

## C++笔试读取输入的操作

控制输出精度

## 常用排序

### 算法框架技巧

dp

回溯 (DFS)

BFS

双向 BFS 优化

二分查找 (zx总结)

滑动窗口

股票问题

打家劫舍

区间问题

SUM问题

二叉树 一

背包问题

DP 序列问题

不同的子序列

最长重复子数组

贪心算法 跳跃游戏

DP 正则表达式

DP 高楼扔鸡蛋

DP 戳气球

智力题

KMP 背!

DP 最小代价构造回文串

链表相关 递归思维

二叉树 二

二叉树 三

二叉搜索树 一

二叉搜索树 二

扁平化嵌套列表迭代器

二叉树的最近公共节点

求完全二叉树的节点数

数据流中位数

被围绕的区域

并查集 背! ☆

LRU (least recently used) 背!

LFU (least frequently used) 背!

设计Twitter! 背!

单调栈 \* 3

滑动窗口的最大值 背!

双栈实现队列 & 队列实现栈

二分搜索 的妙用

O(1) 时间, 查找/删除数组中的任意元素

双指针秒杀 数组/链表删除元素 题

去除重复字母

位运算的妙用

阶乘题

计算素数/质数

n皇后

回溯解决 子集! 排列! 组合!

解数独

合法括号

密码锁 bfs

滑块拼图  
前缀和 数组 技巧  
差分 数组 技巧  
数组中的第K个最大元素  
分治算法  
高效进行模幂运算  
寻找缺失的元素  
寻找缺失和重复的元素  
随机算法之水塘抽样算法  
吃葡萄 几何法!  
煎饼 (烧饼) 排序  
字符串 (大数) 相加、相乘  
万能计算器  
有效的括号  
判断子序列 二分法  
考试座位  
完美矩形  
接雨水  
括号匹配 带\*  
二分法求sqrt ()  
二叉树的右视图  
48. 旋转图像  
岛屿数量  
二叉树的锯齿形层序遍历  
搜索旋转排序数组  
重排链表  
下一个排列  
删除排序链表中的重复元素 II  
二叉树的完全性检验  
合并K个升序链表  
求根节点到叶节点数字之和  
用 Rand7() 实现 Rand10()  
缺失的第一个正数  
排序链表  
堆排序  
冒泡排序  
选择排序  
c++哈希表的实现  
c++简单String类实现  
c++ vector 实现  
字符串转换整数 (atoi)  
寻找旋转排序数组中的最小值  
寻找两个正序数组的中位数  
二叉搜索树的第k大节点  
翻转字符串里的单词  
路径总和 II  
无重复字符的最长子串  
颜色分类  
字典序的第K小数字  
二叉搜索树与双向链表  
两两交换链表中的节点  
二叉树最大宽度  
旋转链表  
旋转数组  
乘积最大子数组  
圆圈中最后剩下的数字  
寻找峰值  
螺旋矩阵 II

Pow(x, n)  
最小栈  
环形链表 II  
x 的平方根  
最大数  
数组中重复的数据  
整数与IP地址间的转换  
多数元素  
回文数  
循环依赖检测  
验证回文串  
搜索二维矩阵  
旋转数组的最小数字  
整数反转  
重复的子字符串

```
#include <vector>
#include <list>
#include <map>
#include <set>
#include <deque>
#include <queue>
#include <stack>
#include <bitset>
#include <algorithm>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include <string>
#include <ctime>
#include <cassert>
#include <climits>
#include <cfloat>
#include <cctype>
using namespace std;

auto seed=std::chrono::system_clock::now().time_since_epoch().count();
std::shuffle(nums.begin(), nums.end(), default_random_engine(seed)); // #include
<ctime> <algorithm> <random> <chrono>
```

头文件<cctype>

**isalpha()**用来判断一个字符是否为字母，如果是字符则返回非零，否则返回零。  
**isalnum()**用来判断一个字符是否为数字或者字母，也就是说判断一个字符是否属于a~z | A~Z | 0~9  
**islower()**判断一个字符是否为小写字母，也就是是否属于a~z。  
**isupper()**判断一个字符是否为大写字母。  
**tolower()**函数是把字符串都转化为小写字母  
**toupper()**函数是把字符串都转化为大写字母

## C++笔试读取输入的操作

待整理：

[Link1](#); [Link2](#)

输入通常使用while+cin>>来进行，这种方式也可以直接读入整行的string；

需要注意的是，这种方式的话，如果使用的是getline，当遇到换行符的时候cin会直接停止继续输入；

```
while(cin>>a>>b);
// 主要是按照类型来进行cin，操作应该是通过空格来分割的。
// 按照我们执行的次数来进行指定次数的读取操作
以指定符号分割的字符串输入
char str[3][11];
cin.getline(str[0], 11, ','); //接收最多10个字符 ,以'，'作为结束符
cin.getline(str[1], 11, ',');
cin.getline(str[2], 11); //默认结束符 enter
```

## 控制输出精度

```
#include <iostream>
#include <iomanip>
using namespace std;

void main()
{
    double pi = 3.14159265;
    cout << pi << endl; // 默认以6精度，所以输出为 3.14159
    cout << setprecision(4) << pi << endl; // 改成4精度，所以输出为3.142
    cout << setprecision(8) << pi << endl; // 改成8精度，所以输出为3.1415927
    cout << fixed << setprecision(4) << pi << endl; // 加了fixed意味着是固定点方式显示，所以这里的精度指的是小数位，输出为3.1416
    cout << pi << endl; // fixed和setprecision的作用还在，依然显示3.1416
    cout.unsetf(ios::fixed); // 去掉了fixed，所以精度恢复成整个数值的有效位数，显示为3.142
    cout << pi << endl;
    cout.precision(6); // 恢复成原来的样子，输出为3.14159
    cout << pi << endl;
}
```

```
double n = 0.001010;
cout.precision(4);
cout << n << endl;
```

这个的输出结果就是0.00101。从第一个1开始为第一个有效位，连续输出4个有效位，最后一位的0被省略。这并没有达到我们要设置小数点后位数的要求，所以在查阅了资料后发现结合std::fixed使用可以控制小数点后的位数，使用方法如下：

```
double n = 0.001010;
cout.precision(4);
cout << fixed << n << endl;
```

这个的输出结果就是0.0010。

需要注意的是 cout.precision(4); 后cout的精度一直都是4个有效位，若设置了fixed，就一直控制小数点后的位数。目前还没有消除precision的方法，只能重新设置成新的有效位，但是针对fixed可是通过 cout.unsetf( ios::fixed );来消除。

## 常用排序

```
void mergeSort(int l ,int r,vector<int>& v ,vector<int>& temp){//归并排序
    if (l >= r) return;
    int m = l +(r-1)/2;
    mergeSort(l,m,v,temp);
    mergeSort(m+1,r,v,temp);
    int i = l,j=m+1;
    for(int k = l;k<=r;k++){
        temp[k]=v[k];
    }
    for(int k= l ;k<=r;k++){
        if(i==m+1||temp[i]>temp[j]) v[k]=temp[j++];
        else if(j==r+1||temp[i]<=temp[j]) v[k]=temp[i++];//等于 先放左
    }
    //return ;
}
```

```
vector<int> temp(v.size(),0);
mergeSort(0,v.size()-1, v, temp);
```

```
void quickSort(int l, int r, vector<int>& v){//快速排序
    if (l >= r) return;
    int i = l, j = r;
    while (i < j)
    {
        while (v[j] >= v[l] && i < j)j--;
        while (v[i] <= v[l] && i < j)i++;
        swap(v[i],v[j]);
    }
    swap(v[i], v[l]);
    quickSort(l, i - 1, v);//递归左边
    quickSort(i + 1, r, v); //递归右边
}

quickSort(0, v.size()-1,v);
```

向上取整是一个常用的算法技巧。大部分编程语言中，如果你想计算 M 除以 N， $M / N$  会向下取整，你想向上取整的话，可以改成  $(M+(N-1)) / N$

```
void preOrderRecur(Node* head) { //前
    if (head == nullptr) {
        return;
    }
    std::cout << head->value << ",";
    preOrderRecur(head->left);
    preOrderRecur(head->right);
}

void inOrderRecur(Node* head) {//中
    if (head == nullptr) {
```

```

    return;
}
inorderRecur(head->left);
std::cout << head->value << ",";
inorderRecur(head->right);
}

void posOrderRecur(Node* head) { //后
    if (head == nullptr) {
        return;
    }
    posOrderRecur(head->left);
    posOrderRecur(head->right);
    std::cout << head->value << ",";
}

```

## 算法框架技巧

### dp

计算机解决问题的方法就是穷举，问题是如何聪明的穷举。

**dp的关键要素：**求最值、重叠子问题（剪枝）、状态转移方程

**dp实现过程**（自底向上 避免剪枝）：

- 找状态，画状态转移树，写状态转移方程。
- 定义dp
- 明确最值选择
- 明确basecase 即开始的几个状态。

### 回溯 (DFS)

回溯问题，实际上就是一个决策树的遍历过程。你只需要思考 3 个问题：

- 1、**路径**：也就是已经做出的选择。
- 2、**选择列表**：也就是你当前可以做的选择。
- 3、**结束条件**：也就是到达决策树底层，无法再做选择的条件。

```

result = []
typestyle backtrack(路径, 选择列表):
    if (满足结束条件)
        result.push_back(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择

```

核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」

比如{1, 2, 3}全排列中，如果以2为开头 [2] 就是「路径」，记录你已经做过的选择；[1,3]`就是「选择列表」，表示你当前可以做出的选择；「结束条件」就是遍历到树的底层，在这里就是选择列表为空的时候\*\*。

定义的 `backtrack` 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层，其「路径」就是一个全排列。

```
for (选择 in 选择列表)
    // 做选择
    将该选择从选择列表移除
    路径.push_back(选择)
    backtrack(路径, 选择列表)
    // 撤销选择
    路径.pop_back(选择)
    将该选择再加入选择列表
```

写 `backtrack` 函数时，需要维护走过的「路径」和当前可以做的「选择列表」，当触发「结束条件」时，将「路径」记入结果集。

## BFS

我们写 BFS 算法都是用「队列」这种数据结构，每次将一个节点周围的所有节点加入队列。

BFS 相对 DFS 的最主要的区别是：BFS 找到的路径一定是最短的，但代价就是空间复杂度比 DFS 大很多

问题的本质就是让你在一幅「图」中找到从起点 `start` 到终点 `target` 的最近距离

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

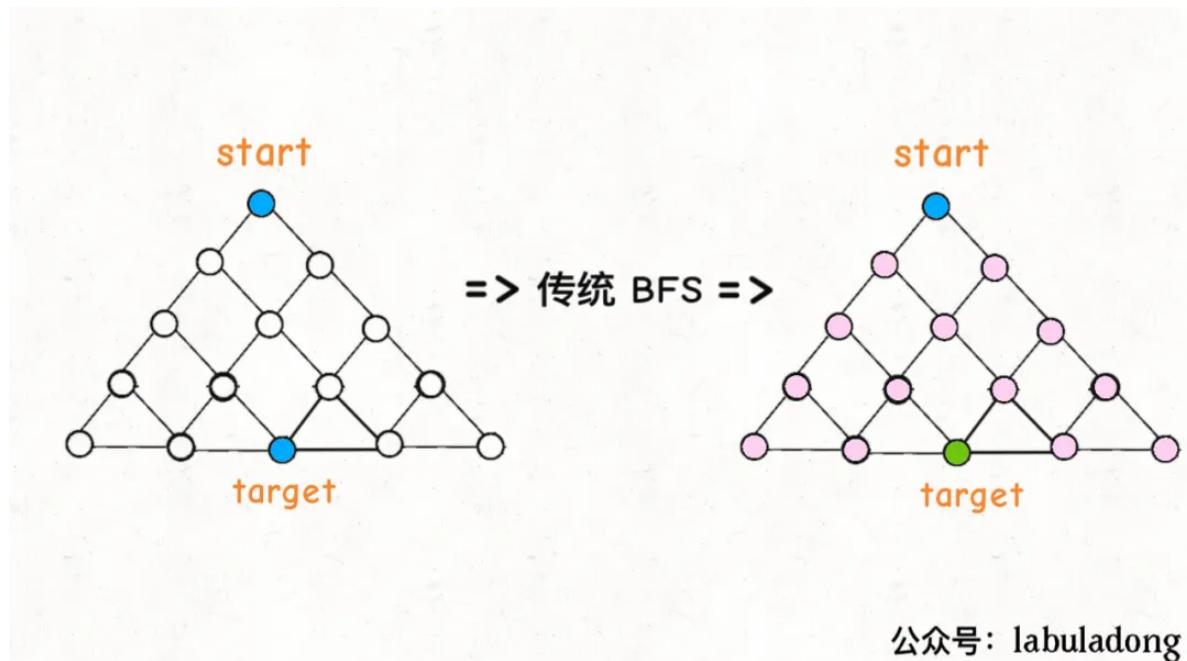
    q.push(start); // 将起点加入队列
    visited.insert(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz=q.size();
        //必须这么写 直接把q.size()写进for循环条件会由于循环中size改变导致跑不到step++
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.pop();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.push(x);
                    visited.insert(x);
                }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

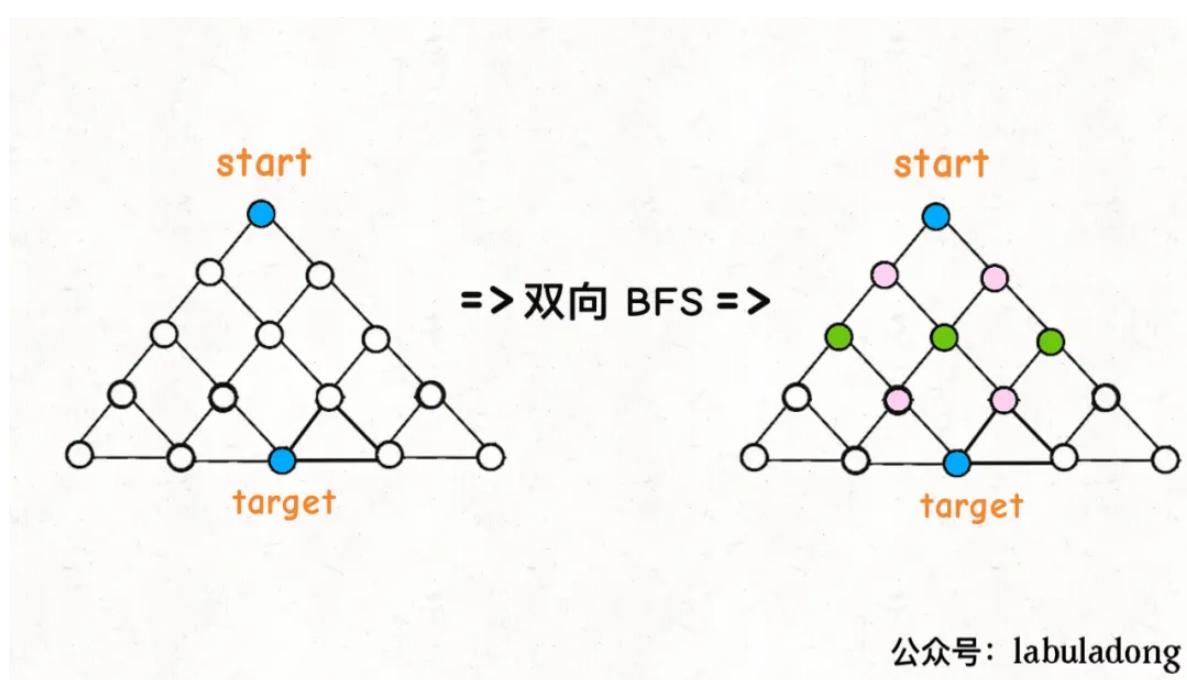
队列 q 就不说了，BFS 的核心数据结构；`cur.adj()` 泛指 `cur` 相邻的节点，比如说二维数组中，`cur` 上下左右四面的位置就是相邻节点；`visited` 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 `visited`。

## 双向 BFS 优化

传统的 BFS 框架就是从起点开始向四周扩散，遇到终点时停止；而双向 BFS 则是从起点和终点同时开始扩散，当两边有交集的时候停止。



公众号：labuladong



公众号：labuladong

## 二分查找 (zx总结)

记忆：左闭开始  $\geq$  <

其中  $\geq$  是左闭区间， $<$  是  $m < \text{target}$

找到值的方式就没什么好说的：我们直接当相等的时候返回就行，我们主要分析一下左侧边界和右侧边界的情况到底是什么含义：

- 左侧边界：满足某个条件的最左边（最小值），比如大于等于二的边界（最左边的那个2）；
- 右侧边界：满足某个条件的最右边（最大值），比如小于等于二的边界（最右边的那个2）；

结论1：两侧的边界是对称的：实际上画一下还是很容易分析出来的 ( $>=$  /  $\leq$  的 direction)：

```
左侧边界: >=
<: l=mid+1
>= : r = mid-1
检测左侧边界是否合理
return l;
```

```
右侧边界: <=
<=: l=mid+1
> : r = mid-1
检测右侧边界是否合理
return r;
```

结论2： $\geq$  换成 $>$ 或者 $\leq$ 换成 $<$ 只需要换一下相等的情况即可：

```
左侧边界: >
<=: l=mid+1
> : r = mid-1
检测左侧边界是否合理
return l;
```

```
右侧边界: <
<: l=mid+1
>= : r = mid-1
检测右侧边界是否合理
return r;
```

都这样写：**while(left <= right)**

**左侧边界： $\geq$  和右侧边界： $>$ 的区别是啥？**

左侧边界： $\geq$  指的是找的是第一个大于等于target

左侧边界： $>$  指的是找的是第一个大于等于target。

## 滑动窗口

滑动窗口算法的代码框架，我连在哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，还不会出边界问题：

```
/* 滑动窗口算法框架 */
void slidingwindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;
    
    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...
        
        /*** debug 输出的位置 ***/
        printf("window: [%d, %d]\n", left, right);
        /******/
        
        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
```

```

    // d 是将移出窗口的字符
    char d = s[left];
    // 左移窗口
    left++;
    // 进行窗口内数据的一系列更新
    ...
}

}

```

其中两处 ... 表示的更新窗口数据的地方，到时候你直接往里面填就行了。

I.<https://leetcode-cn.com/problems/minimum-window-substring/>

LeetCode 76 题，Minimum Window Substring，难度 Hard，我带大家看看它到底有多 Hard：

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

示例：

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

说明：

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

下面是完整代码：

```

string minWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;
    int left = 0, right = 0;
    int valid = 0;
    // 记录最小覆盖子串的起始索引及长度
    int start = 0, len = INT_MAX;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }
        // 判断左侧窗口是否要收缩
        while (valid == need.size()) {
            // 在这里更新最小覆盖子串
            if (right - left < len) {
                start = left;
                len = right - left;
            }
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(s[left - 1])) {
                window[s[left - 1]]--;
                if (window[s[left - 1]] < need[s[left - 1]])
                    valid--;
            }
        }
    }
    return len == INT_MAX ? "" : s.substr(start, len);
}

```

```

        }
        // d 是将移出窗口的字符
        char d = s[left];
        // 左移窗口
        left++;
        // 进行窗口内数据的一系列更新
        if (need.count(d)) {
            if (window[d] == need[d])
                valid--;
            window[d]--;
        }
    }
}

// 返回最小覆盖子串
return len == INT_MAX ?
    "" : s.substr(start, len);
}

```

需要注意的是，当我们发现某个字符在 `window` 的数量满足了 `need` 的需要，就要更新 `valid`，表示有一个字符已经满足要求。而且，你能发现，两次对窗口内数据的更新操作是完全对称的。

[II.https://leetcode-cn.com/problems/permutation-in-string/](https://leetcode-cn.com/problems/permutation-in-string/)

### 字符串的排列

给定两个字符串 `s1` 和 `s2`，写一个函数来判断 `s2` 是否包含 `s1` 的排列。

换句话说，第一个字符串的排列之一是第二个字符串的 **子串**。

- 1、移动 `left` 缩小窗口的时机是窗口大小大于 `t.size()` 时，因为排列嘛，显然长度应该是一样的。
- 2、当发现 `valid == need.size()` 时，就说明窗口中就是一个合法的排列，所以立即返回 `true`。

```

class Solution {
public:
    bool checkInclusion(string s1, string s2) {
        unordered_map<char, int>need ,window;
        for(char a: s1)need[a]++;
        int l=0,r=0;
        int valid=0;

        int len1 =s1.size(),len2=s2.size();
        while(r<len2){
            char c =s2[r];
            r++;
            if(need.count(c)){
                window[c]++;
                if(window[c]==need[c]){
                    valid++;
                }
            }
        }

        while(r-l>=len1){//>=!!!
            if(valid==need.size()){
                return true;
            }
            char d = s2[l];

```

```

    l++;
    if(need.count(d)){
        if(window[d]==need[d])
            valid--;
        window[d]--;
    }
}
return false;
};


```

III.<https://leetcode-cn.com/problems/find-all-anagrams-in-a-string/>

找到字符串中所有字母异位词

给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

所谓的字母异位词，不就是排列吗！相当于，输入一个串 s，一个串 t，找到 s 中所有 t 的排列，返回它们的起始索引。

```

class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        unordered_map<char, int> need,window;
        for(char c: p)need[c]++;
        int l=0,r=0;
        int valid=0;
        vector<int> res;
        while(r<s.size()){
            char c= s[r];
            r++;
            if(need.count(c)){
                window[c]++;
                if(window[c]==need[c]){
                    valid++;
                }
            }
        }
        while(r-l>=p.size()){//>=!!!
            if(valid==need.size()){
                res.push_back(l);
            }
            char d = s[l];
            l++;
            if(need.count(d)){
                if(window[d]==need[d]){
                    valid--;
                }
                window[d]--;
            }
        }
    }
};


```

```

        }
    }
    return res;
}
};

```

## 股票问题

I.<https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/>

买卖一次

$dp[i][0]$  表示第*i*天持有股票所得最多现金

$dp[i][1]$  表示第*i*天不持有股票所得最多现金

```

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        if (len == 0) return 0;
        vector<vector<int>> dp(len, vector<int>(2));
        dp[0][0] -= prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < len; i++) {
            dp[i][0] = max(dp[i - 1][0], -prices[i]);
            dp[i][1] = max(dp[i - 1][1], prices[i] + dp[i - 1][0]);
        }
        return dp[len - 1][1];
    }
};

```

II.<https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/>

买卖多次

- $dp[i][0]$  表示第*i*天持有股票所得现金。
- $dp[i][1]$  表示第*i*天不持有股票所得最多现金

```

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        vector<vector<int>> dp(len, vector<int>(2, 0));
        dp[0][0] -= prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < len; i++) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] - prices[i]); // 注意这里是
和121. 买卖股票的最佳时机唯一不同的地方。
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] + prices[i]);
        }
        return dp[len - 1][1];
    }
};

```

### III/IV. 买卖2/k次 <https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/>

使用二维数组  $dp[i][j]$ ：第*i*天的状态为j，所剩下的最大现金是 $dp[i][j]$

j的状态表示为：

- 0 表示不操作
- 1 第一次买入
- 2 第一次卖出
- 3 第二次买入
- 4 第二次卖出
- .....

大家应该发现规律了吧，除了0以外，偶数就是卖出，奇数就是买入。

题目要求是至多有K笔交易，那么j的范围就定义为  $2 * k + 1$  就可以了。

```
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {

        if (prices.size() == 0) return 0;
        vector<vector<int>> dp(prices.size(), vector<int>(2 * k + 1, 0));
        for (int j = 1; j < 2 * k; j += 2) {
            dp[0][j] = -prices[0];
        }
        for (int i = 1; i < prices.size(); i++) {
            for (int j = 0; j < 2 * k - 1; j += 2) {
                dp[i][j + 1] = max(dp[i - 1][j + 1], dp[i - 1][j] - prices[i]);
                dp[i][j + 2] = max(dp[i - 1][j + 2], dp[i - 1][j + 1] +
prices[i]);
            }
        }
        return dp[prices.size() - 1][2 * k];
    }
};
```

### V. 冷冻期 无限交易

#### [309. 最佳买卖股票时机含冷冻期](#)

$dp[i][j]$ ，第*i*天状态为j，所剩的最多现金为 $dp[i][j]$ 。

**其实本题很多同学搞的比较懵，是因为出现冷冻期之后，状态其实是比较复杂度**，例如今天买入股票、今天卖出股票、今天是冷冻期，都是不能操作股票的。具体可以区分出如下四个状态：

- 状态一：买入股票状态（今天买入股票，或者是之前就买入了股票然后没有操作）
- 卖出股票状态，这里就有两种卖出股票状态
  - 状态二：两天前就卖出了股票，度过了冷冻期，一直没操作，今天保持卖出股票状态
  - 状态三：今天卖出了股票
- 状态四：今天为冷冻期状态，但冷冻期状态不可持续，只有一天！

j的状态为：

- 0: 状态一
- 1: 状态二
- 2: 状态三
- 3: 状态四

```

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if (n == 0) return 0;
        vector<vector<int>> dp(n, vector<int>(4, 0));
        dp[0][0] -= prices[0]; // 持股票
        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], max(dp[i - 1][3], dp[i - 1][1]) -
prices[i]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][3]);
            dp[i][2] = dp[i - 1][0] + prices[i];
            dp[i][3] = dp[i - 1][2];
        }
        return max(dp[n - 1][3], max(dp[n - 1][1], dp[n - 1][2]));
    }
};

```

## 打家劫舍

I.<https://leetcode-cn.com/problems/house-robber/>

一排房子

决定dp[i]的因素就是第i房间偷还是不偷。

如果偷第i房间，那么 $dp[i] = dp[i - 2] + nums[i]$ ，即：第i-1房一定是不考虑的，找出下标i-2（包括i-2）以内的房屋，最多可以偷窃的金额为 $dp[i-2]$ 加上第i房间偷到的钱。

```

class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];
        vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); i++) {
            dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);
        }
        return dp[nums.size() - 1];
    }
};

```

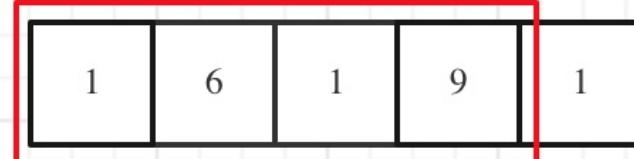
II.<https://leetcode-cn.com/problems/house-robber-ii/>

环形房子

- 情况二：考虑包含首元素，不包含尾元素

下标: 0 1 2 3 4

nums[i]:

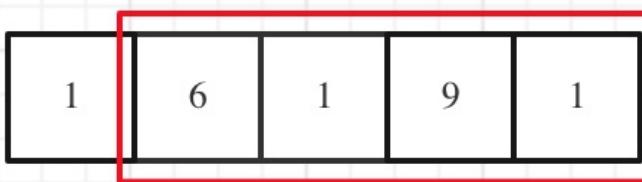


公众号:代码随想录

- 情况三: 考虑包含尾元素, 不包含首元素

下标: 0 1 2 3 4

nums[i]:



公众号:代码随想录

剩下的和[198.打家劫舍](#)就是一样的了。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];
        if (nums.size() == 2) return max(nums[0], nums[1]);
        int len = nums.size();
        int r1=robrange(nums,0,len-1);
        int r2=robrange(nums,1,len);
        return max(r1,r2);
    }
    int robrange(vector<int>& nums,int start,int end){
        vector<int> dp(nums.size(),0);
        dp[start]=nums[start];
        dp[start+1]=max(nums[start],nums[start+1]);
        for(int i =start+2;i<end;i++){
            dp[i]=max(dp[i-2]+nums[i],dp[i-1]);
        }
        return dp[end-1];
    }
};
```

III.<https://leetcode-cn.com/problems/house-robber-iii/>

树形房子

这里我们要求一个节点 偷与不偷的两个状态所得到的金钱, 那么返回值就是一个长度为2的数组。

参数为当前节点, 代码如下:

```
vector<int> robTree(TreeNode* cur) {
```

其实这里的返回数组就是dp数组。

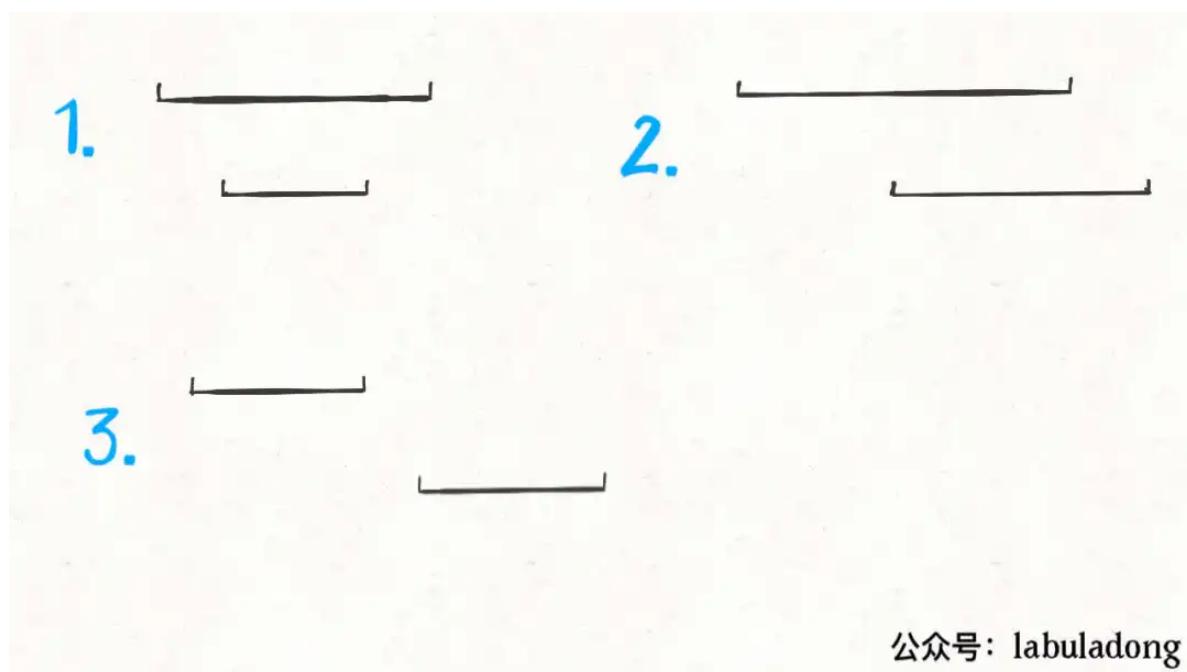
所以dp数组 (dp table) 以及下标的含义：下标为0记录不偷该节点所得到的最大金钱，下标为1记录偷该节点所得到的最大金钱。

```
class Solution {
public:
    int rob(TreeNode* root) {
        vector<int> res = robTree(root);
        return max(res[0], res[1]);
    }
    //不偷 0 偷 1
    vector<int> robTree(TreeNode* cur){
        if(cur==NULL) return {0,0};
        vector<int> left = robTree(cur->left);
        vector<int> right = robTree(cur->right);
        int val1 = cur->val + left[0] + right[0];//偷
        int val2 = max(left[0], left[1]) + max(right[0], right[1]);//不偷
        return {val2, val1};//val2是不偷 别写反位置!!!!!!!!!!!!!!
    }
};
```

## 区间问题

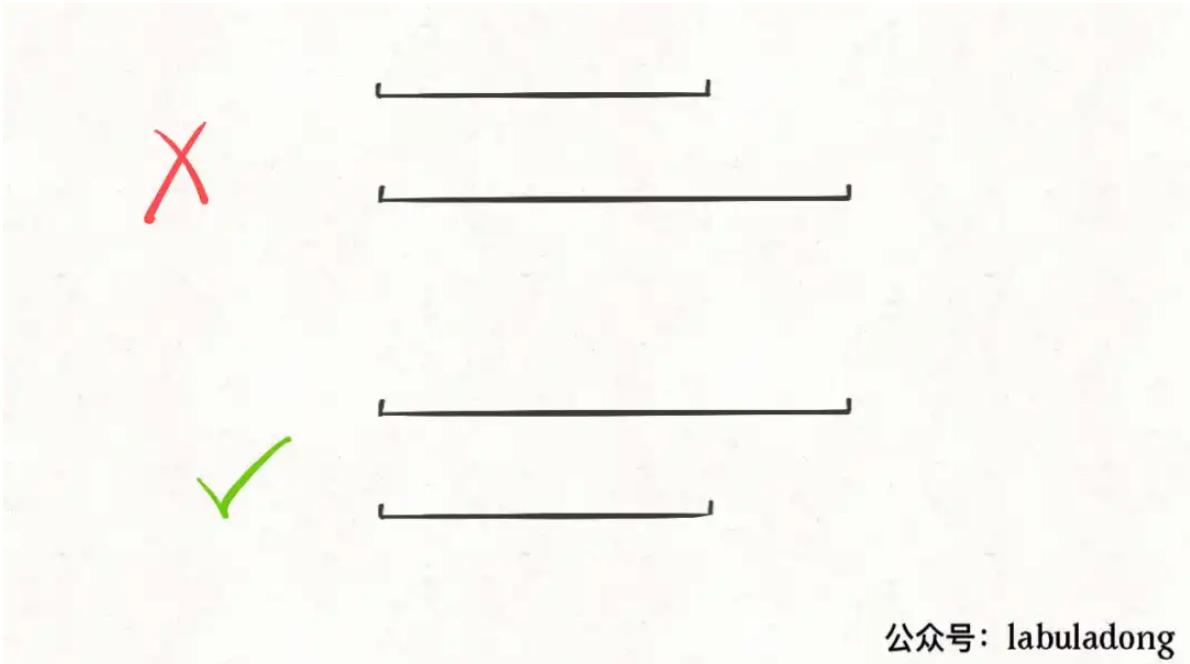
1. <https://leetcode-cn.com/problems/remove-covered-intervals/submissions/>

区间覆盖问题 返回剩余区间数



公众号: labuladong

起点升序排列，终点降序排列的目的是防止如下情况：保证长的那个区间在上面（按照终点降序），这样才会被判定为覆盖，否则会被错误地判定为相交



公众号: labuladong

```

class Solution {
public:
    int removeCoveredIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), [] (const vector<int> a, const
vector<int> b){
            return (a[0]<b[0]) || (a[0]==b[0]&&a[1]>=b[1]);
        });
        int left = intervals[0][0];
        int right = intervals[0][1];
        int res = intervals.size();
        for(int i = 1;i<intervals.size();i++){
            if(left<=intervals[i][0]&&right>=intervals[i][1]){//情况一, 找到了覆盖区
间。
                res--;
            }
            else if(right>intervals[i][0]&&right<intervals[i][1]){//情况二, 两个区
间可以合并, 成一个大区间。
                right=intervals[i][1];
            }
            else if(right<intervals[i][0]){//情况三, 两个区间完全不相交。
                left=intervals[i][0];
                right=intervals[i][1];
            }
        }
        return res;
    }
};

```

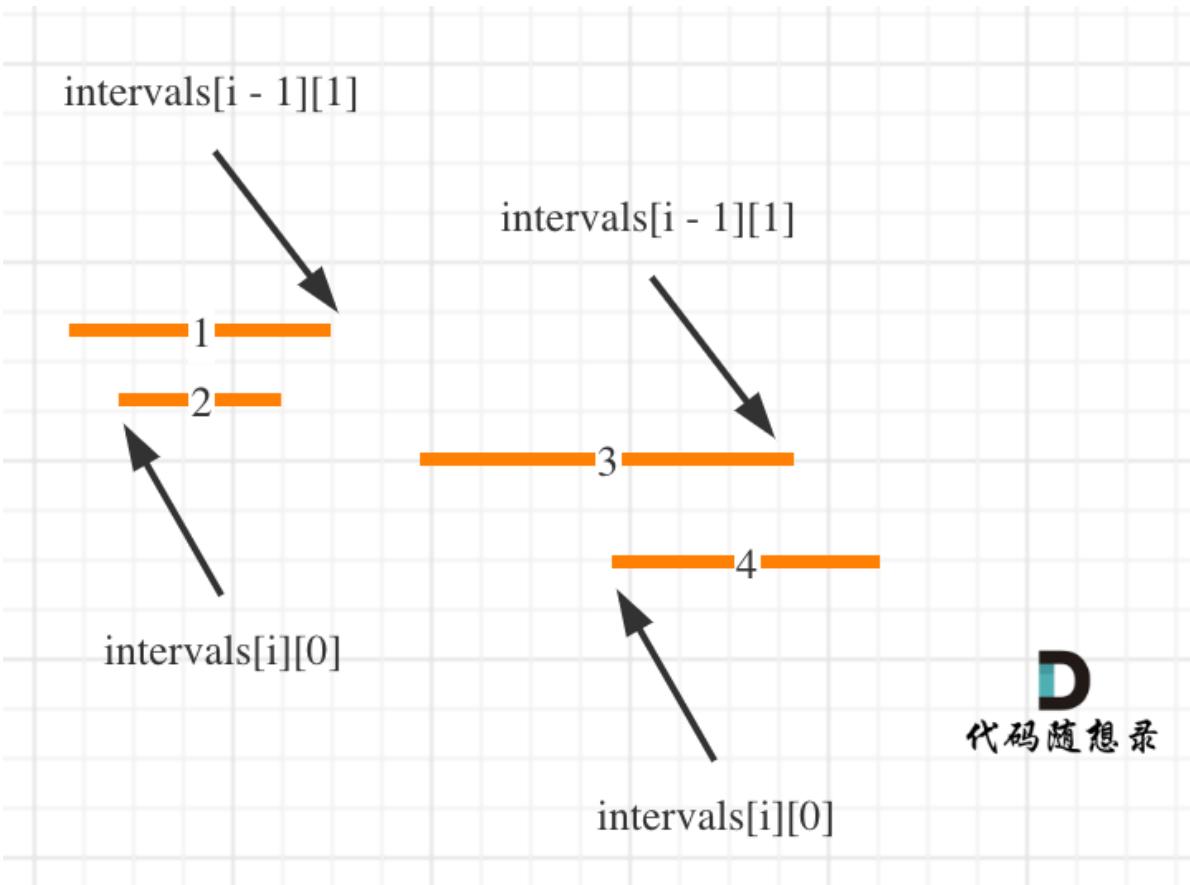
[II. https://leetcode-cn.com/problems/merge-intervals/submissions/](https://leetcode-cn.com/problems/merge-intervals/submissions/)

合并区间

几个相交区间合并后的结果区间  $x$ ,  $x.left$  一定是这些相交区间中  $left$  最小的,  $x.right$  一定是这些相交区间中  $right$  最大的。

$intervals[i]$  的左边界在  $intervals[i - 1]$  左边界和右边界范围内, 那么一定有重复!

这么说有点抽象, 看图: (注意图中区间都是按照左边界排序之后了)



```

class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), [] (const vector<int> &a, const
vector<int> &b){
            return (a[0]<b[0]) || (a[0]==b[0]&&a[1]>b[1]);
        });
        vector<vector<int>> res;
        res.push_back(intervals[0]);
        int len =intervals.size();
        for(int i =1;i<len;i++){
            if(res.back()[1]>=intervals[i][0]){// 合并区间
                res.back()[1]=max(res.back()[1],intervals[i][1]);
            }
            else if(res.back()[1]<intervals[i][0]){// 无法合并 加入下一区间
                res.push_back(intervals[i]);
            }
        }
        return res;
    }
};

```

III. <https://leetcode-cn.com/problems/non-overlapping-intervals/>

找到需要移除区间的最小数量，使剩余区间互不重叠。

难点：

- 难点一：一看题就有感觉需要排序，但究竟怎么排序，按左边界排还是右边界排。
- 难点二：排完序之后如何遍历，如果没有分析好遍历顺序，那么排序就没有意义了。
- 难点三：直接求重复的区间是复杂的，转而求最大非重复区间个数。

- 难点四：求最大非重复区间个数时，需要一个分割点来做标记。

按照右边界排序，就要从左向右遍历，因为右边界越小越好，只要右边界越小，留给下一个区间的空间就越大，所以从左向右遍历，优先选右边界小的。

每次取非交叉区间的时候，都是可右边界最小的来做分割点（这样留给下一个区间的空间就越大）

**记录非交叉区间的个数。最后用区间总数减去非交叉区间的个数就是需要移除的区间个数了。**

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), [] (const vector<int> &a, const vector<int> &b){// 按照区间右边界排序
            return (a[1] < b[1]);
        });
        int end = intervals[0][1];// 记录区间分割点
        int res = 1;// 记录非交叉区间的个数
        for(int i = 1 ;i<intervals.size();i++){
            if(end<=intervals[i][0]){//不相交不重叠
                end = intervals[i][1];
                res++;
            }
        }
        return intervals.size()-res;
    }
};
```

**最少的箭头射爆气球**<https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/>

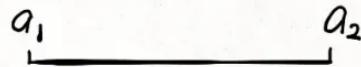
也是求非相交区间，唯一区别是2气球边界相等也算相交。

```
if(end<intervals[i][0]){//不相交不重叠
```

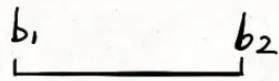
IV.<https://leetcode-cn.com/problems/interval-list-intersections/submissions/>

求两个区间列表的交集

首先，**对于两个区间**，我们用  $[a_1, a_2]$  和  $[b_1, b_2]$  表示在 A 和 B 中的两个区间，那么什么情况下这两个区间**没有交集**呢：



## 情况 1



## 情况 2



只有这两种情况，写成代码的条件判断就是这样：

```
if b2 < a1 or a2 < b1:  
    [a1,a2] 和 [b1,b2] 无交集
```

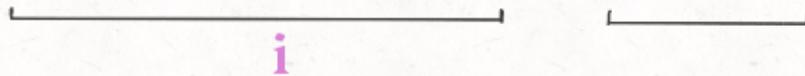
那么，什么情况下，两个区间存在交集呢？根据命题的否定，上面逻辑的否命题就是存在交集的条件：

```
# 不等号取反，or 也要变成 and  
if b2 >= a1 and a2 >= b1:  
    [a1,a2] 和 [b1,b2] 存在交集
```

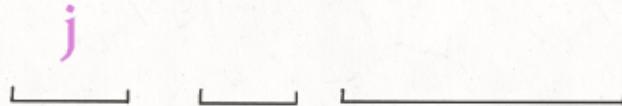
交集区间是  $[c1, c2]$ ，那么  $c1=\max(a1, b1)$ ， $c2=\min(a2, b2)$ ！

是否前进，只取决于  $a2$  和  $b2$  的大小关系：

A



B



```

while i < len(A) and j < len(B):
    # ...
    if b2 < a2:
        j += 1
    else:
        i += 1

```

```

class Solution {
public:
    vector<vector<int>> intervalIntersection(vector<vector<int>>& firstList,
vector<vector<int>>& secondList) {
        vector<vector<int>> res;
        int i=0,j=0;
        while(i<firstList.size()&&j<secondList.size()){
            if(firstList[i][1]>=secondList[j][0]&&secondList[j][1]>=firstList[i]
[0]){//相交时
                res.push_back({max(firstList[i][0],secondList[j]
[0]),min(firstList[i][1],secondList[j][1])});
            }
            if(firstList[i][1]<secondList[j][1]){//谁的右值小谁++
                i++;
            }
            else{
                j++;
            }
        }
        return res;
    }
};

```

## SUM问题

I.

2SUM

\*\*`nums` 中可能有多对儿元素之和都等于 `target`，请你的算法返回所有和为 `target` 的元素对儿，其中不能出现重复\*\*。

左右指针

```

vector<vector<int>> twoSumTarget(vector<int>& nums, int target {
    // 先对数组排序
    sort(nums.begin(), nums.end());
    vector<vector<int>> res;
    int lo = 0, hi = nums.size() - 1;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
        // 记录索引 lo 和 hi 最初对应的值
        int left = nums[lo], right = nums[hi];
        if (sum < target)      lo++;
        else if (sum > target) hi--;
        else {
            res.push_back({left, right});
        }
    }
}

```

```

        // 跳过所有重复的元素
        while (lo < hi && nums[lo] == left) lo++;
        while (lo < hi && nums[hi] == right) hi--;
    }
}

return res;
}

```

II. <https://leetcode-cn.com/problems/3sum/>

## 3SUM

找 `nums` 中和为 0 的三个元素，返回所有可能的三元组 (triple)

泛化一下题目，计算和为 `target` 的三元组

想找和为 `target` 的三个数字，那么对于第一个数字，`nums` 中的每一个元素 `nums[i]` 都有可能！

那么，确定了第一个数字之后，剩下的两个数字其实就是和为 `target - nums[i]` 的两个数字呗，那不就是 `twoSum` 函数解决的问题么？

```

/* 从 nums[start] 开始，计算有序数组
 * nums 中所有和为 target 的二元组 */
vector<vector<int>> twoSumTarget(
    vector<int>& nums, int start, int target) {
    // 左指针改为从 start 开始，其他不变
    int lo = start, hi = nums.size() - 1;
    vector<vector<int>> res;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
        int left = nums[lo], right = nums[hi];
        if (sum < target) lo++;
        else if (sum > target) hi--;
        else {
            res.push_back({left, right});
            // 跳过所有重复的元素
            while (lo < hi && nums[lo] == left) lo++;
            while (lo < hi && nums[hi] == right) hi--;
        }
    }
    return res;
}

/* 计算数组 nums 中所有和为 target 的三元组 */
vector<vector<int>> threeSumTarget(vector<int>& nums, int target) {
    // 数组得排个序
    sort(nums.begin(), nums.end());
    int n = nums.size();
    vector<vector<int>> res;
    // 穷举 threeSum 的第一个数
    for (int i = 0; i < n; i++) {
        // 对 target - nums[i] 计算 twoSum
        vector<vector<int>> tuples = twoSumTarget(nums, i + 1, target - nums[i]);
        // 如果存在满足条件的二元组，再加上 nums[i] 就是结果三元组
        for (vector<int>& tuple : tuples) {
            tuple.push_back(nums[i]);
            res.push_back(tuple);
        }
    }
}

```

```

        // 跳过第一个数字重复的情况，否则会出现重复结果
        while (i < n - 1 && nums[i] == nums[i + 1]) i++; //因为在for循环内此处跳出后i
   还会++ 和2SUM的处理略有区别
    }
    return res;
}

```

III.<https://leetcode-cn.com/problems/4sum/>

4SUM

4sum 完全就可以用相同的思路：穷举第一个数字，然后调用 3sum 函数计算剩下三个数，最后组合出和为 target 的四元组。

```

vector<vector<int>> fourSum(vector<int>& nums, int target) {
    // 数组需要排序
    sort(nums.begin(), nums.end());
    int n = nums.size();
    vector<vector<int>> res;
    // 穷举 foursum 的第一个数
    for (int i = 0; i < n; i++) {
        // 对 target - nums[i] 计算 threesum
        vector<vector<int>>
            triples = threeSumTarget(nums, i + 1, target - nums[i]);
        // 如果存在满足条件的三元组，再加上 nums[i] 就是结果四元组
        for (vector<int>& triple : triples) {
            triple.push_back(nums[i]);
            res.push_back(triple);
        }
        // foursum 的第一个数不能重复
        while (i < n - 1 && nums[i] == nums[i + 1]) i++;
    }
    return res;
}

/* 从 nums[start] 开始，计算有序数组
 * nums 中所有和为 target 的三元组 */
vector<vector<int>>
threeSumTarget(vector<int>& nums, int start, int target) {
    int n = nums.size();
    vector<vector<int>> res;
    // i 从 start 开始穷举，其他都不变
    for (int i = start; i < n; i++) {
        ...
    }
    return res;
}

```

IV.

1000SUM?

如果我让你求 100sum 问题，怎么办呢？其实我们可以观察上面这些解法，统一出一个 nsum 函数：

```

/* 注意：调用这个函数之前一定要先给 nums 排序 */
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {

```

```

        sort(nums.begin(), nums.end());
        return nSum(nums, target, 4, 0); //target==0&&n==4
    }

vector<vector<int>> nSum(vector<int>& nums, int target, int n, int start){
    int len = nums.size();
    vector<vector<int>> res;
    if(n<2 || len<n) return {};
    if(n==2){// 双指针那一套操作
        int left = start;
        int right = len-1;
        while(left<right){
            int sum = nums[left]+nums[right];
            int l=nums[left];
            int r=nums[right];
            if(sum<target)left++;
            else if(sum>target)right--;
            else{
                res.push_back({nums[left], nums[right]});
                while(left<right&&nums[left]==l)left++;
                while(left<right&&nums[right]==r)right--;
            }
        }
    }
    //return res;
}
else if(n>2){// n > 2 时, 递归计算 (n-1)sum 的结果
    for(int i = start;i<len;i++){
        auto arrys=nSum(nums,target-nums[i],n-1,i+1);//!
        for(auto a:arrys){
            a.push_back(nums[i]);
            res.push_back(a);
        }
        while(i<len-1&&nums[i]==nums[i+1])i++;
    }
    //return res;
}
return res;
};

```

## 二叉树一

树的问题就永远逃不开树的递归遍历框架这几行破代码：

```

/* 二叉树遍历框架 */
void traverse(TreeNode root) {
    // 前序遍历
    traverse(root.left)
    // 中序遍历
    traverse(root.right)
    // 后序遍历
}

```

快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后续遍历

I.<https://leetcode-cn.com/problems/invert-binary-tree/>

翻转二叉树

相当于每个节点的左右都反转

前序 后序 都ok， 中序会导致左子树反转完变成父节点的右子树再被反转回来

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) { // 此处后续
        if(root==NULL) return nullptr;
        invertTree(root->left);
        invertTree(root->right);
        TreeNode * l = root->left;
        TreeNode * r = root->right;
        TreeNode * temp = nullptr;
        root->left=r;
        root->right=l;
        return root;
    }
};
```

II.<https://leetcode-cn.com/problems/populating-next-right-pointers-in-each-node/>

填充每个节点的下一个右侧节点指针

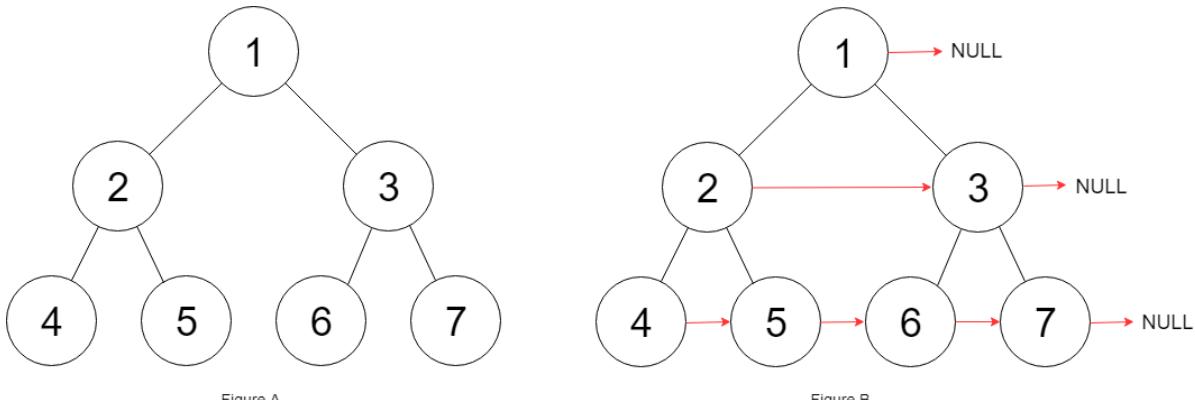


Figure A

Figure B

输入是一棵「完美二叉树」，形象地说整棵二叉树是一个**正三角形**，除了最右侧的节点 `next` 指针会指向 `null`，其他节点的右侧一定有相邻的节点。

节点 5 和节点 6 不属于同一个父节点，那么按照左连右的逻辑，它俩就没办法被穿起来，这是不符合题意的。

**二叉树的问题难点在于，如何把题目的要求细化成每个节点需要做的事情。**一个节点做不到，我们就给他安排两个节点，「将每一层二叉树节点连接起来」可以细化成「将每两个相邻节点都连接起来」：

```
// 主函数
Node connect(Node root) {
    if (root == null) return null;
    connectTwoNode(root.left, root.right);
    return root;
}

// 定义：输入两个节点，将它俩连接起来
void connectTwoNode(Node node1, Node node2) {
    if (node1 == null || node2 == null) {
```

```

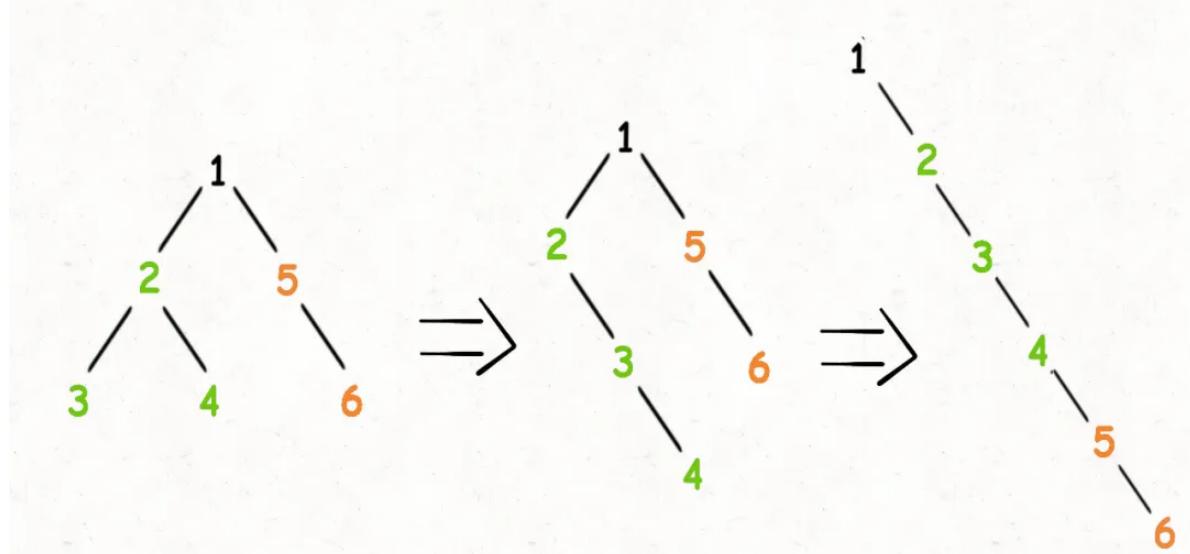
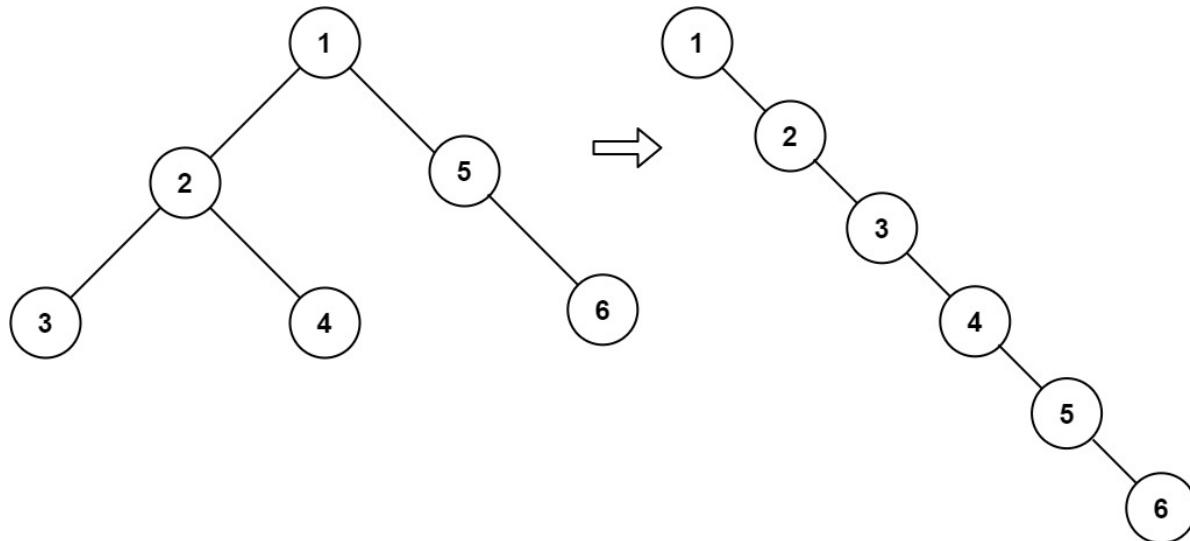
    return;
}
***** 前序遍历位置 *****/
// 将传入的两个节点连接
node1.next = node2;

// 连接相同父节点的两个子节点
connectTwoNode(node1.left, node1.right);
connectTwoNode(node2.left, node2.right);
// 连接跨越父节点的两个子节点
connectTwoNode(node1.right, node2.left);
}

```

III. <https://leetcode-cn.com/problems/flatten-binary-tree-to-linked-list/submissions/>

给你二叉树的根结点 `root`，将它展开为一个单链表



公众号：labuladong

```

// 定义：将以 root 为根的树拉平为链表
void flatten(TreeNode root) {
    // base case
    if (root == null) return;
    flatten(root.left);
    flatten(root.right);

```

```

***** 后序遍历位置 *****
// 1、左右子树已经被拉平成一条链表
TreeNode left = root.left;
TreeNode right = root.right;

// 2、将左子树作为右子树
root.left = null;
root.right = left;

// 3、将原先的右子树接到当前右子树的末端!
TreeNode cur = root;
while (cur.right != null) {
    cur = cur.right;
}
cur.right = right;
}

```

## 背包问题

背包问题大致的描述是什么：

给你一个可装载重量为  $w$  的背包和  $N$  个物品，每个物品有重量和价值两个属性。其中第  $i$  个物品的重量为  $wt[i]$ ，价值为  $val[i]$ ，现在让你用这个背包装物品，最多能装的价值是多少？

### 01 背包问题：

最基本的背包问题就是 01 背包问题：一共有  $N$  件物品，第  $i$  ( $i$  从 1 开始) 件物品的重量为  $w[i]$ ，价值为  $v[i]$ 。在总重量不超过背包承载上限  $W$  的情况下，能够装入背包的最大价值是多少？

### 完全背包问题：

完全背包与 01 背包不同就是每种物品可以有无限多个：一共有  $N$  种物品，每种物品有无限多个，第  $i$  ( $i$  从 1 开始) 种物品的重量为  $w[i]$ ，价值为  $v[i]$ 。在总重量不超过背包承载上限  $W$  的情况下，能够装入背包的最大价值是多少？

可见 01 背包问题与完全背包问题主要区别就是**物品是否可以重复选取**。

### 背包问题解法：

#### 01 背包问题：

如果是 01 背包，即数组中的元素不可重复使用，**外循环遍历  $nums$ ，内循环遍历  $target$ ，且内循环倒序**：

#### 完全背包问题：

- (1) 如果是完全背包，即数组中的元素可重复使用并且**不考虑元素之间顺序**， $nums$  放在外循环（保证  $nums$  按顺序），**target 在内循环。且内循环正序**。
- (2) 如果组合问题需**考虑元素之间的顺序**，需将  $target$  放在外循环，将  $nums$  放在内循环，**且内循环正序**。

I.<https://leetcode-cn.com/problems/partition-equal-subset-sum/submissions/>

给你一个**只包含正整数**的**非空**数组  $nums$ 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

给一个可装载重量为  $\text{sum}/2$  的背包和  $N$  个物品，每个物品的重量为  $\text{nums}[i]$ 。现在让你装物品，是否存在一种装法，能够恰好将背包装满？

按照背包问题的套路，可以给出如下定义：

$\text{dp}[i][j] = \text{x}$  表示，对于前  $i$  个物品，当前背包的容量为  $j$  时，若  $x$  为 `true`，则说明可以恰好将背包装满，若  $x$  为 `false`，则说明不能恰好将背包装满。

base case 就是  $\text{dp}[\dots][0] = \text{true}$  和  $\text{dp}[0][\dots] = \text{false}$ ，因为背包没有空间的时候，就相当于装满了，而当没有物品可选择的时候，肯定没办法装满背包。

如果不把  $\text{nums}[i]$  算入子集，或者说你不把这第  $i$  个物品装入背包，那么是否能够恰好装满背包，取决于上一个状态  $\text{dp}[i-1][j]$ ，继承之前的结果。

如果把  $\text{nums}[i]$  算入子集，或者说你把这第  $i$  个物品装入了背包，那么是否能够恰好装满背包，取决于状态  $\text{dp}[i-1][j-\text{nums}[i-1]]$ 。

```
bool canPartition(vector<int>& nums) {
    int sum=0;
    for(int s:nums)sum+=s;
    if(sum%2==1) return false;
    sum/=2;
    int n=nums.size();
    //dp[i][j] 表示对于前i个物品，当前背包的容量为j时 能否恰好装满
    vector<vector<bool>> dp(n+1,vector<bool>(sum+1,false)); //n+1!!!!sum+1!!!!
    for(int i = 0;i <= n;i++){
        dp[i][0]=true;
    }
    for(int i = 1;i<=n;i++){
        //for(int j = 0;j<=sum;j++){}//第i个物品的重量应该是nums[i-1]
        for(int j = sum;j>=0;j--){} //内循环逆序！！！
        if(nums[i-1]>j) dp[i][j]=dp[i-1][j]; //背包容量不足，不能装入第i个物品，所以只
        取决于上一状态
        else dp[i][j]=dp[i-1][j] | dp[i-1][j-nums[i-1]]; // 装入或不装入背包 有一个为
        true就行
    }
}
return dp[n][sum];
}
```

II.<https://leetcode-cn.com/problems/target-sum>

目标和

给你一个整数数组  $\text{nums}$  和一个整数  $\text{target}$ 。

向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个表达式，

返回运算结果等于  $\text{target}$  的不同 表达式 的数目。

我们想要的  $\text{target} = \text{正数和} - \text{负数和} = x - y$

而已知  $x$  与  $y$  的和是数组总和： $x + y = \text{sum}$

可以求出  $x = (\text{target} + \text{sum}) / 2 = t$

相当于求  $\text{nums}$  中有多少和为  $t$  的序列（01 背包）

```

int findTargetSumWays(vector<int>& nums, int target) {
    int sum=0;
    for(auto n:nums)sum+=n;
    if(target>sum || (sum+target)%2==1) return 0;
    int t = (sum+target)/2;
    vector<int> dp(t+1,0);
    dp[0]=1;
    for(int i = 1;i<=nums.size();i++){
        for(int j = t;j>=0;j--){//内循环逆序！！！
            if(nums[i-1]>j)dp[j]=dp[j];
            else {
                dp[j]=dp[j]+dp[j-nums[i-1]];
            }
        }
    }
    return dp[t];
}

```

III.<https://leetcode-cn.com/problems/coin-change-2/>

零钱兑换II

给定不同面额的硬币和一个总金额。写出函数来计算可凑成总金额的硬币组合数。假设每种面额的硬币无限个。

有一个背包，最大容量为 `amount`，有一系列物品 `coins`，每个物品的重量为 `coins[i]`，**每个物品的数量无限**。请问有多少种方法，能够把背包恰好装满？

这个问题和我们前面讲过的两个背包问题，有一个最大的区别就是，每个物品的数量是无限的，这也就是传说中的「完全背包问题」

使用 `coins` 中的前 `i` 个硬币的面值，若想凑出金额 `j`，有 `dp[i][j]` 种凑法。

```

int change(int amount, vector<int>& coins) {
    int n = coins.size();
    vector<vector<int>> dp(n+1,vector<int>(amount+1,0));
    for(int i = 0 ;i<=n;i++){
        dp[i][0]=1;
    }
    for(int i =1;i<=n;i++){
        for(int j=1;j<=amount;j++){//外n, 内target 内正 (是求组合)
            if(coins[i-1]>j){dp[i][j]=dp[i-1][j];}
            else{
                dp[i][j]=dp[i][j-coins[i-1]]+dp[i-1][j];//注意：与01背包不同 完全背包可重复coins[i-1]能再选。所以dp[i][j]此次选coins[i-1]的上一状态是dp[i][j-coins[i-1]]而不是dp[i-1][j-coins[i-1]]
            }
        }
    }
    return dp[n][amount];
}

```

## DP 序列问题

1、\*\*第一种思路模板是一个一维的 dp 数组\*\*：

```
int n = array.length;
int[] dp = new int[n];

for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        dp[i] = 最值(dp[i], dp[j] + ...)
    }
}
```

举个我们写过的例子[300. 最长递增子序列](#)，在这个思路中 dp 数组的定义是：

**在子数组 `array[0..i]` 中，以\*\*`array[i]`\*\*结尾的目标子序列（最长递增子序列）的长度是 `dp[i]`。**

为啥最长递增子序列需要这种思路呢？前文说得很清楚了，因为这样符合归纳法，可以找到状态转移的关系，这里就不具体展开了。

2、\*\*第二种思路模板是一个二维的 dp 数组\*\*：

```
int n = arr.length;
int[][] dp = new dp[n][n];

for (int i = 0; i < n; i++) {
    for (int j = 1; j < n; j++) {
        if (arr[i] == arr[j])
            dp[i][j] = dp[i][j] + ...
        else
            dp[i][j] = 最值(...)
    }
}
```

这种思路运用相对更多一些，尤其是涉及两个字符串/数组的子序列。本思路中 dp 数组含义又分为「只涉及一个字符串」和「涉及两个字符串」两种情况。

**2.1 涉及两个字符串/数组时**（比如最长公共子序列），dp 数组的含义如下：

**在子数组 `arr1[0..i]` 和子数组 `arr2[0..j]` 中，我们要求的子序列（最长公共子序列）长度为 `dp[i][j]`。**

**2.2 只涉及一个字符串/数组时**（比如本文要讲的最长回文子序列），dp 数组的含义如下：

**在子数组 `array[i..j]` 中，我们要求的子序列（最长回文子序列）的长度为 `dp[i][j]`。**

---

解决两个字符串的动态规划问题，一般都是用两个指针 `i, j` 分别指向两个字符串的最后，然后一步步往前走，缩小问题的规模。

I.<https://leetcode-cn.com/problems/edit-distance/>

编辑距离

给你两个单词 `word1` 和 `word2`，请你计算出将 `word1` 转换成 `word2` 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

插入一个字符

删除一个字符

替换一个字符

```
if s1[i] == s2[j]:  
    return dp(i - 1, j - 1) # 哪都不做  
# 解释:  
# 本来就相等，不需要任何操作  
# s1[0..i] 和 s2[0..j] 的最小编辑距离等于  
# s1[0..i-1] 和 s2[0..j-1] 的最小编辑距离  
# 也就是说 dp(i, j) 等于 dp(i-1, j-1)
```

如果 `s1[i] != s2[j]`，就要对三个操作递归了，稍微需要点思考：

```
dp(i, j - 1) + 1,      # 插入  
# 解释:  
# 我直接在 s1[i] 插入一个和 s2[j] 一样的字符  
# 那么 s2[j] 就被匹配了，前移 j，继续跟 i 对比  
# 操作数加一
```

```
dp(i - 1, j) + 1,      # 删除  
# 解释:  
# 我直接把 s[i] 这个字符删掉  
# 前移 i，继续跟 j 对比  
# 操作数加一
```

```
dp(i - 1, j - 1) + 1 # 替换  
# 解释:  
# 我直接把 s1[i] 替换成 s2[j]，这样它俩就匹配了  
# 同时前移 i, j 继续对比  
# 操作数加一
```

`dp[i][0]`：以下标*i*-1为结尾的字符串word1，和空字符串word2，最近编辑距离为`dp[i][0]`。

那么`dp[i][0]`就应该是*i*，对word1里的元素全部做删除操作，即：`dp[i][0] = i;`

同理`dp[0][j] = j;`

```
class Solution {  
public:  
    int minDistance(string word1, string word2) {  
        int res = 0;  
        int len1=word1.size(),len2=word2.size();  
        vector<vector<int>> dp(len1+1,vector<int> (len2+1));  
        for(int i=0;i<=len1;i++){dp[i][0]=i;}  
        for(int j=0;j<=len2;j++){dp[0][j]=j;}  
  
        for(int i=1;i<=len1;i++){  
            for(int j=1;j<=len2;j++){  
                if(word1[i-1]==word2[j-1])dp[i][j]=dp[i-1][j-1];  
                else{  
                    dp[i][j]=1+min(dp[i-1][j],min(dp[i][j-1],dp[i-1][j-1]));  
                }  
            }  
        }  
        return res;  
    }  
};
```

```

        }
    }
    return dp[len1][len2];
}

;
}

```

[II.https://leetcode-cn.com/problems/russian-doll-envelopes/submissions/](https://leetcode-cn.com/problems/russian-doll-envelopes/submissions/)

俄罗斯套娃信封问题

考虑固定一个维度，再在另一个维度上进行选择。

这个解法的关键在于，对于宽度  $w$  升序排列， $w$  相同的数对，要对其高度  $h$  进行降序排序。因为两个宽度相同的信封不能相互包含的，而逆序排序保证在  $w$  相同的数对中最多只选取一个

相当于另一个维度给定一个序列，我们需要找到一个最长的子序列，使得这个子序列中的元素严格单调递增，即上面要求的：

那么这个问题就是经典的「最长严格递增子序列」问题了：

<https://leetcode-cn.com/problems/longest-increasing-subsequence/submissions/>

```

//俄罗斯套娃信封问题：
int maxEnvelopes(vector<vector<int>>& envelopes) {
    sort(envelopes.begin(), envelopes.end(), //宽度w升序排列，高度h进行降序排序
         [] (const vector<int>& a, const vector<int>& b) {
            return (a[0] < b[0]) || (a[0] == b[0] && a[1] > b[1]);
        });
    int n=envelopes.size();

    vector<int> dp(n, 1);
    for(int i=1;i<n;i++){
        for(int j=0;j<i;j++){
            if(envelopes[j][1]<envelopes[i][1]){
                dp[i]=max(dp[i],dp[j]+1);
            }
        }
    }
    return *max_element(dp.begin(), dp.end());
}

//最长严格递增子序列：
int lengthOfLIS(vector<int>& nums) {
    int n=nums.size();

    vector<int> dp(n, 1);
    for(int i=1;i<n;i++){
        for(int j=0;j<i;j++){
            if(nums[j]<nums[i]){
                dp[i]=max(dp[i],dp[j]+1);
            }
        }
    }
    return *max_element(dp.begin(), dp.end());
}

```

### III. <https://leetcode-cn.com/problems/maximum-subarray/>

#### 最大子数和

给定一个整数数组 `nums`，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

以 `nums[i]` 为结尾的「最大子数组和」为 `dp[i]`

```
int maxSubArray(vector<int>& nums) {
    int n = nums.size();
    if(n==0) return 0;
    if(n==1) return nums[0];
    vector<int> dp(n);
    dp[0]=nums[0];
    for(int i =1;i<n;i++){
        dp[i]=max(dp[i-1]+nums[i],nums[i]);
    }
    return *max_element(dp.begin(),dp.end());
}
```

### IV. <https://leetcode-cn.com/problems/longest-common-subsequence/>

#### 最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在，返回 0。

`dp[i][j]` 表示 `text1` 前 `i` 个字符与 `text2` 前 `j` 个字符的公共子序列长度。

动态规划的边界情况是：当 `i=0` 或 `j=0` 时，`dp[i][j]=0`。（数组为空，肯定没公共子序列）

`text1[i-1]==text2[j-1]` 时，将这两个相同的字符称为公共字符。`dp[i][j]=dp[i-1][j-1]+1`

`text1[i-1]!=text2[j-1]` 时，

- `text1[i-1]` 不在公共序列，则 `text1[i-2]` 与 `text2[j-1]` 比，即 `dp[i][j]=dp[i-1][j]`
- `text2[j-1]` 不在公共序列，则 `text2[j-1]` 与 `text1[i-1]` 比，即 `dp[i][j]=dp[i][j-1]`

(都不在公共子序列中的情况被包含在上述两种情况中)

取最大：`dp[i][j]=max(dp[i-1][j],dp[i][j-1])`

```
int longestCommonSubsequence(string text1, string text2) {
    int n1=text1.size(),n2=text2.size();
    if(n1==0||n2==0) return 0;
    vector<vector<int>> dp(n1+1,vector<int>(n2+1,0));
    for(int i=1;i<=n1;i++){
        for(int j=1;j<=n2;j++){
            if(text1[i-1]==text2[j-1]){
                dp[i][j]=dp[i-1][j-1]+1;
            }else{
                dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
            }
        }
    }
    return dp[n1][n2];
}
```

V. <https://leetcode-cn.com/problems/longest-palindromic-subsequence/submissions/>

最长回文子序列

先回顾最长回文子串<https://leetcode-cn.com/problems/longest-palindromic-substring/submissions/>

给你一个字符串  $s$ ，找到  $s$  中最长的回文子串。

回文串就是正着读和反着读都一样的字符串

解决该类问题的核心是双指针。

寻找回文串的问题核心思想是：从中间开始向两边扩散来判断回文串：

```
for 0 <= i < len(s):
    找到以 s[i] 为中心的回文串
    更新答案
```

但是呢，我们刚才也说了，回文串的长度可能是奇数也可能是偶数，如果是 `abba` 这种情况，没有一个中心字符，上面的算法就没辙了。所以我们可以修改一下：

```
for 0 <= i < len(s):
    找到以 s[i] 为中心的回文串（即：找到以 s[i] 和 s[i] 为中心的回文串）！！
    找到以 s[i] 和 s[i+1] 为中心的回文串
    更新答案
```

```
string res;

string longestPalindrome(string s) {
    for(int i=0;i<s.size();i++){
        string s1= palindrome(s, i, i); //依次对每个字符串做回文串长度判断
        string s2= palindrome(s, i, i+1); //依次对每个字符串与其下一个做回文串长度判断
        res = res.size()>s1.size()?res:s1; //依次比较出最大值
        res = res.size()>s2.size()?res:s2;
    }
    return res;
}

string palindrome(string s,int l,int r){
    while(l>=0&&r<s.size()&&s[l]==s[r]){
        //同步判断 前者防止越界 后者规定扩张条件
        //判断条件这里注意别写出死循环了！
        l--,r++;
    }
    return s.substr(l+1,r-l-1);
}
```

回到子序列：

给定一个字符串  $s$ ，找到其中最长的回文子序列，并返回该序列的长度。

在子串  $s[i..j]$  中，最长回文子序列的长度为  $dp[i][j]$

想求  $dp[i][j]$ ，假设知道了子问题  $dp[i+1][j-1]$  的结果， $dp[i][j]$  的值  $s[i..j]$  中，最长回文子序列的长度 取决于  $s[i]$  和  $s[j]$  的字符

<i>i</i>	<i>i+1</i>			<i>j-1</i>	<i>j</i>
?	b	x	a	b	y

如果它俩相等，那么它俩加上  $s[i+1..j-1]$  中的最长回文子序列就是  $s[i..j]$  的最长回文子序列：

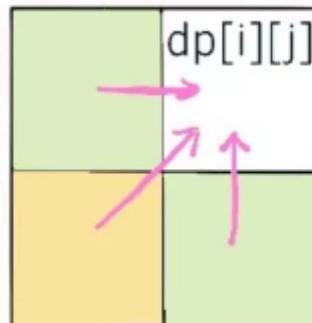
如果它俩不相等，说明它俩不可能同时出现在  $s[i..j]$  的最长回文子序列中，那么把它俩分别加入  $s[i+1..j-1]$  中，看看哪个子串产生的回文子序列更长即可：

以上两种情况写成代码就是这样：

```

if (s[i] == s[j])
    // 它俩一定在最长回文子序列中
    dp[i][j] = dp[i + 1][j - 1] + 2;
else
    // s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长?
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);

```



base case: 只有一个字符，显然最长回文子序列长度是 1，也就是  $dp[i][j] = 1, (i == j)$ 。

因为  $i$  肯定小于等于  $j$ ，所以对于那些  $i > j$  的位置，根本不存在什么子序列，应该初始化为 0。

1					
0	1				
0	0	1			
0	0	0	1		
0	0	0	0	1	

为了保证每次计算 `dp[i][j]`, 左、下、左下三个方向的位置已经被计算出来，只能斜着遍历或者反着遍历：

```
int longestPalindromeSubseq(string s) {
    int n=s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for(int i=0;i<n;i++) dp[i][i]=1;//只有一个字符，显然最长回文子序列长度是 1
    for(int i =n-1;i>=0;i--){
        for(int j = i+1;j<n;j++){ //j>i!!!!
            if(s[i]==s[j]){
                dp[i][j]=dp[i+1][j-1]+2;
            }else{
                dp[i][j]=max(dp[i+1][j],dp[i][j-1]);
            }
        }
    }
    return dp[0][n-1];
}
```

## 不同的子序列

### [115. 不同的子序列](#)

```
//dp[i][j]: 以i-1为结尾的s子序列中出现以j-1为结尾的t的个数为dp[i][j]。
int numDistinct(string s, string t) {
    vector<vector<uint64_t>> dp(s.size() + 1, vector<uint64_t>(t.size() +
1));
    for (int i = 0; i < s.size(); i++) dp[i][0] = 1;
    for (int j = 1; j < t.size(); j++) dp[0][j] = 0;
    for (int i = 1; i <= s.size(); i++) {
        for (int j = 1; j <= t.size(); j++) {
            if (s[i - 1] == t[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[s.size()][t.size()];
}
```

## 最长重复子数组

### [718. 最长重复子数组](#)

```

int findLength(vector<int>& A, vector<int>& B) {
    vector<vector<int>> dp (A.size() + 1, vector<int>(B.size() + 1, 0));
    int result = 0;
    for (int i = 1; i <= A.size(); i++) {
        for (int j = 1; j <= B.size(); j++) {
            if (A[i - 1] == B[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
            if (dp[i][j] > result) result = dp[i][j];
        }
    }
    return result;
}

```

## 贪心算法 跳跃游戏

I.<https://leetcode-cn.com/problems/jump-game/>

跳跃游戏I

给定一个非负整数数组 `nums`，你最初位于数组的 第一个下标。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

从左到右依次遍历当前元素能到达的最远距离

如果最终的最远距离 $\geq n-1$  则可到最后一个下标。

```

bool canJump(vector<int>& nums) {
    int n = nums.size();
    int far = 0;
    for(int i=0;i<n;i++){
        if(far>=i)far=max(far,i+nums[i]); //能到达的最远距离far内的元素才参与遍历
                                            //far外的元素到都到不了
    }
    return far>=n-1;
}

```

II.<https://leetcode-cn.com/problems/jump-game-ii/>

跳跃游戏II

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

**动态规划**

通过 532 ms 16.6 MB :

```

int jump(vector<int>& nums) { //动态规划方法
    int n=nums.size();

```

```

vector<int> dp(n, n); //dp[i]表示当前位置i要走到末尾最少需要的步数。
dp[n-1]=0;
for(int i = n-2;i>=0;i--){
    if(nums[i]==0) dp[i]=INT_MAX-1;
    else{
        for(int j=1;j<=nums[i];j++){
            if(i+j<n-1)
                dp[i] = min(dp[i+j]+1,dp[i]);
            else
                dp[i]=1;
        }
    }
}
return dp[0];
}

```

## 贪心算法

通过 12 ms 15.9 MB :

```

int jump(vector<int>& nums) { //贪心算法
    int n=nums.size();
    int end=0; //跳跃前可选的元素的边界
    int maxpos=0; //跳跃后可选元素的边界
    int jump=0; //跳跃次数
    for(int i=0;i<n-1;i++){ //在遍历数组时，我们不访问最后一个元素，这是因为在边界n-1正好为maxpos的情况下，会增加一次「不必要的跳跃次数」，然后把包括边界前的所有元素纳入可选范围，在跳跃前其实已经能到达边界了
        //if(i<=end)
        maxpos=max(maxpos, i+nums[i]); //遍历来得知能跳到的最远距离
        if(end==i){ //end==i表示 当前可选的元素遍历完了
            jump++; //进行跳跃
            end=maxpos; //将能到达的范围内元素纳入可选元素
        }
    }
    return jump;
}

```

## DP 正则表达式

<https://leetcode-cn.com/problems/regular-expression-matching/>

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '\*' 的正则表达式匹配。

'.' 匹配任意单个字符

'\*' 匹配零个或多个前面的那个元素

所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

$dp[i][j]$  表示 s 的前  $i$  个字符与 p 中的前  $j$  个字符是否能够匹配。

```

bool isMatch(string s, string p) {
    int n1=s.size(),n2=p.size();
    vector<vector<bool>> dp(n1+1,vector<bool>(n2+1,false));
    dp[0][0]=true;

```

```

for(int j = 2; j <= n2; j += 2){
    if(p[j-1] == '*'){
        dp[0][j] = dp[0][j-2]; //例如: s=={}, p=a*b*c* *全部取0
    }
}
for(int i=1;i<=n1;i++){
    for(int j=1;j<=n2;j++){
        if(p[j-1] != '*'){
            if(dp[i-1][j-1] && (p[j-1] == '.' || p[j-1] == s[i-1])) dp[i]
[j]=true;
        }else{
            if(dp[i][j-2]) dp[i][j]=true; /* 取0个
            else if(dp[i-1][j-2] && (s[i-1]==p[j-2] || p[j-2]=='.')) dp[i]
[j]=true; /* 取1个
        }
    }
}
return dp[n1][n2];
}

```

## DP 高楼扔鸡蛋

1. <https://leetcode-cn.com/problems/super-egg-drop/>

高楼扔鸡蛋 |

给你  $k$  枚相同的鸡蛋，并可以使用一栋从第  $1$  层到第  $n$  层共有  $n$  层楼的建筑。

已知存在楼层  $f$ ，满足  $0 \leq f \leq n$ ，任何从 高于  $f$  的楼层落下的鸡蛋都会碎，从  $f$  楼层或比它低的楼层落下的鸡蛋都不会破。

每次操作，你可以取一枚没有碎的鸡蛋并把它从任一楼层  $x$  扔下（满足  $1 \leq x \leq n$ ）。如果鸡蛋碎了，你就不能再使用它。如果某枚鸡蛋扔下后没有摔碎，则可以在之后的操作中 **重复使用** 这枚鸡蛋。

请你计算并返回要确定  $f$  确切的值的 **最小操作次数** 是多少？

**方法一：**

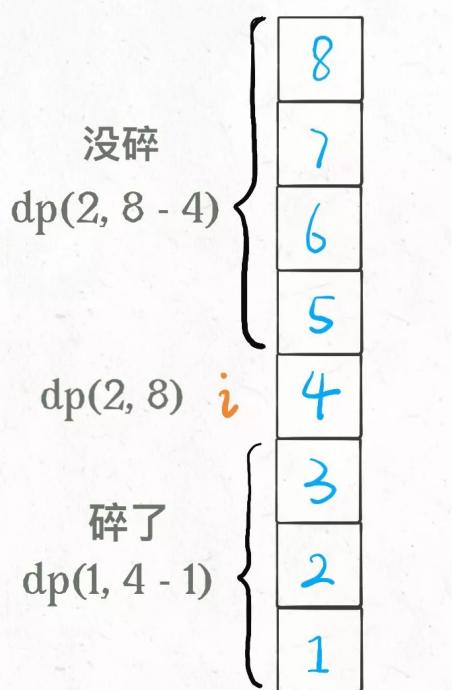
「状态」很明显，就是当前拥有的鸡蛋数  $K$  和需要测试的楼层数  $N$ 。

「选择」其实就是去选择哪层楼扔鸡蛋。

在第  $i$  层楼扔了鸡蛋之后，可能出现两种情况：鸡蛋碎了，鸡蛋没碎。**注意，这时候状态转移就来了：**

**如果鸡蛋碎了**，那么鸡蛋的个数  $K$  应该减一，搜索的楼层区间应该从  $[1..N]$  变为  $[1..i-1]$  共  $i-1$  层楼；

**如果鸡蛋没碎**，那么鸡蛋的个数  $K$  不变，搜索的楼层区间应该从  $[1..N]$  变为  $[i+1..N]$  共  $N-i$  层楼。



公众号: labuladong

因为我们要求的是**最坏情况**下扔鸡蛋的次数，所以鸡蛋在第*i*层楼碎没碎，取决于哪种情况的结果**更大**。

base case 很容易理解：当楼层数  $n$  等于 0 时，显然不需要扔鸡蛋；当鸡蛋数  $k$  为 1 时，只能线性扫描  $n$  层。

base case：



简化为把备忘录每个元素初始化成最大的尝试次数  $dp[i][j] = j$

```
class Solution {//超时
public:
    int superEggDrop(int k, int n) {
        int eggNum = k, floorNum = n;
        if(eggNum < 1 || floorNum < 1)
            return 0;
        //备忘录，存储k个鸡蛋，n层楼条件下的最优化尝试次数
        vector<vector<int>> dp(k+1, vector<int>(n+1));
        //把备忘录每个元素初始化成最大的尝试次数
        for(int i = 1; i <= eggNum; i ++){
            for(int j = 1; j <= floorNum; j++)
                dp[i][j] = j;
        }

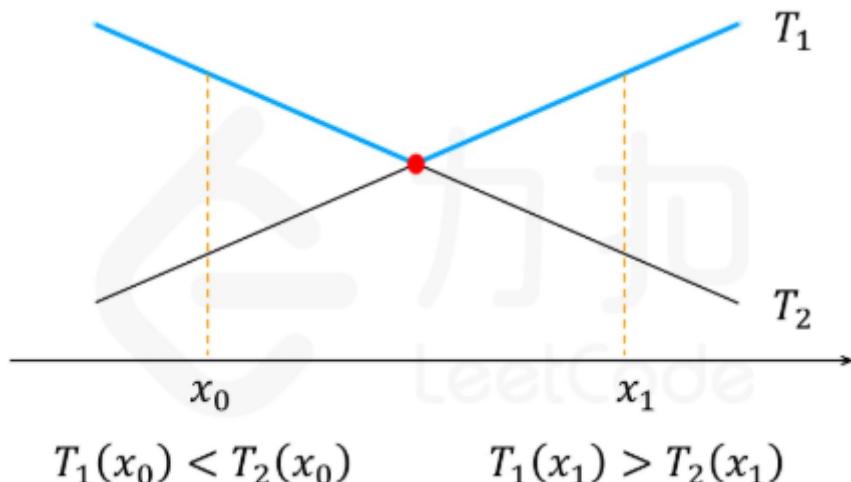
        for(int i=2;i<=k;i++){
            for(int j=1;j<=n;j++){
                //此处改为二分查找
                for(int l=1;l<j;l++){
                    dp[i][j]=min(dp[i][j],max(dp[i-1][l-1],dp[i][j-l])+1); //找到了
                }
            }
        }
    }
}
```

了最合适的楼层1 取大的

```
        }
    }
    return dp[k][n];
}
};
```

我们对转移方程的两个部分进行观察

- 设  $T_1 = dp[K-1][m-1]$ , 在  $K$  固定的情况下, 值随  $m$  的增加而单调递增
  - 设  $T_2 = dp[K][N-m]$ , 在  $K$  固定的情况下, 值随  $m$  的增加而单调递减  
因为我们要找到一个  $m$ , 使得  $\min(1 \leq m \leq N) (\max(dp[K-1][m-1], dp[K][N-m]) + 1)$



所以一定存在一个交点，满足我们的条件，由于必须取离散的值，因此我们需要找到最大的满足  $T_1(x) < T_2(x)$  的  $x_0$ ，以及最小的满足  $T_1(x) > T_2(x)$  的  $x_1$ 。然后比较这两个  $x_0$ 、 $x_1$  的  $dp[K][N]$ ，取较小的那个  $x$  作为丢第一枚鸡蛋的楼层。

```
class Solution {
public:
    int superEggDrop(int k, int n) {
        int eggNum = k, floorNum = n;
        if(eggNum < 1 || floorNum < 1)
            return 0;
        //备忘录，存储k个鸡蛋，n层楼条件下的最优化尝试次数
        vector<vector<int>> dp(k+1, vector<int>(n+1));
        //把备忘录每个元素初始化成线性扫面的最大的尝试次数
        for(int i = 1; i <= eggNum; i++){
            for(int j = 1; j <= floorNum; j++){
                dp[i][j] = j;
            }
        }

        for(int i=2;i<=k;i++){
            for(int j=2;j<=n;j++){
                //用二分查找 找 1~j层里面 最坏情况下次数最小的楼层: l
                int l=1,r=j;//二分查找 左边界>=
                while(l<=r){
                    int m = l+(r-l)/2;
                    int dp1=dp[i-1][m-1];//递增的dp1作为搜索的基准区间
                    int dp2=dp[i][j-m];//dp2选为target
                    if(dp1<dp2){ //< target
                        l=m+1;
                    }else if(dp1>=dp2){ //>= target
                        r=m-1;
                    }
                }
                dp[i][j] = l;
            }
        }
        return dp[k][n];
    }
};
```

```

        r=m-1;
    }
}

//dp[i][j]=dp[i-1][l-1]+1;//dp1 基准 直接用这行是可行的
//dp[i][j]=min(dp[i][j],max(dp[i-1][l-1],dp[i][j-1])+1); //此处无需
与自己对比 因为一定小于等于初始化的自己
dp[i][j]=max(dp[i-1][l-1],dp[i][j-1])+1; //便于理解 找到了最合适的楼层

```

### 1 取大的

```

    }
}
return dp[k][n];
}
};

```

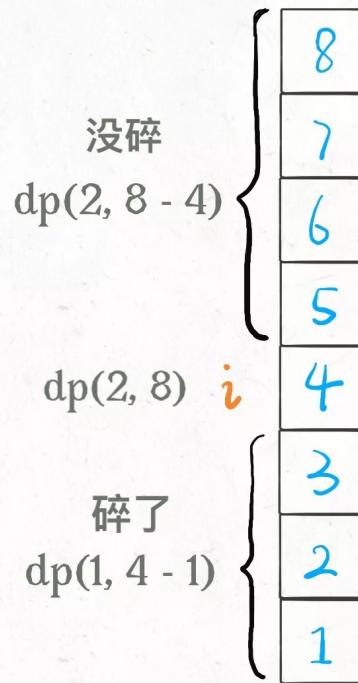
## 方法二：

### 数学法递归

方法三涉及逆向思维，是一种没见过就不太可能想出来，读过题解也很容易忘记的方法。

### 思路：

题目不是给你  $K$  鸡蛋， $N$  层楼，让你求最坏情况下最少的测试次数  $m$  吗？`while` 循环结束的条件是  $dp[K][m] == N$ ，也就是给你  $K$  个鸡蛋，允许测试  $m$  次，最坏情况下最多能测试  $N$  层楼。



公众号：labuladong

这个图描述的仅仅是某一个楼层  $i$ ，原始解法还得线性或者二分扫描所有楼层，要求最大值、最小值。但是现在这种  $dp$  定义根本不需要这些了，基于下面两个事实：

- 1、无论你在哪层楼扔鸡蛋，鸡蛋只可能摔碎或者没摔碎，碎了的话就测楼下，没碎的话就测楼上。
- 2、无论你上楼还是下楼，总的楼层数 = 楼上的楼层数 + 楼下的楼层数 + 1 (+1是当前测出来的这层)。

根据这个特点，可以写出下面的状态转移方程：

```
dp[k][m] = dp[k][m-1] + dp[k-1][m-1] + 1
```

\*\* `dp[k][m - 1]` 就是楼上的楼层数 \*\*，因为鸡蛋个数 `k` 不变，也就是鸡蛋没碎，扔鸡蛋次数 `m` 减一；

\*\* `dp[k - 1][m - 1]` 就是楼下的楼层数 \*\*，因为鸡蛋个数 `k` 减一，也就是鸡蛋碎了，同时扔鸡蛋次数 `m` 减一。

```
int superEggDrop(int k, int n) {
    // m 最多不会超过 N 次 (线性扫描)
    vector<vector<int>> dp(k+1, vector<int>(n+1));
    // base case:
    // dp[0][..] = 0
    // dp[..][0] = 0
    // 默认初始化数组都为 0
    int m = 0;
    while (dp[k][m] < n) {
        m++;
        for (int i = 1; i <= k; i++)
            dp[i][m] = dp[i][m - 1] + dp[i - 1][m - 1] + 1;
    }
    return m;
```

## DP 戳气球

<https://leetcode-cn.com/problems/burst-balloons/>

有 `n` 个气球，编号为 `0` 到 `n - 1`，每个气球上都标有一个数字，这些数字存在数组 `nums` 中。

现在要求你戳破所有的气球。戳破第 `i` 个气球，你可以获得 `nums[i - 1] * nums[i] * nums[i + 1]` 枚硬币。这里的 `i - 1` 和 `i + 1` 代表和 `i` 相邻的两个气球的序号。如果 `i - 1` 或 `i + 1` 超出了数组的边界，那么就当它是一个数字为 `1` 的气球。

求所能获得硬币的最大数量。

### 方法一 回溯（超时）

```
class Solution {//超时
public:
    int res=0;
    int maxCoins(vector<int>& nums) {
        return backtrack(nums, 0);
    }
    int backtrack(vector<int>& nums, int count){
        if(nums.size()==0){
            res=max(res, count);
        }
        for(int i=0;i<nums.size();i++){
            int t = nums[i];//为回溯做准备
            // delta -> 增量
            int delta = nums[i] * (i <= 0 ? 1 : nums[i - 1]) * (i >= nums.size() - 1 ? 1 : nums[i + 1]);
            nums.erase(nums.begin() + i);// 做选择 在 nums 中删除元素 nums[i]
            backtrack(nums, count + delta);// 递归回溯
            nums.insert(nums.begin() + i, t);// 撤销选择 将 t 还原到 nums[i]
        }
    }
}
```

```
    return res;  
}
```

## 方法二 DP

对问题进行一个简单的转化。题目说可以认为 `nums[-1] = nums[n] = 1`，那么我们先直接把这两个边界加进去，形成一个`*新的数组* points`

可以改变问题：**在一排气球 points 中，请你戳破气球 i 和气球 n+1 之间的所有气球（不包括 i 和 n+1），使得最终只剩下气球 i 和气球 n+1 两个气球，最多能够得到多少分？**

定义 dp 数组的含义：

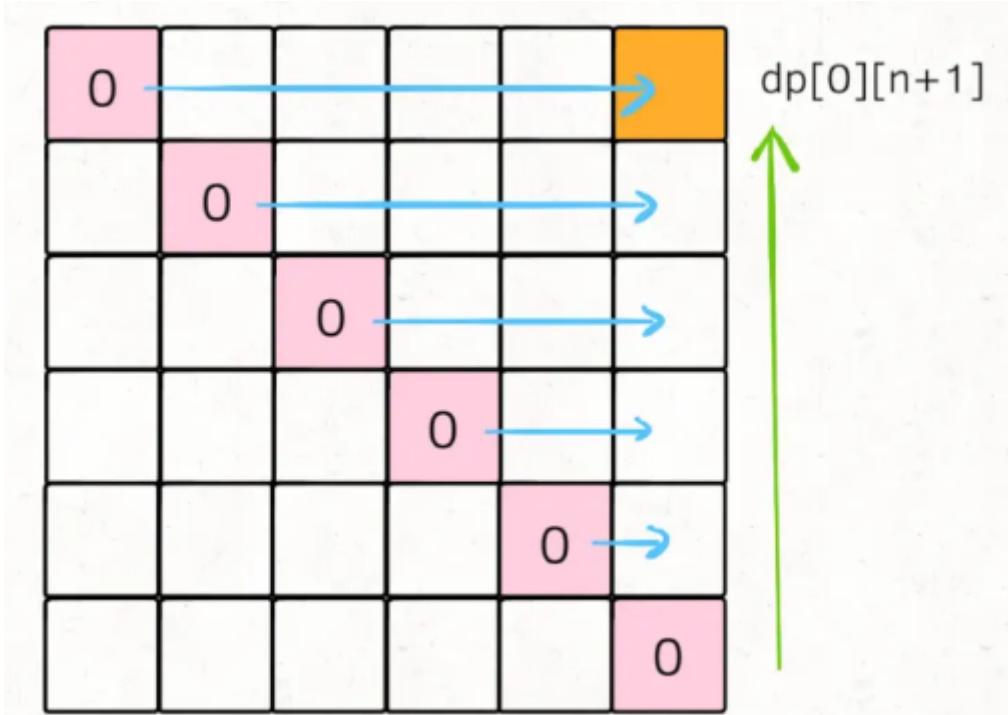
`** dp[i][j] = x 表示，戳破气球 i 和气球 j 之间（开区间，不包括 i 和 j）的所有气球，可以获得的最高分数为 x **。`

题目要求的结果就是 `dp[0][n+1]` 的值，而 base case 就是 `dp[i][i] = 0`，其中 `0 <= i <= n+1, j <= i+1`，因为这种情况下，开区间 `(i, j)` 中根本**没有气球**可以戳。

「正向思考」，就只能写出前文的回溯算法；「反向思考」，想一想气球 i 和气球 j 之间最后一个被戳破的气球可能是哪一个？气球 i 和气球 j 之间的所有气球都可能是最后被戳破的那个，不妨假设为 k。回顾动态规划的套路，这里其实已经找到了「状态」和「选择」：i 和 j 就是两个「状态」，最后戳破的那个气球 k 就是「选择」。

如果最后一个戳破气球 k，`dp[i][j]` 的值应该为：

```
dp[i][j] = dp[i][k] + dp[k][j]  
+ points[i]*points[k]*points[j]
```



```
class Solution {  
public:  
    int maxCoins(vector<int>& nums) {  
        int n = nums.size();  
        // 添加两侧的虚拟气球  
        vector<int> points(n+2);  
        points[0] = points[n+1] = 1;
```

```

        for(int i = 1; i <= n; i++) points[i] = nums[i - 1];

    // base case 初始化为 0
    vector<vector<int>> dp(n+2, vector<int>(n+2, 0));

    for(int i = n; i >= 0; i--) {
        for(int j = i+1; j < n+2; j++){//j>=i+1!!!
            for(int k=i+1; k < j; k++){//遍历之间最后戳破的气球 找最高分
                dp[i][j]=max(dp[i][j], dp[i][k]+dp[k]
[j]+points[i]*points[j]*points[k]);
            }
        }
    }
    return dp[0][n+1];
}
;

```

## 智力题

I.<https://leetcode-cn.com/problems/nim-game/>

Nim游戏

桌子上有一堆石头。

你们轮流进行自己的回合，你作为先手。

每一回合，轮到的人拿掉 1 - 3 块石头。

拿掉最后一块石头的人就是获胜者。

假设你们每一步都是最优解。请编写一个函数，来判断你是否可以在给定石头数量为 n 的情况下赢得游戏。如果可以赢，返回 true；否则，返回 false。

**我们解决这种问题的思路一般都是反着思考：**

如果我能赢，那么最后轮到我取石子的时候必须要剩下 1~3 颗石子，这样我才能一把拿完。

如何营造这样的一个局面呢？显然，如果对手拿的时候只剩 4 颗石子，那么无论他怎么拿，总会剩下 1~3 颗石子，我就能赢。

如何逼迫对手面对 4 颗石子呢？要想办法，让我选择的时候还有 5~7 颗石子，这样的话我就有把握让对方不得不面对 4 颗石子。

如何营造 5~7 颗石子的局面呢？让对手面对 8 颗石子，无论他怎么拿，都会给我剩下 5~7 颗，我就能赢。

这样一直循环下去，我们发现只要踩到 4 的倍数，就落入了圈套，永远逃不出 **4 的倍数**，而且一定会输。所以这道题的解法非常简单：

```

bool canWinNim(int n) {
    // 如果上来就踩到 4 的倍数，那就认输吧
    // 否则，可以把对方控制在 4 的倍数，必胜
    return n % 4 != 0;
}

```

**拓展：**通常的Nim游戏的定义是这样的：有若干堆石子，每堆石子的数量都是有限的，合法的移动是“选择一堆石子并拿走若干颗（不能不拿）”，如果轮到某个人时所有的石子堆都已经被拿空了，则判负（因为他此刻没有任何合法的移动）。

对于一个Nim游戏的局面 $(a_1, a_2, \dots, a_n)$ ，它是P-position（先手必输的局面）当且仅当 $a_1 \wedge a_2 \wedge \dots \wedge a_n = 0$ ，其中 $\wedge$ 表示异或(xor)运算。

II.<https://leetcode-cn.com/problems/stone-game/>

## 石子游戏

亚历克斯和李用几堆石子在做游戏。偶数堆石子排成一行，每堆都有正整数颗石子  $piles[i]$ 。

游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。

亚历克斯和李轮流进行，亚历克斯先开始。每回合，玩家从行的开始或结束处取走整堆石头。这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 true，当李赢得比赛时返回 false。

### 方法一 dp

$dp[i][j]$ : 表示先手玩家（亚历克斯）与后手玩家（李）在区间  $[i, j]$  之间互相拿，当前玩家比后手玩家多的最大石子个数。这是个差值，而且是个最大差值。

对于先手玩家，有两种拿法：

拿开头的  $piles[i]$ ：先手玩家手里有了  $piles[i]$ ，因为在区间  $[i + 1, j]$  中只能由后手玩家来选择，则  $dp[i+1][j]$  表示的是后手玩家在这个区间内，比先手玩家多的最大石子个数，反过来

$-dp[i+1][j]$  表示在这个区间内，先手玩家比后手玩家多的最大石子个数；

状态转移方程： $dp[i][j] = piles[i] + (-dp[i+1][j])$

拿结尾的  $piles[j]$ ：先手玩家手里有了  $piles[j]$ ，因为在区间  $[i, j - 1]$  中只能由后手玩家来选择，则  $dp[i][j-1]$  表示的是后手玩家在这个区间内，比先手玩家多的最大石子个数，反过来  $-dp[i][j-1]$  表示在这个区间内，先手玩家比后手玩家多的最大石子个数；

状态转移方程： $dp[i][j] = piles[j] + (-dp[i][j-1])$

在这两种情况下，选择先手玩家和后手玩家选择石子堆后，石子个数差更大的一种情况： $dp[i][j] = Math.max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1])$

初始化：当只有一个数时  $dp[i][i]$ ，此时先手玩家拿了必赢，所以  $dp[i][i] = piles[i]$

```
bool stoneGame(vector<int>& piles) {
    int n = piles.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for(int i=0;i<n;i++){
        dp[i][i]=piles[i];
    }

    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            dp[i][j]=max(piles[i]+(-dp[i+1][j]),piles[j]+(-dp[i][j-1]));
        }
    }
    return dp[0][n-1]>0;
}
```

## 方法二 智力

以 piles=[2, 1, 9, 5] 讲解

如果我们把这四堆石头按索引的奇偶分为两组，即第 1、3 堆和第 2、4 堆，那么这两组石头的数量一定不同，也就是说一堆多一堆少。因为石头的总数是奇数，不能被平分。

而作为第一个拿石头的人，你可以控制自己拿到所有偶数堆，或者所有的奇数堆。在第一步就观察好，奇数堆的石头总数多，还是偶数堆的石头总数多，然后步步为营，就一切尽在掌控之中了。

```
bool stoneGame(vector<int>& piles) {  
    return true;  
}
```

III.<https://leetcode-cn.com/problems/bulb-switcher/>

灯泡开关

初始时有 n 个灯泡处于关闭状态。

对某个灯泡切换开关意味着：如果灯泡状态为关闭，那该灯泡就会被开启；而灯泡状态为开启，那该灯泡就会被关闭。

第 1 轮，每个灯泡切换一次开关。即，打开所有的灯泡。

第 2 轮，每两个灯泡切换一次开关。即，每两个灯泡关闭一个。

第 i 轮，每 i 个灯泡切换一次开关。而第 n 轮，你只切换最后一个灯泡的开关。

找出 n 轮后有多少个亮着的灯泡。

假设只有 6 盏灯，而且我们只看第 6 盏灯。需要进行 6 轮操作对吧，请问对于第 6 盏灯，会被按下几次开关呢？这不难得出，第 1 轮会被按，第 2 轮，第 3 轮，第 6 轮都会被按。为什么第 1、2、3、6 轮会被按呢？因为  $6=1\times6=2\times3$ 。一般情况下，因子都是成对出现的，也就是说开关被按的次数一般是偶数次。

那么第 16 盏灯会被按几次？ $16=1\times16=2\times8=4\times4$ 。会被按 5 次。

假设现在总共有 16 盏灯，我们求 16 的平方根，等于 4，这就说明最后会有 4 盏灯亮着，它们分别是第  $1\times1=1$  盏、第  $2\times2=4$  盏、第  $3\times3=9$  盏和第  $4\times4=16$  盏。

```
int bulbswitch(int n) {  
    return sqrt(n);  
}
```

IV.<https://leetcode-cn.com/problems/4-keys-keyboard/>

4键键盘

**最优按键序列一定只有两种情况：**

要么一直按 A：A,A,...A（当 N 比较小）。

要么是这么一个形式：A,A,...C-A,C-C,C-V,C-V,...C-V（当 N 比较大时）。

因为字符数量少 (N 比较小) 时, C-A C-C C-V 这一套操作的代价相对比较高, 可能不如一个个按 A ; 而当 N 比较大时, 后期 C-V 的收获肯定很大。这种情况下整个操作序列大致是: **开头连接几个 A, 然后 C-A C-C 组合再接若干 C-V, 然后再 C-A C-C 接着若干 C-V, 循环下去。**

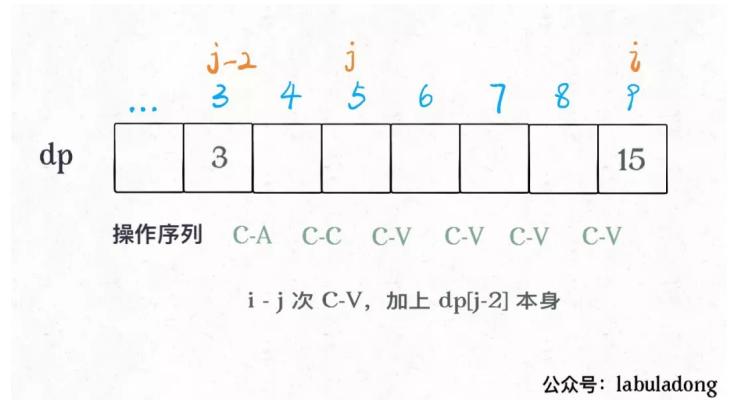
换句话说, 最后一次按键要么是 A 要么是 C-V 。明确了这一点, 可以通过这两种情况来设计算法:

```
int[] dp = new int[N + 1];
// 定义: dp[i] 表示 i 次操作后最多能显示多少个 A
for (int i = 0; i <= N; i++)
    dp[i] = max(
        这次按 A 键,
        这次按 C-V
    )
```

对于「按 A 键」这种情况, 就是状态  $i - 1$  的屏幕上新增了一个 A 而已, 很容易得到结果:

```
// 按 A 键, 就比上次多一个 A 而已
dp[i] = dp[i - 1] + 1;
```

刚才说了, 最优的操作序列一定是 C-A C-C 接着若干 C-V , 所以我们用一个变量 j 作为若干 C-V 的起点。那么 j 之前的 2 个操作就应该是 C-A C-C :



公众号: labuladong

```
int maxA(int n) {
    vector<int> dp(n+1);
    dp[0]=0;
    for(int i =1;i<=n;i++){
        dp[i]=dp[i-1]+1; //先算全是A
        for(int j =2;j<i;j++){ //遍历i次操作下 ctrl+v 从起点从0~i
            dp[i]=max(dp[i],dp[j-2]*(i-j+1));
        }
    }
    return dp[n];
}
```

## KMP 背!

[28. 实现 strStr\(\)](#)

背! 都是 int j = 0 -> 处理next -> while -> if -> ... 的结构

```

int j=0;
next...
for(int i=0or1;i<....size();i++){
    while(j>0&&...[i]!=...[j]){
        j=next[j-1];
    }
    if(...[i]==...[j]){
        j++;
    }
    ...
}

```

```

class Solution {
public:
    int strStr(string haystack, string needle) {
        if(needle.size()==0) return 0;

        int next[needle.size()];
        int j=0;
        getNext(needle, next);
        for(int i=0;i<haystack.size();i++){//int i=0
            while(j>0&&haystack[i]!=needle[j]){
                j=next[j-1];
            }
            if(haystack[i]==needle[j]){
                j++;
            }
            if(j==needle.size())return (i-needle.size()+1);
        }
        return -1;
    }

    void getNext(string needle, int* next){
        next[0]=0;
        int j=0;
        for(int i=1;i<needle.size();i++){//int i=1
            while(j>0&&needle[i]!=needle[j]){
                j=next[j-1];
            }
            if(needle[i]==needle[j]){
                j++;
            }
            next[i]=j;
        }
    }
};

```

## DP 最小代价构造回文串

<https://leetcode-cn.com/problems/minimum-insertion-steps-to-make-a-string-palindrome/>

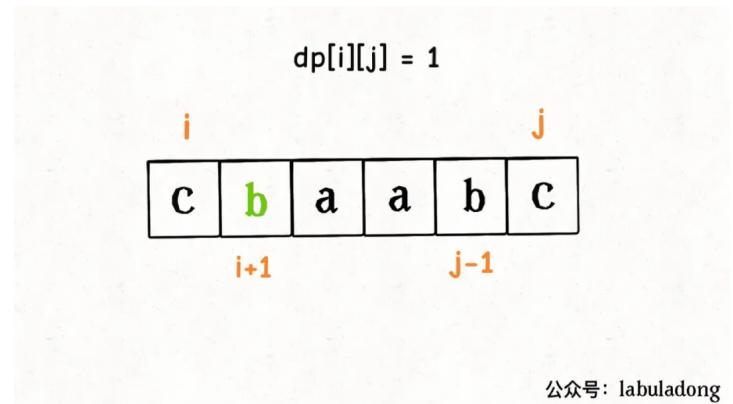
给你一个字符串  $s$ ，每一次操作你都可以在字符串的任意位置插入任意字符。

请你返回让  $s$  成为回文串的 **最少操作次数**。

\*\* $dp[i][j]$  的定义如下：对字符串  $s[i..j]$ ，最少需要进行  $dp[i][j]$  次插入才能变成回文串\*\*。

base case，当  $i == j$  时  $dp[i][j] = 0$ ，因为当  $i == j$  时  $s[i..j]$  就是一个字符，本身就是回文串，所以不需要进行任何插入操作。

如果  $s[i] == s[j]$  的话，我们不需要进行任何插入，只要知道如何把  $s[i+1..j-1]$  变成回文串即可：



翻译成代码就是这样：

```
if (s[i] == s[j]) {  
    dp[i][j] = dp[i + 1][j - 1];  
}
```

当  $s[i] != s[j]$  时，无脑插入两次肯定是可以让  $s[i..j]$  变成回文串，但是不一定是插入次数最少的，最优的插入方案应该被拆解成如下流程：

**步骤一，做选择，先将  $s[i..j-1]$  或者  $s[i+1..j]$  变成回文串。**

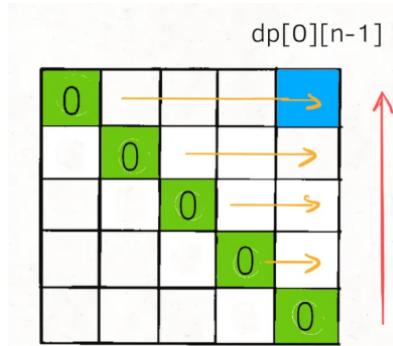
**步骤二，根据步骤一的选择，将  $s[i..j]$  变成回文。**

$s[i] != s[j]$  时的代码逻辑如下：

```
if (s[i] != s[j]) {  
    // 步骤一选择代价较小的  
    // 步骤二必然要进行一次插入  
    dp[i][j] = min(dp[i + 1][j], dp[i][j - 1]) + 1;  
}
```

综合起来，状态转移方程如下：

```
if (s[i] == s[j]) {  
    dp[i][j] = dp[i + 1][j - 1];  
} else {  
    dp[i][j] = min(dp[i + 1][j], dp[i][j - 1]) + 1;  
}
```



```

int minInsertions(string s) {
    int n=s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for(int i =n-1;i>=0;i--){
        for(int j =i+1;j<n;j++){ //j=i+1!!!!j>i
            if(s[i]==s[j])dp[i][j]=dp[i+1][j-1];
            else {
                dp[i][j]=min(dp[i+1][j], dp[i][j-1])+1; //+1!!!
            }
        }
    }
    return dp[0][n-1];
}

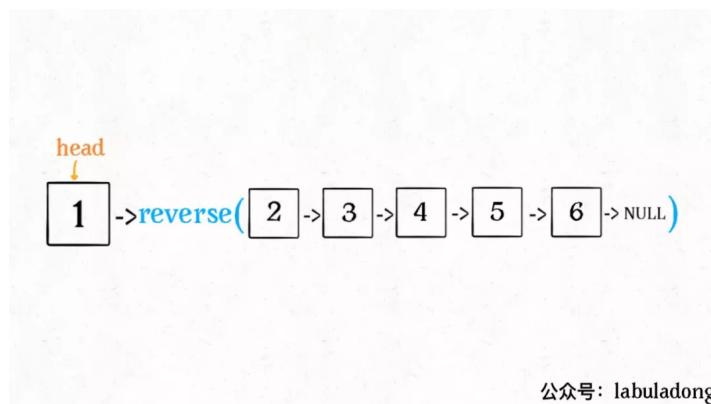
```

## 链表相关 递归思维

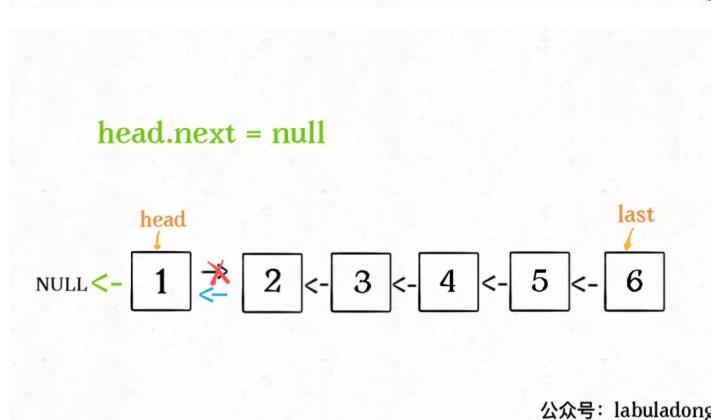
1. <https://leetcode-cn.com/problems/reverse-linked-list-ii/>

给你单链表的头指针 head 和两个整数 left 和 right，其中  $left \leq right$ 。请你反转从位置  $left$  到位置  $right$  的链表节点，返回反转后的链表。

反转整个链表



公众号: labuladong



公众号: labuladong

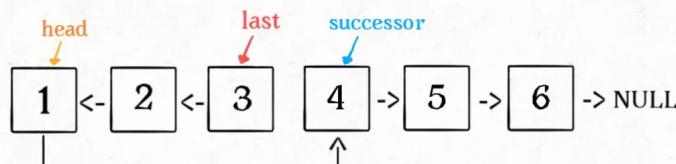
```

ListNode reverse(ListNode head) {
    if (head.next == null) return head;
    ListNode last = reverse(head.next);
    head.next.next = head;
    head.next = null;
    return last;
}

```

先实现反转链表的前n个节点：

- base case 变为 `n == 1`，反转一个元素，就是它本身，同时要记录后驱节点。
- 反转整个链表用递归实现时，直接把 `head.next` 设置为 `null`，因为整个链表反转后原来的 `head` 变成了整个链表的最后一个节点。但现在 `head` 节点在递归反转之后不一定是最后一个节点了，所以要记录后驱 `successor`（第  $n + 1$  个节点），反转之后将 `head` 连接上。



公众号：labuladong

```

ListNode* successor = NULL; // 后驱节点
ListNode* reverse(ListNode* head, int n){
    if(n==1){
        successor=head->next;
        return head;
    }
    auto last = reverse(head->next,n-1);
    head->next->next=head;
    head->next=successor;
    return last;
}

```

实现反转链表的一部分：

base case：如果 `m == 1`，就相当于反转链表开头的 `n` 个元素嘛，也就是我们刚才实现的功能：

```

ListNode reverseBetween(ListNode head, int m, int n) {
    // base case
    if (m == 1) {
        // 相当于反转前 n 个元素
        return reverseN(head, n);
    }
    // ...
}

```

`m != 1` 时，如果我们把 `head` 的索引视为 1，那么我们是想从第 `m` 个元素开始反转；如果把 `head.next` 的索引视为 1，那么相对于 `head.next`，反转的区间应该是从第 `m - 1` 个元素开始的；

```

class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if(left==1){
            return reverse(head, right); //这里是返回! 第一次写错了
        }
        head->next=reverseBetween(head->next, left-1, right-1);
        return head;
    }

    ListNode* successor=NULL; //后驱节点
    ListNode* reverse(ListNode* head, int n){
        if(n==1){
            successor=head->next;
            return head;
        }
        auto last = reverse(head->next, n-1);
        head->next->next=head; //让反转之后的 head 节点和后面的节点连起来
        head->next=successor;
        return last; //每次递归返回的一直是它 原来的尾节点，新的头节点
    }
};

```

II.<https://leetcode-cn.com/problems/reverse-nodes-in-k-group/>

### K个一组反转链表

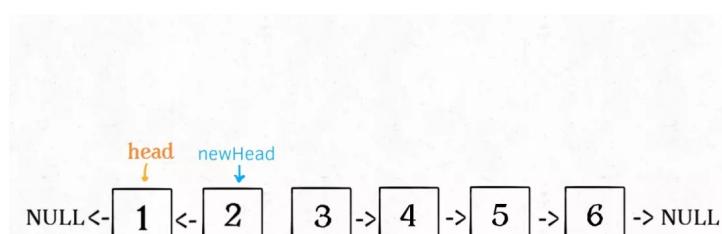
给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

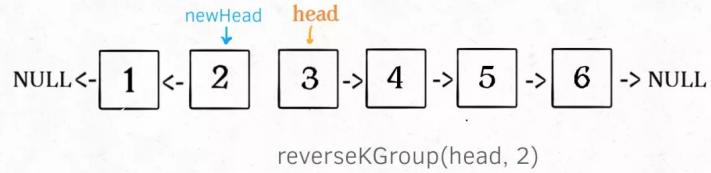
大致的算法流程：

1、先反转以 `head` 开头的 `k` 个元素。



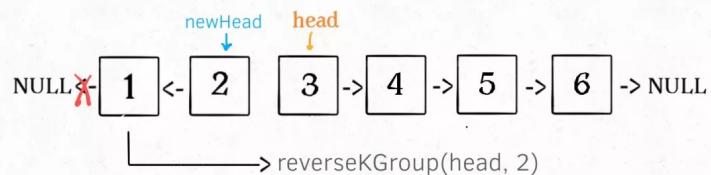
公众号：labuladong

2、将第 `k + 1` 个元素作为 `head` 递归调用 `reverseKGroup` 函数。



公众号: labuladong

### 3、将上述两个过程的结果连接起来。

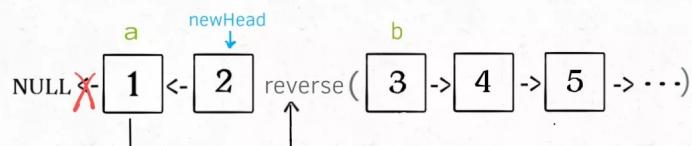


公众号: labuladong

整体思路就是这样了，最后一点值得注意的是，递归函数都有个 base case，对于这个问题是什么呢？

题目说了，如果最后的元素不足  $k$  个，就保持不变。这就是 base case

注意 `reverse` 函数是反转区间  $[a, b)$ ，所以情形是这样的：



公众号: labuladong

```
//反转区间 [a, b) 的元素，注意是左闭右开 反转后 a是尾节点
ListNode* reverse(ListNode* a, ListNode* b){
    ListNode* pre=NULL, *cur=a, *nxt=a;
    while(cur!=b){
        nxt=cur->next;
        cur->next=pre;
        pre=cur;
        cur=nxt;
    }
    return pre;// 返回反转后的头结点
}

ListNode* reverseKGroup(ListNode* head, int k) {
    if(head==NULL) return NULL;
    ListNode* a=head, *b=head;
```

```

for(i=0;i<k;i++){
    if(b==NULL) return head; //凑不够k个 无法反转 返回链表头
    b=b->next;
}
ListNode* newHead = reverse(a,b);
a->next=reverseKGroup(b,k);
return newHead;
}

```

III.<https://leetcode-cn.com/problems/palindrome-linked-list/>

回文链表

### 方法一 递归

如果我想正序打印链表中的 `val` 值，可以在前序遍历位置写代码；反之，如果想倒序遍历链表，就可以在后序遍历位置操作。

实际上就是把链表节点放入一个栈，然后再拿出来，这时候元素顺序就是反的

```

class Solution {// 递归
public:
    ListNode* first;
    bool isPalindrome(ListNode* head) {
        first = head;
        return Pdcheck(head);
    }

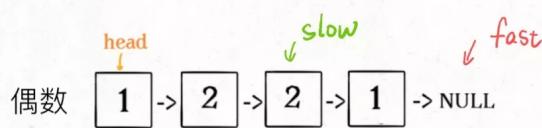
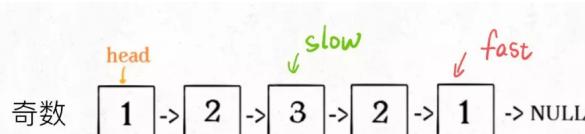
    bool Pdcheck(ListNode* head){
        if(head==NULL) return true;
        bool r = Pdcheck(head->next); //后序遍历
        r = r&& (first->val==head->val);
        first=first->next;
        return r;
    }
};

```

### 方法二 快慢指针

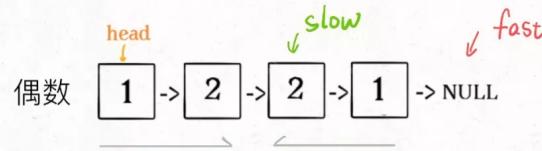
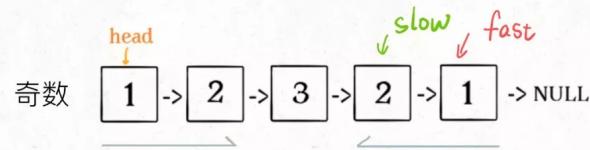
1、先通过 [双指针技巧汇总](#) 中的快慢指针来找到链表的中点：

`slow` 指针现在指向链表中点



公众号: labuladong

2、如果 `fast` 指针没有指向 `null`，说明链表长度为奇数，`slow` 还要再前进一步：



公众号: labuladong

3、从 slow 开始反转后面的链表，现在就可以开始比较回文串了：

```

class Solution {
public:

    bool isPalindrome(ListNode* head) {
        if(head->next==NULL) return true; // 只有一个数的傻逼样例
        ListNode* slow=head;
        ListNode* fast=head;
        while(fast!=NULL&&fast->next!=NULL){ // 快慢指针
            slow=slow->next;
            fast=fast->next->next;
        }
        if(fast!=NULL) slow=slow->next; // 链表长度为奇数，`slow`还要再前进一步

        auto one=head;
        auto two=reverse(slow);

        while(two!=NULL){ // 同时遍历 对比长度为后面链表的长度（最初给的链表为奇数，中间的那个值为多少都无所谓）
            if(one->val==two->val){
                one=one->next;
                two=two->next;
            }else{
                return false;
            }
        } // 成功遍历完后面的列表未返回false 说明成功
        return true; // 在这返回
    }

    // 反转链表 刻在脑子里 返回新链表的头 原来的头a变成的尾
    ListNode* reverse(ListNode*a){
        if(a==NULL) return NULL;
        ListNode* pre = NULL,* cur =a,* nxt=a;
        while(cur!=NULL){
            nxt=cur->next;
            cur->next=pre;
            pre=cur;
            cur=nxt;
        }
        return pre;
    }
};

```

## 二叉树二

I.<https://leetcode-cn.com/problems/maximum-binary-tree/>

最大二叉树

给定一个不含重复元素的整数数组 nums。一个以此数组直接递归构建的最大二叉树 定义如下：

二叉树的根是数组 nums 中的**最大元素**。

左子树是通过数组中 最大值左边部分 递归构造出的最大二叉树。

右子树是通过数组中 最大值右边部分 递归构造出的最大二叉树。

返回有给定数组 nums 构建的最大二叉树。

```
class Solution {
public:
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        return build(0,nums.size()-1,nums);
    }
    TreeNode* build(int l,int r,vector<int>& nums){
        //if(head==NULL) return NULL;
        if(l>r) return NULL;
        int nmax=-1;
        int idx=0;
        for(int i=l;i<=r;i++){//找最大值 做rootval
            if(nums[i]>nmax){
                idx=i;
                nmax=nums[i];
            }
        }
        TreeNode* root=new TreeNode(nmax);
        root->left=build(l, idx-1, nums);
        root->right=build(idx+1, r, nums);
        return root;
    }
};
```

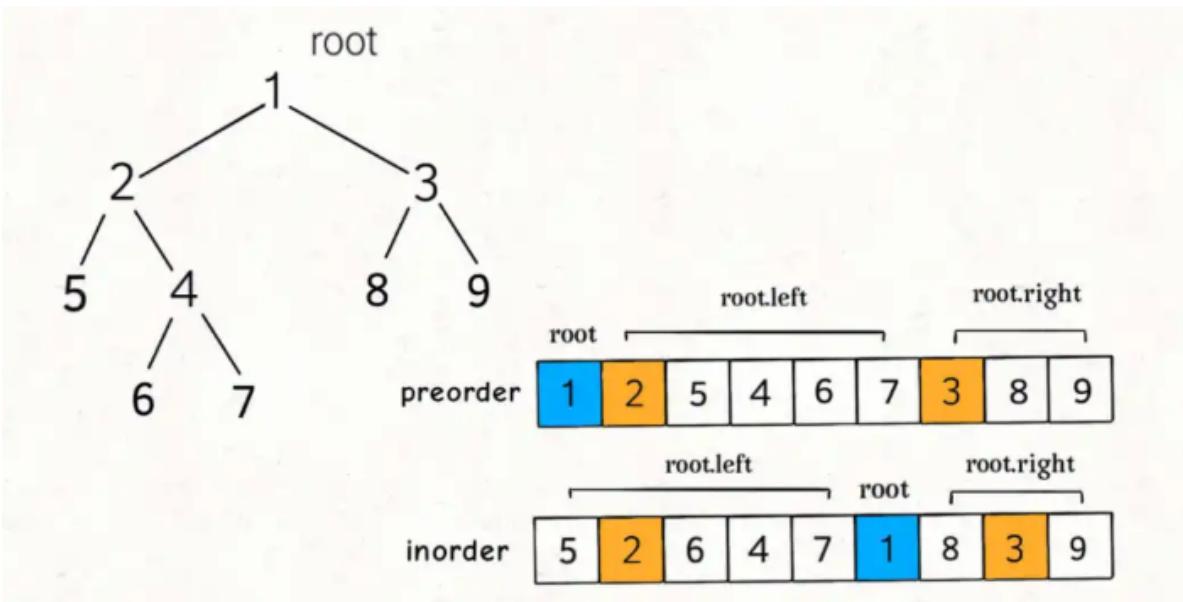
II.<https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>

从前序与中序构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

**注意:**

你可以假设树中没有重复的元素。



```

class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        return build(preorder, inorder, 0, preorder.size() - 1, 0, inorder.size() - 1);
    }

    TreeNode* build(vector<int>& preorder, vector<int>& inorder, int pstart, int pend, int istart, int iend) {
        if(pstart > pend) return NULL; // base case
        // 找到 rootval 在中序遍历数组中的索引
        int rootidx = 0;
        for(int i = istart; i <= iend; i++) {
            if(inorder[i] == preorder[pstart]) {
                rootidx = i;
                break;
            }
        }
        // 先构造出当前根节点
        TreeNode* root = new TreeNode(preorder[pstart]); // 必须放在堆里...
        // 递归构造左右子树
        root->left = build(preorder, inorder, pstart + 1, pstart + rootidx - istart, istart, rootidx - 1);
        root->right = build(preorder, inorder, pstart + rootidx - istart + 1, pend, rootidx + 1, iend);
        return root;
    }
};

```

III. <https://leetcode-cn.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>

从中序与后序构造二叉树

根据一棵树的中序遍历与后序遍历构造二叉树。

**注意:**

你可以假设树中没有重复的元素。

基本同上

```

class Solution {
public:

```

```

TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
    return build(inorder, postorder, 0, inorder.size()-1, 0,
postorder.size()-1);
}
TreeNode* build(vector<int>& inorder, vector<int>& postorder,int istart,int
iend,int pstart,int pend){
    if(istart>iend) return NULL;

    int rootidx=0;
    for(int i=istart;i<=iend;i++){
        if(inorder[i]==postorder[pend]){
            rootidx=i;
            break;
        }
    }

    TreeNode* root = new TreeNode(postorder[pend]);
    root->left= build(inorder, postorder, istart, rootidx-1, pstart, pstart-
(istart-rootidx)-1);
    root->right=build(inorder, postorder, rootidx+1, iend, pstart-(istart-
rootidx), pend-1);
    return root;
}
};


```

## 二叉树三

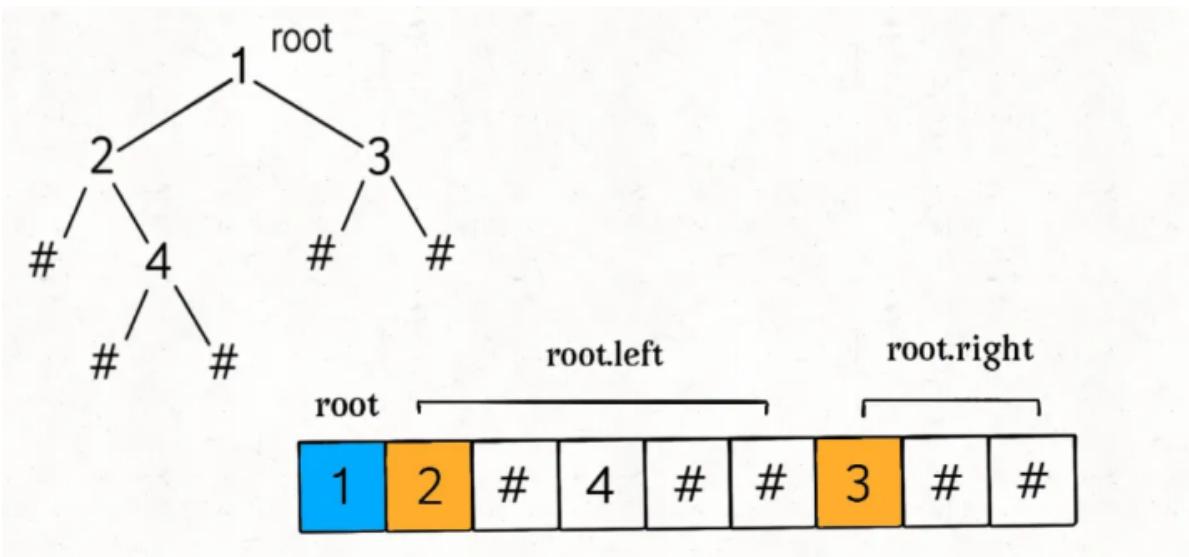
1.<https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/>

二叉树的序列化与反序列化

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

我们可以用 `serialize` 方法将二叉树序列化成字符串，用 `deserialize` 方法将序列化的字符串反序列化成二叉树

**所谓的序列化不过就是把结构化的数据「打平」，其实就是在考察二叉树的遍历方式。**



用 `,` 作为分隔符，用 `#` 表示空指针 `null`，调用完 `serializehelp` 函数以后 `s` 变为：

`"1,2,#,4,#,#,3,#,#,"`

PS：一般语境下，单单前序遍历结果是不能还原二叉树结构的，因为缺少空指针的信息，至少要得到前、中、后序遍历中的两种才能还原二叉树。但是这里的 `node` 列表包含空指针的信息，所以只使用 `node` 列表就可以还原二叉树。

反序列化过程也是一样，先确定根节点 `root`，然后遵循前序遍历的规则，递归生成左右子树即可

由于c++没有split函数 需要自己利用队列实现。

```

class Codec {
public:

    // Encodes a tree to a single string.序列化
    string serialize(TreeNode* root) {
        string s;
        serializehelp(root, s);
        s.pop_back(); // 去掉尾部 ,
        return s;
    }

    string dh=",";
    string jh="#";
    void serializehelp(TreeNode* root, string& s){
        if(root == NULL){
            s.append(jh).append(dh);
            return;
        }
        s.append(to_string(root->val)).append(dh);
        serializehelp(root->left, s);
        serializehelp(root->right, s);
    }

    // Decodes your encoded data to tree.反序列化
    TreeNode* deserialize(string data) {
        queue<string> q=split(data);
        return deserializehelp(q);
    }
}

```

```

queue<string> split(string s){//把string字符串拆分放进队列
    queue<string> q;
    int idx=0;
    for(int i =0;i<=s.size();i++){
        if(i==s.size()||s[i]==','){
            q.push(s.substr(idx,i-idx));
            idx=i+1;
        }
    }
    return q;
}

TreeNode* deserializehelp(queue<string> &q){
    if(q.size()==0) return NULL;//之前没加这一句导致溢出了
    string first = q.front();
    q.pop(); //要先pop再判断是否=="#"，否则"#"在pop前就return
    if(first=="#") return NULL;

    TreeNode* root=new TreeNode(stoi(first));
    root->left=deserializehelp(q);
    root->right=deserializehelp(q);
    return root;
}

};


```

II.<https://leetcode-cn.com/problems/find-duplicate-subtrees/>

### 寻找重复子树

给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。

两棵树重复是指它们具有相同的结构以及相同的结点值。

**你需要知道以下两点：**

- 1、以我为根的这棵二叉树（子树）长啥样？
- 2、以其他节点为根的子树都长啥样？

- 1、二叉树的前序/中序/后序遍历结果可以描述二叉树的结构。所以，我们可以通过拼接字符串的方式把二叉树序列化
- 2、借助一个外部数据结构，让每个节点把自己子树的序列化结果存进去，这样对于每个节点，就可以知道有没有其他节点的子树和自己重复了

```

class Solution {
public:
    unordered_map<string, int> m;//备忘录 序列出现的次数
    vector<TreeNode*> res;

    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        string ss=post(root);

```

```

        return res;
    }
    string post(TreeNode* root){
        if(root==NULL) return "#";
        string s;

        string l=post(root->left);
        string r=post(root->right);
        s.append(l).append(",").append(r).append(",")
.append(to_string(root->val)); //序列化 前序/中序/后序遍历结果都可以

        m[s]++;
        if(m[s]==2) res.push_back(root); //出现两次就输出
        return s;
    }
};


```

## 二叉搜索树 -

I. <https://leetcode-cn.com/problems/kth-smallest-element-in-a-bst/>

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

二叉搜索树 (Binary Search Tree, 后文简写 BST)

BST 的中序遍历结果是有序的 (升序)。

```

class Solution {
public:
    vector<int> r;
    int kthSmallest(TreeNode* root, int k) {
        post(root);
        return r[k-1];
    }
    void post(TreeNode* root){
        if(root==NULL) return ;
        post(root->left);
        r.push_back(root->val);
        post(root->right);
        return;
    }
};

```

II. <https://leetcode-cn.com/problems/convert-bst-to-greater-tree/>

给出二叉搜索树的根节点，该树的节点值各不相同，请你将其转换为累加树 (Greater Sum Tree)，使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

BST 的中序遍历代码可以升序打印节点的值。那如果我想降序打印节点的值只要把递归顺序改一下就行了

维护一个外部累加变量 `sum`，然后把 `sum` 赋值给 BST 中的每一个节点，就将 BST 转化成累加树了

```

class Solution {
public:
    TreeNode* convertBST(TreeNode* root) {
        in(root);
        return root;
    }
    int sum=0;
    void in(TreeNode* root){//反向中序遍历 r->m->l
        if(root==NULL) return;
        in(root->right);
        sum+=root->val;
        root->val=sum;
        in(root->left);
        return;
    }
};

```

## 二叉搜索树 二

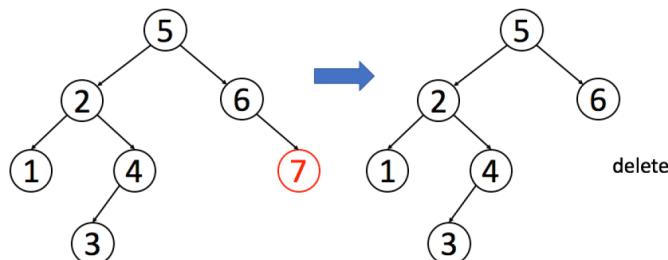
I.<https://leetcode-cn.com/problems/delete-node-in-a-bst/>

删除二叉搜索树中的节点

找到目标节点了，比方说是节点 A，如何删除这个节点，这是难点。因为删除节点的同时不能破坏 BST 的性质。有三种情况，用图片来说明。

**情况 1：** A 恰好是末端节点，两个子节点都为空，那么它可以当场去世了。

Case 1: No Child



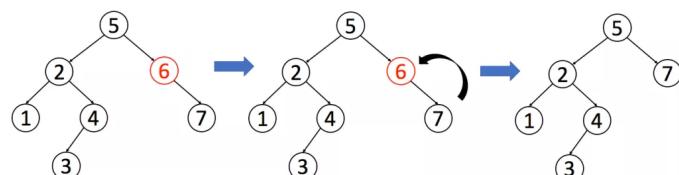
```

if (root.left == null && root.right == null)
    return null;

```

**情况 2：** A 只有一个非空子节点，那么它要让这个孩子接替自己的位置。

Case 2: One Child



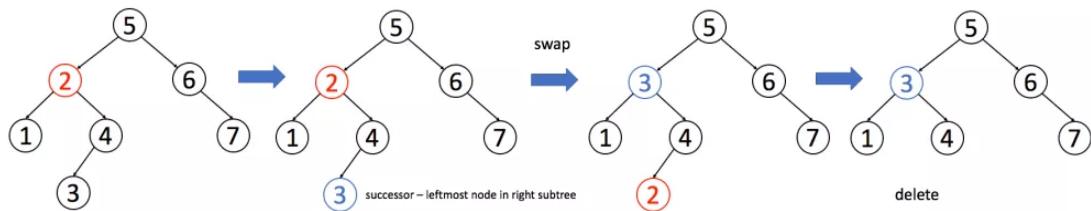
```

// 排除了情况 1 之后
if (root.left == null) return root.right;
if (root.right == null) return root.left;

```

**情况 3:** A 有两个子节点，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点，或者右子树中最小的那个节点来接替自己。我们以第二种方式。

Case 3: Two Children



```
if (root.left != null && root.right != null) {
    // 找到右子树的最小节点
    TreeNode minNode = getMin(root.right);
    // 把 root 改成 minNode
    root.val = minNode.val;
    // 转而去删除 minNode
    root.right = deleteNode(root.right, minNode.val);
}
```

代码如下：

```
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) { // 返回的是删除了某个 .val==key 的子节点的root树
        if (root==NULL) return NULL;
        if (root->val==key){
            //这两个 if 把情况 1 和 2 都正确处理了
            if (root->left==NULL) return root->right;
            if (root->right==NULL) return root->left;

            auto minval=getMin(root->right); // 找到右节点树的最小值!
            root->val=minval->val; // 赋值给自己!
            root->right=deleteNode(root->right, minval->val); //再去右节点树中删掉这个节点!
        }
        else if (root->val>key){
            root->left=deleteNode(root->left, key);
        }
        else if (root->val<key){
            root->right=deleteNode(root->right, key);
        }
        return root;
    }

    TreeNode* getMin(TreeNode* root){ // 找到root树最小值 那自然是最左边的
        while (root->left!=NULL) root=root->left;
        return root;
    }
};
```

II.<https://leetcode-cn.com/problems/insert-into-a-binary-search-tree/>

## 二叉搜索树中的插入

简单，小左大右 直到找到nullptr的位置 创建一个root.val为题目val的节点放上去就行。

```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(root==NULL){
            TreeNode* root=new TreeNode(val);
            return root;
        }

        if(root->val>val){
            root->left=insertIntoBST(root->left,val);
        }else if(root->val<val){
            root->right=insertIntoBST(root->right,val);
        }

        return root;
    }
};
```

III.<https://leetcode-cn.com/problems/search-in-a-binary-search-tree/>

## 二叉搜索树中的搜索

```
TreeNode* searchBST(TreeNode* root, int val) {
    if(root==NULL) return NULL;
    if(root->val==val) return root;
    if(root->val>val){
        root=searchBST(root->left,val);
    }else if(root->val<val){
        root=searchBST(root->right,val);
    }
    return root;
}
```

IV.<https://leetcode-cn.com/problems/validate-binary-search-tree/>

## 验证二叉搜索树

方法一 利用中序遍历的严格递增特性

```
class Solution {
public:
    vector<int> res;
    bool isValidBST(TreeNode* root) {
        in(root);
        for(int i=1;i<res.size();i++){
            if(res[i]<=res[i-1])return false;
        }
        return true;
    }

    void in(TreeNode* root){
        if(root==NULL) return;
        in(root->left);
        res.push_back(root->val);
        in(root->right);
    }
};
```

```

        res.push_back(root->val);
        in(root->right);
        return;
    }
};

```

## 方法二 递归

如果该二叉树的左子树不为空，则左子树上所有节点的值均小于它的根节点的值；若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；它的左右子树也为二叉搜索树。

设计一个递归函数 helper(root, lower, upper) 来递归判断，函数表示考虑以 root 为根的子树，判断子树中所有节点的值是否都在 (l,r) 的范围内（注意是开区间）。如果 root 节点的值 val 不在 (l,r) 的范围内说明不满足条件直接返回，否则我们要继续递归调用检查它的左右子树是否满足，如果都满足才说明这是一棵二叉搜索树。

根据二叉搜索树的性质，在递归调用左子树时，我们需要把上界 upper 改为 root.val，即调用 helper(root.left, lower, root.val)，因为左子树里所有节点的值均小于它的根节点的值。同理递归调用右子树时，我们需要把下界 lower 改为 root.val，即调用 helper(root.right, root.val, upper)。

```

class Solution {
public:
    bool helper(TreeNode* root, long long lower, long long upper) {
        //用long long的原因是样例犯病输入了INT_MAX
        if (root == nullptr) {
            return true;
        }
        if (root -> val <= lower || root -> val >= upper) {
            return false;
        }
        return helper(root -> left, lower, root -> val) && helper(root -> right,
root -> val, upper);
    }
    bool isValidBST(TreeNode* root) {
        return helper(root, LONG_MIN, LONG_MAX);
    }
};

```

## 扁平化嵌套列表迭代器

<https://leetcode-cn.com/problems/flatten-nested-list-iterator/>

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

### 方法一 递归: dfs

这是最简单的方法，但是我认为这不是面试官想要的方法。

在构造函数中提前扁平化整个嵌套列表。那么在 hasNext() 或者 next() 可以很简单地返回迭代器位置的 int。

因为这个嵌套的数据结构有点类似于**多叉树**，所以我们可以按照类似地遍历思路：**递归**。

**承载遍历结果的数据结构可以使用数组**，那么另外需要一个整数标记当前的迭代器指向的位置；也可以使用一个队列，每次调用 next() 方法的时候从队列的开头弹出一个元素。

数组实现：

```
class NestedIterator {
private:
    vector<int> res; //承载遍历结果
    vector<int>::iterator cur; //标记当前的迭代器指向的位置
    //利用dfs提前扁平化迭代器
    void dfs(vector<NestedInteger> &nestedList){
        for(auto n:nestedList){
            if(n.isInteger()){
                res.push_back(n.getInteger());
            }else{
                dfs(n.getList());
            }
        }
        return;
    }
public:

    NestedIterator(vector<NestedInteger> &nestedList) {
        dfs(nestedList); //扁平化
        cur=res.begin();
    }

    int next() {
        return *cur++; //注意是*cur
    }

    bool hasNext() {
        if(cur==res.end())return false; //注意是cur
        return true;
    }
};
```

## 方法二 迭代：利用栈的方法 满足迭代器惰性

在递归方法中，我们在遍历时如果遇到一个嵌套的子list，就立即处理该子list，直到全部展开；在迭代方法中，我们不需要全部展开，只需要把当前list的所有元素放入list中。

由于「栈」的先进后出的特性，我们需要逆序在栈里放入各个元素。

算法整体的流程，通过举例说明。假如输入 [1, [2,3]]。

1. 在构造函数中：栈里面放的应该是 stack = [[2, 3], 1]
2. 在调用 hasNext() 方法时，访问栈顶元素是 1，为 int，那么直接返回 true；
3. 然后调用 next() 方法，弹出栈顶元素 1；
4. 再调用 hasNext() 方法时，访问栈顶元素是 [2,3]，为 list，那么需要摊平，继续放到栈中。当前的栈是 stack = [3, 2]；
5. 然后调用 next() 方法，弹出栈顶元素 2；
6. 然后调用 next() 方法，弹出栈顶元素 3；
7. 再调用 hasNext() 方法时，栈为空，因此返回 false，迭代器运行结束。

为什么在 hasNext() 方法中摊平 list，而不是在 next() 方法中。比如对于 [[]] 的输入，hasNext() 方法是判断其中是否有 int 元素了，则必须把内层的 list 摊平来看，发现是空的，返回 false。

```
class NestedIterator {
private:
```

```

stack<NestedInteger> st;//承载遍历结果

public:
    NestedIterator(vector<NestedInteger> &nestedList) {
        for(int i=nestedList.size()-1;i>=0;i--){//逆序遍历放入栈中 (不摊平)
            st.push(nestedList[i]);
        }
    }

    int next() { //得到当前栈顶的值 也就是迭代器最前面的值
        auto cur=st.top();
        st.pop();
        return cur.getInteger();
    }

    bool hasNext() {
        while(!st.empty()){
            auto cur=st.top();
            if(cur.isInteger()==1){//如果是栈顶是整数 直接返回true
                return true;
            }
            st.pop(); //如果是栈顶是list 先pop然后摊平 逆序 放进来
            for(int i =cur.getList().size()-1;i>=0;i--){
                st.push(cur.getList()[i]);
            }
        }
        return false; //如果栈为空 返回false
    }

};


```

## 二叉树的最近公共节点

<https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/>

灵魂三问：

- 1、这个函数是干嘛的？
- 2、这个函数参数中的变量是什么的是什么？
- 3、得到函数的递归结果，你应该干什么？

首先看第一个问题，这个函数是干嘛的？或者说，你给我描述一下 `lowestCommonAncestor` 这个函数的「定义」吧。

描述：给该函数输入三个参数 `root`, `p`, `q`, 它会返回一个节点。

情况 1，如果 `p` 和 `q` 都在以 `root` 为根的树中，函数返回的即使 `p` 和 `q` 的最近公共祖先节点。

情况 2，那如果 `p` 和 `q` 都不在以 `root` 为根的树中怎么办呢？函数理所当然地返回 `null` 呀。

情况 3，那如果 `p` 和 `q` 只有一个存在于 `root` 为根的树中呢？函数就会返回那个节点。

第二个问题，这个函数的参数中，变量是什么？或者说，你描述一个这个函数的「状态」吧。

描述：函数参数中的变量是 `root`，因为根据框架，`lowestCommonAncestor(root)` 会递归调用 `root.left` 和 `root.right`；至于 `p` 和 `q`，我们要求它俩的公共祖先，它俩肯定不会变化的。

**第三个问题，得到函数的递归结果，你该干嘛？**或者说，得到递归调用的结果后，你做什么「选择」？

先想 base case，如果 `root` 为空，肯定得返回 `null`。如果 `root` 本身就是 `p` 或者 `q`，比如说 `root` 就是 `p` 节点吧，如果 `q` 存在于以 `root` 为根的树中，显然 `root` 就是最近公共祖先；即使 `q` 不存在于以 `root` 为根的树中，按照情况 3 的定义，也应该返回 `root` 节点。

**做选择：**

用递归调用的结果 `left` 和 `right` 来搞点事情。分情况讨论：

情况 1，如果 `p` 和 `q` 都在以 `root` 为根的树中，那么 `left` 和 `right` 一定分别是 `p` 和 `q`（从 base case 看出来的）。

情况 2，如果 `p` 和 `q` 都不在以 `root` 为根的树中，直接返回 `null`。

情况 3，如果 `p` 和 `q` 只有一个存在于 `root` 为根的树中，函数返回该节点。

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==nullptr) return nullptr;
        if(root==p||root==q) return root;
        auto l = lowestCommonAncestor(root->left,p,q);
        auto r = lowestCommonAncestor(root->right,p,q);
        if(l!=NULL&&r!=NULL){
            return root;
        }else if(l==NULL&&r==NULL){
            return NULL;
        }
        return l==nullptr?r:l;
    }
};
```

## 求完全二叉树的节点数

如何求一棵完全二叉树的节点个数呢？

```
// 输入一棵完全二叉树，返回节点总数
int countNodes(TreeNode root);
```

如果是一个**普通**二叉树，显然只要向下面这样遍历一边即可，时间复杂度  $O(N)$ ：

```
public int countNodes(TreeNode root) {
    if (root == null) return 0;
    return 1 + countNodes(root.left) + countNodes(root.right);
}
```

那如果是一棵**满**二叉树，节点总数就和树的高度呈指数关系，时间复杂度  $O(\log N)$ ：

```

public int countNodes(TreeNode root) {
    int h = 0;
    // 计算树的高度
    while (root != null) {
        root = root.left;
        h++;
    }
    // 节点总数就是  $2^h - 1$ 
    return (int) Math.pow(2, h) - 1;
}

```

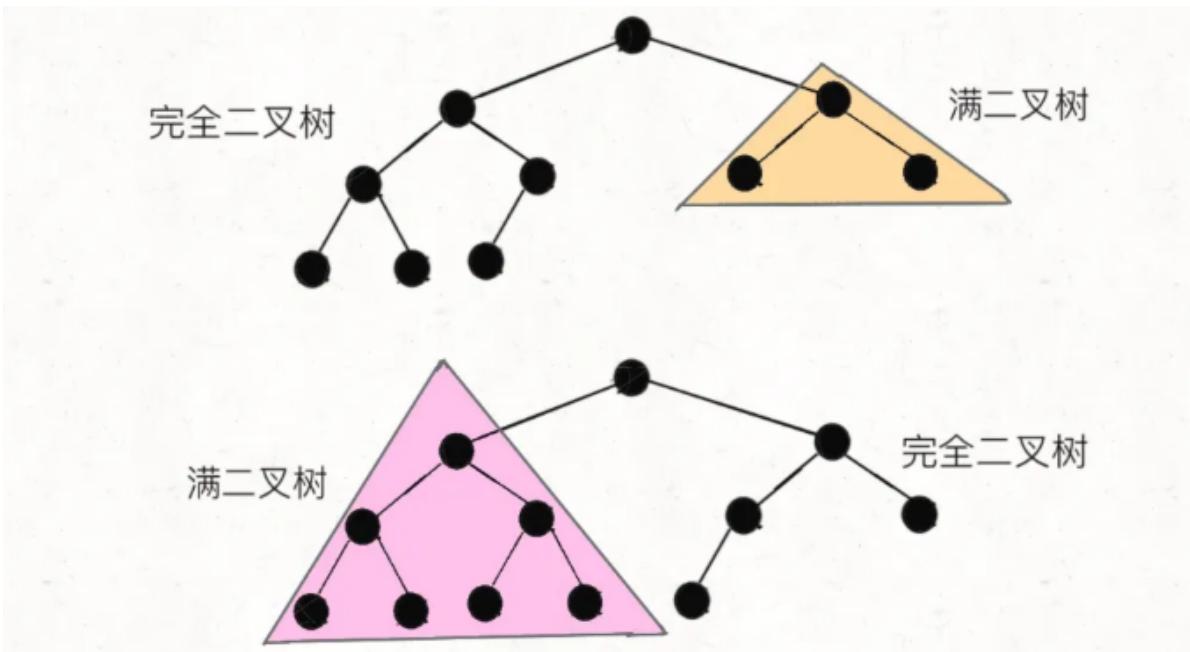
**完全二叉树**比普通二叉树特殊，但又没有满二叉树那么特殊，计算它的节点总数，可以说是普通二叉树和完全二叉树的结合版，先看代码：

```

class Solution {
public:
    int countNodes(TreeNode* root) {
        if(root==NULL) return 0;
        auto l=root->left, r=root->right;
        int lh=1, rh=1;      // 记录左、右子树的高度
        while(l!=NULL){
            l=l->left;
            lh++;
        }
        while(r!=NULL){
            r=r->right;
            rh++;
        }
        // 如果左右子树的高度相同，则是一棵满二叉树
        if(lh==rh){
            return pow(2, lh)-1;
        }
        // 如果左右高度不同，则按照普通二叉树的逻辑计算
        return 1+countNodes(root->left)+countNodes(root->right);
    }
};

```

**关键点在于，这两个递归只有一个会真的递归下去，另一个一定会触发 `h1 == hr` 而立即返回，不会递归下去。**因为一棵完全二叉树的两棵子树，至少有一棵是满二叉树：



## 数据流中位数

<https://leetcode-cn.com/problems/find-median-from-data-stream/>

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是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  $(2 + 3) / 2 = 2.5$

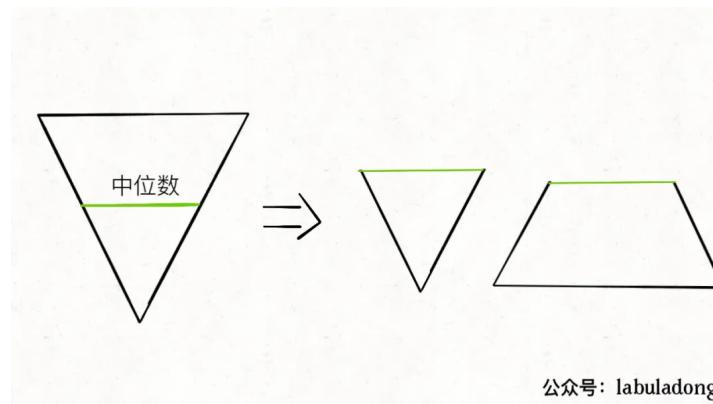
设计一个支持以下两种操作的数据结构：

`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。

**我们必然需要有序数据结构，本题的核心思路是使用两个优先级队列。**

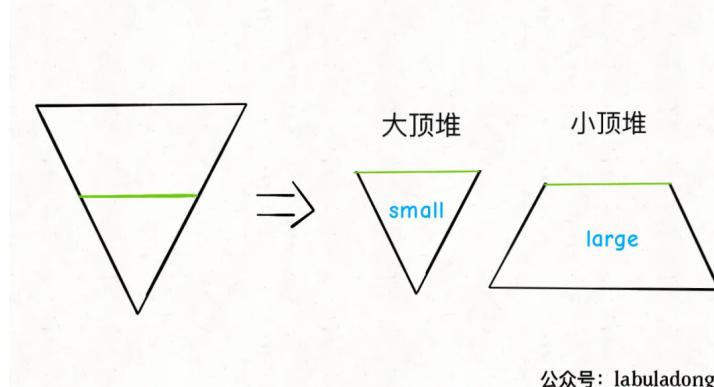
中位数是有序数组最中间的元素算出来的对吧，我们可以把「有序数组」抽象成一个倒三角形，宽度可以视为元素的大小，那么这个倒三角的中部就是计算中位数的元素对吧：



公众号: labuladong

然后我把这个大的倒三角形从正中间切成两半，变成一个小倒三角和一个梯形，这个小倒三角相当于一个从小到大的有序数组，这个梯形相当于一个从大到小的有序数组。

**小倒三角不就是个大顶堆嘛，梯形不就是个小顶堆嘛，中位数可以通过它们的堆顶元素算出来。**



公众号: labuladong

梯形虽然是小顶堆，但其中的元素是较大的，我们称其为 `large`，倒三角虽然是大顶堆，但是其中元素较小，我们称其为 `small`。

当然，这两个堆需要算法逻辑正确维护，才能保证堆顶元素是可以算出正确的中位数，**我们很容易看出来，两个堆中的元素之差不能超过 1。并且不仅要维护 `large` 和 `small` 的元素个数之差不超过 1，还要维护 `large` 堆的堆顶元素要大于等于 `small` 堆的堆顶元素。**

简单说，想要往 `large` 里添加元素，不能直接添加，而是要先往 `small` 里添加，然后再把 `small` 的堆顶元素加到 `large` 中；向 `small` 中添加元素同理。

我们设定要不然两堆大小相等，要不然大顶堆多一个。

优先队列会自动维护有序性。

```

class MedianFinder {
public:
    /** initialize your data structure here. */
    priority_queue<int , vector<int> , greater<int> > small; //小顶
    priority_queue<int,vector<int>,less<int> > big; //大顶

    MedianFinder() {}

    void addNum(int num) {
        if(small.size()==big.size()){//大小相等最终往大顶放
            small.push(num); //num先放小顶
            big.push(small.top());
            small.pop();
        }else{
            big.push(num); //大小不等最终往小顶放
            small.push(big.top()); //num先放大顶
            big.pop();
        }
    }

    double findMedian() {
        if(small.size()==big.size())return static_cast<double>
        (small.top()+big.top())/2; //注意这里要返回double
        return big.top();
    }
};

```

# 被围绕的区域

<https://leetcode-cn.com/problems/surrounded-regions/>

给你一个  $m \times n$  的矩阵 board，由若干字符 'X' 和 'O'，找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例 1：

X	X	X	X
X	O	O	X
X	X	O	X
X	O	X	X

→

X	X	X	X
X	X	X	X
X	X	X	X
X	O	X	X

给定的矩阵中有三种元素：

- 字母 X；
- 被字母 X 包围的字母 O；
- 没有被字母 X 包围的字母 O。

任何边界上的 O 都不会被填充为 X。所有的不被包围的 O 都直接或间接与边界上的 O 相连。我们可以利用这个性质判断 O 是否在边界上，具体地说：

对于每一个边界的 O，我们以它为起点，标记所有与它直接或间接相连的字母 O；

最后我们遍历这个矩阵，对于每一个字母：

- 如果该字母被标记过，则该字母为没有被字母 X 包围的字母 O，我们将其还原为字母 O；
- 如果该字母没有被标记过，则该字母为被字母 X 包围的字母 O，我们将其修改为字母 X。

## 方法一 dfs

```
class Solution {
public:
    void solve(vector<vector<char>>& board) {
        int m=board.size();
        int n =board[0].size();

        for(int i=0;i<m;i++){ //对第一列和最后一列的‘O’dfs
            dfs(board, i, 0);
            dfs(board, i, n-1);
        }
        for(int i=0;i<n;i++){//对第一行和最后一行的‘O’dfs
            dfs(board, 0,i);
            dfs(board, m-1,i);
        }
        //A->O,O->X
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(board[i][j]=='A')board[i][j]='O';
                else if(board[i][j]=='O')board[i][j]='X';
            }
        }
    }
}
```

```

void dfs(vector<vector<char>>& board, int x, int y){
    if(x<0 || x>(board.size()-1) || y<0 || y>(board[0].size()-1) || board[x][y] != 'o') { // board[x][y] != 'o' 放最后! 防止溢出
        return;
    }
    board[x][y] = 'A';
    dfs(board, x, y+1); // 遍历间接相邻的'o'
    dfs(board, x, y-1);
    dfs(board, x-1, y);
    dfs(board, x+1, y);
}
};

```

## 方法二 bfs

```

class Solution {
public:
    void solve(vector<vector<char>>& board) {
        int dx[4]={1,-1,0,0};
        int dy[4]={0,0,1,-1};
        int m=board.size();
        int n=board[0].size();

        queue<pair<int,int>> q;
        for(int i=0;i<m;i++){
            if(board[i][0]=='o'){
                q.emplace(i,0);
            }
            if(board[i][n-1]=='o'){
                q.emplace(i,n-1);
            }
        }
        for(int i=1;i<n-1;i++){
            if(board[0][i]=='o'){
                q.emplace(0,i);
            }
            if(board[m-1][i]=='o'){
                q.emplace(m-1,i);
            }
        }
        while(!q.empty()){
            int x=q.front().first,y=q.front().second;
            q.pop();
            board[x][y]='A';
            for(int i=0;i<4;i++){
                int mx=x+dx[i],my=y+dy[i];
                if (mx < 0 || my < 0 || mx >= m || my >= n || board[mx][my] != 'o') {
                    continue;
                }
                q.emplace(mx,my);
            }
        }
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {

```

```

        if (board[i][j] == 'A') {
            board[i][j] = 'O';
        } else if (board[i][j] == 'O') {
            board[i][j] = 'X';
        }
    }
}

};

}

```

### 方法三 并查集

主要思路是适时增加虚拟节点，想办法让元素「分门别类」，建立动态连通关系。

```

class Solution {
public:
    int dir[4][2] = {{1,0},{0,1}, {-1,0}, {0,-1}};
    void solve(vector<vector<char>>& board) {
        if(board.size() == 0) return;
        int m = board.size();
        int n = board[0].size();
        UnionFind uf = UnionFind(m * n + 1); //m * n + 1个节点，其中+1 放ancetor
        int ancetor = m * n;
        //连接首行末行元素 到ancetor 这样外圈的‘O’就同根了
        for(int i = 0; i < n; i++){
            if(board[0][i] == 'O') uf.unite(i, ancetor);
            if(board[m-1][i] == 'O') uf.unite(i+(m-1)*n, ancetor);
        }
        //连接首列末列元素 到ancetor
        for(int i = 0; i < m; i++){
            if(board[i][0] == 'O') uf.unite(i*n, ancetor);
            if(board[i][n-1] == 'O') uf.unite(i*n+(n-1), ancetor);
        }
        //连接所有O
        for(int i = 1; i < m-1; i++){//注意范围
            for(int j = 1; j < n-1; j++){
                if(board[i][j] == 'O'){
                    for(int k = 0; k < 4; k++){ //内圈O与其四周相连
                        int x = i + dir[k][0];
                        int y = j + dir[k][1];
                        if (board[x][y] == 'O')
                            uf.unite(x * n + y, i * n + j); // (i,j)链接(x,y)
                    }
                }
            }
        }
        //没有和ancetor连接的都要替换
        for(int i = 1; i < m-1; i++){
            for(int j = 1; j < n-1; j++){
                if(!uf.isconnected(i * n + j, ancetor))
                    board[i][j] = 'X';
            }
        }
    };
};

//以下为并查集实现

```

```
class UnionFind{
private:
    int count;
    int* size;
    int* parent;
public:
    UnionFind(int n){
        this->count = n;
        this->parent = new int[n];
        this->size = new int[n];
        for(int i = 0; i < n; i++){
            size[i] = 1;
            parent[i] = i;
        }
    }
    ~UnionFind(){
        delete []size;
        delete []parent;
    }
    void unite(int a, int b){
        int ancestor_a = find(a);
        int ancestor_b = find(b);
        if(ancestor_a == ancestor_b) return;
        //平衡结点
        if(size[ancestor_a] > size[ancestor_b]){
            parent[ancestor_b] = ancestor_a;
            size[ancestor_a] += size[ancestor_b];
        } else {
            parent[ancestor_a] = ancestor_b;
            size[ancestor_b] += size[ancestor_a];
        }
        count--;
    }
    int find(int a){
        while(parent[a] != a){
            //路径压缩
            parent[a] = parent[parent[a]];
            a = parent[a];
        }
        return a;
    }
    bool isconnected(int a, int b){
        int ancestor_a = find(a);
        int ancestor_b = find(b);
        if(ancestor_a == ancestor_b) return true;
        return false;
    }
    int _count(){
        return count;
    }
};
```

## 并查集 背！ ☆

二维坐标  $(x, y)$  可以转换成  $x * m + y$  这个数 ( $m$  是棋盘的行数,  $n$  是棋盘的列数)。敲黑板, **这是将二维坐标映射到一维的常用技巧。**

```
class UnionFind{
private:
    int count; //连通单元数
    int* size; //以某个节点为根的节点数量 包括自己
    int* parent; //根节点
public:
    UnionFind(int n){
        this->count = n;
        this->parent = new int[n];
        this->size = new int[n];
        for(int i = 0; i < n; i++){
            size[i] = 1;
            parent[i] = i;
        }
    }
    ~UnionFind(){
        delete []size;
        delete []parent;
    }
    //函数: 连接unite, 找根节点find, 判断连接isconnected, 连通单元数count
    void unite(int a, int b){
        int ancestor_a = find(a);
        int ancestor_b = find(b);
        if(ancestor_a == ancestor_b) return;
        //平衡结点
        if(size[ancestor_a] > size[ancestor_b]){
            parent[ancestor_b] = ancestor_a;
            size[ancestor_a] += size[ancestor_b];
        } else {
            parent[ancestor_a] = ancestor_b;
            size[ancestor_b] += size[ancestor_a];
        }
        count--;
    }
    int find(int a){
        while(parent[a] != a){
            //路径压缩
            parent[a] = parent[parent[a]];
            a = parent[a];
        }
        return a;
    }
    bool isconnected(int a, int b){
        int ancestor_a = find(a);
        int ancestor_b = find(b);
        if(ancestor_a == ancestor_b) return true;
        return false;
    }
    int _count(){
        return count;
    }
};
```

这个问题用 Union-Find 算法就显得十分优美了。题目是这样：

给你一个数组 `equations`，装着若干字符串表示的算式。每个算式 `equations[i]` 长度都是 4，而且只有这两种情况：`a==b` 或者 `a!=b`，其中 `a,b` 可以是任意小写字母。你写一个算法，如果 `equations` 中所有算式都不会互相冲突，返回 `true`，否则返回 `false`。

比如说，输入 `["a==b", "b!=c", "c==a"]`，算法返回 `false`，因为这三个算式不可能同时正确。

再比如，输入 `["c==c", "b==d", "x!=z"]`，算法返回 `true`，因为这三个算式并不会造成逻辑冲突。

我们前文说过，动态连通性其实是一种等价关系，具有「自反性」「传递性」和「对称性」，其实 `==` 关系也是一种等价关系，具有这些性质。所以这个问题用 Union-Find 算法就很自然。

核心思想是，将 `equations` 中的算式根据 `==` 和 `!=` 分成两部分，先处理 `==` 算式，使得他们通过相等关系各自勾结成门派；然后处理 `!=` 算式，检查不等关系是否破坏了相等关系的连通性。

```
boolean equationsPossible(String[] equations) {
    // 26 个英文字母
    UF uf = new UF(26);
    // 先让相等的字母形成连通分量
    for (String eq : equations) {
        if (eq.charAt(1) == '=') {
            char x = eq.charAt(0);
            char y = eq.charAt(3);
            uf.union(x - 'a', y - 'a');
        }
    }
    // 检查不等关系是否打破相等关系的连通性
    for (String eq : equations) {
        if (eq.charAt(1) == '!') {
            char x = eq.charAt(0);
            char y = eq.charAt(3);
            // 如果相等关系成立，就是逻辑冲突
            if (uf.connected(x - 'a', y - 'a'))
                return false;
        }
    }
    return true;
}
```

## LRU (least recently used) 背！

LUR缓存 <https://leetcode-cn.com/problems/lru-cache/>

实现 LRU Cache 类：

**LRUCache(int capacity)** 以正整数作为容量 capacity 初始化 LRU 缓存

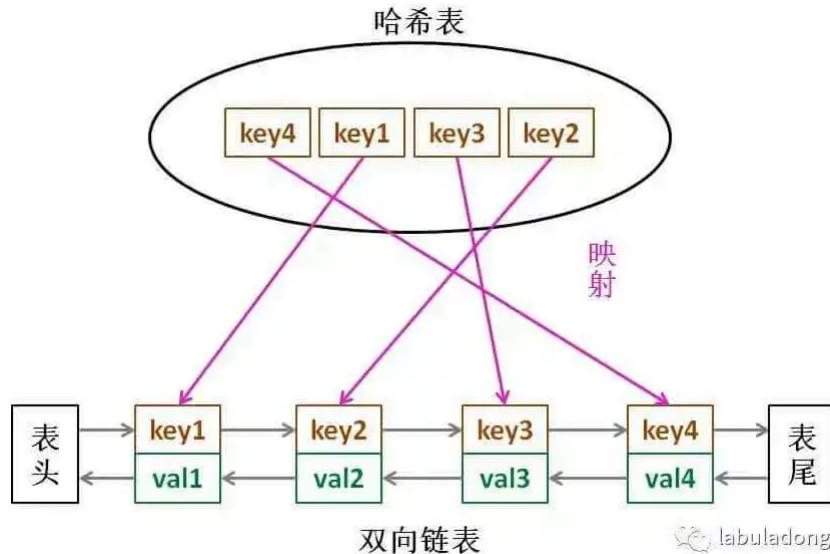
**int get(int key)** 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1。

**void put(int key, int value)** 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最近未使用的数据值，从而为新的数据值留出空间。

在 O(1) 时间复杂度内完成这两种操作

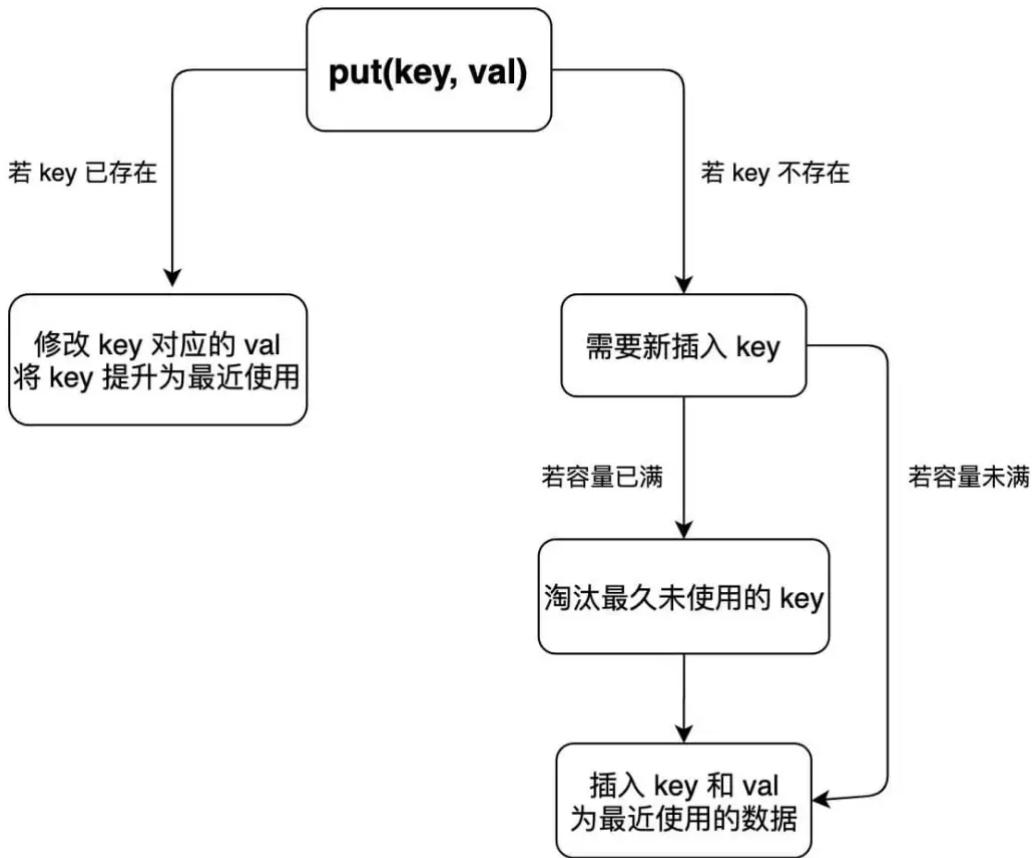
LRU 缓存机制可以通过哈希表辅以双向链表实现，我们用一个哈希表和一个双向链表维护所有在缓存中的键值对。

- 双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。
- 哈希表即为普通的哈希映射，通过缓存数据的键映射到其在双向链表中的位置。



我们首先使用哈希表进行定位，找出缓存项在双向链表中的位置，随后将其移动到双向链表的头部，即可在 O(1) 的时间内完成 get 或者 put 操作。具体的方法如下：

- 对于 get 操作，首先判断 key 是否存在：
  - 如果 key 不存在，则返回 -1-1；
  - 如果 key 存在，则 key 对应的节点是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。
- 对于 put 操作，首先判断 key 是否存在：



- 如果 key 不存在，使用 key 和 value 创建一个新的节点，在双向链表的头部添加该节点，并将 key 和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；
- 如果 key 存在，则与 get 操作类似，先通过哈希表定位，再将对应的节点的值更新为 value，并将该节点移到双向链表的头部。

上述各项操作中，访问哈希表的时间复杂度为 O(1)，在双向链表的头部添加节点、在双向链表的尾部删除节点的复杂度也为 O(1)。而将一个节点移到双向链表的头部，可以分成「删除该节点」和「在双向链表的头部添加节点」两步操作，都可以在 O(1) 时间内完成。

### 小贴士☆

在双向链表的实现中，使用一个伪头部（dummy head）和伪尾部（dummy tail）标记界限，这样在添加节点和删除节点的时候就不需要检查相邻的节点是否存在。

```

class DlinkedNode{//双向链表
public:
    int val,key;
    DlinkedNode* pre;
    DlinkedNode* nxt;
    DlinkedNode():val(0),key(0),pre(nullptr),nxt(nullptr){}
    DlinkedNode(int k,int v):val(v),key(k),pre(nullptr),nxt(nullptr){}//参数列表的
顺序绝对别错！按照下面put的参数顺序来，别太随意
};
//-----
class LRUCache {
private:
    DlinkedNode* head;
    DlinkedNode* tail;
    int size;
    int my_capacity;
}

```

```

unordered_map<int, DlinkedNode*> cache;

public:
    LRUcache(int capacity):my_capacity(capacity),size(0) {
        // 使用伪头部和伪尾部节点
        head = new DlinkedNode(); // 此处必须给头尾各一个实体
        tail = new DlinkedNode();
        head->nxt=tail;
        tail->pre=head;
    }

    int get(int key) {
        if(!cache.count(key)){
            return -1;
        }
        // 如果 key 存在，通过哈希表定位，移到头部，返回val
        moveToHead(cache[key]);
        return cache[key]->val;
    }

    void put(int key, int value) {
        if(!cache.count(key)){
            // 如果 key 不存在，创建一个新的节点
            DlinkedNode* node=new DlinkedNode(key,value);
            cache[key]=node; // 添加进哈希表
            addToHead(node); // 添加至双向链表的头部
            size++; // 记得加size
            if(size>my_capacity){
                // 如果超出容量，删除双向链表的尾部节点
                DlinkedNode* removetail=removeTail();
                cache.erase(removetail->key); // 删除哈希表中对应的项
                delete removetail; // 防止内存泄漏 写不写都行
                size--;
            }
        }else{
            // 如果 key 存在，通过哈希表定位，修改 value，并移到头部
            cache[key]->val=value;
            moveToHead(cache[key]);
        }
    }

    // 熟记以下函数，分别是 加到头后，删除节点，移到头后，删除尾前节点
    void addToHead(DlinkedNode* node){
        node->pre=head;
        node->nxt=head->nxt;
        head->nxt->pre=node;
        head->nxt=node;
    }

    void removeNode(DlinkedNode* node){
        node->pre->nxt=node->nxt;
        node->nxt->pre=node->pre;
    }

    void moveToHead(DlinkedNode* node){
        removeNode(node);
        addToHead(node);
    }

    DlinkedNode* removeTail(){ // 返回该节点 用于删除哈希表中的对应项
        DlinkedNode* node=tail->pre;
        removeNode(node);
    }
}

```

```
    return node;  
}  
  
};
```

## LFU (least frequently used) 背!

LFU缓存<https://leetcode-cn.com/problems/lru-cache/>

### 思路和算法

我们定义**两个哈希表**，第一个 `freq_table` 以频率 `freq` 为索引，每个索引存放一个双向链表，这个链表里存放所有使用频率为 `freq` 的缓存，缓存里存放三个信息，分别为键 `key`，值 `value`，以及使用频率 `freq`。第二个 `key_table` 以键值 `key` 为索引，每个索引存放对应缓存在 `freq_table` 中链表里的内存地址，这样我们就能利用两个哈希表来使得两个操作的时间复杂度均为  $O(1)$ 。同时需要记录一个当前缓存最少使用的频率 `minFreq`，这是为了删除操作服务的。

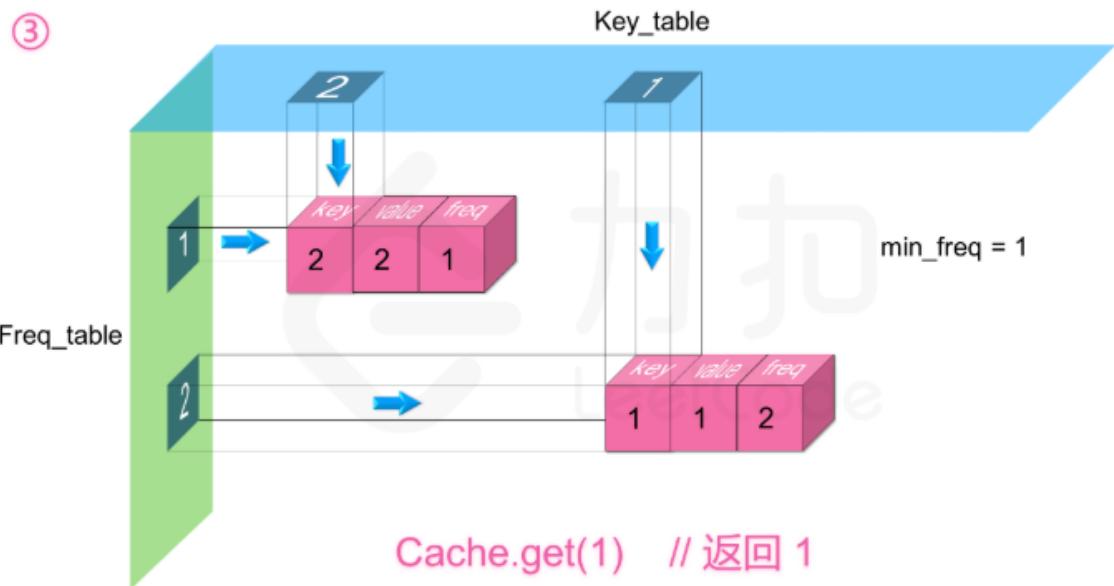
对于 `get(key)` 操作，我们能通过索引 `key` 在 `key_table` 中找到缓存在 `freq_table` 中的链表的内存地址，如果不存在直接返回 `-1`，否则我们能获取到对应缓存的相关信息，这样我们就能知道缓存的键值还有使用频率，直接返回 `key` 对应的值即可。

但是我们注意到 `get` 操作后这个缓存的使用频率加一了，所以我们需要更新缓存在哈希表 `freq_table` 中的位置。已知这个缓存的键 `key`，值 `value`，以及使用频率 `freq`，那么该缓存应该存放到 `freq_table` 中 `freq + 1` 索引下的链表中。所以我们在当前链表中  $O(1)$  删除该缓存对应的节点，根据情况更新 `minFreq` 值，然后将其  $O(1)$  插入到 `freq + 1` 索引下的链表头完成更新。这其中的操作复杂度均为  $O(1)$ 。插入到链表头是为了保证缓存在当前链表中从链表头到链表尾的插入时间是有序的，为下面的删除操作服务。

对于 `put(key, value)` 操作，我们先通过索引 `key` 在 `key_table` 中查看是否有对应的缓存，如果有的话，其实操作等价于 `get(key)` 操作，唯一的区别就是我们需要将当前的缓存里的值更新为 `value`。如果没有的话，相当于是新加入的缓存，如果缓存已经到达容量，需要先删除最近最少使用的缓存，再进行插入。

先考虑插入，由于是**新插入的**，所以**缓存的使用频率一定是 1**，所以我们将缓存的信息插入到 `freq_table` 中 `1` 索引下的列表头即可，同时更新 `key_table[key]` 的信息，以及更新 `minFreq = 1`。

那么剩下的就是删除操作了，由于我们实时维护了 `minFreq`，所以我们能够知道 `freq_table` 里目前最少使用频率的索引，同时因为我们保证了链表中从链表头到链表尾的插入时间是有序的，所以 `freq_table[minFreq]` 的链表中链表尾的节点即为使用频率最小且插入时间最早的节点，我们删除它同时根据情况更新 `minFreq`，整个时间复杂度均为  $O(1)$ 。



思路概览：

双哈希map表示；

☆由 key 从 keyTable 找到其在 freqTable 中的地址 it，具象成 node=it->second，并且 val = node->val，freq = node->freq;先从原 freqTable[freq] 中删除该节点 node，删除后看看 freqTable[freq].size() == 0 就从 freqTable 中 erase(node)，并且若 freq==minfreq，更新 minfreq=freq+1，然后往 freqTable[freq+1] 的头 push\_front(node)，并更新 keyTable[key] 为 freqTable[freq+1].begin();

注意：put() 的时候如果是新节点，minfreq 必然为 1。

```
// 缓存的节点信息
class Node
{
public:
    int key, val, freq;
    Node(int k, int v, int q):key(k), val(v), freq(q){}
};

class LFUCache {
private:
    int mcapacity;
    int minfreq;
    unordered_map<int, list<Node>::iterator> keyTable;
    unordered_map<int, list<Node>> freqTable;
public:
    LFUCache(int capacity):mcapacity(capacity), minfreq(0) {
        keyTable.clear();
        freqTable.clear();
    }

    int get(int key) {
        if(mcapacity==0) return -1;
        auto it = keyTable.find(key);
        if(it==keyTable.end()) return -1;
        auto node = it->second;
        freqTable.erase(it);
        if(freqTable[minfreq].size() == 0)
            freqTable.erase(minfreq);
        minfreq++;
        freqTable[minfreq].push_front(node);
        keyTable[key] = freqTable[minfreq].begin();
        return node->val;
    }

    void put(int key, int value) {
        if(mcapacity==0) return;
        if(keyTable.find(key) != keyTable.end())
            freqTable.erase(keyTable[key]);
        if(freqTable[minfreq].size() == 0)
            freqTable.erase(minfreq);
        minfreq++;
        freqTable[minfreq].push_front({key, value, 1});
        keyTable[key] = freqTable[minfreq].begin();
    }
};
```

```

int val = node->val, freq = node->freq;
freqTable[freq].erase(node);
// 如果当前链表为空，我们需要在哈希表中删除，且更新minFreq
if(freqTable[freq].size() == 0){ //每次删除都要进行大小判断 及时更新freqTable 和
minfreq
    freqTable.erase(freq);
    if(freq == minfreq){
        minfreq++;
    }
}
// 插入到 freq + 1 中
freqTable[freq+1].push_front(Node(key, val, freq+1));
keyTable[key] = freqTable[freq+1].begin();
return val; //记得返回val
}

void put(int key, int value) {
if(mcapacity==0) return;
auto it=keyTable.find(key);
if(it==keyTable.end()){
    if(keyTable.size()==mcapacity){ // 缓存已满，需要进行删除操作
        auto it2=freqTable[minfreq].back(); //这里可别写成end()...
        keyTable.erase(it2.key());
        freqTable[minfreq].pop_back(); //对于链表的末尾 是pop不是erase
        if(freqTable[minfreq].size() == 0){
            freqTable.erase(minfreq); //此处不更新minfreq 因为新放的minfreq必
为1
        }
    }
    freqTable[1].push_front(Node(key, value, 1));
    keyTable[key] = freqTable[1].begin();
    minfreq=1;
} else{ // 与 get 操作基本一致，除了需要更新缓存的值
    auto node = it->second;
    int freq = node->freq; //因为缓存值val要被value覆盖 就不存下来了
    freqTable[freq].erase(node);
    if(freqTable[freq].size() == 0){ //同get()
        freqTable.erase(freq);
        if(freq == minfreq){
            minfreq++;
        }
    }
    freqTable[freq+1].push_front(Node(key, value, freq+1));
    keyTable[key] = freqTable[freq+1].begin();
}
}

};


```

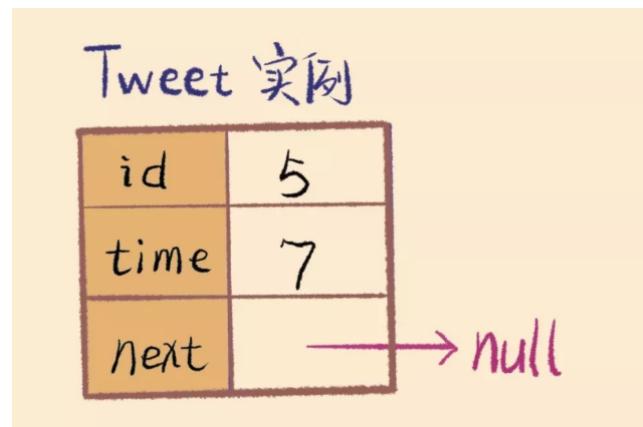
## 设计Twitter! 背!

<https://leetcode-cn.com/problems/design-twitter>

面向对象设计

我们需要一个 **User** 类，储存 user 信息，还需要一个 **Tweet** 类，储存推文信息，并且要作为链表的节点。

### 1. Tweet类实现

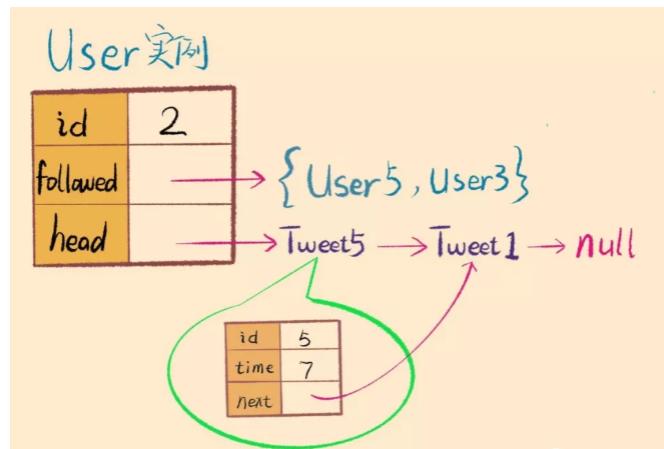


```
//推文类
class Tweet
{
public:
    int id;//推文id
    int time;//发推时间
    Tweet* nxt;//单链表形式 利用头插法表示时间先后
    Tweet(int i,int t):id(i),time(t),nxt(nullptr){} //构造函数
};
```

### 2. User类实现

一个用户需要存储的信息有 userId，关注列表，以及该用户发过的推文列表。

**关注列表**应该用**集合 (Hash Set)**这种数据结构来存，因为**不能重复**，而且需要**快速查找**；**推文列表**应该由**链表**这种数据结构储存，以便于进行**有序合并的操作**。



「关注」、「取关」和「发文」应该是 User 的行为，况且关注列表和推文链表也存储在 User 类中

```
//用户类
class User
{
public:
    int id;//用户id
    unordered_set<int> followed;//set 表示关注的人（包括自己）用set是为了防止重复关注同一个人
    Tweet* head; //发推链表
```

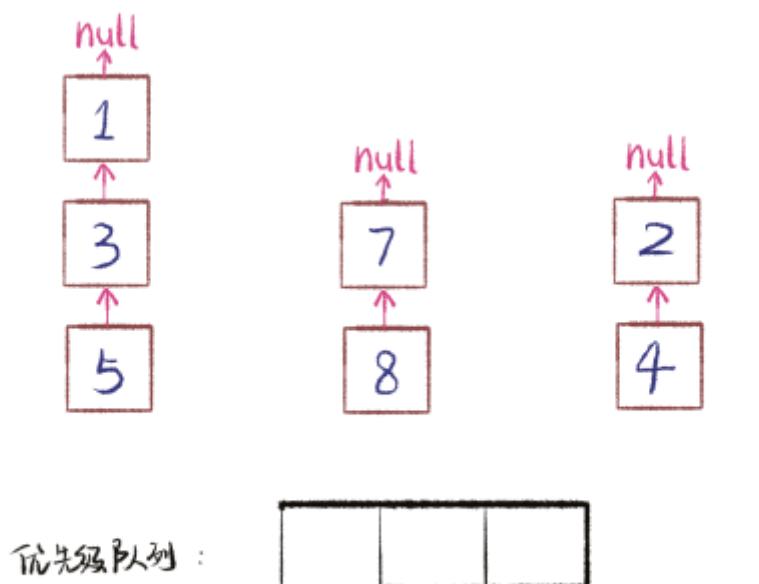
```

User(int i):id(i),head(nullptr){ //创建用户: id ,发推链表为空
    followed.clear(); //凡是这种模板数据结构 都用.clear()初始化!
    followed.insert(i); //自己关注自己 方便检索最近推文时 只要遍历关注set就行 不用单独
处理自己
}
void follow(int uid){ //关注列表更新
    followed.insert(uid);
}
void unfollow(int uid){ //取关
    if(uid!=id){ //不能取关自己!
        followed.erase(uid);
    }
}
void post(int twtid){ //发推
    Tweet* twt = new Tweet(twtid,timestamp); //创建推文实体
    timestamp++; //每次发帖全局时常++
    //头插法 越前面的越新
    twt->nxt=head;
    head=twt;
}
};


```

### 3. Twitter类的实现

☆合并 k 个有序链表的算法需要用到**优先级队列 (Priority Queue)**，这种数据结构是「二叉堆」最重要的应用。



res :

总体代码：

```

int timestamp=0; //全局时间
//推文类

```

```

class Tweet
{
public:
    int id;//推文id
    int time;//发推时间
    Tweet* nxt;//单链表形式 利用头插法表示时间先后
    Tweet(int i,int t):id(i),time(t),nxt(nullptr){}//构造函数
};

//用户类
class User
{
public:
    int id;//用户id
    unordered_set<int> followed;//set 表示关注的人（包括自己）用set是为了防止重复关注同一人
    Tweet* head; //发推链表

User(int i):id(i),head(nullptr){ //创建用户：id ,发推链表为空
    followed.clear(); //凡是这种模板数据结构 都用.clear()初始化!
    followed.insert(i); //自己关注自己 方便检索最近推文时 只要遍历关注set就行 不用单独处理自己
}
void follow(int uid){ //关注列表更新
    followed.insert(uid);
}
void unfollow(int uid){ //取关
    if(uid!=id){ //不能取关自己!
        followed.erase(uid);
    }
}
void post(int twtid){ //发推
    Tweet* twt = new Tweet(twtid,timestamp); //创建推文实体
    timestamp++; //每次发帖全局时间++! ! !
    //头插法 越前面的越新
    twt->nxt=head;
    head=twt;
}
};

//推特类
class Twitter {
private:
    unordered_map<int,User*> userMap;//哈希map 用用户id 检索用户 注意是User*
public:
    /** Initialize your data structure here. */
    Twitter() {
        userMap.clear(); //模板数据结构初始化 .clear()
    }

    /** Compose a new tweet. */
    void postTweet(int userId, int tweetId) { //某用户发推 每次对某个用户进行操作都要先判断是否有该用户存在！
        if(!userMap.count(userId)){ //如果没有该用户 先创建一个该用户
            userMap[userId]=new User(userId);
        }
    }
}

```

```

User* u = userMap[userID];
u->post(tweetId); //用户发推 调用User类的发推函数
}

/** Retrieve the 10 most recent tweet ids in the user's news feed. Each item
in the news feed must be posted by users who the user followed or by the user
herself. Tweets must be ordered from most recent to least recent. */
vector<int> getNewsFeed(int userID) { //检索关注列表 (包括自己的) 一共最新十条推文
    vector<int> res={};
    if(!userMap.count(userID))return res; // twitter中没有该用户 返回空
    unordered_set<int> users=userMap[userID]->followed; //用users 标准userId的
关注列表
    // 关键点难点!!!!!! 优先队列，即大顶堆的声明方式
    // 1.写一个cmp 结构体
    struct cmp
    {
        bool operator()(const Tweet* a, const Tweet* b){//利用重载操作符
operator() 对Tweet* 的 -> time 即发文时间作对比
            return a->time<b->time; //发文时间越大的说明越新 越晚放进堆中, time越小的越
早放进堆 就能达成time大顶堆
        }
    };
    // 2.利用结构体cmp的操作符重载函数 实现大顶堆 注意typename 是Tweet* , typename
sequence 是vector<Tweet*>! !
    priority_queue<Tweet*, vector<Tweet*>, cmp> pq;

    for(int uid:users){//遍历关注列表用户
        Tweet* twt = userMap[uid]->head; //用户的发推链表表头
        if(twt==nullptr)continue; //关注用户未发推就跳过
        pq.push(twt); //否则把该推文加入大顶堆pq
    }

    while(!pq.empty()){ //pq有推文时
        if(res.size()==10)break; // res 凑满10条就跳出
        Tweet* twt =pq.top(); //top()是关注列表所有人的发的所有推文中最新的推文
        res.push_back(twt->id); //添加推文id
        pq.pop(); //从大顶堆删除该最新推文
        if(twt->nxt!=nullptr){ //如果该推文链表还有下一个推文 将其放入大顶堆
            pq.push(twt->nxt);
        }
    }
    return res;
}

/** Follower follows a followee. If the operation is invalid, it should be a
no-op. */
void follow(int followerId, int followeeId) {
    if(!userMap.count(followerId)){//关注用户是否存在 不存在就创建
        userMap[followerId]=new User(followerId);
    }
    if(!userMap.count(followeeId)){
        userMap[followeeId]=new User(followeeId); //被关注用户是否存在 不存在就创建
    }
    userMap[followerId]->follow(followeeId); //调用User的follow()
}

/** Follower unfollows a followee. If the operation is invalid, it should be
a no-op. */

```

```

    void unfollow(int followerId, int followeeId) {
        if(userMap.count(followerId)){//被取关用户是否存在 不存在就无视
            userMap[followerId]->unfollow(followeeId); //存在就调用User的unfollow()
        }
    }
};

```

## 单调栈 \* 3

I. <https://leetcode-cn.com/problems/next-greater-element-i/>

下一个更大元素 I

给你两个 没有重复元素 的数组 `nums1` 和 `nums2`，其中 `nums1` 是 `nums2` 的子集。

请你找出 `nums1` 中每个元素在 `nums2` 中的下一个比其大的值。

`nums1` 中数字 `x` 的下一个更大元素是指 `x` 在 `nums2` 中对应位置的右边的第一个比 `x` 大的元素。如果不存在，对应位置输出 `-1`。

根据题意，数组 `nums1` 视为询问。我们可以：

- 先对 `nums2` 中的每一个元素，求出它的右边第一个更大的元素；
- 将上一步的对应关系放入哈希表（HashMap）中；
- 再遍历数组 `nums1`，根据哈希表找出答案。

维护栈保证单调性：栈中的元素从栈顶到栈底是**单调不降**的。当我们遇到一个新的元素 `nums2[i]` 时，我们判断栈顶元素是否小于 `nums2[i]`，如果是，那么栈顶元素的下一个更大元素即为 `nums2[i]`，我们将栈顶元素出栈。重复这一操作，直到栈为空或者栈顶元素大于 `nums2[i]`。

**单调栈保存的是`nums2`的元素**

```

class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        vector<int> res;
        int n1 = nums1.size(), n2=nums2.size();
        unordered_map<int,int> m;
        stack<int> s;
        //单调栈模板
        for(int i =0;i<n2;i++){
            while(!s.empty() && nums2[i]>s.top()){
                m[s.top()]=nums2[i];//输出对应关系
                s.pop();
            }
            s.push(nums2[i]);
        }
        //遍历nums1 得到答案
        for(int i=0;i<n1;i++){
            if(!m.count(nums1[i])){
                res.push_back(-1);
            }else{
                res.push_back(m[nums1[i]]);
            }
        }
    }
};

```

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

II. <https://leetcode-cn.com/problems/daily-temperatures/>

## 每日温度

请根据每日气温列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 `temperatures = [73, 74, 75, 71, 69, 72, 76, 73]`，你的输出应该是 `[1, 1, 4, 2, 1, 1, 0, 0]`。

**单调栈中保存的是下标**。没啥好说的 熟练掌握：

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        int n = temperatures.size();
        vector<int> res(n, 0); // 右边没有比自己大的就不会被赋值 默认为0
        stack<int> s;
        // 单调栈模板
        for(int i = 0; i < n; i++) {
            while(!s.empty() && temperatures[s.top()] < temperatures[i]) {
                res[s.top()] = i - s.top(); // 返回的是索引差
                s.pop();
            }
            s.push(i);
        }

        return res;
    }
};
```

III. <https://leetcode-cn.com/problems/next-greater-element-ii/>

## 下一个更大元素 II

给定一个**循环数组**（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 `x` 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 `-1`。

基本与上一题一致，只遍历一次序列是不够的，例如序列 `[2,3,1][2,3,1]`，最后单调栈中将剩余 `[3,1][3,1]`，其中元素 `[1][1]` 的下一个更大元素还是不知道的。

一个朴素的思想是，我们可以**把这个循环数组「拉直」**，即复制该序列的前  $n-1$  个元素拼接在原序列的后面。这样我们就可以将这个新序列当作普通序列，用上文的方法来处理。

而在本题中，我们**不需要显性地**将该循环数组「拉直」，而只需要在处理时**对下标取模**即可。

**单调栈中保存的是下标**

```
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {
        int n = nums.size();
        vector<int> res(n, -1); // 右边没有比自己大的就不会被赋值 默认为-1
```

```

stack<int> s;
//单调栈模板
for(int i=0;i<2*n-1;i++){
    while(!s.empty()&& nums[s.top()]<nums[i%n]){
        res[s.top()]=nums[i%n]; //输出的是下标对应的数
        s.pop();
    }
    s.push(i%n);
}
return res;
};


```

## 滑动窗口的最大值 背！

<https://leetcode-cn.com/problems/sliding-window-maximum/>

没啥好说的

```

class Myqueue
{
public:
    deque<int> dq;
    int front(){ //front()当前滑动窗口的最大值
        return dq.front();
    }
    void push(int t){
        while(!dq.empty()&&t>dq.back()){ //维护一个单调不增队列
            dq.pop_back();
        }
        dq.push_back(t);
    }
    void pop(int t){ //只有当pop的数==最大值才pop，因为其他的数可能都甚至不在dq中
        if(t==dq.front()){
            dq.pop_front();
        }
    }
};

class Solution {
public:

    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> res;
        Myqueue dq;
        for(int i = 0 ;i<k;i++){ //先放前k个
            dq.push(nums[i]);
        }
        res.push_back(dq.front());

        for(int i = k ;i<nums.size();i++){
            dq.push(nums[i]);
            dq.pop(nums[i-k]);
            res.push_back(dq.front());
        }
        return res;
    }
};


```

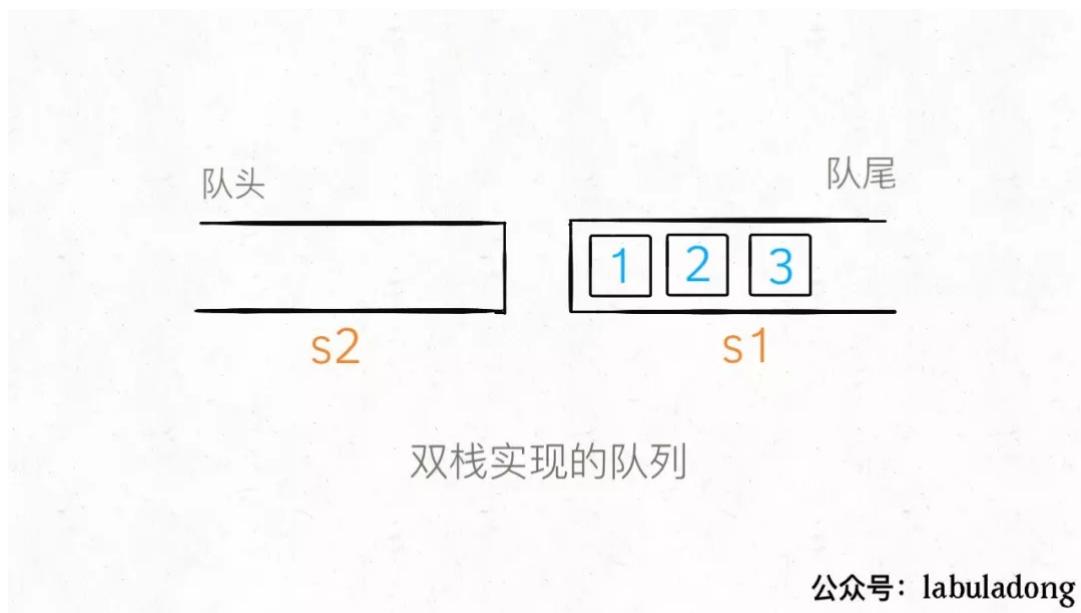
```
    }  
};
```

## 双栈实现队列 & 队列实现栈

I.<https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/>

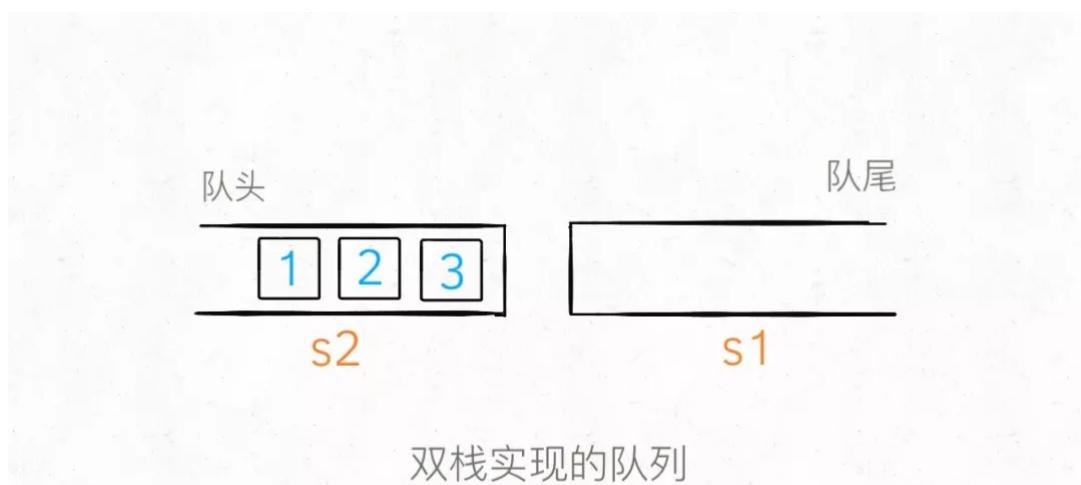
双栈实现队列

调用 `push` 让元素入队时，只要把元素压入 `s1` 即可，比如说 `push` 进 3 个元素分别是 1,2,3，那么底层结构就是这样：



公众号：labuladong

当 `s2` 为空时，可以把 `s1` 的所有元素取出再添加进 `s2`，这时候 `s2` 中元素就是先进先出顺序了



公众号：labuladong

对于 `pop` 操作，只要操作 `s2` 就可以了。

如果两个栈都为空的话，就说明队列为空。

```
class CQueue {  
public:  
    stack<int> s1, s2;
```

```

list<int> q;
CQueue() {
    while(!s1.empty()){
        s1.pop();
    }
    while(!s2.empty()){
        s2.pop();
    }
}

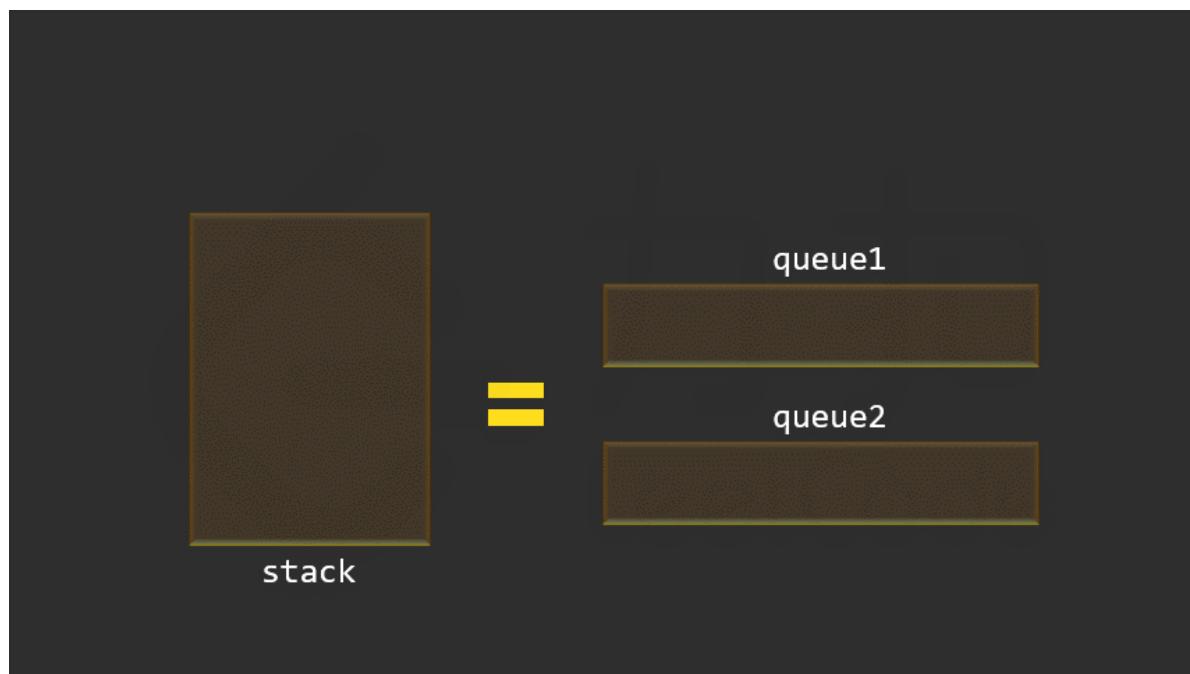
void appendTail(int value){s1.push(value);}//加入队列

int deleteHead(){//退出队列
    if(s2.empty()){
        if(s1.empty())return -1;
        while(!s1.empty()){
            s2.push(s1.top());
            s1.pop();
        }
    }
    if(!s2.empty()){
        int res = s2.top();
        s2.pop();
        return res;
    }
    return -1;
}
};


```

II. <https://leetcode-cn.com/problems/implement-stack-using-queues/>

队列实现栈



```

class MyStack {
public:

```

```

queue<int> q1;
queue<int> q2;//只作为push时候的临时队列
/** Initialize your data structure here. */
MyStack() {
    while(!q1.empty())
        q1.pop();
    while(!q2.empty())
        q2.pop();
}

/** Push element x onto stack. */
void push(int x) {
    q2.push(x);
    while(!q1.empty()){
        q2.push(q1.front());
        q1.pop();
    }
    swap(q1,q2);
}

/** Removes the element on top of the stack and returns that element. */
int pop() {
    int p=q1.front();
    q1.pop();
    return p;
}

/** Get the top element. */
int top() {
    return q1.front();
}

/** Returns whether the stack is empty. */
bool empty() {
    return q1.empty();
}

```

## 二分搜索 的妙用

I.<https://leetcode-cn.com/problems/koko-eating-bananas/>

koko吃香蕉

珂珂喜欢吃香蕉。这里有  $N$  堆香蕉，第  $i$  堆中有  $piles[i]$  根香蕉。警卫已经离开了，将在  $H$  小时后回来。

珂珂可以决定她吃香蕉的速度  $K$  (单位：根/小时)。个小时，她将选择一堆香蕉，从中吃掉  $K$  根。如果这堆香蕉少于  $K$  根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。

返回她可以在  $H$  小时内吃掉所有香蕉的最小速度  $K$  ( $K$  为整数)。

抛开二分查找技巧，想想如何暴力解决这个问题呢？

首先，算法要求的是「 $H$ 小时内吃完香蕉的最小速度」，我们不妨称为 `speed`，请问 `speed` 最大可能为多少，最少可能为多少呢？

显然最少为 1，最大为 `max(piles)`，因为一小时最多只能吃一堆香蕉。那么暴力解法就很简单了，只要从 1 开始穷举到 `max(piles)`，一旦发现某个值可以在  $H$  小时内吃完所有香蕉，这个值就是最小速度。

☆ 连续的空间线性搜索，这就是二分查找可以发挥作用的标志

```
class Solution {
public:
    int minEatingSpeed(vector<int>& piles, int h) {
        //二分查找框架 ez
        int l=1,r=pow(10,9);
        while(l<=r){
            int m = l+(r-l)/2;
            if(eatHour(piles, h, m)){
                r=m-1;
            }else if(!eatHour(piles, h, m)){
                l=m+1;
            }
        }
        return l;
    }

    bool eatHour(vector<int>& piles,int h,int K){
        int time=0;
        for(auto p:piles){
            //向上取整 如果用ceil几点改成double
            time+=ceil(double(p)/double(K));//(p-1)/K + 1
        }
        return time<=h;
    }
};
```

II. <https://leetcode-cn.com/problems/capacity-to-ship-packages-within-d-days/>

在  $D$  天内送达包裹的能力

传送带上的包裹必须在  $D$  天内从一个港口运送到另一个港口。

传送带上的第  $i$  个包裹的重量为 `weights[i]`。每一天，我们都会按给出重量的顺序往传送带上装载包裹。我们装载的重量不会超过船的最大运载重量。

返回能在  $D$  天内将传送带上的所有包裹送达的船的最低运载能力。

和上一题差不多 找到最小的能达成条件的运载量

```
class Solution {
public:
    int shipWithinDays(vector<int>& weights, int days) {
        //常规的二分查找 ez
        int l=1,r=500*5*pow(10,4);
        while(l<=r){
            int m=l+(r-l)/2;
```

```

        if(tranDay(weights, days, m)){
            r=m-1;
        }else if(!tranDay(weights, days, m)){
            l=m+1;
        }
    }
    return l;
}

bool tranDay(vector<int>& weights,int days,int cap){
    int i=0;
    for(int d=0;d<days;d++){
        int maxCap=cap;
        while((maxCap-weights[i])>=0){ //顺序存放！

            maxCap-=weights[i];//注意 先减
            i++;//在让i++
            if(i==weights.size()){//当最后一个包裹送出 索引++ 变成weights.size()
days天内说明全都送出了
                return true;
            }
        }
    }
    return false;
}
};

```

## O(1) 时间，查找/删除数组中的任意元素

I.<https://leetcode-cn.com/problems/insert-delete-getrandom-o1/>

O(1) 时间插入、删除和获取随机元素

设计一个支持在平均 时间复杂度 O(1) 下，执行以下操作的数据结构。

insert(val): 当元素 val 不存在时，向集合中插入该项。

remove(val): 元素 val 存在时，从集合中移除该项。

getRandom: 随机返回现有集合中的一项。每个元素应该有相同的概率被返回。

如果想「等概率」且「在 O(1) 的时间」取出元素，一定要满足：**底层用数组实现，且数组必须是紧凑的**。这样我们就可以直接生成随机数作为索引，从数组中取出该随机索引对应的元素，作为随机元素。

**但如果用数组存储元素的话，插入，删除的时间复杂度怎么可能是 O(1) 呢？**

可以做到！对数组尾部进行插入和删除操作不会涉及数据搬移，时间复杂度是 O(1)。

**所以，如果我们想在 O(1) 的时间删除数组中的某一个元素 val，可以把这个元素交换到数组的尾部，然后再 pop 掉。**

```

class RandomizedSet {
public:
    /** Initialize your data structure here. */
    vector<int> v;// 存储元素的值
    unordered_map<int, int> m;// 记录每个元素对应在 nums 中的索引

```

```

RandomizedSet() {
    v.clear();
    m.clear();
}

/** Inserts a value to the set. Returns true if the set did not already
contain the specified element. */
bool insert(int val) {
    if(m.count(val))return false;// 若 val 已存在，不用再插入
    v.push_back(val);// 若 val 不存在，插入到 nums 尾部
    m[val]=v.size()-1;// 并记录 val 对应的索引值
    return true;
}

/** Removes a value from the set. Returns true if the set contained the
specified element. */
bool remove(int val) {
    if(!m.count(val))return false;// 若 val 不存在，不用再删除
    int validx=m[val];// 先拿到 val 的索引
    m[v.back()]=validx;// 将最后一个元素对应的索引修改为 index
    swap(v.back(), v[validx]); // 交换 val 和最后一个元素
    v.pop_back(); // 在数组中删除元素 val
    m.erase(val); // 删除元素 val 对应的索引
    return true;
}

/** Get a random element from the set. */
int getRandom() {
    return v[rand()%v.size()]; // 随机获取 nums 中的一个元素
}
};

```

II.<https://leetcode-cn.com/problems/random-pick-with-blacklist/>

### 黑名单中的随机数

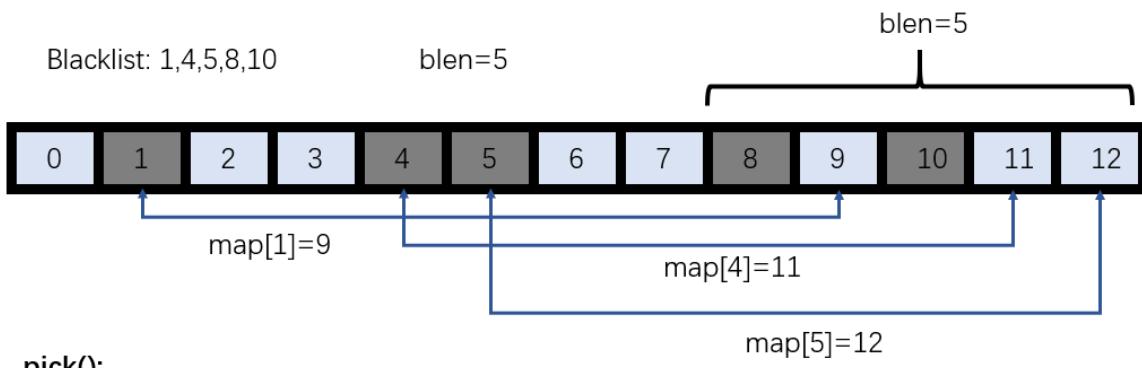
给定一个包含  $[0, n]$  中不重复整数的黑名单 `blacklist`，写一个函数从  $[0, n]$  中返回一个不在 `blacklist` 中的随机整数。

### 黑名单映射

基本思想是：设总名单长度为 `N`，黑名单长度为 `b1en`，则白名单长度为 `N-b1en`。黑名单分散在总名单的各个位置，有的分布在  $[0, N-b1en]$ ，有的则分布在  $[N-b1en, N]$ 。首先我们生成  $[0, N-b1en)$  中的随机数，那么：

- 对于分布在  $[N-b1en, N)$  中的黑名单成员，不用管它们，因为生成的随机数达不到这个范围。
- 对于分布在  $[0, N-b1en)$  中的黑名单成员，完全有可能碰撞到它们。为他们建立一一映射，映射到  $[N-b1en, N)$  范围内的白名单成员中去。

最终的效果就是能够随机均匀挑选到所有白名单成员。图示：



### **pick():**

First: pick index randomly between 0-7 as c  
 Second: if not in Blacklist, return c  
     else return map[c]

```
class Solution {
public:
    int w1;
    unordered_map<int, int> m;

    Solution(int n, vector<int>& blacklist) {
        m.clear();

        w1=n-blacklist.size(); //w1=N-blens
        unordered_set<int> s;
        for(int i=w1;i<n;i++)s.insert(i); //往s中放入[N-blens,N)的值
        for(int x:blacklist)s.erase(x); //把s中的黑名单中的值去掉
        auto it = s.begin(); //指向维护的[N-blens,N)中的白名单成员
        for(int x:blacklist){
            if(x<w1) //一定要注意只有[0,w1)的黑名单成员参与映射。因为是初始条件是[0,n)，所以这里也一样[0,w1)
                m[x]=*it++;
        }
    }

    int pick() {
        int rd=rand()%w1;
        return m.count(rd)?m[rd]:rd;
    }
};
```

## 双指针秒杀 数组/链表删除元素 题

I.<https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/>

删除排序数组中的重复元素

给你一个有序数组 `nums`，请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次**，返回删除后数组的新长度。

☆一切这种题的总体思路都是压缩数组/链表 而不是真的去删除！

遇到要删除的元素fast跳过 注意：while内先写fast跳跃条件！if！

nums

0	0	1	2	2	3	3
---	---	---	---	---	---	---

公众号：labuladong

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty())return 0;
        if(nums.size()==1)return nums[0];
        int slow = 0, fast = 0;
        while(fast<nums.size()){
            if(nums[fast] != nums[slow]){//先写fast跳跃条件! if!
                slow++;
                nums[slow]=nums[fast];
            }
            fast++;
        }
        return slow+1;
    }
};
```

II.<https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/>

删除排序链表中的重复元素

存在一个按升序排列的链表，给你这个链表的头节点 head，请你删除所有重复的元素，使每个元素 只出现一次。

同上！遇到要删除的元素fast跳过

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if(head==NULL) return NULL;
        ListNode* slow=head , * fast = head;
        while(fast!=NULL){
            if(fast->val!=slow->val){//先写fast跳跃条件! if!
                slow=slow->next;
                slow->val=fast->val;
            }
            fast=fast->next;
        }
    }
};
```

```

    slow->next=nullptr;
    return head;
}
};

```

III.<https://leetcode-cn.com/problems/remove-element/>

给你一个数组 `nums` 和一个值 `val`，你需要 **原地** 移除所有数值等于 `val` 的元素，并返回移除后数组的新长度。

思路同上 都是遇到要删除的元素fast跳过

```

class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        if(nums.empty())return 0;
        int slow = 0, fast = 0;
        while(fast<nums.size()){//先写fast跳跃条件! if!
            if(nums[fast] != val){
                nums[slow]=nums[fast];//此处的顺序与上两题不同 思考nums[0]==val的情况
就能想明白
                slow++;
            }
            fast++;
        }
        return slow;
    }
};

```

IV.<https://leetcode-cn.com/problems/move-zeroes/>

移动零

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

同样是**压缩数组**，而不是真的去移动0；

```

class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int slow=0, fast=0;
        while(fast<nums.size()){
            if(nums[fast] != 0){//先写fast跳跃条件! if!
                nums[slow]=nums[fast];
                slow++;
            }
            fast++;
        }
        for(int i=slow;i<nums.size();i++){//后面的无用数据赋值为0就行
            nums[i]=0;
        }
    }
};

```

## 去除重复字母

<https://leetcode-cn.com/problems/remove-duplicate-letters/solution/shi-pin-c-xiao-xu-jiang-jie-zhan-by-huxiaoxu/>

如果忘记怎么做 直接看上面的题解

给你一个字符串  $s$ ，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证 **返回结果的字典序最小**（要求不能打乱其他字符的相对位置）。

```
class Solution {
public:
    string removeDuplicateLetters(string s) {
        string res;

        stack<char> st;
        unordered_map<char ,int> cnt; //记录某字母剩余可用个数
        unordered_map<char, bool> instr; //记录栈中是否有某字母
        for(auto ss:s)cnt[ss]++;
        for(int i=0;i<s.size();i++){
            if(instr[s[i]]){//如果栈中有该元素 跳过
                cnt[s[i]]--;
                continue;
            }
            //当 栈不为空 && 栈顶元素比当前元素大 (字典序) && 栈顶元素后面还有剩下 迭代pop
            while(!st.empty()&& s[i]<st.top() && cnt[st.top()]>0){ //只有st.top还有剩才pop
                instr[st.top()]=false;
                st.pop();
            }
            st.push(s[i]); //当前元素入栈
            cnt[s[i]]--;
            instr[s[i]]=true; //栈中存在当前元素
        }

        while(!st.empty()){ //注意顺序栈中是底->顶尽量满足字典序的 所以是st.top()+res
            res=st.top()+res;
            st.pop();
        }
        return res;
    }
};
```

## 位运算的妙用

### 一 有趣的位操作

#### 1. 利用或操作 | 和空格将英文字符转换为小写

```
('a' | ' ') = 'a'  
('A' | ' ') = 'a'
```

#### 2. 利用与操作 & 和下划线将英文字符转换为大写

```
('b' & '_') = 'B'  
('B' & '_') = 'B'
```

### 3. 利用异或操作 `^` 和空格进行英文字符大小写互换

```
('d' ^ ' ') = 'D'  
( 'D' ^ ' ') = 'd'
```

以上操作能够产生奇特效果的原因在于 ASCII 编码。字符其实也就是数字，恰巧这些字符对应的数字通过位运算就能得到正确的结果

### 4. 判断两个数是否异号

```
int x = -1, y = 2;  
bool f = ((x ^ y) < 0); // true  
  
int x = 3, y = 2;  
bool f = ((x ^ y) < 0); // false
```

**异号返回true。**这个技巧还是很实用的，利用的是补码编码的符号位。如果不用位运算来判断是否异号，需要使用 if else 分支，还挺麻烦的。读者可能想利用乘积或者商来判断两个数是否异号，但是这种方式可能造成溢出，从而出现错误。

### 5. 不用临时变量交换两个数

```
int a = 1, b = 2;  
a ^= b;  
b ^= a;  
a ^= b;  
// 现在 a = 2, b = 1
```

### 6. 加一

```
int n = 1;  
n = -~n;  
// 现在 n = 2
```

### 7. 减一

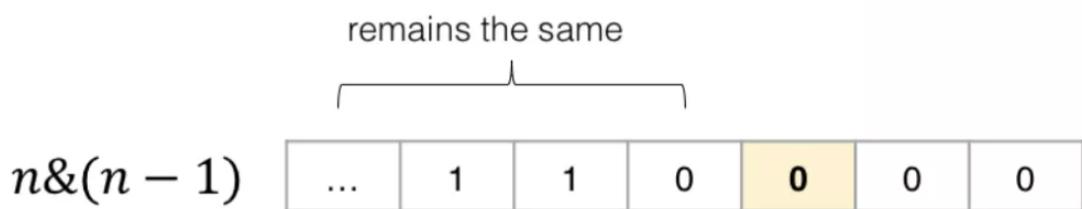
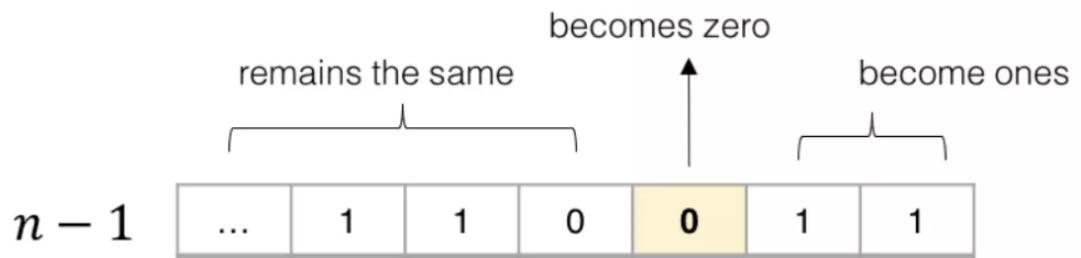
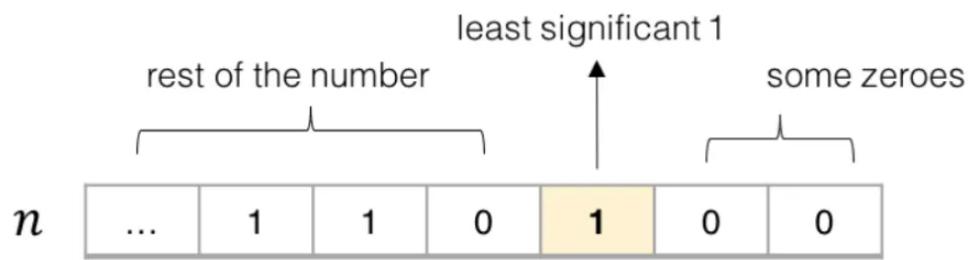
```
int n = 2;  
n = ~-n;  
// 现在 n = 1
```

PS：上面这三个操作就纯属装逼用的，没啥实际用处，大家了解了解乐呵一下就行。

## 二 算法常用操作

1.\*\* `n&(n-1)` 这个操作是算法中常见的，作用是消除数字 `n` 的二进制表示中的最后一个 1\*\*。

看个图就很容易理解了：



核心逻辑就是， $n - 1$  一定可以消除最后一个 1，同时把其后的 0 都变成 1，这样再和  $n$  做一次  $\&$  运算，就可以仅仅把最后一个 1 变成 0 了。

### 计算汉明权重 (Hamming Weight)

## 191. 位1的个数

难度 简单 202 喜欢 例题 文章 分享

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

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：000000000000000000000000000010000000

输出：1

解释：输入的二进制串 000000000000000000000000000010000000 中，共有一位为 '1'。

让你返回 n 的二进制表示中有几个 1。因为  $n \& (n - 1)$  可以消除最后一个 1，所以可以用一个循环不停地消除 1 同时计数，直到 n 变成 0 为止。

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int res=0;
        while(n!=0){
            n=n&(n-1);
            res = ~res;
        }
        return res;
    }
};
```

## 2. 判断一个数是不是 2 的指数

一个数如果是 2 的指数，那么它的二进制表示一定只含有一个 1：

```
2^0 = 1 = 0b0001
2^1 = 2 = 0b0010
2^2 = 4 = 0b0100
```

如果使用位运算技巧就很简单了（注意运算符优先级，括号不可以省略）：

```
bool isPowerOfTwo(int n) {
    if (n <= 0) return false;
    return (n & (n - 1)) == 0;
}
```

### 3. 查找只出现一次的元素

#### 136. 只出现一次的数字

难度 简单 1442 喜欢 举报 文档

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1：

```
输入: [2,2,1]
输出: 1
```

示例 2：

```
输入: [4,1,2,1,2]
输出: 4
```

这里就可以运用异或运算的性质：

一个数和它本身做异或运算结果为 0，即  $a \wedge a = 0$ ；一个数和 0 做异或运算的结果为它本身，即  $a \wedge 0 = a$ 。

对于这道题目，我们只要把所有数字进行异或，成对儿的数字就会变成 0，落单的数字和 0 做异或还是它本身，所以最后异或的结果就是只出现一次的元素：

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int res=0;
        for(int i:nums){
            res^=i;
        }
        return res;
    }
};
```

### 4. 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/>

## 剑指 Offer 56 - II. 数组中数字出现的次数 II

难度 中等 191 ☆ ⬤ 文 ⬤ ⬤

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。

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        //最终的结果值
        int res = 0;
        //int类型有32位，统计每一位1的个数
        for (int i = 0; i < 32; i++) {
            //统计第i位中1的个数
            int oneCount = 0;
            for (int j = 0; j < nums.size(); j++) {
                oneCount += (nums[j] >> i) & 1;
            }
            //如果1的个数不是3的倍数，说明那个只出现一次的数字
            //的二进制位中在这一位是1
            if (oneCount % 3 == 1)
                res |= 1 << i;
        }
        return res;
    }
};
```

## 阶乘题

! <https://leetcode-cn.com/problems/factorial-trailing-zeroes/>

### 172. 阶乘后的零

难度 简单 476 ☆ ⬤ 文 ⬤ ⬤

给定一个整数  $n$ ，返回  $n!$  结果尾数中零的数量。

不可能真去把  $n!$  的结果算出来，阶乘增长可是比指数增长都恐怖，趁早死了这条心吧。

两个数相乘结果末尾有 0，一定是因为两个数中有因子 2 和 5，因为  $10 = 2 \times 5$ 。

也就是说，问题转化为： $n!$  最多可以分解出多少个因子 2 和 5？

比如说  $n = 25$ ，那么  $25!$  最多可以分解出几个 2 和 5 相乘？这个主要取决于能分解出几个因子 5，因为每个偶数都能分解出因子 2，因子 2 肯定比因子 5 多得多。

$25!$  中 5 可以提供一个，10 可以提供一个，15 可以提供一个，20 可以提供一个，25 可以提供两个，总共有 6 个因子 5，所以  $25!$  的结果末尾就有 6 个 0。

现在，问题转化为： $n!$  最多可以分解出多少个因子 5？

难点在于像 25, 50, 125 这样的数，可以提供不止一个因子 5，怎么才能不漏掉呢？

首先， $125 / 5 = 25$ ，这一步就是计算有多少个像 5, 15, 20, 25 这些 5 的倍数，它们一定可以提供一个因子 5。

但是，这些足够吗？刚才说了，像 25, 50, 75 这些 25 的倍数，可以提供两个因子 5，那么我们再计算出  $125!$  中有  $125 / 25 = 5$  个 25 的倍数，它们每人可以额外再提供一个因子 5。

够了吗？我们发现  $125 = 5 \times 5 \times 5$ ，像 125, 250 这些 125 的倍数，可以提供 3 个因子 5，那么我们还得再计算出  $125!$  中有  $125 / 125 = 1$  个 125 的倍数，它还可以额外再提供一个因子 5。

这下应该够了， $125!$  最多可以分解出  $20 + 5 + 1 = 26$  个因子 5，也就是说阶乘结果的末尾有 26 个 0。

```
class Solution {
public:
    int trailingZeroes(int n) {
        int res=0;
        int five =5;
        while(five<=n){
            res+=n/five;
            five*=5;
        }
        return res;
    }
};
```

<https://leetcode-cn.com/problems/preimage-size-of-factorial-zeroes-function/>

### 793. 阶乘函数后 K 个零

难度 困难 山 67 ☆ ⚡ 🔍

$f(x)$  是  $x!$  末尾是 0 的数量。（回想一下  $x! = 1 * 2 * 3 * \dots * x$ ，且  $0! = 1$ ）

例如， $f(3) = 0$ ，因为  $3! = 6$  的末尾没有 0；而  $f(11) = 2$ ，因为  $11! = 39916800$  末端有 2 个 0。给定  $K$ ，找出多少个非负整数  $x$ ，能满足  $f(x) = K$ 。

一个直观地暴力解法就是穷举呗，因为随着  $n$  的增加， $n!$  肯定是递增的， $n!$  后面的 0 数量肯定也是递增的。

对于这种具有单调性的函数，用 for 循环遍历，可以用二分查找进行降维打击

算出最小的  $n$  使得  $n!$  末尾有  $K$  个 0，算出最小的  $n_2$  使得  $n_2!$  末尾有  $K+1$  个 0。一减就得出结果

认真读题，看看题目给的数据范围有多大。这道题目实际上给了限制， $K$  是在  $[0, 10^9]$  区间内的整数，也就是说， $n_{zeroe}(n)$  的结果最多可能达到  $10^9$ 。

$n_{zeroe}(\text{INT\_MAX})$  的结果，比  $10^9$  小一些，那再用  $\text{LONG\_MAX}$  算一下，远超  $10^9$  了，所以  $\text{LONG\_MAX}$  可以作为搜索的上界。

注意为了避免整型溢出的问题， $n_{zeroe}$  函数需要把所有数据类型改成 long：

```
class Solution {
public:
    int preimageSizeFZF(int k) {
        long l =0, r = LONG_MAX;

        while(l<=r){ // 最小的 n 使得 n! 末尾有 K 个 0
            long m = l+(r-l)/2;
```

```

    if(nZero(m)<k){
        l=m+1;
    }else{
        r=m-1;
    }
}//此时l为最小的n

long l2 =0, r2 = LONG_MAX;
while(l2<=r2){ //最小的n2 使得n2! 末尾有k+1个0
    long m = l2+(r2-l2)/2;
    if(nZero(m)<k+1){
        l2=m+1;
    }else{
        r2=m-1;
    }
}//此时l2为最小的n2
return l2-1; //一减就是末尾有k个0的N的个数
}

long nZero(long n){ //n! 末尾有几个0
    long res=0;
    long five=5;
    while(n>=five){
        res+=n/five;
        five*=5;
    }
    return res;
}
};

```

## 计算素数/质数

<https://leetcode-cn.com/problems/count-primes/>

### 一：质朴思路

枚举每个数字是否为素数。判断素数的方法参考定义，对于某个数字  $n$ ，从 2 开始枚举判断是否满足  $n \% i == 0$ ，如果找到了  $n$  的因子，就返回 false。注意  $i$  遍历到最大  $\sqrt{n}$  即可。因为  $n$  如果不是素数，那么至少有一个因子是小于等于  $\sqrt{n}$  的。

### 二：埃氏筛

初始化长度  $O(n)$  的标记数组，表示这个数组是否为质数。数组初始化所有的数都是质数。从 2 开始将当前数字的倍数全都标记为合数。标记到  $\sqrt{n}$  停止即可。

Prime numbers									
2	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

```
class Solution {
public:
    int countPrimes(int n) {
        vector<bool> isPrime(n,true);
        for(int i=2;i*i<n;i++){
            if(isPrime[i]){
                for(int j=i*i;j<n;j+=i)
                    isPrime[j]=false;
            }
        }

        int res=0;
        for(int i = 2; i < n; i++) {
            if (isPrime[i]) {
                res++;
            }
        }

        return res;
    }
};
```

## n皇后

$n \times n$  的棋盘放  $n$  个皇后

```
class Solution {
public:
    vector<vector<string>> res;

    void backtrace(vector<string>& s, int row, int n) {
```

```

        if (row == n) {
            res.push_back(s);
            return;
        }
        for (int i = 0; i < n; i++) {
            if (!Y(s, row, i, n)) {
                continue;
            }
            s[row][i] = 'Q';
            backtrace(s, row + 1, n);
            s[row][i] = '.';
        }
        return;
    }
    bool Y(vector<string>& s, int r, int c, int n) {
        /*for(int i=0;i<n;i++){//这里写错了 因为每行放完就进入下一行 不会出现重复 此处应该
        判断列!
        if(s[r][i]=='Q')return false;
        }*/
        for (int i = 0; i < r; i++) {//列
            if (s[i][c] == 'Q')return false;
        }
        for (int i = r - 1, j = c - 1; i >= 0 && j >= 0; j--, i--) {//左上
            if (s[i][j] == 'Q')return false;
        }
        for (int i = r - 1, j = c + 1; i >= 0 && j < n; j++, i--) {//右上
            if (s[i][j] == 'Q')return false;
        }
        return true;
    }
    vector<vector<string>> solveNQueens(int n) {
        vector<string> s(n, string(n, '.'));
        backtrace(s, 0, n);
        return res;
    }
};

int main() {
    Solution s;
    int a;
    cin >> a;
    s.solveNQueens(a);

    for (int i = 0; i < s.res.size(); i++) {
        for (auto it = s.res[i].begin(); it != s.res[i].end(); it++) {
            cout << *it << ',';
        }
        cout << endl;
    }
    return 0;
}

```

# 回溯解决子集！排列！组合！

## 78. 子集

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

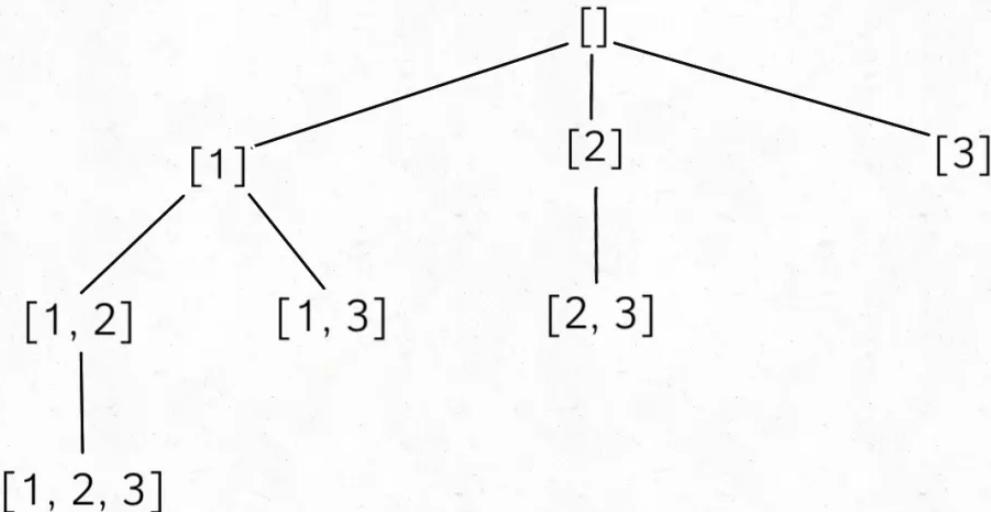
解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

### 方法一 递归（推荐）

典型的递归结构，`[1, 2, 3]` 的子集可以由 `[1, 2]` 追加得出，`[1, 2]` 的子集可以由 `[1]` 追加得出，base case 显然就是当输入集合为空集时，输出子集也就是一个空集。

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        if(nums.size() == 0) {
            return {{}}; // 是{{}}!!
        }
        int nb = nums.back();
        nums.pop_back();
        vector<vector<int>> res = subsets(nums);
        int len = res.size(); // 这里一定要先记录当前res的大小 因为进入循环以后res大小会改变
        for(int i=0; i<len; i++) {
            res.push_back(res[i]); // 给res原本元素制造一份副本
            res.back().push_back(nb); // 给副本加入nb
        }
        return res;
    }
};
```

### 方法二 回溯



公众号：labuladong

```
class Solution {
public:
    vector<vector<int>> res;
    void backtrace(vector<int>& nums, vector<int>& v, int start){
```

```

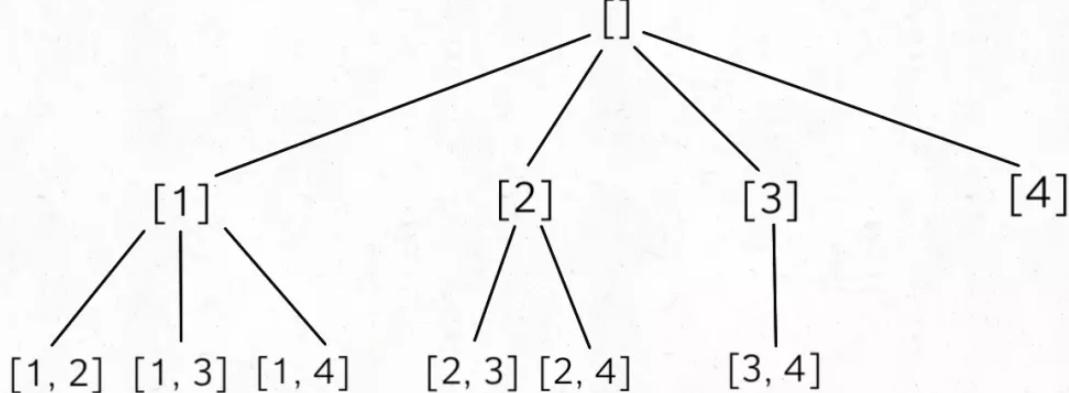
    res.push_back(v);
    for(int i = start; i < nums.size(); i++) {
        v.push_back(nums[i]);
        backtrace(nums, v, i+1);
        v.pop_back();
    }
}
vector<vector<int>> subsets(vector<int>& nums) {
    vector<int> v;
    backtrace(nums, v, 0);
    return res;
}
};

```

## 77. 组合

给定两个整数  $n$  和  $k$ , 返回  $1 \dots n$  中所有可能的  $k$  个数的组合。

典型的回溯算法,  $k$  限制了树的高度,  $n$  限制了树的宽度, 直接套我们以前讲过的回溯算法模板框架就行了:



公众号: labuladong

```

class Solution {
public:
    vector<vector<int>> res;
    void backtrace(vector<int> v, int n, int k, int start){
        if(v.size() == k){
            res.push_back(v); return;
        }
        for(int i = start; i <= n; i++){
            v.push_back(i);
            backtrace(v, n, k, i+1);
            v.pop_back();
        }
    }
    vector<vector<int>> combine(int n, int k) {
        vector<int> v;
        backtrace(v, n, k, 1);
        return res;
    }
};

```

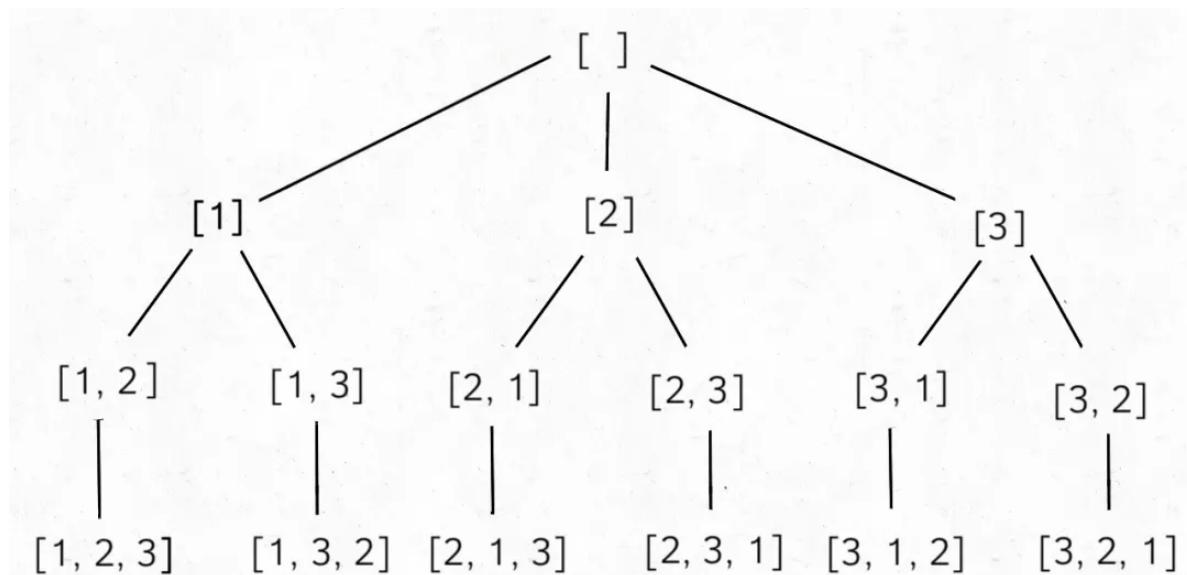
```

        return res;
    }
};


```

## 46. 全排列

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



公众号: labuladong

```

class Solution {
public:
    vector<vector<int>> res;
    void backtrace(vector<int>& nums, vector<int>& v, unordered_set<int> &s){
        if(v.size()==nums.size()){
            res.push_back(v);
            return ;
        }

        for(int i=0;i<nums.size();i++){
            if(s.count(nums[i])){
                continue;
            }
            v.push_back(nums[i]);
            s.insert(nums[i]);
            backtrace(nums, v, s);
            v.pop_back();
            s.erase(nums[i]);
        }
        return ;
    }
    vector<vector<int>> permute(vector<int>& nums) {
        vector<int> b;
        unordered_set<int> s;
        backtrace(nums, b, s);
        return res;
    }
};

```

排列问题的树比较对称，而组合问题的树越靠右节点越少。

排列问题每次通过unordered\_set的 count 方法来排除在 v 中已经选择过的数字；而组合问题通过传入一个 start 参数，来排除 start 索引之前的数字。

以上，就是排列组合和子集三个问题的解法，总结一下：

子集问题可以利用数学归纳思想（推荐），假设已知一个规模较小的问题的结果，思考如何推导出原问题的结果。也可以用回溯算法，要用 start 参数排除已选择的数字。

组合问题利用的是回溯思想，结果可以表示成树结构，我们只要套用回溯算法模板即可，关键点在于要用一个 start 排除已经选择过的数字。

排列问题是回溯思想，也可以表示成树结构套用算法模板，不同之处在于使用 count 方法排除已经选择的数字，前文有详细分析，这里主要是和组合问题作对比。

## 解数独

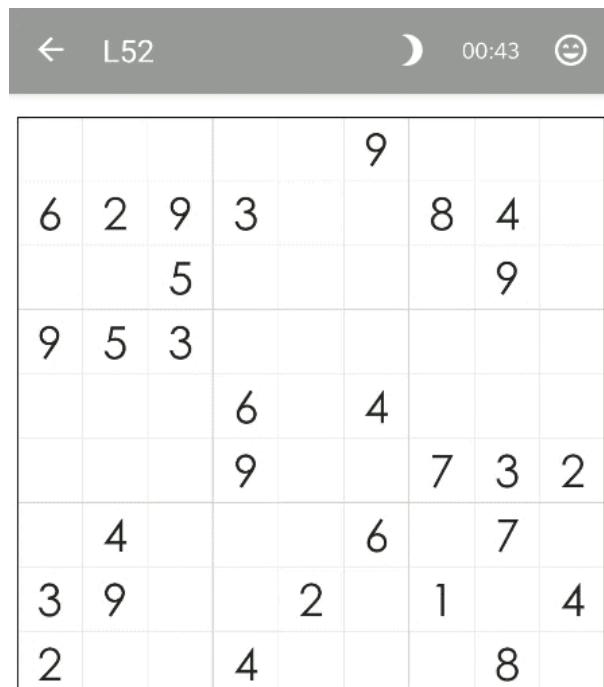
### 37. 解数独

编写一个程序，通过填充空格来解决数独问题。

数独的解法需遵循如下规则：

1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。（请参考示例图）

数独部分空格内已填入了数字，空白格用 ‘.’ 表示。



公众号：labuladong

算法的核心思路非常非常的简单，就是对每一个空着的格子穷举 1 到 9，如果遇到不合法的数字则跳过，如果找到一个合法的数字，则继续穷举下一个空格子。

当 j 到达超过最后一个索引时，转为增加 i 开始穷举下一行。

\*\* r == m 的时候就说明穷举完了最后一行，完成了所有的穷举，就是 base case\*\*。

为了减少复杂度，我们可以让 backtrack 函数返回值为\*\* bool，如果找到一个可行解就返回 true，这样就可以阻止后续的递归。只找一个可行解\*\*，也是题目的本意。

```
class Solution {
public:
    bool isvalue(vector<vector<char>>&b, int r, int c, char v){
        for(int i=0;i<9;i++){
            if(b[r][i]==v) return false;// 行
            else if(b[i][c]==v) return false;// 列
            else if(b[(r/3)*3+i/3][(c/3)*3+i%3]==v) return false;// 3 x 3 方框!!!
        }
        return true;
    }
    void solveSudoku(vector<vector<char>>& board) {
        dfs(board, 0, 0);
    }
    bool dfs(vector<vector<char>>&b, int r, int c){
        if(r==9){
            return true;// 找到一个可行解，触发 base case
        }
        if(c==9){
            return dfs(b, r+1, 0);// 穷举到最后一列的话就换到下一行重新开始。
        }
        if(b[r][c]!='.'){
            return dfs(b, r, c+1);// 如果有预设数字，不用我们穷举
        }
        for(char ch='1';ch<='9';ch++){
            if(!isvalue(b, r, c, ch)){// 如果遇到不合法的数字，就跳过
                continue;
            }
            b[r][c]=ch;
            if(dfs(b, r, c+1)){// 如果找到一个可行解，立即结束
                return true;// 只能在这里判断可行解
            }
            b[r][c]='.';
        }
        return false;// 穷举完 1~9，依然没有找到可行解，此路不通
    }
};
```

## 合法括号

### 22. 括号生成

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

示例 1：

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

思路：

1、一个「合法」括号组合的左括号数量一定等于右括号数量，这个显而易见。

2、对于一个「合法」的括号字符串组合 $\text{p}$ ，必然对于任何  $0 \leq i < \text{len}(\text{p})$  都有：子串  $\text{p}[0..i]$  中左括号的数量都大于或等于右括号的数量。

改写成如下问题：

现在有  $2n$  个位置，每个位置可以放置字符 ( 或者 )，组成的所有括号组合中，返回合法的组合

对于  $2n$  个位置，必然有  $n$  个左括号， $n$  个右括号，所以我们不是简单的记录穷举位置  $i$ ，而是用 `left` 记录还可以使用多少个左括号，用 `right` 记录还可以使用多少个右括号。

```
class Solution {
public:
    vector<string> res;
    string re;
    void backtrace(string& re, int l, int r){
        if(r < l) return; // 若左括号剩下的多，不合法
        if(l < 0 || r < 0) return; // 数量小于 0，不合法的
        if(l == 0 && r == 0){ // 当所有括号都恰好用完时，得到一个合法的括号组合
            res.push_back(re);
            return;
        }
        // 尝试放一个左括号
        re.push_back('(');
        backtrace(re, l-1, r);
        re.pop_back();
        // 尝试放一个右括号
        re.push_back(')');
        backtrace(re, l, r-1);
        re.pop_back();
    }
    vector<string> generateParenthesis(int n) {
        backtrace(re, n, n); // 开始可用的左括号数量为 n 个，可用的右括号数量为 n 个
        return res;
    }
};
```

## 密码锁 bfs

### [752. 打开转盘锁](#)

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字：'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'。每个拨轮可以自由旋转；例如把 '9' 变为 '0'，'0' 变为 '9'。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000'，一个代表四个拨轮的数字的字符串。

列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 `target` 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

```
class Solution {
public:
    string addstring(string cur, int j ){
        if(cur[j] == '9'){
            cur[j] = '0';
        }else{
            cur[j]++;
        }
    }
};
```

```

        return cur;
    }
    string minusstring(string cur,int j){
        if(cur[j]=='0'){
            cur[j]='9';
        }else{
            cur[j]--;
        }
        return cur;
    }

int bfs(vector<string>& deadends,string start,string target){
    queue<string> q;
    unordered_set<string> visit;
    for(auto dds:deadends){//deadends加入visit
        visit.insert(dds);
    }
//标准的bfs模板
    visit.insert(start);
    q.push(start);
    int step=0;

    while(!q.empty()){
        int sz= q.size();
        for(int i=0;i<sz;i++){
            auto cur = q.front();
            if(cur==target) return step;
            q.pop();
            for(int j=0;j<4;j++){
                string up=addstring(cur,j);
                string down = minusstring(cur,j);
                if(!visit.count(up)){
                    visit.insert(up);
                    q.push(up);
                }
                if(!visit.count(down)){
                    visit.insert(down);
                    q.push(down);
                }
            }
        }
        step++;//for循环外step++
    }
    return -1;
}

int openLock(vector<string>& deadends, string target) {
    for(auto dds:deadends) if(dds=="0000") return -1;//傻逼样例把起点放进deadends
    return bfs(deadends,"0000",target);
}
};

```

# 滑块拼图

## 773. 滑动谜题

在一个  $2 \times 3$  的板上 (board) 有 5 块砖瓦，用数字 1~5 来表示，以及一块空缺用 0 来表示。

一次移动定义为选择 0 与一个相邻的数字（上下左右）进行交换。

最终当板 board 的结果是  $[[1, 2, 3], [4, 5, 0]]$  谜板被解开。

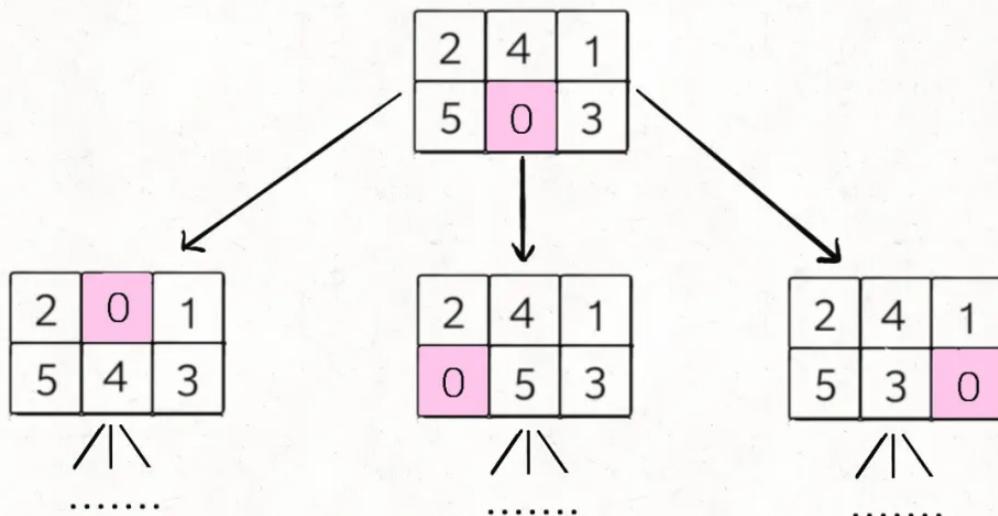
给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1。

最终目标：

1	2	3
4	5	0

计算最小步数的问题，我们就要敏感地想到 BFS 算法

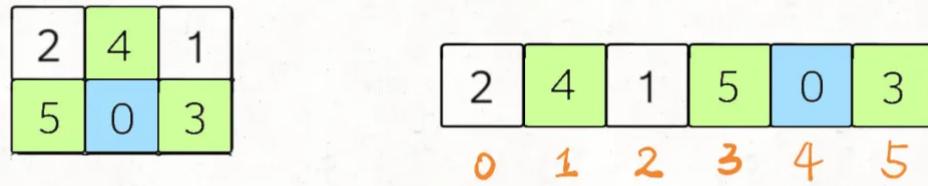
如何穷举出 board 当前局面下可能衍生出的所有局面？这就简单了，看数字 0 的位置呗，和上下左右的数字进行交换就行了：



公众号：labuladong

其中比较有技巧性的点在于，二维数组有「上下左右」的概念，压缩成一维后，如何得到某一个索引上下左右的索引？很简单，我们只要手动写出来这个映射就行了：

```
vector<vector<int>> neighbor = {
    { 1, 3 },
    { 0, 4, 2 },
    { 1, 5 },
    { 0, 4 },
    { 3, 1, 5 },
    { 4, 2 }
};
```



$$\text{neighbor}[4] = \{1, 3, 5\}$$

公众号: labuladong

```

class Solution {
public:
    int slidingPuzzle(vector<vector<int>>& board) {
        return bfs(board);
    }

    int bfs(vector<vector<int>>& board) {
        vector<vector<int>> neighbor={// 记录一维字符串的相邻索引
            { 1, 3 },
            { 0, 4, 2 },
            { 1, 5 },
            { 0, 4 },
            { 3, 1, 5 },
            { 4, 2 }
        };
        string target="123450";
        string start="";
        for(int i =0;i<board.size();i++){// 将 2x3 的数组转化成字符串
            for(int j=0;j<board[0].size();j++){
                //start.push_back(board[i][j]);
                start.push_back(board[i][j] + '0');
            }
        }

        queue<string> q;
        unordered_set<string> visit;
        q.push(start);
        visit.insert(start);
        int step=0;
        while(!q.empty()){
            int sz = q.size();
            for(int i =0;i<sz;i++){
                auto cur = q.front();
                q.pop();
                if(cur==target) return step;// 判断是否达到目标
                // 找到数字 0 的索引
                int idx=0;
                for(auto c:cur){
                    if(c!='0'){
                        idx++;
                    }
                }
            }
        }
    }
}

```

```

        }else{
            break;
        }
    }
    // 将数字 0 和相邻的数字交换位置!
    for(int adj:neibor[idx]){
        string tempcur = cur;//创建副本!
        swap(tempcur[idx], tempcur[adj]);
        if(!visit.count(tempcur)){// 防止走回头路
            visit.insert(tempcur);
            q.push(tempcur);
        }
    }
    step++;
}
return -1;
};


```

## 前缀和 数组 技巧

前缀和主要适用的场景是原始数组不会被修改的情况下，频繁查询某个区间的累加和。

### 1. 两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值 `target`** 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

这样我们创建一个哈希表，对于每一个  $x$ ，我们首先查询哈希表中是否存在  $target - x$ ，然后将  $x$  插入到哈希表中，即可保证不会让  $x$  和自己匹配。

```

class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> m;
        for(int i=0;i<nums.size();i++){
            if(m.count(target-nums[i])&&m[target-nums[i]]!=i){
                return {i,m[target-nums[i]]};
            }
            m[nums[i]]=i;
        }
        return {};
    }
};


```

### 560. 和为K的子数组

给定一个整数数组和一个整数  $k$ ，你需要找到该数组中和为  $k$  的连续的子数组的个数。

前缀和的思路是这样的，对于一个给定的数组 `nums`，我们额外开辟一个前缀和数组进行预处理：

```
class PrefixSum {
public:
    vector<int> prefix;
    // 输入一个数组，构造前缀和
    PrefixSum(vector<int> nums) {
        vector<int> prefix(nums.size() + 1, 0);
        for (int i = 1; i < prefix.size(); i++) {
            prefix[i] = prefix[i - 1] + nums[i - 1];
        }
    }
    // 查询闭区间 [i, j] 的累加和
    int query(int i, int j) {
        return prefix[j + 1] - prefix[i];
    }
}
```

```
int n = nums.length;
// 前缀和数组
vector<int> preSum(n + 1, 0);
preSum[0] = 0;
for (int i = 0; i < n; i++)
    preSum[i + 1] = preSum[i] + nums[i];
```

	0	1	2	3	4	5
nums	3	5	2	-2	4	1

	0	1	2	3	4	5	6
preSum	0	3	8	10	8	12	13

公众号：labuladong

```
class Solution {//超时
public:
    int subarraySum(vector<int>& nums, int k) {
        int n = nums.size();
        int res=0;
        // 构造前缀和
        vector<int> sum(n+1, 0); //sum[k]为前k个数的和
        for(int i=0;i<n;i++){
            sum[i+1]=sum[i]+nums[i];
        }
```

```

        for(int i=1;i<=n;i++){
            for(int j=0;j<i;j++){
                if(sum[i]-sum[j]==k)
                    res++;
            }
        }
        return res;
    };
}

```

上面的代码，发现该代码虽然用到了前缀和数组，但是对比暴力法并没有提升性能，时间复杂度仍为  $O(n^2)$ ，空间复杂度成了  $O(n)$ 。

### 前缀 优化：

需要注意的是，从左往右边更新边计算的时候已经保证了  $m[presum[i]-k]$  里记录的  $presum[j]$  的下标范围是  $0 \leq j \leq i$ 。同时，由于  $presum[i]$  的计算只与前一项的答案有关，因此我们可以不用建立  $presum$  数组，直接用  $presum$  变量来记录  $presum[i-1]$  的答案即可。

```

class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> m;
        m[0]=1;// 初始化需要把0插入，来覆盖全部的和或者第一个即为k的情况
        int res=0;
        int sum=0;
        for(int n:nums){
            sum+=n;
            if(m.count(sum-k)){
                res+=m[sum-k];
            }
            m[sum]++;
        }
        return res;
    };
}

```

## 差分 数组 技巧

差分数组的主要适用场景是频繁对原始数组的某个区间的元素进行增减。

对  $nums$  数组构造一个  $diff$  差分数组， $**diff[i]$  就是  $nums[i]$  和  $nums[i-1]$  之差\*\*：

```

vector<int> prefix(nums.size());
// 构造差分数组
diff[0] = nums[0];
for (int i = 1; i < nums.size(); i++) {
    diff[i] = nums[i] - nums[i - 1];
}

```

nums	8	2	6	3	1
------	---	---	---	---	---

diff	8	-6	4	-3	-2
------	---	----	---	----	----

公众号: labuladong

这样构造差分数组 `diff`, 就可以快速进行区间增减的操作, 如果你想对区间 `nums[i..j]` 的元素全部加 3, 那么只需要让 `diff[i] += 3`, 然后再让 `diff[j+1] -= 3` 即可

只要花费 O(1) 的时间修改 `diff` 数组, 就相当于给 `nums` 的整个区间做了修改。多次修改 `diff`, 然后通过 `diff` 数组反推, 即可得到 `nums` 修改后的结果。

把差分数组抽象成一个类, 包含 `increment` 方法和 `result` 方法:

```
class Difference {
    // 差分数组
    vector<int> diff;

    Difference(vector<int> nums) {
        assert nums.size() > 0;
        vector<int>diff(nums.size());
        // 构造差分数组
        diff[0] = nums[0];
        for (int i = 1; i < nums.size(); i++) {
            diff[i] = nums[i] - nums[i - 1];
        }
    }

    /* 给闭区间 [i,j] 增加 val (可以是负数) */
    void increment(int i, int j, int val) {
        diff[i] += val;
        if (j + 1 < diff.length) {
            diff[j + 1] -= val;
        }
    }

    vector<int> result() {
        vector<int> res (diff.size());
        // 根据差分数组构造结果数组
        res[0] = diff[0];
        for (int i = 1; i < diff.length; i++) {
            res[i] = res[i - 1] + diff[i];
        }
        return res;
    }
}
```

[1109. 航班预订统计](#)

这里有  $n$  个航班，它们分别从 1 到  $n$  进行编号。

有一份航班预订表 `bookings`，表中第  $i$  条预订记录 `bookings[i] = [firsti, lasti, seatsi]` 意味着在从 `firsti` 到 `lasti`（包含 `firsti` 和 `lasti`）的每个航班上预订了 `seatsi` 个座位。

请你返回一个长度为  $n$  的数组 `answer`，其中 `answer[i]` 是航班  $i$  上预订的座位总数。

问题转换为：

初始数组：输入一个长度为  $n$  的数组 `nums`，其中所有元素都是 0。

`[firsti, lasti, seatsi]` 相当于三元组 `(i, j, k)`，每个三元组的含义就是要求你给 `nums` 数组的闭区间 `[i-1, j-1]` 中所有元素都加上 `k`。

```
class Solution {
public:
    vector<int> corpFlightBookings(vector<vector<int>>& bookings, int n) {
        vector<int> diff(n, 0);
        for(auto bks:bookings){
            int i=bks[0]-1;
            int j=bks[1]-1;
            int val = bks[2];
            add(i, j, val, diff);
        }

        vector<int> res(diff.size());
        res[0]=diff[0];
        for(int i=1;i<diff.size();i++){
            res[i]=res[i-1]+diff[i];
        }
        return res;
    }

    void add(int i,int j,int val,vector<int>&diff){
        diff[i]+=val;
        if(j+1<diff.size()){
            diff[j+1]-=val;//
        }
    }
};
```

## 数组中的第K个最大元素

### [215. 数组中的第K个最大元素](#)

在未排序的数组中找到第  $k$  个最大的元素。请注意，你需要找的是数组排序后的第  $k$  个最大的元素，而不是第  $k$  个不同的元素。

#### 方法一 快速排序

```
class Solution {
public:
    int findkthLargest(vector<int>& nums, int k) {
        quicksort(nums, 0, nums.size()-1);
```

```

        return nums[nums.size()-k];
    }
    void quicksort(vector<int>& nums, int l, int r){
        if(l>=r) return ;
        int i = l, j=r;
        while(i<j){
            while(i<j&&nums[j]>=nums[l]) j--;
            while(i<j&&nums[i]<=nums[l]) i++;
            swap(nums[i], nums[j]);
        }
        swap(nums[i], nums[l]);
        quicksort(nums, l, i-1);
        quicksort(nums, i+1, r);
    }
};

```

## 方法二 快速选择

快排每次都能定下来基准数的位置，当基准数的索引是n-k时，第k大的数字就找到了。

```

class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        return quickchoose(nums, 0, nums.size()-1, k);
    }

    int quickchoose( vector<int>& nums,int l,int r,int k){
        //if(l>=r) return ;
        int i = l, j=r;
        while(i<j){
            while(i<j&&nums[j]>=nums[l]) j--;
            while(i<j&&nums[i]<=nums[l]) i++;
            swap(nums[i], nums[j]);
        }
        swap(nums[i], nums[l]);

        if(i>nums.size()-k){//基准索引在n-k右，对基准左排序
            return quicksort(nums, l, i-1,k); }
        else if(i<nums.size()-k){//基准索引在n-k左，对基准右排序
            return quicksort(nums, i+1, r,k); }
        else { // 基准索引为n-k 找到所求的值
            return nums[i];
        }
    }
};

```

## 方法三 优先队列（推荐）

此方法简单高效

维护一个大小为n-k的大顶堆就行。

```

class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int, vector<int>, less<>> pq; //大顶堆
        for(auto n:nums)pq.push(n); //往堆中放入所有的数
        for(int i=0;i<k-1;i++){//留下来n-k个，即pop掉k-1个
            pq.pop();
        }
        return pq.top();
    }
};

```

## 分治算法

### [241. 为运算表达式设计优先级](#)

给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 +, - 以及 \*。

```

输入: "2*3-4*5"
输出: [-34, -14, -10, -10, 10]
解释:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10

```

输入这样一个算式：

```
1 + 2 * 3 - 4 * 5
```

显然我们有四种加括号方式：

```

(1) + (2 * 3 - 4 * 5)
(1 + 2) * (3 - 4 * 5)
(1 + 2 * 3) - (4 * 5)
(1 + 2 * 3 - 4) * (5)

```

单独说上面的第三种情况：

```
(1 + 2 * 3) - (4 * 5)
```

我们用减号 - 作为分隔，把原算式分解成两个算式  $1 + 2 * 3$  和  $4 * 5$ 。

这一步就是把原问题进行了「分」，我们现在要开始「治」了。

$1 + 2 * 3$  可以有两种加括号的方式，分别是：

```

(1) + (2 * 3) = 7
(1 + 2) * (3) = 9

```

或者我们可以写成这种形式：

```
1 + 2 * 3 = [9, 7]
```

这就是典型的分治思路，先「分」后「治」，先按照运算符将原问题拆解成多个子问题，然后通过子问题的结果来合成原问题的结果。

```
class Solution {
public:
    vector<int> diffWaysToCompute(string expression) {
        vector<int> res;
        for(int i=0;i<expression.size();i++){
            char ch=expression[i];
            if(ch=='+' || ch=='-' || ch=='*'){
                auto left=diffWaysToCompute(expression.substr(0,i));
                auto right=diffWaysToCompute(expression.substr(i+1));
                for(int l:left){
                    for(int r:right){
                        switch(ch){
                            case '+':
                                res.push_back(l+r);
                                break;
                            case '-':
                                res.push_back(l-r);
                                break;
                            case '*':
                                res.push_back(l*r);
                                break;
                        }
                    }
                }
            }
            if(res.empty()){// res为空说明当前无运算符，只是单独的数字，直接放入
                res.push_back(stoi(expression));
            }
        }
        return res;
    }
};
```

## 高效进行模幂运算

### [372. 超级次方](#)

你的任务是计算  $a^b$  对 1337 取模， $a$  是一个正整数， $b$  是一个非常大的正整数且会以数组形式给出。

```
输入：a = 2, b = [1,0]
输出：1024
```

模运算在算法中比较常见： $** (a \cdot b) \% k = (a \% k)(b \% k) \% k **$

可以发现这样的一个规律：

$$\begin{aligned}
 & a^{[1,5,6,4]} \\
 & = a^4 \times a^{[1,5,6,0]} \\
 & = a^4 \times (a^{[1,5,6]})^{10}
 \end{aligned}$$

问题规模的缩小标准递归：

```

superPow(a, [1,5,6,4])
=> superPow(a, [1,5,6])

```

```

class Solution {
public:
    int base=1337;
    int mypow(int a,int n){
        int res=1;
        a%=base;
        for(int i=0;i<n;i++){
            res*=a;// 这里有乘法，是潜在的溢出点
            res%=base;
        }
        return res;
    }
    int superPow(int a, vector<int>& b) {
        if(b.empty()){
            return 1;
        }
        int p1=mypow(a, b.back());
        b.pop_back();
        int p2=superPow(a,b,10);
        return (p1*p2)%base;
    }
};

```

### 高效求幂 mypow优化

利用幂运算的性质，我们可以写出这样一个递归式：

$$a^b = \begin{cases} a \times a^{b-1}, & b \text{ 为奇数} \\ (a^{b/2})^2, & b \text{ 为偶数} \end{cases}$$

```

int mypow(int a, int k) {
    if (k == 0) return 1;
    a %= base;

    if (k % 2 == 1) {
        // k 是奇数
        return (a * mypow(a, k - 1)) % base;
    } else {
        // k 是偶数
        int sub = mypow(a, k / 2);
        return (sub * sub) % base;
    }
}

```

## 寻找缺失的元素

### [268. 丢失的数字](#)

给定一个包含  $[0, n]$  中  $n$  个数的数组  $\text{nums}$ ，找出  $[0, n]$  这个范围内没有出现在数组中的那个数。

#### 方法一 快排

```

class Solution {
public:
    int missingNumber(vector<int>& nums) {
        quicksort(0, nums.size() - 1, nums);
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == i + 1) return i;
        }
        return nums.size();
    }

    void quicksort(int l, int r, vector<int>& nums) {
        if (l >= r) return;
        int i = l, j = r;
        while (i < j) {
            while (i < j && nums[j] >= nums[l]) j--;
            while (i < j && nums[i] <= nums[l]) i++;
            swap(nums[i], nums[j]);
        }
        swap(nums[i], nums[l]);
        quicksort(l, i - 1, nums);
        quicksort(i + 1, r, nums);
    }
};

```

#### 方法二 set

```

class Solution {
public:
    int missingNumber(vector<int>& nums) {
        set<int> s;
        for (int n : nums) {
            s.insert(n);
        }

```

```

        for(int i=0;i<=nums.size();i++){
            if(!s.count(i)){
                return i;
            }
        }
        return -1;
    }
};

```

### 方法三 异或

```

class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int res=0;
        for(int n:nums){
            res^=n;
        }
        for(int i=0;i<=nums.size();i++){
            res^=i;
        }
        return res;
    }
};

```

### 方法四 等差数列求和

```

class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int n=nums.size();
        int s=(0+n)*(n+1)/2;
        int ss=0;
        for(int nn:nums){
            ss+=nn;
        }
        return s-ss;
    }
};

```

### 方法五 最优解 边加边减防溢出

```

class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int res=0;
        for(int i=0;i<nums.size();i++){
            res=res-nums[i]+i;
        }
        res+=nums.size();
        return res;
    }
};

```

# 寻找缺失和重复的元素

## 645. 错误的集合

集合 `s` 包含从 1 到 `n` 的整数。不幸的是，因为数据错误，导致集合里面某一个数字复制了成了集合里面的另外一个数字的值，导致集合 **丢失了一个数字 并且 有一个数字重复**。

给定一个数组 `nums` 代表了集合 `S` 发生错误后的结果。

请你找出重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。

看到**重复**和**缺失**立即反应：**异或**

```
class Solution {
public:
    vector<int> findErrorNums(vector<int>& nums) {
        // 首次异或得到 x^y
        int xoy=0;
        for(int n:nums){
            xoy^=n;
        }
        for(int i=1;i<=nums.size();i++){
            xoy^=i;
        }
        // 取最低位不同，用来分割nums和[1,n]
        int lastone=xoy& (~(xoy-1));
        // 第二次分组异或得到x和y
        int x=0, y=0;
        for(int n:nums){
            if(n&lastone){
                x^=n;
            }else{
                y^=n;
            }
        }
        for(int i=1;i<=nums.size();i++){
            if(i&lastone){
                x^=i;
            }else{
                y^=i;
            }
        }
        // 再次遍历nums去判断x是缺失还是重复
        for(int n:nums){
            if(n==x) return {x,y};
        }
        return {y,x};
    };
};
```

# 随机算法之水塘抽样算法

核心问题：当内存无法加载全部数据时，如何从包含未知大小的数据流中随机选取k个数据，并且要保证每个数据被抽取到的概率相等。

对于前  $k$  个数，全部保留，对于第  $i (i > k)$  个数，以  $\frac{k}{i}$  的概率保留第  $i$  个数，并以  $\frac{1}{k}$  的概率与前面已选择的  $k$  个数中的任意一个替换。

### 382. 链表随机节点

给定一个单链表，随机选择链表的一个节点，并返回相应的节点值。保证每个节点被选的概率一样。

进阶：

如果链表十分大且长度未知，如何解决这个问题？你能否使用常数级空间复杂度实现？

```
class Solution {
public:
    ListNode* mhead;
    Solution(ListNode* head) :mhead(head) {

}

/** Returns a random node's value. */
int getRandom() {
    int cnt = 1;
    ListNode* cur = mhead;
    int res = cur->val;
    while(cur){ //相当于遍历一遍链表
        if(rand()%cnt==0){ //第i个节点的值有1/i的概率被保留
            res = cur->val;
        }
        cur = cur->next;
        cnt++;
    }
    return res;
}
};
```

### 398. 随机数索引

给定一个可能含有重复元素的整数数组，要求随机输出给定的数字的索引。您可以假设给定的数字一定存在于数组中。

```
class Solution {
public:
    vector<int>  nums;
    int n;

    Solution(vector<int>& nums) :nums(nums),n(nums.size()){

}

int pick(int target) {
    int cnt = 1;
    int res = 0;
    for(int i=0;i<n;i++){
        if(nums[i]==target){
            if(rand()%cnt==0){
                res = i;
            }
        }
        cnt++;
    }
    return res;
}
};
```

```
        }
        cnt++;
    }
}
return res;
};

};
```

## 吃葡萄 几何法！

吃葡萄

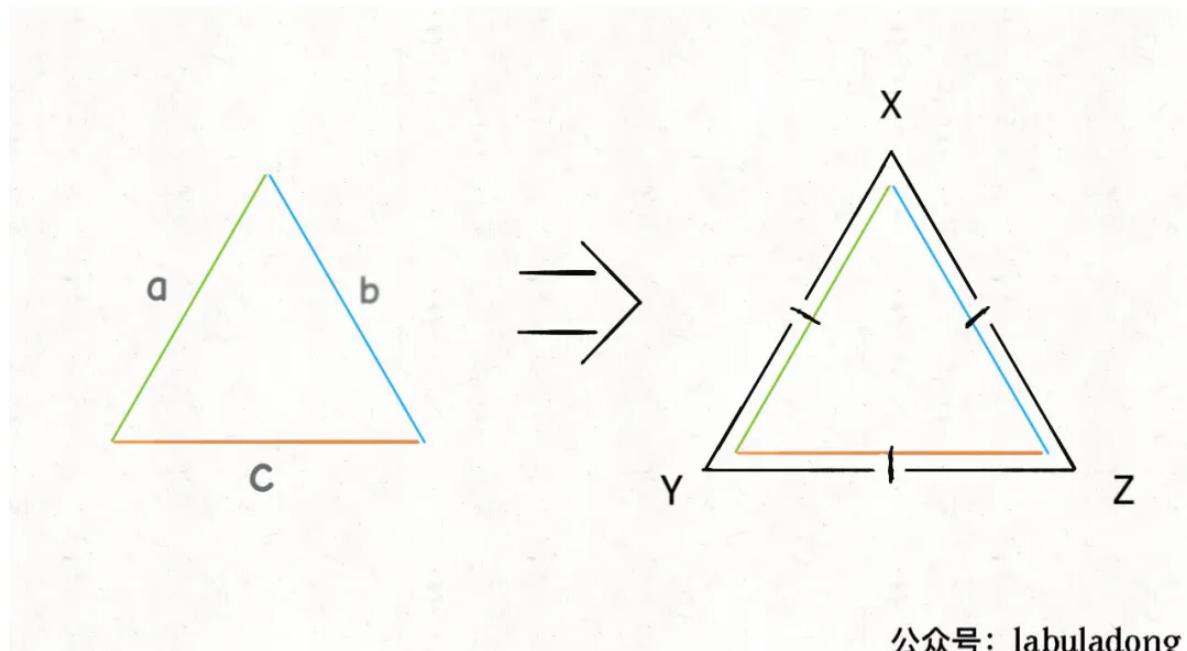
<https://www.nowcoder.com/questionTerminal/14c0359fb77a48319f0122ec175c9ada>

思路要回到如何「尽可能地平均分配」上面

如果把葡萄的颗数  $a, b, c$  作为三条线段，我们的目的可以转化成「尽可能平分这个几何图形的周长」

三角形是要满足两边之和大于第三边的，假设  $a < b < c$ ，那么有下面两种情况：

如果  $a + b > c$

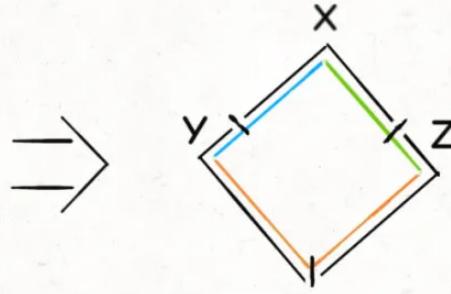


公众号：labuladong

如果  $a + b \leq c$ ，这三条边就不能组成一个封闭的图形了，那么我们可以将最长边  $c$  「折断」，也就是形成一个四边形。

这里面有两种情况：

### 情况一



### 情况二



公众号: labuladong

对于情况一， $a + b$  和  $c$  的差距还不大的时候，可以看到依然能够让三个人平分这个四边形，那么吃的最多的人最少可以吃到的葡萄颗数依然是  $(a+b+c+2)/3$ 。

随着  $c$  的不断增大，就会出现情况二，此时  $c > 2*(a+b)$ ，由于每人口味的限制， $X$  顶多吃完  $a$  和  $b$ ，为了尽可能平分， $c$  边需要被  $Y$  或  $Z$  平分，也就是说此时吃的最多的人最少可以吃到的葡萄颗数就是  $(c+1)/2$ ，即平分  $c$  边向上取整。

```
long Solution(long a, long b, long c){
    vector<long> num={a,b,c};
    long s=a+b+c;
    sort(num.begin(),num.end());
    if(num[0]+num[1]>num[2]){
        return (s+2)/3;
    }else if(2*(num[0]+num[1])<num[2]){
        return (num[2]+1)/2;
    }
    return (s+2)/3;
}

int main(){
    int n;
    cin>>n;
    long pt;
    vector<long> res;
    vector<long> putao;
    vector<vector<long>> putaos;
    for(int i=0;i<n;i++){
        for(int j=0;j<3;j++){
            cin>>pt;
            putao.push_back(pt);
        }
        putaos.push_back(putao);
        putao.clear();
    }

    for(int i=0;i<n;i++){
        auto num=putaos[i];
        res.push_back(Solution(num[0],num[1],num[2]));
    }
}
```

```

    }
    for(int i=0;i<n;i++){
        cout<<res[i]<<endl;
    }
    return 0;
}

```

## 煎饼（烧饼）排序

### 969. 煎饼排序

给你一个整数数组 arr，请使用煎饼翻转完成对数组的排序。

一次煎饼翻转的执行过程如下：

- 选择一个整数 k， $1 \leq k \leq arr.length$
- 反转子数组 arr[0...k-1] (下标从 0 开始)

例如，`arr = [3, 2, 1, 4]`，选择 `k = 3` 进行一次煎饼翻转，反转子数组 `[3, 2, 1]`，得到 `arr = [1, 2, 3, 4]`。

以数组形式返回能使 arr 有序的煎饼翻转操作所对应的 k 值序列。任何将数组排序且翻转次数在  $10 * arr.length$  范围内的有效答案都将被判断为正确。

递归思路：

- 1、找到 n 个饼中最大的那个。
- 2、把这个最大的饼移到最底下。
- 3、递归调用 `reversecake(arr[], n - 1)`。

base case：`n == 1` 时，排序 1 个饼时不需要翻转。

如何设法将某块烧饼翻到最后呢？

其实很简单，比如第 3 块饼是最大的，我们想把它换到最后，也就是换到第 n 块。可以这样操作：

- 1、用锅铲将前 3 块饼翻转一下，这样最大的饼就翻到了最上面。
- 2、用锅铲将前 n 块饼全部翻转，这样最大的饼就翻到了第 n 块，也就是最后一块。

```

class Solution {
public:
    vector<int> res;
    vector<int> pancakeSort(vector<int>& arr) {
        res.clear();
        reversecake(arr, arr.size());
        return res;
    }
    void reversecake(vector<int>& arr, int n){
        if(n==1) return ;
        int max=-1;
        int midx=-1;
        for(int i=0;i<n;i++){// 寻找最大饼的索引
            if(arr[i]>max){
                max=arr[i];
                midx=i;
            }
        }
        flip(arr, midx);
        flip(arr, n-1);
    }
    void flip(vector<int>& arr, int n){
        int left=0;
        int right=n-1;
        while(left<right){
            swap(arr[left], arr[right]);
            left++;
            right--;
        }
    }
};

```

```

        }
    }
    res.push_back(midx+1);
    reverse(arr.begin(), arr.begin()+midx+1); // 第一次翻将最大饼翻到最上面
    res.push_back(n);
    reverse(arr.begin(), arr.begin()+n); // 第二次翻将最大饼翻到最下面
    reversecake(arr, n-1); //对前n-1排序
}
};


```

## 字符串（大数）相加、相乘

### [415. 字符串相加](#)

给定两个字符串形式的非负整数 `num1` 和 `num2`，计算它们的和。

统一在指针当前下标处于负数的时候返回 00，等价于**对位数较短的数字进行了补零操作**，这样就可以除去两个数位数不同情况的处理

```

class Solution {
public:
    string addStrings(string num1, string num2) {
        int i=num1.size()-1, j=num2.size()-1;
        int add=0; //进位
        string res="";
        while(i>=0 || j>=0 || add!=0){
            int n1=i>=0?num1[i]-'0':0;
            int n2=j>=0?num2[j]-'0':0;
            res.push_back((n1+n2+add)%10+'0'); //记得+进位
            add=(n1+n2+add)/10;
            i--;j--;
        }
        reverse(res.begin(),res.end());
        return res;
    }
};

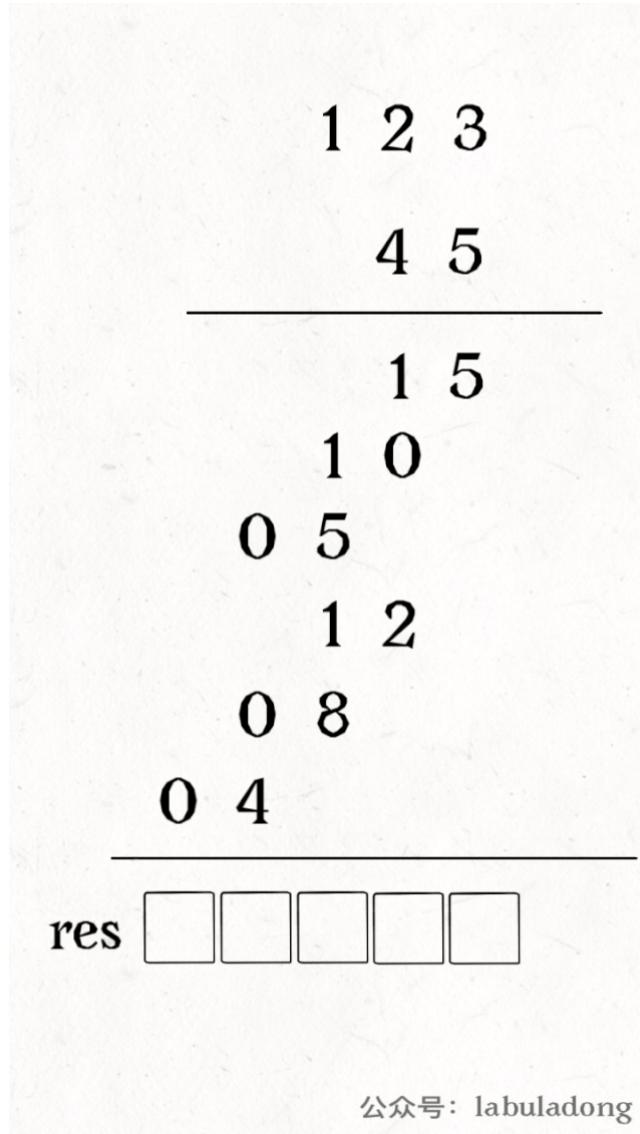
```

### [43. 字符串相乘](#)

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

竖式相加法

**有两个指针 `i`, `j` 在 `num1` 和 `num2` 上游走，计算乘积，同时将乘积叠加到 `res` 的正确位置：**



公众号: labuladong

`** num1[i] 和 num2[j] 的乘积对应的就是 res[i+j] 和 res[i+j+1] 这两个位置**。`

```

class Solution {
public:
    string multiply(string num1, string num2) {
        string res="";
        vector<int> nums(num1.size()+num2.size(),0);
        for(int i=num1.size()-1;i>=0;i--){
            for(int j=num2.size()-1;j>=0;j--){
                int n1=num1[i]-'0',n2=num2[j]-'0';
                int mul=n1*n2;
                int sum = mul+nums[i+j+1]+nums[i+j]*10;//重新计算这两位放的数
                nums[i+j+1]=sum%10;//个位
                nums[i+j]=sum/10;//十位
            }
        }
        int i =0;
        while(i<nums.size()&&nums[i]==0)i++;//num其一为0, 结果就会是一排0
        for(;i<nums.size();i++){
            res.push_back(nums[i]+'0');
        }
        return res==""?":res;
    }
};

```

# 万能计算器

秒杀基本计算器I、II、III

表达式字符串只包含非负整数，算符 +、-、\*、/，左括号 ( 和右括号 ) 和若干空格。整数除法需要 向下截断。

单栈法

```
class Solution {
public:
    int i=0;// 全局变量 一次遍历 记录遍历位置
    int calculate(string s) {
        int res=0;
        stack<int> sta;
        char sign='+';// 默认符号是+ 简化处理
        int num=0;
        for(;i<s.size();i++){
            char c=s[i];
            if(isdigit(c)){//#include<cstdio> isdigit()
                num=10*num+(c-'0');// 如果是数字就累计计算
            }
            if(c=='('){// 递归开始 迈过'(' 这样不会无限递归
                i++;//迈过'('
                num = calculate(s);
            }
            if((!isdigit(c)&&c!=' ')||i==s.size()-1){// 处理 非数字和非空格即符号 或者 因为长度到达而滞留的数字
                int pre=0;
                switch(sign){
                    case '+':
                        sta.push(num);
                        break;
                    case '-':
                        sta.push(-num);
                        break;
                    case '*':
                        pre=sta.top();
                        sta.pop();
                        sta.push(pre*num);// 当场计算了 *的结果
                        break;
                    case '/':
                        pre=sta.top();
                        sta.pop();
                        sta.push(pre/num);// 当场计算了/ 的结果
                        break;
                }
                sign=c;// 更新符号
                num=0;// 清空当前数字
            }
            // 是跳出循环的关键 不能像'('一样先迈过 而是需要回到上层 让上层处理
            // 上层在处理'('时进入的递归 因为pos是全局变量 再向后循环一步就迈过了 ')'
            if(c==')')break;
        }
    }
}
```

```

        while(!sta.empty()){
            int n=sta.top();
            sta.pop();
            res+=n;
        }
        return res;
    }
};

```

## 有效的括号

### 20. 有效的括号

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 s，判断字符串是否有效。

栈是一种先进后出的数据结构，处理括号问题的时候尤其有用。

```

bool isValid(string s) {
    stack<char> sta;
    for(int i=0;i<s.size();i++){
        if(!sta.empty()){
            if(
                (s[i]=='(')&&sta.top()=='(') || // 栈顶是左括号
                (s[i]==')'&&sta.top()=='(') || // 当前字符是相应右括号
                (s[i]==']'&&sta.top()=='[')
            ){
                // 则两者相消
                sta.pop();
                continue;
            }
        }
        sta.push(s[i]);
    }
    return sta.empty(); // 如果括号合法栈为空 没削干净则不合法
}

```

## 判断子序列 二分法

### 392. 判断子序列

给定字符串 s 和 t，判断 s 是否为 t 的子序列。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，“ace”是“abcde”的一个子序列，而“aec”不是）。

进阶：

如果有大量输入的 S，称作 S1, S2, … , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？

不考虑进阶 直接用双指针：

```

class Solution {
public:
    bool isSubsequence(string s, string t) {
        int i=0, j=0;
        while(i<s.size() && j < t.size()){
            if(s[i]==t[j]) i++;
            j++;
        }
        return i==s.size();
    }
};

```

此解法处理每个 `s` 时间复杂度仍然是  $O(N)$ ，而如果巧妙运用二分查找，可以将时间复杂度降低，大约是  $O(M \log N)$ ， $M$  为 `s` 的长度。由于  $N$  相对  $M$  大很多，所以后者效率会更高。

### 二分思路：

二分思路主要是对 `t` 进行预处理，用一个字典 `m` 将每个字符出现的索引位置按顺序存储下来

0	1	2	3	4	5	6	
<code>t</code>	c	a	c	b	h	b	c

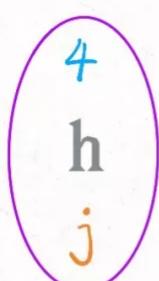
index:

<code>a</code> :	[1]
<code>b</code> :	[3, 5]
<code>c</code> :	[0, 2, 6]
<code>h</code> :	[4]

公众号: labuladong

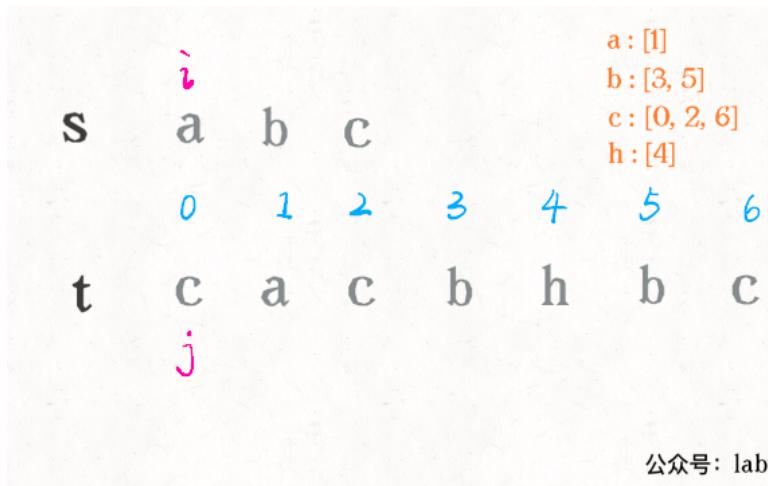
按照之前的解法，我们需要 `j` 线性前进扫描字符 "c"。但现在借助 `index` 中记录的信息，**可以二分搜索 `index[c]` 中比 `j` 大的那个索引**，在上图的例子中，就是在 `[0, 2, 6]` 中搜索比 4 大的那个索引：

<code>s</code>	<code>a</code>	<code>b</code>	<code>c</code>	<code>i</code>	<code>c : [0, 2, 6]</code>	
<code>t</code>	<code>c</code>	a	c	<code>4</code>	5	6



公众号: labuladong

大致流程：



公众号: labuladong

```

class Solution {
public:
    bool isSubsequence(string s, string t) {
        unordered_map<char, vector<int>> m;
        for(int i=0;i<t.size();i++){
            m[t[i]].push_back(i);
        }
        int j=0;//t上指针
        for(int i=0;i<s.size();i++){
            char c=s[i];
            if(m.count(c)==0) return false;//字典中没该字符
            int pos=leftbound(m[c],j); //返回索引
            if(pos==m[c].size()) return false;//j右边没有该字符了
            //j = pos+1;错误
            j = m[c][pos]+1;
        }
        return true;
    }

    int leftbound(vector<int> v,int t){ //>=
        int l=0,r=v.size()-1;
        while(l<=r){
            int m = l+(r-l)/2;
            if(v[m]<t){
                l=m+1;
            }else if(v[m]>=t){
                r=m-1;
            }
        }
        return l;
    }
};

```

## 考试座位

[855. 考场就座](#)

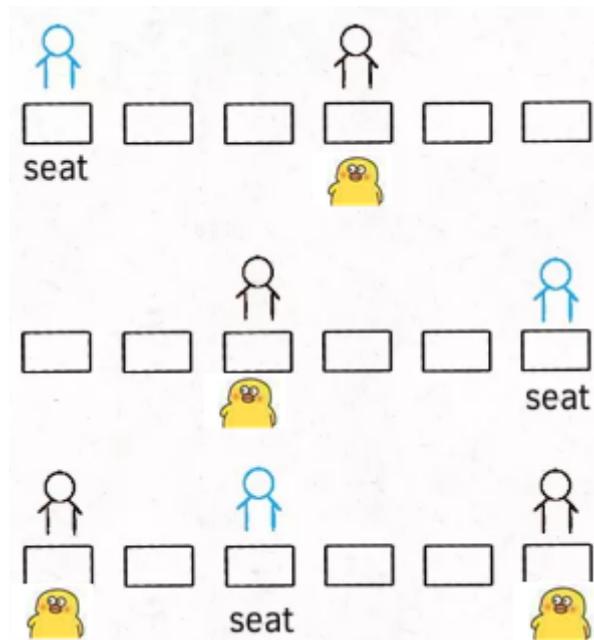
在考场里，一排有  $N$  个座位，分别编号为  $0, 1, 2, \dots, N-1$ 。

当学生进入考场后，他必须坐在能够使他与离他最近的人之间的距离达到最大化的座位上。如果有多个这样的座位，他会坐在编号最小的座位上。(另外，如果考场里没有人，那么学生就坐在  $0$  号座位上。)

返回 `ExamRoom(int N)` 类，它有两个公开的函数：其中，函数 `ExamRoom.seat()` 会返回一个 `int` (整型数据)，代表学生坐的位置；函数 `ExamRoom.leave(int p)` 代表坐在座位  $p$  上的学生现在离开了考场。每次调用 `ExamRoom.leave(p)` 时都保证有学生坐在座位  $p$  上。

但凡遇到在动态过程中取最值的要求，肯定要使用有序数据结构，我们常用的数据结构就是二叉堆 (priority\_queue) 和平衡二叉搜索树 (set) 了。

将每两个相邻的考生看做线段的两端点，新安排考生就是找最长的线段，然后让该考生在中间把这个线段「二分」，中点就是给他分配的座位。`leave(p)` 其实就是去除端点  $p$ ，使得相邻两个线段合并为一个。



情况1：放最左边

情况2：放两个中间

情况3：放最右边

```
class ExamRoom {
public:
    set<int> s;
    int n;
    ExamRoom(int n):n(n) {
        s.clear();
    }

    int seat() {
        if(s.empty()){//考场空 放0
            s.insert(0);
            return 0;
        }
        //从最左边开始判断 包含情况一、二
        //然后依次计算二者间隔距离，找到最大距离并计算相应的pos
        int pos=0,pre=-1,maxDist=0;
```

```

        for(int cur:s){
            int dist = (cur-pre)/2;
            if(dist>maxDist){
                //pos=cur+maxDist;//如果看的是s的第一个数pos就肯定不是放在 -1和cur的中间
                pos = pre == -1 ? 0 : pre + dist;//而是放最左边
                //maxDist =dist;//如果看的是s的第一个数就maxDist肯定不是(cur-(-1))/2
                maxDist= pre== -1? cur:dist;//而是cur 因为pos==0
            }
            pre=cur;
        }
        //还剩下一种 放在最右边的情况 看看是不是更大
        if((n-1)-pre>maxDist)pos=n-1;
        s.insert(pos);
        return pos;
    }

    void leave(int p) {
        s.erase(p);
    }
};

//seat()这么写可能好理解一点:
int seat() {
    if(s.empty()){//考场空 放0
        s.insert(0);
        return 0;
    }
    int pos=0,pre=-1,maxDist=0;
    bool first=true;
    for(int cur:s){
        if(first){//新来的直接放在最左边的情况 情况一
            //pos=0;
            pre=cur;
            maxDist=cur;
            first=false;
            continue; //跳过 计算后面两两之间的情况 情况二
        }
        int dist = (cur-pre)/2;
        if(dist>maxDist){
            pos=pre+dist;//如果看的是s的第一个数pos就肯定不是放在 -1和cur的中间
            //pos = pre == -1 ? 0 : pre + dist;//而是放最左边
            maxDist =dist;//如果看的是s的第一个数就maxDist肯定不是(cur-(-1))/2
            //maxDist= pre== -1? cur:dist;//而是cur 因为pos==0
        }
        pre=cur;
    }
    //还剩下一种 新来的放在最右边的情况 看看是不是更大 情况三
    if((n-1)-pre>maxDist)pos=n-1;
    s.insert(pos);
    return pos;
}

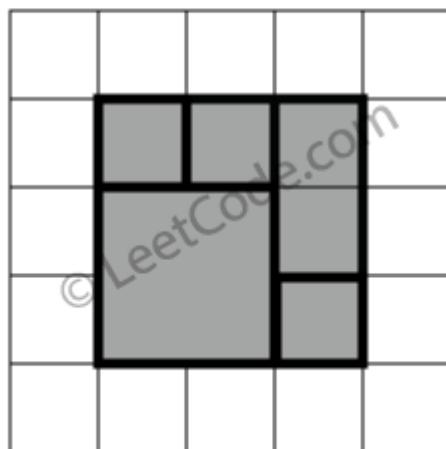
```

# 完美矩形

## 391. 完美矩形

我们有  $N$  个与坐标轴对齐的矩形，其中  $N > 0$ ，判断它们是否能精确地覆盖一个矩形区域。

每个矩形用左下角的点和右上角的点的坐标来表示。例如，一个单位正方形可以表示为  $[1,1,2,2]$ 。（左下角的点的坐标为  $(1, 1)$  以及右上角的点的坐标为  $(2, 2)$ ）。



### 解题思路

- 1,总面积等于小块的面积和
- 2,除大矩形4个顶点外其它点都出现偶数(2或4)次

//基于红黑树(RB-Tree)的set与map是可以使用std::pair的，而unoredered\_set与unordered\_map的内部实现是基于哈希表(HashTable)，并没有直接提供pair接口

```
//括号中的const表示参数b对象不会被修改，最后的const表明函数所调用的函数对象不会被修改！
//字符重载也是个函数，在函数末尾加CONST 这样的函数叫常成员函数。常成员函数可以理解为是一个“只读”函数，它既不能更改数据成员的值，也不能调用那些可能引起数据成员值变化的成员函数，它只能调用const成员函数。
struct pairhash {
    size_t operator()(const pair<int,int> & p) const {
        return p.first*10 + p.second;
    }
};
class Solution {
public:

    bool isRectangleCover(vector<vector<int>>& rectangles) {
        int left=INT_MAX,right=INT_MIN;
        int bottom=INT_MAX,top=INT_MIN;
        int s=0;
        //unordered_map<pair<int,int>,int> m; //直接用map也行
        unordered_map<pair<int,int>,int,pairhash> m; //用于保存每个顶点数量
        for(vector<int>& a:rectangles){
            left = min(left,a[0]); //找最大矩形
            right= max(right,a[2]);
            bottom=min(bottom,a[1]);
            top = max(top,a[3]);
            s+=(a[2]-a[0])*(a[3]-a[1]);
            m[{a[0],a[1]}]++;
        }
        if(s==m.size())
            return true;
        else
            return false;
    }
};
```

```

        m[{a[2],a[3]}]++;
        m[{a[0],a[3]}]++;
        m[{a[2],a[1]}]++;
    }
    if(s != (right-left)*(top-bottom))return false;
    //如果是完美矩形此时除了4角都有应该是偶数
    m[{left,bottom}]++;
    m[{left,top}]++;
    m[{right,bottom}]++;
    m[{right,top}]++;
    //把大矩形有4个角放入后,所有点都应该是偶数了
    for(auto it=m.begin();it != m.end(); it++)if((*it).second %2 ==1) return
false;
    return true;
}
};

```

## 接雨水

### [42. 接雨水](#)

```

//双指针
class Solution {
public:
    int trap(vector<int>& height) {
        int n=height.size();
        int l=0 ,r=n-1;
        int lmax=0,rmax=0;
        int res=0;
        while(l<r){
            if(height[l]<=height[r]){//左边比右边小 结果增量取决于 左最大-左当前
                if(height[l]>lmax){
                    lmax=height[l];
                }
                res+=lmax-height[l];
                l++;
            }else{//右边比左边小 结果增量取决于 右最大-右当前
                if(height[r]>rmax){
                    rmax=height[r];
                }
                res+=rmax-height[r];
                r--;
            }
        }
        return res;
    }
};

//暴力法 + 备忘录优化
class Solution {
public:
    int trap(vector<int>& height) {
        if(height.empty())return 0;
        int n=height.size();
        vector<int> lmax(n),rmax(n); //记得初始化 (n)
        lmax[0]=height[0],rmax[n-1]=height[n-1];

```

```

        for(int i=1;i<n;i++){
            lmax[i]=max(lmax[i-1],height[i]);
        }
        for(int i=n-2;i>=0;i--){
            rmax[i]=max(rmax[i+1],height[i]);
        }
        int res=0;
        for(int i=1;i<n-1;i++){
            res+=min(lmax[i],rmax[i])-height[i];
        }
        return res;
    }
}

```

## 括号匹配 带\*

### [678. 有效的括号字符串](#)

```

//贪心
bool checkValidString(string s) {
    int l=0,r=0;//l、r表示 左括号至少几个、至多几个
    for(auto c:s){
        if(c=='('){
            l++;r++;
        }else if(c==')'){
            if(l>0)l--;
            if(r--==0) return false;
        }else{
            if(l>0)l--;
            r++;
        }
    }
    return l==0;
}

//双stack
class Solution {
public:
    stack<int> s1,s2;
    //一栈存左括号，一栈存星号 遍历过程中，同时判断是否有足够的右括号使他们出栈
    //优先抵消左括
    bool checkValidString(string s) {
        for(int i=0;i<s.size();i++){
            char c=s[i];
            if(c=='(')s1.push(i);
            else if(c=='*')s2.push(i); //存的是idx
            else if(c==')'){
                if(!s1.empty()){
                    s1.pop();
                    continue;
                }else if(!s2.empty()){
                    s2.pop();
                }else return false;
            }
        }
    }
}

```

```

        while(!s1.empty()&&!s2.empty()){
            if(s1.top()>s2.top())return false;
            s1.pop();
            s2.pop();
        }
        if(!s1.empty())return false;
        return true;
    };
}

```

## 二分法求sqrt ()

利用二分法求一个数的平方根,精度要求  $e < 10^{-6}$

```

#include <math.h>
double sqrt(double x )
{
    double l = 0, r = x;
    while (1) {
        double mid = l + (r - l) / 2;
        if (fabs(x / mid - mid) < 1e-6) return mid;
        else if (x / mid > mid) left = mid + 1;
        else right = mid - 1;
    }
}

```

## 二叉树的右视图

[199. 二叉树的右视图](#)

```

//bfs
vector<int> rightSideView(TreeNode* root) {
    if(!root) return {};
    queue<TreeNode*> q;
    vector<int> res;
    q.push(root);

    while(!q.empty()){
        int sz=q.size();
        for(int i=0;i<sz;i++){
            auto f=q.front();
            q.pop();
            if(i==sz-1)res.push_back(f->val);
            if(f->left)q.push(f->left);
            if(f->right)q.push(f->right);
        }
    }
    return res;
}

```

## 48. 旋转图像

### [48. 旋转图像](#)

```
void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    // 水平翻转
    for (int i = 0; i < n / 2; ++i) {
        for (int j = 0; j < n; ++j) {
            swap(matrix[i][j], matrix[n - i - 1][j]);
        }
    }
    // 主对角线翻转
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
}
```

## 岛屿数量

### [200. 岛屿数量](#)

```
//dfs
void dfs(vector<vector<char>>& g, int r, int c) {
    int nr = g.size();
    int nc = g[0].size();
    if (r < 0 || r >= nr || c < 0 || c >= nc || g[r][c] == '0') return;
    g[r][c] = '0';
    dfs(g, r+1, c);
    dfs(g, r, c+1);
    dfs(g, r-1, c);
    dfs(g, r, c-1);
}
int numIslands(vector<vector<char>>& grid) {
    int nr = grid.size(), nc = grid[0].size();
    if (!nr) return 0;
    int res = 0;
    for (int r = 0; r < nr; ++r) {
        for (int c = 0; c < nc; ++c) {
            if (grid[r][c] == '1') {
                ++res;
                dfs(grid, r, c);
            }
        }
    }
    return res;
}
//bfs
int numIslands(vector<vector<char>>& grid) {
    int nr = grid.size();
    if (!nr) return 0;
    int nc = grid[0].size();
    int res = 0;
    for (int r = 0; r < nr; ++r) {
```

```

        for (int c = 0; c < nc; ++c) {
            if (grid[r][c] == '1') {
                ++res;
                grid[r][c] = '0';
                queue<pair<int, int>> neighbors;
                neighbors.push({r, c});
                while (!neighbors.empty()) {
                    auto rc = neighbors.front();
                    neighbors.pop();
                    int row = rc.first, col = rc.second;
                    if (row - 1 >= 0 && grid[row-1][col] == '1') {
                        neighbors.push({row-1, col});
                        grid[row-1][col] = '0';
                    }
                    if (row + 1 < nr && grid[row+1][col] == '1') {
                        neighbors.push({row+1, col});
                        grid[row+1][col] = '0';
                    }
                    if (col - 1 >= 0 && grid[row][col-1] == '1') {
                        neighbors.push({row, col-1});
                        grid[row][col-1] = '0';
                    }
                    if (col + 1 < nc && grid[row][col+1] == '1') {
                        neighbors.push({row, col+1});
                        grid[row][col+1] = '0';
                    }
                }
            }
        }

        return res;
    }
}

```

//并查集

```

class UnionFind {
public:
    UnionFind(vector<vector<char>>& grid) {
        count = 0;
        int m = grid.size();
        int n = grid[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '1') {
                    parent.push_back(i * n + j);
                    ++count;
                }
                else {
                    parent.push_back(-1);
                }
                rank.push_back(0);
            }
        }
        int find(int i) {
            if (parent[i] != i) {

```

```

        parent[i] = find(parent[i]);
    }
    return parent[i];
}

void unite(int x, int y) {
    int rootx = find(x);
    int rooty = find(y);
    if (rootx != rooty) {
        if (rank[rootx] < rank[rooty]) {
            swap(rootx, rooty);
        }
        parent[rooty] = rootx;
        if (rank[rootx] == rank[rooty]) rank[rootx] += 1;
        --count;
    }
}

int getCount() const {
    return count;
}

private:
    vector<int> parent;
    vector<int> rank;
    int count;
};

class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int nr = grid.size();
        if (!nr) return 0;
        int nc = grid[0].size();
        UnionFind uf(grid);
        int num_islands = 0;
        for (int r = 0; r < nr; ++r) {
            for (int c = 0; c < nc; ++c) {
                if (grid[r][c] == '1') {
                    grid[r][c] = '0';
                    if (r - 1 >= 0 && grid[r-1][c] == '1') uf.unite(r * nc + c,
(r-1) * nc + c);
                    if (r + 1 < nr && grid[r+1][c] == '1') uf.unite(r * nc + c,
(r+1) * nc + c);
                    if (c - 1 >= 0 && grid[r][c-1] == '1') uf.unite(r * nc + c,
r * nc + c - 1);
                    if (c + 1 < nc && grid[r][c+1] == '1') uf.unite(r * nc + c,
r * nc + c + 1);
                }
            }
        }
        return uf.getCount();
    }
};

```

## 二叉树的锯齿形层序遍历

### [103. 二叉树的锯齿形层序遍历](#)

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    if(!root) return {};
    queue<TreeNode*> dq;
    dq.push(root);
    vector<vector<int>> res;
    int n=1;
    while(!dq.empty()){
        int sz=dq.size();
        deque<int> v;
        for(int i=0;i<sz;i++){
            auto f=dq.front();
            dq.pop();
            if(n%2==1){
                v.push_back(f->val);
            }else if(n%2==0){
                v.push_front(f->val);
            }
            if(f->left)dq.push(f->left);
            if(f->right)dq.push(f->right);
        }
        res.emplace_back(vector<int>{v.begin(), v.end()});
        n++;
    }
    return res;
```

## 搜索旋转排序数组

### [33. 搜索旋转排序数组](#)

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

# 重排链表

## 143. 重排链表

```
void reorderList(ListNode* head) {
    if (head == nullptr) {
        return;
    }
    ListNode* mid = middleNode(head);
    ListNode* l1 = head;
    ListNode* l2 = mid->next;
    mid->next = nullptr;
    l2 = reverseList(l2);
    mergeList(l1, l2);
}

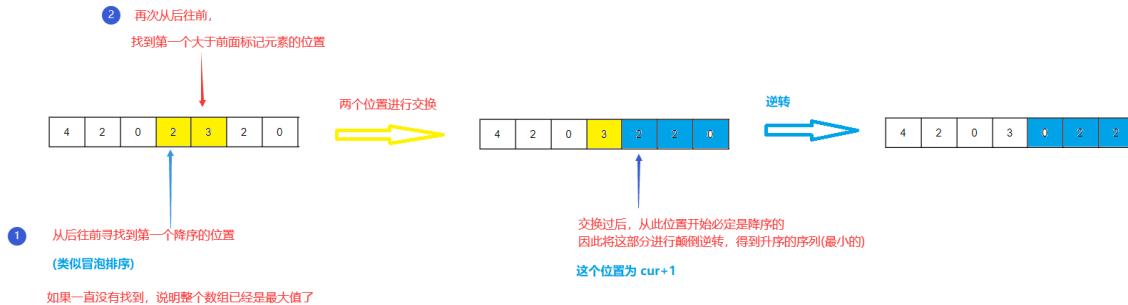
ListNode* middleNode(ListNode* head) {//找到原链表的中点
    ListNode* slow = head;
    ListNode* fast = head;
    while (fast->next != nullptr && fast->next->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}

ListNode* reverseList(ListNode* head) {//将原链表的右半端反转
    ListNode* prev = nullptr;
    ListNode* curr = head;
    while (curr != nullptr) {
        ListNode* nextTemp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}

void mergeList(ListNode* l1, ListNode* l2) {//将原链表的两端合并
    ListNode* l1next;
    ListNode* l2next;
    while (l1 != nullptr && l2 != nullptr) {
        l1next = l1->next;
        l2next = l2->next;
        l1->next = l2;
        l1 = l1next;
        l2->next = l1;
        l2 = l2next;
    }
}
```

# 下一个排列

## 31. 下一个排列



```
void nextPermutation(vector<int>& nums) {  
    int cur=nums.size()-2;  
    while(cur>=0&&nums[cur]>=nums[cur+1]) cur--;  
    if(cur<0) sort(nums.begin(),nums.end());  
    else {  
        int pos=nums.size()-1;  
        while(nums[pos]<=nums[cur]) pos--;  
        swap(nums[pos],nums[cur]);  
        reverse(nums.begin()+cur+1, nums.end());  
    }  
}
```

## 删除排序链表中的重复元素 II

```
ListNode* deleteDuplicates(ListNode* head) {  
    if (!head || !head->next) return head;  
    if (head->val != head->next->val) {  
        head->next = deleteDuplicates(head->next);  
    } else {  
        ListNode* move = head->next;  
        while (move && head->val == move->val) {  
            move = move->next;  
        }  
        return deleteDuplicates(move);  
    }  
    return head;  
}
```

## 二叉树的完全性检验

```
bool isCompleteTree(TreeNode* root) {  
    queue<TreeNode*> q;  
    q.push(root);  
    bool reachnull=false;  
    while(!q.empty()){  
        int sz=q.size();  
        for(int i=0;i<sz;i++){  
            auto f=q.front();  
            q.pop();  
            if(!f){  
                reachnull=1;
```

```

        continue;
    }else{
        if(reachnull) return false; //null结点后又出现空结点
        q.push(f->left);
        q.push(f->right);
    }
}
return true;
}

```

## 合并K个升序链表

```

ListNode* mergeKLists(vector<ListNode*>& lists) {
    struct cmp{
        bool operator()(ListNode* a,ListNode* b){
            return a->val>b->val;
        }
    };
    priority_queue<ListNode*, vector<ListNode*>, cmp> p;
    for(auto l:lists){
        if(l)p.push(l);
    }
    ListNode* d=new ListNode(-1);
    auto res=d;
    while(!p.empty()){
        auto top=p.top();
        p.pop();
        d->next=top;
        d=d->next;
        if(top->next)p.push(top->next);
    }
    return res->next;
}

```

## 求根节点到叶节点数字之和

```

int sumNumbers(TreeNode* root) {
    return sum(root,0);
}
int sum(TreeNode* root,int n){ //前序+回溯
    if(!root) return 0;
    int t=10*n+root->val;
    if (!root->left&& !root->right)
        return t;
    int l=sum(root->left,t);
    int r=sum(root->right,t);
    return l+r;
}

```

## 用 Rand7() 实现 Rand10()

(rand\_X() - 1) \* Y + rand\_Y() ==> 可以等概率的生成[1, X \* Y]范围的随机数,即实现了 rand\_XY()

```
int rand10() {
    int n=(rand7()-1)*7+rand7();
    while(n>40){
        n=(rand7()-1)*7+rand7();
    }
    return (n-1)%10+1;
}
```

## 缺失的第一个正数

```
int firstMissingPositive(vector<int>& nums) {
    int n=nums.size();
    bool have1=0;
    for(auto& a:nums){
        if(a==1)have1=1;
        else if(a<1)a=n+1;
    }
    if(!have1) return 1;
    for(int i=0;i<nums.size();i++){
        int num=abs(nums[i]);
        if(num<=n)nums[num-1]=-abs(nums[num-1]);
    }
    for(int i=0;i<nums.size();i++){
        if(nums[i]>0) return i+1;
    }
    return n+1;
}
```

## 排序链表

```
ListNode* sortList(ListNode* head) {
    if(!head || !head->next) return head;
    ListNode* pre = head, *slow = head, *fast = head;
    while(fast && fast->next) {
        pre = slow;
        slow = slow->next;
        fast = fast->next->next;
    }
    pre->next = nullptr;
    return merge(sortList(head), sortList(slow));
}
ListNode* merge(ListNode* h1, ListNode* h2) {
    if(!h1) return h2;
    if(!h2) return h1;
    if(h1->val < h2->val) {
        h1->next = merge(h1->next, h2);
        return h1;
    }
    else {
        h2->next = merge(h1, h2->next);
    }
}
```

```
        return h2;
    }
}
```

## 堆排序

```
void down(vector<int> &v, int parent, int n){
    int tmp = v[parent];//临时保存要下沉的元素
    int child = parent * 2 + 1;//定位左孩子节点的位置
    while (child <= n){//开始下沉
        if (child + 1 <= n&&v[child] < v[child + 1]){//从第一个非叶子节点开始调整
            child++; // 如果右孩子节点比左孩子大，则定位到右孩子
        }
        if (v[child] <= tmp)break;// 如果孩子节点小于或等于父节点，则下沉结束
        v[parent] = v[child];// 父节点进行下沉
        parent = child;
        child = 2 * parent + 1;
    }
    v[parent] = tmp;
}

void heap(vector<int> &v){
    int n = v.size();
    for (int i = (n - 2) / 2; i >= 0; i--){
        down(v, i, n-1);//构建大顶堆
    }
    for (int i = n - 1; i >= 1; i--){
        swap(v[i], v[0]);// 把堆顶元素与最后一个元素交换
        down(v, 0, i - 1);// 把打乱的堆进行调整，恢复堆的特性
    }
}
```

## 冒泡排序

```
void bubble(vector<int>& v){
    int sz = v.size();
    for (int i = 0; i < sz - 1; i++){
        for (int j = 0; j <sz-1-i; j++){
            if (v[j]>v[j+1]){
                std::swap(v[j], v[j + 1]);
            }
        }
    }
}
```

## 选择排序

```

void select(vector<int>& v){
    int n = v.size();
    for (int i = 0; i < n - 1; i++){
        int min = i;
        for (int j = i + 1; j < n ; j++){
            if (v[j] < v[min]){
                min = j;
            }
        }
        swap(v[i], v[min]);
    }
}

```

## C++哈希表的实现

```

#include <iostream>
#include <vector>

using namespace std;

template <typename K, typename V>
class HashNode
{
public:
    HashNode(const K& key = K(), const V& value = V())
    {
        key_ = key;
        value_ = value;
    }

    K key_;
    V value_;
    bool initialized_ = false;
};

template <typename K, typename V>
class HashTable
{
public:
    HashTable(int size)
    {
        node_num_ = 0;
        nodes_.resize(size);
    }

    int hashFunction(const K& key)
    {
        return hash_index_ % hash(key);
    }

    V find(const K& key)
    {
        int index = hashFunction(key);
        while(nodes_[index].initialized_)
        {

```

```
    if(nodes_[index].key_ == key)
    {
        return nodes_[index].value_;
    }
    index++;
    if(index == node_num_)
    {
        index = 0;
    }
}

void insert(const K& key, const V& value)
{
    /* 当哈希表的长度不够用时，应该进行扩容，设立一个扩容长度表 */
    int index = hashFunction(key);
    while(nodes_[index].initialized_)
    {
        index++;
        if(index == node_num_)
        {
            index = 0;
        }
    }

    nodes_[index].key_ = key;
    nodes_[index].value_ = value;
    nodes_[index].initialized_ = true;
}

int hash(const K& key)
{
    return key;
}

private:
    std::vector<HashNode<K, V>> nodes_;
    int node_num_;
    int hash_index_;
};

int main(int argc, char** argv)
{
    HashTable<int, int> table(10);
    table.insert(1, 8);
    table.insert(2, 10);
    table.insert(3, 18);
    table.insert(4, 5);
    cout<<table.find(3)<<endl;

    return 0;
}
```

## c++简单String类实现

```
#include <iostream>
#include <string>
using namespace std;

class String
{
public:
    String(const char* str = NULL); //通用构造函数, string("abc")
    String(const String &str); //拷贝构造
    ~String();

    String& operator=(const String &str); //赋值运算符。返回引用
    String operator+(const String &str) const;
    String& operator+=(const String &str); //+=操作符。返回引用
    char& operator[](int n) const; //下标操作符。返回引用
    bool operator==(const String &str) const;

    int size() const; //字符串实际大小，不包括结束符
    const char *c_str() const; //将string转为char *

private:
    char *data;
    int length;
};

String::String(const char* str) //通用构造
{
    if (!str)
        { //为空。String a()
            length = 0;
            data = new char[1];
            *data = '\0';
        }
    else
    {
        length = strlen(str);
        data = new char[length + 1];
        strcpy(data, str); //会拷贝源的结束符
    }
}

String::String(const String &str) //拷贝构造，深拷贝
{
    length = str.size();
    data = new char[length + 1];
    strcpy(data, str.c_str());
}

String::~String()
{
    delete[] data;
    length = 0;
}
```

```
String& String::operator=(const String &str)//赋值操作符4步
{
    if (this == &str) return *this;//1 自我赋值，返回自身引用

    delete[] data;//2 删除原有数据

    length = str.size();//3 深拷贝
    data = new char[length + 1];
    strcpy(data, str.c_str());

    return *this;//4 返回自身引用
}

String String::operator+(const String &str) const//+操作符3步
{//新建对象包括新空间，拷贝两个数据，返回新空间
    String newString;
    newString.length = length + str.size();
    newString.data = new char[newString.length + 1];
    strcpy(newString.data, data);
    strcat(newString.data, str.data);
    return newString;
}

String& String::operator+=(const String &str)//+=操作符5步
{//重分配新空间，拷贝两个数据，删除自己原空间，赋值为新空间，返回引用
    length += str.size(); //成员length是实际长度
    char *newdata = new char[length + 1];
    strcpy(newdata, data);
    strcat(newdata, str.c_str());
    delete[] data;
    data = newdata;
    return *this;
}

char& String::operator[](int n) const
{//下标操作符，返回引用
    if (n >= length) return data[length - 1];//如果越界，返回最后一个字符
    else return data[n];
}

bool String::operator==(const String &str) const
{
    if (length != str.size()) return false;
    return strcmp(data, str.c_str()) ? false : true;
}

int String::size() const
{
    return length;
}

const char *String::c_str() const
{
    return data;
}

int main()
{
    char a[] = "Hello", b[] = "world!";
}
```

```

String s1(a), s2(b);
cout << s1.c_str() << endl;
cout << s2.c_str() << endl;
s1 += s2;
cout << s1.c_str() << endl;
s1 = s2;
cout << s1.c_str() << endl;
cout << (s1 + s2).c_str() << endl;
cout << s1.size() << endl;
cout << s1[1] << endl;

if (s1 == s2)
    cout << "相等" << endl;
}

```

## C++ vector 实现

```

template<class T>
class vector {
public://-----迭代器-----
    typedef vector<T> self;
    typedef T* iterator;
    typedef const T* const_iterator;
iterator begin() const {
    return _start;
}
iterator end() const {
    return _finish;
}

public://-----init-----
vector()
:_start(nullptr)
, _finish(nullptr)
, _endofstorage(nullptr)
{}
vector(int n, const T& data)
:_start(new T[n])
{
    for (int i = 0;i < n;i++) {
        _start[i] = data;
    }
    _finish = _endofstorage = _start + n;
}
template<class Iterator>
vector(Iterator frist, Iterator last) {
    size_t count = 0;
    auto it = frist;
    while (it != last) {
        it++;
        count++;
    }
    _start = new T[count];

    for (size_t i = 0;i < count;i++) {
        _start[i] = *frist++;
    }
}

```

```

        _finish = _endofstorage = _start + count;
    }
    vector(const self& v) {
        _start = _finish = new T[v.size()];
        _endofstorage = _start + v.size();
        iterator it = v.begin();
        while (_finish != _endofstorage) {
            *_finish = *it;
            _finish++;
            it++;
        }
    }
    ~vector() {
        clear();
    }
public://-----重载-----
T& operator[](int i) {
    return *(_finish + i);
}
//const T& operator[](int i) {
//    return *(_finish + i);
//}
//public://-----function-----
size_t size()const {
    return _finish - _start;
}
size_t capacity()const {
    return _endofstorage - _start;
}
bool empty() {
    return _start == _finish;
}
void reserve(size_t n) {
    size_t oldsize = size();
    if (n <= capacity()) {
        return;
    }
    T* p = new T[n];
    memcpy(p, _start, sizeof(T)*size());
    delete[] _start;
    _start = p;
    _finish = p + oldsize;
    _endofstorage = _start + n;
}
void resize(size_t newsize, const T& data) {
    reserve(newsize);
    auto it = newsize - size() + _finish;
    for (;_finish < it;_finish++) {
        *_finish = data;
    }
}
iterator insert(iterator pos, const T& data) {
    if (_finish == _endofstorage) {
        reserve(capacity() * 2);
    }
    auto it = _finish();
    while (it != pos) {

```

```

        *it = *(it - 1);
        it--;
    }
    *it = data;
    ++_finish;
    return pos;
}
void push_back(const T& data) {
    if (_finish == _endofstorage) {
        reserve(size() * 2);
    }
    *_finish = data;
    _finish++;
}
void pop_back() {
    _finish--;
}
T front() {
    return *_start;
}
T back() {
    return *(_finish - 1);
}
void clear() {
    if (_start == nullptr) {
        return;
    }
    delete[] _start;
    _start = _endofstorage = _finish = nullptr;
}

public://-----
-----
private:
    T* _start;
    T* _finish;
    T* _endofstorage;
};

```

## 字符串转换整数 (atoi)

[字符串转换整数 \(atoi\)](#) 和剑指offer那题一样

```

int myAtoi(string str) {
    if(str.size()==0) return 0;
    int l=0;
    while(str[l]==' ') {
        l++;
        if(l>=str.size()) return 0;
    }
    int firstchar=1;
    if(str[l]=='-'){
        firstchar=-1;
    }
    if(str[l]=='-' || str[l]=='+'){
        l++;
    }
    ...
}

```

```

int res=0;
int max=INT_MAX/10;
for(int i=1;i<str.size();i++){
    auto c=str[i];
    if(c>'9'||c<'0'){
        break;
    }
    if(res>max||(res==max&&c>'7')){
        return firstchar==-'1'?INT_MIN:INT_MAX;
    }
    res=res*10+(c-'0');
}
return firstchar*res;
}

```

## 寻找旋转排序数组中的最小值

### [寻找旋转排序数组中的最小值](#)

```

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

```

## 寻找两个正序数组的中位数

### [4. 寻找两个正序数组的中位数](#)

#### 解析

把2个数组看做一个虚拟的数组A，A有 $2m+2n+2$ 个元素，割在 $m+n+1$ 处，所以我们只需找到 $m+n+1$ 位置的元素和 $m+n+2$ 位置的元素就行了。

左边： $A[m+n+1] = \text{Max}(LMax1, LMax2)$  / 右边： $A[m+n+2] = \text{Min}(RMin1, RMin2)$

为了效率，我们肯定是选长度较短的做二分，假设为 $c1$ 。

```

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    int n = nums1.size(), m = nums2.size();
    if(n>m) return findMedianSortedArrays(nums2, nums1); //保证数组1一定最短
    // c1 为第i个数组的割, 比如c1为2时表示第1个数组只有2个元素
    int lmax1, lmax2, rmin1, rmin2, c1, c2, lo = 0, hi = 2 * n; //虚拟加了'#'所以nums1是 $2*n$ 长度
    while (lo <= hi) //二分
    {
        c1 = (lo + hi) / 2; //c1是二分的结果
        c2 = m + n - c1;
        lmax1 = (c1 == 0) ? INT_MIN : nums1[(c1 - 1) / 2];
        rmin1 =
    }
}

```

```

        rmin1 = (c1 == 2 * n) ? INT_MAX : nums1[c1 / 2];
        lmax2 = (c2 == 0) ? INT_MIN : nums2[(c2 - 1) / 2];
        rmin2 = (c2 == 2 * m) ? INT_MAX : nums2[c2 / 2];
        if (lmax1 > rmin2) hi = c1 - 1;
        else if (lmax2 > rmin1) lo = c1 + 1;
        else break;
    }
    return (max(lmax1, lmax2) + min(rmin1, rmin2)) / 2.0;
}

```

## 二叉搜索树的第k大节点

[剑指 Offer 54. 二叉搜索树的第k大节点](#)

```

int res;
int kthLargest(TreeNode* root, int k) {
    kth(root, k);
    return res;
}
void kth(TreeNode* root, int& k){
    if(!root) return;
    kth(root->right, k);
    k--;
    if(k==0) res=root->val;
    kth(root->left, k);
}

```

## 翻转字符串里的单词

[151. 翻转字符串里的单词](#)

```

//一
string reverseWords(string s) {
    if(s=="")return "";
    string res="";
    int l=0,r =s.size()-1;
    while(s[r]==' '){
        if(s[r]==' ')r--;//右边界 去空格
        if(r<0)break;
    }
    int i=r ,j=r;
    while(i>=0){
        while(i>=0&&s[i]!=' ')i--;
        res.append(s.substr(i+1,j-i));
        res.push_back(' ');
        while(i>=0&&s[i]==' ')i--;
        j=i;
    }
    res.pop_back();
    return res;
}
//二
string reverseWords(string s) {
    auto q = split(s);
    reverse(q.begin(),q.end());
    string res="";
    for(auto it=q.begin();it!=q.end();it++)
        res+=*it+" ";
}
```

```

        for(auto st:q){
            if(st!="")res.append(st).append(" ");
        }
        res.pop_back();
        return res;
    }

    vector<string> split(string s){
        int idx=0;
        vector<string> q;
        for(int i=0;i<=s.size();i++){
            if(i==s.size()||s[i]==' '){
                q.push_back(s.substr(idx,i-idx));
                idx=i+1;
            }
        }
        return q;
    }
}

```

## 路径总和 II

[路径总和 II](#)

```

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    dfs(root, targetSum);
    return res;
}

vector<vector<int>> res;
vector<int> v;
void dfs(TreeNode* root,int sum){
    if(!root) return;
    v.push_back(root->val);
    if(sum==root->val&&!root->left&&!root->right){
        res.push_back(v);
    }
    dfs(root->left, sum-root->val);
    dfs(root->right,sum-root->val);
    v.pop_back();
}

```

## 无重复字符的最长子串

[无重复字符的最长子串](#)

```

int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> m;
    int l=0,r=0;
    int res=0;
    while(r<s.size()){
        auto c=s[r];
        r++;
        m[c]++;
        while(m[c]>1){
            auto d=s[l];
            l++;
            m[d]--;
        }
        res=max(res,r-l);
    }
    return res;
}

```

```

        }
        res=max(res,r-1);
    }
    return res;
}

```

## 颜色分类

### [75. 颜色分类](#)

```

//一
void sortColors(vector<int>& nums) {
    int p1=0,p0=0;
    for(int i=0;i<nums.size();i++){
        int n=nums[i];
        nums[i]=2;
        if(n<2){
            nums[p1++]=1;
        }
        if(n<1){
            nums[p0++]=0;
        }
    }
}

//二
void sortColors(vector<int>& nums) {
    int n=nums.size();
    if(n<2) return;
    int zero=0,i=0,two=n;
    while(i<two){
        if(nums[i]==0){
            swap(nums[i], nums[zero]);
            i++;
            zero++;
        }else if(nums[i]==1){
            i++;
        }else{
            two--;
            swap(nums[i], nums[two]);
        }
    }
}

```

## 字典序的第K小数字

### [字典序的第K小数字](#)

```

int getnodes(int n,long cur){
    long total=0;
    long next=cur+1;
    while(cur<=n){
        total+=min(n-cur+1,next-cur);
        cur*=10;next*=10;
    }
    return total;
}

```

```

}
int findkthNumber(int n, int k) {
    long cur=1;
    k--;
    while(k>0){
        int nodes=getnodes(n,cur);
        if(nodes<=k){
            cur++;
            k-=nodes;
        }else{
            cur*=10;
            k--;
        }
    }
    return cur;
}

```

## 二叉搜索树与双向链表

[二叉搜索树与双向链表](#)

```

Node* treeToDoublyList(Node* root) {
    if(!root) return nullptr;
    change(root);
    head->left=pre;
    pre->right=head;
    return head;
}
Node* head = nullptr;
Node* pre = nullptr;
void change(Node* root){
    if(!root) return;

    change(root->left);

    if(!pre)head=root;
    else pre->right=root;
    root->left=pre;
    pre=root;

    change(root->right);
    return ;
}

```

## 两两交换链表中的节点

[两两交换链表中的节点](#)

```

ListNode* swapPairs(ListNode* head) {
    if(!head||!head->next) return head;
    auto newhead=head->next;
    head->next=swapPairs(newhead->next);
    newhead->next=head;
    return newhead;
}

```

## 二叉树最大宽度

### [二叉树最大宽度](#)

```
int widthOfBinaryTree(TreeNode* root) {
    int res=0;
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});
    while(!q.empty()){
        int sz=q.size();
        int start=q.front().second, end;
        for(int i=0;i<sz;i++){
            auto f=q.front();
            q.pop();
            end=f.second;
            if(f.first->left)q.push({f.first->left, end*2-2*start});
            if(f.first->right)q.push({f.first->right, end*2+1-2*start});
        }
        res=max(res, end-start+1);
    }
    return res;
}
```

## 旋转链表

### [旋转链表](#)

```
ListNode* rotateRight(ListNode* head, int k) {
    if(!head||!head->next||k==0) return head;
    auto p=head;
    int n=1;
    while(p->next){
        n++;
        p=p->next;
    }
    int K=k%n;
    if(!K) return head;
    p->next=head;
    p=head;
    for(int i=0;i<n-K-1;i++){
        p=p->next;
    }
    auto res=p->next;
    p->next=nullptr;
    return res;
}
```

## 旋转数组

### [189. 旋转数组](#)

```

//原始数组 1 2 3 4 5 6 7
//翻转所有元素      7 6 5 4 3 2 1
//翻转[0,k mod n-1] 区间的元素 5 6 7 4 3 2 1
//翻转[k mod n,n-1] 区间的元素 5 6 7 1 2 3 4
void rotate(vector<int>& nums, int k) {
    k %= nums.size();
    reverse(nums.begin(),nums.end());
    reverse(nums.begin(),nums.begin()+k);
    reverse(nums.begin()+k, nums.end());
}

```

## 乘积最大子数组

### [乘积最大子数组](#)

```

int maxProduct(vector<int>& nums) {
    int dpmax=nums[0],dpmin=nums[0],dp=nums[0];
    for(int i=1;i<nums.size();i++){
        int t=dpmax;
        dpmax=max(max(dpmax*nums[i],dpmin*nums[i]),nums[i]);
        dpmin=min(min(t*nums[i],dpmin*nums[i]),nums[i]);
        dp=max(dp,dpmax);
    }
    return dp;
}

```

## 圆圈中最后剩下的数字

### [圆圈中最后剩下的数字](#)

```

int lastRemaining(int n, int m) {
    int pos=0;
    for(int i=2;i<=n;i++){
        pos = (pos+m)%i;
    }
    return pos;
}

```

## 寻找峰值

### [162. 寻找峰值](#)

```

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

```

## 螺旋矩阵 II

### [59. 螺旋矩阵 II](#)

```
vector<vector<int>> generateMatrix(int n) {
    vector<vector<int>> matrix(n, vector<int>(n));
    int u = 0;// 上下左右
    int d = n-1;
    int l = 0;
    int r = n-1;
    int num = 1;
    while(true){
        for(int i=l; i <= r; i++) matrix[u][i] = num++;
        if (++u > d) break;
        for(int i=u; i <= d; i++) matrix[i][r] = num++;
        if (--r < l) break;
        for(int i=r; i >= l; i--) matrix[d][i] = num++;
        if (--d < u) break;
        for(int i=d; i >= u; i--) matrix[i][l] = num++;
        if (++l > r) break;
    }
    return matrix;
}
```

## Pow(x, n)

### [50. Pow\(x, n\)](#)

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

## 最小栈

### [最小栈](#)

```
stack<int> s1,s2;
MinStack() {}
void push(int val) {
    s1.push(val);
    if(s2.empty()) s2.push(val);
    else{
```

```

        if(val<=s2.top())
            s2.push(val);
    }
}

void pop() {
    if(!s2.empty()&&s2.top()==s1.top())s2.pop();
    s1.pop();
}

```

## 环形链表 II

### [142. 环形链表 II](#)

```

ListNode *detectCycle(ListNode *head) {
    ListNode* f=head,*s=head;
    while(1){
        if(!f||!f->next) return nullptr;
        s=s->next;
        f=f->next->next;
        if(s==f)break;
    }
    f=head;
    while(s!=f){
        s=s->next;
        f=f->next;
    }
    return s;
}

```

## x 的平方根

### [x 的平方根](#)

计算并返回  $x$  的平方根，其中  $x$  是非负整数。

```

int mySqrt(int x) {
    if(x==1) return 1;
    int l = 0, r = x ,res=-1;
    while (l<=r) {
        double mid = l + (r - l) / 2;
        if ((long long)mid*mid<=x) {
            res=mid;
            l = mid + 1;
        }
        else r = mid - 1;
    }
    return res;
}

```

## 最大数

### [最大数](#)

```

string largestNumber(vector<int>& nums) {
    string s="";

```

```

        sort(nums.begin(), nums.end(), [] (int const a, int const b) {
            string sa=to_string(a);
            string sb=to_string(b);
            if(sa+sb>sb+sa) return true;
            return false;
        });

        if(nums[0]==0) return "0";
        for(int n:nums){
            s+=to_string(n);
        }
        return s;
    }
}

```

## 数组中重复的数据

### [442. 数组中重复的数据](#)

```

//使用 x->x-1x->x-1 的映射并采用置负的方式标记
vector<int> findDuplicates(vector<int>& nums) {
    vector<int> res;
    for(int i = 0; i < nums.size(); ++i){
        if(nums[abs(nums[i])-1] < 0) res.push_back(abs(nums[i]));
        else nums[abs(nums[i])-1] *= -1;
    }
    return res;
}

```

## 整数与IP地址间的转换

### [整数与IP地址间的转换](#)

```

int main()
{
    unsigned int a, b, c, d;
    char ch;
    while (cin >> a >> ch >> b >> ch >> c >> ch >> d)
    {
        cout << ((a << 24) | (b << 16) | (c << 8) | d) << endl;
        cin >> a;
        cout << ((a & 0xff000000) >> 24) << "." << ((a & 0x00ff0000) >> 16) <<
        "." << ((a & 0x0000ff00) >> 8) << "." << (a & 0x000000ff) << endl;
    }
    return 0;
}

```

## 多数元素

### [169. 多数元素](#)

```

int majorityElement(vector<int>& nums) {
    int candidate=0, vote=0;
    for(int n:nums){
        if(!vote)candidate=n;
        vote += candidate==n?1:-1;
    }
    return candidate;
}

```

## 回文数

### [9. 回文数](#)

```

bool isPalindrome(int x) {
    //当 x < 0 时, x 不是回文数。
    //最后一位是0, 其第一位数字也应该是0 只有 0 满足
    if (x < 0 || (x % 10 == 0 && x != 0)) return false;
    int re = 0;
    while (x > re) {
        re = re * 10 + x % 10;
        x /= 10;
    }
    // 当数字长度为奇数, 通过 revertedNumber/10 去除中位数字。
    // 当输入为 12321 时, 在 while 循环的末尾我们可以得到 x = 12, re = 123
    return x == re || x == re / 10;
}

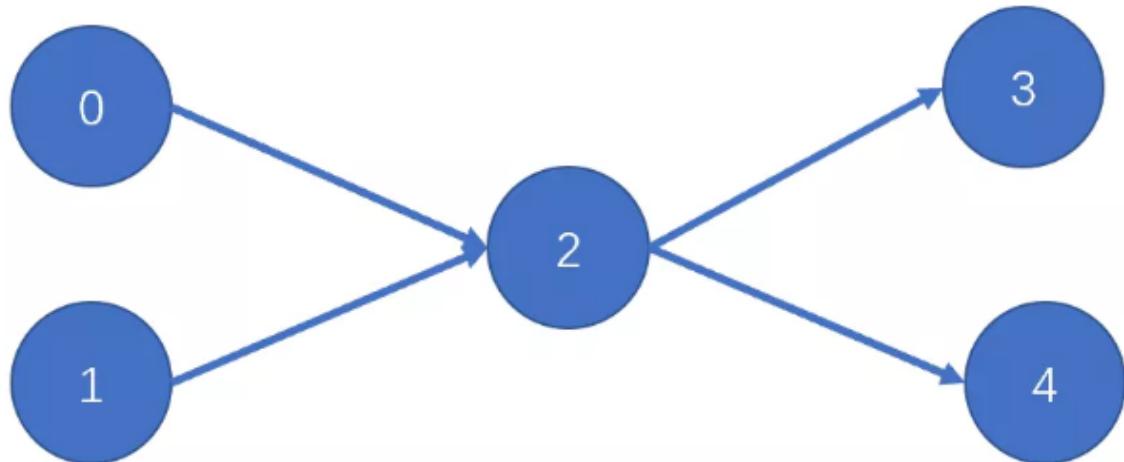
```

## 循环依赖检测

循环依赖检测。如，[['A', 'B'], ['B', 'C'], ['C', 'D'], ['B', 'D']] => false，[['A', 'B'], ['B', 'C'], ['C', 'A']] => true

拓扑排序算法过程：

1. 选择图中一个入度为0的点，记录下来
2. 在图中删除该点和所有以它为起点的边
3. 重复1和2，直到图为空或没有入度为0的点。



```

g = [[2]      #表示0->2
      [2]      #表示1->2
      [3, 4]   #表示2->3, 2->4
      []       #表示没有以3为起点的边
      []]     #表示没有以4为起点的边

```

```

vector<int> haveCircularDependency(int n, vector<vector<int>> &prerequisites) {
    vector<vector<int>> g(n); //邻接表存储图结构
    vector<int> indeg(n); //每个点的入度
    vector<int> res; //存储结果序列
    for(int i = 0; i < prerequisites.size(); i++) {
        int a = prerequisites[i][0], b = prerequisites[i][1];
        g[a].push_back(b);
        indeg[b]++;
    }
    queue<int> q;
    //一次性将入度为0的点全部入队
    for(int i = 0; i < n; i++) {
        if(indeg[i] == 0) q.push(i);
    }
    while(q.size()) {
        int t = q.front();
        q.pop();
        res.push_back(t);
        //删除边时，将终点的入度-1。若入度为0，果断入队
        for(int i = 0; i < g[t].size(); i++) {
            int j = g[t][i];
            indeg[j]--;
            if(indeg[j] == 0) {
                q.push(j);
            }
        }
    }
    if(res.size() == n) return res;
    else return {};
}

```

## 验证回文串

### [125. 验证回文串](#)

```

bool isPalindrome(string s) {
    string newStr;
    for (auto ch : s)
    {
        if(isdigit(ch)){//数字
            newStr.push_back(ch);
        }
        else if (isalpha(ch)) //大小写字母
        {
            ch |= ' '; //等价于ch =tolower(ch) 大小写都变成小写
            newStr.push_back(ch);
        }
    }
    string temp = newStr;
    reverse(newStr.begin(), newStr.end());

```

```

        if (newStr == temp)  return true;
        return false;
    }
}

```

## 搜索二维矩阵

### [74. 搜索二维矩阵](#)

```

bool searchMatrix(vector<vector<int>> matrix, int target) {
    auto row = upper_bound(matrix.begin(), matrix.end(), target, [] (const
int tar, const vector<int> &a) {
        return tar < a[0];
}); //upper_bound 从matrix迭代器中找到第一个比target大的a[0]
//参数中的lambda表达式 因为matrixb.begin()指向的还是一个vector,比较每行第一个数字用
a[0]
    if (row == matrix.begin()) {
        return false;
    } // 这表明 target比第一行开头的数还小 直接返回false
    --row; //返回上一行
    return binary_search(row->begin(), row->end(), target); //STL二分法
}

```

## 旋转数组的最小数字

### [剑指 Offer 11. 旋转数组的最小数字](#)

```

int minArray(vector<int>& numbers) {
    int l=0,r=numbers.size()-1;
    while(l<=r){
        int m=l+(r-l)/2;
        if(numbers[m]>numbers[r])l=m+1;
        else if(numbers[m]<numbers[r])r=m;
        else r--;//由于重复元素的存在，不能确定numbers[m]在最小值左侧还是右侧
        //唯一可知它们值相同，所以可以忽略右端点
    }
    return numbers[l];
}

```

## 整数反转

### [7. 整数反转](#)

```

int reverse(int x) {
    int res=0;
    while(x!=0){
        int tmp=x%10;
        x/=10;
        if(res>INT_MAX/10 || (res==INT_MAX&&tmp>7))return 0;
        if(res<INT_MIN/10 || (res==INT_MIN&&tmp<-8))return 0;
        res=res*10+tmp;
    }
    return res;
}

```

## 重复的子字符串

### [459. 重复的子字符串](#)

```
//将两个s连在一起，并移除第一个和最后一个字符  
//如果s是该字符串的子串，那么s就满足题目要求。  
bool repeatedSubstringPattern(string s) {  
    //从位置1开始查询，并希望查询结果不为位置n，这与移除字符串的第一个和最后一个字符是等价的。  
    return (s + s).find(s, 1) != s.size();  
}
```