[剑指 Offer 03. 数组中重复的数字](https://leetcode-cn.com/problems/shu-zu-zhong-zhong-fu-de-shu-zi-lcof/)

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

**限制：**

2 <= n <= 100000

#include <vector>

#include <algorithm>

#include <set>

using namespace std;

class Solution {

public:

    int findRepeatNumber(vector<int>& nums) {

        //1.排序

        sort(nums.begin(), nums.end());

        //2.遍历找出

        for (int i = 0; i < nums.size() - 2; i++) {

            if(nums[i]==nums[i + 1])

            {

                return nums[i];

            }

        }

        return -1;

    }

};

class Solution2 {

public:

    int findRepeatNumber(vector<int>& nums) {

        set<int> myset;

//unordered\_set<int> myset;

//用unordered更快

        for (size\_t i = 0; i < nums.size(); i++) {

            if(!myset.insert(nums[i]).second)

            {

                return nums[i];

            }

        }

        return -1;

    }

};

int main(){

    vector<int> nums;

    int n, temp;

    scanf("%d", &n);

    for(int i = 0; i < n; ++i){

        scanf("%d", &temp);

        nums.push\_back(temp);

    }

    Solution s;

    int res = s.findRepeatNumber(nums);

    printf("res = %d", res);

    getchar();

    getchar();

    return 0;

[剑指 Offer 04. 二维数组中的查找](https://leetcode-cn.com/problems/er-wei-shu-zu-zhong-de-cha-zhao-lcof/)

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。

class Solution{

public:

    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target){

        int i = matrix.size() - 1;

        int j = 0;

        while(i >= 0 && j < matrix[0].size()){

            if(matrix[i][j] == target)

                return true;

            if(matrix[i][j] > target)

                i--;

            else

                j++;

        }

        return false;

    }

};

// 时间复杂度：O(n+m)。访问到的下标的行最多增加 n 次，列最多减少 m 次，因此循环体最多执行 n + m 次。

// 空间复杂度：O(1)。

int main(){

    vector<vector<int>> matrix = {{1, 4, 7, 11, 15}, {2, 5, 8, 12, 19},

    {3, 6, 9, 16, 22}, {10, 13, 14, 17, 24}, {18, 21, 23, 26, 30}};

    Solution\* s = new Solution();

    printf("%d", s->findNumberIn2DArray(matrix, 5));

    return 0;

}

[剑指 Offer 05. 替换空格](https://leetcode-cn.com/problems/ti-huan-kong-ge-lcof/)

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."

**限制：**

0 <= s 的长度 <= 10000

#include <string>

#include <iostream>

#include <windows.h>

using namespace std;

class Solution{

public:

    string replaceSpace(string s){

        string res;

        for(auto c : s){

            if(c == ' ')

                res += "%20";

            else

                res += c;

        }

        return res;

    }

};

// 时间复杂度：O(n)。遍历字符串 s 一遍。

// 空间复杂度：O(n)。额外创建字符数组，长度为 s 的长度的 3 倍

int main(){

    string str;

    Solution s;

    str.resize(100);

    scanf("%[^\n]", &str[0]);

    // getline(cin, str);

    //cin接收输入时，:>> 是会过滤掉不可见字符（如 空格 回车，TAB 等）；从第一个非空白符开始读，知道空白符或文件结束为止

    //可以用getline：

    str = s.replaceSpace(str);

    printf("%s", str.c\_str());

    // cout << s.replaceSpace(str) << endl;

    system("pause");

    return 0;

}

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

struct ListNode

{

    int val;

    ListNode\* next;

    ListNode(int x) : val(x), next(NULL){}

};

// 思路一：反转

class Solution {

public:

    vector<int> reversePrint(ListNode\* head) {

        vector<int> res;

        while (head){

            res.push\_back(head->val);

            head = head->next;

        }

        reverse(res.begin(), res.end());

        return res;

    }

};

// 复杂度分析

// 时间复杂度：O(n)，reverse() 的时间复杂度为O(n)，遍历了一遍数组，复杂度也为O(n)。

// 空间复杂度：O(n)，使用了额外的 res。

// 思路二：递归

class Solution {

public:

    vector<int> res;

    vector<int> reversePrint(ListNode\* head) {

        if (!head) return res;

        reversePrint(head->next);

        res.push\_back(head->val);

        return res;

    }

};

// 复杂度分析

// 时间复杂度：O(n)，递归 n 次，时间间复杂度为O(n)，递归函数中的操作时间复杂度为O(1)，总时间复杂度为 O(n)\times O(1)=O(n)×O(1)=O(n)。

// 空间复杂度：O(n)，递归将占用链表长度的栈空间。

// 思路三：堆栈

class Solution {

public:

    vector<int> res;

    vector<int> reversePrint(ListNode\* head) {

        stack<int> st;

        while(head){// push

            st.push(head->val);

            head = head->next;

        }

        while(!st.empty()){ // pop

            res.push\_back(st.top());

            st.pop();

        }

        return res;

    }

};

// 复杂度分析

// 时间复杂度：O(n)，push 的间复杂度为 O(n)，pop 的间复杂度为O(n)。

// 空间复杂度：O(n)，使用了额外的 res 和 堆栈。

class Solution {

public:

    vector<int> res;

    vector<int> reversePrint(ListNode\* head) {

        //方法4：改变链表结构

        ListNode \*pre = nullptr;

        ListNode \*next = head;

        ListNode \*cur = head;

        while(cur){

            next = cur->next;//保存当前结点的下一个节点

            cur->next = pre;//当前结点指向前一个节点，反向改变指针

            pre = cur;//更新前一个节点

            cur = next;//更新当前结点

        }

        while(pre){//上一个while循环结束后，pre指向新的链表头

            res.push\_back(pre->val);

            pre = pre->next;

        }

        return res;

    }

};

[剑指 Offer 07. 重建二叉树](https://leetcode-cn.com/problems/zhong-jian-er-cha-shu-lcof/)

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

例如，给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

3

/ \

9 20

/ \

**限制：**

0 <= 节点个数 <= 5000

15 7

struct TreeNode {

     int val;

     TreeNode \*left;

     TreeNode \*right;

     TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

class Solution {

public:

    vector<int> Preorder ;

    map<int,int> dic;

    TreeNode\* build(int pre\_root, int in\_left, int in\_right){

        //如果左边界大于右边界说明到过了叶子

        if(in\_left > in\_right)

            return NULL;

        //pre\_root 是先序里面的索引 ！！

        TreeNode\* root = new TreeNode(Preorder[pre\_root]);

        //获取先序中的节点在中序中的节点， 即index 左边就是这节点的左子树，index右边就是节点的右子树

        int index = dic[Preorder[pre\_root]];

        //当前节点左树即为先序索引+1 （没了话会在下一次迭代返回NULL）

        //我的理解，节点传入的是前序的节点，而left和right传入的位置则是中序，这是因为根据dic能够通过前序节点的位置找到中序节点的位置，

        //这样我们通过前序来储存节点，而通过中序来定义位置，进行判定；

        root->left = build(pre\_root + 1, in\_left, index - 1);

        //当前节点右树即为 根结点在前序中的索引+左树所有节点数（即节点在中序中的索引）-左边界+1 ，下一次的左边界为根在中序的索引+1

        root->right = build(pre\_root + index - in\_left + 1, index + 1, in\_right);

        return root;

    }

    TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) {

        //赋值至外部变量

        Preorder = preorder;

        //使用map映射inorder的值和索引，提高找到索引效率

        for(int i = 0; i < inorder.size(); i++){

            dic[inorder[i]] = i;

        }

        return build(0, 0, preorder.size() - 1);

    }

};

// 时间复杂度：O(n)。对于每个节点都有创建过程以及根据左右子树重建过程。

// 空间复杂度：O(n)。存储整棵树的开销。

//后序+中序重构

//[9, 15, 7, 20 ,3]

//[9, 3, 15, 20, 7]

class Solution{

    vector<int> Postorder;

    map<int, int> mp;

public:

    TreeNode\* build(int post\_root, int left, int right){

        if(left > right)

            return NULL;

        TreeNode\* root = new TreeNode(Postorder[post\_root]);

        int index = mp[Postorder[post\_root]];

        root->right = build(post\_root - 1, index + 1, right);

        root->left = build(post\_root - right + index - 1, left, index - 1);

    }

    TreeNode\* buildTree(vector<int>& postorder, vector<int>& inorder){

        Postorder = postorder;

        for(int i = 0; i < inorder.size(); ++i){

            mp[inorder[i]] = i;

        }

    }

};

[剑指 Offer 09. 用两个栈实现队列](https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/)

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
* 最多会对 appendTail、deleteHead 进行 10000 次调用

class CQueue {

    stack<int> stack1,stack2;

public:

    CQueue() {

        while (!stack1.empty()) {

            stack1.pop();

        }

        while (!stack2.empty()) {

            stack2.pop();

        }

    }

    void appendTail(int value) {

        stack1.push(value);

    }

 int deleteHead() {

        // 如果第二个栈为空

        if (stack2.empty()) {

            while (!stack1.empty()) {

                stack2.push(stack1.top());

                stack1.pop();

            }

        }

        if (stack2.empty()) {

            return -1;

        } else {

            int deleteItem = stack2.top();

            stack2.pop();

            return deleteItem;

        }

    }

};

/\*\*

 \* Your CQueue object will be instantiated and called as such:

 \* CQueue\* obj = new CQueue();

 \* obj->appendTail(value);

 \* int param\_2 = obj->deleteHead();

 \*/

[剑指 Offer 10- I. 斐波那契数列](https://leetcode-cn.com/problems/fei-bo-na-qi-shu-lie-lcof/)

写一个函数，输入 n ，求斐波那契（Fibonacci）数列的第 n 项。斐波那契数列的定义如下：

F(0) = 0,   F(1) = 1

F(N) = F(N - 1) + F(N - 2), 其中 N > 1.

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

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：**n = 2

**输出：**1

**示例 2：**

**输入：**n = 5

**输出：**5

**提示：**

* 0 <= n <= 100

class Solution {

public:

    int fib(int n) {

        if(n==0) return 0;

        std::vector<int> dp(n+1);

        dp[0] = 0;

        dp[1] = 1;

        for(int i=2; i<=n;  i++){

            dp[i] = (dp[i-1] + dp[i-2]) % 1000000007 ;

        }

        return dp[n];

    }

    int fib2(int n){

        int f = 0,  g = 1;

        while(n--){

            g = g + f;

            f = g - f;

        }

        return g;

    }

    //O(n),空间为O(1)

};

[剑指 Offer 10- II. 青蛙跳台阶问题](https://leetcode-cn.com/problems/qing-wa-tiao-tai-jie-wen-ti-lcof/)

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

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

class Solution {

public:

    int numWays(int n) {

        vector<int> dp(n + 1, 1);

        for(int i = 2; i <= n; ++i){

            dp[i] = (dp[i - 1] + dp[i - 2]) % 1000000007;

        }

        return dp[n];

    }

};

[剑指 Offer 11. 旋转数组的最小数字](https://leetcode-cn.com/problems/xuan-zhuan-shu-zu-de-zui-xiao-shu-zi-lcof/)

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一个旋转，该数组的最小值为1。

**示例 1：**

**输入：**[3,4,5,1,2]

**输出：**1

**示例 2：**

**输入：**[2,2,2,0,1]

**输出：**0

//二分法，时间复杂度O(log(n)),空间复杂度O(1);

class Solution{

public:

    int minArray(vector<int> numbers){

        int size = numbers.size();

        int left = 0;

        int right = size-1;

        if(size==0) return 0;

        if(size==1) return numbers[0];

        while(left  <  right){

            int mid = (left + right) >> 1;

            if(numbers[mid] > numbers[right])

                left = mid + 1;

            else if(numbers[mid] == numbers[right])

                right--;

            else

                right = mid;

        }

        return numbers[left];

    }

};

[剑指 Offer 12. 矩阵中的路径](https://leetcode-cn.com/problems/ju-zhen-zhong-de-lu-jing-lcof/)

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如，在下面的3×4的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。

[["a","**b**","c","e"],  
["s","**f**","**c**","s"],  
["a","d","**e**","e"]]

但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

**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：**

* 1 <= board.length <= 200
* 1 <= board[i].length <= 200

class Solution{

public:

    bool exist(vector<vector<char>>& board, string word){

        for(int i = 0; i < board.size(); ++i){

            for(int j = 0; j < board[0].size(); ++j){

                if(dfs(board, word, i, j, 0))

                    return true;

            }

        }

        return false;

    }

    bool dfs(vector<vector<char>>& board, string word, int i, int j, int k){

        if(i >= board.size() || i < 0 || j >= board[0].size() || j < 0 || board[i][j] != word[k])

            return false;

        if(k == word.size() - 1)

            return true;

        char temp = board[i][j];

        board[i][j] = '/';

        bool res = dfs(board, word, i + 1, j, k + 1) || dfs(board, word, i - 1, j, k + 1)

                    || dfs(board, word, i, j + 1, k + 1) || dfs(board, word, i, j - 1, k + 1);

        board[i][j] = temp;

        return res;

    }

};

// 复杂度分析：

// M,N 分别为矩阵行列大小， K 为字符串 word 长度。

// 时间复杂度 O(3^KMN)：最差情况下，需要遍历矩阵中长度为 K 字符串的所有方案，时间复杂度为 O(3^K)

//  矩阵中共有 MN 个起点，时间复杂度为 O(MN) 。

// 方案数计算： 设字符串长度为 K ，搜索中每个字符有上、下、左、右四个方向可以选择，舍弃回头（上个字符）的方向，剩下 33 种选择，因此方案数的复杂度为 O(3^K)

//  ) 。

// 空间复杂度 O(K) ： 搜索过程中的递归深度不超过 K ，因此系统因函数调用累计使用的栈空间占用 O(K)（因为函数返回后，系统调用的栈空间会释放）。最坏情况下 K = MN，递归深度为 MN ，此时系统栈使用 O(MN)的额外空间。

[剑指 Offer 13. 机器人的运动范围](https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/)

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

**提示：**

* 1 <= n,m <= 100
* 0 <= k <= 20

 int digit\_sum(int i){

        int sum = 0;

        while(i>0){

            sum += i%10;

            i/=10;

        }

        return sum;

}

//方法一，深度优先遍历

class Solution {

public:

    int dfsSolver(int i, int j, int m, int n, int k, vector<vector<bool>>& vislted){

        if(i>=m||j>=n||digit\_sum(i)+digit\_sum(j)>k||vislted[i][j]==true) return 0;

        vislted[i][j] = true;

        return 1+dfsSolver(i+1,j,m,n,k,vislted)+dfsSolver(i,j+1,m,n,k,vislted);

    }

    int movingCount(int m, int n, int k) {

        vector<vector<bool>> vislted(m,vector<bool>(n));

        int res = dfsSolver(0,0,m,n,k,vislted);

        return res;

    }

};

// 时间复杂度：O(mn)

// 空间复杂度：O(mn)

// 方法二：广度优先搜索

// 我们将行坐标和列坐标数位之和大于 k 的格子看作障碍物，那么这道题就是一道很传统的搜索题目，我们可以使用广度优先搜索或者深度优先搜索来解决它，本文选择使用广度优先搜索的方法来讲解。

// 那么如何计算一个数的数位之和呢？我们只需要对数 x 每次对 10 取余，就能知道数 x 的个位数是多少，然后再将 x 除 10，这个操作等价于将 x 的十进制数向右移一位，删除个位数（类似于二进制中的 >> 右移运算符），不断重复直到 x 为 0 时结束。

// 同时这道题还有一个隐藏的优化：我们在搜索的过程中搜索方向可以缩减为向右和向下，而不必再向上和向左进行搜索。如下图，我们展示了 16 \* 16 的地图随着限制条件 k 的放大，可行方格的变化趋势，每个格子里的值为行坐标和列坐标的数位之和，蓝色方格代表非障碍方格，即其值小于等于当前的限制条件 k。我们可以发现随着限制条件 k 的增大，(0, 0) 所在的蓝色方格区域内新加入的非障碍方格都可以由上方或左方的格子移动一步得到。而其他不连通的蓝色方格区域会随着 k 的增大而连通，且连通的时候也是由上方或左方的格子移动一步得到，因此我们可以将我们的搜索方向缩减为向右或向下。

class Solution {

public:

    int movingCount(int m, int n, int k) {

        if(k==0) return 1;

        queue<pair<int,int>> Q;

        int dx[2] = {0, 1};

        int dy[2] = {1, 0};

        vector<vector<int>> vis(m, vector<int>(n,0));

        vis[0][0] = 1;

        Q.push(make\_pair(0,0));

        vis[0][0] = 1;

        int res = 1;

        while(!Q.empty()){

            auto [x,y] = Q.front();

            Q.pop();

            for(int i=0;i<2;i++){

                int tx = dx[i] + x;

                int ty = dy[i] + y;

                if(tx >= m || ty>=n || vis[tx][ty] || digit\_sum(tx) + digit\_sum(ty) > k) continue;

                vis[tx][ty] = 1;

                res++;

                Q.push(make\_pair(tx,ty));

            }

        }

        return res;

    }

};

// 时间复杂度：O(mn)，其中 m 为方格的行数，n 为方格的列数。考虑所有格子都能进入，那么搜索的时候一个格子最多会被访问的次数为常数，所以时间复杂度为 O(2mn)=O(mn)。

// 空间复杂度：O(mn)，其中 m 为方格的行数，n 为方格的列数。搜索的时候需要一个大小为 O(mn)的标记结构用来标记每个格子是否被走过。

// 方法三：递推

// 考虑到方法一提到搜索的方向只需要朝下或朝右，我们可以得出一种递推的求解方法。

// 定义 vis[i][j] 为 (i, j) 坐标是否可达，如果可达返回 1，否则返回 0。

// 首先 (i, j) 本身需要可以进入，因此需要先判断 i 和 j 的数位之和是否大于 k ，如果大于的话直接设置 vis[i][j] 为不可达即可。

// 否则，前面提到搜索方向只需朝下或朝右，因此 (i, j) 的格子只会从 (i - 1, j) 或者 (i, j - 1) 两个格子走过来（不考虑边界条件），那么 vis[i][j] 是否可达的状态则可由如下公式计算得到：

// 即只要有一个格子可达，那么 (i, j) 这个格子就是可达的，因此我们只要遍历所有格子，递推计算出它们是否可达然后用变量 ans 记录可达的格子数量即可。

// 初始条件 vis[i][j] = 1 ，递推计算的过程中注意边界的处理。

class Solution {

public:

    int movingCount(int m, int n, int k) {

        if(k==0) return 1;

        vector<vector<int> > vis(m, vector<int>(n, 0));

        int res = 1;

        vis[0][0] = 1;

        for (int i = 0; i < m; ++i) {

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

                if ((i == 0 && j == 0) || get(i) + get(j) > k) continue;

                // 边界判断

                if (i - 1 >= 0) vis[i][j] |= vis[i - 1][j];

                if (j - 1 >= 0) vis[i][j] |= vis[i][j - 1];

                res += vis[i][j];

            }

        }

        return res;

}

};

// 时间复杂度：O(mn)，其中 m 为方格的行数， n 为方格的列数。一共有 O(mn)个状态需要计算，每个状态递推计算的时间复杂度为 O(1)，所以总时间复杂度为 O(mn)。

// 空间复杂度：O(mn)，其中 m 为方格的行数，n 为方格的列数。我们需要 O(mn)大小的结构来记录每个位置是否可达。

[剑指 Offer 14- I. 剪绳子](https://leetcode-cn.com/problems/jian-sheng-zi-lcof/)

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

class Solution {

public:

    int cuttingRope(int n) {

        if(n<=3) return n-1;

        int a = n/3,b = n%3;

        if(b==0) return pow(3,a);

        if(b==1) return pow(3,a-1)\*4;

        return pow(3,a)\*2;

    }

};

//运用完全背包问题去解

int cuttingRope(int n) {

    vector<int> dp(n+1,0);

    dp[1]=1;

    for(int i=2;i<=n;i++){

        for(int j=1;j<i;j++){

            dp[i] = max(dp[i],dp[j]\*dp[i-j]);

        }

        if(i!=n) dp[i] = max(dp[i],i);

    }

    return dp[n];

}

[剑指 Offer 15. 二进制中1的个数](https://leetcode-cn.com/problems/er-jin-zhi-zhong-1de-ge-shu-lcof/)

请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

**示例 1：**

**输入：**00000000000000000000000000001011

**输出：**3

**解释：**输入的二进制串 **00000000000000000000000000001011** 中，共有三位为 '1'。

**示例 2：**

**输入：**00000000000000000000000010000000

**输出：**1

**解释：**输入的二进制串 **00000000000000000000000010000000** 中，共有一位为 '1'。

**示例 3：**

**输入：**11111111111111111111111111111101

**输出：**31

**解释：**输入的二进制串 **11111111111111111111111111111101** 中，共有 31 位为 '1'。

按位判断

这是最直观的方式，依次判断整数的每一位是否为 1 。

判断第 1 位：n & 1

判断第 2 位：n & (1 << 1)

......

判断第32位：n & (1 << 31)

class Solution {

public:

    int hammingWeight(uint32\_t n) {

        int ret = 0;

        for (int i = 0; i < 32; i ++)

            if (n & (1 << i))

                ret ++;

        return ret;

    }

};

N & (N - 1)​

使用这个方法，基于这样一个事实：

一个数 n 与一个比它小 1的数（n - 1）进行与运算（&）之后，结果会消除 n 中最低位的 1.

示例1: 7 & 6

00111

      &   =》 00110

00110

示例2: 8 & 7

01000

      &   =》 00000

00111

可以看到，n&(n−1) 得到的结果，就是将 n 最低位 1，换成 0 之后的值。

根据这个，编码思路就是，每次都使用该运算法则消去 1 ，每运算一次计数器+1，直至 n 为 0 .

class Solution2 {

public:

    int hammingWeight(uint32\_t n) {

        int ret = 0;

        while (n != 0) {

            n &= n-1;

            ret ++;

        }

        return ret;

    }

};

[剑指 Offer 16. 数值的整数次方](https://leetcode-cn.com/problems/shu-zhi-de-zheng-shu-ci-fang-lcof/)

实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

**示例 1:**

**输入:** 2.00000, 10

**输出:** 1024.00000

**示例 2:**

**输入:** 2.10000, 3

**输出:** 9.26100

**示例 3:**

**输入:** 2.00000, -2

**输出:** 0.25000

**解释:** 2-2 = 1/22 = 1/4 = 0.25

**说明:**

* -100.0 < *x* < 100.0
* *n* 是 32 位有符号整数，其数值范围是 [−231, 231− 1] 。

所以我们可以考虑比如我们计算 x^8，就是 x^2 \* x^2 \* x^2 \* x^2，当我们计算出来 x^2 之后就可以只进行三次乘法就可以了，相对于之前的 7 次乘法，时间大大减少了。

也就是 x^n 可以分解成若干个 x^i 的乘积

我们这里使用快速幂进行求解。我们看一下 n 的二进制形式一定是若干个 1 和 0 构成，比如 9 = 1001 = 1\*2^3 + 0\*2^2 + 0\*2^1 + 1\*2^0

x^9 = x^[2^0] \* x^[2^1 \* 0] \* x^[2^2 \* 0] \* x^[2^3];

所以我们可以看出来，每次乘的值都是前一个值的2倍，当 n 对应位为0时跳过

负数幂和正数幂相同，因为除以一个数就相当于乘这个数的倒数。

class Solution{

public:

    double myPow(double x, int n){

        if(x == 1 || n == 0)

            return 1;

        double ans = 1;

        long num = n;

        if(n < 0){

            num = -num;

            x = 1 / x;

        }

        while(num){

            if(num & 1)

                ans \*= x;

            x \*= x;

            num >>= 1;

            printf("%f %ld\n", ans, num);

/\*

在printf中%d用于int或者比int小的整数类型。比int小的类型被转型成int。

%ld用于long类型，%lld用于long long类型。

%x标识的数会被当成int进行读取，所以long long类型的数如果超过int的范围会被截断，得不到正确的结果。而且因为它多占了4个字节，还会影响后面的其它标识符的输出。

另外%f标识的数会被当成double读取，即取出8个字节读取。

\*/

        }

        return ans;

    }

};

int main(){

    Solution\* s = new Solution();

    double x;

    int n;

    /\*

    注意scanf函数和printf函数是不同寻常的函数，因为它们都没有将函数的参数限制为固定数量。scanf函数和printf函数又可变长度的参数列表。当调用带可变长度参数列表的函数时，编译器会安排float参数自动转换成为double类型，其结果是printf函数无法区分float型和double型的参数。因此在printf函数调用中%f既可以表示float型又表示double型的参数。

    另一方面，scanf函数是通过指针指向变量的。%f告诉scanf函数在所传地址位置上存储一个float型值，而%lf告诉scanf函数在所传地址位置上存储一个double型值。这里float和double的区别是非常重用的。如果给出了错误的转换说明，那么scanf函数将可能存储错误的字节数量（没有提到的是，float型的为模式可能不同于double型的位模式）。

    \*/

    scanf("%lf %d", &x, &n);

    printf("%f", s->myPow(x, n));

    getchar();

    getchar();

    return 0;

}

[剑指 Offer 17. 打印从1到最大的n位数](https://leetcode-cn.com/problems/da-yin-cong-1dao-zui-da-de-nwei-shu-lcof/)

输入数字 n，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

**示例 1:**

**输入:** n = 1

**输出:** [1,2,3,4,5,6,7,8,9]

说明：

* 用返回一个整数列表来代替打印
* n 为正整数

 vector<int> printNumbers(int n) {

        vector<int> res;

        if (n == 0) return res;

        for (int i=1,max=pow(10,n);i<max;i++)

            res.push\_back(i);

        return res;

    }

//大数版本

class Solution2{

public:

    void printNumbers(int n, int index, string& str, vector<int>& res){

        if(index == n){

            int num = atoi(str.c\_str());

            if(num != 0)

                res.push\_back(num);

            return;

        }

        for(int i = 0; i < 10; ++i){

            //每一位 从0到9依次排列

            str[index] = i + '0';

            printNumbers(n, index + 1, str, res);

        }

    }

    vector<int> printNumbers(int n){

        vector<int> res;

        string str;

        str.resize(n);

        printNumbers(n, 0, str, res);

        return res;

    }

};

[剑指 Offer 18. 删除链表的节点](https://leetcode-cn.com/problems/shan-chu-lian-biao-de-jie-dian-lcof/)

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。

**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.

struct ListNode

{

    int val;

    ListNode \*next;

    ListNode(int x) : val(x),  next(NULL){}

};

class Solution {

public:

    ListNode\* deleteNode(ListNode\* head, int val) {

        if(head == NULL) return NULL;

        if(head->val == val) return head->next;

        ListNode \*p = head;

        while(p->next){

            if(p->next->val == val){

                p->next = p->next->next;

                break;

            }

            p = p->next;

        }

        return head;

    }

};

[剑指 Offer 19. 正则表达式匹配](https://leetcode-cn.com/problems/zheng-ze-biao-da-shi-pi-pei-lcof/)

请实现一个函数用来匹配包含'. '和'\*'的正则表达式。模式中的字符'.'表示任意一个字符，而'\*'表示它前面的字符可以出现任意次（含0次）。在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab\*ac\*a"匹配，但与"aa.a"和"ab\*a"均不匹配。

**示例 1:**

**输入:**

s = "aa"

p = "a"

**输出:** false

**解释:** "a" 无法匹配 "aa" 整个字符串。

**示例 2:**

**输入:**

s = "aa"

p = "a\*"

**输出:** true

**解释:** 因为 '\*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。

**示例 3:**

**输入:**

s = "ab"

p = ".\*"

**输出:** true

**解释:** ".\*" 表示可匹配零个或多个（'\*'）任意字符（'.'）。

**示例 4:**

**输入:**

s = "aab"

p = "c\*a\*b"

**输出:** true

**解释:** 因为 '\*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。

**示例 5:**

**输入:**

s = "mississippi"

p = "mis\*is\*p\*."

**输出:** false

* s 可能为空，且只包含从 a-z 的小写字母。
* p 可能为空，且只包含从 a-z 的小写字母以及字符 . 和 \*，无连续的 '\*'。

    bool isMatch(string s, string p) {

        int n = s.size();

        int m = p.size();

        vector<vector<bool>> f(n + 1, vector<bool>(m + 1, false));

        for(int i = 0; i <= n; ++i){

            for(int j = 0; j <= m; ++j){

                //分成空正则和非空正则两种

                if(j == 0)

                    f[i][j] = i == 0;

                else

                {

                    //非空正则分为两种情况 \* 和 非\*

                    if(p[j - 1] != '\*'){

                        if(i > 0 && (s[i - 1] == p[j - 1] || p[j - 1] == '.'))

                            f[i][j] = f[i - 1][j - 1];

                    }

                    else

                    {   //碰到 \* 了，分为看和不看两种情况

                        //不看

                        if(j >= 2)

                            f[i][j] = f[i][j - 2];//注意这里的f[j]对应的是字符串中第j个元素,从1开始,下面的p[j]中的是下标;

                        //看

                        if(i >= 1 && j >= 2 && (s[i - 1] == p[j - 2] || p[j - 2] == '.'))

                            f[i][j] = f[i - 1][j];

                    }

                }

            }

        }

        return f[n][m];

    }

解题思路

特判，同时也是递归出口，如果p是空串，返回s是否为空串。如果p不为空，保证一定存在p[1]（可能是字符串结尾\0）

假如p[1] == \* 的话，可以尝试两种情况：情况一是递归比较s和p.substr(2)；情况二是当s[0]可以匹配p[0]时, 尝试递归比较s.substr(1)和p，这里没有必要比较s.substr(1) 和 p.substr(2)，因为这种情况已经包含在递归比较s.substr(1)和p当中了

假如p[1] != \*,如果p[0]不匹配s[0]，返回false，否则递归判断s.substr(1)和p.substr(1)

class Solution {

public:

    bool isMatch(string s, string p) {

        if(p.empty()) return s.empty();

        if(p[1] == '\*'){

            return isMatch(s, p.substr(2)) || (!s.empty() && (s[0] == p[0] || p[0] == '.')) && isMatch(s.substr(1), p);

        }

        else{

            return !s.empty() && (s[0] == p[0] || p[0] == '.') && (isMatch(s.substr(1), p.substr(1)));

        }

    }

};

[剑指 Offer 20. 表示数值的字符串](https://leetcode-cn.com/problems/biao-shi-shu-zhi-de-zi-fu-chuan-lcof/)

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）。例如，字符串"+100"、"5e2"、"-123"、"3.1416"、"-1E-16"、"0123"都表示数值，但"12e"、"1a3.14"、"1.2.3"、"+-5"及"12e+5.4"都不是。

class Solution {

private:

    // 整数的格式可以用[+|-]B表示, 其中B为无符号整数

    bool scanInteger(const string s, int& index){

        if(s[index] == '+' || s[index] == '-')

            ++index;

        return scanUnsignedInteger(s, index);

    }

    bool scanUnsignedInteger(const string s, int& index){

        int befor = index;

        while(index != s.size() && s[index] >= '0' && s[index] <= '9')

            index ++;

        return index > befor;

    }

public:

    // 数字的格式可以用A[.[B]][e|EC]或者.B[e|EC]表示，

    // 其中A和C都是整数（可以有正负号，也可以没有），而B是一个无符号整数

    bool isNumber(string s) {

        if(s.size() == 0)

            return false;

        int index = 0;

        //字符串开始有空格，可以返回true

        while(s[index] == ' ')  //书中代码没有该项测试

            ++index;

        bool numeric = scanInteger(s, index);

        // 如果出现'.'，接下来是数字的小数部分

        if(s[index] == '.'){

            ++index;

            // 下面一行代码用||的原因：

            // 1. 小数可以没有整数部分，例如.123等于0.123；

            // 2. 小数点后面可以没有数字，例如233.等于233.0；

            // 3. 当然小数点前面和后面可以有数字，例如233.666

            numeric = scanUnsignedInteger(s, index) || numeric;

        }

        // 如果出现'e'或者'E'，接下来跟着的是数字的指数部分

        if(s[index] == 'e' || s[index] == 'E'){

            ++index;

            // 下面一行代码用&&的原因：

            // 1. 当e或E前面没有数字时，整个字符串不能表示数字，例如.e1、e1；

            // 2. 当e或E后面没有整数时，整个字符串不能表示数字，例如12e、12e+5.4

            numeric = numeric && scanInteger(s ,index);

        }

        //字符串结尾有空格，可以返回true

        while(s[index] == ' ')

            ++index;

        return numeric && index == s.size();

    }

};

[剑指 Offer 21. 调整数组顺序使奇数位于偶数前面](https://leetcode-cn.com/problems/diao-zheng-shu-zu-shun-xu-shi-qi-shu-wei-yu-ou-shu-qian-mian-lcof/)

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，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

**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。

**提示：**

1. 1 <= nums.length <= 50000
2. 1 <= nums[i] <= 10000

首尾双指针定义头指针 left ，尾指针 right .

left 一直往右移，直到它指向的值为偶数right 一直往左移， 直到它指向的值为奇数

交换 nums[left]和nums[right] .重复上述操作，直到 left == right.

  vector<int> exchange(vector<int>& nums) {

        int left = 0, right = nums.size() - 1;

        while (left < right) {

            if ((nums[left] & 1) != 0) {

                left ++;

                continue;

            }

            if ((nums[right] & 1) != 1) {

                right --;

                continue;

            }

            swap(nums[left++], nums[right--]);

        }

        return nums;

}

// 快慢双指针

// 定义快慢双指针 fast和 low，fast 在前， low在后 .

// fast的作用是向前搜索奇数位置，low的作用是指向下一个奇数应当存放的位置

// fast向前移动，当它搜索到奇数时，将它和 nums[low]交换，此时 low向前移动一个位置 .

// 重复上述操作，直到 fast指向数组末尾

    vector<int> exchange(vector<int>& nums) {

        int low = 0, fast = 0;

        while (fast < nums.size()) {

            if (nums[fast] & 1) {

                swap(nums[low], nums[fast]);

                low ++;

            }

            fast ++;

        }

        return nums;

}

[剑指 Offer 22. 链表中倒数第k个节点](https://leetcode-cn.com/problems/lian-biao-zhong-dao-shu-di-kge-jie-dian-lcof/)

数组与链表

数组，所有元素都连续的存储于一段内存中，且每个元素占用的内存大小相同。这使得数组具备了通过下标快速访问数据的能力。

但连续存储的缺点也很明显，增加容量，增删元素的成本很高，时间复杂度均为 O(n)。

增加数组容量需要先申请一块新的内存，然后复制原有的元素。如果需要的话，可能还要删除原先的内存。

删除元素时需要移动被删除元素之后的所有元素以保证所有元素是连续的。增加元素时需要移动指定位置及之后的所有元素，然后将新增元素插入到指定位置，如果容量不足的话还需要先进行扩容操作。

总结一下数组的优缺点：

优点：可以根据偏移实现快速的随机读写。

缺点：扩容，增删元素极慢。

链表，由若干个结点组成，每个结点包含数据域和指针域。

一般来讲，链表中只会有一个结点的指针域为空，该结点为尾结点，其他结点的指针域都会存储一个结点的内存地址。链表中也只会有一个结点的内存地址没有存储在其他结点的指针域，该结点称为头结点。

链表的存储方式使得它可以高效的在指定位置插入与删除，时间复杂度均为 O(1)。

在结点 p 之后增加一个结点 q 总共分三步：

申请一段内存用以存储 q (可以使用内存池避免频繁申请和销毁内存)。

将 p 的指针域数据复制到 q 的指针域。

更新 p 的指针域为 q 的地址。

删除结点 p 之后的结点 q 总共分两步：

将 q 的指针域复制到 p 的指针域。

释放 q 结点的内存。

面试问题总结

无法高效获取长度，无法根据偏移快速访问元素，是链表的两个劣势。然而面试的时候经常碰见诸如获取倒数第k个元素，获取中间位置的元素，判断链表是否存在环，判断环的长度等和长度与位置有关的问题。这些问题都可以通过灵活运用双指针来解决。

Tips：双指针并不是固定的公式，而是一种思维方式~

先来看"倒数第k个元素的问题"。设有两个指针 p 和 q，初始时均指向头结点。首先，先让 p 沿着 next 移动 k 次。此时，p 指向第 k+1个结点，q 指向头节点，两个指针的距离为 k 。然后，同时移动 p 和 q，直到 p 指向空，此时 p 即指向倒数第 k 个结点。

    ListNode\* getKthFromEnd(ListNode\* head, int k) {

        ListNode \*p = head, \*q = head; //初始化

        while(k--)   //将 p指针移动 k 次

            p = p->next;

        while(p != nullptr) {//同时移动，直到 p == nullptr

            p = p->next;

            q = q->next;

        }

        return q;

}

获取中间元素的问题。设有两个指针 fast 和 slow，初始时指向头节点。每

次移动时，fast向后走两次，slow向后走一次，直到 fast 无法向后走两次。

这使得在每轮移动之后。fast 和 slow 的距离就会增加一。

设链表有 n 个元素，那么最多移动 n/2 轮。当 n 为奇数时，slow 恰好指向中间结点，

当 n 为 偶数时，slow 恰好指向中间两个结点的靠后一个(可以考虑下如何使其指向前一个结点呢？)。

    ListNode\* middleNode(ListNode\* head) {

        ListNode \*p = head, \*q = head;

        while(q != nullptr && q->next != nullptr) {

            p = p->next;

            q = q->next->next;

        }

        return p;

}

是否存在环的问题。如果将尾结点的 next 指针指向其他任意一个结点，那么链表就存在了一个环。

当一个链表有环时，快慢指针都会陷入环中进行无限次移动，然后变成了追及问题。想象一下在操场跑步的场景，只要一直跑下去，快的总会追上慢的。当两个指针都进入环后，每轮移动使得慢指针到快指针的距离增加一，同时快指针到慢指针的距离也减少一，只要一直移动下去，快指针总会追上慢指针。

根据上述表述得出，如果一个链表存在环，那么快慢指针必然会相遇。实现代码如下：

    bool hasCycle(ListNode \*head) {

        ListNode \*slow = head;

        ListNode \*fast = head;

        while(fast != nullptr) {

            fast = fast->next;

            if(fast != nullptr)

                fast = fast->next;

            if(fast == slow)

                return true;

            slow = slow->next;

        }

        return nullptr;

}

最后一个问题，如果存在环，如何判断环的长度呢？方法是，快慢指针相遇后继续移动，直到第二次相遇。两次相遇间的移动次数即为环的长度。

[剑指 Offer 24. 反转链表](https://leetcode-cn.com/problems/fan-zhuan-lian-biao-lcof/)

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

双指针

class Solution {

public:

    ListNode\* reverseList(ListNode\* head) {

        ListNode\* cur = NULL, \*pre = head;

        while (pre != NULL) {

            ListNode\* t = pre->next;

            pre->next = cur;

            cur = pre;

            pre = t;

        }

        return cur;

    }

};

递归

class Solution {

public:

    ListNode\* reverseList(ListNode\* head) {

        if (head == NULL || head->next == NULL) {

            return head;

        }

        ListNode\* ret = reverseList(head->next);

        head->next->next = head;

        head->next = NULL;

        return ret;

    }

};

[剑指 Offer 25. 合并两个排序的链表](https://leetcode-cn.com/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/)

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

    ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {//迭代

        ListNode\* head = new ListNode(1);

        ListNode\* ret = head;

        while (l1 != NULL && l2 != NULL) {

            if (l1->val < l2->val) {

                head->next = l1;

                l1 = l1->next;

            } else {

                head->next = l2;

                l2 = l2->next;

            }

            head = head->next;

        }

        head->next = l1 == NULL ? l2 : l1;

        return ret->next;}

    ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {//递归

        if (l1 == NULL) {

            return l2;

        }

        if (l2 == NULL) {

            return l1;

        }

        if (l1->val <= l2->val) {

            l1->next = mergeTwoLists(l1->next, l2);

            return l1;

        }

        l2->next = mergeTwoLists(l1, l2->next);

        return l2;

}

}

[剑指 Offer 26. 树的子结构](https://leetcode-cn.com/problems/shu-de-zi-jie-gou-lcof/)

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

    bool help(TreeNode\* A, TreeNode\* B){

        if(A == NULL || B == NULL){

            return B == NULL;

        }

        if(A->val != B->val)

            return false;

        return help(A->left, B->left) && help(A->right, B->right);

    }

    bool isSubStructure(TreeNode\* A, TreeNode\* B) {

        if(A == NULL || B == NULL)

            return false;

        return help(A, B) || isSubStructure(A->left, B) || isSubStructure(A->right, B);

}

[剑指 Offer 27. 二叉树的镜像](https://leetcode-cn.com/problems/er-cha-shu-de-jing-xiang-lcof/)

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

class Solution {//递归，先序遍历

public:

    TreeNode\* mirrorTree(TreeNode\* root) {

        if (root == NULL) {

            return NULL;

        }

        swap(root->left, root->right);

        mirrorTree(root->left);

        mirrorTree(root->right);

        return root;

    }

};

栈模拟

class Solution {

public:

    TreeNode\* mirrorTree(TreeNode\* root) {

        stack<TreeNode\*> s;

        s.push(root);

        while (!s.empty()) {

            TreeNode\* node = s.top();

            s.pop();

            if (node == NULL) {

                continue;

            }

            swap(node->left, node->right);

            s.push(node->left);

            s.push(node->right);

        }

        return root;

    }

};

队列模拟

class Solution {

public:

    TreeNode\* mirrorTree(TreeNode\* root) {

        queue<TreeNode\*> q;

        q.push(root);

        while (!q.empty()) {

            TreeNode\* node = q.front();

            q.pop();

            if (node == NULL) {

                continue;

            }

            swap(node->left, node->right);

            q.push(node->left);

            q.push(node->right);

        }

        return root;

    }

};

[剑指 Offer 28. 对称的二叉树](https://leetcode-cn.com/problems/dui-cheng-de-er-cha-shu-lcof/)

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：**

0 <= 节点个数 <= 1000

方法一： 递归法

class Solution {

public:

    bool isSymmetric(TreeNode\* root) {

        bool res = true;

        if (root!=NULL)

        {

            res = helper(root->left,root->right);

        }

        return res;

    }

    bool helper(TreeNode\*A, TreeNode\* B)

    {

        // 先写递归终止条件

        if (A==NULL && B==NULL)

            return true;

        // 如果其中之一为空，也不是对称的

        if (A==NULL || B==NULL)

            return false;

        // 走到这里二者一定不为空

        if (A->val != B->val)

            return false;

        // 前序遍历

        return helper(A->left,B->right) && helper(A->right,B->left);

    }

};

时间复杂度：O(n)，其中n是树中结点的总数。

空间复杂度：在最糟糕的情况下，由栈上的递归调用造成的空间复杂度为O(n)。

方法二： 迭代法

class Solution {

public:

    bool isSymmetric(TreeNode\* root) {

        if (root ==NULL)

            return true;

        //用队列保存节点

        queue<TreeNode\*> q;

        //将根节点的左右孩子放到队列中

        q.push(root->left);

        q.push(root->right);

        while(!q.empty())

        {

            //从队列中取出两个节点，再比较这两个节点

            TreeNode\* A = q.front();

            q.pop();

            TreeNode\* B = q.front();

            q.pop();

            //如果两个节点都为空就继续循环，两者有一个为空就返回false

            if (A==NULL && B==NULL)

                continue;

            if (A==NULL || B==NULL)

                return false;

            if (A->val != B->val)

                return false;

            //将左子树的左孩子， 右子树的右孩子放入队列

            q.push(A->left);

            q.push(B->right);

            //将左子树的右孩子，右子树的左孩子放入队列

            q.push(A->right);

            q.push(B->left);

        }

        return true;

    }

};

时间复杂度：O(n)

空间复杂度：O(n)

[剑指 Offer 29. 顺时针打印矩阵](https://leetcode-cn.com/problems/shun-shi-zhen-da-yin-ju-zhen-lcof/)

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

    vector<int> spiralOrder(vector<vector<int>>& matrix) {

        if(matrix.size() == 0)

            return {};

        int l = 0, r = matrix[0].size() - 1, t = 0, b = matrix.size() - 1, x = 0;

        vector<int> res((r + 1) \* (b + 1));

        while(1){

            for(int i = l; i <= r; ++i)

                res[x++] = matrix[t][i];

            if(++t > b)

                break;

            for(int i = t; i <= b; ++i)

                res[x++] = matrix[i][r];

            if(--r < l)

                break;

            for(int i = r; i >= l; --i)

                res[x++] = matrix[b][i];

            if(--b < t)

                break;

            for(int i = b; i >= t; --i)

                res[x++] = matrix[i][l];

            if(++l > r)

                break;

        }

        return res;

    }

[剑指 Offer 30. 包含min函数的栈](https://leetcode-cn.com/problems/bao-han-minhan-shu-de-zhan-lcof/)

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.

class MinStack {

public:

    MinStack() {

    }

    stack<int>s;

    stack<int>Min;

    void push(int x) {

        s.push(x);

        if(Min.empty()||x<=Min.top())Min.push(x);

    }

    void pop() {

        if(!s.empty()){

            if(s.top()==Min.top())Min.pop();

            s.pop();

        }

    }

    int top() {

        return s.top();

    }

    int getMin() {

        return Min.top();

    }

};

[剑指 Offer 31. 栈的压入、弹出序列](https://leetcode-cn.com/problems/zhan-de-ya-ru-dan-chu-xu-lie-lcof/)

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

**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。

    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {

        stack<int> s;

        int n = pushed.size();

        int j = 0;

        for(int i = 0; i < n; ++i){

            s.push(pushed[i]);

            while(!s.empty() && j < n && s.top() == popped[j]){

                s.pop();

                j++;

            }

        }

        return s.empty();

}

[剑指 Offer 32 - I. 从上到下打印二叉树](https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/)

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如:  
给定二叉树: [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回：

[3,9,20,15,7]

    vector<int> levelOrder(TreeNode\* root) {

        vector<int>ret;

        queue<TreeNode\*> que;

        que.push(root);

        if(!root)return ret;

        TreeNode\* temp;

        while(!que.empty()){

            temp=que.front();

            que.pop();

            ret.push\_back(temp->val);

            if(temp->left)que.push(temp->left);

            if(temp->right)que.push(temp->right);

        }

        return ret;

}

[剑指 Offer 32 - II. 从上到下打印二叉树 II](https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-ii-lcof/)

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

例如:  
给定二叉树: [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回其层次遍历结果：

[

[3],

[9,20],

[15,7]

]

DFS：

class Solution {

public:

    vector<vector<int>> levelOrder(TreeNode\* root) {

        vector<vector<int>> res;

        dfs(root,res,0);

        return res;

    }

    void dfs(TreeNode\* root,vector<vector<int>>& res,int level)

    {

        if(!root) return;

        if(level>=res.size()) res.emplace\_back(vector<int>());

        res[level].emplace\_back(root->val);

        dfs(root->left,res,level+1);

        dfs(root->right,res,level+1);

    }

};

BFS:

class Solution {

public:

    vector<vector<int>> levelOrder(TreeNode\* root) {

        queue<TreeNode\*> que;

        vector<vector<int>> res;

        if(root == NULL)

            return res;

        vector<int> tmp;

        que.push(root);

        while(!que.empty()){

            int n = que.size();

            tmp.clear();

            for(int i = 0; i < n; ++i){

                TreeNode\* a = que.front();

                que.pop();

                tmp.push\_back(a->val);

                if(a->left) que.push(a->left);

                if(a->right) que.push(a->right);

            }

            if(!tmp.empty())

                res.push\_back(tmp);

        }

        return res;

    }

};

[剑指 Offer 32 - III. 从上到下打印二叉树 III](https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/)

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

例如:  
给定二叉树: [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回其层次遍历结果：

[

[3],

[20,9],

[15,7]

]

 vector<vector<int>> levelOrder(TreeNode\* root) {

        deque<TreeNode\*> q; vector<vector<int>> res;

        q.push\_back(root);

        bool flag = true; //从左向右打印为true，从右向左打印为false

        while(!q.empty()){

            int size = q.size();

            vector<int> level;

            for(int i = 0; i < size; ++i){

                if(flag){   // 前取后放：从左向右打印，所以从前边取，后边放入

                    TreeNode\* rt = q.front();

                    q.pop\_front();

                    if(!rt)  continue;

                  level.push\_back(rt->val);

                    if(rt->left) q.push\_back(rt->left); // 下一层顺序存放至尾

                    if(rt->right)  q.push\_back(rt->right);

                }else{  //后取前放： 从右向左，从后边取，前边放入

                    TreeNode\* rt = q.back();

                    q.pop\_back();

                    if(!rt) continue;

                    level.push\_back(rt->val);

                    if(rt->right) q.push\_front(rt->right);// 下一层逆序存放至首

                    if(rt->left)  q.push\_front(rt->left);

                }

            }

            flag = !flag;

            if(!level.empty())

                res.push\_back(level);

        }

        return res;

    }

[剑指 Offer 33. 二叉搜索树的后序遍历序列](https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-hou-xu-bian-li-xu-lie-lcof/)

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 true，否则返回 false。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

5

/ \

2 6

/ \

1 3

**示例 1：**

**输入:** [1,6,3,2,5]

**输出:** false

**示例 2：**

**输入:** [1,3,2,6,5]

**输出:** true

bool verifyPostorder(vector<int>& postorder) {

       return recu(postorder, 0, postorder.size() - 1);

}

 bool recu(vector<int>& postorder, int left, int right){

        bool ans = true;

        if(left >= right)

            return ans;

        int i = left;

        for(;i < right; ++i){

            if(postorder[i] > postorder[right])

                break;

        }

        for(int j = i; j < right; ++j){

            if(postorder[j] < postorder[right])

                return false;

        }

        ans = ans && recu(postorder, left, i - 1);

        ans = ans && recu(postorder, i, right - 1);

        return ans;

}

时间复杂度：O(nlog(n)),最差O(n^2)

空间复杂度：O(log(n)),最差O(n)

[剑指 Offer 34. 二叉树中和为某一值的路径](https://leetcode-cn.com/problems/er-cha-shu-zhong-he-wei-mou-yi-zhi-de-lu-jing-lcof/)

输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

**示例:**  
给定如下二叉树，以及目标和 sum = 22，

**5**

/ \

**4** **8**

/ / \

**11** 13 **4**

/ \ / \

7 **2** **5** 1

返回:

[

[5,4,11,2],

[5,8,4,5]

]

class Solution {

public:

    vector<vector<int>> res;

    vector<int> nums;

    vector<vector<int>> pathSum(TreeNode\* root, int sum) {

        dfs(root, sum, nums);

        return res;

    }

    void dfs(TreeNode\* root, int sum, vector<int>& nums){

        if(root == NULL)

            return;

        nums.push\_back(root->val);

        if(root->val == sum && root->left == NULL && root->right == NULL){

            res.push\_back(nums);

        }

        dfs(root->left, sum - root->val, nums);

        dfs(root->right, sum - root->val, nums);

        nums.pop\_back();

        return ;

    }

};

[剑指 Offer 35. 复杂链表的复制](https://leetcode-cn.com/problems/fu-za-lian-biao-de-fu-zhi-lcof/)

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。

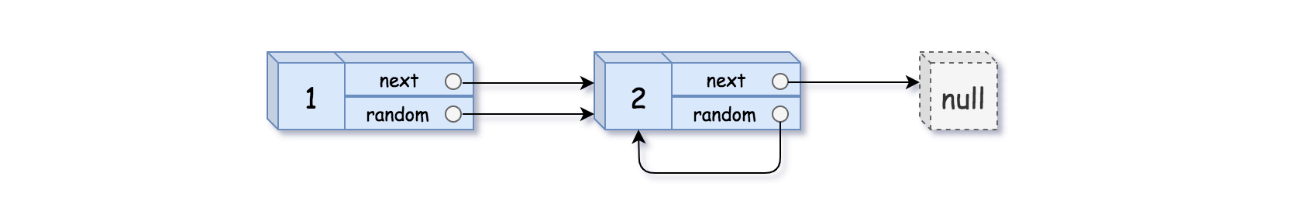
**示例 1：**



**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。

class Node {

public:

    int val;

    Node\* next;

    Node\* random;

    Node(int \_val) {

        val = \_val;

        next = NULL;

        random = NULL;

    }

};

算法(迭代)

    Node\* copyRandomList(Node\* head) {

        if (!head) return nullptr;

        unordered\_map<Node\*, Node\*> mp;

        for (Node \*it = head; it; it = it->next) {

            mp[it] = new Node(it->val);

        }

        for (Node \*it = head; it; it = it->next) {

            if (it->next) mp[it]->next = mp[it->next];

            if (it->random) mp[it]->random = mp[it->random];

        }

        return mp[head];

}

算法(dfs)

    Node\* copyRandomList(Node\* head) {

        unordered\_map<Node\*, Node\*> used;

        return dfs(head, used);

    }

    Node\* dfs(Node\* head, unordered\_map<Node\*, Node\*>& used) {

        if (!head) return nullptr;

        if (used.count(head)) return used[head];

        Node \*res = new Node(head->val);

        used[head] = res;

        res->next = dfs(head->next, used);

        res->random = dfs(head->random, used);

        return res;

}

算法(bfs)

    Node\* copyRandomList(Node\* head) {

        return bfs(head);

    }

    Node\* bfs(Node\* head) {

        if (!head) return nullptr;

        unordered\_map<Node\*, Node\*> used;

        used[head] = new Node(head->val);

        queue<Node\*> q;

        q.push(head);

        while (!q.empty()) {

            int siz = q.size();

            for (int i = 0; i < siz; i++) {

                Node \*t = q.front(); q.pop();

                if (t->next) {

                    if (!used[t->next]) {

                        used[t->next] = new Node(t->next->val);

                        q.push(t->next);

                    }

                    used[t]->next = used[t->next];

                }

                if (t->random) {

                    if (!used[t->random]) {

                        used[t->random] = new Node(t->random->val);

                        q.push(t->random);

                    }

                    used[t]->random = used[t->random];

                }

            }

        }

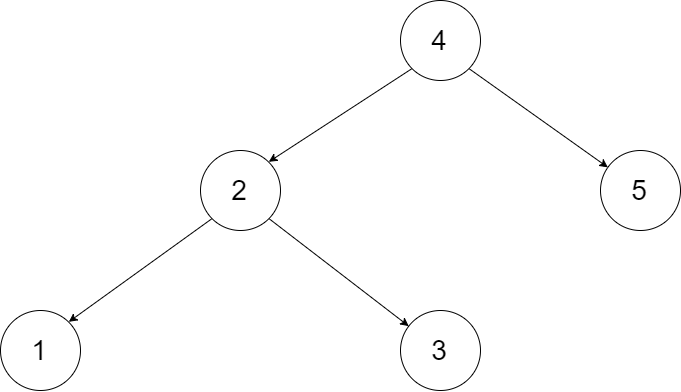
        return used[head];

    }

[剑指 Offer 36. 二叉搜索树与双向链表](https://leetcode-cn.com/problems/er-cha-sou-suo-shu-yu-shuang-xiang-lian-biao-lcof/)

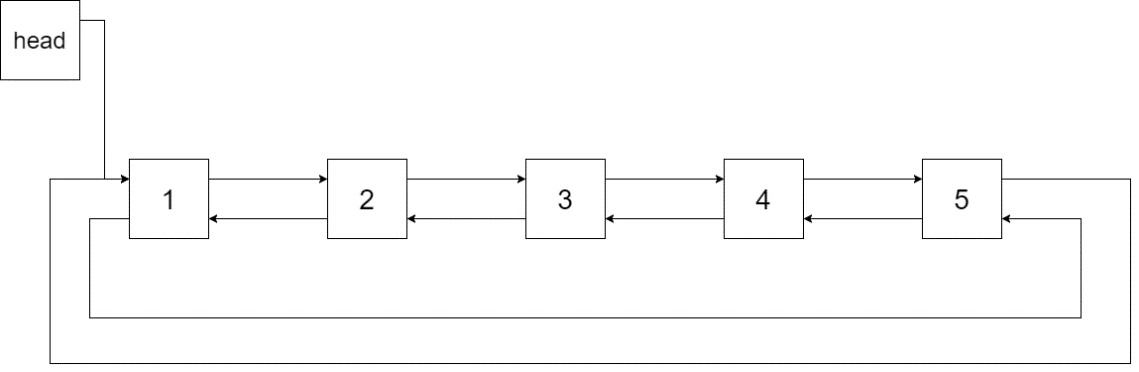
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

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。

class Node{

public:

    int val;

    Node\* left;

    Node\* right;

    Node(){}

    Node(int x){

        val = x;

        left = NULL;

        right = NULL;

    }

    Node(int x, Node\* l, Node\* r){

        val = x;

        left = l;

        right = r;

    }

};

    Node\* treeToDoublyList(Node\* root) {

        if(!root) return nullptr;

        Node\* head = nullptr, \*pre = nullptr;

        helper(root, head, pre);

        head->left = pre;

        pre->right = head;

        return head;

    }

    void helper(Node\* root, Node\*& head, Node\*& pre) {

        if(!root)  return;

        helper(root->left, head, pre);

        if(!head) {

            head = root;   // 找到head

            pre = root;    // 对pre进行初始化

        } else {

            pre->right = root;

            root->left = pre;

            pre = root;

        }

        helper(root->right, head, pre);

    }

[剑指 Offer 37. 序列化二叉树](https://leetcode-cn.com/problems/xu-lie-hua-er-cha-shu-lcof/)

请实现两个函数，分别用来序列化和反序列化二叉树。

**示例:**

你可以将以下二叉树：

1

/ \

2 3

/ \

4 5

序列化为 "[1,2,3,null,null,4,5]"

    string serialize(TreeNode\* root) {

        ostringstream out;

        queue<TreeNode\*> q;

        q.push(root);

        while (!q.empty()) {

            TreeNode\* tmp = q.front();

            q.pop();

            if (tmp) {

                out<<tmp->val<<" ";

                q.push(tmp->left);

                q.push(tmp->right);

            } else {

                out<<"null ";

            }

        }

        return out.str();

    }

    TreeNode\* deserialize(string data) {

        istringstream input(data);

        string val;

        vector<TreeNode\*> vec;

        while (input >> val) {

            if (val == "null") {

                vec.push\_back(NULL);

            } else {

                vec.push\_back(new TreeNode(stoi(val)));

            }

        }

        int j = 1;                          // i每往后移动一位，j移动两位，j始终是当前i的左子下标

        for (int i = 0; j < vec.size(); ++i) {  // 肯定是j先到达边界，所以这里判断j < vec.size()

            if (vec[i] == NULL) continue;                   // vec[i]为null时跳过。

            if (j < vec.size()) vec[i]->left = vec[j++];    // 当前j位置为i的左子树

            if (j < vec.size()) vec[i]->right = vec[j++];   // 当前j位置为i的右子树

        }

        return vec[0];

    }

[**剑指 Offer 38. 字符串的排列**](https://leetcode-cn.com/problems/zi-fu-chuan-de-pai-lie-lcof/)

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"]**

class Solution {

public:

vector<string> permutation(string s) {

vector<string> res;

vector<bool> vis(s.size(), false);

string temp;

sort(s.begin(), s.end());

backtrack(temp, s, vis, res);

return res;

}

void backtrack(string& temp, string s, vector<bool>& vis, vector<string>& res){

if(temp.size() == s.size()){

res.push\_back(temp);

return;

}

for(int i = 0; i < s.size(); ++i){

if(vis[i]) continue;

if(i > 0 && s[i - 1] == s[i] && !vis[i - 1]) continue;

vis[i] = true;

temp.push\_back(s[i]);

backtrack(temp, s, vis, res);

temp.pop\_back();

vis[i] = false;

}

}

};

class Solution {

public:

    vector<string> permutation(string s) {

        set<string> s1;

        backtrack(s, s1, 0);

        return vector<string>(s1.begin(), s1.end());

    }

void backtrack(string& s, set<string>& s1, int start){

        if(start == s.size()){

            s1.insert(s);

            return;

        }

        for(int i = start; i < s.size(); ++i){

            swap(s[i], s[start]);

            backtrack(s, s1, start + 1);

            swap(s[i], s[start]);

        }

    }

};

[剑指 Offer 39. 数组中出现次数超过一半的数字](https://leetcode-cn.com/problems/shu-zu-zhong-chu-xian-ci-shu-chao-guo-yi-ban-de-shu-zi-lcof/)

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

int majorityElement (std::vector<int> &nums) {

    assert(!nums.empty());

    // count计入当前队伍剩余人数

    // majority计入当前队伍人数

    int count = 0, majority = 0;

    for(auto num : nums){

        // 当队伍人数为0时，记录下次遇到的队伍号

        if(!count){

            majority = num;

        }

        // 判断是同队伍还是不同队伍

        if(num == majority){

            // 同队伍，队员增加

            count += 1;

        } else{

            // 不同队伍，互相抵消

            count -= 1;

        }

    }

    return majority;

}

[剑指 Offer 40. 最小的k个数](https://leetcode-cn.com/problems/zui-xiao-de-kge-shu-lcof/)

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]

堆

class Solution {

public:

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

        int endIndex = arr.size() - 1;

        buildHeap(arr, endIndex);

        while(endIndex){

            swap(arr[0], arr[endIndex]);

            endIndex--;

            heap(arr, 0, endIndex);

        }

        vector<int> res(k, 0);

        for(int i = 0; i < k; i++){

            res[i] = arr[i];

        }

        return  res;

    }

    //初始化建堆的时间复杂度为O(n)

    void buildHeap(vector<int>& arr, int endIndex){

        for(int i = (endIndex - 1)/2; i >= 0; i--){

            heap(arr, i, endIndex);

        }

    }

    void heap(vector<int>& arr, int startIndex, int endIndex){

        while(true){

            int finalIndex = startIndex;

            if((startIndex \* 2 + 1) <= endIndex && arr[finalIndex] < arr[startIndex \* 2 + 1])

                finalIndex = startIndex \* 2 + 1;

            if((startIndex \* 2 + 2) <= endIndex && arr[finalIndex] < arr[startIndex \* 2 + 2])

                finalIndex = startIndex \* 2 + 2;

            if(finalIndex == startIndex)

                break;

            swap(arr[startIndex], arr[finalIndex]);

            startIndex = finalIndex;

        }

    }

};

重建堆一共需要n-1次循环，每次循环的比较次数为log(i)，则相加为：log2+log3+…+log(n-1)+log(n)≈log(n!)。可以证明log(n!)和nlog(n)是同阶函数：

初始化建堆的时间复杂度为O(n)，排序重建堆的时间复杂度为nlog(n)，所以总的时间复杂度为O(n+nlogn)=O(nlogn)。

//优先级队列

class Solution {

public:

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

        vector<int>vec(k, 0);

        if (k == 0) return vec; // 排除 0 的情况

        priority\_queue<int>Q;

        for (int i = 0; i < k; ++i) Q.push( arr[i]);

        for (int i = k; i < (int)arr.size(); ++i) {

            if (Q.top() > arr[i]) {

                Q.pop();

                Q.push(arr[i]);

            }

        }

        for (int i = 0; i < k; ++i) {

            vec[i] = Q.top();

            Q.pop();

        }

        return vec;

    }

};

时间复杂度：O(nlogk)，其中 n是数组 arr 的长度。由于大根堆实时维护前 k小值，所以插入删除都是 O(\log k)O(logk) 的时间复杂度，最坏情况下数组里 nn 个数都会插入，所以一共需要O(nlogk) 的时间复杂度。

空间复杂度：O(k)，因为大根堆里最多 k个数。

快排

class Solution {

public:

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

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

        vector<int> res(k, 0);

        for(int i = 0; i < k; i++){

            res[i] = arr[i];

        }

        return res;

    }

    int getStandard(vector<int>& arr, int i, int j){

        int key = arr[i];

        while(i < j){

            while(i < j && arr[j] >= key)

                j--;

            if(i < j)

                arr[i] = arr [j];

            while(i < j && arr[i] <= key)

                i++;

            if(i < j)

                arr[j] = arr[i];

        }

        arr[i] = key;

        return i;

    }

    void quickSort(vector<int>& arr, int low, int high){

        if(low < high){

            int standard = getStandard(arr, low, high);

            quickSort(arr, low, standard - 1);

            quickSort(arr, standard + 1, high);

        }

    }

};

[剑指 Offer 41. 数据流中的中位数](https://leetcode-cn.com/problems/shu-ju-liu-zhong-de-zhong-wei-shu-lcof/)

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]

方法：二分查找插入

方法一的缺点在于对数组进行了排序操作，导致时间复杂度较高，假如每次插入一个值前数组已经排好序了呢？这样我们只需考虑每次将值插在合适的位置即可，所以使用二分查找来找到这个合适的位置，会将时间复杂度降低到O(n)（查找:O(logn)，插入: O(n)）。

class MedianFinder {

    vector<int> store; // resize-able container

public:

    // Adds a number into the data structure.

    void addNum(int num)

    {

        if (store.empty())

            store.push\_back(num);

        else

            store.insert(lower\_bound(store.begin(), store.end(), num), num);     // binary search and insertion combined

    }

    // Returns the median of current data stream

    double findMedian()

    {

        int n = store.size();

        return n & 1 ? store[n / 2] : (store[n / 2 - 1] + store[n / 2]) \* 0.5;

    }

};

复杂度分析

时间复杂度：O(n)。O(logn)+O(n)≈O(n)。

空间复杂度：O(n)。使用了数组保存输入。

方法三：优先队列（堆）

我们将中位数左边的数保存在大顶堆中，右边的数保存在小顶堆中。这样我们可以在O(1) 时间内得到中位数。

class MedianFinder {

    priority\_queue<int> lo;                              // 大顶堆

    priority\_queue<int, vector<int>, greater<int>> hi;   // 小顶堆

public:

    // Adds a number into the data structure.

    void addNum(int num)

    {

        lo.push(num);                                    // 加到大顶堆

        hi.push(lo.top());                               // 平衡

        lo.pop();

        if (lo.size() < hi.size()) {                     // 维护两个堆元素个数

            lo.push(hi.top());

            hi.pop();

        }

    }

    // Returns the median of current data stream

    double findMedian()

    {

        return lo.size() > hi.size() ? (double) lo.top() : (lo.top() + hi.top()) \* 0.5;

    }

};

时间复杂度：O(logn)。堆插入和删除需要 O(logn)，查找中位数需要O(1)。

空间复杂度：O(n)。

[剑指 Offer 42. 连续子数组的最大和](https://leetcode-cn.com/problems/lian-xu-zi-shu-zu-de-zui-da-he-lcof/)

输入一个整型数组，数组中的一个或连续多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。

**示例1:**

**输入:** nums = [-2,1,-3,4,-1,2,1,-5,4]

**输出:** 6

**解释:** 连续子数组 [4,-1,2,1] 的和最大，为 6。

  int maxSubArray(vector<int>& nums) {

        int res = nums[0];

        for(int i = 1; i < nums.size(); i++){

            nums[i] = max(nums[i - 1] + nums[i], nums[i]);

            res = max(nums[i], res);

        }

        return res;

}

[剑指 Offer 43. 1～n整数中1出现的次数](https://leetcode-cn.com/problems/1nzheng-shu-zhong-1chu-xian-de-ci-shu-lcof/)

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

解析

需要通过找规律来分析。假设我们对5014这个数字求解。

（1）个位上1出现的个数：记高位为high=501，当前位为cur=4。

那么高位从0~500变化的过程中，每一个变化中1只出现1次，即（高位1）这样的数字；

高位是501时，因为当前位是4，所以1只能出现一次，即5011。

所以总共出现的次数为high\*1+1=502。

（2）十位1出现的个数：记高位high=50，当前位为cur=1，低位为low=4。

那么高位从0~ 49变化的过程中，每一个变化中1出现10次，即（高位10）~（高位19）这样的数字；

高位为50的时候，因为当前位是1，所以我们要看低位来决定出现的次数，因为低位为4，所以此时出现5次，即5010~5014这样的数字。

所以总共出现的次数为high\*10+4=505。

（3）百位1出现的个数：记高位high=5，当前位cur=0，低位为low=14。

那么高位从0~ 4的过程中，每一个变化1出现100次，即（高位100）~（高位199）这样的数字；

高位为5的时候，因为当前位为0，所以不存在出现1的可能性。

所以总共出现的次数为high\*100+0=500。

（4）千位1出现的次数：记高位high=0，当前位cur=5，低位low=014。

那么因为没有高位所以直接看当前位，因为当前位为5，所以1出现的次数为1000，即1000~1999这样的数字。

所以总共出现的次数为high\*1000+1000=1000。

综上，最终的结果将每个位置出现1的次数累加即可。

结论

我们假设高位为high，当前位为cur，低位为low，i代表着需要统计的位置数（1对应个位，10对应十位，100对应百位），则对每一位的个数count有：

cur=0,count = high\*i;

cur=1,count=high\*i+low+1;

cur>1,count=high\*i+i

最终累加所有位置上的个数即最终答案。

class Solution {

public:

    int countDigitOne(int n) {

       int count = 0;

       long i = 1;//指向遍历的位数，如i=1即个位，i=10即十位，...，因为n可以有31位，所以10^31用long存储

       while(n/i!=0){

            //n/i控制遍历的次数，将所有的位数都遍历完毕

            long high = n/(10\*i);//将当前位之前的所有高位都存在high中

            long cur = (n/i)%10;//将当前位记录在cur中，即我们每次都需要统计当前位上1出现的次数

            long low = n-(n/i)\*i;

            if(cur == 0){

                count += high \* i;

            } else if(cur == 1){

                count += high \* i + low + 1;

            } else {

                count += high \* i + i;

            }

            i = i \* 10;//准备遍历下一位

       }

       return count;

    }

};

[剑指 Offer 44. 数字序列中某一位的数字](https://leetcode-cn.com/problems/shu-zi-xu-lie-zhong-mou-yi-wei-de-shu-zi-lcof/)

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

class Solution {

public:

    int findNthDigit(int n) {

        // 计算该数字由几位数字组成，由1位：digits = 1；2位：digits = 2...

        long base = 9,digits = 1;

        while (n - base \* digits > 0){

            n -= base \* digits;

            base \*= 10;

            digits ++;

        }

        // 计算真实代表的数字是多少

        int idx = n % digits;  // 注意由于上面的计算，n现在表示digits位数的第n个数字

        if (idx == 0)idx = digits;

        long number = 1;

        for (int i = 1;i < digits;i++)

            number \*= 10;

        number += (idx == digits)? n/digits - 1:n/digits;

        // 从真实的数字中找到我们想要的那个数字

        for (int i=idx;i<digits;i++) number /= 10;

        return number % 10;

    }

}

[剑指 Offer 45. 把数组排成最小的数](https://leetcode-cn.com/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/)

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"

class Solution {

public:

    string minNumber(vector<int>& nums) {

        vector<string>strs;

        string ans;

        for(int i = 0; i < nums.size(); i ++){

            strs.push\_back(to\_string(nums[i]));

        }

        sort(strs.begin(), strs.end(), [](string& s1, string& s2){return s1 + s2 < s2 + s1;});

        for(int i = 0; i < strs.size(); i ++)

            ans += strs[i];

        return ans;

    }

};

[剑指 Offer 46. 把数字翻译成字符串](https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/)

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"

滚动数组法

class Solution {

public:

    int translateNum(int num) {

        string src = to\_string(num);

        int p = 0, q = 0, r = 1;

        for (int i = 0; i < src.size(); ++i) {

            p = q;

            q = r;

            r = 0;

            r += q;

            if (i == 0) {

                continue;

            }

            auto pre = src.substr(i - 1, 2);

            if (pre <= "25" && pre >= "10") {

                r += p;

            }

        }

        return r;

    }

};

我的理解: 有点类似于青蛙跳阶梯，运用了滚动数组，比如 1 1 2 3 5；到了3的时候单独圈出3，那 1 1 2 为一种， 看 2  3 那么 1 1 为一种 所以f(i - 1) + f(i - 2);

复杂度分析，记num=n。

时间复杂度：循环的次数是n的位数，故渐进时间复杂度为O(logn)。

空间复杂度：虽然这里用了滚动数组，动态规划部分的空间代价是O(1) 的，但是这里用了一个临时变量把数字转化成了字符串，故渐进空间复杂度也是O(logn)。

递归

class Solution {

public:

    int translateNum(int num) {

        if (num < 10) return 1;

        return (num%100 < 10 || num%100 > 25) ? translateNum(num/10) : translateNum(num/10) + translateNum(num/100);

    }

};

[剑指 Offer 47. 礼物的最大价值](https://leetcode-cn.com/problems/li-wu-de-zui-da-jie-zhi-lcof/)

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

状态表示：dp[i][j]表示到达i,j位置处的最大价值

转移方程：dp[i][j] = max{dp[i-1][j],dp[i][j-1]} + grid[i][j];

dp数组初始状态：dp[0][j] = {0}, dp[i][0] = {0} （就是将第一行和第一列全部设为0）

其实这里的dp数组我们可以将它优化成一维的，优化之后的一维dp保存的是当前上一行的最大价值，然后我们从做到有去更新这个数组即可

class Solution {

public:

    int maxValue(vector<vector<int>>& grid) {

        int n = grid.size(), m = grid[0].size();

        vector<int> res(m+1, 0);

        for(int i = 1; i <= n; ++i){

            for(int j = 1; j <= m; ++j){

                res[j] = max(res[j - 1], res[j]) + grid[i - 1][j - 1];

            }

        }

        return res[m];

    }

};

时间复杂度：O(MN)

空间复杂度：O(N)

[剑指 Offer 48. 最长不含重复字符的子字符串](https://leetcode-cn.com/problems/zui-chang-bu-han-zhong-fu-zi-fu-de-zi-zi-fu-chuan-lcof/)

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。

p w w k e w

刚开始左右指针都指向 p, 右指针最远移到 w，此时长度为 2

然后左指针右移到 w，右指针又可以移动到 e, 此时长度位 3

然后左指针右移到 k，右指针移动到 w结束。

在这个过程中，左右指针都最多遍历整个字符串 1 遍，所以时间复杂度为 O(n)

class Solution {

public:

    int lengthOfLongestSubstring(string s) {

        int maxLen = 0;

        if(s.size() == 0) return 0;

        unordered\_set<char> chars;

        int right = 0;

        for(int left = 0; left < s.size(); left ++){

            while(right < s.size() && !chars.count(s[right])){

                chars.insert(s[right]);

                right ++;

            }

            maxLen = max(maxLen, right - left);

            if(right == s.size()) break;

            chars.erase(s[left]);

        }

        return maxLen;

    }

};

而下面这个解法是进一步优化第二种解法

以 i 结尾的不重复子串，其实就是前面不重复字符的最大下标和 i 之间的子串

第二种解法是移动左右指针，我们其实可以更大步的移动左指针，一旦检测到重复字符，左指针可以直接跳到重复部分，这是很自然的，不重复子串肯定不会把重复部分包含进去。

而检测是否重复其实不必要使用set了，因为一共有127个字符，所以直接开辟128长度的数组即可。首先把每个字符的对应值设为 -1

每次把 m[s[i]] 设置为 i，是为了能够让左指针及时跳到重复部分。

举个例子：p w w k e w

在 w 处检测到重复，所以left跳到第一个 w 处，然后中最后一个 w 处又检测到重复，所以 left 跳到第二个 w 处。之所以跳到前一个重复的位置，是因为 [i,j] 之间的长度为 j - i + 1，所以就没跳到重复位置的右边，当然都是可以的。

class Solution {

public:

    int lengthOfLongestSubstring(string s) {

        vector<int> m(128, -1);

        int res = 0, left = -1;

        for(int i = 0; i < s.size(); ++i) {

            left = max(left, m[s[i]]);

            m[s[i]] = i;

            res = max(res, i - left);

        }

        return res;

        }

};

[剑指 Offer 49. 丑数](https://leetcode-cn.com/problems/chou-shu-lcof/)

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。

class Solution {

public:

    int nthUglyNumber(int n) {

        vector<int> dp(n, 0);

        int p2 = 0, p3 = 0, p5 = 0;

        dp[0] = 1;

        for(int i = 1; i < n; ++i){

            dp[i] = min(min(dp[p2] \* 2, dp[p3] \* 3), dp[p5] \* 5);

            if(dp[i] == dp[p2] \* 2)

                p2++;

            if(dp[i] == dp[p3] \* 3)

                p3++;

            if(dp[i] == dp[p5] \* 5)

                p5++;

        }

        return dp[n - 1];

    }

};

[剑指 Offer 50. 第一个只出现一次的字符](https://leetcode-cn.com/problems/di-yi-ge-zhi-chu-xian-yi-ci-de-zi-fu-lcof/)

在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

**示例:**

s = "abaccdeff"

返回 "b"

s = ""

返回 " "

    char firstUniqChar(string s) {

        if(s == " ")  return ' ';

        unordered\_map<char, int> mp;

        for(int i = 0; i < s.size(); ++i)

            mp[s[i]]++;

        for(char& c : s)

            if(mp[c] == 1)

                return c;

        return ' ';

    }

[剑指 Offer 51. 数组中的逆序对](https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/)

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

class Solution {

public:

    int mergeSort(vector<int>& nums, vector<int>& temp, int l, int r){

        if(l >= r)

            return 0;

        int mid = (l + r) >> 1;

        int count = mergeSort(nums, temp, l, mid) + mergeSort(nums, temp, mid + 1, r);

        int pos = l, i = l, j = mid + 1;

        while(i <= mid && j <= r){

            if(nums[i] <= nums[j]){

                temp[pos++] = nums[i++];

            }else{

                temp[pos++] = nums[j++];

                count += mid - i + 1;

            }

        }

        for(int k = i; k <= mid; k++){

            temp[pos++] = nums[k];

        }

        for(int k = j; k <= r; k++){

            temp[pos++] = nums[k];

        }

        copy(temp.begin() + l, temp.begin() + r + 1, nums.begin() + l);

        return count;

    }

    int reversePairs(vector<int>& nums) {

        int n = nums.size();

        vector<int> temp(n);

        return mergeSort(nums, temp, 0, n - 1);

    }

};

[剑指 Offer 52. 两个链表的第一个公共节点](https://leetcode-cn.com/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/)

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**：**

[](https://assets.leetcode-cn.com/aliyun-lc-upload/uploads/2018/12/14/160_statement.png)

在节点 c1 开始相交。

**示例 1：**

[](https://assets.leetcode.com/uploads/2018/12/13/160_example_1.png)

**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。

 Definition for singly-linked list.

  struct ListNode {

      int val;

      ListNode \*next;

      ListNode(int x) : val(x), next(NULL) {}

 };

class Solution {

public:

    ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

        ListNode\* node1 = headA;

        ListNode\* node2 = headB;

        while(node1 != node2){

            node1 = node1 != NULL ? node1->next : headB;

            node2 = node2 != NULL ? node2->next : headA;

        }

        return node1;

    }

};

[剑指 Offer 53 - I. 在排序数组中查找数字 I](https://leetcode-cn.com/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof/)

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

class Solution {

public:

    int search(vector<int>& nums, int target) {

        //搜索有边界 right

        int i = 0, j = nums.size() - 1;

        while(i <= j){

            int m = (i + j) >> 1;

            if(nums[m] <= target)

                i = m + 1;

            else

                j = m - 1;

        }

        int right = i;

        //若数组中无target 则提前返回

        if(j >= 0 && nums[j] != target)

            return 0;

        //搜索左边界 left

        i = 0, j = nums.size() - 1;

        while(i <= j){

            int m = (i + j) >> 1;

            if(nums[m] < target)

                i = m + 1;

            else

                j = m - 1;

        }

        int left = j;

        return right - left - 1;

    }

};

[剑指 Offer 53 - II. 0～n-1中缺失的数字](https://leetcode-cn.com/problems/que-shi-de-shu-zi-lcof/)

难度简单62收藏分享切换为英文关注反馈

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

int missingNumber(vector<int>& nums) {

        int i = 0, j = nums.size() - 1;

        while(i <= j){

            int m = (i + j) >> 1;

            if(nums[m] == m)

                i = m + 1;

            else

                j = m - 1;

        }

        return i;

}

[剑指 Offer 54. 二叉搜索树的第k大节点](https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-di-kda-jie-dian-lcof/)

给定一棵二叉搜索树，请找出其中第k大的节点。

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

class Solution {

public:

    int res,k;

    int kthLargest(TreeNode\* root, int k) {

        this->k = k;

        dfs(root);

        return res;

    }

    void dfs(TreeNode\* root){

        if(root == NULL)

            return;

        dfs(root->right);

        if(k == 0)

            return;

        if(--k == 0)

            res = root->val;

        dfs(root->left);

    }

};

[剑指 Offer 55 - I. 二叉树的深度](https://leetcode-cn.com/problems/er-cha-shu-de-shen-du-lcof/)

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 。

class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if(root == NULL) return 0;

        queue<TreeNode\*> q;

        int ans = 0;

        q.push(root);

        while(!q.empty()){

            int size = q.size();

            for (int i = 0; i < size; i ++) {

                TreeNode\* node = q.front();

                q.pop();

                if (node->left) q.push(node->left);

                if (node->right) q.push(node->right);

            }

            ans ++;

        }

        return ans;

    }

};

[剑指 Offer 55 - II. 平衡二叉树](https://leetcode-cn.com/problems/ping-heng-er-cha-shu-lcof/)

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。

**示例 1:**

给定二叉树 [3,9,20,null,null,15,7]

3

/ \

9 20

/ \

15 7

返回 true 。

**示例 2:**

给定二叉树 [1,2,2,3,3,null,null,4,4]

1

/ \

2 2

/ \

3 3

/ \

4 4

返回 false 。

方法一：后序遍历 + 剪枝 （从底至顶）

class Solution {

public:

    bool isBalanced(TreeNode\* root) {

        return recur(root) != -1;

    }

    int recur(TreeNode\* root){

        if(root == NULL)

            return 0;

        int left = recur(root->left);

        if(left == -1)

            return -1;

        int right = recur(root->right);

        if(right == -1)

            return -1;

        return abs(left - right) < 2 ? max(left, right) + 1 : -1;

    }

};

时间复杂度O(N)：N 为树的节点数；最差情况下，需要递归遍历树的所有节点。

空间复杂度O(N)：最差情况下（树退化为链表时），系统递归需要使用O(N) 的栈空间。

方法二：先序遍历 + 判断深度 （从顶至底）

class Solution {

public:

    bool isBalanced(TreeNode\* root) {

        if(root == NULL)

            return true;

        return abs(depth(root->left) - depth(root->right)) <= 1 && isBalanced(root->left) && isBalanced(root->right);

    }

    int depth(TreeNode\* root){

        if(root == NULL)

            return 0;

        return max(depth(root->left), depth(root->right)) + 1;

    }

};

时间复杂度 O(N log N)

空间复杂度 O(N)： 最差情况下（树退化为链表时），系统递归需要使用 O(N) 的栈空间。

[剑指 Offer 56 - I. 数组中数字出现的次数](https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-lcof/)

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]

class Solution {

public:

    vector<int> singleNumbers(vector<int>& nums) {

        int res = 0;

        for(int num : nums){

            res ^= num;

        }

        int div = 1;

        while((res & div) == 0)

            div <<= 1;

        int a = 0, b = 0;

        for(int num : nums){

            if(num & div)

                a ^= num;

            else

                b ^= num;

        }

        return vector<int> {a, b};

    }

};

[剑指 Offer 56 - II. 数组中数字出现的次数 II](https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-ii-lcof/)

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

class Solution {

public:

    int singleNumber(vector<int>& nums) {

        int res = 0;

        for(int i = 0; i < 32; ++i){

            int count = 0;

            for(int num : nums){

                if(num & (1 << i))

                    count++;

            }

            if(count % 3 == 1)

                res ^= 1 << i;

        }

        return res;

    }

};

[剑指 Offer 57. 和为s的两个数字](https://leetcode-cn.com/problems/he-wei-sde-liang-ge-shu-zi-lcof/)

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]

class Solution {

public:

    vector<int> twoSum(vector<int>& nums, int target) {

        vector<int> res;

        int sum, i = 0, j = nums.size() - 1;

        while(i < j){

            int sum = nums[i] + nums[j];

            if(sum == target){

                res.push\_back(nums[i]);

                res.push\_back(nums[j]);

                break;

            }

            else if(sum > target){

                j--;

            }else{

                i++;

            }

        }

        return res;

    }

};

[剑指 Offer 57 - II. 和为s的连续正数序列](https://leetcode-cn.com/problems/he-wei-sde-lian-xu-zheng-shu-xu-lie-lcof/)

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]]

class Solution {

public:

    vector<vector<int>> findContinuousSequence(int target) {

        vector<vector<int>> res;

        vector<int> temp;

        int l = 1, r = 2;

        while(l < r){

            int sum = (l + r) \* (r - l + 1) / 2;

            if(sum == target){

                temp.clear();

                for(int i = l; i <= r; ++i){

                    temp.push\_back(i);

                }

                res.push\_back(temp);

                l++;

            }else if(sum < target){

                r++;

            }else{

                l++;

            }

        }

        return res;

    }

};

[剑指 Offer 58 - I. 翻转单词顺序](https://leetcode-cn.com/problems/fan-zhuan-dan-ci-shun-xu-lcof/)

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。

class Solution {

public:

    string reverseWords(string s) {

        if(s.size() == 0)

            return s;

        int len = 0;

        string res = "";

        for(int i = s.size() - 1; i >= 0; --i){

            if(s[i] != ' ')

                len++;

            if(s[i] == ' ' && len != 0){

                res += s.substr(i + 1, len) + " ";

                len = 0;

                continue;

            }

        }

        if(len != 0){

            res += s.substr(0, len) + " ";

        }

        if(res.size() > 0){

            res.erase(res.size() - 1, 1);

        }

        return res;

    }

};

[剑指 Offer 58 - II. 左旋转字符串](https://leetcode-cn.com/problems/zuo-xuan-zhuan-zi-fu-chuan-lcof/)

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"

class Solution {

public:

    void reverse(string& s, int l, int r){

        while(l < r){

            swap(s[l], s[r]);

            l++;

            r--;

        }

    }

    string reverseLeftWords(string s, int n) {

        int size = s.size();

        if(size <= 1)

            return s;

        reverse(s, 0, n - 1);

        reverse(s, n, size - 1);

        reverse(s, 0, size - 1);

        return s;

    }

};

[剑指 Offer 59 - I. 滑动窗口的最大值](https://leetcode-cn.com/problems/hua-dong-chuang-kou-de-zui-da-zhi-lcof/)

给定一个数组 nums 和滑动窗口的大小 k，请找出所有滑动窗口里的最大值。

**示例:**

**输入:** *nums* = [1,3,-1,-3,5,3,6,7], 和 *k* = 3

**输出:** [3,3,5,5,6,7]

**解释:**

滑动窗口的位置 最大值

--------------- -----

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

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

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

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

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

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

class Solution {

public:

    vector<int> maxSlidingWindow(vector<int>& nums, int k) {

        vector<int> res;

        deque<int> q;

        for(int i = 0; i < nums.size(); ++i){

            while(!q.empty() && nums[i] > nums[q.back()])//每新增一个值就与前面的比较，比他小的统统丢掉，因为新增值肯定在滑动窗口内；

                q.pop\_back();

            if(!q.empty() && q.front() + k < i + 1) //判断是否在滑动窗口内

                q.pop\_front();

            q.push\_back(i);

            if(i >= k - 1)

                res.push\_back(nums[q.front()]);

        }

        return res;

    }

};

[剑指 Offer 59 - II. 队列的最大值](https://leetcode-cn.com/problems/dui-lie-de-zui-da-zhi-lcof/)

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]

class MaxQueue {

    queue<int> q;

    deque<int> d;

public:

    MaxQueue() {

    }

    int max\_value() {

        if(d.empty())

            return -1;

        return d.front();

    }

    void push\_back(int value) {

        while(!d.empty() && d.back() < value)

            d.pop\_back();

        q.push(value);

        d.push\_back(value);

    }

    int pop\_front() {

        if(q.empty())

            return -1;

        int ans = q.front();

        if(ans == d.front())

            d.pop\_front();

        q.pop();

        return ans;

    }

};

[剑指 Offer 60. n个骰子的点数](https://leetcode-cn.com/problems/nge-tou-zi-de-dian-shu-lcof/)

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]

class Solution {

public:

    vector<double> twoSum(int n) {

        int dp[70];

        memset(dp, 0, sizeof(dp));

        for(int i = 0; i <= 6; ++i){

            dp[i] = 1;

        }

        for(int i = 2; i <= n; ++i){

            for(int j = 6 \* i; j >= i; --j){

                dp[j] = 0;

                for(int cur = 1; cur <= 6; ++cur){

                    if(j - cur < i - 1)

                        break;

                    dp[j] += dp[j - cur];

                }

            }

        }

        int all = pow(6, n);

        vector<double> res;

        for(int i = n; i <= 6 \* n; ++i){

            res.push\_back(dp[i] \* 1.0 / all);

        }

        return res;

    }

};

[剑指 Offer 61. 扑克牌中的顺子](https://leetcode-cn.com/problems/bu-ke-pai-zhong-de-shun-zi-lcof/)

从扑克牌中随机抽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

class Solution {

public:

    bool isStraight(vector<int>& nums) {

        bool flag[15];

        memset(flag, 0, sizeof(flag));

        int ma = 0, mi = 14;

        for(auto num : nums){

            if(num == 0)

                continue;

            if(flag[num])

                return false;

            flag[num] = true;

            ma = max(ma, num);

            mi = min(mi, num);

        }

        return ma - mi < 5;

    }

};

class Solution {

public:

    bool isStraight(vector<int>& nums) {

        sort(nums.begin(), nums.end());

        int joker = 0;

        for(int i = 0; i < 4; i++){

            if(nums[i] == 0)

                joker++;

            else if(nums[i] == nums[i + 1]){

                return false;

            }

        }

        return nums[4] - nums[joker] < 5;

    }

};

[剑指 Offer 62. 圆圈中最后剩下的数字](https://leetcode-cn.com/problems/yuan-quan-zhong-zui-hou-sheng-xia-de-shu-zi-lcof/)

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。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

class Solution {

public:

    int lastRemaining(int n, int m) {

        int res = 0;

        for(int i = 2; i <= n; ++i){

            res = (m + res) % i;

        }

        return res;

    }

};

[剑指 Offer 63. 股票的最大利润](https://leetcode-cn.com/problems/gu-piao-de-zui-da-li-run-lcof/)

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。

**第一题，k = 1**

直接套状态转移方程，根据 base case，可以做一些化简：

dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])

dp[i][1][1] = max(dp[i-1][1][1], dp[i-1][0][0] - prices[i])

            = max(dp[i-1][1][1], -prices[i])

解释：k = 0 的 base case，所以 dp[i-1][0][0] = 0。

现在发现 k 都是 1，不会改变，即 k 对状态转移已经没有影响了。

可以进行进一步化简去掉所有 k：

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

dp[i][1] = max(dp[i-1][1], -prices[i])

直接写出代码：

int n = prices.length;

int[][] dp = new int[n][2];

for (int i = 0; i < n; i++) {

    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);

    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);

}

return dp[n - 1][0];

显然 i = 0 时 dp[i-1] 是不合法的。这是因为我们没有对 i 的 base case 进行处理。可以这样处理：

for (int i = 0; i < n; i++) {

    if (i - 1 == -1) {

        dp[i][0] = 0;

        // 解释：

        //   dp[i][0]

        // = max(dp[-1][0], dp[-1][1] + prices[i])

        // = max(0, -infinity + prices[i]) = 0

        dp[i][1] = -prices[i];

        //解释：

        //   dp[i][1]

        // = max(dp[-1][1], dp[-1][0] - prices[i])

        // = max(-infinity, 0 - prices[i])

        // = -prices[i]

        continue;

    }

    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);

    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);

}

return dp[n - 1][0];

第一题就解决了，但是这样处理 base case 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，其实不用整个 dp 数组，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间复杂度降到 O(1):

// k == 1

int maxProfit\_k\_1(int[] prices) {

    int n = prices.length;

    // base case: dp[-1][0] = 0, dp[-1][1] = -infinity

    int dp\_i\_0 = 0, dp\_i\_1 = Integer.MIN\_VALUE;

    for (int i = 0; i < n; i++) {

        // dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

        dp\_i\_0 = Math.max(dp\_i\_0, dp\_i\_1 + prices[i]);

        // dp[i][1] = max(dp[i-1][1], -prices[i])

        dp\_i\_1 = Math.max(dp\_i\_1, -prices[i]);

    }

    return dp\_i\_0;

}

class Solution {

public:

    int maxProfit(vector<int>& prices) {

        if(prices.size() == 0)

            return 0;

        int n = prices.size();

        int dp\_i\_0 = 0, dp\_i\_1 = INT\_MIN;

        for(int i = 0; i < n; ++i){

            dp\_i\_0 = max(dp\_i\_0, dp\_i\_1 + prices[i]);

            dp\_i\_1 = max(dp\_i\_1, -prices[i]);

        }

        return dp\_i\_0;

    }

};

**第二题，k = +infinity**

如果 k 为正无穷，那么就可以认为 k 和 k - 1 是一样的。可以这样改写框架：

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

            = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])

我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态了：

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])

直接翻译成代码：

    int maxProfit(vector<int>& prices) {

        int n = prices.size();

        int dp\_i\_0 = 0, dp\_i\_1 = INT\_MIN;

        for(int i = 0;i < n; ++i){

            int temp = dp\_i\_0;

            dp\_i\_0 = max(dp\_i\_0, dp\_i\_1 + prices[i]);

            dp\_i\_1 = max(dp\_i\_1, temp - prices[i]);

        }

        return dp\_i\_0;

    }

**第三题，k = +infinity with cooldown**

每次 sell 之后要等一天才能继续交易。只要把这个特点融入上一题的状态转移方程即可：

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])

解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1 。

翻译成代码：

    int maxProfit(vector<int>& prices) {

        int n = prices.size();

        int dp\_i\_0 = 0, dp\_i\_1 = INT\_MIN;

        int dp\_pre\_0 = 0;//这里是前天的数据;

        int temp = dp\_i\_0;//这是昨天的数据

        for(int i = 0; i < n; ++i){

            temp = dp\_i\_0;

            dp\_i\_0 = max(dp\_i\_0, dp\_i\_1 + prices[i]);

            dp\_i\_1 = max(dp\_i\_1, dp\_pre\_0 - prices[i]);//这里用的是前天的数据

            dp\_pre\_0 = temp;//昨天的数据给到前天；以此类推

        }

        return dp\_i\_0;

    }

**第四题，k = +infinity with fee**

每次交易要支付手续费，只要把手续费从利润中减去即可。改写方程：

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)

解释：相当于买入股票的价格升高了。

在第一个式子里减也是一样的，相当于卖出股票的价格减小了。

直接翻译成代码：

int maxProfit\_with\_fee(int[] prices, int fee) {

    int n = prices.length;

    int dp\_i\_0 = 0, dp\_i\_1 = Integer.MIN\_VALUE;

    for (int i = 0; i < n; i++) {

        int temp = dp\_i\_0;

        dp\_i\_0 = Math.max(dp\_i\_0, dp\_i\_1 + prices[i]);

        dp\_i\_1 = Math.max(dp\_i\_1, temp - prices[i] - fee);

    }

    return dp\_i\_0;

}

**第五题，k = 2**

k = 2 和前面题目的情况稍微不同，因为上面的情况都和 k 的关系不太大。要么 k 是正无穷，状态转移和 k 没关系了；要么 k = 1，跟 k = 0 这个 base case 挨得近，最后也没有存在感。

这道题 k = 2 和后面要讲的 k 是任意正整数的情况中，对 k 的处理就凸显出来了。我们直接写代码，边写边分析原因。

原始的动态转移方程，没有可化简的地方

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

按照之前的代码，我们可能想当然这样写代码（错误的）：

int k = 2;

int[][][] dp = new int[n][k + 1][2];

for (int i = 0; i < n; i++)

    if (i - 1 == -1) {

        //处理一下 base case

    }

    dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);

    dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);

}

return dp[n - 1][k][0];

为什么错误？我这不是照着状态转移方程写的吗？

还记得前面总结的「穷举框架」吗？就是说我们必须穷举所有状态。其实我们之前的解法，都在穷举所有状态，只是之前的题目中 k 都被化简掉了。比如说第一题，k = 1：

这道题由于没有消掉 k 的影响，所以必须要对 k 进行穷举：

int max\_k = 2;

int[][][] dp = new int[n][max\_k + 1][2];

for (int i = 0; i < n; i++) {

    for (int k = max\_k; k >= 1; k--) {

        if (i - 1 == -1) {

            //处理 base case

         }

        dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);

        dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);

    }

}

// 穷举了 n × max\_k × 2 个状态，正确。

return dp[n - 1][max\_k][0];

如果你不理解，可以返回第一点「穷举框架」重新阅读体会一下。

这里 k 取值范围比较小，所以可以不用 for 循环，直接把 k = 1 和 2 的情况全部列举出来也可以：

dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i])

dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i])

dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])

dp[i][1][1] = max(dp[i-1][1][1], -prices[i])

    int maxProfit(vector<int>& prices) {

        int dp\_i\_10 = 0, dp\_i\_11 = INT\_MIN;

        int dp\_i\_20 = 0, dp\_i\_21 = INT\_MIN;

        for(auto price : prices){

            dp\_i\_20 = max(dp\_i\_20, dp\_i\_21 + price);

            dp\_i\_21 = max(dp\_i\_21, dp\_i\_10 - price);

            dp\_i\_10 = max(dp\_i\_10, dp\_i\_11 + price);

            dp\_i\_11 = max(dp\_i\_11, -price);

        }

        return dp\_i\_20;

}

**第六题，k = any integer**

有了上一题 k = 2 的铺垫，这题应该和上一题的第一个解法没啥区别。但是出现了一个超内存的错误，原来是传入的 k 值会非常大，dp 数组太大了。现在想想，交易次数 k 最多有多大呢？

一次交易由买入和卖出构成，至少需要两天。所以说有效的限制 k 应该不超过 n/2，如果超过，就没有约束作用了，相当于 k = +infinity。这种情况是之前解决过的。

    int maxProfit\_k\_inf(vector<int>& prices) {

        int n = prices.size();

        int dp\_i\_0 = 0, dp\_i\_1 = INT\_MIN;

        for(int i = 0;i < n; ++i){

            int temp = dp\_i\_0;

            dp\_i\_0 = max(dp\_i\_0, dp\_i\_1 + prices[i]);

            dp\_i\_1 = max(dp\_i\_1, temp - prices[i]);

        }

        return dp\_i\_0;

    }

    int maxProfit(int k, vector<int>& prices) {

        int n = prices.size();

        if(k > n / 2)

            return maxProfit\_k\_inf(prices);

        vector<vector<vector<int>>> dp(n, vector<vector<int>>(k + 1, vector<int>(2, 0)));

        for(int i = 0; i < n; ++i){

            for(int j = k; j >= 1; --j){

                if(i - 1 == -1){

                    dp[i][j][0] = 0;

                    dp[i][j][1] = -prices[i];

                    continue;

                }

                dp[i][j][0] = max(dp[i - 1][j][0], dp[i - 1][j][1] + prices[i]);

                dp[i][j][1] = max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]);

            }

        }

        return dp[n - 1][k][0];

    }

[剑指 Offer 64. 求1+2+…+n](https://leetcode-cn.com/problems/qiu-12n-lcof/)

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

class Solution {

public:

    int sumNums(int n) {

        n && (n += sumNums(n - 1));

        return n;

    }

};

[剑指 Offer 65. 不用加减乘除做加法](https://leetcode-cn.com/problems/bu-yong-jia-jian-cheng-chu-zuo-jia-fa-lcof/)

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

int add(int a, int b) {

        return b ? add(a^b, (unsigned int)(a&b)<<1) : a;

    }

class Solution {

public:

    int add(int a, int b) {

        while (b) {

            int carry = (unsigned int)(a & b) << 1;

            a ^= b;

            b = carry;

        }

        return a;

    }

};

[剑指 Offer 66. 构建乘积数组](https://leetcode-cn.com/problems/gou-jian-cheng-ji-shu-zu-lcof/)

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 中的元素 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]

class Solution {

public:

    vector<int> constructArr(vector<int>& a) {

        int n = a.size();

        vector<int> L(n, 1);

        vector<int> R(n, 1);

        for(int i = 1; i < n; ++i){

            L[i] = L[i - 1] \* a[i - 1];

        }

        for(int i = n - 2; i >= 0; --i){

            R[i] = R[i + 1] \* a[i + 1];

        }

        for(int i = 0; i < n; ++i){

            L[i] = L[i] \* R[i];

        }

        return L;

    }

};

[剑指 Offer 67. 把字符串转换成整数](https://leetcode-cn.com/problems/ba-zi-fu-chuan-zhuan-huan-cheng-zheng-shu-lcof/)

写一个函数 StrToInt，实现把字符串转换成整数这个功能。不能使用 atoi 或者其他类似的库函数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 [−231,  231− 1]。如果数值超过这个范围，请返回  INT\_MAX (231− 1) 或 INT\_MIN (−231) 。

示例 1: 输入: "42" 输出: 42

示例 2: 输入: " -42" 输出: -42

解释: 第一个非空白字符为 '-', 它是一个负号。

  我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。

示例 3 : 输入: "4193 with words" 输出: 4193

解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。

示例 4: 输入: "words and 987" 输出: 0

解释: 第一个非空字符是 'w', 但它不是数字或正、负号。

因此无法执行有效的转换。

    int strToInt(string str) {

        int res = 0;

        int i = 0;

        int flag = 1;

        // 1. 检查空格

        while (str[i] == ' ') { i++; }

        // 2. 检查符号

        if (str[i] == '-') { flag = -1; }

        if (str[i] == '+' || str[i] == '-') { i++; }

        // 3. 计算数字

        while (i < str.size() && isdigit(str[i])) {

            int r = str[i] - '0';

            // ------ 4. 处理溢出，这是关键步骤 ------2^31 - 1 = 2 147 483 647

            if (res > INT\_MAX / 10 || (res == INT\_MAX / 10 && r > 7)) {

                return flag > 0 ? INT\_MAX : INT\_MIN;

            }

            // ------------------------------------

            res = res \* 10 + r;

            i++;

        }

        return flag > 0 ? res : -res;

    }

[剑指 Offer 68 - I. 二叉搜索树的最近公共祖先](https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-zui-jin-gong-gong-zu-xian-lcof/)

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。

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。

    TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

        while(root != NULL){

            if(root->val < p->val && root->val < q->val)

                root = root->right;

            else if(root->val > p->val && root->val > q->val)

                root = root->left;

            else

                break;

        }

        return root;

    }

    TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

        if(root->val < p->val && root->val < q->val)

            return lowestCommonAncestor(root->right, p, q);

        if(root->val > p->val && root->val > q->val)

            return lowestCommonAncestor(root->left, p, q);

        return root;

    }

[剑指 Offer 68 - II. 二叉树的最近公共祖先](https://leetcode-cn.com/problems/er-cha-shu-de-zui-jin-gong-gong-zu-xian-lcof/)

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。

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。

class Solution {

public:

    TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

        if(root == NULL || root == p || root == q)

            return root;

        TreeNode\* left = lowestCommonAncestor(root->left, p, q);

        TreeNode\* right = lowestCommonAncestor(root->right, p, q);

        if(left == NULL)

            return right;

        if(right == NULL)

            return left;

        return root;

    }

};