题型合集

排序

快速排序

- 将整个数组中的元素大致分成几类
- 核心思想是用一个指针遍历整个数组，再在需要填入数字的地方再做一个指针

将数组分成两类：

```
//假设数组nums中0和1交错排布，现在需要将0放在左边，1放在右边
int left = 0;//标记
for(int i=0;i<nums.size();i++){
    if(nums[i]==0){
        swap(nums[i],nums[left]);//需要注意这里交换完之后nums[i]是否需要再换一次（这里是不需要的-）。因为这段代码执行完之后i向右移了一位，有可能这里的nums[i]在后面处理不到了导致问题。
        left++;
    }
}
```

将数组分成三段：75. 颜色分类

这里需要开两个指针用于标记下一个元素应该放的位置，再用一个指针遍历整个数组

```
var sortColors = function(nums) {
    let n=nums.length;
    let left=0;
    let right=n-1;
    let i=0;
    while(i<=right){
        if(nums[i]==0){
            nums[i]=nums[left];
            nums[left]=0;
            left++;
        }else if(nums[i]==2){
                nums[i]=nums[right//注意到这里nums[right]->nums[i]的过程可能将一个2或0赋给了nums[i]，故还需要重新操作一遍这个i（后面的continue），不i++
                nums[right]=2;
                right--;
                continue;
            }

        i++;
    }
    return nums;
};
```

面向链表的模拟题（迭代、递归）

- 主要有迭代和递归两种写法，根据情况选择具体写法，貌似没有特别能笼盖所有情况的结论。以下举几个链表题的例子：

203. 移除链表元素：

- **递归写法：**

  递归的核心思想是将问题转化为子问题。对这道题，大问题为返回原链表删除所有值符合的节点后的结果，这个问题可以拆解为：对该大链表的头节点+后面的少一个节点的链表，若头节点==val，那么直接返回后面链表的处理结果；若头节点!=val，返回头节点->后面链表的处理结果。边界条件为对长度为1的链表直接返回它本身

```
ListNode* removeElements(ListNode* head, int val) {
    if(head==NULL){
        return head;
    }
    head->next=removeElements(head->next, val);
    return head->val==val ? head : head->next;
}
```

- 警告：每次调用递归的函数时候尽可能想一下只调一次，不要调两次！

  这里又差点这么写：

```
// Incorrect answer
ListNode* removeElements(ListNode* head, int val) {
    if(head==NULL){
        return head;
    }
    if(head->val==val){
        return removeElements(head->next, val);
    }else{
        head->next=removeElements(head->next, val);
        return head;//这样写调用了两次removeElements，浪费了很多
    }
}
```

- **迭代写法：**

  对于这道题来说不难，只要用一个travel节点遍历整条链表，把node->next->val==val的节点的next全部变为下一个节点的next即可

  注意到**头节点可能被删除，因此需要新建一个dummyHead哑节点**

```
//这里直接抄官方题解了
    ListNode* removeElements(ListNode* head, int val) {
        struct ListNode* dummyHead = new ListNode(0, head);//健一个哑节点，防止头节点丢失
        struct ListNode* temp = dummyHead;
        while (temp->next != NULL) {
            if (temp->next->val == val) {
                //若下一个节点应该删除，则直接指向下下个节点。注意下下个节点也可能被删除，故temp不移动到temp->next
                temp->next = temp->next->next;
            } else {
                temp = temp->next;
            }
        }
        return dummyHead->next;
    }
```

206. 反转链表：

- **递归写法：**对于大链表的子链表head->next，子链表反转之后，头节点应该插到反转后的子链表结尾。这里的写法是先存储一下temp=head->next（翻转之后该节点地址不会变化），然后temp变成了链表结尾，temp->next=head

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

- **迭代写法**：注意这里一个很容易的迭代写法是把所有节点压进栈里，然后一个个取出并连起来。但是这种方法空间为o(n)，我们应当找到空间复杂度为o(1)的写法

  迭代与递归的主要差别是迭代从头到尾，递归从尾到头。这里我们应该用节点cur表示正在遍历的节点，用prev表示节点前面已经翻转好了的链表头，next表示cur后面还未被遍历到的链表的头。

```
ListNode* reverseList(ListNode* head) {
    if(head==NULL||head->next==NULL){
        return head;
    }
    ListNode*cur=head;
    ListNode*prev=NULL;
    ListNode*next=NULL;
    while(cur!=NULL){
        next=cur->next;
        cur->next=prev;
        prev=cur;
        cur=next;
    }
    return prev;
}
```

判断环形链表

- 哈希表法（略）
- 双指针法（快慢指针）

  对于要返回循环起点的题目142. 环形链表 II，由官方题解，注意到从头节点head到循环起点的距离是快慢指针相遇的点到循环起点的距离（具体证明过程略），故可再初始化一个ans节点从head开始走，直到与慢指针相遇。代码如下：

```
var detectCycle = function(head) {
    let slow=head;
    let fast=head;
    while(fast!=null){
        slow=slow.next;
        fast=fast.next;
        if(fast==null){
            return null;
        }
        fast=fast.next;
        if(slow==fast){
            let ans=head;
            while(ans!=slow){
                ans=ans.next;
```

```
                slow=slow.next
            }
            return ans;
        }

    }
    return null;
};
```

双指针

- 双指针：涉及连续子数组时需要想到（e.g.最长的不出现重复字母的子串）

同向双指针

此类题特点：有**连续子数组**，且这个子数组的增长/减少一定会带来不利的影响，满足"只要[l,r]不满足某个要求，那么[l+1，r]也不满足/[l,r+1]也不满足"**(也即一定"单调性")**

1658. 将 x 减到 0 的最小操作数反向思维可知题目要数组中某个连续子序列为定值

209. 长度最小的子数组要某个连续子数组之和大于某个target

3. 无重复字符的最长子串要某个连续子数组之内不含相同元素

此类题特点：有**连续子数组**，且这个子数组的增长/减少一定会带来不利的影响，满足"只要[l,r]不满足某个要求，那么[l+1，r]也不满足/[l,r+1]也不满足"**(也即一定"单调性")**

- 模板：核心思想为向右追加->while(判断是否valid->更新答案->左边缩进)

```
/* 同向双指针算法框架 */
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
            ...
        }
    }
}
```

- 2023.9.23追加：不只是数组，只要是满足上述**“单调性”**，其他线性表如字符串也可以用该方法

典型的有如下这种判断子串包含某字符串所有字母的题：

76. 最小覆盖子串：如上算法模板即可。

```
class Solution {
public:
    string minWindow(string s, string t) {
        int ns=s.size();
        int nt=t.size();
        if(ns<nt){
            return "";
        }
        unordered_map<int,int>window, need;
        for(char ch:t){
            need[ch]++;
        }
        int left=0,right=0;
        int valid=0;
        int start=0,len=INT_MAX;
        while(right<ns){
            char c=s[right];
            right++;
            if(need.count(c)){
                window[c]++;
                if(need[c]==window[c]){
                    valid++;
                }
            }
            while(valid==need.size()){
                if(right-left<len){
                    start=left;
                    len=right-left;
                }
                char cc=s[left];
                left++;
                if(need.count(cc)){
                    window[cc]--;
                    if(window[cc]<need[cc]){
                        valid--;
                    }
                }
            }
        }
        return len==INT_MAX?"":s.substr(start,len);
    }
};
```

滑动窗口：特定长度连续子数组，提示很明显

2379. 得到 K 个黑块的最少涂色次数

相向双指针：

- 有顺序（或先行排序）

  表示的往往是分立的点而不是区间（可能是与同向双指针的区别？）

  分立的数之和（或差、积等）是定值而不是区间nums[i]+nums[j]==target

  （P.S.另一种情况：是区间，则用二分。如"lower<=nums[i]+nums[j]<=upper"->"nums[i]>=lower-nums[j]&&nums[i]<=upper-nums[j]"：6355. 统计公平数对的数目）

例：

167. 两数之和 II - 输入有序数组

15. 三数之和：对第一个数进行0->n-3遍历，然后另两个数等价于两数之和

P.S.本题另一个小细节：防止重复三元组的出现，三个指针每一个在移动时都要移动至和前一个数不一样的位置

- “接雨水”类题：前后缀分解+相向双指针

1237. 找出给定方程的正整数解：函数单调增故能用相向双指针

本题也可以用固定x=x0，用二分查找y的方法

前缀和与差分：多次算和用前缀和，多次加数用差分

- 前缀和的功用：

  将**字符串子串的运算（求和等）转化成两点之间的运算**，进而下一步处理一个**与哈希表结合的方法**见下

- 思路有点像前缀和的一题：1139. 最大的以 1 为边界的正方形：算每个点的左、上连续1的个数
- 异或和：同样也可以用前缀和

  S_n=a_0一直异或到a_n，则a_i到a_j的异或和为S_j^S_{n-1}

哈希表找到两个数运算为target（不用相向双指针的原因是不单调）（常配合前缀和使用）

- 简易题模板：面试题 17.05. 字母与数字：遍历原数组，对每个num[i]若能在表中找到target-num[i]，则找到了；否则将num[i]插入表中。这样的复杂度：o(n)
- 一类用前缀和+哈希表处理：常常处理**子数组内部所有数求值**问题，这里会用到前缀和
  - 面试题 17.05. 字母与数字：用前缀和算i位置前的字母数与数字数的差值。差值为0->字母与数字相等。此后，用哈希表找到前缀和相等的两个位置的距离最大值
  - 1590. 使数组和能被 P 整除：和上一题大同小异
  - 2488. 统计中位数为 K 的子数组:

- 2845. 统计趣味子数组的数目: 注意到这里配合了模运算，因此先将前缀和取模，本质大同小异

贪心

半贪心半dp的一种题：

每次讨论新增一个数对前面整体的影响

1798. 你能构造出连续值的最大数目

贪心+后悔堆

核心：拿完之后发现有更好的，从堆里面弹出来，拿更好的

630. 课程表 III:

首先，优先拿截止日期靠前的（这样一定不亏：因同样情况下，先拿时间靠后+再拿时间靠前够的话，先拿时间考前+再拿时间靠后一定时间也够；但反过来不一定成立）

其次，遍历所有course，若能拿一定拿，拿不了和已经拿的最长的比较，若新遍历到的时间短，则换一下

```
class Solution {
public:
    struct cmp{
        bool operator()(vector<int>& a, vector<int>& b){
            return a[1]<b[1];
        }
    };
    int scheduleCourse(vector<vector<int>>& courses) {
        sort(courses.begin(),courses.end(),cmp());
        priority_queue<int> pq;
        int day=0;
        for(auto&c: courses){
            int duration=c[0],last_day=c[1];
            if(day+duration<=last_day){
                day+=duration;
                pq.push(duration);
            }else if(!pq.empty()&&duration<pq.top()){
                day-=pq.top()-duration;
                pq.pop();
                pq.push(duration);
            }
        }
        return pq.size();
    }
};
```

871. 最低加油次数: 大同小异。先尽可能不加，如果不够了，再看前面哪里加油加的最多

```
class Solution {
public:
    int minRefuelStops(int target, int startFuel, vector<vector<int>>&
stations) {
```

```
        if(stations.size()==0){
            if(target>startFuel){
                return -1;
            }else{
                return 0;
            }
        }
        vector<int> add;
        add.push_back(target);
        add.push_back(0);
        stations.push_back(add);
        int ans=0;
        int currentFuel=startFuel;
        int currentDistance=0;
        priority_queue<int>pq;
        for(vector<int> station:stations){
            int this_distance=station[0]-currentDistance;
            if(this_distance>currentFuel){
                if(pq.empty()){
                    return -1;
                }
                while(!pq.empty()&&this_distance>currentFuel){
                    currentFuel+=pq.top();
                    pq.pop();
                    ans++;
                }
                if(this_distance>currentFuel){
                    return -1;
                }
            }
            currentFuel-=this_distance;
            pq.push(station[1]);
            currentDistance=station[0];
        }
        return ans;
    }
};
```

单调栈

- 单调栈模板：

496. 下一个更大元素 I：

一个形象的理解：后面的矮个被前面的高个“挡住了”，故从后向前遍历，遍历后入栈。若碰到一个“高个”（即大于后面的至少一个数），那么后面的矮个对这个高个前面的所有数就没有价值了，故弹出这些矮个，高个入栈。

```cpp
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> ans(nums.size()); // 存放答案的数组
    stack<int> s;
    for (int i = nums.size() - 1; i >= 0; i--) { // 倒着往栈里放
        while (!s.empty() && s.top() <= nums[i]) { // 判定个子高矮
            s.pop(); // 矮个起开，反正也被挡着了。。。
        }
        ans[i] = s.empty() ? -1 : s.top(); // 这个元素身后的第一个高个
        s.push(nums[i]); // 进队，接受之后的身高判定吧！
    }
    return ans;
}
```

另一种写法：

从前面i=0开始遍历，先比较该节点与栈顶的大小，遇到符合情况的更新答案并出栈；将该节点入栈；直到所有节点遍历完毕

这一点在1019. 链表中的下一个更大节点这题中体现，此题为链表，不能从后向前遍历

```python
def nextLargerNodes(self, head: Optional[ListNode]) -> List[int]:
    stk = list()
    ans = list()
    cur = head
    idx = -1
    while cur != None:
        idx+=1
        ans.append(0)
        while stk and stk[-1][1] < cur.val:
            ans[stk[-1][0]] = cur.val
            stk.pop()
        stk.append((idx,cur.val))
        cur = cur.next
    return ans
```

- 解决一段子数组的和>target的最长问题

先把子数组的和通过前缀和转化为两点之差，然后等价于上面的矮个高个，(左边的数+target<右边的数)

不同的是，这里需要找到右边最后一个“高个”而不是第一个“高个”，故模板不一样

1124. 表现良好的最长时间段：

理解一下：若j>i时sums[j]>sums[i]，则j不可能作为左侧数（i肯定比j赚），故j不入栈

再从右边遍历到左边找最大

```cpp
int longestWPI(vector<int>& hours) {
    int ans=0;
    stack<int> stk;
    int n = hours.size();
    vector<int> sums(n+1,0);
    stk.push(0);
    for(int i=0;i<n;i++)
    {
        if(hours[i]>8)
```

```
            sums[i+1]=sums[i]+1;
        else
            sums[i+1]=sums[i]-1;
        if(sums[i+1]<sums[stk.top()])
        {
            stk.push(i+1);
        }
    }
    for(int j=n;j>0;j--)
    {
        while(!stk.empty()&&sums[j]>sums[stk.top()])
        {
            ans = max(ans,j-stk.top());
            stk.pop();
        }
    }
    return ans;
}
```

这题看上去和常规的单调栈有一些不同，因这题是找最远距离而非最长距离，所以需要先从左边遍历入栈，再从右边遍历更新答案

单调队列

- 解决一段子数组的和>target的最短问题

862. 和至少为K的最短子数组:

为什么不能用同向双指针？数据流中有负数，不能确定每次指针移动对子数组的和是亏是赚

先算出前缀和，将子数组的和转化为分立的两点之差（通过这种方法也一定程度上解决了有负数的问题）

然后遍历前缀和数组sums，对sums[i]讨论，

若sums[i] - sums[deq.front()] >= target，则这可能是答案，故更新答案；且可能存在更优解，但以deq.front()开始的数组一定不可能是更优解了，故deq.pop_front()

若sums[i] <= sums[deq.back()]，则最优解不可能以deq.back()开头了（因为以i开头的一定更优），故deq.pop_back()

讨论完后将i插入队列

```
class Solution {
public:
    int shortestSubarray(vector<int>& nums, int k) {
        int n=nums.size();
        vector<long long> sums(n+1,0);
        for(int i=0;i<n;i++)
        {
            sums[i+1] = sums[i] + nums[i];
        }
        int ans = n+1;
        deque<int> que;
        for(int i=0;i<n+1;i++)
        {
            while(!que.empty()&&sums[i]-sums[que.front()]>=k)
```

```
            {
                ans = min(ans,i-que.front());
                que.pop_front();
            }
            while(!que.empty()&&sums[i]<=sums[que.back()])
            {
                que.pop_back();
            }
            que.push_back(i);
        }
        if(ans<n+1)
        return ans;
        else
        return -1;
    }
};
```

单调队列和同向双指针

209. 长度最小的子数组：

因为本题所有数均为正数，故当一段和>=target时，必然左指针右移是赚的，故可以用双指针做。用单调队列依然可行，但是空间复杂度会更高

有关优先队列

两个堆维护中位数：

295. 数据流的中位数：维护两个堆，一个大根堆负责维护数据中较小的一部分，一个小根堆负责维护数据中较大的一部分；对每个新进来的数，使其与两个堆堆顶的大小比较，并依次决定该数应该放到哪个堆中。若两个堆元素数量差值大于1，则将数量多的堆的堆顶放入数量少的堆。

```
class MedianFinder {
    priority_queue<int,vector<int>,less<int>> lower;
    priority_queue<int,vector<int>,greater<int>> upper;
public:
    MedianFinder() {
        ;
    }

    void addNum(int num) {
        if(lower.size()==upper.size()){
            if(lower.size()==0){
                lower.push(num);
                return;
            }
            if(num>=upper.top()){
                lower.push(upper.top());
                upper.pop();
                upper.push(num);
                return;
            }else{
                lower.push(num);
                return;
            }
```

```
        }else{
            if(num>lower.top()){
                upper.push(num);
                return;
            }
            upper.push(lower.top());
            lower.pop();
            lower.push(num);
            return;
        }
    }

    double findMedian() {
        if(lower.size()-upper.size()==1){
            return (double)lower.top();
        }
        return ((double)lower.top()+(double)upper.top())/2;
    }
};
```

配合哈希表实现“每次取最大+延后删除”：

2034. 股票价格波动：

可以将每天的价格与时间戳直接放进优先队列，同时更新哈希表。若堆顶记录与哈希表中记录不同，直接pop，直到与记录相同。

二分

原生二分：

- 题目特点：数组部分升降序/完全升降序，找一个特定需求的值
- 核心思想：红蓝染色，先找一个点，然后确定目标值在左侧/右侧，目标值不在的那一侧染色(目标值左点红色，目标值及其右点染成蓝色)

162. 寻找峰值：对于某组left和right，讨论mid位置的值相对所求值与left、right处的关系，据此更新left或right的值

153. 寻找旋转排序数组中的最小值：

思路一：（暂且不认为是自然的）mid值和数组最后一位比较，若大于则必然在mid右侧，小于则必然在mid左侧（易得），注意这里不能用双闭区间，因为mid值可能就是所要求的（双闭区间会将left更新为mid+1或将right更新为mid-1）

认为更自然：正常声明left、right，若[mid]>[left]，则left=mid+1，否则right=mid

原生二分的进阶：

二分答案：

- 如果题目中有「最大化最小值」或者「最小化最大值」，一般都是二分答案
- 二分答案的关键在于如何设计check函数

2517. 礼盒的最大甜蜜度：**（最大化最小值）**排序后（显然要排序），首先确定答案的范围一定是在(0, $\frac{price.back()-price[0]}{k-1}+1$)之间（即均匀选择每一个礼物），然后在这个区间里通过二分"寻找"答案。

然后在这个范围内进行二分，对于每次算出的mid都可以看比预期答案大/小，直到没有可二分的为止，即找到答案。

875. 爱吃香蕉的珂珂:**限制总时长求最小速度（最小化最大值）**

注意这里二分的left和right不要搞反了，这里$(总香蕉根数/速度)$是可能实现答案的最小值，最大值应为10^9.

思路不难，但有细节问题：

1. 注意speed不要有0的可能，带进函数会出错（这就对于二分的left设定有标准了，在left侧为闭区间的情况下，left不能为0）
2. 边界条件（本题为left）一定要想明白开/闭区间，想清楚能不能取到（本题有一个样例为piles=[2,2], h=2, 若left取total/h且为开区间就会出事）

码蹄集·BD202302·蛋糕划分: check函数的设计是关键！遍历横切，考虑让每一个格子都小于等于可能的ans的值，看竖切数够不够用

三个数运算暴力枚举的优化

982. 按位与为零的三元组：先将两个数按位与的值存在一个容器中，再第三个数与这个容器中所有数运算

状态压缩：

用一个二进制数表示一种状态，常用于【选或不选】这一思路中

这是一个技巧而非一种算法

快速幂：

对于指数n，底数a，o(log n)复杂度计算某数的幂

将n化成二进制，假设n=19，化为10011,则

$a^{19}=a^{2^0+2^1+2^4}=a\cdot a^2\cdot\{[(a^2)^2]^2\}^2$

故可用位运算的方法，用一数temp存储a的2^i次幂（i从1到n的二进制位数）逢位数为1时ans*=temp

```
double cal(double a,long long n)
{
    if(n==0)
    {
        return 1;
    }
    double ans = 1;
    double temp = a;
    while(n>0)
```

```
    {
        if(n&1)
        {
            ans*=temp;
        }
        temp *= temp;
        n = n>>1;
    }
    return ans;
}
double myPow(double x, int n) {
    long long N= n;
    if(N>=0)
    {
        return cal(x,N);
    }
    else
    {
        return 1.0/cal(x,-N);
    }
}
```

快速乘：与快速幂接近

```
int quick_add(int x,int y){//return x*y
    int res=0;
    while(x){
        if(x&1){
          res+=y;
        }
        x>>1;
    }
    return res;
}
```

递归

- 递归基本思路：

> 类似**数学归纳法**：
>
> 1.判断出口条件（如二叉树递归到最后的空节点）；这一点类似数学归纳法中k=1的情形
>
> 2.从一个问题到下一个子问题

100. 相同的树：

应该return的边界条件：某一棵树遍历到空节点时，看另一棵树这里是否为空来return

递归条件：两个对应节点不相同时return false，否则return子问题

- 恰当的时候可以在递归参数中加入层数：

199. 二叉树的右视图：

应该return的边界条件：遍历到NULL的节点

递归条件：从一个节点先调用右孩子再调用左孩子（因右视图先看到右孩子）

如果递归层数等于ans数组长度，将节点值放入ans

回溯

回溯三问：

> 1.当前操作？是枚举什么？
>
> 2.子问题？
>
> 3.下一个子问题？

78. 子集：回溯模板题，两个模板：

- 1.枚举nums中的每个数，看是否放进去（以输入的视角）

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans = []
        temp = []
        def dfs(i):
            if i == n:
                ans.append(temp.copy())
                return
            else:
                dfs(i+1) #nums[i]不放进去
                temp.append(nums[i])
                dfs(i+1) #nums[i]放进去
                temp.pop() #第i个数放进去所有情况讨论完了之后清空temp这一位，否则枚举下一个数的时候这个数还没清掉

        dfs(0)
        return ans
```

- 2.枚举放完nums[i]之后下一个放nums的哪一个（以答案的视角）

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans = []
        temp = []
        def dfs(i):
            ans.append(temp.copy())
            if i == n:
                return
            else:
                for j in range(i,n):
                    temp.append(nums[j])
                    dfs(j+1)
                    temp.pop()
                    # temp这个位置为nums[j]的所有情况讨论完后，要把这个位置清除，以便讨论该位置为其他数的情况

        dfs(0)
```

```
        return ans
```

排列型回溯

- 46. 全排列：排列型回溯模板题

递归参数用层数+还没枚举到的数的集合表示

```
ans = []
n = len(nums)
path = [0] * n

def dfs(i,s):  # s用来表示还没枚举到的所有数的集合
    if i == n: # 出口条件：所有数字都填进来了，即填了n个数字了
        ans.append(path.copy()) # 一定要复制副本！
        return
    for num in s:
        path[i] = num
        dfs(i+1,s-{num})

dfs(0,set(nums))
return ans
```

一定要复制副本path.copy()，而不是直接将path给append进去

另一种写法：不将还没列举到的数用集合放到递归参数中，而是作为全局变量显示哪种还没放进去

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        ans = []
        n = len(nums)
        check = [False] * n
        path = [0] * n
        def dfs(i):
            if i == n:
                ans.append(path.copy())
                return
            for j in range(n):
                if check[j] == False:
                    path[i] = nums[j]
                    check[j] = True
                    dfs(i+1)
                    check[j] = False
        dfs(0)
        return ans
```

- **2023.10.27更新：似乎有些生疏，需要复习了**
- 进阶：51. N 皇后：

基本原理：对于n*n棋盘要放n个皇后，必然有每行、每列各一个

那么对于以行号为索引的列表，必然是1~n-1的一个排列

如：

```
.Q.. # 第1列
...Q # 第3列
Q... # 第0列
..Q. # 第2列
```

对应：

```
[1,3,0,2]
```

故问题转化为排列型回溯

此外，还需验证一下对角线不能放置，这个在每行放的时候验证一下

- 递归入口：从第0行开始枚举
- 递归出口：当n行枚举完后，将答案放入ans中
- 从一个子问题到另一个子问题：放完第i行的后放第i+1行的，已经放置的列不能再放

```python
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        ans = []
        col = [0] * n
        check = [False] * n

        def valid(r,c): # 检验第r行，第c列能不能放（在前面的r-1行已经放好的情况下）
            for row in range(r):
                C = col[row]
                if row + C == c + r or row - C == r - c:
                    return False
            return True


        def dfs(i):
            if i == n:
                ans.append(['.'*c+'Q'+'.'*(n-c-1) for c in col]) # 生成答案
                return
            for j in range(n):
                if check[j] == False and valid(i,j): # 遍历所有没放过的列，并检验这一列能不能放了
                    col[i] = j
                    check[j] = True
                    dfs(i+1)
                    check[j] = False # 这种情况结束后记得清理现场

        dfs(0)
        return ans
```

组合型回溯

77. 组合：由于组合是任选其中几个数字但任意顺序均可的，故可指定要返回的每个集合都按从小到大排序（这样可以避免重复）

这样，从1开始枚举，决定每个数字"选"还是"不选"，选就放进path里。每当path长度达到k，就放进答案中。

```
var combine = function(n, k) {
    path=[];
    ans=[];
    function dfs(i){//i表示枚举到哪个数字了
        if(path.length==k){//如果长度为k，直接放进答案
            ans.push(path.concat());//.concat()是为了深拷贝path，与python的copy()类似
            return;
        }
        for(let j=i+1;j<=n;j++){//枚举下一个要选的数
            path.push(j);
            dfs(j);
            path.pop();
        }
    }
    dfs(0);
    return ans;
};
```

回溯+剪枝

> 一些问题不需要将所有情况回溯一遍，一些不可能实现的结果直接return掉

待施工

212. 单词搜索 II

记忆化搜索/动态规划

- DP萌新三步：
  - 思考回溯要怎么写
    - 首先是寻找子问题：在通过某个操作搞掉一小部分后，剩下的一个小一点的区段的处理方法应和原问题解决方法相同
    - 入参和返回值
    - 递归到哪里
    - 递归边界和入口
  - 改成记忆化搜索
  - 1:1翻译成递归

例1.198. 打家劫舍:

- 首先是考虑写成回溯：

  这里是子集型回溯，有【选或不选】【选哪个】这两种思路

  这里更适合用【选或不选】这个思路

  入参i应为前n个房屋待抢劫的数量

  边界条件应为：当i<0时，返回0（没有房子可以选了）

```
int dfs(int i)//选前i个房子的最大金额数量
{
    if(i<0)
    {
        return 0;
    }
    return max(dfs(i-1),dfs(i-2)+nums[i]);
}
```

- 优化为记忆化搜索：

```
int n=nums.size();
vector<int>cache(n,-1);//-1表示没被计算过
int dfs(int i)
{
    if(i<0)
    {
        return 0;
    }
    if(cache[i]!=-1)//以前已经计算过前i给的最大值
    {
        return cache[i];
    }
    int res=max(dfs(i-1),dfs(i-2)+nums[i]);
    cache[i]=res;
    return res;
}
```

- 逐步翻译为递推：

```
int n=nums.size();
vector<int>dp(n,-1);
dp[0]=nums[0];
dp[1]=nums[1];
for(int i=2;i<n;i++)
{
    dp[i]=max(dp[i-1],dp[i-2]+nums[i]);
}
return dp[n-1];
```

DP和递归的核心思想不同：递归是从后往前（从顶到底），DP是从前往后（从底到顶）

1105. 填充书架：

- 这里的子问题为，将几本书放到一层以后，剩下的所有书应该怎样排才能使总height最小（这与原问题结构是相同的）
- 从递归开始

```python
# 暴力递归，会超时
class Solution:
    def minHeightShelves(self, books: List[List[int]], shelf_width: int) ->
int:
        def dfs(i: int) -> int:
            if i < 0: return 0  # 没有书了，高度是 0
            res, max_h, left_w = inf, 0, shelf_width
            for j in range(i, -1, -1):
                left_w -= books[j][0]
                if left_w < 0: break  # 空间不足，无法放书
                max_h = max(max_h, books[j][1])  # 从 j 到 i 的最大高度
                res = min(res, dfs(j - 1) + max_h)
            return res
        return dfs(len(books) - 1)
```

- 翻译成递推：

  dp[i+1]表示到物品下标为i的最小书架高度

  这里状态转移方程有点特殊，遍历到物品i时，考虑将i与它前面的几件物品放到最后一层（直到这一层放不下为止），那么更新dp[i+1]为这一层物品的高度height+这些物品前面所有物品的高度最小值dp[j]

```python
class Solution:
    def minHeightShelves(self, books: List[List[int]], shelf_width: int) ->
int:
        n = len(books)
        f = [0] + [inf] * n  # 在前面插入一个状态表示 dfs(-1)=0
        for i in range(n):
            max_h, left_w = 0, shelf_width
            for j in range(i, -1, -1):
                left_w -= books[j][0]
                if left_w < 0: break  # 空间不足，无法放书
                max_h = max(max_h, books[j][1])  # 从 j 到 i 的最大高度
                f[i + 1] = min(f[i + 1], f[j] + max_h)
        return f[n]  # 翻译自 dfs(n-1)
```

- 自己写法：

```cpp
class Solution {
public:
    int minHeightShelves(vector<vector<int>>& books, int shelfWidth) {
        int n=books.size();
        vector<int> dp(n+1,1000000);
        dp[0]=0;
        for(int i=0;i<n;i++)
        {
            int cur_width=shelfWidth-books[i][0];
            int height=books[i][1];
            dp[i+1]=dp[i]+height;
            for(int j=i-1;j>=0;j--)
            {
                if(books[j][0]>cur_width)
                {
                    break;
```

```
                }
                cur_width-=books[j][0];
                height=max(height,books[j][1]);
                dp[i+1]=min(dp[i+1],dp[j]+height);
            }
        }
        return dp[n];
    }
};
```

通过此题，可以明白，dp[i]可以表示的含义不止有**“选了第i个”**（如上面一题），还可以有**“以第i个为某层结束”**等含义

背包DP：”选与不选“思想的代表

0-1背包

0-1背包原型：一定容量背包，一些物品，选或不选，总价值最大

从序号最后面的物品w[i]开始枚举

dfs(i,c)表示枚举到第i个物品时（也即当任意选取所有>=i的物品的状态）总容量为c的背包能装下的最大价值

- 出口条件：

```
if(i<0)//没有物品可供枚举了
{
    return 0;
}
```

- 状态转移方程：

```
dfs(i,c)=max(dfs(i-1,c),dfs(i-1,c-v[i])+w[i])
```

- 这里注意边界：c-v[i]可能小于0：当剩余容量小于v[i]时就没必要再算了，直接return dfs(i-1,c)
- 转化为递推：

```
//dp[i][c]表示当背包容量为c时，枚举到第i个物品时能够达到的最大价值
//状态转移方程：
dp[i][c]=max(dp[i-1][c],dp[i-1][c-w[i]]+v[i]);
//括号中前者是总容量为c，从前i-1个物品中选可以获得的最大价值（不选物品i）；后者是从前i-1个物品中总容量为c-w[i]的最大价值加上第i个物品的价值（选第i个物品）
```

- 变形：

> 至多装capacity，求方案数/最大价值和
>
> 恰好装capacity，求方案数/最大/最小价值和
>
> 至少装capacity，求方案数/最小价值和

- 一个变形实例：494. 目标和.

  原题可转化为从数组中选取一些数和为(target+sum(所有数之和)) / 2，故转化为0-1背包

用dp[i][c]表示枚举到第下标为i的数时和位c的方法数

- 入口条件：dp[0][0] = 1（一个不选，和为0是一种情况）
- 状态转移方程：dp[i][c] = dp[i-1][c](不选i) + dp[i][c-nums[i]](选i)

完全背包

- 出口条件：

```
if(i<0)//没有物品可供枚举了
{
    return 0;
}
```

- 状态转移方程：

```
dfs(i,c)=max(dfs(i-1,c),dfs(i,c-v[i])+w[i])//注意唯一的区别是括号里右侧i-1变成了i，这意味着枚举容量时可以把放过i物品后的背包继续放i物品
```

- 这里注意边界：c-v[i]可能小于0：当剩余容量小于v[i]时就没必要再算了，直接return dfs(i-1,c)
- 转化为递推：

```
//dp[i][c]表示当背包容量为c时，枚举到第i个物品时能够达到的最大价值
//状态转移方程：
dp[i][c]=max(dp[i-1][c],dp[i][c-w[i]]+v[i]);
//括号中前者是总容量为c，从前i-1个物品中选可以获得的最大价值（不选物品i）；后者是从前i个物品中总容量为c-w[i]的最大价值加上第i个物品的价值（选一次第i个物品）
```

- 完全背包实例：322. 零钱兑换.

完全背包的特点是：每一件货物不限次数拿取，求方案数/与方案数有关的东西

本题中，设置状态dp[i][j]表示枚举到硬币i时，凑出价值j的最小硬币数

- 状态转移方程：

```
dp[i][j]=Math.min(dp[i-1][j],dp[i][j-coins[i]]+1);
//表示可以选择一枚硬币i不拿，只拿i-1以及之前的（左边的等式）；也可以选择再拿一枚i（右边的等式）
```

代码：

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int m=coins.length;
        int[] dp1=new int[amount+1];
        int[] dp2=new int[amount+1];
        for(int j=0;j<=amount;j++){
            dp1[j]= j%coins[0]==0?j/coins[0]:10000000;
        }
        for(int i=1;i<m;i++){
            if(i%2==1){
                for(int j=0;j<=amount;j++){
```

```
                    if(j<coins[i])
                    {
                        dp2[j]=dp1[j];
                    }else{
                        dp2[j]=Math.min(dp1[j],dp2[j-coins[i]]+1);
                    }
                }
            }else{
                for(int j=0;j<=amount;j++){
                    if(j<coins[i]){
                        dp1[j]=dp2[j];
                    }else{
                        dp1[j]=Math.min(dp2[j],dp1[j-coins[i]]+1);
                    }
                }
            }
        }
        if(m%2==1){
            return dp1[amount]>=10000000?-1:dp1[amount];
        }else{
            return dp2[amount]>=10000000?-1:dp2[amount];
        }
    }
}
```

思考：正向DP和反向DP

有时，从正向递推到某处不能判断某个解是否是最优的（可以理解为，状态转移方程在i后面的情况未知时，无法推出；也即不满足“无后效性”）此时，可以考虑从反向递推

174. 地下城游戏:

本题如果从(0, 0)位置开始正向递推，由于遍历到格子(i,j)时，每条路后面的格子具体情况未知，无法判断哪个是最优解，也就无法进行每一步递推。这里从最后一个格子向前推，得出每个格子到最后一个格子所需的最大生命值，一步步推到(0,0)

```
class Solution {
public:
    int calculateMinimumHP(vector<vector<int>>& dungeon) {
        int n = dungeon.size(), m = dungeon[0].size();
        vector<vector<int>> dp(n + 1, vector<int>(m + 1, INT_MAX));
        dp[n][m - 1] = dp[n - 1][m] = 1;
        for (int i = n - 1; i >= 0; --i) {
            for (int j = m - 1; j >= 0; --j) {
                int minn = min(dp[i + 1][j], dp[i][j + 1]);
                dp[i][j] = max(minn - dungeon[i][j], 1);
            }
        }
        return dp[0][0];
    }
};
```

同样类型的还有：1690. 石子游戏 VII

思考：所给矩阵/数组各部分无差异的类型

2312. 卖木头块：注意这道题每个区块的木头是一样的，这意味着选择从(0,0)到(2,3)的木头块和选择从(10,0)到(12,3)的木头块没什么不同。因此dp[i][j]应该设计成高i宽j的木头块的最大价值，而非给的大木头块(0,0)到(i,j)的最大价值。（如果设计成后者复杂度会大很多）这样，状态转移方程就是自然的了

作为对比，打家劫舍/最长递增子序列这类题由于数组各部分不一样，必须按顺序从0到n进行dp

```python
class Solution:
    def sellingWood(self, m: int, n: int, prices: List[List[int]]) -> int:
        dic = dict()
        for price in prices:
            dic[price[0]*1000+price[1]]=price[2]
        dp = [[0 for __ in range(n+1)] for _ in range(m+1)]
        for _ in range(1,m+1):
            for __ in range(1,n+1):
                if _*1000+__ in dic:
                    dp[_][__]=dic[_*1000+__]
                for ___ in range(1,_):
                    dp[_][__]=max(dp[_][__],dp[___][__]+dp[_-___][__])
                for ___ in range(1,__):
                    dp[_][__]=max(dp[_][__],dp[_][___]+dp[_][__-___])
        return dp[m][n]
```

子序列问题

1027. 最长等差数列：

##########待施工##########

区间DP

区间DP和线性DP的区别：线性DP在前缀/后缀上转移；区间DP从小区间转移到大区间

母题：516. 最长回文子序列

状态的设计：用二维的空间表示一段数组，分别存储头和尾，dp[i][j]表示数组[i,j]区间内的最长回文子序列长度

状态转移方程：

```cpp
if(s[i]==s[j]){
    dp[i][j]=dp[i+1][j-1];//若s[i]和s[j]相等，必然两个都要取啊
}
else{
    dp[i][j]=max(dp[i][j-1],dp[i+1][j])//若s[i]和s[j]不相等，那么这两者只能取一个
}
```

注意枚举顺序！

注意到，这里更新dp[i]需要dp[i+1]的值，故i只能倒序枚举；更新dp[][j]需要dp[][j-1]，故j需要正序枚举

代码：

```
var longestPalindromeSubseq = function(s) {
    let n=s.length;
    let dp=[];
    for(let i=0;i<n;i++){
        dp[i]=[];
        for(let j=0;j<n;j++){
            dp[i][j]=0;
        }
    }

    for(let i=n-1;i>=0;i--){
        dp[i][i]=1;
        for(let j=i+1;j<n;j++){
            if(s[i]==s[j]){
                dp[i][j]=dp[i+1][j-1]+2;
            }else{
                dp[i][j]=Math.max(dp[i+1][j],dp[i][j-1]);
            }
        }
    }
    return dp[0][n-1];
};
```

引申题：1039. 多边形三角剖分的最低得分

状态的设计：dp[i][j]表示从i到j这些顶点组成的多边形剖分的最低得分

状态转移方程：遍历i到j之间的每一个节点k，将多边形划分为(i,j,k)组成的三角形+两侧的多边形。

```
dp[i][j]=INT_MAX;
for(int k=i+1;k<j;k++){
    dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]+values[i]*values[k]*values[j]);
}
```

注意到每次更新dp[i][j]时需要用到dp[i][k]和dp[k][j]，其中i<k<j，故i需要从后向前枚举，j需要从前往后枚举

区间DP和回文串

结论：用o(n^2)的复杂度可以计算出一个长字符串的每个子字符串是不是回文串

132. 分割回文串 II：先用上述结论将每个子串是否为回文串计算出，然后用线性dp从0到n正常递推即可

状态机DP(以股票问题为代表)

122. 买卖股票的最佳时机 II：（模板题）

- 对于状态[i, hold]表示遍历完前i天且第i天是否持有股票的最大利润
- 这里直接认为买到的股票是亏钱（如手上持有5块钱的股票认为亏5块钱）
- 若i<0则达到递归出口，由于一天都没有故利润为0.这里对于[-1, True]是不合法的，直接定义为-inf

```python
# 递归写法：
@cache
dfs(i, hold):
    if i < 0:
        return -inf if hold else 0 #若第-1天还hold则不合法
    if hold:
        return max(dfs(i-1, True), dfs(i-1, False) + prices[i])
    else:
        return max(dfs(i-1),True + prices[i], dfs(i-1, False))

# 迭代写法：
n = len(prices)
dp = [[0,0] for i in range(n+1)]
dp[0][0] = 0
dp[0][1] = -inf
for i in range(1,n+1):
    dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i-=])
    dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i-1])
```

- 股票问题的通用解法：

```
dp[i][k][0] //第i天 还可以交易k次 手中没有股票
dp[i][k][1] //第i天 还可以交易k次 手中有股票
```

```
dp[i][k][0] = max(dp[i-1][k][0],dp[i-1][k][1]+prices[i])
//今天没有持有股票，等于昨天没有持有股票今天不动和昨天持有股票今天卖了的最大值
dp[i][k][1] = max(dp[i-1][k][1],dp[i-1][k-1][0]-prices[i])
//今天持有股票，等于昨天持有股票今天不动和昨天没持有今天买了的最大值

//这里可以开两个二维数组而非三维数组
```

188. 买卖股票的最佳时机 IV：（含最多售卖次数的）

其他没什么太多可提，上面思路会了就会了。注意初始化dp[0][j][0]和dp[0][j][1](j>0)为无穷小（因为不可能实现）

换根DP

310. 最小高度树：

任找一节点作为根节点开始DFS，同时记录每个节点处的子树高度。这样，每经过一次相邻根节点的换根，新根的树的高度=max{前一个步骤算出的子树高度，上一个根节点的不经过新根的最大子树高度}，这样就可以在o(n)时间解决问题。

```cpp
class Solution {
public:
    void dfs1(vector<vector<int>>& graph, vector<int>&height0, int u){
        height0[u]=1;
        int h=0;
        for(int next:graph[u]){
            if(height0[next]!=0){
                continue;
            }
            dfs1(graph,height0,next);
```

```
            h=max(h,height0[next]);
        }
        height0[u]=h+1;
    }
    void dfs2(vector<vector<int>>& graph, vector<int>& height0,
vector<int>&height, int u){
        int first=0;
        int second=0;
        for(int next:graph[u]){
            if(height0[next]>first){
                second=first;
                first=height0[next];
            }else if(height0[next]>second){
                second=height0[next];
            }
        }
        height[u]=first+1;
        for(int next:graph[u]){
            if(height[next]!=0){
                continue;
            }
            height0[u]=(height0[next] != first?first:second)+1;
            dfs2(graph,height0,height,next);
        }
    }
    vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
        vector<vector<int>> graph(n);
        for(const auto&e:edges){
            graph[e[0]].push_back(e[1]);
            graph[e[1]].push_back(e[0]);
        }
        vector<int> height0(n,0);
        vector<int> height(n,0);
        dfs1(graph, height0, 0);
        dfs2(graph,height0,height,0);
        vector<int> ans;
        int h=n;
        for(int i=0;i<n;i++){
            if(height[i]<h){
                h=height[i];
                ans.clear();
            }
            if(height[i]==h){
                ans.push_back(i);
            }
        }
        return ans;
    }
};
```

834. 树中距离之和:

首先，任找一节点（为了方便一般选0节点）作为根节点算出其到每个节点的距离之和（复杂度o(n)），然后从该点开始进行换根DP。根节点每移动一次后，相对它之前的那个节点prev，在prev这一侧的每个节点的距离都加一，而反方向的每个节点的距离都减一

```
var sumOfDistancesInTree = function(n, edges) {
    let size=new Array(n).fill(1);
    let ans=new Array(n).fill(0);
    let tree = Array(n).fill(null).map(() => []); // tree[x] 表示 x 的所有邻居
    for (const [x, y] of edges) {
        tree[x].push(y);
        tree[y].push(x);
    }
    let dfs=function(i,fa,depth){//这一次dfs同时算出每个子树的节点个数（size）和节点0到各个节点的距离之和（用depth计算）
        ans[0]+=depth;
        for(const j of tree[i]){
            if(j!=fa){
                dfs(j,i,depth+1);
                size[i]+=size[j];
            }
        }
    };
    dfs(0,-1,0);
    let dfs2=function(i,fa){
        for(const j of tree[i]){
            if(j!=fa){

                ans[j]=ans[i]+n-size[j]*2;
                dfs2(j,i);
            }
        }
    }
    dfs2(0,-1);
    return ans;
};
```

动态规划题目积累：

1125. 最小的必要团队:

一个细节：可以状态压缩

确认回溯类型：子集型回溯，选与不选

以此开始动态规划，dp[i]表示职业技能为i（一个二进制数，表示现已有的状态）所需的最少人数的情况的所有人的集合

10. 正则表达式匹配:

构造状态、状态转移方程的方式比较特殊，属于非套路型题

- dp[i][j]表示s的前i位与p的前j位能否正则匹配，能为true，不能为false.为了便于dp，这里将s、p数组的第一位加一个空位（即所有字母往后移动一位）此时p[0]和s[0]想象为空，而后面每一位p[i]即是原来的p[i-1]，s一样

- 首先将dp[0][i]以及dp[i][0]的所有值初始化好（这个表示第一位空字符是否配对）
- i、j从1开始的状态转移方程：第一重循环i为枚举的每一个字母，第二重循环为遍历p的每一个字母。
    - 遍历到(i,j)位时，若s[i]能与p[j]配对（即字母相同或有'.'），则dp[i][j]=dp[i-1][j-1]
    - 若p[i]=='*'，则考虑两种情况：
        1. dp[i][j-2]==true，即p[j-1]和p[j]两位可以不用直接去掉，dp[i][j]=true
        2. p[j]能与s[i]配对（p[j]为'.'或与s[i]相同），则dp[i][j]=dp[i-1][j]（具体逻辑为：字母+'*'的组合相比于遍历到i-1又重复了一次，故为true）
    - 如此遍历即可
- 答案为dp[s.length][p.length]
- （2023.10.7第二次更新）注意两点：1. 一定记得预留s、p前面的空位，且注意p开头是字母+'*'的组合，和空位是可以匹配的；2. 因为dp数组是从前往后更新的，当遍历到p[j]='*'时，只需要考虑dp[i][j-2]这一个就行了，不需要从头再遍历一遍
- 类似题目：44. 通配符匹配

视频：视频图解 动态规划 正则表达式 - 正则表达式匹配 - 力扣（LeetCode）

1388. 3n 块披萨：

注意到可以转化为环形数组不相邻选n个数，后面和打家劫舍接近，略。

1031. 两个非重叠子数组的最大和：

前缀和+DP，这题是讨论以i分别为first的后缀和second的后缀，再考虑前i-first（或i-second）个数取second（或first）子数组的最大值。这样就可以实现o(n)

```
public int maxSumTwoNoOverlap(int[] nums, int firstLen, int secondLen) {
    int[] dp1 = new int[nums.length + 1]; // 以firstLen为长度，子数组的最大值
    int[] dp2 = new int[nums.length + 1]; // 以secondLen为长度，子数组的最大值
    int[] dp3 = new int[nums.length + 1]; // 答案dp
    int[] preSum = new int[nums.length + 1]; // 数组的前缀和
    for (int i = 1; i <= nums.length; i++) {
        preSum[i] = preSum[i - 1] + nums[i - 1];
        // 分别计算前缀子数组的最大值
        // 分成三种情况，1.firstLen后缀+dp2   2.secondLen后缀+dp1  3.最大值不包含nums[i]
        if (i>=firstLen){
            dp1[i] = Math.max(dp1[i - 1], preSum[i] - preSum[i - firstLen]);
        }
        if(i>=secondLen) {
            dp2[i] = Math.max(dp2[i - 1], preSum[i] - preSum[i - secondLen]);
        }
        if(i>=firstLen+secondLen){
            dp3[i] = Math.max(preSum[i] - preSum[i - secondLen] + dp1[i - secondLen], dp3[i]);
            dp3[i] = Math.max(preSum[i] - preSum[i - firstLen] + dp2[i - firstLen], dp3[i]);
            dp3[i] = Math.max(dp3[i], dp3[i - 1]);
        }
    }
    return dp3[nums.length];
```

```
}
```

1696. 跳跃游戏 VI：或许可以代表状态转移方程为`dp[i]=dp前k项的最值+nums[i]`这一类的动态规划问题

可以用一个priority_queue配合滑动窗口思想维护前k项的最值

```
int maxResult(vector<int>& nums, int k) {
    int n=nums.size();
    priority_queue<pair<int,int>> pq;//pair<得分,下标>
    vector<int> dp(n,0);
    dp[0]=nums[0];
    pq.push(make_pair(dp[0],0));
    for(int i=1;i<n;i++){
        auto best=pq.top();
        while(best.second<i-k){
            pq.pop();
            best=pq.top();
        }
        dp[i]=nums[i]+best.first;
        pq.push(make_pair(dp[i],i));
    }
    return dp[n-1];
}
```

字符串相关

Z函数（扩展KMP）

- **基础用法**：计算字符串`str[0,n)`中间每个点i（0<i<n）起始的与该字符串前缀重合的字符串长度

如：`z("aaabaab")=[0,2,1,0,2,1,0]`

实现：易见朴素的做法复杂度是O(n^2)的

- 这里提供O(n)的做法：

1. 初始化l,r，从1至n-1遍历整个字符串，维护l和r为当前遍历到的位置中，r最大的、与前缀重合的子串下标

   如上述`"aaabaab"`，当枚举i==4结束时，l=4，r=5

2. 枚举i时，如果i在`[l,r]`的范围内，由于`[l,r]`与`[0,r-l]`是一样的，故`[i,r]`段与前缀重合的部分必与`[i-l,r-l]`相同，那么现在获取`z[i-l]`，如果`z[i-l]<r-i+1`，可直接获取`z[i]=z[i-l]`而不用一个个比较；如果`z[i-l]>=r-i+1`，也会有`[i,r]`段和`[i-l,r-l]`相同，这一段也不需要重复比较了。直接比较r后面的即可，如果r后面还有与前缀重合的，更新l和r。

3. 枚举i时，如果i不在`[l,r]`范围内，则需要从i开始与前缀比较。如果有与前缀重合区间记得更新l和r

- **code**：

```
vector<int> z_function(string& s){
```

```
    int n=s.length();
    vector<int> z(n,0);
    int l=0,r=0;
    for(int i=1;i<n;i++){
        if(i<=r&&z[i-l]<r-i+1){
            z[i]=z[i-l];
        }else{
            z[i] = max(0, r - i + 1);//看i后面是否有之前已经算的和前缀重合的部分
            while (i + z[i] < n && s[z[i]] == s[i + z[i]])//枚举r之后的部分
            {
                ++z[i];
            }
            if(i+z[i]-1>r){//如果比较到r之后的值了，更新r
                r=i+z[i]-1;
                l=i;
            }
        }
    }
    return z;
}
```

- 时空复杂度均为`O(n)`

例题：[将单词恢复初始状态所需的最短时间 II]

注意到，该题等价于需要找到按题示操作t次后，剩余字符串（没被前面砍掉的）刚好为原字符串的前缀，答案为t. 如果一直到原字符串全被移除都找不到满足上述要求的前缀，答案即为移除原字符串的操作次数（因后面的可以自己任意加）。明白这点，解题是自然的

图

建图：

一般不需要使用过于复杂的数据结构，开二维数组就行

有向图找环（拓扑排序）

[802. 找到最终的安全状态]:

- 方法1：三色法，将没被遍历过的节点设为0（白色），不安全的（在环上的）或在递归栈中的节点设为1（灰色），已经确认安全的节点为黑色。那么从一个节点开始DFS，把路径标为灰色，碰到灰色则此DFS路径不安全，碰到黑色则安全。那么一个节点的安全与否可以转化为多个子问题：它所有的出边指向的节点是否安全？

```
dfs(i):
    if color[i]==1:
        return False
    elif color[i]==2:
        return True # 这几行是边界条件，即该节点已经被遍历过
    color[i]=1      # 若该节点之前没被遍历，现在遍历到了，染灰
    for j in graph[i]:
        if dfs(j) == False:
            return False # 找到一条路不安全
    color[i]=2 # 所有路都安全，记得把这个节点的颜色改掉
    return True
```

（补注*递归思路：若本身为1（不安全）或2（安全），直接返回；将自己设置为1（表明已经经过了）；遍历所有子路，有一条不安全就不安全，全部安全后回溯到该节点（该节点所有路都遍历完了），该节点变为安全）

方法大同小异：207. 课程表

拓扑排序(Topological Sort)：

- 能进行拓扑排序的**充要条件：是一个有向无环图**

拓扑排序原生用法：确立顺序

1. 判断是否能进行拓扑排序

同上：207. 课程表

2. 给出一个拓扑排序

210. 课程表 II: 给出拓扑排序的模板题，有结合DFS和BFS两种思路

结合DFS思路：先从某个指定节点往下搜索，全部搜索完成后，在“归”的环节将这个节点放入路径中。用vis数组标明某个节点已经遍历/正在遍历队列中，辅助dfs

```
class Solution {
    vector<int>ans;
    vector<vector<int>>grid;
    vector<int>vis;
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>&
prerequisites) {
        grid.resize(numCourses);
        vis.resize(numCourses,0);
        for(vector<int> ver:prerequisites){
            grid[ver[0]].push_back(ver[1]);
        }
        for(int i=0;i<numCourses;i++){
            if(vis[i]==0){
                bool res=dfs(i);
                if(!res)
                {
                    return vector<int>(0);
                }
            }
        }
        return ans;
    }
    bool dfs(int i){
        if(vis[i]==2){
            return true;//遍历到安全节点，这个节点肯定已经加入答案中，直接返回
        }
        if(vis[i]==1){
            return false;//如果这个节点在遍历过程中又遇到了自己，说明存在环，答案不存在
        }
        vis[i]=1;
        for(int next:grid[i]){
            if(!dfs(next)){
                return false;
```

```
            }
        }
        vis[i]=2;
        ans.push_back(i);//在“归”的环节中将节点压入答案队列
        return true;
    }
};
```

也有一种结合栈的DFS写法，这里给出（直接抄官方题解了）：

```
class Solution {
private:
    // 存储有向图
    vector<vector<int>> edges;
    // 存储每个节点的入度
    vector<int> indeg;
    // 存储答案
    vector<int> result;

public:
    vector<int> findOrder(int numCourses, vector<vector<int>>&
prerequisites) {
        edges.resize(numCourses);
        indeg.resize(numCourses);
        for (const auto& info: prerequisites) {
            edges[info[1]].push_back(info[0]);
            ++indeg[info[0]];
        }

        queue<int> q;
        // 将所有入度为 0 的节点放入队列中
        for (int i = 0; i < numCourses; ++i) {
            if (indeg[i] == 0) {
                q.push(i);
            }
        }

        while (!q.empty()) {
            // 从队首取出一个节点
            int u = q.front();
            q.pop();
            // 放入答案中
            result.push_back(u);
            for (int v: edges[u]) {
                --indeg[v];
                // 如果相邻节点 v 的入度为 0，就可以选 v 对应的课程了
                if (indeg[v] == 0) {
                    q.push(v);
                }
            }
        }

        if (result.size() != numCourses) {
            return {};
        }
```

```
        return result;
    }
};
```

结合BFS思路：这个用队列实现的迭代是和递归是反过来的，迭代必须从没有入度的点开始，而递归理论上最好从“众矢之的”的点开始（在上面的DFS递归中，其实从任意点开始也行）

用队列的核心思路是：将所有入度为0的节点入列，然后开始遍历每个队列头节点的下一层节点，去掉它们的一个入度（即把前面那些入度为0的节点删去），再把入度为0的节点放入队列中，如此往复。从队列中弹出的序列就是最终答案

```
class Solution {
private:
    // 存储有向图
    vector<vector<int>> edges;
    // 存储每个节点的入度
    vector<int> indeg;
    // 存储答案
    vector<int> result;

public:
    vector<int> findOrder(int numCourses, vector<vector<int>>&
prerequisites) {
        edges.resize(numCourses);
        indeg.resize(numCourses);
        for (const auto& info: prerequisites) {
            edges[info[1]].push_back(info[0]);
            ++indeg[info[0]];
        }

        queue<int> q;
        // 将所有入度为 0 的节点放入队列中
        for (int i = 0; i < numCourses; ++i) {
            if (indeg[i] == 0) {
                q.push(i);
            }
        }

        while (!q.empty()) {
            // 从队首取出一个节点
            int u = q.front();
            q.pop();
            // 放入答案中
            result.push_back(u);
            for (int v: edges[u]) {
                --indeg[v];
                // 如果相邻节点 v 的入度为 0，就可以选 v 对应的课程了
                if (indeg[v] == 0) {
                    q.push(v);
                }
            }
        }

        if (result.size() != numCourses) {
            return {};
```

```
        }
        return result;
    }
};
```

3. 判断前导节点：

模板题:1462. 课程表 IV

这题与上面不同，需要找出每个节点的前导节点。因此仅给出一个可行的拓扑序列的解是不能解决问题的。需要判断某a是否在某b前面。注意，如果a和b无关（比如都不需要前导课程），那么对query=[a,b]应当给出false，但如果只给出了一个可行的拓扑排序结果，那么a与b一定有先后顺序，这并不是我们想要的。

因此，这题（似乎）只能采用从“基础课程”（没有入度的点）逐渐向“高阶课程”（要求前导课程比较多的课程）迭代的方式（因为不一定能找到一个能统领全局，居于最高地位的课程，没有这样的课的话从顶到下的递归无法进行）。注意到，每从一个基础课cur->进阶课next的转化中，必有isPre[cur][next]==1；其次，cur的所有前导课pre必有isPre[pre][next]==1。因此，每枚举一个next，都要对所有课程进行扫描以根据它是否与cur产生关系退出它是否与next有关系。

代码：

```
class Solution {
public:
    vector<bool> checkIfPrerequisite(int numCourses, vector<vector<int>>& prerequisites, vector<vector<int>>& queries) {
        vector<vector<int>> grid(numCourses);
        queue<int> que;
        vector<int> inputs(numCourses,0);
        vector<vector<int>> isPre(numCourses,vector<int>(numCourses,0));
        for(vector<int>pre: prerequisites){
            grid[pre[0]].push_back(pre[1]);
            inputs[pre[1]]++;
        }
        for(int i=0;i<numCourses;i++){
            if(inputs[i]==0){
                que.push(i);
            }
        }
        while(!que.empty()){
            int cur=que.front();
            que.pop();
            for(int next:grid[cur]){
                isPre[cur][next]=1;
                for(int i=0;i<numCourses;i++){
                    isPre[i][next]=isPre[i][next]|isPre[i][cur];
                }
                --inputs[next];
                if(inputs[next]==0){
                    que.push(next);
                }
            }
        }
        vector<bool>res;
        for(auto& query:queries){
            res.push_back(isPre[query[0]][query[1]]);
```

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

拓扑排序另一用法：四周包围中心，逐渐收紧，可看作“反向BFS”，也常用来判断最小路径

310. 最小高度树：从叶子节点（度为1的）开始删除，删去一圈叶子节点后，中心的部分又变成一颗小一点的树。最终收缩到最中间的节点后，该节点就是要求的根节点，高度即为收缩的厚度加上和中间相连的部分

```python
class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) ->
List[int]:
        in_degree, connect = [0] * n, defaultdict(list)
        for a, b in edges:
            in_degree[a] += 1
            in_degree[b] += 1
            connect[a].append(b)
            connect[b].append(a)
        nodes = [i for i, v in enumerate(in_degree) if v <= 1]
        while n > 2:
            n -= len(nodes)
            nxt = []
            for node in nodes:
                for other in connect[node]:
                    in_degree[other] -= 1
                    if in_degree[other] == 1:
                        nxt.append(other)
            nodes = nxt
        return nodes
```

内向基环树

- 概要：一个有n个节点的有向图，每个节点均有且仅有一条出边指向另一节点，那么这个图一定由若干环+指向环上某节点的枝叶组成。
- 处理思想：用一次拓扑排序剪掉枝叶（经过一次拓扑排序后，所有枝叶入度变为0，环上入度全为1），便于后续处理；同时建立反图，便于从环上到枝叶的遍历

有向图访问计数：先用拓扑排序区分枝叶和环；再用反图遍历环，有叶子时即遍历到叶子上

```cpp
class Solution {
public:
    vector<int> countVisitedNodes(vector<int>& edges) {
        int n=edges.size();
        vector<vector<int>> rg(n);
        vector<int> deg(n,0);
        for(int i=0;i<n;i++){
            rg[edges[i]].push_back(i);
            deg[edges[i]]++;
        }
        queue<int> que;
        for(int i=0;i<n;i++){
```

```
            if(deg[i]==0){
                que.push(i);
            }
        }
        while(!que.empty()){
            int cur=que.front();que.pop();
            if(--deg[edges[cur]]==0){
                que.push(edges[cur]);
            }
        }
        vector<int> ans(n,0);
        function<void(int,int)> rdfs=[&](int x, int depth){
            ans[x]=depth;
            for(int next:rg[x]){
                if(deg[next]==0){
                    rdfs(next,depth+1);
                }
            }
        };
        for(int i=0;i<n;i++){
            if(deg[i]==1){
                vector<int> ring;
                for(int x=i;;x=edges[x]){
                    ring.push_back(x);deg[x]=-1;
                    if(edges[x]==i){
                        break;
                    }
                }
                for(int node:ring){
                    rdfs(node,ring.size());
                }
            }
        }
        return ans;
    }
};
```

Tarjan算法：

- Tarjan算法核心：
  - 用DFS跑一张无向图
  - 按DFS的遍历顺序记录所有节点的“时间戳”，即遍历每个节点的时间顺序，放入dfn数组中
  - 遍历的过程中，若某节点的邻居是之前遍历过的节点（且不是它的父亲），那么这个节点的low数组对应值变为它这个邻居的时间戳。同时，它所有（一路遍历过来的）父亲节点的low值也变为该值。
  - 若两个相邻节点x，y满足low[x]>dfn[y]，说明到达x节点必须经过y，故这是一座“桥”

1192. 查找集群内的关键连接: **Tarjan算法模板题**

```
class Solution {
    vector<vector<int>> grid;//图
```

```
    vector<vector<int>> ans;
    vector<int> dfn;//时间戳
    vector<int> low;
    vector<int> vis;
    int cur;
public:
    void tarjan(int now, int fa){
        vis[now]=1;
        dfn[now]=cur;
        low[now]=cur++;
        for(int next:grid[now]){
            if(next==fa){
                continue;
            }
            if(!vis[next]){
                tarjan(next,now);
                low[now]=min(low[now],low[next]);
                if(dfn[now]<low[next]){//因next时间戳大于now，故不可能dfn[next]<low[now]，所以只考虑一种情况
                    ans.push_back({now,next});
                }
            }else{
                low[now]=min(low[now],dfn[next]);//若又遍历到一个已经经过的节点，这条边不可能是答案（因为来时的路证明可以有另一条通路），故不更新答案
            }
        }
    }
    vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
        this->cur=0;
        grid.resize(n);
        dfn.resize(n);
        low.resize(n);
        vis.resize(n,0);
        for(vector<int>con:connections){
            grid[con[0]].push_back(con[1]);
            grid[con[1]].push_back(con[0]);
        }
        tarjan(0,-1);
        return ans;
    }
};
```

最短路相关

Dijkstra

定义略

例题：1976. 到达目的地的方案数.

思路是以0为起点跑一个`dijkstra`，同时记录到每个点的最短路径数，路径数的记录有一定记忆化搜索的思维（每算出一个最短路径的点都会把它 的邻居更新）

（略匆忙，直接抄题解了，注意priority_queue即便是复杂的pair也可以用greater<pair<long long, int>>，另外注意emplace的用法）

```cpp
class Solution {
public:
    using LL = long long;
    int countPaths(int n, vector<vector<int>>& roads) {
        const long long mod = 1e9 + 7;
        vector<vector<pair<int, int>>> e(n);
        for (const auto& road : roads) {
            int x = road[0], y = road[1], t = road[2];
            e[x].emplace_back(y, t);
            e[y].emplace_back(x, t);
        }
        vector<long long> dis(n, LLONG_MAX);
        vector<long long> ways(n);

        priority_queue<pair<LL, int>, vector<pair<LL, int>>,
greater<pair<LL, int>>> q;//注意这里
        q.emplace(0, 0);//也注意这里
        dis[0] = 0;
        ways[0] = 1;

        while (!q.empty()) {
            auto [t, u] = q.top();
            q.pop();
            if (t > dis[u]) {
                continue;
            }
            for (auto &[v, w] : e[u]) {
                if (t + w < dis[v]) {
                    dis[v] = t + w;
                    ways[v] = ways[u];
                    q.emplace(t + w, v);//这里
                } else if (t + w == dis[v]) {
                    ways[v] = (ways[u] + ways[v]) % mod;
                }
            }
        }
        return ways[n - 1];
    }
};
```

DFS

二叉树DFS模板：

- 迭代：

```
初始化栈;
while(栈非空||结点非空)
{
    while(结点非空)
    {
        stk.push(head);
```

```
        head=head->left;
    }
    head=stk.top();
    stk.pop();
    head=head->right;
}
//此方法可以将所有节点过一遍
//在合适的位置进行适当操作
```

- 递归：

```
void dfs(tree*node)
{
    /***************/
    if(node==NULL)
    {
        return;
    }
    /***************/
    dfs(tree->left);
    dfs(tree->right);
}
```

图DFS模板：

- 递归：

```
vector<vector<int>> map;//map[i]中装了所有i节点的下一个节点
vector<int> vis;//存储是否被遍历过，vis[i]==0--遍历过；vis[i]==1--没遍历过
void dfs(int cur)
{
    vis[cur]=0;
    for(int next:map[cur])
    {
        if(!vis[next])
        {
            //do something
            dfs(next);
        }
    }
}
```

- 另一种写法：（不用vis数组而是用father参数避免无限循环）

```
vector<vector<int>> map;//map[i]中装了所有i节点的下一个节点
void dfs(int cur,int fa){
    for(int next:map[cur]){
        if(next!=fa){
            //do something
            dfs(next,cur);
        }
    }
}
dfs(0,-1)//从编号为0的节点开始，-1表示它没有父节点
```

补充：在图中，将图转化为树，算树高（不是二叉树的处理方法）

```
void dfs1(vector<vector<int>>& graph, vector<int>& height0, int u) {
    height0[u] = 1;
    int h = 0;
    for (int v : graph[u]) {
        if (height0[v] != 0) continue;
        dfs1(graph, height0, v);
        h = max(h, height0[v]);
    }
    height0[u] = h + 1;
}
```

连通集个数：

547. 省份数量：常规bfs就行，**用一个数组记录哪些节点被遍历过**，那么这些节点就不需要再遍历了

200. 岛屿数量：同理，只不过这里用数组表示节点。**将已经遍历过的地点换一个数（比如-1）**，就不需要再遍历了

BFS:

- **BFS模板：**

```
//假设一共n个节点
vector<vector<int>> grid;//每个数组i记录它的所有相邻节点
queue<int> que;
vector<int> vis(n,0);
que.push(i);//假设以节点i为中心作BFS
while(!que.empty()){
    int cur=que.front();
    que.pop();
    vis[cur]=1;
    //do something
    for(int next:grid[cur]){
        if(!vis[next]){
            que.push(next);
```

```
        }
    }
}
```

BFS提醒：千万要记得开vis数组或者其他方法保证不重复遍历！否则超时！

BFS常用于发掘一条最短的路径

这最好别拿递归实现！就用队列！

递归虽然可以参数加一个depth，但实际运行还是一个枝一个枝跑的（和DFS一样），有些情况会出问题

- 二维数组中的BFS模板题：1926. 迷宫中离入口最近的出口 不要拿dfs的递归来写！遍历过的直接变成墙避免重复遍历（用DFS写的话，复杂的枝可能把简单的路堵上）

```
int nearestExit(vector<vector<char>>& maze, vector<int>& entrance) {
        int m = maze.size();
        int n = maze[0].size();
        vector<int> dx = {1, 0, -1, 0};
        vector<int> dy = {0, 1, 0, -1};
        queue<tuple<int, int, int>> q;
        q.emplace(entrance[0], entrance[1], 0);
        maze[entrance[0]][entrance[1]] = '+';
        while (!q.empty()){
            auto [cx, cy, d] = q.front();
            q.pop();
            for (int k = 0; k < 4; ++k){
                int nx = cx + dx[k];
                int ny = cy + dy[k];

                if (nx >= 0 && nx < m && ny >= 0 && ny < n && maze[nx][ny] == '.'){
                    if (nx == 0 || nx == m - 1 || ny == 0 || ny == n - 1){
                        return d + 1;
                    }
                    maze[nx][ny] = '+';
                    q.emplace(nx, ny, d + 1);
                }
            }
        }
        return -1;
    }
```

关于此代码的声明：

变体可以有：

1. 拿dis数组装最短路，que每次push的数组就不用记录距离状态了，vis数组也不需要了

注意：不要忘记队列的pop，不要忘记剪掉已经vis过的状态！

补充题：

码蹄集·BD202301·公园: 找到两个人一起走的最短路径，这里需要对目的地、两个人分别为中心点做一次BFS，从而找到每个点距离这三者的距离。最后遍历每个点，通过这个点距离三者的距离算出以这个点为交点的结果，并更新答案。

BFS寻找带状态的最短路

864. 获取所有钥匙的最短路径：这题的特殊之处在于存在有钥匙和无钥匙两种状态，能到达的范围不一样。因此对每个格子位置应当设置所有的状态（状态总数为2^(场上钥匙数)），某些状态下的移动不能经过某些锁。其余思路不变。

一种BFS的常见变体：字符变换

127. 单词接龙：这题关键在于如何构建两个单词之间“可以一步转化”。如果暴力枚举两个单词则需$o(n^2 \cdot c)$时间（c为字符长度），超时。这里很巧妙地将每个字符串的每一个字符均变为'*'一次，将每个这样的新字符串放入图中，便构建了中间节点，用$o(n \cdot c)$的复杂度建图。之后常规BFS即可。

```
class Solution {
    private Map<String, Integer> wordId;
    private List<List<Integer>> grid;
    private int nodeNum;
    private void addEdge(String str){
        addWord(str);
        int id1=wordId.get(str);
        char[] charArray=str.toCharArray();
        for(int i=0;i<str.length();i++){
            char temp=charArray[i];
            charArray[i]='*';
            String targetArray=new String(charArray);
            addWord(targetArray);
            int id2=wordId.get(targetArray);
            grid.get(id1).add(id2);
            grid.get(id2).add(id1);
            charArray[i]=temp;
        }
    }
    private void addWord(String word){
        if(!wordId.containsKey(word)){
            wordId.put(word, nodeNum++);
            grid.add(new ArrayList<Integer>());
        }
    }
    public int ladderLength(String beginWord, String endWord, List<String>
wordList) {
        int n=wordList.size();
        nodeNum=0;
        this.grid = new ArrayList<List<Integer>>();
        this.wordId = new HashMap<String, Integer>();
        addEdge(beginWord);
        for(int i=0;i<n;i++){
            addEdge(wordList.get(i));
        }
        if(!wordId.containsKey(endWord)){
            return 0;
        }
        Queue<Integer> que=new LinkedList<Integer>();
        List<Integer> dis=new ArrayList<Integer>();
        List<Boolean> vis=new ArrayList<Boolean>();
        for(int i=0;i<nodeNum;i++){
            dis.add(0);
```

```
            vis.add(false);
        }
        dis.set(0,0);
        que.offer(wordId.get(beginWord));
        while(!que.isEmpty()){
            int cur=que.peek();que.poll();
            vis.set(cur,true);
            int dist=dis.get(cur);
            for(int next:grid.get(cur)){
                if(!vis.get(next)){
                    dis.set(next,dist+1);
                    que.offer(next);
                }
            }
        }
        if(!vis.get(wordId.get(endWord))){
            return 0;
        }
        return dis.get(wordId.get(endWord))/2+1;
    }
}
```

2023.10.21更新：BFS的一个弊端

在非探究最短路问题中，有时会拿BFS来遍历整个图。这时如果每次出队列时才更新vis数组的状态，可能会导致同一层的两个节点后面都连接某个点node，这样由于vis[node]在找这两个节点的下一层的for循环中均为false，故node会两次进队列，造成多余。

2316. 统计无向图中无法互相到达点

树

前序、中序、后序遍历

##########待施工#########

还原二叉树

106. 从中序与后序遍历序列构造二叉树:

题目给出了中序和后序遍历。注意到后序遍历的顺序是“左->右->根”，那么如果倒序遍历后序遍历数组postorder，得到的顺序是“根->右->左”，这个从根节点出发的顺序才是我们更希望看到的（显然，我们构造这棵树需要从根节点逐步往下递归来写）。但这时我们无法确定后序遍历的倒序从哪里开始“拐弯”，即“根->右”转为“右->左”（换一种理解，无法确定哪里的儿子是NULL）。这时需要用中序遍历来规约某一棵子树的“范围”。若后序遍历倒序枚举到一个节点，这个节点在中序遍历的右侧已经没有节点可枚举了，那么说明这个节点已经处于这颗子树的最右侧，应该向左拐弯了。

补充一句：（个人看法）这道题整体是跟着后序遍历的倒序枚举在画这棵树，然后用中序遍历来约束这棵树哪里的枝条被阻拦（NULL）

```
class Solution {
    int post_pointer;
    unordered_map<int,int> reflect;
public:
```

```
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int n=inorder.size();
        post_pointer=n-1;
        for(int i=0;i<n;i++){
            reflect[inorder[i]]=i;
        }
        return helper(0,n-1,inorder,postorder);
    }
    TreeNode* helper(int left,int right,vector<int>& inorder, vector<int>&
postorder){
        if(right<left){
            return NULL;
        }
        int node_val=postorder[post_pointer];
        int index=reflect[node_val];
        TreeNode* root=new TreeNode(node_val);
        post_pointer--;
        root->right=helper(index+1,right,inorder,postorder);
        root->left=helper(left,index-1,inorder,postorder);
        return root;
    }
};
```

449. 序列化和反序列化二叉搜索树: 和上一题接近，不同点在于这题是二叉搜索树，根据节点值即可确定下一个节点到底“拐”不“拐”弯。

```
public class Codec {
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        List<Integer> list=new ArrayList<Integer>();
        postOrder(root,list);
        String str=list.toString();
        return str.substring(1,str.length()-1);
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data.isEmpty()){
            return null;
        }
        String[] arr=data.split(", ");
        Deque<Integer> stack=new ArrayDeque<Integer>();
        int length=arr.length;
        for(int i=0;i<length;i++){
            stack.push(Integer.parseInt(arr[i]));
        }
        return construct(Integer.MIN_VALUE, Integer.MAX_VALUE, stack);
    }
    public void postOrder(TreeNode root, List<Integer> list){
        if(root==null){
            return;
        }
        postOrder(root.left, list);
        postOrder(root.right, list);
        list.add(root.val);
```

```
    }
        private TreeNode construct(int lower, int upper, Deque<Integer>
stack) {
         if (stack.isEmpty() || stack.peek() < lower || stack.peek() > upper)
{
             return null;
         }
         int val = stack.pop();
         TreeNode root = new TreeNode(val);
         root.right = construct(val, upper, stack);
         root.left = construct(lower, val, stack);
         return root;
    }
}
```

两个序列还原二叉树更深层的思考

一个序列数组用来当“画笔”，另一个序列数组用于“调整方向”，或称之“约束范围”

用`TreeNode* helper(int left, int right)`来画出用于约束范围的那个数组在[left, right]区间内应有的树

核心问题就是两个，1.如何确定当前范围内[left,right]树的根节点，2. 如何确定左右子树在“约束范围”数组内分别对应的两个子范围

这两个问题都解决了，就可以递归了。

先解决第一个问题。

注意到两个序列中前序和后序至少有一个，那么我们在有前序时取前序做“画笔”，无前序时取“后序”做画笔。这两者的区别就是先从后遍历（后序）数组还是从前遍历（前序）数组的区别

在整个过程开始前先初始化指针p指向当前遍历到的位置。那么当前序做”画笔“时p置为0，后序做画笔时p置为`preorder.length()-1`

这样整棵树根节点就很好确定了。我们知道前序第一个元素是根节点，后序最后一个元素是根节点。其实在后面枚举子树时依然保持p指向的是根节点。

第二个问题：

首先明确一点，取左右子树在“约束范围”数组中的意义更多是为了确定“画笔”数组需要在这颗子树上操作多少次，即p会移动几次。left和right起到最大的作用就是right-left的值的大小（这也决定了这颗子树有多大，p需要移动多少次），而画笔定位则靠p来实现

这里如果“约束范围”数组是中序遍历，很容易根据上述根节点的值在中序数组中找到下标index，将数组[left,right]分成两半，即[left,index-1]和[index+1,right]。这里需要的树的左子树就成了helper(left,index-1)，右子树成了helper(index+1,right)。而这里这些left、right、index依然对画笔不起定位作用，而是靠p来定位。

注意这里：

```
res->left=helper(left,index-1);
res->right=helper(index+1,right);
```

在res左子树的绘制中，p已经移动了index-left次，因此刚好会移动到res右子树的根结点处，刚好进入下一行的递归。因此p不需要什么特别的调整

这里也能发现，当后序遍历作为“画笔”数组时，需要先画右子树再画左子树，而每画一个节点p减一。

树、图特殊题目积累

979. 在二叉树中分配硬币：

注意到，任取一条边将树分成两个部分，则通过这条边的硬币数必须使得两个部分的硬币数和其节点数相等。移动次数等于所有边的通过数。这样通过树形DP就很容易算了

```
var distributeCoins = function(root) {
    let n=0;
    function dfs(r){
        if(r==null){
            return;
        }
        n++;
        dfs(r.left);
        dfs(r.right);
    };
    dfs(root);
    let ans=0;
    function helper(r){
        if(r==null){
            return [0,0];//nodeNums, coinNums
        }
        let lef=helper(r.left);
        let rig=helper(r.right);
        ans+=Math.abs(lef[0]-lef[1]);
        ans+=Math.abs(rig[0]-rig[1]);
        return [lef[0]+rig[0]+1,lef[1]+rig[1]+r.val];
    };
    helper(root);
    return ans;
};
```

并查集

- 大致思路：将一个大集合分堆，用一个链表指明某个头元素的根节点在哪里，根节点相同的元素属于同一堆

并查集模板：

```
class UnionFind{
    vector<int> parent;
    UnionFind(int n){
        parent.resize(n);
        for(int i=0;i<n;i++){
            parent[i]=i;
        }
    }
    int union_set(int index1,int index2){//将index1对应的集合和index2对应的集合合并
```

```
        parent[find(index2)]=parent[find(index1)];//这里为什么用find? 可能index2或index1的直系父亲还没有被更新为根节点；同时保证index2的所有祖先都指向根节点,index1的所有祖先都指向根节点
    }
    int find(int index){
        if(parent[index]!=index){
            parent[index]=find(parent[index]);//递归实现一直向上找，并将每个路过的节点的parent设为对应的根节点
        }
        return parent[index];
    }
};
```

721. 账户合并：**并查集模板题**

首先遍历每一个账户，每一个账户内先合并成一个小并查集；再遍历所有邮箱，find每个邮箱的根，把还没连完全的连好；最后把所有根相同的放到一个数组里，排序

```
class UnionFind {
public:
    vector<int> parent;

    UnionFind(int n){
        parent.resize(n);
        for(int i=0;i<n;i++){
            parent[i]=i;
        }
    }
    void UnionSet(int index1,int index2){//将1的根赋给2的根->2合并到1下面
        parent[find(index2)]=find(index1);
    }
    int find(int index){//找index的根，同时将所有路径上经过的点的根都标好
        if(parent[index]!=index){
            parent[index]=find(parent[index]);
        }
        return parent[index];
    }
};

class Solution {

public:
    vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
        map<string,int> email_index;
        map<string,string> email_name;
        int index=0;
        for(auto account:accounts){
            string name=account[0];
            int n=account.size();
            for(int i=1;i<n;i++){
                string email=account[i];
                if(!email_index.count(email)){
                    email_index[email]=index++;
                    email_name[email]=name;
                }
```

```
            }
        }
        UnionFind uf(index);
        for(auto account: accounts){
            string firstEmail=account[1];
            int firstIndex=email_index[firstEmail];
            int size=account.size();
            for(int i=2;i<size;i++){
                string nextEmail=account[i];
                int nextIndex=email_index[nextEmail];
                uf.UnionSet(firstIndex,nextIndex);
            }
        }
        map<int,vector<string>> index_emails;
        for(auto& [email,_]:email_index){
            int index=uf.find(email_index[email]);
            vector<string>& account=index_emails[index];
            account.push_back(email);
            index_emails[index]=account;
        }
        vector<vector<string>> merged;
        for(auto& [_,emails]:index_emails){
            sort(emails.begin(),emails.end());
            string& name=email_name[emails[0]];
            vector<string> account;
            account.push_back(name);
            for(auto email:emails){
                account.push_back(email);
            }
            merged.push_back(account);
        }
        return merged;
    }
};
```

2382. 删除操作后的最大子段和：不典型的并查集题目

一个额外要点：注意可以逆向思维从数组反向开始加

注意到这点后，就是常规的并查集题目。注意到这里的union_set函数不用写，每个节点i的直系父亲可以设为它右边的节点i+1，故其根节点就是它右边的节点i+1的根节点to；合并完成后再将sums[to]更新为sums[to]+sums[i]+nums[i]。这里sums[i]就不会用到了，因为根已经变成to了，故不用更新sums[i]

```
class Solution {
public:
    vector<long long> maximumSegmentSum(vector<int> &nums, vector<int>
&removeQueries) {
        int n = nums.size();
        int fa[n + 1];
        iota(fa, fa + n + 1, 0);
        long long sum[n + 1];
        memset(sum, 0, sizeof(sum));
        function<int(int)> find = [&](int x) -> int { return fa[x] == x ? x
: fa[x] = find(fa[x]); };
```

```
10
11          vector<long long> ans(n);
12          for (int i = n - 1; i > 0; --i) {
13              int x = removeQueries[i];
14              int to = find(x + 1);
15              fa[x] = to; // 合并 x 和 x+1
16              sum[to] += sum[x] + nums[x];
17              ans[i - 1] = max(ans[i], sum[to]);
18          }
19          return ans;
20      }
21  };
22  //(抄自灵神题解)
```

priority_queue相关问题

单调非减、非负整数数组的第k小的子序列和

思路：注意从小到大的顺序，依次是`nums[0]`, `nums[1]`, `min(nums[0]+nums[1], nums[2])`, `max(nums[0]+nums[1], nums[2])`, ...

注意到，对每个当前的最小值(val, i)`(val代表当前值，i代表这个子序列的最后一个数为nums[i])`，放入`(val+nums[i+1], i+1)`, `(val+nums[i+1],i+1)`，那么下一个最小值一定会在优先队列里，这样就可以进行递推更新了。

```
1  vector<int> nums;//单调非减、非负整数数组
2  int k;//要求第k小的子序列和
3  if(k==1){
4      return 0;//每个元素都不拿也是一种子序列
5  }
6  priority_queue<pair<int,int>> pq;
7  pq.emplace(nums[0],0);
8  for(int i=1;i<k;i++){
9      auto cur=pq.top();pq.pop();
10     if(i==k-1){
11         return cur.first;
12     }
13     int nex=cur.second+1;
14     if(nex>=n){
15         continue;//避免数组越界情况
16     }
17     pq.push(cur.first+nums[nex],nex);
18     pq.push(cur.first+nums[nex]-nums[nex-1],nex);
19 }
```

案例：2386. 找出数组的第 K 大和.

最大矩形面积问题：常转化为高度数组

85. 最大矩形：将每个坐标处的从之开始的向上的连续1的个数存到数组里，那么每一行的数据就可以转化为84题（高度数组求最大矩形）的做法，用单调栈即可

位运算

异或性质：偶数个相同为0，奇数个相同得自己

136. 只出现一次的数字

- 几道暂时不能归类的题：

260. 只出现一次的数字 III：分组异或，每组各有一个目标值和其他成对的值

137. 只出现一次的数字 II：每个位模三，算出目标值在该位上是0还是1

数学

算最大公约数

```
int gcd(int x,int y){
    return y?gcd(y,x%y):x;
}
```

思路特殊的题目积累：

（周赛332）6356. 子字符串异或查询：暴力把字符串能表示的所有二进制数（答案能取到的）全放进容器里，然后无脑查找

56. 合并区间：根据左界排序

1163. 按字典序排在最后的子串：字典序最大的子串结尾一定在原字符串末尾

979. 在二叉树中分配硬币：DFS，注意答案是每棵子树的硬币树和节点数之差绝对值的和

41. 缺失的第一个正数：只允许常数级别额外空间，原地哈希

常见约束（C++版）

1.爆int（注意两个int相加再赋值给long long也会爆，需要先强制类型转换）

2.数组越界，判断条件里加

3.数组长度很短的时候需要先在最开始return一下结果

4.很多遍历图记得考虑如何区分已遍历/未遍历！不然很容易爆内存/时间！