    }
    bool isSymmetric(TreeNode* root) {
        if(!root)
            return true;
        return compare(root->left, root->right);
    }
};
```

迭代:

```
 bool isSymmetric(TreeNode* root) {
        if(!root)
            return true;
        queue<TreeNode*> que;
        que.push(root->left);
        que.push(root->right);

        while(!que.empty()){
            TreeNode *p = que.front();
            que.pop();
            TreeNode *q = que.front();
            que.pop();
            if(!p && !q)
                continue;
            if(!p && q)
                return false;
            if(p && !q)
                return false;
            if(p->val != q->val)
                return false;
            que.push(p->left);
            que.push(q->right);
            que.push(p->right);
            que.push(q->left);
        }

        return true;
```

```
    }
```

剑指 Offer 32 - III. 从上到下打印二叉树 III

I II都是简单的层序遍历，就不放了

III有点难度，可以在II的基础上加逻辑，设定奇数层（从1开始）从左向右遍历，偶数层从右向左遍历。

```
vector<vector<int>> levelOrder(TreeNode* root) {
        if(!root)
            return {};
        queue<TreeNode*> que;
        que.push(root);
        vector<vector<int>> res;
        bool isright = true;
        while(!que.empty()){
            int sz = que.size();
            vector<int> lvl(sz);
            int right = 0, left = sz - 1;
            for(int i = 0; i < sz; i++){
                TreeNode *cur = que.front();
                que.pop();
                if(isright){
                    lvl[right++] = cur->val;
                }
                else{
                    lvl[left--] = cur->val;
                }
                if(cur->left)
                    que.push(cur->left);
                if(cur->right)
                    que.push(cur->right);
            }
            res.push_back(lvl);
            isright = !isright;
        }

        return res;
    }
```

剑指 Offer 33. 二叉搜索树的后序遍历序列

递归分治

1.确定根节点：后序遍历的根节点总是当前递归数组的最末端

2.确定左右子树：对于后序遍历序列，必有 `postorder[l, i] < root` 和 `postorder[i + 1, r] < root`

3.递归三部曲

难点：如何判断左右子树的正确性

```
    bool verifyPostorder(vector<int>& postorder) {
        return recur(postorder, 0, postorder.size() - 1);
    }
    bool recur(vector<int>& postorder, int l, int r){
        if(l >= r)
            return true;
        int i = l;
        while(postorder[i] < postorder[r])  i++;
        //mid是第一个大于root的节点
        int mid = i;
        while(postorder[i] > postorder[r])  i++;
        return i == r && recur(postorder,l, mid - 1) && recur(postorder, mid, r
- 1);
    }
```

剑指 Offer 34. 二叉树中和为某一值的路径

给你二叉树的根节点 root 和一个整数目标和 targetSum ， 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。

叶子节点 是指没有子节点的节点。

示例 1：

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

输入：root = [1,2,3], targetSum = 5
输出：[]
示例 3：

输入：root = [1,2], targetSum = 0
输出：[]

提示：

树中节点总数在范围 [0, 5000] 内
-1000 <= Node.val <= 1000
-1000 <= targetSum <= 1000

同112. 路径总和，又复习了一遍。

递归

```
vector<vector<int>> res;
```

```
    vector<int> path;
    void backTracking(TreeNode *root, int target){
        if(!root->left && !root->right){
            if(target == 0)
                res.push_back(path);
            return;
        }
        if(root->left){
            path.push_back(root->left->val);
            backTracking(root->left, target - root->left->val);
            path.pop_back();
        }
        if(root->right){
            path.push_back(root->right->val);
            backTracking(root->right, target - root->right->val);
            path.pop_back();
        }

    }
    vector<vector<int>> pathSum(TreeNode* root, int target) {
        if(!root)
            return res;
        path.push_back(root->val);
        backTracking(root, target - root->val);
        return res;
    }
```

剑指 Offer 36. 二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

为了让您更好地理解问题，以下面的二叉搜索树为例：

我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。

特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

思路：

BST，自然中序遍历。
难点：确定链表头尾
做不出来的原因：没意识到中序遍历一直遍历到末尾可以把前驱节点遍历到链表尾巴上

```
```

```
class Solution {
public:
    Node *head, *pre;
    void recur(Node* cur){
        if(!cur)
            return;
        recur(cur->left);
        if(pre){
            pre->right = cur;
            cur->left = pre;
        }
        else{
            head = cur;
        }
        pre = cur;
        recur(cur->right);
    }
    Node* treeToDoublyList(Node* root) {
        if(!root)
            return nullptr;
        recur(root);
        Node *rear = pre;
        head->left = rear;
        rear->right = head;

        return head;
    }
};
```

剑指 Offer 37. 序列化二叉树

请实现两个函数，分别用来序列化和反序列化二叉树。

你需要设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

提示：输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

示例：

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

思路：

正序列化和反序列化都采用层序遍历

被恶心了一整晚，正序列化一看就想到层序遍历独立搞出来了，反序列化半天不会，最后还是乖乖看别人题解解决。

反序遍历难点->找出左右子树的根节点->用一个i指向出队结点的左右子树根节点

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Codec {
public:
    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if(!root)
            return "";
        queue<TreeNode*> que;
        que.push(root);
        string str;

        while(!que.empty()){
            TreeNode *cur = que.front();
            que.pop();
            if(!cur){
                str += "n ";
                continue;
            }
            else{
                str += to_string(cur->val);
            }
            que.push(cur->left);
            que.push(cur->right);
            str += ' ';
        }
        str.pop_back();
        return str;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data.empty())
            return nullptr;

        istringstream iss(data);
        vector<string> ser;
        string str;
        while(iss >> str){
            ser.push_back(str);
        }

        queue<TreeNode*> que;
        TreeNode *root = new TreeNode(stoi(ser[0]));
        que.push(root);
        int i = 1;
        while(!que.empty()){
            TreeNode *cur = que.front();
```

```
            que.pop();
            if(ser[i] != "n"){
                cur->left = new TreeNode(stoi(ser[i]));
                que.push(cur->left);
            }
            if(ser[i + 1] != "n"){
                cur->right = new TreeNode(stoi(ser[i + 1]));
                que.push(cur->right);
            }
            i += 2;
        }

        return root;
    }

};

// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.deserialize(codec.serialize(root));
```

剑指 Offer 54. 二叉搜索树的第k大节点

给定一棵二叉搜索树，请找出其中第 k 大的节点的值。

示例 1:

输入: root = [3,1,4,null,2], k = 1

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

输出: 4

示例 2:

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

```
     5
    / \
   3   6
  / \
 2   4
/
1
```

输出: 4

限制:

1 ≤ k ≤ 二叉搜索树元素个数

思路：

反向中序遍历，每次遍历递减k直到k == 0 就是当前值，终止程序

```
    int kthLargest(TreeNode* root, int k) {
        stack<TreeNode*> st;
        while(!st.empty() || root != nullptr){
            if(root != nullptr){
                st.push(root);
                root = root->right;
            }
            else{
                root = st.top();
                k--;
                if(k == 0){
                    return root->val;
                }
                st.pop();
                root = root->left;
            }
        }

        return -1;
    }
```

剑指 Offer 55 - I. 二叉树的深度

输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

例如：

给定二叉树 [3,9,20,null,null,15,7]，

```
3
```

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

返回它的最大深度 3 。

提示：

节点总数 <= 10000

思路：后序遍历的应用

```
int maxDepth(TreeNode* root) {
        if(!root)
            return 0;
        //分别求出左子树和右子树的深度
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        // 左右子树中的最大深度加上当前节点就是当前树的最大深度
        return 1 + max(leftDepth, rightDepth);
    }
```

剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

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

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]

示例 1:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
示例 2:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。

说明:

所有节点的值都是唯一的。
p、q 为不同节点且均存在于给定的二叉搜索树中。

思路:

基于BST的性质，根据BST查找元素的思维查找到在[p, q]（默认p < q）内的节点即可。

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(p->val > q->val){
            std::swap(p,q);
        }
        while(root){
            if(root->val < p->val)
                root = root->right;
            else if(root->val > q->val)
                root = root->left;
            else
                break;
```

```
        }

        return root;
    }
```

剑指 Offer 68 - II. 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 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

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

示例 1:

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

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

说明:

所有节点的值都是唯一的。
p、q 为不同节点且均存在于给定的二叉树中。

思路：对于一般的二叉树，可采用后序遍历，将p和q节点回溯给公共祖先，公共祖先再将自己的位置汇报给根节点。

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root || root->val == p->val || root->val == q->val)
            return root;
        TreeNode* lnode = lowestCommonAncestor(root->left,p,q);
        TreeNode* rnode = lowestCommonAncestor(root->right,p,q);
        //三种情况 p和q分别在不同子树 pq都在左子树 pq都在右子树
        if(lnode && rnode)
            return root;
        if(lnode)
            return lnode;
        else
            return rnode;
    }
```

字符串

剑指 Offer 20. 表示数值的字符串

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。

数值（按顺序）可以分成以下几个部分：

若干空格
一个 小数 或者 整数
（可选）一个 'e' 或 'E' ，后面跟着一个 整数
若干空格
小数（按顺序）可以分成以下几个部分：

（可选）一个符号字符（'+' 或 '-'）
下述格式之一：
至少一位数字，后面跟着一个点 '.'
至少一位数字，后面跟着一个点 '.' ，后面再跟着至少一位数字
一个点 '.' ，后面跟着至少一位数字
整数（按顺序）可以分成以下几个部分：

（可选）一个符号字符（'+' 或 '-'）
至少一位数字
部分数值列举如下：

["+100", "5e2", "-123", "3.1416", "-1E-16", "0123"]
部分非数值列举如下：

["12e", "1a3.14", "1.2.3", "+-5", "12e+5.4"]

示例 1：

输入：s = "0"
输出：true
示例 2：

输入：s = "e"
输出：false
示例 3：

输入：s = "."
输出：false
示例 4：

输入：s = "   .1 "
输出：true

提示：

1 <= s.length <= 20
s 仅含英文字母（大写和小写），数字（0-9），加号 '+' ，减号 '-' ，空格 ' ' 或者点 '.' 。

```
class Solution {
public:
    //数据格式可简化为 A[.B][E|e]C
    //其中A C是有符号数 可不带符号 B是无符号整数
    bool scanSignedInt(string &s, int &idx){
```

```
        if(s[idx] == '+' || s[idx] == '-')
            idx++;
        return scanUnsignedInt(s,idx);
    }
    bool scanUnsignedInt(string &s, int &idx){
        int befor = idx;
        while(idx < s.size() && s[idx] >= '0' && s[idx] <= '9')
            idx++;
        return befor < idx;
    }
    bool isNumber(string s) {
        int idx = 0;
        //去前导空格
        while(s[idx] == ' ')
            idx++;
        //检测A
        bool isDigit = scanSignedInt(s,idx);
        //检测B
        if(s[idx] == '.'){
            idx++;
            isDigit = scanUnsignedInt(s,idx) ||  isDigit;
        }
        cout << isDigit << " " << idx << endl;

        //检测C
        if(s[idx]  == 'E' | s[idx] == 'e'){
            idx++;
            isDigit = scanSignedInt(s, idx) && isDigit;
        }

        //去后置空格
        while(s[idx] == ' ')
            idx++;

        return idx == s.size() && isDigit;
    }
};
```

剑指 Offer 67. 把字符串转换成整数

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

函数 myAtoi(string s) 的算法如下：

读入字符串并丢弃无用的前导空格
检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。 确定最终结果是负数还是正数。 如果两者都不存在，则假定结果为正。
读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
将前面步骤读入的这些数字转换为整数（即，"123" -> 123， "0032" -> 32）。如果没有读入数字，则整数为 0 。必要时更改符号（从步骤 2 开始）。
如果整数数超过 32 位有符号整数范围 [−231，231 − 1] ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 −231 的整数应该被固定为 −231 ，大于 231 − 1 的整数应该被固定为 231 − 1 。

返回整数作为最终结果。
注意：

本题中的空白字符只包括空格字符 ' ' 。
除前导空格或数字后的其余字符串外，请勿忽略 任何其他字符。

示例 1：

输入：s = "42"
输出：42
解释：加粗的字符串为已经读入的字符，插入符号是当前读取的字符。
第 1 步："42"（当前没有读入字符，因为没有前导空格）
^
第 2 步："42"（当前没有读入字符，因为这里不存在 '-' 或者 '+'）
^
第 3 步："42"（读入 "42"）
^
解析得到整数 42 。
由于 "42" 在范围 [-231, 231 - 1] 内，最终结果为 42 。
示例 2：

输入：s = "   -42"
输出：-42
解释：
第 1 步："   -42"（读入前导空格，但忽视掉）
^
第 2 步："   -42"（读入 '-' 字符，所以结果应该是负数）
^
第 3 步："   -42"（读入 "42"）
^
解析得到整数 -42 。
由于 "-42" 在范围 [-231, 231 - 1] 内，最终结果为 -42 。
示例 3：

输入：s = "4193 with words"
输出：4193
解释：
第 1 步："4193 with words"（当前没有读入字符，因为没有前导空格）
^
第 2 步："4193 with words"（当前没有读入字符，因为这里不存在 '-' 或者 '+'）
^
第 3 步："4193 with words"（读入 "4193"；由于下一个字符不是一个数字，所以读入停止）
^
解析得到整数 4193 。
由于 "4193" 在范围 [-231, 231 - 1] 内，最终结果为 4193 。

提示：

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

思路：自主做出，做过20，67就比较懂了，而且题目的要求和过程已经说得够清楚了。

首先去掉前导空格，然后判断符号，确定数字头尾部分，注意取到的数字也要进行过滤，比如前导0的情况。

```
int strToInt(string s) {
        //开头数字，数字串尾哨(左闭右开)，扫描索引
        int beg, end, idx = 0;
        //去前导空格和0
        while(s[idx] == ' ')
            idx++;

        //检测符号
        bool isNeg = false;
        if(s[idx] == '+' || s[idx] == '-'){
            if(s[idx] == '-')
                isNeg = true;
            idx++;
        }

        //检测数字部分
        beg = idx;
        while(idx < s.size() && s[idx] >= '0' && s[idx] <= '9')
            idx++;
        end = idx;
        //头尾重合，说明没有读入数字，返回0
        if(beg == end)
            return 0;

        //数字转换
        int ans = 0;
        //去前导0
        while(s[beg] == '0')
            beg++;
        //排除肯定溢出的结果
        if(end - beg > 10)
            return isNeg ? INT_MIN : INT_MAX;
        long pow = 1;
        for(int i = end - 1; i >= beg; i--){
            if(ans <= INT_MAX - (s[i] - '0') * pow){
                ans += (s[i] - '0') * pow;
                pow *= 10;
            }
            else{
                return isNeg ? INT_MIN : INT_MAX;
            }
        }

        return isNeg ? -ans : ans;
    }
```

看了下大佬们的题解，没想到的思路：

可以直接用一个整数代表符号位和结果相乘确定正负

每次循环将结果*10就能容纳当前遍历数，不需要另外设pow遍历

扫描数字过程中将INT_MAX和INT_MIN断乘10更容易判断溢出

扫描数字过程中遇到非数字直接break，可以单遍扫描，不需要先确定数字的区间

```
int strToInt(string str) {
        int sign = 1;
        int idx = 0;
        while(str[idx] == ' ')
            idx++;
        if(str[idx] == '+' || str[idx] == '-'){
            if(str[idx] == '-')
                sign = -sign;
            idx++;
        }

        int res = 0;
        while(idx < str.size()){
            if(str[idx] < '0' || str[idx] > '9')
                break;
            //防十位数溢出的情况 即土214748365x
            if(res > INT_MAX / 10)
                return (sign == 1) ? INT_MAX : INT_MIN;
            //防个位数溢出的情况 如-2147483649 2147483648 2147483649
            if(res == INT_MAX / 10 && str[idx] > '7')
                return (sign == 1) ? INT_MAX : INT_MIN;
            res = res * 10  + (str[idx] - '0');
            idx++;
        }
        return sign * res;
    }
```

贪心思想

剑指 Offer 14- II. 剪绳子 II

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0],k[1]...k[m - 1] 。请问 k[0]*k[1]*...*k[m - 1] 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1
示例 2:

输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36

提示：

2 <= n <= 1000

思路：

有大数用例，因此剑指 Offer 14- I. 剪绳子的dp解法用不了了，只能找规律。

n = 1时，剪不了，只能返回0

n = 2时，有1 x 1 = 1 < 1 + 1, 返回1

n = 3时，有2 x 1 = 2 < 2 + 1, 返回2

n = 4时，有2 x 2 = 4 < 2 + 2, 切不切都无所谓

综上n > 4时，1 2 3 都是不能切的，根据贪心思路应尽量剪出长度为3的段助益最大，其次是2, 1对乘积没有增益，因此要尽量避免剪出长度为1的段，当剩余长为4时，由于2 x 2 > 3 x 1, 因此可以贡献一个段3，来避免剪出段1。

最终将绳子分为多个长为2和长为3的子段后，计算所有字段的乘积。

剑指原解是这样：

```
 int cuttingRope(int n) {
        if(n < 4)
            return n - 1;
        int timesof3 = n / 3;
        if(n - 3 * timesof3 == 1)
            timesof3 -= 1;
        int timesof2 = (n - 3 * timesof3) / 2;
        return (int)pow(2,timesof2) * (int)pow(3,timesof3);
    }
```

LC这题坑爹的是，有大数用例，直接弄乘法会溢出，只能循环乘的同时求余，来避免大数溢出

```
int cuttingRope(int n) {
        if(n < 4)
            return n - 1;
        long ret = 1;
        if(n % 3 == 1){
            ret = 4;
            n -= 4;
        }
        if(n % 3 == 2){
            ret = 2;
            n -= 2;
        }
        while(n){
            ret = ret * 3 % 1000000007;
            n -= 3;
```

```
        }

        return ret;
    }
```

搜索/回溯

剑指 Offer 12. 矩阵中的路径

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。

例如，在下面的 3×4 的矩阵中包含单词 "ABCCED"（单词中的字母已标出）。

示例 1：

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

输入：board = [["a","b"],["c","d"]], word = "abcd"
输出：false

提示：

m == board.length
n = board[i].length
1 <= m, n <= 6
1 <= word.length <= 15
board 和 word 仅由大小写英文字母组成

思路：

当时一想到就是dfs, 只不过没想到每个格子都可能是起点。

总结了dfs的框架，在DFS_BFS小结中。

```
class Solution {
public:
    vector<vector<char>> board;
    string word;
    int dir[4][2] = {{0,1},{0,-1},{1,0},{-1,0}};
    bool inArea(int row, int col){
        if(row < 0 || row >= board.size() || col < 0 || col >= board[0].size())
```

```
            return false;
        return true;
    }
    bool dfs(int idx, int row, int col){
        if(idx == word.size() - 1){
            return word[idx] == board[row][col];
        }
        if(word[idx] == board[row][col]){
            //确定下一步
            for(int i = 0; i < 4; i++){
                int newrow = row + dir[i][0];
                int newcol = col + dir[i][1];
                if(inArea(newrow,newcol) && board[newrow][newcol] != '.'){
                    board[row][col] = '.';
                    if(dfs(idx + 1, newrow,newcol))
                        return true;
                    board[row][col] = word[idx];
                }
            }

        }
        return false;
    }
    bool exist(vector<vector<char>>& board, string word) {
        this->board = board;
        this->word = word;
        int m = board.size(), n = board[0].size();

        //每个格子都可能是起点
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(dfs(0, i,j))
                    return true;
            }
        }
        return false;
    }
};
```

面试题13. 机器人的运动范围

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 [35, 37] ，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

示例 1：

输入：m = 2, n = 3, k = 1
输出：3
示例 2：

输入：m = 3, n = 1, k = 0
输出：1

思路：可将数组看作一颗以(0,0)为根节点的二叉树作后序遍历

```
int m, n, k;
    vector<vector<int>> vis;
    int dfs(int i, int j){
        if(i >= m || j >= n || (i % 10 + i / 10) + (j % 10 + j / 10) > k ||
vis[i][j])
            return 0;
        vis[i][j] = true;
        return 1 + dfs(i + 1, j) + dfs(i, j + 1);
    }
    int movingCount(int m, int n, int k) {
        this->m = m;
        this->n = n;
        this->k = k;
        vis = vector<vector<int>>(m, vector<int>(n));
        return dfs(0, 0);
    }
```

```
    vector<string> res;
    string path;
    void backTracking(string &s, vector<bool> &used){
        if(path.size() == s.size()){
            res.push_back(path);
            return;
        }

        for(int i = 0; i < s.size(); i++){
            if(i > 0 && !used[i - 1] && s[i - 1] == s[i])
                continue;
            if(used[i])
                continue;
            used[i] = true;
            path.push_back(s[i]);
            backTracking(s,used);
            path.pop_back();
            used[i] = false;

        }
    }
    vector<string> permutation(string s) {
        sort(s.begin(), s.end());
        vector<bool> used(s.size());
        backTracking(s,used);
        return res;
```

```
    }
```

剑指 Offer 38. 字符串的排列

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

输入：s = "abc"
输出：["abc","acb","bac","bca","cab","cba"]

限制：

1 <= s 的长度 <= 8

思路：

和47. 全排列 II是几乎一样的，然而树层去重怎么弄给忘了，惭愧。

其实就是看前一个元素的used，如果前一个元素的used状态是false且前一个元素值等于当前元素，那就说明当前值下的树枝已经遍历过了，可直接跳过。

```
vector<string> res;
    string path;
    void backTracking(string &s, vector<bool> &used){
        if(path.size() == s.size()){
            res.push_back(path);
            return;
        }

        for(int i = 0; i < s.size(); i++){
            if(i > 0 && !used[i - 1] && s[i - 1] == s[i])
                continue;
            if(used[i])
                continue;
            used[i] = true;
            path.push_back(s[i]);
            backTracking(s,used);
            path.pop_back();
            used[i] = false;

        }
    }
    vector<string> permutation(string s) {
        sort(s.begin(), s.end());
        vector<bool> used(s.size());
        backTracking(s,used);
        return res;
    }
```

动态规划

剑指 Offer 14- I. 剪绳子

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0],k[1]...k[m-1] 。请问 k[0]*k[1]*...*k[m-1] 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

示例 1：

输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1
示例 2:

输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36
提示：

2 <= n <= 58

思路同343. 整数拆分。

dp[i]定义：整数i的最大乘积值

dp[i]可分为**不拆分**，**拆分为**`i - j`**和**`j`，以及**前者基础上dp[i - j]再拆分**讨论

要求dp[i]的最大乘积值, 综上有`dp[i] = max(dp[i], max( (i - j) * j, dp[i - j] * j )`

相比96. 不同的二叉搜索树

相比96. 不同的二叉搜索树都是将每个情况二分相乘, 不过这里被二分的元素还要考虑再拆分的情况，因此比96要难。

```
    int cuttingRope(int n) {
        vector<int> dp(n + 1);
        dp[1] = 1;
        for(int i = 2; i <= n; i++){
            for(int j = 1; j < i; j++){
                dp[i] = max(dp[i], max((i - j) * j, dp[i - j] * j));
            }
        }

        return dp[n];
    }
```

剑指 Offer 19. 正则表达式匹配

`dp[i][j]` **定义：** s前`i`个字符能否和p前`j`个字符匹配 (即s[0, i - 1]能否和 p[0, j - 1]匹配)

`dp` **初始化：** `dp[0][0]`要为true， 因为空字符串匹配空字符串。

当`p[j - 1]` == '*' 时，应有空字符串和`p[j - 2]*`匹配，因为`p[j - 2]*`占两个字符，因此循环步长为2

```
//初始化
dp[0][0] = true;
for(int j = 2; j <= p.size(); j += 2){
    if(p[j - 1] == '*' && dp[0][j - 2]){
        dp[0][j] = true;
    }
}
```

`dp` **递推方程:**

按`p[j - 1]`是否为`*`来讨论, 若`p[j - 1]` == '*' 则

```
dp[i][j] = {
                1 true if dp[i][j - 2]
                2 true if dp[i - 1][j] && s[i - 1] == p[j - 2]
                3 true if dp[i - 1][j] && p[j - 2] == '.'
            }
```

情况1对应的是`s[i - 1]` 和`p[j - 2]*`匹配0次时的情况，这时候需要检查`s[0, i)`和`p[0, j - 2)`是否匹配

i

s:**...xxxa**
p:**...xx**b*

j

情况2对应的是`s[i - 1]` 和`p[j - 2]*`匹配1次时的情况, 这时候检查的是`s[0, i - 1)`和`p[0, j)`能否匹配

i

s: **.....xxx**a
p:**.....xxa***

j

情况3可以和情况2合并，也是`s[i - 1]`和`p[j - 2]*`匹配的情况

若`p[j - 1]` != '*' 则有

```
dp[i][j] = {
                1 true if dp[i - 1][j - 1] && s[i - 1] == p[j - 1]
                2 true if dp[i - 1][j - 1] && p[j - 1] == '*'
            }
```

上述两个都是`s[i - 1]`匹配`p[j - 1]`的情况，可以合并，只看余下的`s[0, i - 1)`和`p[0, j - 1)`是否也匹配。

代码：

```
  bool isMatch(string s, string p) {
        //dp[i][j]定义 s[0,i)和 p[0,j)是否匹配
        int m = s.size(), n = p.size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));

        //初始化
        dp[0][0] = true;
        for(int j = 2; j <= p.size(); j += 2){
            if(p[j - 1] == '*' && dp[0][j - 2]){
                dp[0][j] = true;
            }
        }
        for(int i = 1; i <= s.size(); i++){
            for(int j = 1; j <= p.size(); j++){
                if(p[j - 1] == '*'){
                    //p[j - 2]* 出现0次的情况
                    if(dp[i][j - 2]) dp[i][j] = true;
                    //p[j - 2]* 出现1次的情况
                    if((s[i - 1] == p[j - 2] || p[j - 2] == '.') && dp[i - 1]
[j]) dp[i][j] = true;
                }
                else{
                    if((s[i - 1] == p[j - 1] || p[j - 1] == '.') && dp[i - 1][j
- 1]) dp[i][j] = true;
                }
            }
        }
        return dp[m][n];
    }
```

剑指 Offer 46. 把数字翻译成字符串

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a” ，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

示例 1:

输入: 12258
输出: 5
解释: 12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"mzi"

提示:

0 <= num < 231

解:

设数字n组成为x位，为n1n2...nx

`dp[i]` **定义**：n1n2...ni+1能被翻译的结果数，即数字n的前i位能被翻译的结果数

`dp[i]` **初始化**：空字符翻译结果为0, dp[0] = 1; 只有一个字符的翻译结果必为1， dp[1] = 1

`dp[i]` **递推**：每次至少要翻译1位字符，至多翻译2位字符；

能够翻译2位字符需要保证ni * 10 + ni+1 ∈ [10, 25]， 否则只能单独翻译

既能2位字符翻译也能单独翻译的情况下，`i`位字符的翻译结果是单独翻译第i位字符情况下`i`位字符的翻译结果数 + 组合翻译第`i - 1`,`i`位字符情况下`i`位字符的翻译结果数，即`dp[i] = dp[i - 1] + dp[i - 2]`

只能单独翻译的情况下，有`dp[i] = dp[i - 1]`,

综上有

```
dp[i] = dp[i - 1] + dp[i - 2], 10 * ni + ni-1  ∈ [10, 25]
        dp[i - 1], else
```

```
    int translateNum(int num) {
        string nums = to_string(num);
        //dp[i]定义：数字能被翻译的结果数
        vector<int> dp(nums.size() + 1);
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i <= nums.size(); i++){
            int xi_1 = nums[i - 2] - '0';
            int xi = nums[i - 1] - '0';
            if(xi_1 * 10 + xi >= 10 && xi_1 * 10 + xi <= 25)
                dp[i] = dp[i - 1] + dp[i - 2];
            else
                dp[i] = dp[i - 1];
        }

        return dp[nums.size()];
    }
```

剑指 Offer 47. 礼物的最大价值

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

示例 1:

输入:
[
 [1,3,1],
 [1,5,1],
 [4,2,1]
]
输出: 12
解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物

提示：

0 < grid.length <= 200
0 < grid[0].length <= 200

思路：62. 不同路径换了个皮。

```
int maxValue(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        for(int i = 1; i < m; i++)  grid[i][0] += grid[i - 1][0];
        for(int j = 1; j < n; j++)  grid[0][j] += grid[0][j - 1];

        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                grid[i][j] = grid[i][j] + max(grid[i - 1][j], grid[i][j - 1]);
            }
        }


        return grid[m - 1][n - 1];
    }
```

剑指 Offer 49. 丑数

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

示例:

输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
说明:

1 是丑数。
n 不超过1690。

解法：

优先队列

构建一个小根堆，初始放入1，之后每次都循环出队一个元素，将其 *2 *3 *5的放入， 这个队列最终会被弹出n次，第n次弹出的结果就是最小的丑数。

时间复杂度:O(NlogN) 空间复杂度O(N)

```
    int nthUglyNumber(int n) {
        priority_queue<double,vector<double>,greater<double>> pq;
        set<double> num_set;
        vector<int> factor{2,3,5};
        pq.push(1);
        num_set.insert(1);

        double num = 0;
        for(int i = 1; i <= n; i++){
            num = pq.top();
            for(int j = 0; j < 3; j++){
                if(num_set.count(num * factor[j]) == 0){
                    pq.push(num * factor[j]);
                    num_set.insert(num * factor[j]);
                }
            }
            pq.pop();
        }

        return num;
    }
```

动态规划

由于每个丑数的组成因子都是1,2,3,5，意味着一个丑数乘以**1,2,3,5**仍然会是丑数，因此可以维护三个序列，分别存放丑数*2, 丑数 * 3和 丑数 * 5的结果并用三个指针指向它们。那么动态规划的状态递推逻辑本质上就是这三个序列的归并排序。

参考：https://leetcode.cn/problems/chou-shu-lcof/solution/chou-shu-ii-qing-xi-de-tui-dao-si-lu-by-mrsate/

```
nums2 = {1*2, 2*2, 3*2, 4*2, 5*2, 6*2, 8*2...}
nums3 = {1*3, 2*3, 3*3, 4*3, 5*3, 6*3, 8*3...}
nums5 = {1*5, 2*5, 3*5, 4*5, 5*5, 6*5, 8*5...}
# 注意 7 不是丑数.
# 2, 3, 5 这前 3 个丑数一定要乘以其它的丑数， 所得的结果才是新的丑数， 所以上例中没有出现
7*2, 7*3, 7*5

dp = {1,2,3,4,5,6,8,9,10,12,16....}
```

时间复杂度:O(N) 空间复杂度O(N)

```
int nthUglyNumber(int n) {
        //dp[i]: 第i个丑数
        vector<double> dp(n);
        dp[0] = 1;
        int p2 = 0, p3 = 0, p5 = 0;
        for(int i = 1; i < n; i++){
            int num2 = dp[p2] * 2, num3 = dp[p3] * 3, num5 = dp[p5] * 5;
            dp[i] = min(num2, min(num3, num5));
            //由于num2,num3,num5也可能有重复的情况，故三个判断都是if
            //例：dp[p2] = 5 dp[p5] = 2 会有 2 * 5 = 5 * 2
            if(dp[i] == num2)
                p2++;
            if(dp[i] == num3)
                p3++;
            if(dp[i] == num5)
                p5++;
        }

        return dp[n - 1];
    }
```

剑指 Offer 63. 股票的最大利润

假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少?

示例 1:

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

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

限制：

0 <= 数组长度 <= 10^5

注意：本题与主站 121 题相同：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/

思路：

卡哥题解刷过一次股票问题，直接秒杀。

```
    int maxProfit(vector<int>& prices) {
        //dp[i][j]定义 在第i天持有j份股票的现金数
        int n = prices.size();
        if(n == 0)
            return 0;
        vector<vector<int>> dp(n, vector<int>(2));
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for(int i = 1; i < n; i++){
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = max(dp[i - 1][1], -prices[i]);
        }
        return dp[n - 1][0];
    }
```

由于`dp[i]`只由`dp[i - 1]`推出且只用一次，因此可以把时间复杂度优化到O(1),

```
int maxProfit(vector<int>& prices) {
        int n = prices.size();
        //dp0代表当天不买股票的现金数 dp1代表当天买股票后的现金数
        int dp0 = 0;
        int dp1 = -prices[0];
        for(int i = 1; i < n; i++){
            dp0 = max(dp0, dp1 + prices[i]);
            dp1 = max(dp1, -prices[i]);
        }
        return dp0;
    }
```

再稍微改一改就变成贪心解法了。。。

```
 int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if(n == 0)
            return 0;
        int profit = 0;
        int cost = prices[0];
        for(int i = 1; i < n; i++){
            profit = max(profit, prices[i] - cost);
            cost = min(cost, prices[i]);
        }
        return profit;
    }
```

二分

剑指 Offer 11. 旋转数组的最小数字

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。

给你一个可能存在 重复 元素值的数组 numbers ，它原来是一个升序排列的数组，并按上述情形进行了一次旋转。请返回旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一次旋转，该数组的最小值为 1。

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]] 。

示例 1：

输入：numbers = [3,4,5,1,2]
输出：1
示例 2：

输入：numbers = [2,2,2,0,1]
输出：0

提示：

n == numbers.length
1 <= n <= 5000
-5000 <= numbers[i] <= 5000
numbers 原来是一个升序排序的数组，并进行了 1 至 n 次旋转

评价：旋转数组的题做了不少，直接秒杀。

```
int minArray(vector<int>& nums) {
      //消二段性
      int l = 0, r = nums.size() - 1;
      while(r > 0 && nums[0] == nums[r])
          r--;
      //排除开始就顺序的情况
      if(r == 0 || nums[0] < nums[r])
          return nums[0];
      //找旋转点
      while(l + 1 < r){
          int mid = l + (r - l) / 2;
          if(nums[mid] <= nums[r])
              r = mid;
          else
              l = mid;
      }
      return nums[r];
  }
```

其他【旋转数组】题目：

81. 搜索旋转排序数组 II

面试题 10.03. 搜索旋转数组

剑指 Offer 53 - I. 在排序数组中查找数字 I

统计一个数字在排序数组中出现的次数。

示例 1:

输入: nums = [5,7,7,8,8,10], target = 8
输出: 2
示例 2:

输入: nums = [5,7,7,8,8,10], target = 6
输出: 0

提示：

0 <= nums.length <= 105
-109 <= nums[i] <= 109
nums 是一个非递减数组
-109 <= target <= 109

注意：本题与主站 34 题相同（仅返回值不同）：https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/

思路：

找出目标元素的第一个元素和最后一个元素的出现位置然后相减 `+1`，考察二分的熟练程度。

```
int search(vector<int>& nums, int target) {
        if(nums.size() == 0)
            return 0;
        int beg = find_first(nums,target);
        int end_ = find_last(nums,target);
        if(beg == -1 || end_ == -1)
            return 0;
        else
            return end_ - beg + 1;
    }
    int find_first(vector<int>& nums, int target){
        int l = 0, r = nums.size() - 1;
        while(l < r){
            //下面这行代码和下面r = mid协同使得mid在nums[mid] >= target的情况下不断往左偏移
            int mid = l + (r - l) / 2;
            if(nums[mid] < target){
                l = mid + 1;
```

```
            }
            else{
                r = mid;
            }
        }
        return nums[l] == target ? l : -1;
    }

    int find_last(vector<int>& nums, int target){
        int l = 0, r = nums.size() - 1;
        while(l < r){
            //下面这行代码和下面l = mid协同使得mid在nums[mid] <= target的情况下不断往右
偏移
            int mid = l + (r - l + 1) / 2;
            if(nums[mid] <= target){
                l = mid;
            }
            else{
                r = mid - 1;
            }
        }
        return nums[r] == target ? r : -1;
    }
```

剑指 Offer 53 - II. 0～n-1中缺失的数字

一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

示例 1:

输入: [0,1,3]
输出: 2
示例 2:

输入: [0,1,2,3,4,5,6,7,9]
输出: 8

限制:

1 <= 数组长度 <= 10000

思路:

有且只有一个数字不在该数组中，而且这个数字必有`nums[i] != i`， 可以转化为找到第一个`nums[i] != i`的问题去写个范围为[0, n - 1]的lower_bound。

```
    int missingNumber(vector<int>& nums) {
        int l = 0, r = nums.size();
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] == mid){
                l = mid + 1;
            }
            else{
                r = mid;
            }
        }
        return r;
    }
```

排序

剑指 Offer 40. 最小的k个数

输入整数数组 arr ，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]
示例 2：

输入：arr = [0,1,2,1], k = 1
输出：[0]

限制：

0 <= k <= arr.length <= 10000
0 <= arr[i] <= 10000

重新复习一遍快排

```
vector<int> getLeastNumbers(vector<int>& arr, int k) {

        sort(arr, 0, arr.size() - 1);

        vector<int> res;
        for(int i = 0; i < k; i++){
            res.push_back(arr[i]);
        }
```

```
        return res;
    }

    void sort(vector<int>& nums, int l, int r){
        if(l >= r)
            return;

        int ran = (rand() % (r - l + 1)) + l;


        int lt = l + 1, i = lt, gt = r;
        std::swap(nums[ran], nums[l]);
        int pivot = nums[l];

        // nums[0, lt) < pivot
        // nums[lt, i) == pviot
        // nums (gt,r] > pivot
        while(i <= gt){
            if(nums[i] > pivot){
                std::swap(nums[i], nums[gt]);
                gt--;
            }
            else if(nums[i] < pivot){
                std::swap(nums[i], nums[lt]);
                lt++;
                i++;
            }
            else{
                i++;
            }
        }

        // for(int i : nums){
        //     cout << i << " ";
        // }
        // cout << endl;
        // printf("lt=%d gt=%d pivot=%d\n",lt,gt,pivot);

        sort(nums, l, lt - 1);
        sort(nums, gt + 1, r);

    }
```

剑指 Offer 51. 数组中的逆序对

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

示例 1:

输入: [7,5,6,4]
输出: 5

限制:

0 <= 数组长度 <= 50000

来源：力扣（LeetCode）

解法:

暴力（超时） 时间复杂度O(N) 空间复杂度O(1)

```
int res = 0;
    int reversePairs(vector<int>& nums) {
        for(int i = 0; i < nums.size() - 1; i++){
            for(int j = i + 1; j < nums.size(); j++){
                if(nums[i] > nums[j])
                    res++;
            }
        }

        return res;
    }
```

利用归并排序计算逆序对

时间复杂度为O(logN) 空间复杂度O(N)

更快的依据:

归并排序的时间复杂度为O(logN)

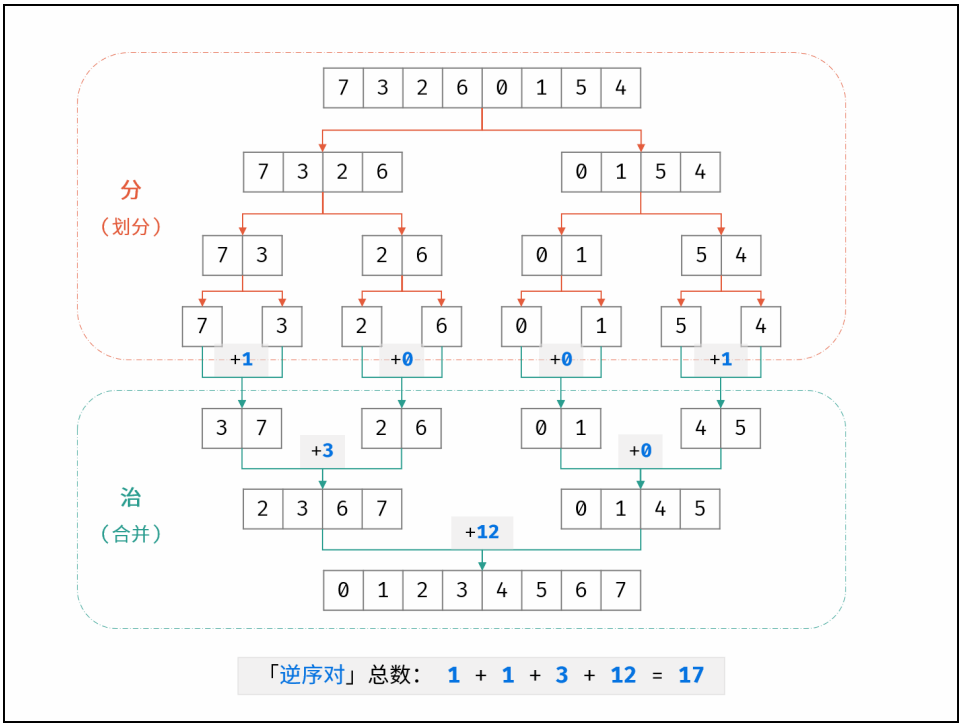
逆序对计算只发生在跨两个有序数组时，不用考虑有序子数组内部

逆序对计算公式：mid - i + 1，原因：当`nums[i] > nums[j]`时,由于有序递增，故`num[i,mid]`内元素都大于`num[j]`，对`nums[j]`有mid - i + 1个逆序对。

```
    int cnt = 0;
    int reversePairs(vector<int>& nums) {
        vector<int> tmp(nums.size());
        mergeSort(nums,0, nums.size() - 1,tmp);
        return cnt;
    }

    void mergeSort(vector<int>& nums, int l, int r,vector<int>& tmp){
        if(l >= r)
            return;
        int mid = l + (r - l) / 2;
        mergeSort(nums,l,mid,tmp);
        mergeSort(nums,mid + 1,r,tmp);
        if(nums[mid] <= nums[mid + 1])
            return;
        mergeTwoArrs(nums,l,mid,r,tmp);
    }
```

```
    void mergeTwoArrs(vector<int>& nums, int l, int mid, int r, vector<int>&
tmp){
        for(int k = l; k <= r; k++)
            tmp[k] = nums[k];
        int i = l, j = mid + 1;
        for(int k = l; k <= r; k++){
            if(i > mid && j <= r)
                nums[k] = tmp[j++];
            else if(j > r && i <= mid)
                nums[k] = tmp[i++];
            else if(tmp[i] <= tmp[j])
                nums[k] = tmp[i++];
            else{
                nums[k] = tmp[j++];
                cnt += (mid - i + 1);
            }
        }

    }
```

演示：



数学

剑指 Offer 15. 二进制中1的个数

o(n)解法

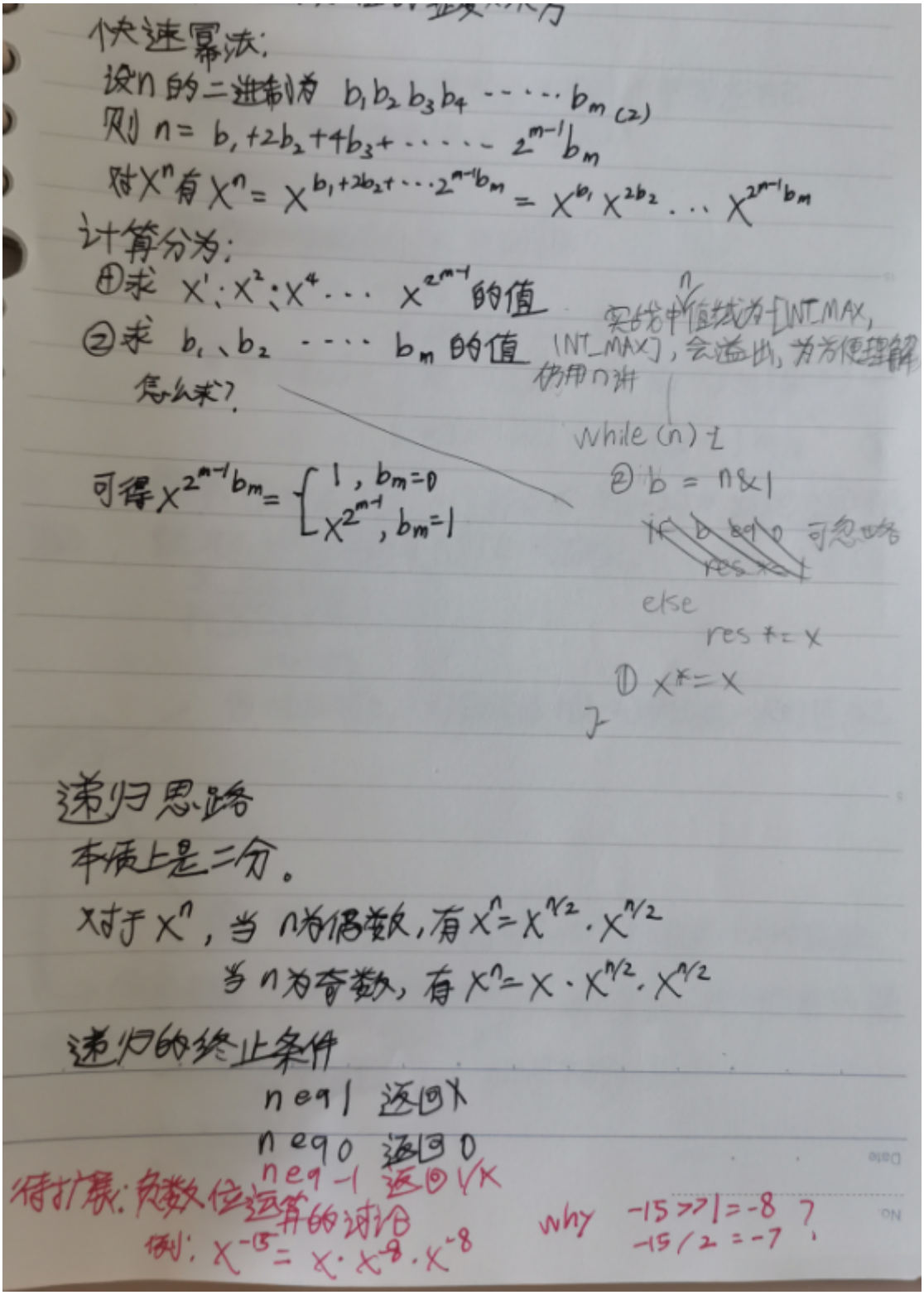
```
  int hammingweight(uint32_t n) {
        int res = 0;
        while(n){
            res += 1;
            n = n & (n - 1);
        }
        return res;
    }
```

o(logn)解法

```
int hammingweight(uint32_t n) {
        int cnt = 0;
        int run = 0;
        while(n){
            n = n & (n - 1);     //每次都会把最末位的1消掉，不管1有多远，故相比每次右移一位找1的方法更高效
            cnt++;
            run++;
        }
        return cnt;
    }
```

剑指 Offer 16. 数值的整数次方

思路:



迭代:

```
double myPow(double x, int n) {
        long m = n; //坑爹的是 这题测试用例也很大，不得不用long来装
        if(m < 0){
            x = 1 / x;
            m = -m;
        }
        double res = 1;
        while(m){
            int b = m & 1;
```

```
            if(b == 1){
                res *= x;
            }
            x *= x;
            m >>= 1;
        }

        return res;
    }
```

递归：

```
double Pow(double x, long m){
        if(m == 0)
            return 1;
        if(m == 1)
            return x;
        double halfpow = Pow(x, m / 2);
        if(m % 2){
            return x * halfpow * halfpow;
        }
        else{
            return halfpow * halfpow;
        }
    }
    double myPow(double x, int n) {
        long m = n;
        if(m < 0){
            x = 1 / x;
            m = -m;
        }
        return Pow(x,m);
    }
```

剑指 Offer 43. 1～n 整数中 1 出现的次数💀

> 输入一个整数 n ，求1～n这n个整数的十进制表示中1出现的次数。

例如，输入12，1～12这些整数中包含1 的数字有1、10、11和12，1一共出现了5次。

示例 1：

输入：n = 12
输出：5
示例 2：

输入：n = 13
输出：6

限制：

1 <= n < 2^31

思路：

n 值可达 2 ^ 31，直接对1到n逐位遍历计数显然会TLE。

一种比较好的思路是，将数字个位，十位，百位，。。。。最高位的1个数逐步累加起来。

设数字n是个x位数，组成为$n_x n_{x-1}...n_2 n_1$。

称n_i为当前位，记为cur

$n_x n_{x-1}..n_{i+1}$为高位，记为high

$n_{i-1} n_{i-2}....n_1$为低位,记为low

10^i称为位因子，记为digit

①当cur = 0时，1的个数只由高位决定，为high x digit，原因是当cur = 0时，对于高位high ∈ [0, $n_x n_{x-1}...n_{i+1}$），high从0开始每次加1均会使得cur位上1的统计数增加digit次。

例如1230567, 设cur为第4位0，则位因子为10^3 = 1000，出现1的范围为：

0001000 - 1221999

对于high = 0, 1, 2 ... 122，每个元素都有1000种情况，共有123000种情况，而当high = 123时，没有配对的cur出现1的情况，故总共123000个情况

②当cur = 1时，1的个数由高位，低位共同决定，为high x digit + low + 1, 原因是当cur = 1时，对于高位high ∈ [0, $n_x n_{x-1}...n_{i+1}$），high从0开始每次加1均会使得cur位上1的统计数增加digit次。而当high = $n_x n_{x-1}...n_{i+1}$时，cur出现1的情况和low有关， 对于low ∈ [0, $n_i n_{i-1}...n_1$], low从0开始每次加1都会使得cur为1的统计增加1次。

例如1231567，设cur为第4位0，则位因子为10^3 = 1000，出现1的范围为：

0001000 - 1231567

对于high = 0,1,2, ... 122, 共有123000种情况，而当high固定为123时，low = 0,1,... 567每个元素对应1种情况，共568种情况，总共为123568个情况

③当cur=[2, 9]，高位和低位的讨论同②，但是low恒定有digit种情况，因此只需要讨论高位，cur上出现1的情况的计算公式为(high + 1) * digit

例如1234567设cur为第4位0，则位因子为10^3 = 1000，出现1的范围为：

0001000 - 1231999,

对于high = 0,1,2, ... 122, 共有123000种情况，而当high固定为123时，low = 0,1,... 999每个元素对应1种情况，共1000种情况，总共为124000个情况

讨论完cur的情况，就可以写代码了：

```
int countDigitOne(int n) {
        //设数的格式为abcde...
        int res = 0;
        //高位 当前位 低位 位因子
        int high = n / 10, cur = n % 10, low = 0;
        long digit = 1;
        //从低到高计算每个位的1个数
        //终止条件 high == 0 或 cur == 0, 因为要考虑只有high = 0 但 cur != 0的情况 比如只有1位数
        while(high != 0 || cur != 0){
            if(cur == 0) res += high * digit;
            else if(cur == 1) res += high * digit + low + 1;
            else res += (high + 1) * digit;

            low = cur * digit + low;
            cur = high % 10;
            high /= 10;
            digit *= 10;

        }

        return res;
    }
```

剑指 Offer 44. 数字序列中某一位的数字💀

数字以0123456789101112131415...的格式序列化到一个字符序列中。在这个序列中，第5位（从下标0开始计数）是5，第13位是1，第19位是4，等等。

请写一个函数，求任意第n位对应的数字。

示例 1：

输入：n = 3
输出：3
示例 2：

输入：n = 11
输出：0

限制：

0 <= n < 2^31

思路：

这题实际上给自然数分区间：

范围	位数	数字个数	数位个数
[0, 9]	1	9	9
[10,99]	2	90	180
[100,999]	3	900	2700
[1000,9999]	4	9000	36000
...			
[a,b]	digit	$9 * 10^{digit-1}$	$9 * 10^{digit-1} * digit$

根据上表的规律，可以发现解题的思路就是：

①找到n落在哪个区间

②找到n落在该区间的哪个自然数上

③找到n落在自然数上哪一位

设位数为`digit`，数字个数为`cnt`,则数位个数就是`cnt * digit`。

根据如上思路，代码如下：

```
int findNthDigit(int n) {
        //数字位数 数字个数 区间起始数
        long digit = 1;
        long cnt = 9;
        long start = 1;
        //找出目标数字所在区间
        while(n >= cnt * digit){
            n -= cnt * digit;
            start += cnt;
            cnt *= 10;
            digit++;
        }

        //找出目标数字
        int target = start + (n - 1) / digit;
        //找出求的是第几位，设为offset,最高位从1开始计数
        int offset = n % digit;
        //如果偏移量为0，说明n落在target个位
        if(offset == 0)
            return target % 10;
}
```

剑指 Offer 45. 把数组排成最小的数💀

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

示例 1:

输入: [10,2]
输出: "102"
示例 2:

输入: [3,30,34,5,9]
输出: "3033459"

提示:

0 < nums.length <= 100
说明:

输出结果可能非常大，所以你需要返回一个字符串而不是整数
拼接起来的数字可能会有前导 0，最后结果不需要去掉前导 0

解：与其说是排序，不如说这题是数学，因为排序规则非常难想，此外还要证明。

本身是自定义一个排序规则，使得数组满足题目条件，描述如下：

设满足题目条件的序列为A, 则设A中各元素存在 x <' y， 使得有 x + y <= y + x 且 x 排在 y 前面。

例如[3,30,34,5,9]排序后的答案[30,3,34,5,9] 我们说其有 30 <' 3 <' 34 <' 5 <' 9。

根据这个规则来写排序。

参考：https://leetcode.cn/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/solution/yukiyama-jian-zhi-by-yukiyama-r29u/

冒泡排序

```
    //自定义<'比较
    bool meet_cond(string &lhs, string &rhs){
        return lhs + rhs <= rhs + lhs;
    }
    string minNumber(vector<int>& nums) {
        vector<string> numstr;
        for(int i : nums)
            numstr.push_back(to_string(i));

        for(int i = 0; i < nums.size(); i++){
            for(int j = 1; j < nums.size(); j++){
                //排序对象为numsstr[j]
                if(!meet_cond(numstr[j - 1],numstr[j]))
                    std::swap(numstr[j - 1], numstr[j]);
            }
        }
```

```
        string ans;
        for(string str : numstr)
            ans += str;
        return ans;
    }
```

快排

```
    //自定义<'比较
    bool meet_cond(string &lhs, string &rhs){
        return lhs + rhs <= rhs + lhs;
    }
    string minNumber(vector<int>& nums) {
        vector<string> numstr;
        for(int i : nums)
            numstr.push_back(to_string(i));

        quickSort(numstr,0,nums.size() - 1);

        string ans;
        for(string str : numstr)
            ans += str;
        return ans;
    }

    void quickSort(vector<string> &nums, int l, int r){
        if(l >= r)
            return;
        int mid = partiton(nums, l, r);
        quickSort(nums,l, mid - 1);
        quickSort(nums,mid + 1, r);
    }

    int partiton(vector<string> &nums, int l, int r){
        //这段随机化其实可有可无
        // int ran = (rand() % (r - l + 1)) + l;
        // std::swap(nums[l],nums[ran]);

        int lt = l + 1, gt = r;
        string &pivot = nums[l];
        while(true){
            while(lt < r && meet_cond(nums[lt], pivot)) lt++;
            while(l < gt && meet_cond(pivot, nums[gt])) gt--;
            if(lt >= gt)
                break;
            std::swap(nums[lt],nums[gt]);

        }

        std::swap(pivot,nums[gt]);
        return gt;
    }
```

如何证明这样排个序就能AC呢，可以从传递性证明和反证法两方面来证，这里放传递性证明：

```
字符串 xy < yx ，yz < zy ，需证明 xz < zx 一定成立。

设十进制数 x，y，z 分别有 a，b，c 位，则有：
（左边是字符串拼接，右边是十进制数计算，两者等价）
xy = x * 10^b + y
yx = y * 10^a + x

则 xy < yx 可转化为：
x * 10^b + y < y * 10^a + x
x (10^b - 1) < y (10^a - 1)
x / (10^a - 1) < y / (10^b - 1)      ①

同理， 可将 yz < zy 转化为：
y / (10^b - 1) < z / (10^c - 1)      ②

将 ① ② 合并，整理得：
x / (10^a - 1) < y / (10^b - 1) < z / (10^c - 1)
x / (10^a - 1) < z / (10^c - 1)
x (10^c - 1) < z (10^a - 1)
x * 10^c + z < z * 10^a + x
∴  可推出 xz < zx ，传递性证毕
```

剑指 Offer 60. n个骰子的点数

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

示例 1:

输入: 1
输出: [0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]
示例 2:

输入: 2
输出:
[0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.05556,0.02778]

限制：

1 <= n <= 11

二维dp

注意范围：n个骰子的点数范围为[n , 6n]， 取值个数为 5n + 1

`dp[i][j]` 定义：`i` 个骰子总和为 `j` 的情况总数

初始化： 1个骰子情况下，点数1-6出现的情况都是1

递推逻辑：分别讨论第n个骰子点数从1到6共六种情况，n个骰子总点数为s的情况总数是这六种情况下，【前n - 1个骰子的和】加上【第n个骰子的值】刚好等于s的情况总和，即有

```
f(n,s) = f(n - 1, s - 1) + f(n - 1, s - 2), f(n - 1, s - 3) + f(n - 1, s - 4) +
f(n - 1, s - 5) + f(n - 1, s - 6)
```

```
vector<double> dicesProbability(int n) {
        vector<vector<int>> dp(n + 1, vector<int>(6 * n + 1) );
        vector<double> ans(5 * n + 1);
        double allcases = pow(6,n);
        for(int i = 1; i <= 6; i++)
            dp[1][i] = 1;

        for(int i = 1; i <= n; i++){
            for(int j = i; j <= 6 * i; j++){
                for(int k = 1; k <= 6; k++){
                    if(j - k >= 1)
                        dp[i][j] += dp[i - 1][j - k];
                }
                if(i == n)
                    ans[j - i] = dp[i][j] / allcases;
            }
        }


        return ans;
    }
```

剑指 Offer 62. 圆圈中最后剩下的数字

0,1,⋯,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字（删除后从下一个数字开始计数）。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1：

输入: n = 5, m = 3
输出: 3
示例 2:

输入: n = 10, m = 17
输出: 2

限制:

1 <= n <= 10^5
1 <= m <= 10^6

思路:

约瑟夫环的问题，第一眼就想用队列模拟，但是看到n,m的范围，放弃了。

动态规划参考K神的解法:

dp[i]定义：[i, m]问题的解

初始化: dp[i] = 0

递推方程: dp[i] = (dp[i - 1] + m) % i;

```
int lastRemaining(int n, int m) {
    vector<int> dp(n + 1);
    dp[1] = 0;
    for(int i = 2; i <= n; i++){
        dp[i] = (dp[i - 1] + m) % i;
    }
    return dp[n];
}c
```

由于`dp[i]`只由`dp[i - 1]`获得且前面的dp值不会再被用到，因此可以把空间复杂度简化到O(1);

```
int lastRemaining(int n, int m) {
    vector<int> dp(n + 1);
    int x = 0;
    for(int i = 2; i <= n; i++){
        x = (x + m) % i;
    }
    return x;
}
```

位运算

剑指 Offer 56 - I. 数组中数字出现的次数

一个整型数组 `nums` 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

示例 1：

```
输入：nums = [4,1,4,6]
输出：[1,6] 或 [6,1]
```

示例 2：

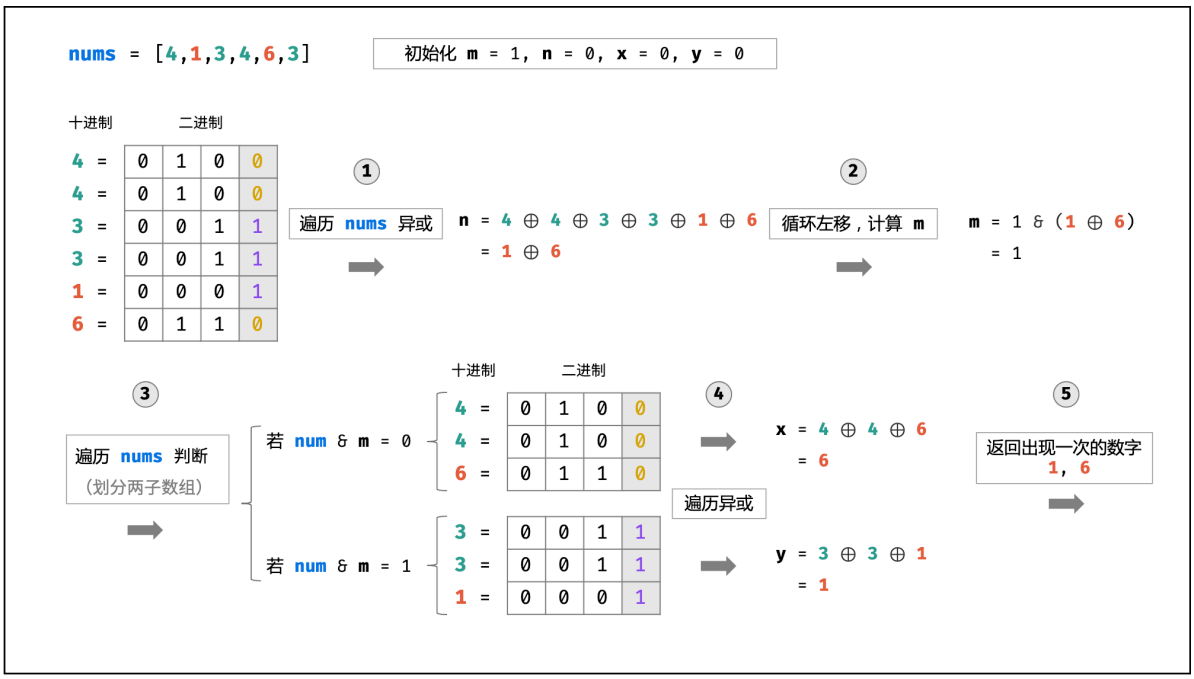
```
输入：nums = [1,2,10,4,1,4,3,3]
输出：[2,10] 或 [10,2]
```

限制：

- `2 <= nums.length <= 10000`

思路：参考了K神的方法。

```
vector<int> singleNumbers(vector<int>& nums) {
        //步骤1 将所有元素异或运算求出两个目标数的异或结果 n = x⊕y
        int n = 0, m = 1, x = 0, y = 0;
        for(int i : nums){
            n ^= i;
        }
        //步骤2 找出x和y所在两个子数组的分组掩码m
        while((n & m) == 0)
            m <<= 1;
        //步骤3 根据分组掩码将数组二分，分别对两个数组遍历异或找出x和y
        for(int i : nums){
            if((i & m) == 0){
                x ^= i;
            }
            else{
                y ^= i;
            }
        }

        return vector<int>{x,y};
    }
```



剑指 Offer 56 - II. 数组中数字出现的次数 II

在一个数组 nums 中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

示例 1：

输入：nums = [3,4,3,3]
输出：4
示例 2：

输入：nums = [9,1,7,9,7,9,7]
输出：1

限制：

1 <= nums.length <= 10000
1 <= nums[i] < 2^31

状态机法

定义一个运算@，使得对于任意输入a，存在a@a@a = 0, 其设计原理类似于异或运算，都是位运算，只不过异或的状态是两种0和1，而这题我们需要定义3种状态0,1,2, 一个二进制位是不够表达的，因此我们需要用两个二进制位来表达三种状态，设高位为A，低位为B，输出为C，其真值表如下：

```
// 画出关于 A 的卡诺图（AB为11的结果是不重要的，用 x 表示）：
        //  AB\C |  0  |  1
        //  =================
```

```
        //    00 |  0  |  0
        //    01 |  0  |  1        ====> 得到 A = BC + AC'
        //    11 |  x  |  x
        //    10 |  1  |  0

        //  画出关于 B 的卡诺图
        //  AB\C |  0  |  1
        //  =================
        //    00 |  0  |  1
        //    01 |  1  |  0        ====> 得到 B = BC' + A'B'C
        //    11 |  x  |  x
        //    10 |  0  |  0
```

根据真值表设计@，然后将数组的每个数执行@运算就行，比如：3@3@3@5 = 5

```
int singleNumber(vector<int>& nums) {
        int A = 0, B = 0;
        for (int C : nums) {
            int tmp = A;
            A = (B & C) | (A & ~C);
            B = (B & ~C) | (~tmp & ~B & C);
        }
        return B;
    }
```

剑指 Offer 65. 不用加减乘除做加法

写一个函数，求两个整数之和，要求在函数体内不得使用 “+”、“-”、“*”、“/” 四则运算符号。

示例:

输入: a = 1, b = 1
输出: 2

提示:

a, b 均可能是负数或 0
结果不会溢出 32 位整数

思路:

看了K神题解才知道自己应该列个表格讨论所有运算情况然后提取公式。。

讨论a和b的每一个对应的二进制位，其对应位的加法值相当于异或运算，用`a ^ b`计算, 而是否进位则可用与运算检测，由于进位要作用在更高一位, 为`(a & b) >> 1`。

加法和即为【进位】和【对应位加法值】相异或的结果，终止条件即不再出现进位的时候。

8.30 ADD:

别TM想太多了，这题本质是模拟全加器，算是计算机组成原理的锅。

全加器的原理是，将两个加数从各自个位分别相加，产生的进位用于下一位的运算，一直加到最高位为止。

至于为什么用unsigned，是因为表示进位的电平只有0和1两个状态，没有负状态。

```
c-->+-----+---> ((a & b) << 1)
a-->|全加器|
    | x位 |
b-->+-----+ --> a ^ b


                    -->+-----+
                    -->|全加器|
                       | 2位 |
          -->+-----+-->+-----+
          -->|全加器|
             | 1位 |
-->+-----+-->+-----+-->
-->|全加器|
   | 0位 |
-->+-----+-->
```

演示：

```
        a = 20          0001 0100
        b = 17          0001 0001
    tmp = a ^ b         0000 0101
carry = (a & b) << 1    0010 0000
tmp ^ carry             0010 0101(37)
```

```
int add(int a, int b) {
        while(b != 0){
            int tmp = a ^ b;
            //C++ 不支持负数左移 需要用unsigned
            int carry = ((unsigned int)(a & b) << 1);
            a = tmp;
            b = carry;
        }
        return a;
    }
```

其他

剑指 Offer 64. 求1+2+...+n

求 1+2+...+n ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

示例 1:

输入: n = 3
输出: 6
示例 2:

输入: n = 9
输出: 45

限制:

1 <= n <= 10000

思路:

纯智力题，考验候选人的发散思维。

方法一：用构造函数模拟循环

```
int res = 0;
int n_ = 0;
class Tmp{
    public:
        Tmp(){
            //这里res会变成垃圾值 都已经初始化过了不知道为什么 又得重新初始化一次
            cout << res << " " << n_ << endl;
            res += (1 + n_);
            cout << res << "\n\n";
        }
};
class Solution {
public:
    int sumNums(int n) {
        res = 0;
        n_ = n;
        Tmp *pt = new Tmp[n];
        delete[] pt;
        return res >> 1;
    }
};
```

方法二：用一个函数模拟递归 另一个函数模拟终止 用函数指针动态调用

```
typedef int (*pf)(int);
int sum_terminate(int n){
    return 0;
}
int sum_do(int n){
```

```
    pf pfun[2] = {sum_terminate, sum_do};
    return pfun[!!n](n - 1) + n;
}
class Solution {
public:
    int sumNums(int n) {
        return sum_do(n);
    }
};
```

总结

涉及列表格讨论情况的题目: