# 一、链表

1. 二叉树层序遍历：

<https://blog.csdn.net/weixin_42109012/article/details/92250160?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param>

层序遍历只需要用队列queue即可，每次访问节点后将左右儿子push到queue里面去。这里有个特殊的点就是需要记录每一层的节点的数量，然后每次需要一个for循环访问整层的数据

1. 单链表的反转

<https://blog.csdn.net/bladeandmaster88/article/details/52396795?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-5.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-5.channel_param>

注意最终的链表的末尾节点的next要附为NULL，否则其指向不明，会报错

1. 链表的节点的加入写法

ListNode \*head;

ListNode \*p=head;//游动指针

ListNode \*node=new ListNode(1);

p->next=node;

p=node;

p=NULL;

#### 4. [删除排序链表中的重复元素 II](https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list-ii/)

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现 的数字。

**示例 1:**

**输入:** 1->2->3->3->4->4->5

**输出:** 1->2->5

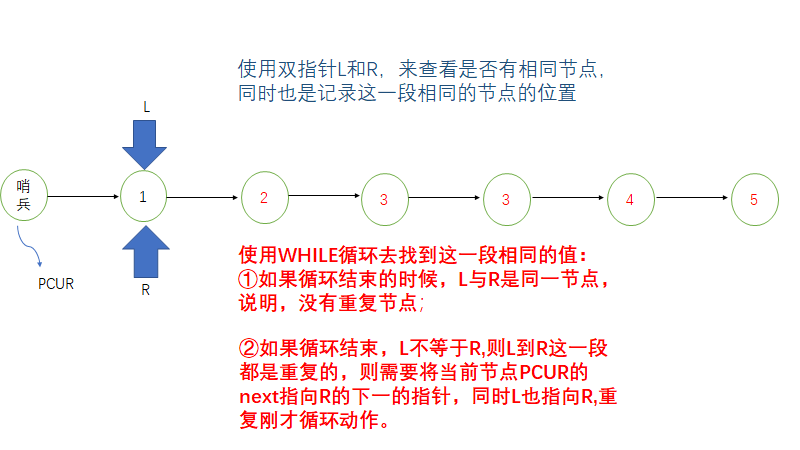
**示例 2:**

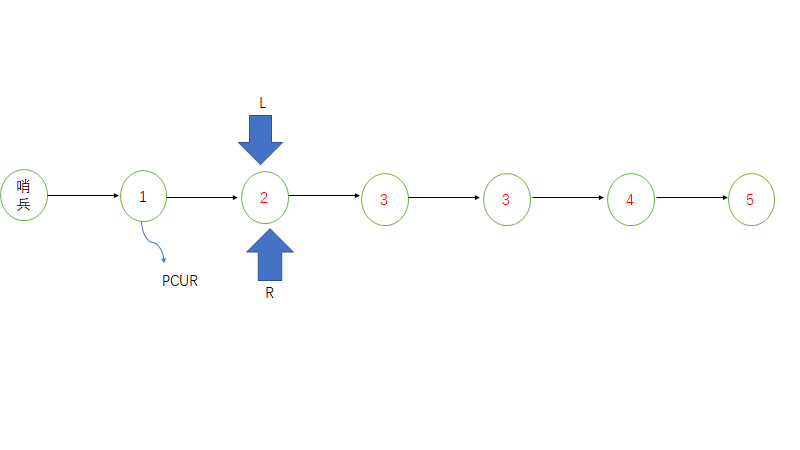
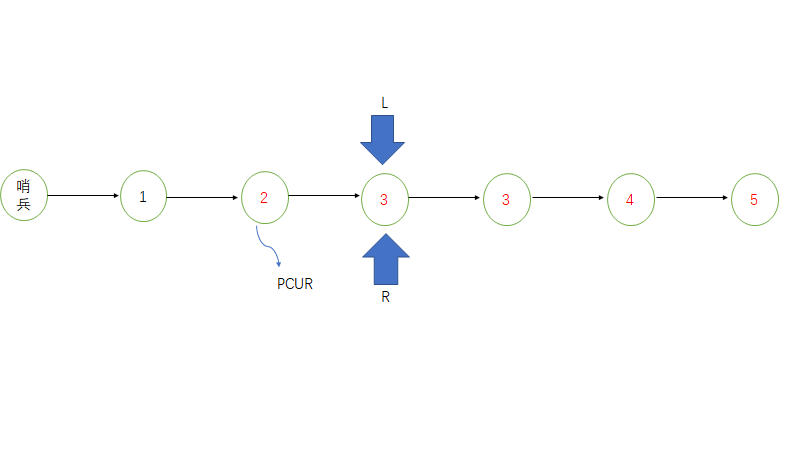
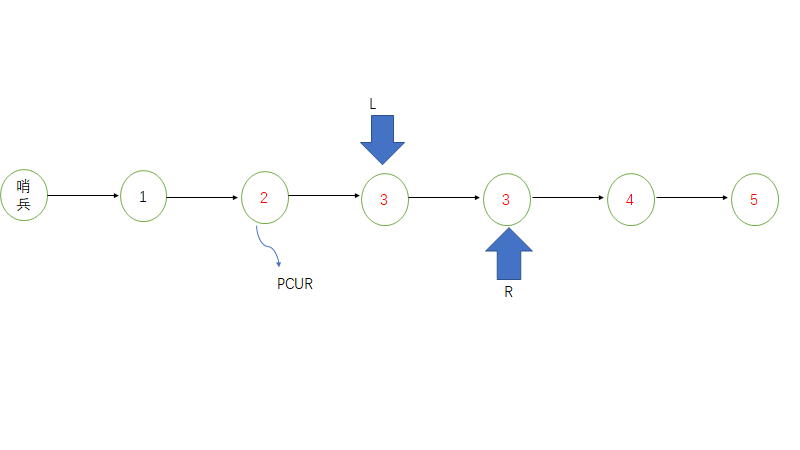
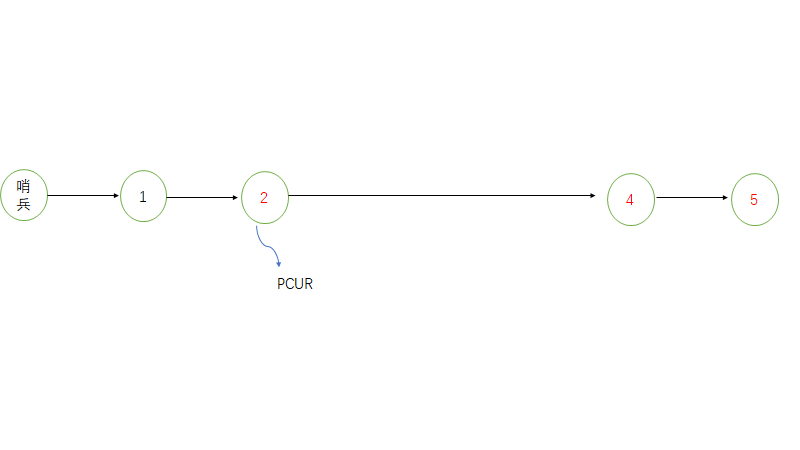
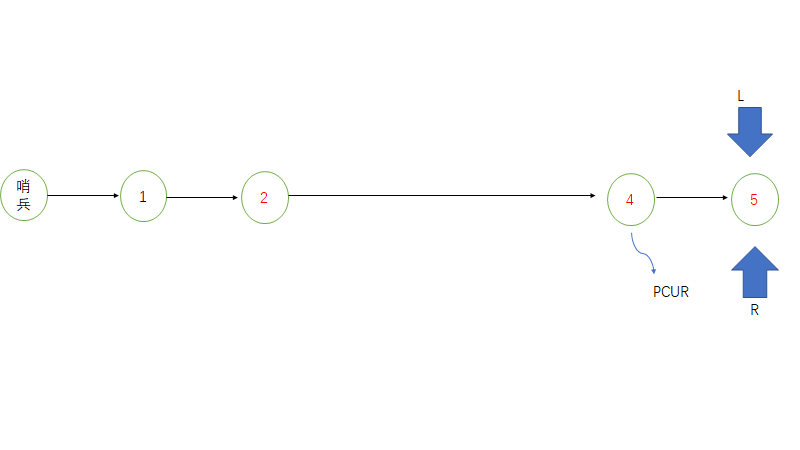
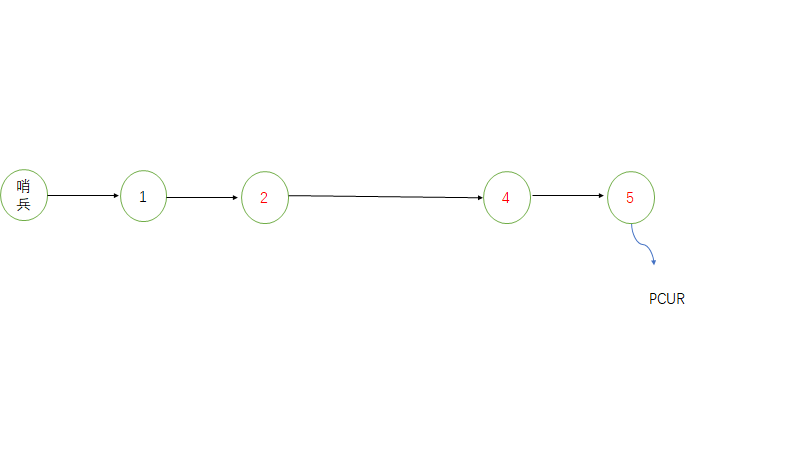
**输入:** 1->1->1->2->3

**输出:** 2->3

法一）为了考虑到头结点为空，或者所有节点值相同时最终结果为空，引入哑结点。主要就是两层嵌套while循环，外层负责循环完整个链表，内层负责找这个节点后面相邻的值相等的节点（当前节点一定与挨着的节点不相同），然后如果值相同的节点数n>1，则先删除后面n-1个节点，然后判断第n个节点是否为尾结点，需要特殊处理。这个方法太慢，速度在后10%

法二）三指针法



# 二、数组

1. 给定一个可能包含重复元素的整数数组 ***nums***，返回该数组所有可能的子集（幂集）。

**说明：**解集不能包含重复的子集。

法一）用map

用map可以很方便地求出每个值在集合中出现的次数。

map<int,int>num;

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

            num[nums[i]]++;  //map的int初始值为0，键值对对应的就是键在nums中出现的次数

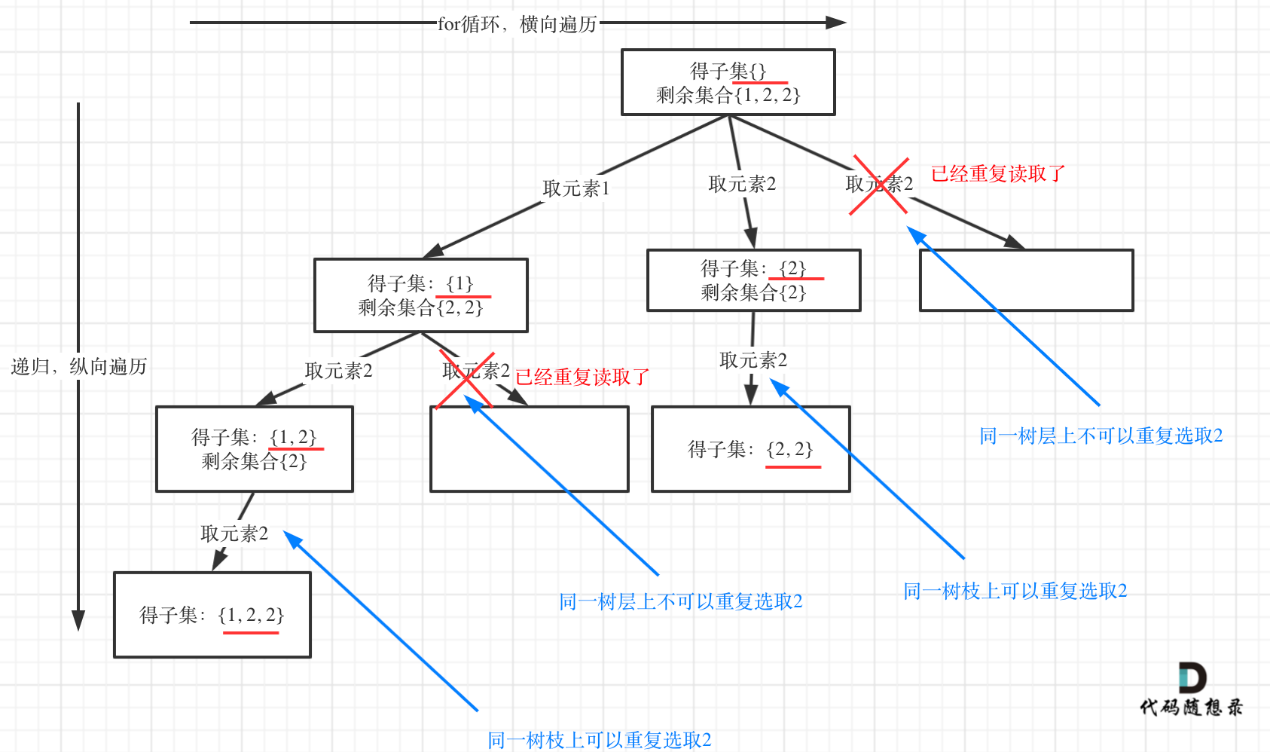
而不同的集合只要满足每个元素在现有已存的集合中出现的次数分别是[1,n]次，n代表这个元素在集合中的频数。

所谓去重，其实就是使用过的元素不能重复选取。

都知道组合问题可以抽象为树形结构，那么“使用过”在这个树形结构上是有两个维度的，一个维度是同一树枝上使用过，一个维度是同一树层上使用过。没有理解这两个层面上的“使用过” 是造成大家没有彻底理解去重的根本原因。

所以要明确我们要去重的是同一树层上的“使用过”。

用示例中的[1, 2, 2] 来举例，如图所示：

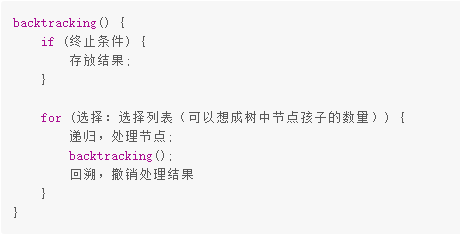


从图中可以看出，同一树层上对重复取2 就要过滤掉，同一树枝上就可以重复取2，因为同一树枝上元素的集合才是唯一子集！

2. 给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。candidates 中的数字可以无限制重复被选取。

寻找所有可行解的问题，都可以尝试 搜索回溯

递归的层级嵌套可以解决多层for循环



class Solution {

private:

    vector<vector<int>> ans;

    vector<int> combine;//b保存子集

public:

    //深度优先搜索，传参为顺序排列的原数组，现在还需要的数字和，结果，现在开始的序号

    void dfs(vector<int>& candidates,int target\_now, int idx)

    {

        if(target\_now==0)

        {

            ans.push\_back(combine);

            return;

        }

        else if(target\_now<candidates[idx])

        {

            return;

        }

        else//target\_now>candidates[idx]

        {

            for(int i=idx;i<candidates.size();++i)//遍历当前层所有可能的值，都先把当前层放进去，然后遍历下一层，然后再把当前层放进去的值pop出来

            {

                combine.push\_back(candidates[i]);

                dfs(candidates,target\_now-candidates[i],i);

                combine.pop\_back();

            }

        }

    }

    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

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

        dfs(candidates,target, 0);

        return ans;

    }

};

递归的终止条件必须想清楚，这里的终止条件是当dfs当前节点的下一个节点时，如果剩余的数据为0，则到达终止条件。

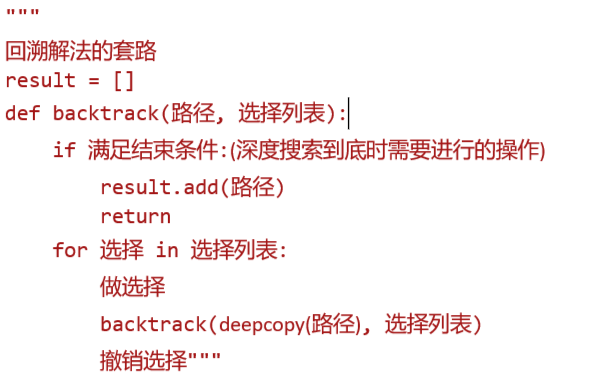
三、回溯

1.数组的全排列

给定一个**没有重复** 数字的序列，返回其所有可能的全排列。

用回溯的方法，要点：

1. 需要一个数组来保存某个数字是否已经在排列中，如果存在，需要for循环continue调过
2. 递归的终止条件应该是当前排列的size与数组的size一致
3. For循环调用递归的时候应该是在递归前将当前数字push进排列，调了递归后将数字pop出排列



一般我们首先会将数组进行升序排列，然后画出树状图，观察如何出现重复的，然后找相关的策略

2.回文字符：给定一个字符串 s ，返回其通过重新排列组合后所有可能的回文字符串，并去除重复的组合。如不能形成任何回文排列时，则返回一个空列表。

思路：首先将string排序，用map数据结构统计每个字符出现的次数。若出现奇数次的字符数量多于1个，则不可能生成回文字符串。然后我们将所有的字符串各取一半拿出来进行全排列，这里如果有奇数个的字符串也成立。为了排列不重复，树的每一层不能有相同的字符，且出现过的字符不能再出现。最终得到一半字符，然后将这半字符反转，若有奇数个的字符，则用前半字符+奇数个的字符+后半字符，得到答案，这里注意奇数个的字符不能为char，否则自带的\0会截断整个字符，导致后半字符加失败。

这里也得注意奇数个的字符也必须参加全排列。

3.找出最长回文子串：只要找到最长的奇数串，以及所有的偶数串，即可

# 栈

1. string的查找分割常用函数

pos = s.find(delimiter)) != string::npos;//在s字符串里找到delimiter第一次出现的位置,从0开始

s.earse(0,length)删除s从0开始，长为length的字符串，无返回值

五、二叉查找树

1.AVL平衡数的旋转是在一棵已经是AVL树上插入一个新节点。所以不是直接对一个树做旋转操作，而是一个个地节点建立一个AVL树，然后旋转操作变平衡。

但是这里其实原树是一个二叉搜索树，所以只需要中序遍历树，便可得到排好序的数字，然后直接构造一棵树即可

六 位运算

1.0 ^ x = x

  x ^ x = 0；

  x & ~x = 0,

  x & ~0 =x;

注意：异或 0^0=0,1^0=1,0^1=1,1^1=0

异或的自反性： a^b^b=a,a^b^a=b

x&(-x)最右边为1的值，其余置0：-x=~x+1：x=010 –x=~x+1=101+1=110,x&(-x)=010,右边第2位为1

统计1的数量：int num=0; while((x!=0)(x-1)&(x)) {x&=(x-1);}x&(x-1)会将倒数第一个1变为0

七、树

1.根据前序中序遍历结果构造二叉树

对于任意一颗树而言，前序遍历的形式总是

[ 根节点, [左子树的前序遍历结果], [右子树的前序遍历结果] ]

即根节点总是前序遍历中的第一个节点。而中序遍历的形式总是

[ [左子树的中序遍历结果], 根节点, [右子树的中序遍历结果] ]

只要我们在中序遍历中定位到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有左右括号进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地对构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。

2.树的层级遍历。

树的每一层作为一个向量，最终输出vector<vector<int>> ans.

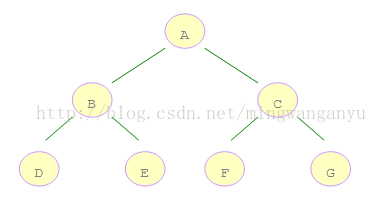
在用队列进行层级遍历的时候，每次都把队列中的所有元素都挨个遍历完。

3.树的深度和广度优先搜索

**二叉树的深度优先遍历（DFS）与广度优先遍历（BFS）**

**深度优先遍历：**从根节点出发，沿着左子树方向进行纵向遍历，直到找到叶子节点为止。然后回溯到前一个节点，进行右子树节点的遍历，直到遍历完所有可达节点为止。

**广度优先遍历：**从根节点出发，在横向遍历二叉树层段节点的基础上纵向遍历二叉树的层次。



DFS:ABDECFG

BFS:ABCDEFG

**DFS实现：**

数据结构：栈

父节点入栈，父节点出栈，先右子节点入栈，后左子节点入栈。递归遍历全部节点即可

**BFS实现：**

数据结构：队列

父节点入队，父节点出队列，先左子节点入队，后右子节点入队。递归遍历全部节点即可

4.树的迭代遍历

前序遍历：栈实现，每次右子入栈，再左子入栈（尽可能需要先一直往左下走，直到无路可走，再弹出所有访问过的节点，弹出之后看看这个节点是否有右子树，如果有的话，当前节点换为右子节点，然后再以这个右子节点为根节点，一直向左）

后序遍历：前序是根左右，后序是左右根，反过来就是根右左，所以只需要前序先尽可能往右下走，最终将结果反一下即可

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

        vector<int> ans;

        if(!root) return ans;

        stack<TreeNode\*> stack\_node;

        while(root!=NULL||!stack\_node.empty())

        {

            while(root)

            {

                ans.push\_back(root->val);

                stack\_node.push(root);

                root=root->right;

            }

            TreeNode\* cur=stack\_node.top();

            stack\_node.pop();

            root=cur->left;

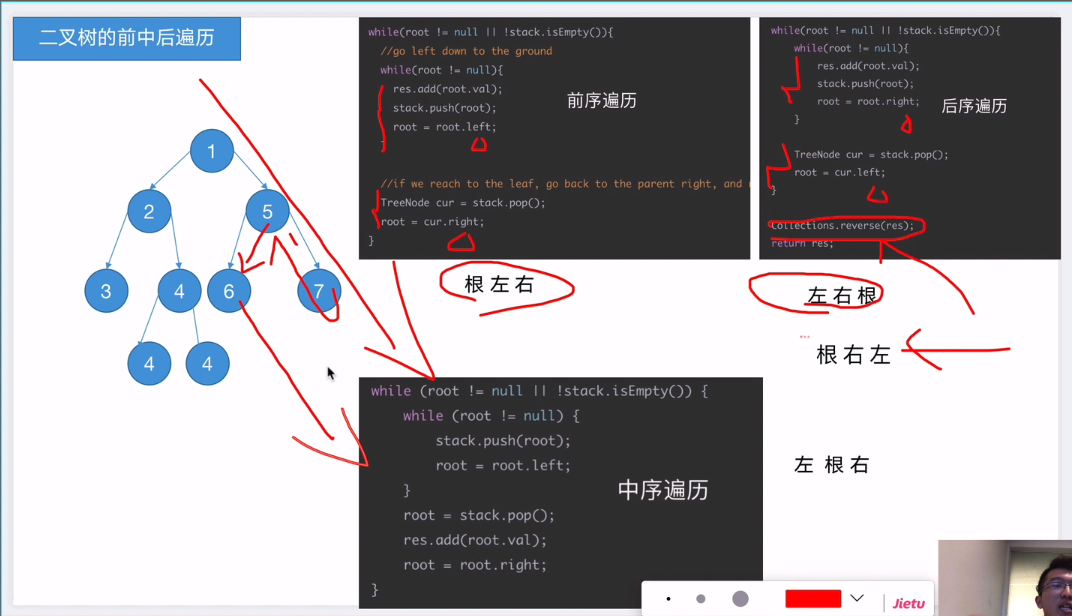
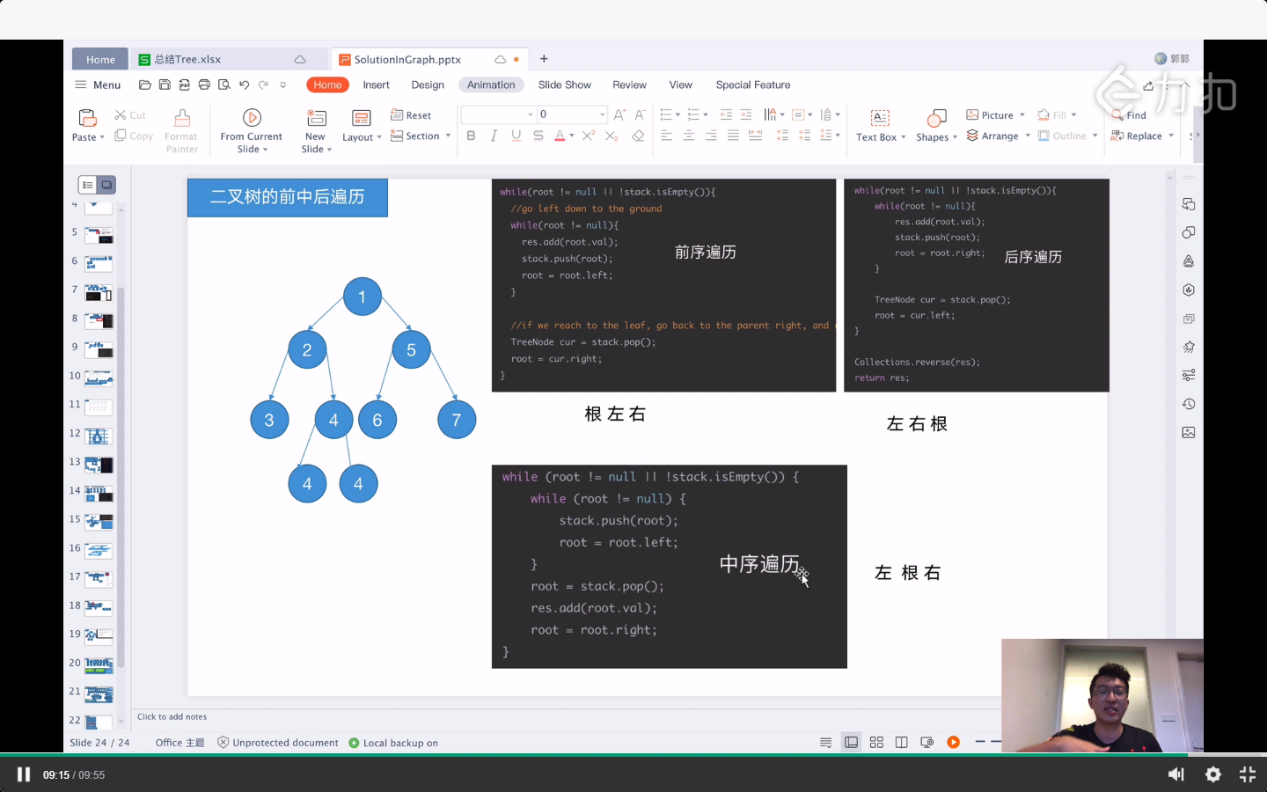
        }

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

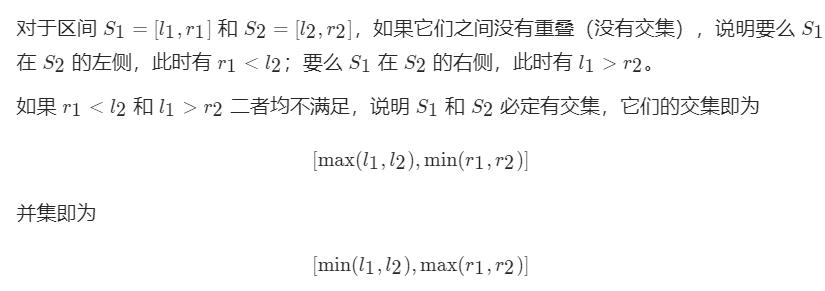
        return ans;

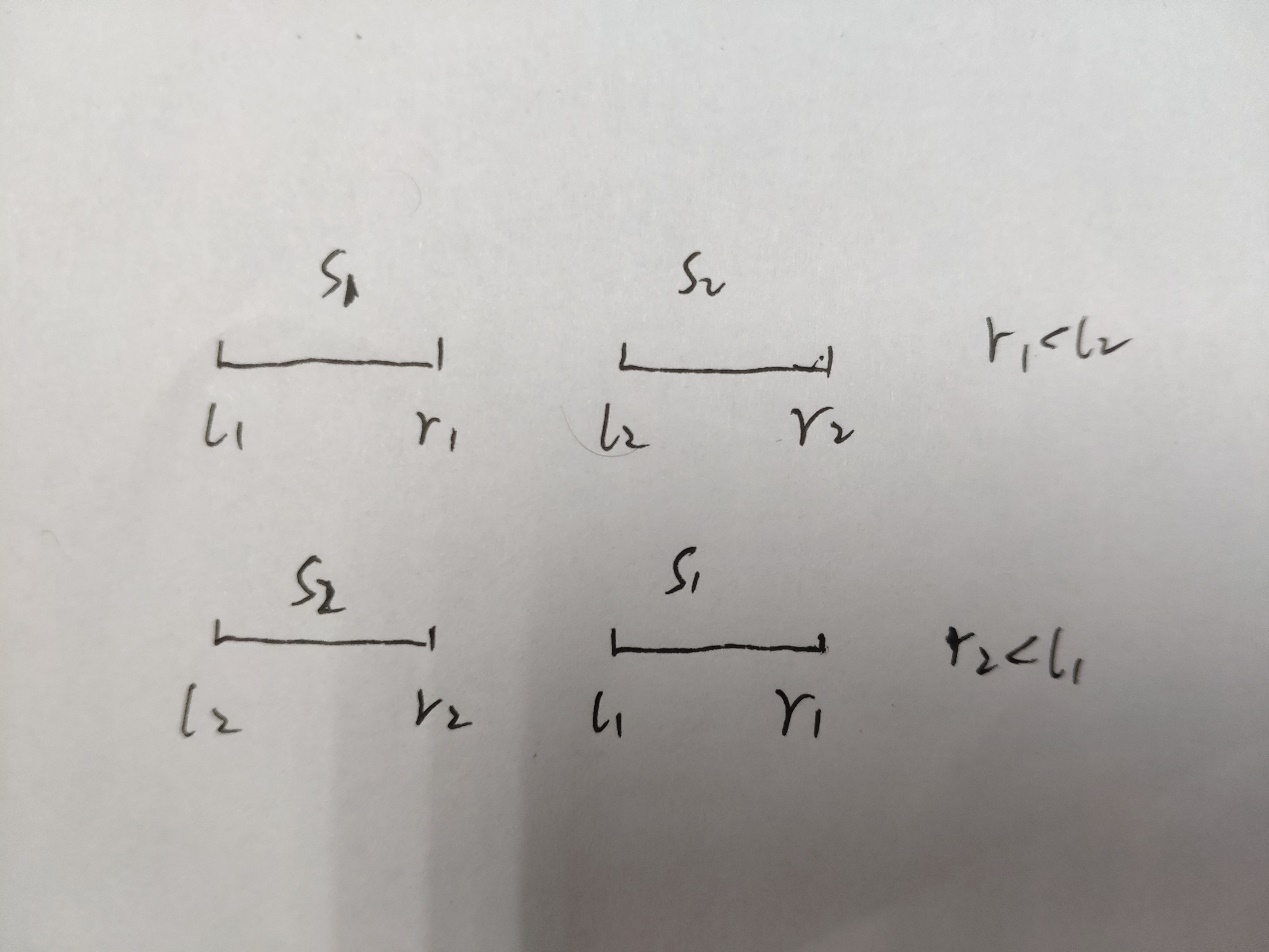
    }

中序遍历：压栈时不进行元素的遍历，而是到无路可走的时候才把第一个值放到容器中



区间的交并补判断（<https://leetcode-cn.com/problems/insert-interval/>）：





取交集即是取区间左侧的最大值，区间右侧的最小值

取并集即是取区间左侧的最小值，区间右侧的最大值

取了并集之后的区间注意要更新区间的左右两侧的值

统计x中1的数量

法一：

int num=0;

        while(x)

        {

            num+=x%2;

            x/=2;

        }

法二：int num=0;

while((x!=0)(x-1)&(x))

{x&=(x-1);}//x&(x-1)会将最右边的1去掉

双指针：<https://leetcode-cn.com/problems/palindrome-linked-list/>判断是否为回文链表

 ListNode\* fast=head;

        ListNode\* slow=head;

        vector<int> front\_left;

        //双指针法：如果为偶数个元素，则slow指向中间偏右；如果为奇数个元素，slow指针刚好指向中间的元素。所以slow是中间或中间偏右

//元素奇偶判定：fast==NULL 偶数个 fast->next=NULL 奇数

        bool even=false;//偶数个元素

        while(fast&&fast->next)//进入这个循环必须有两个元素

        {

            front\_left.push\_back(slow->val);

            slow=slow->next;

            fast=fast->next->next;

        }

法二：快慢指针找出中间的指针，反转后面的链表，比较是否回文，再将后部链表反转回来，时间O(n),空间O(1)

/\*\* 反转区间 [a, b) 的元素， 注意是左闭右开 \*/  
ListNode reverse(ListNode a, ListNode b) {  
ListNode pre, cur, nxt;  
pre = null; cur = a; nxt = a;//cur是当前准备当新的head的节点，next是cur的下一个节点，免得cur的next指向prev后next丢失  
// while 终⽌的条件改⼀下就⾏了  
while (cur != b) {  
nxt = cur.next;  
cur.next = pre;  
pre = cur;  
cur = nxt;  
} /  
/ 返回反转后的头结点  
return pre;  
}

对于数组中几个数字嘉禾超出int的边界导致溢出的处理：<https://leetcode-cn.com/problems/count-of-range-sum/>

判断是否溢出不应该利用加法去判断，而是应该用INT\_MAX或INT\_MIN做减法

if(sum>0&&nums[j]>INT\_MAX-sum)

         break;

     else if(sum<0&&nums[j]<INT\_MIN-sum)

         break;

归并排序：递归，自下而上，数组被分为两半，左右两半分别有序。

Sort(left,mid)

Sort(mid+1,right)

Merge(left,right)

时间复杂度O(nlog(n)) 空间复杂度O(n)

先计算前缀和preSum，则原问题即为



动态规划

求最值，核心是穷举

三要素：有重叠子问题、最优子结构（子问题互相独立，求解最值互不干扰，即可以通过子问题的最优解，推出更大规模问题的最优解）、状态转移方程（hardest）

状态转移方程框架：明确[状态]->定义dp数组/函数的含义->明确[选择]->明确base case

斐波那契数列：直接递归存在大量重复计算，可以使用带备忘录的递归。要确定base case（即不能分解的）每次计算一个值之后不急着返回，而是先记到备忘录里面再返回；每次遇到一个子问题时先去备忘录里面查找，如果存在，则直接返回。则这样的话我们每次返回备忘录里面的值即可（可能是刚算的，也可能是以前算好的）。备忘录可以使用数组，也可以使用哈希字典。

带备忘录的递归是自顶向下，动态规划是自底向上。

int fib(int n) {

        //法一：直接递归

        if(n==0)    return 0;

        if(n==1)    return 1;

        return fib(n-1)+fib(n-2);

    }

法二：备忘录法，仍然是自顶向下

vector<int> memo(n+1,0);//初始化为n+1个元素，因为我们是从0到n

        return helper(memo,n);

    int helper(vector<int>& memo,int n)

    {

        //base case

        if(n==0||n==1)

        {

            memo[n]=n;

            return memo[n];

        }

        //已经计算过

        if(memo[n]!=0)  return memo[n];

        memo[n]=helper(memo,n-1)+helper(memo,n-2);

        return memo[n];

    }

//法三：Dp tatle，自底向上

        if(n==0)    return 0;

        if(n==1)    return 1;

        vector<int> dp(n+1,0);

        dp[1]=1;

        for(int i=2;i<dp.size();++i)

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

        return dp[n];

确定状态转移方程也就是找一个穷举的公式，然后再思考如何聪明地穷举，而Dp table（自底向上）和备忘录（自顶向下）就是用空间换时间，消除重叠

coinChange: <https://leetcode-cn.com/problems/coin-change/submissions/>

//超时

int dp(vector<int>& coins,int n,vector<int>& memo)

    {

        //base case

        if(n<0) return -1;//表示不存在这种情况

        if(n==0) return 0;

        if(memo[n]!=0)  return memo[n];//先查备忘录

        int min\_num=INT\_MAX;

        for(int coin:coins)

        {

            int num=dp(coins,n-coin,memo);//递归项

            if(num==-1) continue;

            min\_num=min(min\_num,num+1);

        }

        if(min\_num!=INT\_MAX)

        {

            memo[n]=min\_num;

            return memo[n];

        }

        else

        {

            return -1;

        }

    }

    int coinChange(vector<int>& coins, int amount) {

        //自顶向下递归，带备忘录

        //如果是base case，直接返回。每次先查备忘录，如果存在，直接返回。不存在，递归计算，计入备忘录，返回相应的备忘录值

        vector<int> memo(amount+1,0);

        return dp(coins,amount,memo);

    }

法二：Dp table，自底向上，不过需要判断某个面值是否被成功组成

 //自底向上，dp table迭代

        vector<int> dp(amount+1,INT\_MAX);//每个值值硬币数量最多就是全用1

        dp[0]=0;//base case

        for(int i=1;i<=amount;++i)//从1算到amount，则最终的dp[amount]就是答案

        {

            int min\_num=INT\_MAX;

            for(int coin:coins)//穷举所有的可能

            {

                if(i-coin<0)    continue;

                if(dp[i-coin]==INT\_MAX) continue;

                min\_num=min(min\_num,dp[i-coin]+1);

            }

            dp[i]=min\_num;

        }

        return (dp[amount]==INT\_MAX)?-1:dp[amount];

最优子结构：能从小规模的最值求出更大规模的最值。他不是动态规划独有的性质，但却是动态规划的必要性质。

动态规划就是简单的base case往后推导，只有符合最优子结构的问题，才有这种链式反应的性质。

找最优子结构就是证明状态转移方程正确性。只要符合最优子结构，则可以使用暴力求解，然后观察有没有重叠子问题，有则优化，无则OK

二维Dp table可以正向，反向，斜向遍历。只要保证遍历的过程所需的状态已经计算完成；遍历的终点是存储结果的位置

// 斜着遍历数组，从第一行的第一个右下斜着遍历，然后从第一行的第二个右下斜着遍历  
for (int l = 1; l <= n; l++) {//l确定j的范围  
 for (int i = 0; i <= n - l; i++) {  
 int j = l + i - 1;  
 // 计算 dp[i][j]  
 }  
}