一．数组

 # 704 二分查找 时间复杂度 O(log n)　循环 while(left<=right)中更新 left，right 和 mid。注意边界条件，左闭右闭边界减一加一，

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int l=0, r=nums.size()-1;
        int mid=0;
        while(l<=r){ 为什么是<=? 如果只有一个元素的话，也能执行循环了
            mid = l + (r-l)/2;
            if(nums[mid]<target){
                l = mid+1;
            }
            else if(nums[mid]>target){
                r = mid-1;
            }
            else{
                return mid;
            }
        }
        return -1;
    }
};
35
```

唯一不同点是 return left;

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int temp;
        int left = 0, right = nums.size()-1;
        int mid = (left + right) /2;
        if(nums[mid] == target){
            return mid;
        }else if(nums[mid]>target){
            temp = 0;//往左找
            right = mid-1;
        }else{
            temp = 1;
            left = mid+1;
        }

        while(left<=right){
            int mid = (left + right) /2;
            if(nums[mid] == target){
                return mid;
            }else if(nums[mid]>target){
                if(temp == 1){
                    return mid-1;
                }
                right = mid-1;
            }else{
                if(temp == 0){
                    return mid+1;
                }
                left = mid+1;
            }
        }
```

```
    }
};
```

 # 27. Remove Element 移除元素（删）用快慢指针
快慢指针（不用真的定义 ListNode，而是双指针的思想，定义的是 i 和 j 就好，都从下标 index=0 的位置开始）

```
# Solution 2, double pointer 双指针法
# 快指针快速往下走，过一遍，看哪些值是要的（除了要删的值之外），慢指针慢慢把要的值保存下来（下标在走，然后存值）
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int slowIndex=0, n=nums.size();
        for(int fastIndex=0; fastIndex<n; fastIndex++){
            if(nums[fastIndex]!=val){ #要
                nums[slowIndex] = nums[fastIndex]; //赋值，就赋在 nums 里了
                slowIndex++; //下标在走
            }
        }
        return slowIndex; #返回数组长度
    }
};

# Solution 1, double loop
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int n=nums.size();
        for(int i=0;i<n;i++){
            if(nums[i]==val){
                for(int j=i;j<n-1;j++){
                    nums[j]=nums[j+1];
                }
                i--; /*非常重要的一句话！i 位置是 val，后面的全都提前一位覆盖，这没错。但是！如果覆盖后的 i 位置还是 val，那就检测不到了（因为 i++，已经往下走了）*/
                n--;
            }
        }
        return n;
    }
};
```

 # 977. Squares of a Sorted Array 有序数组的平方 （含负数）
双指针，谁大谁先被存进去

```
# 注意 i,j,k 的值代表啥，注意初始值，使用的时候，怎么改变它
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        int n = nums.size();
        vector<int> result(n, 0); //长度为 n，初始化值为 0，vector 当数组用
        int i = 0, j = n-1; //两端
        int k = n-1; //从最大值开始存

        while(i<=j){//循环终止条件，
            if(nums[i]*nums[i]>nums[j]*nums[j]){
```

```
            result[k]=nums[i]*nums[i];
            i++;
        }
        else{
            result[k]=nums[j]*nums[j];
            j--;
        }
        k--;
    }
    return result;
  }
};
```

这两道题总结：双指针可以做到，在全局比较中，选择要的元素存入新数组

209. 长度最小的子数组 滑动窗口

数值加起来不超过 target 的子数组，的元素个数
滑动窗口用一个 for 解决两个 for 能做到的事，O(n2)变 O(n)
如何？就是不断的调节子序列的起始位置 i 和终止位置 j，从而得出我们要想的结果。先确定终止位置 j，确定 sum>target，再动态移动起始位置 i，找最小长度。
???? 还是不是很理解，start = i,

```
class Solution {
public:
  int minSubArrayLen(int target, vector<int>& nums) {
    int i = 0, n=nums.size(), result = INT32_MAX, sum=0;
# result = INT32_MAX 是为了应付你 case3 测试用例情况：输入：target = 11, nums = [1,1,1,1,1,1,1,1,1] 输出：0（此时 result=8 小于 target，完全没有进入 while 循环，所以返回的不能是 result 初始值，而应该是 0。总之 result 初始值设置为无穷大，只是为了记录是否进入过循环（所有数值加起来都超不过 target），若无，则返回 0 即可
    for (int j=0;j<n;j++){
      sum += nums[j];       /*怎么求 sum*/
      while(sum>=target){  //sum 大于 target 了，再确定起始位置 i
        int start = i, end = j;
        int temp = j-i+1;   //因为这道题只需要知道长度大小，不需要给出具体 nums[i]
        result = result > temp ? temp : result;
        sum -=nums[i];
        i++;
      }
    }
    return result == INT32_MAX ? 0 : result;
  }
};
```

58. 区间和　转换为前缀和

暴力搜索会超时，时间复杂度是 O(n * m)，m 是查询次数。

前缀和的思想是重复利用计算过的子数组之和，从而降低区间查询需要累加计算的次数。
前缀和 在涉及计算区间和的问题时非常有用！

下标 2 到下标 5 之间的累加和，就用 p[5] - p[1] 就可以了。

0059.螺旋矩阵 II

传统的二维数组如 int arr[n][n] 的大小必须在编译时确定。使用 vector 可以在运行时定义数组的大小，可变大小，动态分配内存和释放内存 new/delete。可使用 push_back、resize 等函数。

vector<vector<int>> res(n, vector<int>(m, 0)); 会创建一个 n 行 m 列的二维数组，初始值为 0。

你首先要知道赋值几圈？答 n/2 圈
每一圈就是 4 个 for 循环，分别赋值四条边，注意每条边赋值都是左闭又开，这样才不会打架（重复赋值）。只用 offset 一个变量即可（代码随想录里面有 startx 和 starty 两个变量，只是便于理解，不是必需品
最后，如果 n 是奇数，那么中间元素还需要赋值为 n 的平方

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        vector<vector<int>> res(n,vector<int>(n,0));
        int count=1, loop=n/2;
        int offset=0;
        while(loop--){
            int i=0,j=0;
            for(j=offset;j<n-1-offset;j++){
                res[i+offset][j]=count++;
            }
            for(i=offset;i<n-1-offset;i++){
                res[i][j]=count++;
            }
            for(j=n-offset-1;j>offset;j--){
                res[n-1-offset][j]=count++;
            }
            for(i=n-offset-1;i>offset;i--){
                res[i][offset]=count++;
            }
            offset++;
// 这里有一行空格，runtime 就是 0ms；没有就是 2ms。奇怪
        }
        if(n%2==1){res[n/2][n/2]=n*n;}
        return res;
    }
};
```

二・链表
链表会走路，指针可以走，cur=cur->next
如果要创建一个节点空间，才 ListNode* cur= new ListNode(0); //分配空间
如果要删除第 n 个或者倒数第 n 个节点，善用 n 到达你想去的位置。while(n--)
虚拟头节点巧妙一致化，不用再单独考虑头节点。
ListNode* dummyHead = new ListNode(0);
dummyHead->next = head;

链表：203.移除链表元素 是给值 val，不是给下标 index（也是从 0 开始）

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        ListNode* dummyhead= new ListNode(0); //分配空间
        ListNode* cur = dummyhead;
        dummyhead->next = head;
        while(cur->next != NULL){
            if(cur->next->val == val){
                cur->next = cur->next->next; #但运行时间 1ms，解决办法: 用一个 tmp 放要删掉的
节点，然后 delete 掉它，释放内存空间
                ListNode* tmp = cur->next;
                cur->next = cur->next->next;
                delete tmp;
            }else{ # 不是所有情况都执行 cur = cur->next; 一定要写在 else 里面！！！！
                cur = cur->next;
            }
        }
        return dummyhead->next;//返回头节点
    }
};
```

707. 设计链表
注意：cur=dummyhead，而不是 cur=dummyhead->next，原因？？？？

```
class MyLinkedList {
private:
    int _size;
    LinkedNode* _dummyHead;

public:
    // 定义链表节点结构体
    struct LinkedNode {
        int val;
        LinkedNode* next;
        LinkedNode(int val):val(val), next(nullptr){}
    };
```

```
    // 初始化链表 有一个虚拟头节点：简化操作逻辑。它的存在使得所有链表操作可以统一处理，而不需要单独处理头结点的特殊情况。
    MyLinkedList() {
        _dummyHead = new LinkedNode(0); // 这里定义的头结点 是一个虚拟头结点，而不是真正的链表头结点
        _size = 0;
    }

    // 获取到第 index 个节点数值，如果 index 是非法数值直接返回-1， 注意 index 是从 0 开始的，第 0 个节点就是头结点
    //get 只是查询，不需要改变 size 的值（size++或者 size--
    int get(int index) {
        if (index > (_size - 1) || index < 0) {
            return -1;
        }
        LinkedNode* cur = _dummyHead->next;
        while(index--){ // 如果--index 就会陷入死循环
            cur = cur->next;
        }
        return cur->val;
    }

    // 在链表最前面插入一个节点，插入完成后，新插入的节点为链表的新的头结点
    void addAtHead(int val) {
        LinkedNode* newNode = new LinkedNode(val);
        newNode->next = _dummyHead->next;
        _dummyHead->next = newNode;
        _size++;
    }

    // 在链表最后面添加一个节点
    void addAtTail(int val) {
        LinkedNode* newNode = new LinkedNode(val);
        LinkedNode* cur = _dummyHead;
        while(cur->next != nullptr){
            cur = cur->next;
        }
        cur->next = newNode;
        _size++;
    }

    // 在第 index 个节点之前插入一个新节点，例如 index 为 0，那么新插入的节点为链表的新头节点。
    // 如果 index 等于链表的长度，则说明是新插入的节点为链表的尾结点
    // 如果 index 大于链表的长度，则返回空
    // 如果 index 小于 0，则在头部插入节点
    void addAtIndex(int index, int val) {

        if(index > _size) return;
        if(index < 0) index = 0;
        LinkedNode* newNode = new LinkedNode(val);
        LinkedNode* cur = _dummyHead;
        while(index--) {
            cur = cur->next;
        }
        newNode->next = cur->next;
        cur->next = newNode;
```

```
        _size++;
    }

    // 删除第 index 个节点，如果 index 大于等于链表的长度，直接 return，注意 index 是从 0 开始的
    void deleteAtIndex(int index) {
        if (index >= _size || index < 0) {
            return;
        }
        LinkedNode* cur = _dummyHead;
        while(index--) {
            cur = cur ->next;
        }
        LinkedNode* tmp = cur->next;
        cur->next = cur->next->next;
        delete tmp;
        //delete 命令指示释放了 tmp 指针原本所指的那部分内存，
        //被 delete 后的指针 tmp 的值（地址）并非就是 NULL，而是随机值。也就是被 delete 后，
        //如果不再加上一句 tmp=nullptr,tmp 会成为乱指的野指针
        //如果之后的程序不小心使用了 tmp，会指向难以预想的内存空间
        tmp=nullptr;
        _size--;
    }

    // 打印链表
    void printLinkedList() {
        LinkedNode* cur = _dummyHead;
        while (cur->next != nullptr) {
            cout << cur->next->val << " ";
            cur = cur->next;
        }
        cout << endl;
    }
};
```

0206.翻转链表

改变指针指向，用一个 pre，一个 cur

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* cur = head;
        ListNode* pre = NULL;
        while(cur!=NULL){
            ListNode* tmp=cur->next;
            cur->next = pre;
            pre=cur;
            cur=tmp;
```

```
        }
        return pre;
    }
};
```

24. Swap Nodes in Pairs

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode* cur = dummyHead;
        while (cur->next != NULL && cur->next->next != NULL) { // 修改条件，避免空指针
            ListNode* tmp = cur->next;
            ListNode* tmp2 = cur->next->next;

            // 调整指针顺序，正确交换节点
            cur->next = tmp2;
            tmp->next = tmp2->next;
            tmp2->next = tmp;

            // 移动 cur 指针，准备交换下一对
            cur = tmp;
        }
        return dummyHead->next;
    }
};
```

19. Remove Nth Node From End of List 删除倒数第 N 个节点
快慢指针

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* fast = head;
        ListNode* slow = head;
        while(n--){
            fast = fast->next;
        }
        if(fast==NULL){
            head = head->next;
        }
        else{
            while(fast->next!=NULL){
                fast=fast->next;
                slow=slow->next;
            }
            slow->next=slow->next->next;
        }
```

```
        return head;
    }

};
```

142. Linked List Cycle II 环形链表

错误 1:
```
ListNode * index1 = new ListNode(0);
ListNode * index2 = new ListNode(0);
index1 = head;
index2 = slow;
```
index1 和 index2 的初始化可以直接赋值为 head 和 slow，无需创建新节点。//不需要创建一个物理位置给他

错误 2：链表的循环终止条件我总是拿不准

while(fast && fast->next)

因为 fast->next 可以是最后一个节点，fast->next->next 空了进入下一次 while 就不执行循环了

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode * fast = head;
        ListNode * slow = head;
        while(fast && fast->next){
            fast = fast->next->next;
            slow = slow->next;
            if(fast==slow){
                ListNode * index1 = head;
                ListNode * index2 = slow;
                while(index1!=index2){
                    index1=index1->next;
                    index2=index2->next;
                }
                return index1;
            }
        }
        return NULL; //没有环，没找到，那么就返回 NULL
    }
};
```

三．哈希表 判断一个元素是否在集合中出现过
利用数组，index 存元素值，value 存 0 或 1，表示是否存在。前提是因为题目限制了数值的大小。
字母：a 和 b 的种类和个数互相满足（相等，=0）；只需要 b 的满足 a（单方面相等，不=0 就 false）；求 abcde…最小个数（min(hash, temp)）；

0242.有效的字母异位词

给定两个字符串 s 和 t，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1: 输入: s = "anagram", t = "nagaram" 输出: true

示例 2: 输入: s = "rat", t = "car" 输出: false

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        // int hash[26]=0;
        int hash[26]={0};
        for(int i=0;i<s.size();i++){
            hash[s[i]-'a']++;       // 并不需要记住字符 a 的 ASCII，只要求出一个相对数值就可以了
        }
        for(int i=0;i<t.size();i++){
            hash[t[i]-'a']--;
        }
        for(int i=0;i<26;i++){
            if(hash[i]!=0) return false;       // record 数组如果有的元素不为零 0，说明字符串 s 和 t 一定是谁多了字符或者谁少了字符。
        }
        return true;            // record 数组所有元素都为零 0，说明字符串 s 和 t 是字母异位词
    }
};
```

383. Ransom Note 赎金信
magazine 字母的种类和个数满足 ransom note 即可，不需要 a 和 b 正好互相满足

canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true
这道题目和 242.有效的字母异位词很像，242.有效的字母异位词相当于求 字符串 a 和 字符串 b 是否可以相互组成 ，而这道题目是求 字符串 a 能否组成字符串 b，而不用管字符串 b 能不能组成字符串 a。

```
class Solution {
public:
    bool canConstruct(string ransomNote, string magazine) {
        int record[26] = {0};
        //add
        if (ransomNote.size() > magazine.size()) { // 肯定满足不了 a 了
            return false;
        }
        for (int i = 0; i < magazine.length(); i++) {
            // 通过 record 数据记录 magazine 里各个字符出现次数
            record[magazine[i]-'a'] ++;
        }
        for (int j = 0; j < ransomNote.length(); j++) {
            // 遍历 ransomNote，在 record 里对应的字符个数做--操作
```

```
            record[ransomNote[j]-'a']--;
            // 如果小于零说明 ransomNote 里出现的字符，magazine 没有
            if(record[ransomNote[j]-'a'] < 0) {      // 满足不了就 return false
                return false;
            }
        }
        return true;            //都能满足就 return true
    }
};
```

1002. Find Common Characters 查找常用字符

多个字符串，共同的字母和共同的出现个数 是哪些？

// for 循环中记录最小值，不就是共同拥有的个数嘛

string s(1, 字符) 的意思是：构造一个长度为 1 的字符串，内容是这个字符。char 转换为 string

```
class Solution {
public:
    vector<string> commonChars(vector<string>& words) {
        vector<string> result;
        int hash[26] = {0}; //最后的结果

        for(char i:words[0]){
            hash[i-'a']++;
        }

        for(int i=1; i<words.size();i++){
             int temp[26] = {0}; //每次清零，计算每个字符串中每个字母出现的次数
            for(char j:words[i]){
                temp[j-'a']++;
            }
            for(int j=0;j<26;j++){
                hash[j] = min(hash[j], temp[j]); // 不能放在上一个 for 里面顺便做，因为一个字符串里面可能有多个同一个字母，需要遍历完才知道他的个数
            }
        }

        for(int i=0;i<26;i++){
            while(hash[i]!=0){
                string s = string(1,i +'a'); // char 转换为 string 类型
                result.push_back(s);
                hash[i]--;
            }
        }
        return result;
    }
};
```

349. Intersection of Two Arrays 两个数组的交集

unordered_set 去重，无顺序要求

的函数：

1）初始化

unordered_set<int> set;

unordered_set<int> set(vector.begin(),vector.end()); 用一段范围内的所有元素来初始化这个集合。nums1.begin() 是指向第一个元素的指针，nums1.end() 是指向最后一个元素后面那个位置的指针。

2）判断某个元素是否在 set 里面出现过

set.find(value1) != set.end()
3）set.insert(value1)

示例 1：

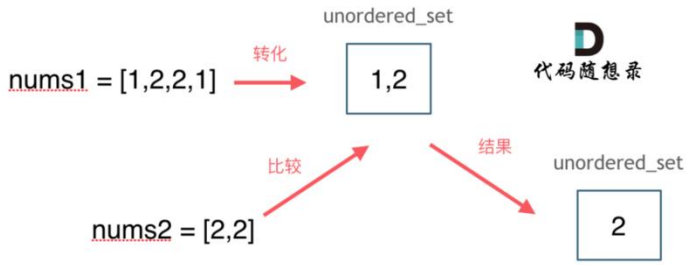
```
输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]
```

示例 2：

```
输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]
```

说明：输出结果中的每个元素一定是唯一的。我们可以不考虑输出结果的顺序。

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> result;
        int hash[1005] = {0};
        for(int i : nums1){
            hash[i] = 1;    //i 这个值出现过，就赋值为 1
        }
        for(int i : nums2){
            if(hash[i] == 1){
                result.insert(i);
            }
        }
        return vector<int> (result.begin(),result.end());
    }
};
```




```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> result; // 结果用 unordered_set，因为要求不重复且无顺序要求
        unordered_set nums1_set(nums1.begin(),nums1.end()); //vector 转为 unordered_set,
相当于去重了
        for(int i : nums2){                //遍历 nums2
            if(nums1_set.find(i)!=nums1_set.end()){ //判断是否在 nums1 中出现过
                result.insert(i);
            }
        }
        return vector<int>(result.begin(),result.end());
    }
};
```

202.快乐数

输入：19
输出：true
解释：
1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1

```
class Solution {
public:
    int calSum(int num){
        int sum = 0;
        while(num){
            sum += (num % 10) * (num % 10);
            num /= 10;
        }
        return sum;
    }
    bool isHappy(int n) {
        unordered_set<int> set;
        int sum = calSum(n);
        while(true){
            if(sum==1){
                return true;
            }
            if(set.find(sum)!=set.end()){  //陷入无限循环就表示不可能是快乐数了，要能出来早出来了  it loops endlessly in a cycle
                return false;
            }
            set.insert(sum);
            sum = calSum(sum);

        }
    }
};
```

1. Two Sum 两数之和

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> map;
        for(int i=0;i<nums.size();i++){
            auto iter = map.find(target-nums[i]);
            if(iter!=map.end()){
                return {iter->second, i};
            }
            map.insert(pair<int, int>(nums[i],i));
        }
        return {};
    }
};
```

错误：先 for 一遍把 nums[i]和 i 存进 map。
会造成：if 当前 nums[i] 的值是 5，而 target 是 10。此时 target - nums[i] 等于 5。 find(target - nums[i]) 会找到自己。

454. 四数相加 II

a+b+c+d=0

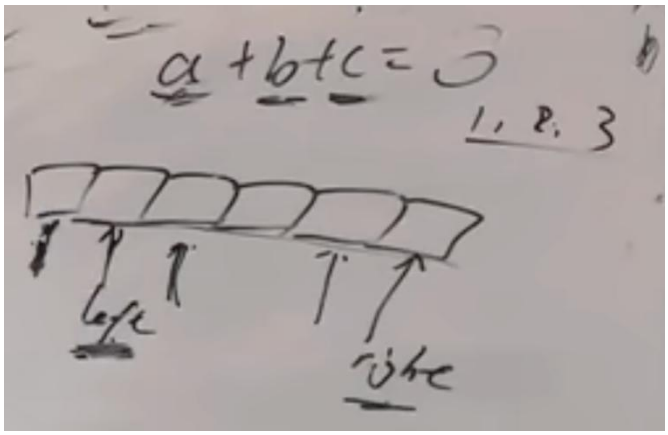
a+b = -(c+d)

```
class Solution {
public:
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3,
vector<int>& nums4) {
        unordered_map<int,int> umap;
        for(int a:nums1){ //遍历 vector 的方法
            for(int b:nums2){
                umap[a+b]++;
            }
        }
        int count = 0;
        for(int c:nums3){
            for(int d:nums4){
                if(umap.find(0-(c+d))!=umap.end())
                    count += umap[0-(c+d)];
            }
        }
        return count;
    }
};
```

15. 三数之和

一层 for 循环+双指针的思路比较简单，但是关键是去重。不可以[1,-1,0]出现两次，但可以[1,1,-2]中 1 出现两次。

去重 a 比较 nums[i] 和 nums[i-1]：-1,-1,2



i 固定。大于零，right--。小于零，left++。

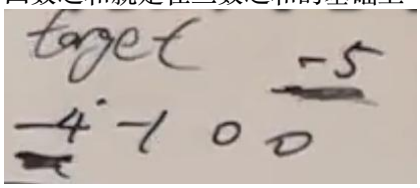
```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        sort(nums.begin(),nums.end());
        // 3 个数，第一个是 nums[i]，后面两个是双指针
        for(int i=0;i<nums.size();i++){
            if(nums[i]>0){  // 因为从小到大排序了，既然第一个数都比 0 大了，那不用往后再加了
                return result;
            }
            // 去重 a
```

```
            if(i>=1 && nums[i]==nums[i-1]){
                continue;
            }
            int left = i+1;
            int right = nums.size()-1;
            while(left<right){
                if(nums[i]+nums[left]+nums[right]<0){
                    left++;
                }else if(nums[i]+nums[left]+nums[right]>0){
                    right--;
                }else{ // 相等的情况！
                    result.push_back(vector<int>{nums[i],nums[left],nums[right]}); //找到了就往里放
吧（result，咱们的蛇皮袋）                    // b 和 c 的去重
                    while(right>left && nums[right]==nums[right-1]){
                        right--;
                    }
                    while(right>left && nums[left]==nums[left+1]){
                        left++;
                    }
                    //继续往下走吧
                    right--;
                    left++;
                }

            }
        }
        return result;
    }
};
```

18. 四数之和

三数之和 target 是 0，四数之和目标值是 target

四数之和就是在三数之和的基础上，外面再套一层 for 循环



不要判断 nums[k] > target 就返回了，负数加负数会更小-4 比-5 大了，但是再加个别的负数比如-1 就会变小了

四 · 字符串
reverse 库函数，时间复杂度是 O(n)，空间复杂度是 O(1)
reverse(start_index, end_index+1) 左闭右开

344. 反转字符串

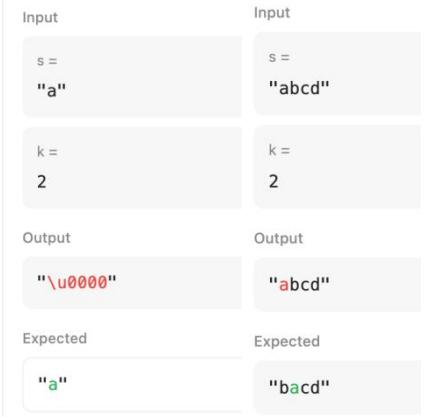
Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]
利用双指针反转字符串

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        for(int i=0, j=s.size()-1; i<s.size()/2; i++, j--){
            swap(s[i],s[j]);
        }
    }
};
```

不是 for(int i=0, int j=s.size()-1; i<j; s.size()/2; i++, j--){

0541. 反转字符串 II

输入: s = "abcdefg", k = 2
输出: "bacdfeg"




```
class Solution {
public:
    string reverseStr(string s, int k) {
        int n = s.size();
        for(int i=0;i+2*k<=n;i=i+2*k){
// 为什么不是 i<n，为什么不是 i+2*k<n
            reverse(s.begin()+i,s.begin()+i+k);
        }
        // 处理剩余的部分，不足 2k 的部分
        int rest_start = n - n%(2*k); // 长度减去余数就是剩下部分的起始 index
        if(n%(2*k) < k){
            reverse(s.begin()+ rest_start, s.end()); // 我感觉用 s.end() - 很容易出错
        }else{
            reverse(s.begin()+ rest_start, s.begin()+ rest_start+k);
             // 因为 reverse 函数是左闭右开
        }
        return s;
    }
};
```

字符串：替换数字

151.翻转字符串里的单词

```
class Solution {
public:
    string reverseWords(string s) {
        // Step 1: 去除多余空格
        int slow=0;
        bool exist = false;
        for(int i=0; i<s.size(); i++){
            // i ++ following for loop
            if(s[i]!=' '){
                s[slow++]=s[i];
                exist = true;
                continue;
            }
            if(exist){
                s[slow++] = ' ';
                exist = false;
            }
            // 所以最后如果有空格，那也会变成 1 个空格，删掉它就好了
        }
        if(s[slow-1]==' '){s.resize(slow-1);}
        else{s.resize(slow);}
        // Step 2: 翻转
        reverse(s.begin(),s.end());
        // Step 3: 再每个单词分别反转
        int start = 0;
        for(int i=0;i<s.size();i++){
            if(s[i]==' '){
                // 如何表示到达 string 结尾 i==s.size()
                reverse(s.begin()+start,s.begin()+i);
                // reverse(s.begin() + start, (i == s.size() - 1) ? s.begin() + i + 1 : s.begin() + i);
                start = i+1;
            }
            if(i==s.size()-1){
                reverse(s.begin()+start, s.begin()+i+1);
            }
        }
        return s;
    }
};
```

字符串：右旋字符串

KMP 算法

字符串：459.重复的子字符串

判断它是否可以由它的一个子串重复多次构成

移动匹配：s+s 得到一个新的字符串，然后 erase 首字母和尾字母，然后调用库函数 ss.find(s)

```cpp
void RemoveExtraSpace(string &s){
    // 双指针保留所有单词，不同单词中间只留一个空格
    // 快指针指到字符了说明结束位置需要一个空格分割
    int slow = 0, fast = 0, need = false;
    for( ; fast < s.size(); fast++){
        if(s[fast] != ' '){
            s[slow++] = s[fast];
            need = true;
            continue;
        }
        if(need){
            s[slow++] = ' ';
            need = false;
        }
    }
    // 现在最后有没有空格取决于原字符最后一个单词后面有没有空格
    // 有的话直接去了就行
    if (s[slow-1] == ' '){
        s.resize(slow-1);
    }else{
        s.resize(slow);
    }
}
```

栈与队列
232. 用栈实现队列 Implement Queue using Stacks
StIn 和 StOut

20. 有效的括号 Valid Parentheses
括号匹配是使用栈解决的经典问题。如果 s[i]是左括号，那么就放对应的右括号进 stack，else pop，最后 stack 为空就是有效的括号。
三种情况：
1）左方向括号，数量多余
2）右方向括号，数量多余
3）没有多余，但括号类型不匹配

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        for(int i=0;i<s.size();i++){
            if(s[i]=='('){
                st.push(')');
            }else if(s[i]=='['){
                st.push(']');
            }else if(s[i]=='{'){
                st.push('}');
            }
            //不是左括号就是右括号了
            //值不相等
            // else if(s[i]!=st.top()||st.empty()){return false;} //RE
            else if(st.empty()||s[i]!=st.top()){return false;} //要先保证 st 不为空
            else{
                st.pop();
            }
        }
        return st.empty();//左多
    }
};
```

1047. 删除字符串中的所有相邻重复项
20. 有效的括号 是匹配左右括号，本题是匹配相邻元素，本题也是用栈来解决的经典题目。
栈的目的，就是存放遍历过的元素，当遍历当前的这个元素的时候，去栈里看一下我们是不是遍历过相同数值的相邻元素。

```
class Solution {
public:
    string removeDuplicates(string s) {
        stack<char> st;
        for(char i:s){//无就放进去，有就拿出来，剩下的反转字符串就是 result
            if(st.empty()||i!=st.top()){ //考慮了最開始栈为空的情况。且这种情况要写在||前面
                st.push(i);
            }
            else{
                st.pop();
            }
        }
        string result;
        while(!st.empty()){
            result+=st.top();
```

```
            st.pop();
        }
        reverse(result.begin(),result.end());
        return result;
    }
};
```

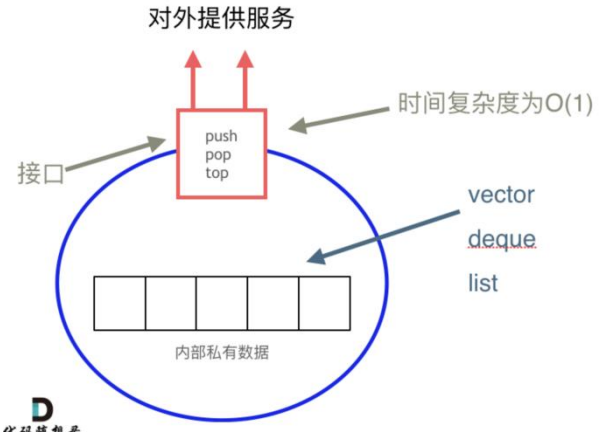
150. 逆波兰表达式求值

如果 s[i]是运算符，就把 st 中上两个 pop 出来计算

逆波兰表达式是一种后缀表达式，运算符写在后面。(1 + 2) * (3 + 4) 逆波兰表达式写法为 ((1 2 +) (3 4 +) *) 。两个优点：

- 去掉括号后表达式无歧义，上式即便写成 1 2 + 3 4 + * 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到运算符则取出栈顶两个数字进行计算，并将结果压入栈中。

遍历和栈在某种程度上是可以转换的。后序遍历的方式把二叉树序列化了



239. **滑动窗口最大值** 是使用**单调队列**的经典题目。

window，窗口，会滑动，存放最大值。三个函数操作：

pop：是大的值，但不得不 pop，因为窗口要往后移动了。If(que.front()==val)就是这种情况。不然就不用 pop 了，因为这个最大值还在窗口里（尽管右移一位了）

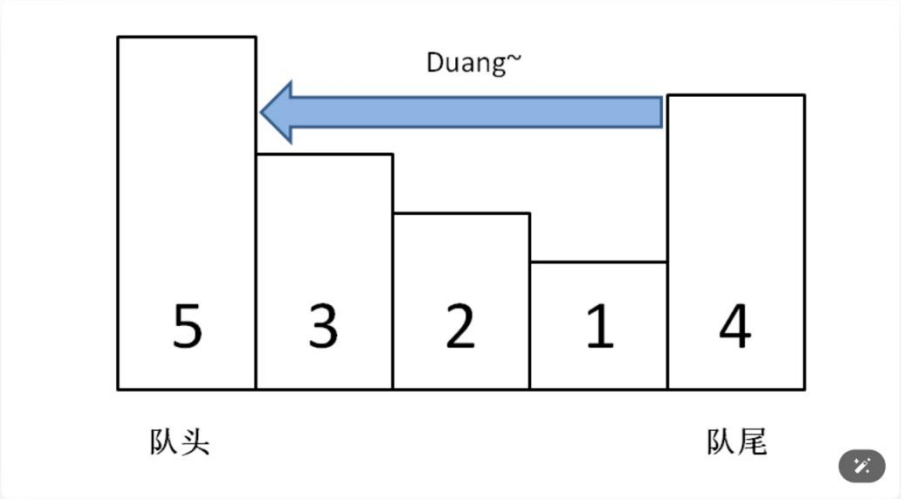
push：如果该元素比之前的都大，那就把前面的元素全部弹出（想象每个前来的元素，只要是值更大，就把前面所有的弹出）（没必要去维护之前比 3 还小的元素）{2, 3, 5, 1 ,4}，单调队列里只维护{5, 4} 就够了，保持单调队列里单调递减，此时队列出口元素就是窗口里最大元素。{ 5, 2 ,3 ,1}

get max value，维护出口处是最大值。return que.front()。不是每个值都会进到单调队列里面。window 里面放的一直是“最大值”（窗口里，k 个元素，的最大值）。

第一个滑动窗口只 push，不 pop，所以单独拎出来处理。窗口往后滑，前面一个 pop，后面一个 push。

```
class Solution {
private:
    class MonotonicQueue{
        public:
            deque<int> que;
            void pop(int val){
                if(!que.empty()&& val==que.front()){
                    que.pop_front();
```

```
            }
        }
        void push(int val){
            while(!que.empty()&&val>que.back()){// 5,2,1,3
                que.pop_back();
            }
            que.push_back(val);
        }
        int max(){
            // if(!que.empty()) //因为是返回 int 的函数，如果 que 为空的话，没有返回的值了。会 CE
            return que.front();
        }
    };
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        MonotonicQueue window;
        vector<int> res;
        for(int i=0;i<k;i++){
            window.push(nums[i]);
        }
        res.push_back(window.max());
        for(int i = k;i<nums.size();i++){
            window.push(nums[i]);
            window.pop(nums[i-k]);
            res.push_back(window.max());
        }
        return res;
    }
};
```



如果每个元素被加入时都这样操作，最终单调队列中的元素大小就会保持一个**单调递减**的顺序，因此我们的 `max` 方法就很好写了，只要把队头元素返回即可；`pop` 方法也是操作队头，如果队头元素是待删除元素 `n`，那么就删除它：

347.前 K 个高频元素

【优先级队列】的底层实现就是【堆】。大顶堆，小顶堆。

当然可以用排序算法来排序 value，十种任选其一，但是！时间复杂度都大于 nlogn。

优化：只维护 k 个元素的排序。Priority_queue 就是 push 进去，他自己会排序。top 是最大值（默认）/最小值（overwrite 比较函数）。pop 会顶弹出堆顶元素。

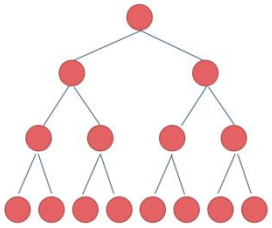
比较函数：左大于右对于快速排序大顶堆（递减，大的值在前面），左大于右对于优先级队列是小顶堆。

```
class Solution {
public:
    class mycomparison {
        public:
            bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs) {
                return lhs.second > rhs.second;
            }
        };
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        priority_queue<pair<int, int>, vector<pair<int, int>>, mycomparison> pri_que;
        //值和对应的频率放进 key value 中了
        unordered_map<int, int> map;
        for(int i=0;i<nums.size();i++){
            map[nums[i]]++;
        }
        //排序 value，输出前 k 个。遍历 map 的 value（就是利用优先级队列排序数值，只不过数值在 map 里
        for(unordered_map<int, int>::iterator it = map.begin();it!=map.end();it++){
            pri_que.push(*it);
            if(pri_que.size()>k){
                pri_que.pop();
            }
        }
            // 找出前 K 个高频元素，因为小顶堆先弹出的是最小的，所以倒序来输出到数组
        vector<int> result(k);
        for (int i = k - 1; i >= 0; i--) {
            result[i] = pri_que.top().first;
            pri_que.pop();
        }
        return result;
    }
};
```

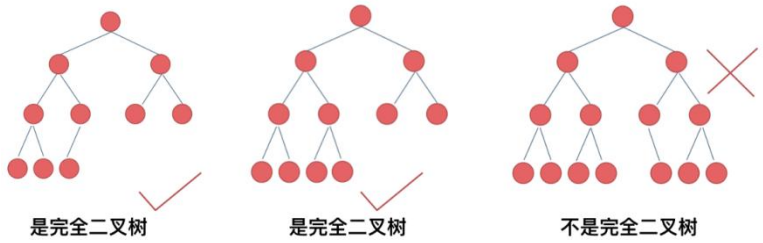
题目最终需要返回的是前 k 个频率最大的元素，所以我们只维护 k 个元素的排序即可（借助堆），从而进一步优化时间复杂度。
pop 会把堆顶弹出（最小值）。

二叉树

1. 二叉树的种类

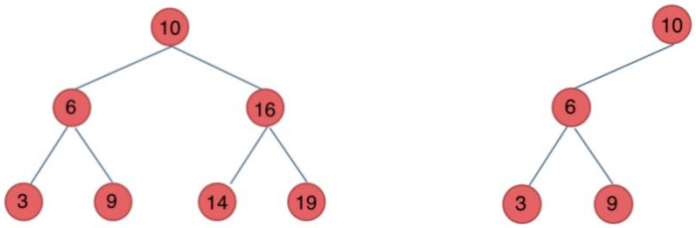


满二叉树

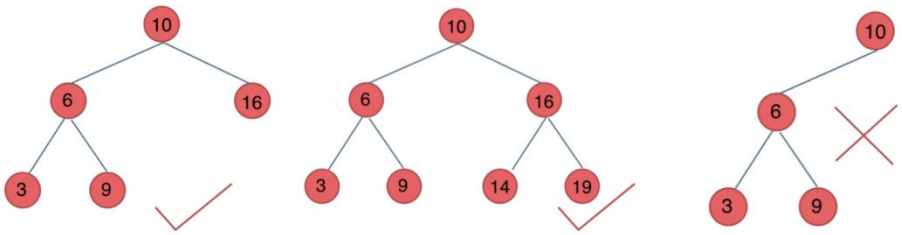


完全二叉树：左满，右可空（指最后一行）

- 优先级队列其实是一个堆，堆就是一棵完全二叉树，同时保证父子节点的顺序关系。



二叉搜索树有数值，数值有顺序：左小右大



最后一棵 不是平衡二叉树，因为它的左右两个子树的高度差的绝对值超过了1。

二叉平衡搜索树：左右子树高度差不超过 1

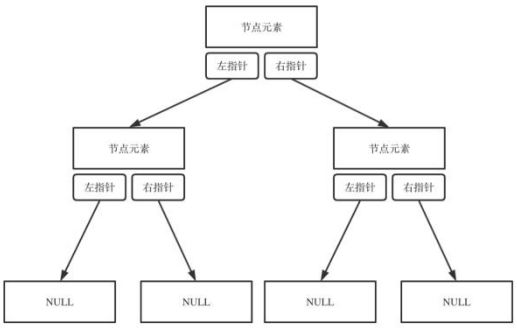
- C++中 map、set、multimap，multiset 的底层实现都是平衡二叉搜索树，所以 map、set 的增删操作时间时间复杂度是 logn，注意我这里没有说 unordered_map、unordered_set，unordered_map、unordered_set 底层实现是哈希表。
- 所以大家使用自己熟悉的编程语言写算法，一定要知道常用的容器底层都是如何实现的，最基本的就是 map、set 等等，否则自己写的代码，自己对其性能分析都分析不清楚!

2. 二叉树的存储方式

- 链式存储：用链表，易于理解
- 顺序存储：用数组，内存连续

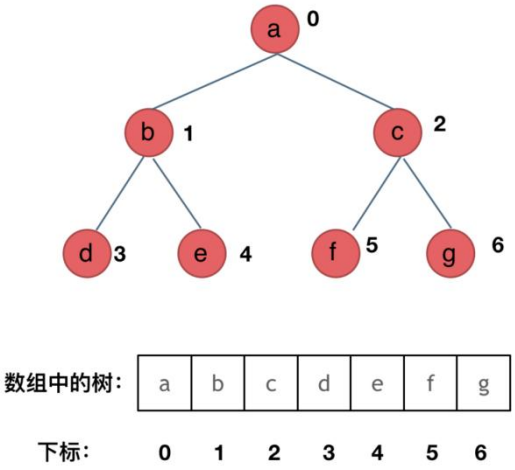
```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
```

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



链式存储是大家很熟悉的一种方式，那么我们来看看如何顺序存储呢？

其实就是用数组来存储二叉树，顺序存储的方式如图：

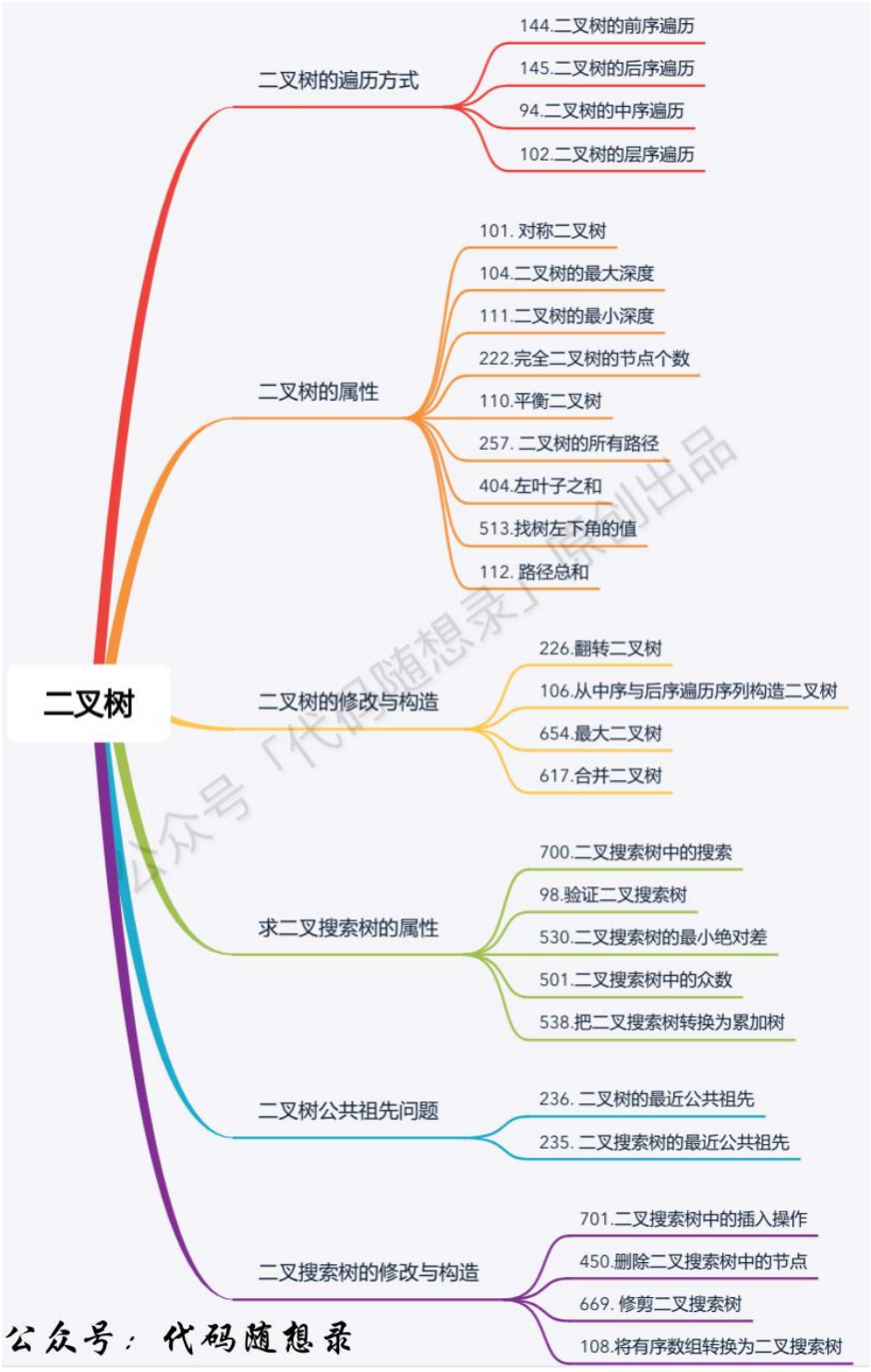


左 2i+1，右 2i+2

3. 二叉树的遍历方式

- **深度优先搜索：**前中后序遍历（递归法，迭代法）
  - 中间节点的遍历顺序，就是所谓的前中后序
- **广度优先搜索：**层序遍历（迭代法）

这两种遍历也是图论中最基本的两种遍历方式。



144, 145, 94 Binary Tree Preorder/Postorder/Inorder Traversal 前中后序遍历（递归法）

```
class Solution {
public:
    void traversal(TreeNode* cur, vector<int>& vec) {  // 1. 确定递归函数的参数和返回值
        if (cur == NULL) return;  // 2. 确定终止条件
        // 3. 确定单层递归的逻辑
        vec.push_back(cur->val);    // 中
        traversal(cur->left, vec);  // 左
        traversal(cur->right, vec); // 右
    }
```

```
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result; //让 result 这个变量去淌这趟浑水，记录结果，所以递归函数不需要返回值（返回 void 即可）。
        traversal(root, result);
        return result;
    }
};
```

5. 二叉树的层序遍历 102. Binary Tree Level Order Traversal

队列先进先出，符合一层一层遍历的逻辑。（用栈先进后出适合模拟深度优先遍历也就是递归的逻辑。这种层序遍历方式就是图论中的广度优先遍历，只不过现在我们应用在二叉树上。

两层循环

```
while(!que.empty())          遍历二叉树的每一层，放进 result 每一行
for(int i = 0; i < size; i++)  遍历当前层的每一个节点
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        // vector<int> que; //取出来，还需要他的左右孩子信息
        queue<TreeNode*> que;
        if(root!=NULL) que.push(root);

        // vector<int> vec;
        vector<vector<int>> result; //result 是个二维数组，把每一层的节点存进 result 的一行
        TreeNode* node;

        while(!que.empty()){ //二叉树的每一层，result 每一行  这里不是 while(size--)!!! 因为 size 是 queue 的长度，which 动态变化
            int size = que.size();
            vector<int> vec; // 定义在这里，也有清空 vec 的作用
            for(int i = 0; i < size; i++){//当前层的每一个节点
                node = que.front();//pop 之前，赋给一个 temp 变量
                que.pop();
                vec.push_back(node->val);
                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
                // size=que.size();
            }
            result.push_back(vec);
        }
        return result;
    }
};
```

- 102.二叉树的层序遍历
- 107.二叉树的层次遍历II
- 199.二叉树的右视图
- 637.二叉树的层平均值
- 429.N叉树的层序遍历
- 515.在每个树行中找最大值
- 116.填充每个节点的下一个右侧节点指针
- 117.填充每个节点的下一个右侧节点指针II
- 104.二叉树的最大深度
- 111.二叉树的最小深度

6. 翻转二叉树 226. Invert Binary Tree

其实就把每一个节点的左右孩子交换一下就可以了。关键在于遍历顺序，前中后序应该选哪一种遍历顺序？答

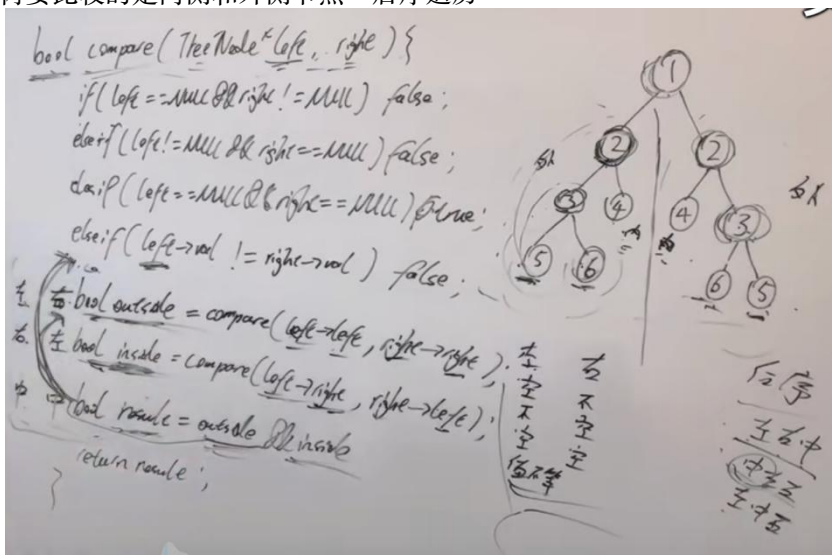
```
TreeNode* invertTree(TreeNode* root) {
        if(root==NULL){   //递归的终止条件
            return NULL;
        }
        swap(root->left, root->right); //swap 函数是啥？可以直接用诶
        invertTree(root->left);
        invertTree(root->right);
        return root; //递归 就是 函数内调用自己，就是相当于 while，终于乱码七糟完事，跳出来了，返回 root 就好了
    }
```

7. 总结

8. 对称二叉树 101.

根节点的左子树和右子树是否可以翻转？比较的是左右子树，不是左右子节点

遍历左右子树要比较的是内侧和外侧节点。后序遍历



```
class Solution {
public:
    bool compare(TreeNode* left, TreeNode* right){
        // 为空
        if(left==NULL && right==NULL) {return true;}
        if(left!=NULL && right==NULL) {return false;}
        if(left==NULL && right!=NULL) {return false;}
        if(left->val != right->val) {return false;}
       // 左右非空且相等 就再往里判断
        return compare(left->left,right->right)&&compare(left->right,right->left);//外侧，内侧
    }
    bool isSymmetric(TreeNode* root) {
        if(root==NULL) {return false;}
        return compare(root->left,root->right);
    }
};
```

后序便利：

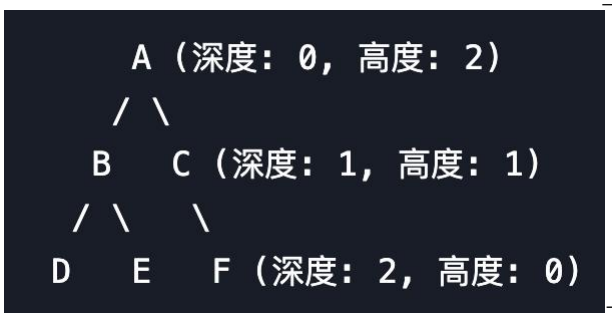
9. 二叉树的最大深度 ，就是根节点的高度　104. Maximum Depth of Binary Tree

~~到根叫深度，到最遠的葉子節點高度。~~

该叶子结点到根节点的距离是深度，整棵树的高度，叶子结点的高度。前序求的是深度，后序求的是高度。

父节点知道子节点的高度，直接加个一就可以了

_____root 根



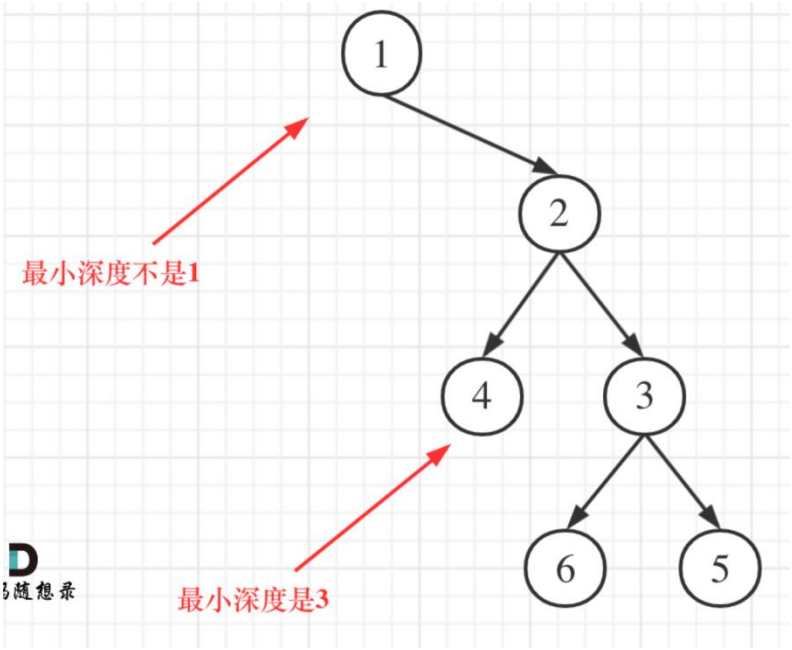
_____高度的基准线

```
class Solution {
public:
    int childDepth(TreeNode* node){
        if(node == NULL) return 0;
        int leftDepth = childDepth(node->left);
        int rightDepth = childDepth(node->right);
        return max(leftDepth,rightDepth)+1;
    }
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;
        return childDepth(root);
    }
};
```

10. **二叉树的最小深度**　111.

题目有说，找根节点到叶子结点，最小距离，最少节点数。

和最大深度的区别在于处理左右孩子不为空的逻辑。




```
class Solution {
public:
    int minDepth(TreeNode* root) {
        if(root==NULL) return 0;
        if(root->left==NULL&&root->right!=NULL){
            return 1+minDepth(root->right);
        }
        if(root->right==NULL&&root->left!=NULL){
            return 1+minDepth(root->left);
        }
        int l = minDepth(root->left);           // 左
        int r = minDepth(root->right);                  // 右
        return 1+min(l,r);                      // 中
    }
};
```

222.　完全二叉树的节点个数

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
private:
    int getNodesNum(TreeNode* root){
        if(root==NULL) {return 0;}
        int leftNum = getNodesNum(root->left);
        int rightNum = getNodesNum(root->right);
        return 1 + leftNum + rightNum;
    }
```

```
public:
    int countNodes(TreeNode* root) {
        return getNodesNum(root);
    }
};
```

或层序遍历，记录一下点的个数

110.　平衡二叉树

一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。
求树的高度用后序遍历，求树的深度用前序遍历。

递归法

```
class Solution {
public:
    // 返回以该节点为根节点的二叉树的高度，如果不是平衡二叉树了则返回-1
    int getHeight(TreeNode* node) {
        if (node == NULL) {
            return 0;
        }
        int leftHeight = getHeight(node->left);
        if (leftHeight == -1) return -1;
        int rightHeight = getHeight(node->right);
        if (rightHeight == -1) return -1;
        return abs(leftHeight - rightHeight) > 1 ? -1 : 1 + max(leftHeight, rightHeight);
    }
    bool isBalanced(TreeNode* root) {
        return getHeight(root) == -1 ? false : true;
    }
};
```

二叉树：257.二叉树的所有路径

本周总结！（二叉树）
二叉树：404.左叶子之和
二叉树：513.找树左下角的值
二叉树：112.路径总和
二叉树：106.构造二叉树
二叉树：654.最大二叉树
本周小结！（二叉树）
二叉树：617.合并两个二叉树
二叉树：700.二叉搜索树登场！
二叉树：98.验证二叉搜索树
二叉树：530.搜索树的最小绝对差
二叉树：501.二叉搜索树中的众数
二叉树：236.公共祖先问题
本周小结！（二叉树）
二叉树：235.搜索树的最近公共祖先
二叉树：701.搜索树中的插入操作
二叉树：450.搜索树中的删除操作
二叉树：669.修剪二叉搜索树
二叉树：108.将有序数组转换为二叉搜索树
二叉树：538.把二叉搜索树转换为累加树
二叉树：总结篇！（需要掌握的二叉树技能都在这里了）

单调栈

739. Daily Temperatures 每日温度赤裸裸的单调栈问题

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        stack<int> st;

        vector<int> result(temperatures.size(),0);
        st.push(0);
        for(int i =1;i<temperatures.size();i++){

            // 大于栈顶元素，那么计算出 result，并弹出
            if(temperatures[i]<temperatures[st.top()]){
                st.push(i);
            }else if(temperatures[i]==temperatures[st.top()]){
                st.push(i);
            }else{
                while(!st.empty() && temperatures[i]>temperatures[st.top()]){
                    result[st.top()] = i - st.top();
                    st.pop();
                }
                st.push(i);
            }
        }
       return result;
    }

};
```

然后发现简化版

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        stack<int> st;

        vector<int> result(temperatures.size(),0);
        st.push(0);
        for(int i =1;i<temperatures.size();i++){
            while(!st.empty() && temperatures[i]>temperatures[st.top()]){
                    result[st.top()] = i - st.top();
                    st.pop();
            }
            st.push(i);
        }
       return result;
    }
};
```

stack 里面存的是 index，因为你有 index 就能找到 value，但是存 value 找不到 index

栈里面放 index，栈顶放比较出来的较大值。

Backtracking 回溯
回溯是一种搜索的方式。回溯是递归的副产品，只要有递归就会有回溯。
回溯法的性能如何呢？

虽然回溯法很难，不好理解，但是回溯法并不是什么高效的算法。回溯的本质是穷举，穷举所有可能，然后选出我们想要的答案，如果想让回溯法高效一些，可以加一些剪枝的操作，但也改不了回溯法就是穷举的本质。
那么既然回溯法并不高效为什么还要用它呢？

因为没得选，一些问题能暴力搜出来就不错了，撑死了再剪枝一下，还没有更高效的解法。此时大家应该好奇了，都什么问题，这么牛逼，只能暴力搜索。

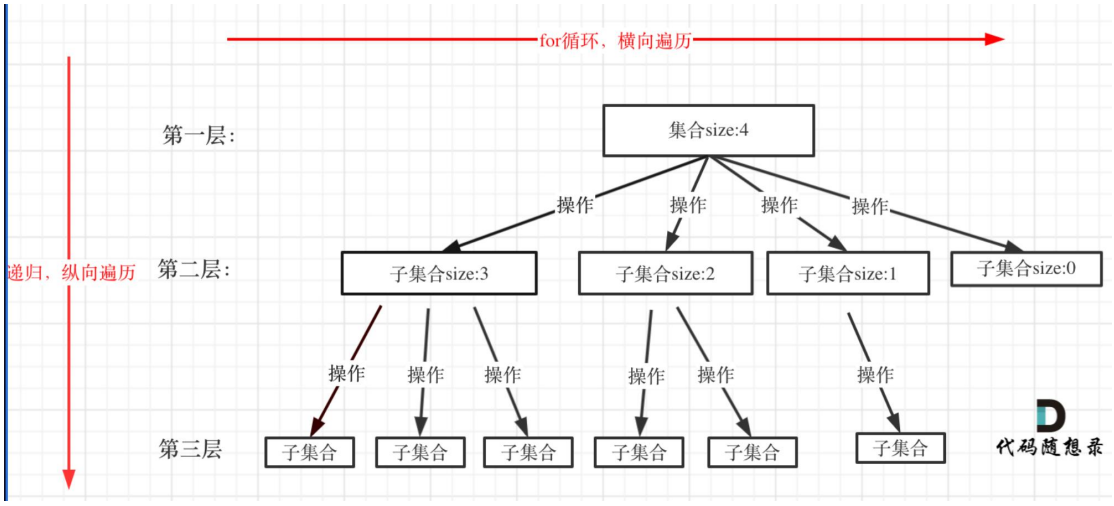
回溯法，一般可以解决如下几种问题：
组合问题：N 个数里面按一定规则找出 k 个数的集合
切割问题：一个字符串按一定规则有几种切割方式
子集问题：一个 N 个数的集合里有多少符合条件的子集
排列问题：N 个数按一定规则全排列，有几种排列方式
棋盘问题：N 皇后，解数独等等
（组合无序，排列有序）

如何理解回溯法？
所有回溯法解决的问题都可以抽象为树形结构。
因为回溯法解决的都是在集合中递归查找子集，集合的大小就构成了树的宽度，递归的深度就构成了树的深度。

回溯法模板（回溯三部曲

回溯函数模板返回值以及参数
1. void backtracking(参数)
2. 终止条件
3. 遍历

```
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
        处理节点;
        backtracking(路径，选择列表); // 递归
        回溯，撤销处理结果
    }
}
```



236. 二叉树的最近公共祖先 LCA
因为我想从下往上看，如果是 p 或 q（左，右）就告诉中间的（如果当前节点 root 是 p 或 q 就 return root），正好用后序遍历
一个节点左有 p，右有 q，那这个节点上就是 LCA；一个节点左或右有 p 或 q，且自己是 p 或 q，他自己就是 LCA。
！所以遍历过程就是，如果它是 p 或 q，就返回这个节点。类似于是，就往上报“报———”

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // if(root==NULL) return root; // 可以合并到一起 反正 root 就是 NULL
        if(root==NULL||root==p||root==q) return root; //终止条件

        TreeNode* left = lowestCommonAncestor(root->left,p,q); //左
        TreeNode* right = lowestCommonAncestor(root->right,p,q); //右

       // 中，哪边不为空就返回哪边，其实是四种情况，这里也是把都为空的情况合并进去了
        if(left!=NULL && right!=NULL){
            return root;
        }
        if(left==NULL && right!=NULL){
            return right;
        }
        return left;
    }
};
```

77. 组合

```
class Solution {
private:
    vector<vector<int>> result; // 存放符合条件结果的集合
    vector<int> path; // 用来存放符合条件结果
    void backtracking(int n, int k, int startIndex) {
        if (path.size() == k) {
            result.push_back(path);
            return;
        }
        for (int i = startIndex; i <= n; i++) {
            path.push_back(i); // 处理节点
```

```
            backtracking(n, k, i + 1); // 递归
            path.pop_back(); // 回溯，撤销处理的节点
        }
    }
public:
    vector<vector<int>> combine(int n, int k) {
        result.clear(); // 可以不写
        path.clear();  // 可以不写
        backtracking(n, k, 1);
        return result;
    }
};
```

216. Combination Sum III　和为 n 的 k 个数的集合

```
class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(int k, int n, int startIndex, int sum){
        if(sum==n){
            if(path.size()==k){
                result.push_back(path);
            }
                return ;
        }
        // 固定三层
        for(int i =startIndex; i<=9;i++){
            sum+=i;
            path.push_back(i);
            backtracking(k,n,i+1,sum);
            sum-=i;
            path.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum3(int k, int n) {
        backtracking(k, n, 1, 0);
        return result;
    }
};
```

贪心
贪心的本质是选择每一阶段的局部最优，从而达到全局最优。
贪心算法并没有固定的套路。唯一的难点就是如何通过局部最优，推出整体最优。
刷题或者面试的时候，手动模拟一下感觉可以局部最优推出整体最优，而且想不到反例，那么就试一试贪心。常识性推导加上举反例。

455 分发饼干 cookies
Sort(g.begin(),g.end());
Sort(s.begin(),s.end());
让尽可能多的人吃到。尽可能多的满足不同的胃口。
先遍历胃口，再遍历饼干。（思路就是，我拿着饼干给你们分。大饼干先来满足大胃口。）先分大饼干，能吃的话就给他，不吃的话给下一个胃口的孩子。满足不了你就算，有人家胃口比你小的。

376. 摆动序列
找最长的子序列，满足差值为正负正负相间。

135. **分发糖果** 每人至少有一个糖果，然后看 ratings 的值，
分发糖果这道题的规则是：他比左或者右的大（比邻居的大），就要比邻居的糖果更多
1. 从前往后便利，从第二个开始便利
if 它比前面大，then 前一项加一，s[i-1]+1
2. 从后往前便利，从倒数第二个开始便利
if 它比后面的大，then 取 max(s[i], s[i+1]+1) //注意这个逻辑，不是直接=后一项加一，因为还要考虑第一次遍历的结果

如果你旁边的大，要多分一个糖果。所以！既要考虑右比左大，又要考虑左比右大。也就是两次遍历。但注意遍历顺序！不能都从前往后便利！！！不然第一次便利的结果就失效了！

！ 从后往前的时候， max

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        vector<int> ncandy (ratings.size(), 1);
        // ncandy[0] = 1;
        for(int i=1;i<ratings.size();i++){
            if(ratings[i]>ratings[i-1]){
                ncandy[i] = ncandy[i-1]+1;
            }
        }
        for(int i=ratings.size()-2; i>=0; i--){
            if(ratings[i]>ratings[i+1]){
                // ncandy[i] = ncandy[i+1]+1;
                ncandy[i] = max(ncandy[i], ncandy[i+1]+1);
            }
        }
        int result=0;
        for(int i=0; i<ratings.size(); i++){
            result+=ncandy[i];
        }
        return result;
    }
};
```

10 柠檬水找零
柠檬水一杯五块。这道题逻辑简单 所以直接写就完了。
初始手里没钱，顾客给，手里才有。
最后都能找零成功就 return true

收到 20 块的找零策略：
先看有没有 10，有的话就用 10+5 找零；否则用 3 张 5 块找零
why？因为 5 是万能的，既能找零 10 块，又能找零 5 块。所以留着用。局部最优。

0406.根据身高重建队列
h 身高
k 表示前面有 k 个比他的 h 大的（大于等于）

和 分发糖果一样，从两个维度考虑。 先确定一个维度，再去考虑另一个维度。
如果按 k 从小到大排
如果按 h 从大到小排，一样时，k 从小到大。可行，先确定身高维度。【6，1】插到第二个位置（因为前面只能有一个比他大的，所以他排在下标为 k 的位置）并不影响 7，1 这一项。
不满足前面有 k 项大于等于它的 h 值的（？代码？不用真的比较，直接按序插入即可），所以去了下标为 k 的位置。

```
class Solution {
public:
    static bool cmp(const vector<int>& a, const vector<int>& b){
        if(a[0]==b[0]){
            return a[1]<b[1];
        }
        return a[0]>b[0];
    }
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people){
        sort(people.begin(),people.end(),cmp); //从左到右对身高排序，大到小

        vector<vector<int>> que; //二维数组，就是一列，两列（第一列 h，第二列 k）
        for(int i=0;i<people.size();i++){
            que.insert(que.begin()+people[i][1], people[i]);
        }
        return que;
    }
};
```

0452.用最少数量的箭引爆气球 尽可能的找重叠区间
对左边界排序。
result 初始化为 1。
如果左边界大于上一个的右边界，那么 result++。否则的话，更新右边界，取 min

```
class Solution {
public:
    //左边界排序，递增
    //虽然 points 是二维，但是比较的时候是两个元素之间比较。比如[1,1][1,2]
    static bool cmp(const vector<int>& a, const vector<int>& b){
        return a[0] < b[0]; //返回 true 的话，递增
    }
```

```
    //一定要写 static（类成员函数中，必须加 static，因为 std::sort 不接受非静态成员函数。
    如果定义为全局函数，不需要 static。）
    //<= 是否可以：不可以，因为 std::sort 的比较函数要求严格弱序，只能使用 < 或 >。）

    int findMinArrowShots(vector<vector<int>>& points) {
        int result = 1; //一定需要一个，如果 points.size>=1
        sort(points.begin(),points.end(),cmp);
        for(int i=1;i<points.size();i++){
            if(points[i][0]>points[i-1][1]){
                result++;
            }else{
                points[i][1]=min(points[i-1][1],points[i][1]); //更新右边界
            }
        }
        return result;
    }
};
```

435. 无重叠区间 尽可能的留不重叠区间
尽可能少的保留区间 min(item[i][1], item[i-1][1])
思考逻辑：
cmp 是不是还要先考虑 length，再排序左边界？不用不用不用，直接递增排序左边界，然后处理右边界的值=min()就好了
if i 左大于等于 i-1 右，那么 no overlap，做 nothing
else {???}

```
class Solution {
public:
    static bool cmp(const vector<int>& a, const vector<int>& b){
        return a[0]<b[0];
    }
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        int result = 0;
        sort(intervals.begin(),intervals.end(),cmp);
        for(int i=1;i<intervals.size();i++){
            if(intervals[i][0]<intervals[i-1][1]){
                result++;
                intervals[i][1] = min(intervals[i-1][1],intervals[i][1]);
            }
        }
        return result;
    }
};
```

763.划分字母区间 记录下所有元素的最远出现位置。
S = "ababcbacadefegdehijhklij"
每个字母最多出现在一个片段中。所以第一个片段要包含所有的 a，往下走的途中，遇到了 b，所以该片段也要包含所有的 b 了。
Hashmap? key: 该字母，value：最远出现位置的下标
S[i] - 'a'作用：用数字代表了字母 abcd...

```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        vector<int> result;
        int hash[26]={0};
        for(int i=0;i<s.size();i++){
            hash[s[i]-'a'] = i;
```

```
        }
        int left=0;
        int right=0;
        for(int i=0;i<s.size();i++){
            right = max(right, hash[s[i]-'a']); //不是 hash[i]
            if(i == right){
                result.push_back(right-left+1);
                left = right+1;
            }
        }
        return result;
    }
};
```

56. 合并区间 Merge Intervals

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b){
        return a[0]<b[0];
    }
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), cmp);
        vector<vector<int>> result;
        result.push_back(intervals[0]);
        for(int i=1;i<intervals.size();i++){
            if(intervals[i][0]<=result.back()[1]){ //不是 intervals[i-1][1]
                result.back()[1]=max(result.back()[1],intervals[i][1]);
            }else{
                result.push_back(intervals[i]);
            }
        }
        return result;
    }
};
```

动态规划
动态规划中每一个状态一定是由上一个状态推导出来的，这一点就区分于贪心。
动态规划是目的导向。状态转移。
贪心没有状态推导，而是从局部直接选最优的。由局部最优，得到全局最优。

509.斐波那契数
不需要维护整个数组，只需要维护 3 个变量。
sum=dp[i]+dp[i-1];
dp[i]=sum;
dp[i-1]=dp[i];

```
class Solution {
public:
    int fib(int n) {
        vector<int> dp(n+1);
        if(n<2){
            return n;
        }
        dp[0]=0;
        dp[1]=1;
        for(int i=2;i<=n;i++){
            dp[i]=dp[i-1]+dp[i-2];
        }
        return dp[n];
    }
};
```

70. 爬楼梯 Climbing Stairs

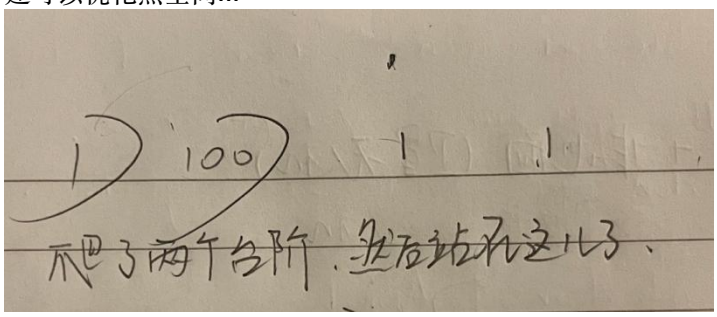
```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n+1);
        if(n<3){
            return n;
        }
        dp[1]=1;
        dp[2]=2;
        for(int i=3;i<=n;i++){
            dp[i]=dp[i-1]+dp[i-2];
        }
        return dp[n];
    }
};
```

0746.使用最小花费爬楼梯
1. dp[i]数组下标及其含义: 站到 i 这个位置上了，花费掉的体力。
2. 确定递推公式 状态推倒：dp[i]要么是从 dp[i-1]来的，要么是从 dp[i-2]来的。因为要么走一步，要么走两步。
dp[i] = min(dp[i-1] + cost[i-1], dp[i-2] + cost[i-2]), i>=2
首先到达了这个位置，然后从这个位置出发走一步到达 i
3. dp 数组如何初始化？题目：“你可以选择从下标为 0 或下标为 1 的台阶开始爬楼梯”
dp[0] = 0;
~~dp[1] = cost[0];~~ 因为可以直接从下标为 1 的台阶开始爬。所以 dp[1] = 0;
4. 确定遍历顺序
遍历 cost 数组

5. 举例推导 dp 数组/有错误就打印出来 dp 数组，看和预计的有啥区别

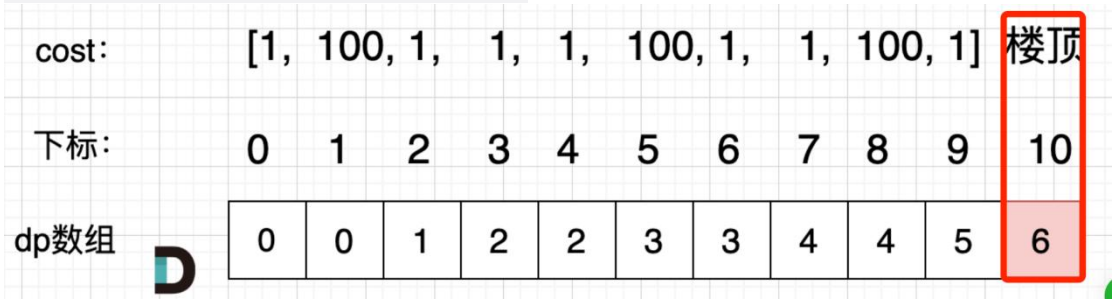
```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        vector<int> dp(cost.size()+1);
        dp[0]=0;
        dp[1]=0;
        for(int i = 2; i <= cost.size(); i++){
            dp[i] = min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2]);
        }
        return dp[cost.size()];  //不是 return dp[cost.size()-1];
    }
};
```

还可以优化点空间...



最后一步不用花费体力。数值多少都无所谓。

输入：cost = [10,15,20]
输出：15
解释：你将从下标为 1 的台阶开始。
- 支付 15 ，向上爬两个台阶，到达楼梯顶部。
总花费为 15 。



62. 不同路径

m*n 网格，有多少种途径可以到达右下角的终点。


```
class Solution {
public:
    int uniquePaths(int m, int n) {
```

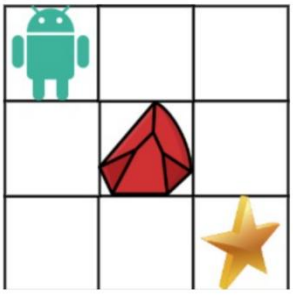
```
        vector<vector<int>> dp(m, vector<int>(n,0));
        for(int i=0;i<n;i++) dp[0][i] = 1;
        for(int i=0;i<m;i++) dp[i][0] = 1;
        for(int i=1;i<m;i++){
            for(int j=1;j<n;j++){
                dp[i][j]=dp[i-1][j]+dp[i][j-1];
                //不是  dp[i][j]=dp[i-1][j]+1+dp[i][j-1]+1; 因为到那了只能往下走一步到达终点了。
            }
        }
        return dp[m-1][n-1];
    }
};
```

改进，减少内存空间。

63. 不同路径 II

网格中有障碍物了(obstacleGrid 值为 1)。

解法就是：有障碍物就保持初始状态。无障碍物再 dp 操作。没什么难的，只需要多考虑这一点即可!



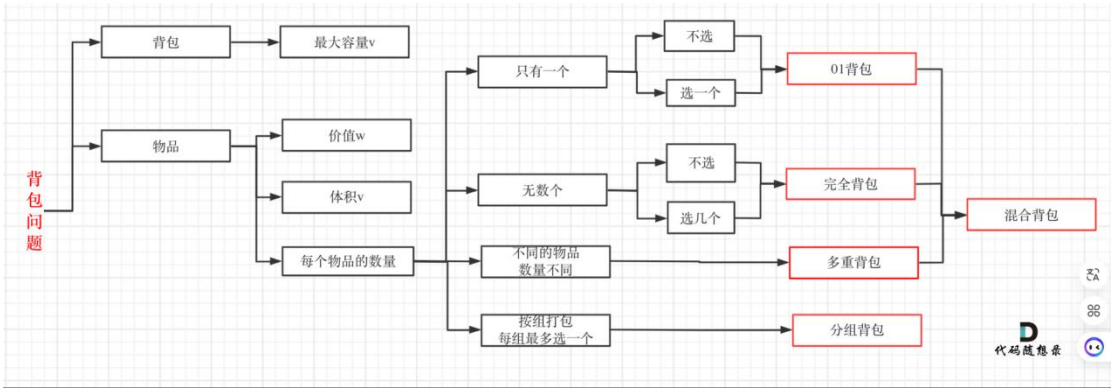
```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size(); //没有直接传入行数和列数

        if(obstacleGrid[0][0]==1||obstacleGrid[m-1][n-1]==1) return 0; //如果起点/终点有障碍物，那么直接返回 0

        vector<vector<int>> dp(m, vector<int>(n,0));
        for(int i=0;i<n && !obstacleGrid[0][i];i++) dp[0][i] = 1;
        for(int i=0;i<m && !obstacleGrid[i][0];i++) dp[i][0] = 1;
        for(int i=1;i<m;i++){
            for(int j=1;j<n;j++){
                if(obstacleGrid[i][j]==0){ //无障碍物再 dp
                    dp[i][j]=dp[i-1][j]+dp[i][j-1];
                }
            }
        }
        return dp[m-1][n-1];
    }
};
```

343. 整数拆分

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

背包问题：01 背包和完全背包，重点是 初始化 和 遍历顺序
分割等和子集问的是能不能装满这个背包
最后一块石头的重量是求背包里面能装的最大重量
给我们一个背包的容量，问我们有多少种方式能把这个背包装满

题意理解：
有一个最多能背重量/容量为 w 的背包和 n 件物品。
第 i 件物品的重量是 weight[i]，对应的价值是 value[i] 。
01 背包只指每件物品都只有一个，所以对应装进去 or 不装。
求解背包最多装价值多少的东西。价值最大化



dp[i][j]：背包装的东西的最大价值是，其中 i 表示 index 为 0-i 的物品可选，j 表示背包的能够承受的重量/承载的容量
装第 i 个物品，价值 dp[i][j] = max(dp[i-1][j-weight[i]], value[i])
不装第 i 个物品，价值 dp[i][j] = dp[i-1][j]
如果 j<weight[i]，那么 dp[i][j] 还是 dp[i-1][j]，都不需要比较

用二维数组实现 01 背包
dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]); //状态转移公式（二维数组方法）

```
for(int i = 1; i < weight.size(); i++) { // 遍历物品 // weight 数组的大小就是物品个数
    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        if (j < weight[i]) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
    }
}
```

用一维数组实现 01 背包（滚动数组），上一层 dp[i-1] 拷贝到 dp[i]来，不用存
dp[j] = max(dp[j], dp[j - weight[i]] + value[i]); // 等式右边的 dp[j]其实是上一层的值

```
for(int i = 0; i < weight.size(); i++) { // 只能第一层循环是遍历物品，第二层循环遍历重量！ why
    for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量，倒叙遍历 why？
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

二维数组利用的是正上方 dp[i-1][j]，和，左上 dp[i-1][j-w[i]]的数据
转换为一维数组，就要利用左边数据，如果正序遍历就会覆盖左边数据（更新成这一行的对应的 dp[j]，但是需要的是上一层的 dp[j]），所以得倒叙遍历
eg. g[10] = max(g[10], g[10-6] + 6)
-重量 +价值

如果正序，g[4]已经更新成这一层的 g[4]了，不再是我们需要的上一层的 g[4]

https://kamacoder.com/problempage.php?pid=1046

```
// 一维 dp 数组实现
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // 读取 M 和 N
    int M, N;
    cin >> M >> N;

    vector<int> costs(M);
    vector<int> values(M);

    for (int i = 0; i < M; i++) {
        cin >> costs[i];
    }
    for (int j = 0; j < M; j++) {
        cin >> values[j];
    }

    // 创建一个动态规划数组 dp，初始值为 0
    vector<int> dp(N + 1, 0);

    // 外层循环遍历每个类型的研究材料：每行，考虑 index=第 i 个研究材料的时候
    for (int i = 0; i < M; ++i) {
        // 内层循环从 N 空间逐渐减少到当前研究材料所占空间
        for (int j = N; j >= costs[i]; --j) {
            // 考虑当前研究材料选择和不选择的情况，选择最大值
            dp[j] = max(dp[j], dp[j - costs[i]] + values[i]); //当 j<costs[i]是，dj[j]的值不用变
        }
    }

    // 输出 dp[N]，即在给定 N 行李空间可以携带的研究材料最大价值
    cout << dp[N] << endl;

    return 0;
}
```

416. 分割等和子集

是否可以将 nums 数组分割成两个子集，使得两个子集的元素和相等。
输入: [1, 5, 11, 5]
输出: true
解释: 数组可以分割成 [1, 5, 5] 和 [11].

这道题为什么可以抽象为 01 背包问题？
这道题我们不关心 value，value[i]和 weight[i]的值都等于 nums[i]
最后只关心 dp[j]是否等于 j，其中 j=sum/2。if yes，return true

```
class Solution {
private:
    int sumCal(vector<int>& nums){
        int sum = 0;
        for(int i=0;i<nums.size();i++){
```

```
            sum += nums[i];
        }
        return sum;
    }
public:
    bool canPartition(vector<int>& nums) {
        int sum = sumCal(nums);
        int m = nums.size();
        int n = sum/2;
        vector<int> dp(n+1,0);
        if(sum % 2 == 1){
            return false;
        }
        for(int i=0;i<m;i++){
            for(int j=n;j>=nums[i];j--){
                dp[j] = max(dp[j],dp[j-nums[i]]+nums[i]);
            }
        }
        if(dp[n] == n) return true;
        else return false;
    }
};
```

1049. Last Stone Weight II 最后一块石头的重量 II

和 416.分割等和子集 一模一样！唯一区别是最后 return sum-dp[n]-dp[n];

两块石头，将它们一起粉碎。
如果 x == y，那么两块石头都会被完全粉碎；
如果 x < y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
最后，最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下，就返回 0。

本题其实是尽量让石头分成重量相同的两堆（尽可能相同） sum / 2 ，相撞之后剩下的石头就是最小的。那么此时问题就是有一堆石头，每个石头都有自己的重量，是否可以 装满 最大重量为 sum / 2 的背包。不能就计算下差值就是结果 sum-dp[j]=另一堆石头的重量。再减去 dp[j] 就是两堆石头的差值。

```
class Solution {
private:
    int sumCal(vector<int>& nums){
        int sum = 0;
        for(int i=0;i<nums.size();i++){
            sum += nums[i];
        }
        return sum;
    }
public:
    int lastStoneWeightII(vector<int>& stones) {
        int sum = sumCal(stones);
        int m = stones.size();
        int n = sum/2;
        vector<int> dp(n+1,0);

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

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

};
```

494. **Target Sum 目标和** ~~？？？我没懂！~~

求 i 个数凑出 j 的方法数，使表达式结果为 target 的组合数

这道题目咋眼一看和动态规划背包啥的也没啥关系。

本题要如何使表达式结果为target，

既然为target，那么就一定有 left组合 - right组合 = target。

left + right = sum，而sum是固定的。right = sum - left

left - (sum - left) = target 推导出 left = (target + sum)/2 。

target是固定的，sum是固定的，left就可以求出来。

此时问题就是在集合nums中找出和为left的组合。

```
class Solution {
    int sumCal(vector<int>& nums){
        int sum = 0;
        for(int i=0;i<nums.size();i++){
            sum += nums[i];
        }
        return sum;
    }
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = sumCal(nums);

       // 这两个提前的剪枝很巧妙
        if((target+sum)%2==1) return 0; //
        if (abs(target) > sum) return 0; // 比如 nums = [1, 2, 3, 4]，那么全部加起来：1 + 2 + 3 + 4 = 10；全部减起来：-1 -2 -3 -4 = -10。所以所有可能的结果，都落在 [-10, 10] 之间。target 的绝对值大于 sum 就永远不可能了

        int left = (target+sum)/2;
        vector<int> dp(left+1,0);
        dp[0] = 1;
        for(int i=0;i<nums.size();i++){
            for(int j=left;j>=nums[i];j--){
                dp[j] = dp[j] + dp[j-nums[i]]; //
            }
        }
        return dp[left];
        }

};
```

474.一和零
dp[j]表示=dp[j]+dp[j-str[i]].contains(0)]

完全背包

二维数组解决 01 背包问题，内层循环遍历物品也好，外层循环遍历物品也好，正序遍历也好，倒序也好，反正每一个值都会存下来，所以都可以。但是用一位数组解决 01 背包问题，只能外层循环遍历物品，内层循环遍历容量，且倒序！一维数组解决完全背包问题，求组合数，内层循环遍历容量且正序！求排列数，内层循环遍历物品！且正序

如果求组合数就是外层 for 循环遍历物品，内层 for 遍历背包。？？

如果求排列数就是外层 for 遍历背包，内层 for 循环遍历物品。？？

```
for (int i = 1; i < n; i++) { // 遍历物品
    for(int j = 0; j <= bagWeight; j++) { // 遍历背包容量
        if (j < weight[i]) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = max(dp[i - 1][j], dp[i][j - weight[i]] + value[i]);
    }
}
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = weight[i]; j <= bagWeight; j++) { // 遍历背包容量
        if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

518. Coin Change II 零钱兑换 II 求组合数

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

如果大家认真做完：分割等和子集， 最后一块石头的重量 II 和 目标和。应该会知道类似这种题目：给出一个总数，一些物品，问能否凑成这个总数。这是典型的背包问题!

本题求的是装满这个背包的物品组合数是多少。又因为每一种面额的硬币有无限个，所以这是完全背包。但本题和纯完全背包不一样，纯完全背包是凑成背包最大价值是多少，而本题是要求凑成总金额的物品组合个数!

5 = 2 + 2 + 1

5 = 2 + 1 + 2

这是一种组合，都是 2 2 1。

如果问的是排列数，那么上面就是两种排列了。组合不强调元素之间的顺序，排列强调元素之间的顺序。 其实这一点我们在讲解回溯算法专题的时候就讲过。

如果求组合数就是外层 for 循环遍历物品，内层 for 遍历背包。

如果求排列数就是外层 for 遍历背包，内层 for 循环遍历物品。

```
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        vector<uint64_t> dp(amount+1,0); // uint64_t 是 C++ 中定义的 无符号 64 位整数类型，存放非常大的非负整数，能表示的 比 int 大太多了，避免因为数值太大而造成的溢出错误
        dp[0] = 1;
        for(int i=0;i<coins.size();i++){
            for(int j=coins[i];j<=amount;j++){
                dp[j] = dp[j] + dp[j-coins[i]];
            }
        }
        return dp[amount];
    }
};
```

377. 组合总和IV

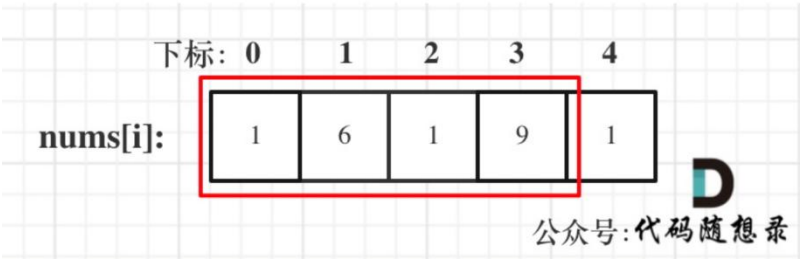
打家劫舍 是 dp 的经典问题
你是一个专业的小偷，计划偷窃沿街的房屋。原则：两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。求要想不被抓，最多能偷多少钱
当前房屋偷与不偷 取决于 前一个房屋和前两个房屋是否被偷了。（这种依赖关系是动规的递推公式。）
dp[i]：考虑下标 i（包括 i）以内的房屋，最多可以偷窃的金额为 dp[i]。
如果不偷第 i 间，那么值为 dp[i - 1]; 如果偷第 i 间，那么值为 dp[i - 2] + nums[i]。所以 dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);
初始化，dp[0] = nums[0], dp[1] = max(nums[0], nums[1]);
dp[i] 是根据 dp[i - 2] 和 dp[i - 1] 推导出来的，那么一定是从前到后遍历!

0198. House Robber

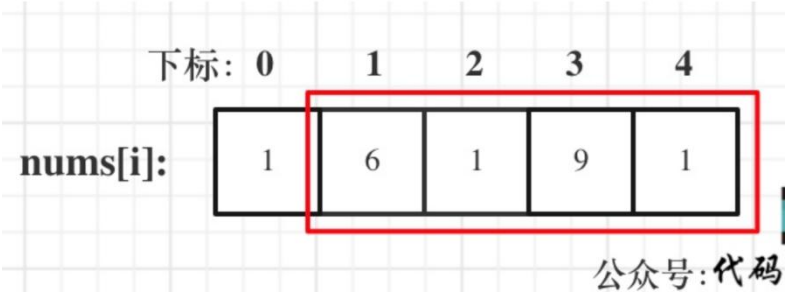
```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0; // 个人觉得没必要
        if (nums.size() == 1) return nums[0];
        vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size();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

213.打家劫舍 II
环形问题 拆解成两个线性问题
因为相邻的两个房子不能同时偷，所以要么偷第一间，要么偷最后一间。

- 情况二：考虑包含首元素，不包含尾元素



- 情况三：考虑包含尾元素，不包含首元素



我的疑问：如果偷第一个房子，dp[0] 还应该考虑倒数第二个房子的 value

```
class Solution {
public:
    int rob(vector<int>& nums) {
```

```
        if(nums.size()==1) return nums[0]; // 没有这句话会 RE，题目说 size>=1，所以也要考虑
size=1 的时候
        if(nums.size()==2) return max(nums[0],nums[1]);
        return max(robRange(nums, 1, nums.size()-1), robRange(nums, 0, nums.size()-2));
    }
    int robRange(vector<int>& nums, int start, int end){
        // 以 start 为起始，类似数组 index=0
        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];
    }
};
```

股票系列　　dp[i][0]表示第 i 天持有股票的收益，dp[i][1]表示第 i 天不持有的收益

121. Best Time to Buy and Sell Stock 买卖股票的最佳时机

```
// 版本一　时间复杂度：O(n)　空间复杂度：O(n)
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        vector<vector<int>> dp(prices.size(),vector<int>(2));
        // i 从 0 开始
        dp[0][0] = -prices[0];
        dp[0][1] = 0;

        for(int i=1;i<prices.size();i++){
            // 第 i 天持有股票的收益：第 i 天买的，或者之前就买了
            dp[i][0] = max(dp[i-1][0], -prices[i]);
            // 第 i 天不持有股票的收益: 之前就卖掉了，或者第 i 天刚卖掉的
            dp[i][1] = max(dp[i-1][1], dp[i-1][0]+prices[i]);
        }
        return dp[prices.size()-1][1];
    }
};
```

从递推公式可以看出，dp[i]只是依赖于 dp[i - 1]的状态。那么我们只需要记录 当前天的 dp 状态和前一天的 dp 状态就可以了，可以使用滚动数组来节省空间??????????????????????

```
// 版本二 时间复杂度：O(n)　空间复杂度：O(1)
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        vector<vector<int>> dp(2, vector<int>(2)); // 注意这里只开辟了一个 2 * 2 大小的二维数组
        dp[0][0] -= prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < len; i++) {
            dp[i % 2][0] = max(dp[(i - 1) % 2][0], -prices[i]);
            dp[i % 2][1] = max(dp[(i - 1) % 2][1], prices[i] + dp[(i - 1) % 2][0]);
        }
```

```
        return dp[(len - 1) % 2][1];
    }
};
```

122. 买卖股票的最佳时机 II

唯一的区别是：股票可多次买卖

```
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];
// 版本二
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        vector<vector<int>> dp(2, vector<int>(2)); // 注意这里只开辟了一个 2 * 2 大小的二维数组
        dp[0][0] -= prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < prices.size(); i++) {
            dp[i % 2][0] = max(dp[(i - 1) % 2][0], dp[(i - 1) % 2][1] - prices[i]);
            dp[i % 2][1] = max(dp[(i - 1) % 2][1], dp[(i - 1) % 2][0] + prices[i]);
        }
        return dp[(prices.size() - 1) % 2][1];
    }
};
    }
};
```

0123. 买卖股票的最佳时机 III

最多可以完成 两笔 交易。

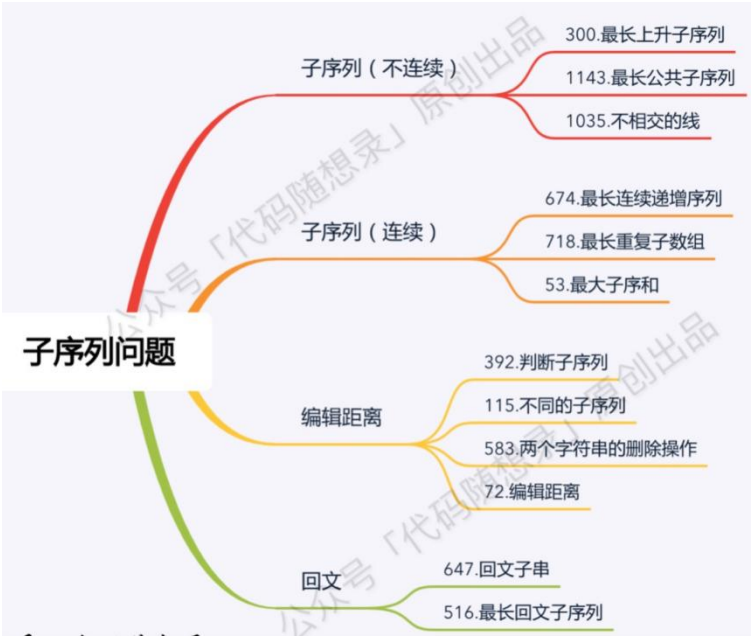
188. 买卖股票的最佳时机 IV

最多可以完成 k 笔交易。

309. 最佳买卖股票时机含冷冻期

714. 买卖股票的最佳时机含手续费

子序列系列


子序列问题
子序列（不连续）
300.最长上升子序列
1143.最长公共子序列
1035.不相交的线
子序列（连续）
674.最长连续递增序列
718.最长重复子数组
53.最大子序和
编辑距离
392.判断子序列
115.不同的子序列
583.两个字符串的删除操作
72.编辑距离
回文
647.回文子串
516.最长回文子序列

图论 深搜 dfs 和广搜 bfs

797. All Paths From Source to Target 所有可达路径

模版题，从起点 0 到终点 n-1 所有可能的路径

这道题里 graph[x] 是邻居表! 不是邻接矩阵。所以 graph.size()就是节点的个数

```
class Solution {
public:
    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        temp.push_back(0);
        dfs(graph, 0); // 为什么需要 graph
        return result;
    }
private:
    vector<vector<int>> result;
    vector<int> temp; // 单条路径
    void dfs(vector<vector<int>>& graph, int x){
        if(x == graph.size()-1){        //graph.size()就是节点个数
            result.push_back(temp);
            return ;
        }
        for(int j = 0; j< graph[x].size(); j++){
            temp.push_back(graph[x][j]);
            dfs(graph, graph[x][j]);
            temp.pop_back();     // 回溯：撤销最近一次的搜索操作
        }
    }

};
```

200. Number of Islands 岛屿的数量深搜

DFS 非常适合用来遍历连通区域（与 相连的）。这道题是 dfs 的原因：往周围走上来就递归，但是我们把压力给到终止条件

这个是我们找到的相邻的陆地，且没有被访问过，并以它再为起点，继续 dfs

“当它是一块陆地,
它会向上下左右试探,
侵略使其成为一个整体

就像一个泡泡,
如果它四周也有泡泡,
他们会合并变成一个大泡泡”

```
class Solution {
private:
    void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int x, int y){
        int dir[4][2] = {0,1,1,0,-1,0,0,-1}; // 0 维代表 x 方向，1 维代表 y 方向
        for(int i=0;i<4;i++){ // 往 4 个方向走
            int nextx = x + dir[i][0];
            int nexty = y + dir[i][1];

            if(nextx<0||nextx>=grid.size()||nexty<0||nexty>=grid[0].size()){ //越界 过
             // grid[i][j]：第 i 行第 j 列
                continue; // 因为 for 循环，所以 continue
            }
            if(grid[nextx][nexty]=='0' || visited[nextx][nexty]){ // 0 或者 visited 过了 过
```

```
            visited[nextx][nexty]=true; // 我自己的一点思考，但是没用！ 其实只对 1 的位置
visited 重要
            continue ;
        }
        if(!visited[nextx][nexty] && grid[nextx][nexty]=='1'){ // 扩张
            visited[nextx][nexty] = true;
            dfs(grid, visited, nextx, nexty); // 再以新殖民地为起点扩张，dfs
        }
    }
  }
public:
  int numIslands(vector<vector<char>>& grid) { // 这道题是 char，'1' 不是 1
    int result=0;
    int m = grid.size();
    int n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n,false));

    for(int i=0;i<grid.size();i++){ // 行数
      for(int j=0;j<grid[0].size();j++){ // 列数
        if(!visited[i][j] && grid[i][j]=='1'){  // 只有根儿是陆地的地方才会开始侵略的种子
          visited[i][j] = true;
          result++;
          dfs(grid, visited, i, j);
        }
      }
    }
    return result;
  }
};
```

695. Max Area of Island 岛屿的最大面积

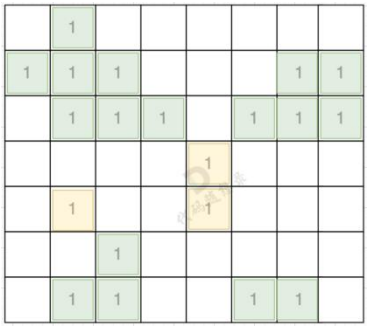
```
class Solution {
private:
  int temp = 0; //全局变量，计算每次进入递归时候合并岛屿数量 temp++，每次从 dfs 出来
和 result 比较谁大更新 result
  void dfs(vector<vector<int>>& grid, vector<vector<bool>>& visited, int x, int y){
    int dir[4][2] = {0,1,1,0,0,-1,-1,0};
    for(int i=0;i<4;i++){ //不是两层 for 循环，而是一行的[0][1]来代表往一个方向走
      int nextx = x + dir[i][0]; // 行 i
      int nexty = y + dir[i][1];
      if(nextx<0||nextx>=grid.size()||nexty<0||nexty>=grid[0].size()){
       // 错误写成了：if(nextx<0 || nextx>=grid[0].size() || nexty<0 || nexty>=grid.size())
       // 误把 i 理解成 nextx，现在的代码对应的是横着的是 y 坐标，竖着的是 x 坐标，x
坐标 duiyingi 行数，y 坐标对应 j 列数
        continue;
      }
      if(visited[nextx][nexty] || grid[nextx][nexty]==0){
        continue;
      }
      if(grid[nextx][nexty]==1 && !visited[nextx][nexty]){
        visited[nextx][nexty] = 1;
        temp++;
        dfs(grid, visited, nextx, nexty);
      }
    }
  }
public:
```

```
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int result=0;
        int m = grid.size(); // 行数
        int n = grid[0].size(); // 列数
        vector<vector<bool>> visited(m, vector<bool>(n, false));
        for(int i=0;i<grid.size();i++){
            for(int j=0;j<grid[0].size();j++){

                if(grid[i][j]==1 && !visited[i][j]){ // 每次找到新陆地的时候
                    visited[i][j] = true;
                    temp=1;
                    dfs(grid, visited, i,j);
                    result = max(result, temp);
                }


            }
        }
        return result;
    }
};
```

1254. Number of Closed Islands 孤岛的个数! 不是总面积

一定要四周都是水，这样的陆地才叫孤岛
先从四周出发，用 DFS 把“接触到边界的陆地”标记为访问过（这些不是闭合的）然后再在内部寻找没被访问过的陆地块（这些才是闭合岛屿）本质思路是先消掉开放的，再数剩下的封闭的
不再修改 grid 数组，只使用 visited 数组标记访问状态

首先把 1 变成 0: 地图四个边相连的陆地。然后再 找岛屿数量
在遍历地图周围四个边，靠地图四边的陆地，都为绿色，从边缘开始 DFS，把所有与边缘相连的陆地（岛屿）标记为水域



```
class Solution {
private:
    void dfs(vector<vector<int>>& grid, vector<vector<bool>>& visited, int x, int y) {
        int dir[4][2] = {{0,1}, {1,0}, {-1,0}, {0,-1}};
        for(int i = 0; i < 4; i++) {
            int nextx = x + dir[i][0];
            int nexty = y + dir[i][1];

            if(nextx < 0 || nextx >= grid.size() || nexty < 0 || nexty >= grid[0].size()) {
                continue;
            }
            if(grid[nextx][nexty] == 1 || visited[nextx][nexty]) {
```

```
                continue;
            }
            visited[nextx][nexty] = true;
            dfs(grid, visited, nextx, nexty);
        }
    }

public:
    int closedIsland(vector<vector<int>>& grid) {
        int count = 0;
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<bool>> visited(m, vector<bool>(n, false));

        // 处理四周的陆地
        for(int i = 0; i < m; i++) {
            if(grid[i][0] == 0 && !visited[i][0]) {
                visited[i][0] = true;
                dfs(grid, visited, i, 0);
            }
            if(grid[i][n-1] == 0 && !visited[i][n-1]) {
                visited[i][n-1] = true;
                dfs(grid, visited, i, n-1);
            }
        }
        for(int j = 0; j < n; j++) {
            if(grid[0][j] == 0 && !visited[0][j]) {
                visited[0][j] = true;
                dfs(grid, visited, 0, j);
            }
            if(grid[m-1][j] == 0 && !visited[m-1][j]) {
                visited[m-1][j] = true;
                dfs(grid, visited, m-1, j);
            }
        }

        // 统计封闭岛屿
        for(int i = 1; i < m-1; i++) {
            for(int j = 1; j < n-1; j++) {
                if(grid[i][j] == 0 && !visited[i][j]) {
                    visited[i][j] = true;
                    count++;
                    dfs(grid, visited, i, j);
                }
            }
        }
        return count;
    }
};
```

463. Island Perimeter 周长

数组：
数组在内存中的存储方式: 数组是存放在**连续内存空间（所以在删/增添元素的时候，要移动其他元素的地址）**上的相同类型数据的集合。
数组可以方便的通过下标索引的方式获取到下标对应的数据。（从 0 开始）
二维数组在内存的空间地址是连续的么？ 对于 C++语言，是的。对于 Java，否。

link

连续内存空间：数组需要连续的内存，链表不需要，因为它会指向下一个节点。

内存空间大小：链表节点 ListNode 还需要一份空间保存指针（引用），因此链表比数组占用更多的内存空间

python 代码：首先 初始化各个节点，然后 构建节点之间的引用。

```
n0 = ListNode(1)
n1 = ListNode(3)
n2 = ListNode(2)
n0.next = n1
n1.next = n2
```

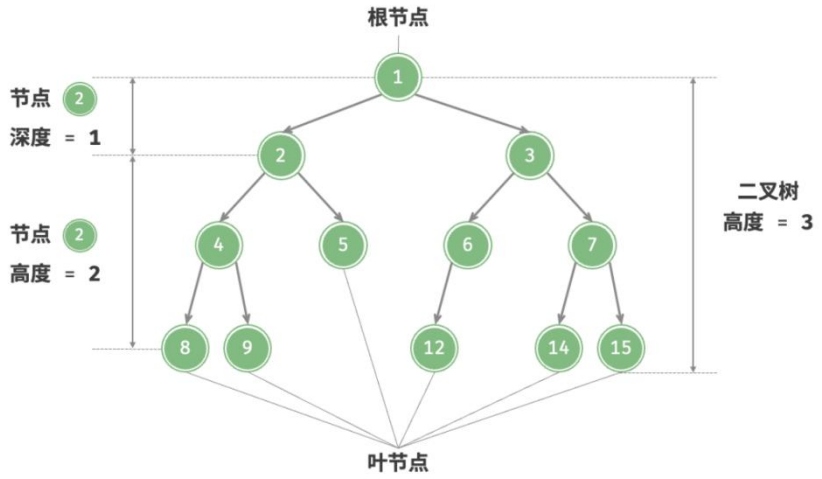
链表插入节点：空间复杂度 O(1)，拉踩数组 O(n)，所有后面的都要往后移动一位

```
P.next = n1
n0.next = P
```

链表删除节点：n0.next = n1

访问节点效率低，tmd 得从第一个开始报数，时间复杂度 O(n)，拉踩数组 O(1)

- 根节点（root node）：位于二叉树顶层的节点，没有父节点。
- 叶节点（leaf node）：没有子节点的节点，其两个指针均指向 `None` 。
- 边（edge）：连接两个节点的线段，即节点引用（指针）。
- 节点所在的层（level）：从顶至底递增，根节点所在层为 1 。
- 节点的度（degree）：节点的子节点的数量。在二叉树中，度的取值范围是 0、1、2 。
- 二叉树的高度（height）：从根节点到最远叶节点所经过的边的数量。
- 节点的深度（depth）：从根节点到该节点所经过的边的数量。
- 节点的高度（height）：从距离该节点最远的叶节点到该节点所经过的边的数量。



广度优先遍历 breadth-first traversal

深度 depth-first 先走到尽头，再回溯继续

1071. 字符串的最大公因子

```
class Solution:
    def gcdOfStrings(self, str1: str, str2: str) -> str:
        # WHY range(min(len(str1), len(str2)),0, 1)), NOT START FROM 1
        # [) range(), array[]
        # 从，长度上的最大值，开始
        for i in range(min(len(str1), len(str2)),0, -1):
            # 长度上也要既能除尽它，又能除尽它
            # 是%，不是//，%看能否除尽，是不是整数倍，长度是不是整数倍
            if(len(str1) % i == 0 and len(str2) % i == 0):
            # 取数  重复个数/次数
            # 想法先取 str1 各部分试，再 str2，但最后要找的 x 如果在 str2 存在，str1 必存在
            # x 的开始是固定的，一定是 str1 的第一个
            # so，这道题重点就是长度上的要求，然后从 str1 的 index=0 开始取就完事
                if(str1[:i] * (len(str1)//i) == str1 and str1[:i] * (len(str2)//i) == str2): # 每次对两个字符串
做拼接和比较，耗时 O(len1 + len2)
                    return str1[:i]
        return ''
```

1207. 独一无二的出现次数

```
# 每个数的出现次数都是独一无二的，而不是 每个数都是独一无二的
class Solution:
    def uniqueOccurrences(self, arr: List[int]) -> bool:
        num = Counter(arr).values() # count
        return len(set(num)) == len(num) # set 去掉了重复的元素后，长度还是不变，说明无重复
```

跟随代码随想录的顺序
暴力解法就是比如双层 for 循环，时间复杂度 O(n2)，

time
space

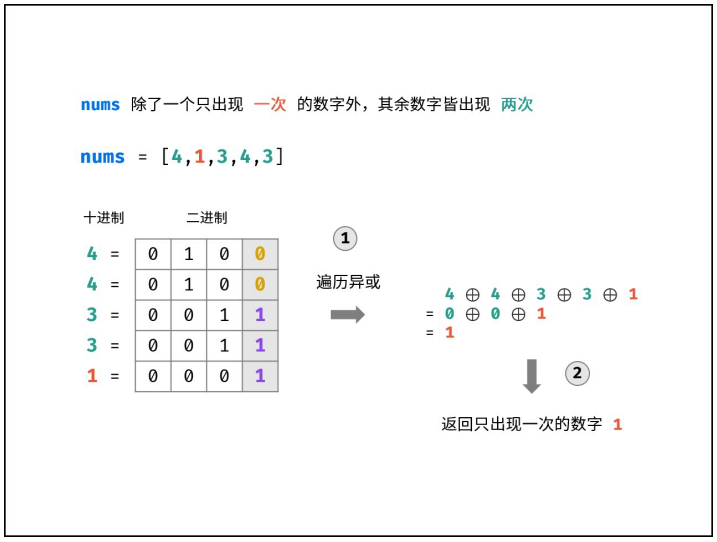
Misc

136. Single Number
所有数字都是成对出现，只有一个只出现一次，找出它
巧妙的解法，这世界还存在一种位运算叫异或运算 ^=，消消乐

$$nums = [a, a, b, b, ..., x]$$

异或运算有个重要的性质，两个相同数字异或为 0，即对于任意整数 a 有 $a \oplus a = 0$ 。因此，若将 $nums$ 中所有数字执行异或运算，留下的结果则为 **出现一次的数字** x ，即：

$$\begin{aligned} & a \oplus a \oplus b \oplus b \oplus ... \oplus x \\ = \ & 0 \oplus 0 \oplus ... \oplus x \\ = \ & x \end{aligned}$$




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