

第一周：代码随想录

数组部分

数组是存放在连续的内存空间上的相同类型数据的集合

数组的内存空间是连续的

正是因为数组的在内存空间的地址是连续的，所以我们在删除或者增添元素的时候，就难免要移动其他元素的地址。

Vector和array的区别：

严格来说vector是容器，不是数组

二分法

二分查找

左闭右闭和左闭右开

使用二分法的前提：**前提是数组为有序数组**，同时题目还强调**数组中无重复元素**，因为一旦有重复元素，使用二分查找法返回的元素下标可能不是唯一的

快慢指针&双指针

移除元素，有序元素的平凡

滑动窗口法

还需要做题

长度最小的子数组

螺旋矩阵

```
vector<vector<int>> res(n, vector<int>(n, 0));
```

`vector<vector<int>> res;`：声明了一个名为 `res` 的二维vector。

- `(n, vector<int>(n, 0))`：这是vector的构造函数调用。在这里，它创建了一个包含n个元素的vector，每个元素都是一个包含n个整数的vector，并且这些整数都初始化为0。
- 因此，`res` 是一个n行n列的二维矩阵，其中所有的元素都初始化为0。

保证严格的左闭右开原则，就不会乱套

第二周：代码随想录

链表

单链表中的指针域只能指向节点的下一个节点

双链表：每一个节点有两个指针域，一个指向下一个节点，一个指向上一个节点。

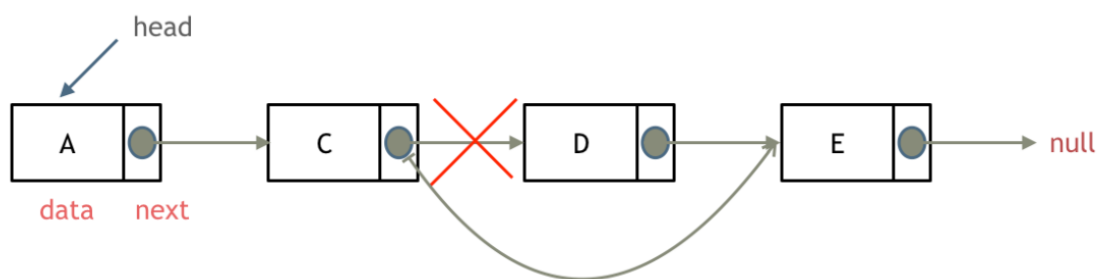
双链表 既可以向前查询也可以向后查询。

链表中的节点在内存中不是连续分布的，而是散乱分布在内存中的某地址上，分配机制取决于操作系统的内存管理。

面试时要自己手写链表的定义

c++里的定义链表方式：

```
// 单链表
struct ListNode {
    int val; // 节点上存储的元素
    ListNode *next; // 指向下一个节点的指针
    ListNode(int x) : val(x), next(NULL) {} // 节点的构造函数
};
```



只要将C节点的next指针 指向E节点就可以了。

那有同学说了，D节点不是依然存留在内存里么？只不过是没在这个链表里而已。

是这样的，所以在C++里最好是**再手动释放**这个D节点，释放这块内存。

Java、Python，就有自己的内存回收机制，就不用自己手动释放了

移除链表元素

移除头结点和移除其他节点的操作是不一样的

那么可不可以 以一种统一的逻辑来移除 链表的节点呢。

其实**可以设置一个虚拟头结点**，这样原链表的所有节点就都可以按照统一的方式进行移除了

反转链表

不需要再定义一个新的链表，只需要改变链表的next指针的指向

首先定义一个cur指针，指向头结点，再定义一个pre指针，初始化为null

把 cur->next 节点用tmp指针保存

将cur->next 指向pre，此时已经反转了第一个节点了。之后循环移动pre和cur指针

迭代是通过循环结构来实现的，和递归不同

两两交换链表中的节点

使用虚拟头节点会方便很多，因为不用额外处理头节点的逻辑

其中while循环的判定是`while(cur->next != nullptr && cur->next->next != nullptr)`

删除链表中的倒数第N个节点

依旧使用虚拟头节点，使用fast和slow节点

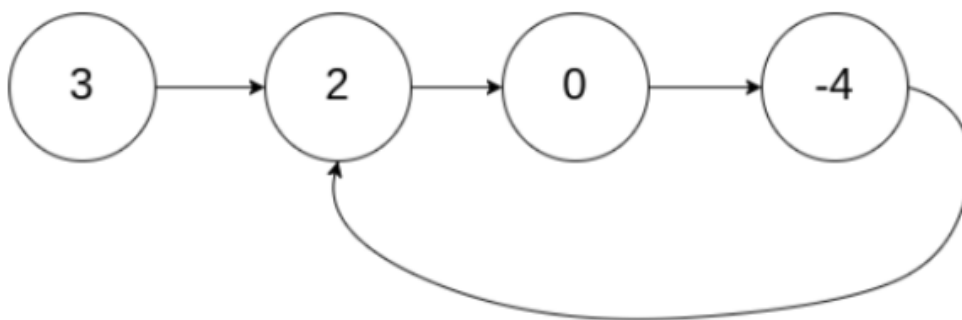
fast从dummyhead先走n+1步，然后slow，fast一起走，fast走到null即结尾时，slow正好走到倒数n+1个，即n的前一个

最后return head是错误的，当测试用例是[1]并且n=1时，会把1删掉，也就是head指针被删除了，直接return dummyhead->next才是正确的

环形链表

判断链表中是否有环，并且找到环的入口

因为是链表，所以如果有环一定呈现下面的样子

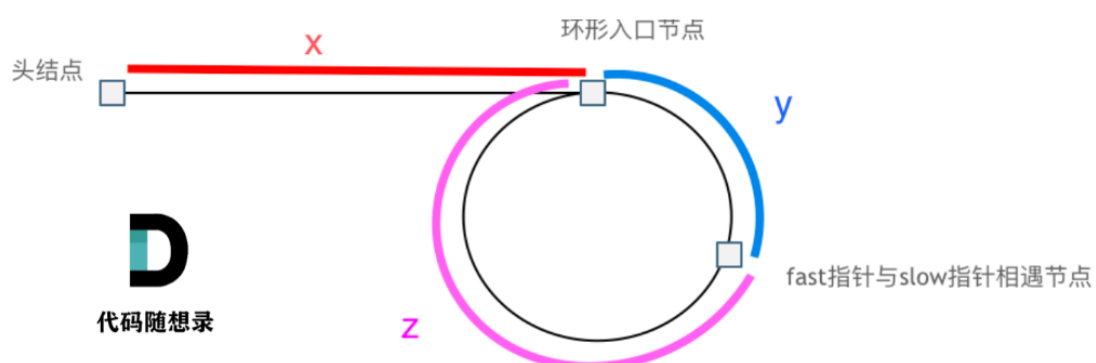


即环的后面不会再有节点了

使用fast和slow节点（fast走两步，slow1步）

如果有环，fast和slow一定会相遇，相当于fast追逐slow，至此可以判断出有没有环

找环的入口（需要数学证明）



那么相遇时：slow指针走过的节点数为： $x + y$ ，fast指针走过的节点数为： $x + y + n(y + z)$ ，n为fast指针在环内走了n圈才遇到slow指针， $(y+z)$ 为一圈内节点的个数A。

$$(x + y) * 2 = x + y + n(y + z)$$

两边消掉一个 $(x+y)$: $x + y = n (y + z)$

因为要找环形的入口，那么要求的是 x ，因为 x 表示 头结点到 环形入口节点的距离。

所以要求 x ，将 x 单独放在左面： $x = n (y + z) - y$ ，

再从 $n(y+z)$ 中提出一个 $(y+z)$ 来，整理公式之后为如下公式： $x = (n - 1) (y + z) + z$ 注意这里 n 一定是大于等于1的，因为 fast指针至少要多走一圈才能相遇slow指针。

当 n 为1的时候，公式就化解为 $x = z$ ，

这就意味着，从**头结点**出发一个指针，从**相遇节点**也出发一个指针，这两个指针每次只走一个节点，那么当这两个指针相遇的时候就是环形入口的节点

如果 n 大于1情况也是一样的，不过就是index1在环里多转了 $(n-1)$ 圈

哈希表

又叫散列表

在做面试题目的时候遇到需要**判断一个元素是否出现过的**场景也应该第一时间想到哈希法

哈希法是**牺牲了空间换取了时间**，因为我们要使用额外的数组，set或者是map来存放数据

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	$O(\log n)$	$O(\log n)$
std::multiset	红黑树	有序	是	否	$O(\log n)$	$O(\log n)$
std::unordered_set	哈希表	无序	否	否	$O(1)$	$O(1)$

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

题1

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

说明: 你可以假设字符串只包含小写字母。

解:

因为是看他的出现次数——想到哈希表，而他指定了是26个字母之内，所以可以用数组（因为限制大小了）

此外，直接用字符串和'a'之间的相对差作数组下标即可，并不需要知道ASCII码

字符串

swap可以有两种实现。

一种就是常见的交换数值：

```
int tmp = s[i];
s[i] = s[j];
s[j] = tmp;
```

一种就是通过位运算：

```
s[i] ^= s[j];
s[j] ^= s[i];
s[i] ^= s[j];
```

如果是删除数组/字符串中的，如果是常规的从前往后删，时间复杂度为 $O(n^2)$

为了满足时间复杂度为 $O(n)$ ，且为原地改变，使用双指针

如果是将某一个元素变为需要用更多元素容器装的，比如1变为number，那数组长度变大，则双指针从后往前替换， $s[fast]=s[slow]$ ，最后记得j减小

如果是删除某个元素，如空格或特定值，则用双指针的从前往后替换，for循环里令 $s[slow++]=s[fast]$

151.翻转字符串里的单词

力扣[题目链接](#)

给定一个字符串，逐个翻转字符串中的每个单词。

示例 1:

输入: "the sky is blue"

输出: "blue is sky the"

示例 2:

输入: " hello world! "

输出: "world! hello"

解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。

示例 3:

输入: "a good example"

输出: "example good a"

解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

`s[fast]s[fast-1]&&s[fast]'` 不可以连写，要写成两个条件

思路:

第一步：消除空格——和消除数组中消除特定元素是一样的——用双指针，原地消除，并且 $O(n)$ 的时间复杂度

第二步：全部reverse

第三步：遇到空格reverse单词

右旋字符串

卡码网[题目链接](#)

字符串的右旋转操作是把字符串尾部的若干个字符转移到字符串的前面。给定一个字符串 s 和一个正整数 k ，请编写一个函数，将字符串中的后面 k 个字符移到字符串的前面，实现字符串的右旋转操作。

例如，对于输入字符串 "abcdefg" 和整数 2，函数应该将其转换为 "fgabcde"。

输入：输入共包含两行，第一行为一个正整数 k ，代表右旋转的位数。第二行为字符串 s ，代表需要旋转的字符串。

输出：输出共一行，为进行了右旋转操作后的字符串。

样例输入：

```
1 2
2 abcdefg
```

样例输出：

```
1 fgabcde
```

数据范围： $1 \leq k < 10000$, $1 \leq s.length < 10000$;

这个题就是上面一个题的简单版，只需要将他全部翻转，再按 k 局部反转就可以了

KMP算法

实现字符串的匹配问题

首先获得next数组，然后用该数组进行匹配

时间复杂度分析

其中 n 为文本串长度， m 为模式串长度，因为在匹配的过程中，根据前缀表不断调整匹配的位置，可以看出匹配的过程是 $O(n)$ ，之前还要单独生成next数组，时间复杂度是 $O(m)$ 。所以整个KMP算法的时间复杂度是 $O(n+m)$ 的。

暴力的解法显而易见是 $O(n \times m)$ ，所以KMP在字符串匹配中极大地提高了搜索的效率。

题目：

给你两个字符串 `haystack` 和 `needle`，请你在 `haystack` 字符串中找出 `needle` 字符串的第一个匹配项的下标（下标从 0 开始）。如果 `needle` 不是 `haystack` 的一部分，则返回 -1。

输入：haystack = "sadbutsad", needle = "sad"

输出：0

解释："sad" 在下标 0 和 6 处匹配。

第一个匹配项的下标是 0，所以返回 0。

解答:

直接背吧。。

```
class Solution {
public:
    //getNext函数
    void getNext(int *next,const string&s){
        int j=0;
        next[0]=0;
        for(int i=1;i<s.size();i++){
            while(j>0&&s[i]!=s[j]){
                j=next[j-1];
            }
            if(s[i]==s[j])
            {
                j++;
            }
            next[i]=j;
        }
    }

    int strStr(string haystack, string needle) {
        if(needle.size()==0){
            return 0;
        }
        int next[needle.size()];
        getNext(next,needle);
        //开始基于next数组的匹配
        int j=0;
        for (int i = 0; i < haystack.size(); i++) {
            while(j > 0 && haystack[i] != needle[j]) {
                j = next[j - 1];
            }
            if(haystack[i]==needle[j]){
                j++;
            }
            //检查j的大小是否等于needle.size(), 如果等于就匹配结束了
            if(j==needle.size())
                return (i-needle.size()+1);
        }
        return -1;
    }
};
```

衍生题:

给定一个非空的字符串 `s` , 检查是否可以通过由它的一个子串重复多次构成。

示例 1:

输入: `s = "abab"`

输出: `true`

解释: 可由子串 `"ab"` 重复两次构成。

解:

匹配字符串, KMP, 但是运用到数学推理公式

```
//是next[i]=j!!!!!!!!!!!!!!!!!!!!!!  
//不要再写成next[j]=j了  
next[i]=j;
```

```
//这里是next数组来解题, 和你s没有什么关系  
//到底用哪个数组请分清  
if(next[len-1]!=0&&len%(len-(next[len-1]))==0)
```

第三周

开始写150题里面的题了

双指针法

回文判断

我被0P害惨啦! 有没有可能你是用的大小写ascii差32, 0和P刚好也差32!
背一下ASCII表吧

判断子序列

双指针

两数之和

题目说了是非递归顺序排列。双指针不断判断大小向内收敛

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 **非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则 $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$ 。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 **只对应唯一的答案**，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

示例 1:

输入: `numbers = [2, 7, 11, 15]`, `target = 9`

输出: `[1, 2]`

解释: 2 与 7 之和等于目标数 9。因此 `index1 = 1`, `index2 = 2`。
返回 `[1, 2]`。

个人感觉两数之和和盛水的题很像，不过一个是排好顺序的一个没有，没排好顺序的用贪心算法（不一定会更好但至少不会更坏），用 `res` 记录最大结果不断更新就好了


盛最多水的容器


双指针，不断向内收敛，主要就是想清数学性正确，收敛的时候一步一步来，不要考虑太多，类似于贪心算法


11. 盛最多水的容器

已解答 

中等

 相关标签

 相关企业

 提示

Aa

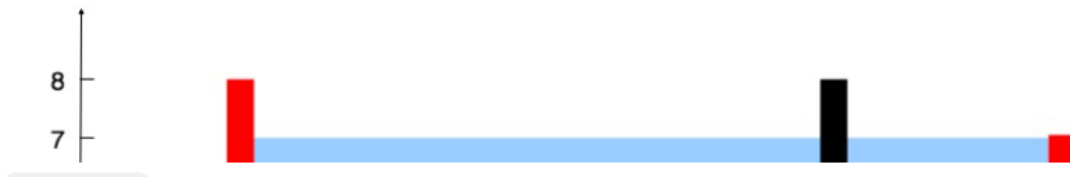
给定一个长度为 n 的整数数组 `height`。有 n 条垂线，第 i 条线的两个端点是 $(i, 0)$ 和 $(i, \text{height}[i])$ 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1：



三数之和

给你一个整数数组 `nums`，判断是否存在三元组 $[\text{nums}[i], \text{nums}[j], \text{nums}[k]]$ 满足 $i \neq j$ 、 $i \neq k$ 且 $j \neq k$ ，同时还满足 $\text{nums}[i] + \text{nums}[j] + \text{nums}[k] == 0$ 。请

你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

输入： `nums = [-1,0,1,2,-1,-4]`

输出： `[[-1,-1,2],[-1,0,1]]`

先排序；三数之和，固定一个，剩下2个就用双指针法，同两数之和。

while里套while出问题了，所以把while套在一个可能情况比较少的if分支里了

记住 `sort(nums.begin(), nums.end());`，这是vector的函数

总结

犹豫不决先排序，步步逼近双指针

栈和队列

栈的应用：

编译器在 词法分析的过程中处理括号、花括号等这个符号的逻辑，也是使用了栈这种数据结构。

linux系统中，cd这个进入目录的命令我们应该再熟悉不过了。

```
cd a/b/c/../../
```

这个命令最后进入a目录，系统是如何知道进入了a目录呢，这就是栈的应用（其实可以出一道相应的面试题了）

用栈实现队列

设计一个输入栈和输出栈，只用一个栈实现队列是不可能的，因为他是单边的，所以必须要有两个栈

用队列实现栈

原本想着仿造上面想着用一个输入队列，一个输出队列，就可以模拟栈的功能，但是并不可以

队列是先进先出的规则，把一个队列中的数据导入另一个队列中，数据的顺序并没有变，并没有变成先进后出的顺序。

所以用栈实现队列，和用队列实现栈的思路还是不一样的

用两个队列模拟时：一个用来备份数据，操作完再还原回去

队列模拟栈，其实一个队列就够了，因为能知道队列的size，只要把队列pop的元素放到队列最后就行，记录好操作次数就行

"向零截断"是指在除法运算中，将除法结果舍入到最接近的整数，而不是四舍五入。具体来说，向零截断是指将小数部分丢弃，只保留整数部分

后缀的计算

用栈

还有后缀变中缀，中缀变后缀等题，可以看之前ppt

滑动窗口的最大值

给定一个数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

进阶：

你能在线性时间复杂度内解决此题吗？

示例：

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

输出： `[3,3,5,5,6,7]`

解释：

使用了单调栈

使用了deque双向队列。用法有：

```
deque<int>q
q.front();
q.back();
q.pop_front();    q.pop_back();
q.push_front();   q.push_back();
```

使用deque来实现单调栈最为合适，在文章[栈与队列：来看看栈和队列不为人知的一面 \(opens new window\)](#)中，我们就提到了常用的queue在没有指定容器的情况下，deque就是默认底层容器。

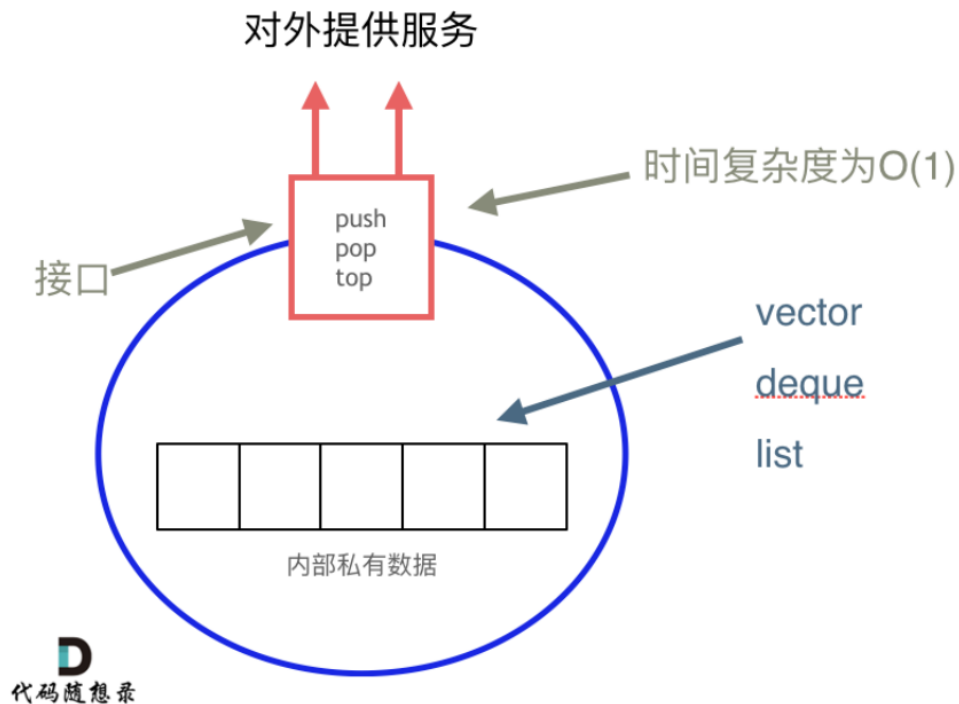
接下来介绍的栈和队列也是SGI STL里面的数据结构，知道了使用版本，才知道对应的底层实现

栈提供push 和 pop 等等接口，所有元素必须符合先进后出规则，所以栈

栈是以底层容器完成其所有的工作，对外提供统一的接口，底层容器是可插拔的（也就是说我们可以控制使用哪种容器来实现栈的功能）。

所以STL中栈往往不被归类为容器，而被归类为container adapter（容器适配器）。

从下图中可以看出，栈的内部结构，栈的底层实现可以是vector，deque，list 都是可以的，主要就是数组和链表的底层实现。



我们常用的SGI STL，如果没有指定底层实现的话，默认是以deque为栈和队列的底层结构

deque是一个双向队列，只要封住一段，只开通另一端就可以实现栈的逻辑了

```
stack<int, vector<int> > third; // 使用vector为底层容器的栈
```

队列中先进先出的数据结构，同样不允许有遍历行为，不提供迭代器

第三周

二叉树

满二叉树，完全二叉树，二叉搜索树，平衡二叉搜索树（AVL）——左右两个子树的高度差绝对值不超过1（空树也是平衡搜索二叉树）

- 深度优先遍历
 - 前序遍历（递归法，迭代法）
 - 中序遍历（递归法，迭代法）
 - 后序遍历（递归法，迭代法）
- 广度优先遍历
 - 层次遍历（迭代法）

看如下中间节点的顺序，就可以发现，中间节点的顺序就是所谓的遍历方式

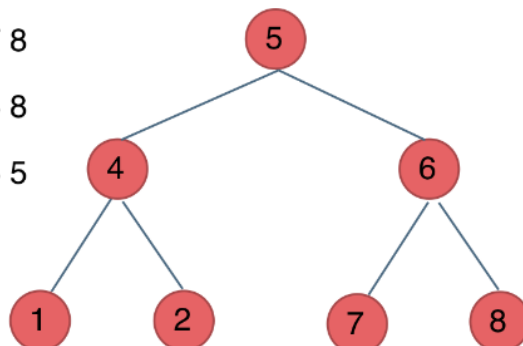
- 前序遍历：中左右
- 中序遍历：左中右
- 后序遍历：左右中

大家可以对着如下图，看看自己理解的前后中序有没有问题。

前序遍历（中左右）：5 4 1 2 6 7 8

中序遍历（左中右）：1 4 2 5 7 6 8

后序遍历（左右中）：1 2 4 7 8 6 5



二叉树有两种存储方式顺序存储，和链式存储，顺序存储就是用数组来存

对于链表存储的二叉树节点的定义方式：

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

在现场面试的时候 面试官可能要求手写代码，所以数据结构的定义以及简单逻辑的代码一定要锻炼白纸写出来。

处理的时候一定记得树为空的情况，不然指针为空都不知道去哪里找错！！

遍历

- 递归型

三要素考虑：函数的参数和返回类型、函数终止条件，函数的单个逻辑

- 递归型

- 前序遍历

用栈。先将根节点放入栈中，然后将右孩子加入栈，再加入左孩子。

为什么要先加入右孩子，再加入左孩子呢？因为这样出栈的时候才是中左右的顺序。

注意下面这个地方，空节点不入栈

```
if (node->right) st.push(node->right); // 右（空节点不入栈）
    if (node->left) st.push(node->left); // 左（空节点不入栈）
```

- 中序

```

class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        TreeNode* cur = root;
        while (cur != NULL || !st.empty()) {
            if (cur != NULL) { // 指针来访问节点，访问到底层
                st.push(cur); // 将访问的节点放进栈
                cur = cur->left; // 左
            } else {
                cur = st.top(); // 从栈里弹出的数据，就是要处理的数据（放进
                result数组里的数据）
                st.pop();
                result.push_back(cur->val); // 中
                cur = cur->right; // 右
            }
        }
        return result;
    }
};

```

◦ 后序

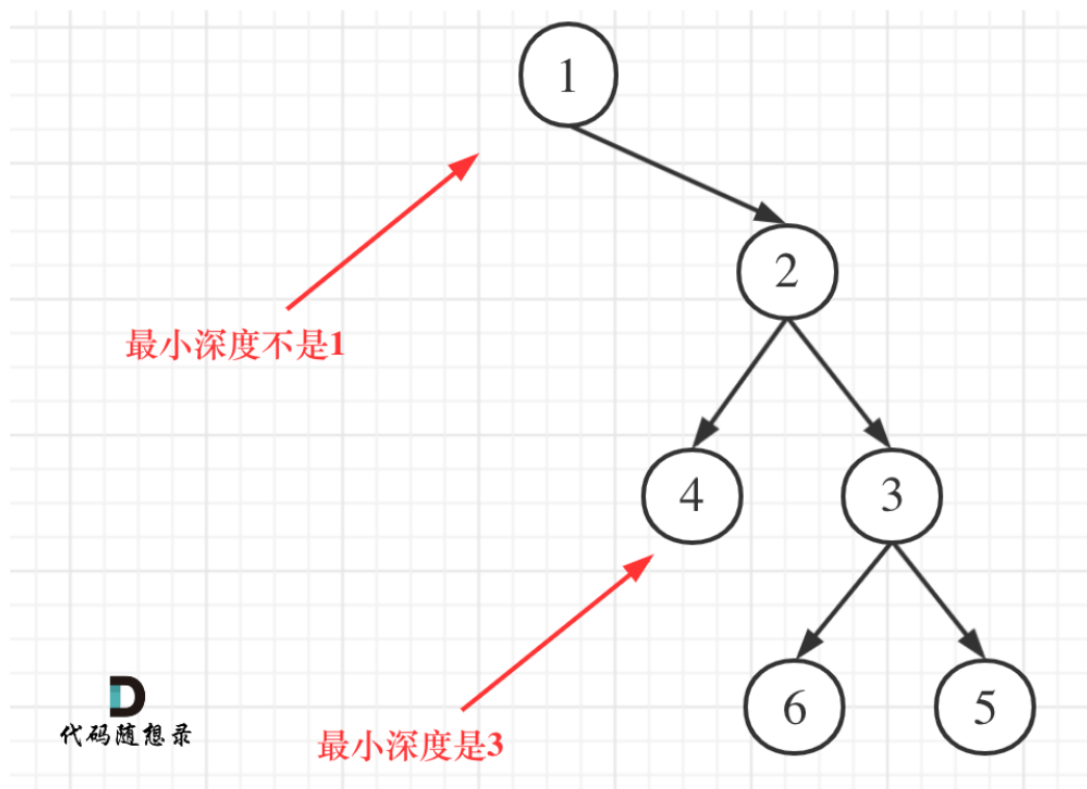
后序遍历为左右中，他的反向就是中右左，那么就是前序遍历的中左右调一下放在栈里的顺序即可

二叉树的统一迭代法

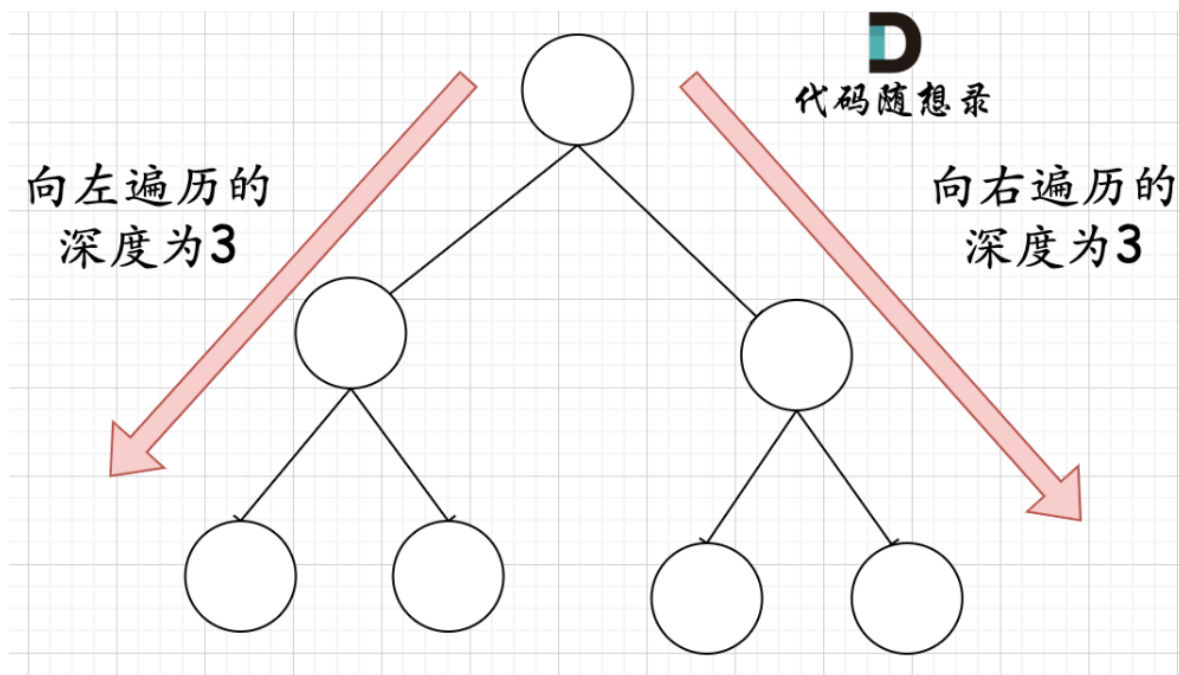
待看

根节点的高度就是二叉树的最大深度

最小深度是从根节点到最近叶子节点的最短路径上的节点数量



在完全二叉树中，如果递归向左遍历的深度等于递归向右遍历的深度，那说明就是满二叉树。



对一个数进行左移操作是指将这个数的二进制表示向左移动指定的位数

假设我们有一个数 5，其二进制表示为 101。左移一位会在原数的右侧添加一个零，相当于乘以 2。。所以 5 左移一位后变为 10，二进制表示为 1010。

- 1 左移 1 位: $1 \ll 1 = 2$ ，相当于 $1 * 2^1 = 2$
- 1 左移 2 位: $1 \ll 2 = 4$ ，相当于 $1 * 2^2 = 4$
- 2 左移 1 位: $2 \ll 1 = 4$ ，相当于 $2 * 2^1 = 4$
- 3 左移 1 位: $3 \ll 1 = 6$ ，相当于 $3 * 2^1 = 6$

平衡二叉树

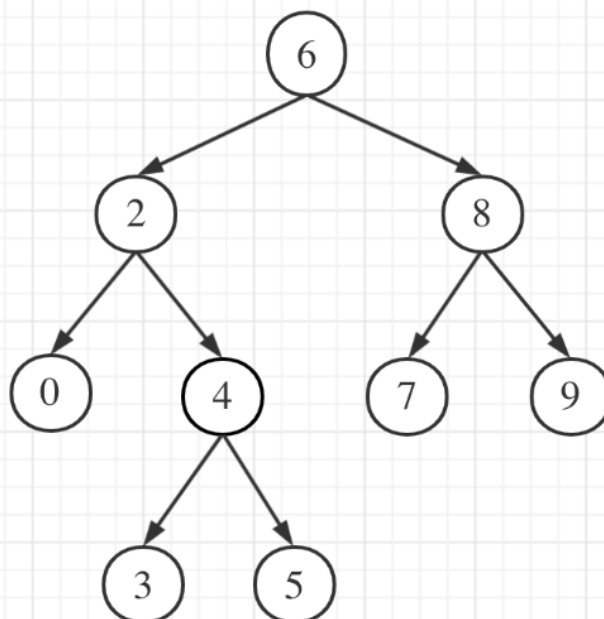
- 二叉树节点的深度：指从根节点到该节点的最长简单路径边的条数。
- 二叉树节点的高度：指从该节点到叶子节点的最长简单路径边的条数。

节点的高度：4，深度：1

节点的高度：3，深度：2

节点的高度：2，深度：3

节点的高度：1，深度：4



第四周

回溯算法

回溯是递归的副产品，只要有递归就会有回溯。

所以下讲解中，回溯函数也就是递归函数，指的都是一个函数。

组合是不强调元素顺序的，排列是强调元素顺序。

例如：{1, 2} 和 {2, 1} 在组合上，就是一个集合，因为不强调顺序，而要是排列的话，{1, 2} 和 {2, 1} 就是两个集合了。

```
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  
    }  
}
```

组合问题中的参考代码

第77题. 组合

[力扣题目链接](#)

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

示例: 输入: $n = 4, k = 2$ 输出: $[[2,4], [3,4], [2,3], [1,2], [1,3], [1,4],]$

```
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;
    }
};

```

电话号码的字母组合

注意：输入1 * #按键等等异常情况

代码中最好考虑这些异常情况，但题目的测试数据中应该没有异常情况的数据，所以我就没有加了。

但是要知道会有这些异常，如果是现场面试中，一定要考虑到！

```

string path;
path.erase(path.size()-1);
//string可以用加法，但是不可以用减法！

```

去重问题

所谓去重，其实就是使用过的元素不能重复选取。这么一说好像很简单！

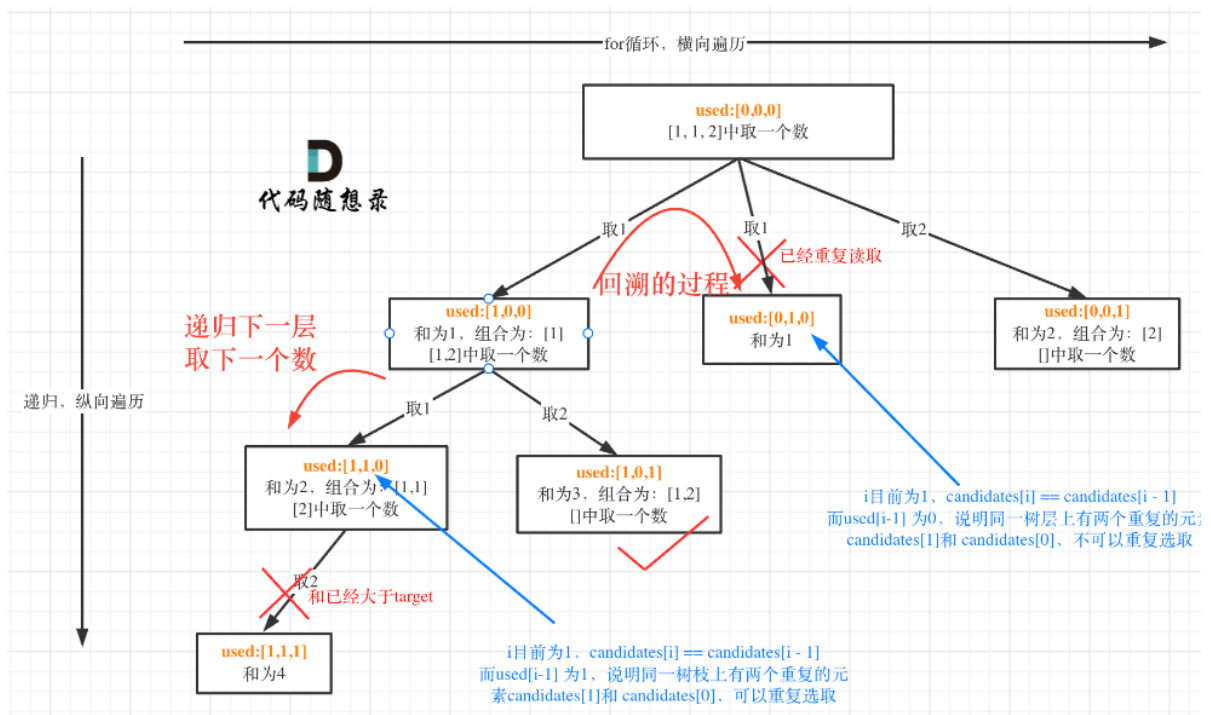
都知道组合问题可以抽象为树形结构，那么“使用过”在这个树形结构上是有两个维度的，**一个维度是同一树枝上使用过，一个维度是同一树层上使用过。**没有理解这两个层面上的“使用过”是造成大家没有彻底理解去重的根本原因。

如果 `candidates[i] == candidates[i - 1]` 并且 `used[i - 1] == false`，就说明：前一个树枝，使用了 `candidates[i - 1]`，也就是说同一树层使用过 `candidates[i - 1]`。

此时for循环里就应该做continue的操作。

- `used[i - 1] == true`，说明同一树枝 `candidates[i - 1]` 使用过
- `used[i - 1] == false`，说明同一树层 `candidates[i - 1]` 使用过

可能有的录友想，为什么 **`used[i - 1] == false` 就是同一树层呢**，因为同一树层，`used[i - 1] == false` 才能表示，当前取的 `candidates[i]` 是从 `candidates[i - 1]` 回溯而来的。而 `used[i - 1] == true`，说明是进入下一层递归，去下一个数，所以是树枝上，如下图



记住使用该方法在树层上去重时必须先排序!!!!

学到的一些string的新用法

```
string temp=s.substr(i,i-start+1);
//其中i是子串开始的位置, 第二个参数是子串的长度
```

```
string s;
s.erase(s.size()-1);
//删掉s的最后一个元素, 不可以用减法
```

如果把 子集问题、组合问题、分割问题都抽象为一棵树的话, **那么组合问题和分割问题都是收集树的叶子节点, 而子集问题是找树的所有节点!**

遍历树的时候, 把所有节点都记录下来, 就是要求的子集集合。

```
unordered_set<int> uset;//去重版
if (uset.find(nums[i]) != uset.end()) {
    continue;
}
//表示没有找到, 没找到的时候就会返回uset.end()
```

491. 非递减子序列

中等

🔖 相关标签

🏢 相关企业

Aa

给你一个整数数组 `nums`，找出并返回所有该数组中不同的递增子序列，递增子序列中 **至少有两个元素**。你可以按 **任意顺序** 返回答案。

数组中可能含有重复元素，如出现两个整数相等，也可以视作递增序列的一种特殊情况。

示例 1:

输入: `nums = [4,6,7,7]`

输出: `[[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]`

示例 2:

该问题中在树层上也要去重

而本题求自增子序列，是不能对原数组进行排序的，排完序的数组都是自增子序列了。

所以不能使用之前的去重逻辑！

本题收集结果有所不同，题目要求递增子序列大小至少为2，所以代码如下：

```
if (path.size() > 1) {
    result.push_back(path);
    // 注意这里不要加return，因为要取树上的所有节点
}
```

看到递归函数上面的 `uset.insert(nums[i]);`，下面却没有对应的 `pop` 之类的操作，应该很不习惯吧

`unordered_set<int> uset;` 是记录本层元素是否重复使用，新的一层 `uset` 都会重新定义（清空），所以要知道 `uset` 只负责本层！

代码如下：

```
// 版本一
class Solution {
private:
    vector<vector<int>>> result;
    vector<int> path;
    void backtracking(vector<int>& nums, int startIndex) {
        if (path.size() > 1) {
            result.push_back(path);
            // 注意这里不要加return，要取树上的节点
        }
        //uset放在了for外面定义，所以每一层都会重新定义
```

```

        unordered_set<int> uset; // 使用set对本层元素进行去重
        for (int i = startIndex; i < nums.size(); i++) {
            if ((!path.empty() && nums[i] < path.back())
                || uset.find(nums[i]) != uset.end()) {
                continue;
            }
            uset.insert(nums[i]); // 记录这个元素在本层用过了，本层后面不能再用了
            path.push_back(nums[i]);
            backtracking(nums, i + 1);
            path.pop_back();
        }
    }
}

public:
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        result.clear();
        path.clear();
        backtracking(nums, 0);
        return result;
    }
};

```

以上代码用我用了 `unordered_set<int>` 来记录本层元素是否重复使用。

其实用数组来做哈希，效率就高了很多。

注意题目中说了，数值范围 $[-100,100]$ ，所以完全可以用数组来做哈希。

程序运行的时候对 `unordered_set` 频繁的 `insert`，`unordered_set` 需要做哈希映射（也就是把key通过hash function映射为唯一的哈希值）相对费时间，而且**每次重新定义set，insert的时候其底层的符号表也要做相应的扩充**，也是费事的。

全排列问题

46. 全排列

中等

🏷 相关标签

🔒 相关企业

Ax

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

示例 1：

输入：nums = [1,2,3]

输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

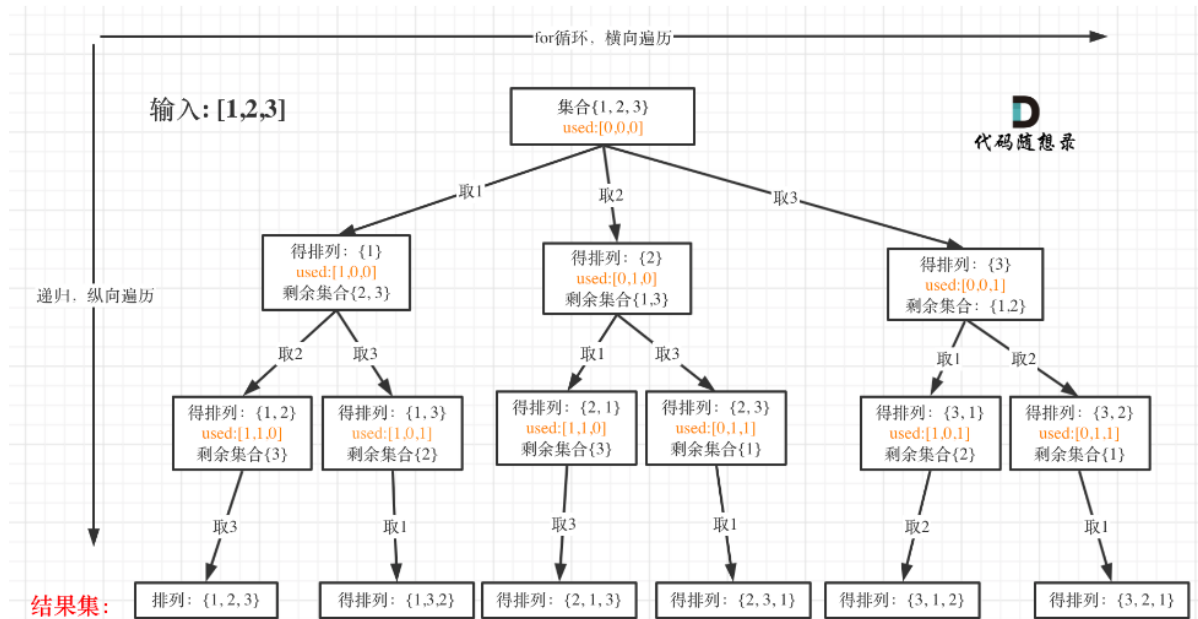
示例 2：

输入：nums = [0,1]

输出：[[0,1],[1,0]]

二刷 5.

因为是全排列，所以for循环里不用记录startindex，直接令i=0，但是需要used数组记录哪些数字已经在path里了



对于vector的初始化参数

```
vector<bool>used(nums.size(), false);  
//第一个参数为大小，第二个参数为初始化为多少
```



直接记模板代码吧

```
class Solution {  
public:  
    vector<vector<int>> res;  
    vector<int> path;  
    void backtrack(vector<int>& nums, vector<bool>& used) {  
        if (path.size() == nums.size()) {  
            res.push_back(path);  
            return;  
        }  
        //=====ATTENTION=====  
        for (int i = 0; i < nums.size(); i++) {  
            if (used[i] == true) continue;  
            //=====ATTENTION=====  
            used[i] = true;  
            path.push_back(nums[i]);  
            backtrack(nums, used);  
            path.pop_back();  
            used[i] = false;  
        }  
    }  
    vector<vector<int>> permute(vector<int>& nums) {  
        vector<bool> used(nums.size(), false);  
        backtrack(nums, used);  
        return res;  
    }  
};
```

47. 全排列 II

已解答 

中等

 相关标签 相关企业

A*

给定一个可包含重复数字的序列 `nums`，**按任意顺序** 返回所有不重复的全排列。

示例 1:

输入: `nums = [1,1,2]`

输出:

```
[[1,1,2],  
 [1,2,1],  
 [2,1,1]]
```

示例 2:

输入: `nums = [1,2,3]`

输出: `[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]`

依旧是要树层去重，树枝不去

有两种解法：

- 使用 `unordered_set`，就不用对 `nums` 排序
- 如果不用 `set`，就要排序（为了判断 `nums[i] == nums[i-1]`，使用 `used` 数组记录
 - 如果要对**树层**中前一位去重，就用 `used[i - 1] == false`
 - 如果要对**树枝**前一位去重用 `used[i - 1] == true`。
- 对于全排列中是否已经进入 `path` 用的是 `used[i]`
- 判断是否去重用的是 `used[i-1]`

能不用 `set` 还是不用吧，因为 `set` 的操作比较耗时，直接用数组就解决了，代码性能会更快

```
class Solution {  
public:  
    vector<vector<int>> res;  
    vector<int> path;  
  
    void backtrack(vector<int>& nums, vector<bool>& used) {  
        if (nums.size() == path.size()) {  
            res.push_back(path);  
            return;  
        }  
        // 放在for循环外，每一层都会更新
```



```

//unordered_set<int>uset;
for(int i=0;i<nums.size();i++){
    //used放在for循环里，会传到树枝中
    // if(uset.find(nums[i])!=uset.end()||used[i]==true)
    if((i>0&&nums[i]==nums[i-1]&&used[i-1]==false)|| (used[i]==true))
        continue;
    //别忘了插入
    // uset.insert(nums[i]);
    //别忘了设置它已经used了!!! 一些预先操作别忘了!
    used[i]=true;
    path.push_back(nums[i]);
    backtrack(nums,used);
    path.pop_back();
    used[i]=false;
    //但是uset在这里不需要弹出，因为每一层都会自己更新
}
}

vector<vector<int>> permuteUnique(vector<int>& nums) {
    vector<bool>used(nums.size(),false);
    //可以不用unordered_set，就不用对nums排序，如果不用set，就要排序，能不用还是不用
    //吧，因为set的操作比较耗时，直接用数组就解决了，代码性能会更快
    sort(nums.begin(),nums.end());
    backtrack(nums,used);
    return res;
}
};

```

回溯问题的时空复杂度

1.子集问题和组合问题

- 时间复杂度：

$$O(n \times 2^n)$$

因为每一个元素的状态无外乎取与不取，所以时间复杂度为

$$O(2^n)$$

，构造每一组子集都需要填进数组，又有需要

$$O(n)$$

，最终时间复杂度：

$$O(n \times 2^n)$$

- 空间复杂度：

$$O(n)$$

递归深度为n，所以系统栈所用空间为 $O(n)$ ，**每一层递归所用的空间都是常数级别**，注意代码里的result和path都是全局变量，就算是放在参数里，传的还是引用，并不会新申请内存空间，最终空间复杂度为 $O(n)$ 。

2.排列问题

- 时间复杂度：

$$O(n!)$$

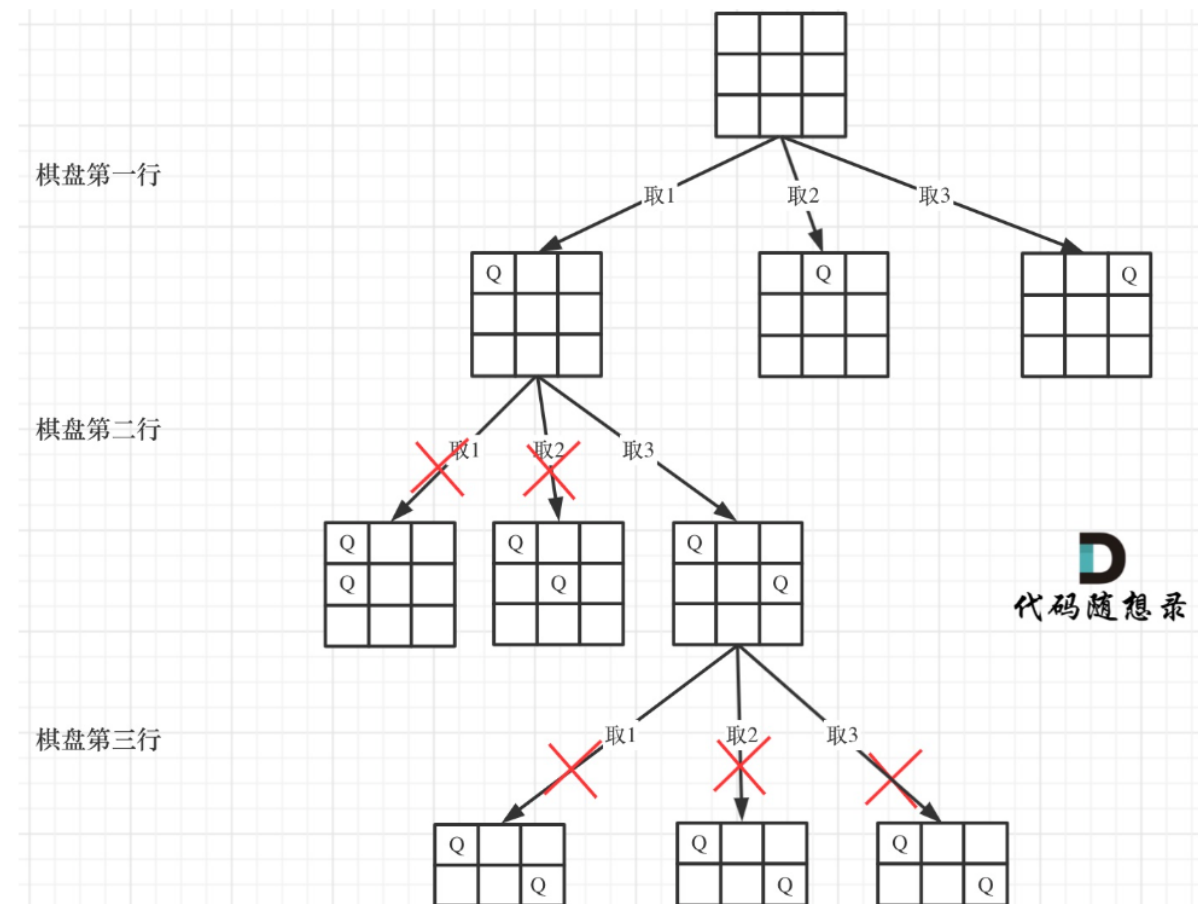
这个可以从排列的树形图中很明显发现，每一层节点为 n ，第二层每一个分支都延伸了 $n-1$ 个分支，再往下又是 $n-2$ 个分支，所以一直到叶子节点一共就是 $n * n-1 * n-2 * \dots * 1 = n!$ 。每个叶子节点都会有一个构造全排列填进数组的操作（对应的代码：`result.push_back(path)`），该操作的复杂度为 $O(n)$ 。所以，最终时间复杂度为： $n * n!$ ，简化为 $O(n!)$ 。

- 空间复杂度： $O(n)$ ，和子集问题同理。

N皇后

```
//要用&&连起来
for(int i=level,j=col;i>=0&& j<n;i--,j++){
//初始化方法
vector<string>path(n,string(n,'.'));
```

下面我用一个 $3 * 3$ 的棋盘，将搜索过程抽象为一棵树，如图：



代码用了`isvalid`来判断是否合法，当`for`循环里面东西太多了就考虑用一个函数抽象出来

一开始写的时候实在代码太多了，会弄混逻辑

此外，这个题用一个二维数组，看哪里是Q就行了，不用再额外用二维数组记录了，很麻烦，因为大题都是".", 所以只用"Q"所在的位置来判断就好了

```
class Solution {
public:
```

```

vector<vector<string>>res;

void backtrack(int n,int count,vector<string>&path){
    if(count==n){
        res.push_back(path);
        return;
    }
    for(int i=0;i<n;i++){
        if(isvalid(i,count,path)){
            path[count][i]='Q';
            backtrack(n,count+1,path);
            path[count][i]='.';
        }
    }
}

bool isvalid(int col,int level,vector<string>&path){
    //上
    for(int i=0;i<level;i++){
        if(path[i][col]=='Q')
            return false;
    }
    //左上
    for(int i=level,j=col;i>=0&&j>=0;i--,j--){
        if(path[i][j]=='Q')
            return false;
    }
    //右上
    int n=path.size();
    for(int i=level,j=col;i>=0&&j<n;i--,j++){
        if(path[i][j]=='Q')
            return false;
    }
    return true;
}

vector<vector<string>> solveNQueens(int n) {
    vector<string>path(n,string(n,'.'));
    backtrack(n,0,path);
    return res;
}

};

```

79. 单词搜索

已解答 ✓

中等

🏷 相关标签

🔒 相关企业

A+

给定一个 $m \times n$ 二维字符网格 `board` 和一个字符串单词 `word` 。如果 `word` 存在于网格中，返回 `true` ；否则，返回 `false` 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1:

A	B	C	E
S	F	C	S
A	D	E	E

输入: `board = [["A","B","C","E"], ["S","F","C","S"], ["A","D","E","E"]]`, `word = "ABCCED"`

输出: `true`

注意：C++在做递归回溯算法相关题目时，递归函数形参传值和传引用运行速度有很大的差异。主要区别是一个是传值，一个是传引用。前者执行超时，后者可以通过

超时的時候試試引用傳參

第五周

图


DFS——深度优先搜索


对于有向无环图

797. 所有可能的路径

已解答 

中等

 相关标签

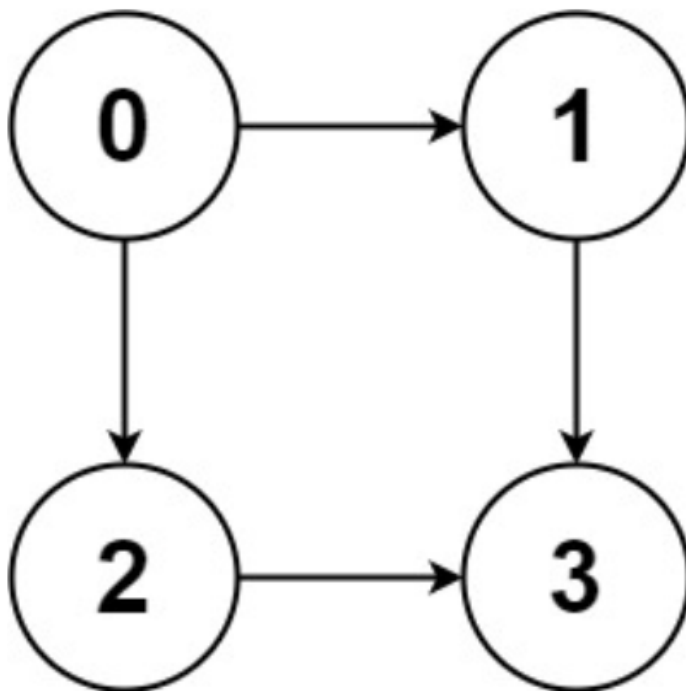
 相关企业

Aa

给你一个有 n 个节点的 **有向无环图 (DAG)**，请你找出所有从节点 0 到节点 $n-1$ 的路径并输出（**不要求按特定顺序**）

`graph[i]` 是一个从节点 i 可以访问的所有节点的列表（即从节点 i 到节点 `graph[i][j]` 存在一条有向边）。

示例 1:



输入: `graph = [[1,2],[3],[3],[]]`

输出: `[[0,1,3],[0,2,3]]`

解释: 有两条路径 $0 \rightarrow 1 \rightarrow 3$ 和 $0 \rightarrow 2 \rightarrow 3$

这里图的表示用的是二维数组，并且图的节点值是0, 1, 2, 3递增的，所以写DFS的很tricky

DFS就是回溯的一个子集，即要用回溯的方法

```

class Solution {
public:
    vector<vector<int>>>res;
    vector<int>path;
    void backtrack(vector<vector<int>>&graph,int count){
        if(count==graph.size()-1){
            res.push_back(path);
            return;
        }
        for(int i=0;i<graph[count].size();i++){
            path.push_back(graph[count][i]);
            backtrack(graph,graph[count][i]);
            path.pop_back();
        }
    }
    vector<vector<int>>> allPathsSourceTarget(vector<vector<int>>& graph) {
        path.push_back(0);
        backtrack(graph,0);
        return res;
    }
};

```

BFS——广度优先搜索

只要BFS只要搜到终点一定是一条最短路径。是因为BFS一圈一圈的遍历方式，所以一旦遇到终止点，那么一定是一条最短路径。

208.Trie前缀树

Trie（发音类似 "try"）或者说 **前缀树** 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

请你实现 Trie 类：

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word`。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true`（即，在检索之前已经插入）；否则，返回 `false`。
- `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix`，返回 `true`；否则，返回 `false`。

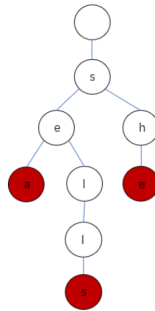
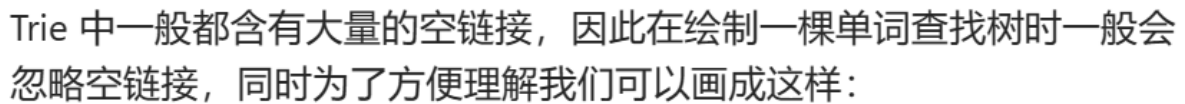
Trie 的结点是这样的(假设只包含'a'~'z'中的字符)：

```

struct TrieNode {
    bool isEnd; //该结点是否是一个串的结束
    TrieNode* next[26]; //字母映射表
};

```

它的真实情况是这样的：



```
class Trie {
private:
    bool isEnd;
    Trie* next[26];
public:
    Trie() {
        isEnd=false;
        //memset(next,0,sizeof(next));
        for (int i = 0; i < 26; ++i) {
            next[i] = nullptr;
        }
    }

    void insert(string word) {
        Trie* node=this;
        for(char c:word){
            if(node->next[c-'a']==NULL){
                node->next[c-'a']=new Trie();
            }
            node=node->next[c-'a'];
        }
    }
};
```

```

    }
    node->isEnd=true;
}

bool search(string word) {
    Trie*node=this;
    for(char c:word){
        if(node->next[c-'a']==nullptr)
            return false;
        else
            node=node->next[c-'a'];
    }
    return node->isEnd;
}

bool startswith(string prefix) {
    Trie*node=this;
    for(char c:prefix){
        if(node->next[c-'a']==nullptr)
            return false;
        else
            node=node->next[c-'a'];
    }
    return true;
}
};

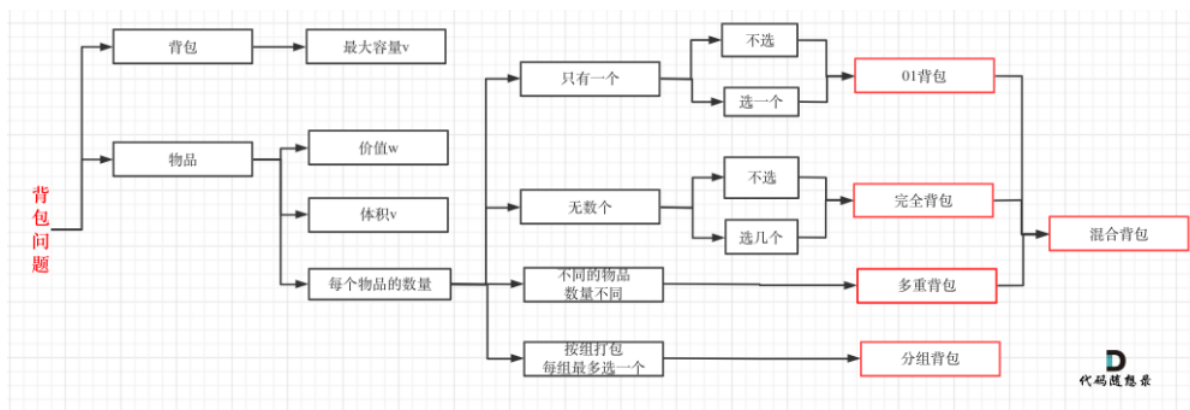
```

动态规划

动态规划问题五步曲：

1. 确定dp数组（dp table）以及下标的含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序
5. 举例推导dp数组

01背包



494. 目标和

已解答 ✓

中等

🔖 相关标签

🔒 相关企业

Aa

给你一个非负整数数组 `nums` 和一个整数 `target` 。

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

- 例如，`nums = [2, 1]`，可以在 2 之前添加 '+'，在 1 之前添加 '-'，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同 **表达式** 的数目。

示例 1：

输入：`nums = [1,1,1,1,1]`，`target = 3`

输出：5

解释：一共有 5 种方法让最终目标和为 3。

`-1 + 1 + 1 + 1 + 1 = 3`

`+1 - 1 + 1 + 1 + 1 = 3`

`+1 + 1 - 1 + 1 + 1 = 3`

`+1 + 1 + 1 - 1 + 1 = 3`

`+1 + 1 + 1 + 1 - 1 = 3`

求的是组合数目

解答：

假设加法的总和为 x ，那么减法对应的总和就是 $\text{sum} - x$ 。

所以我们要求的是 $x - (\text{sum} - x) = \text{target}$

$x = (\text{target} + \text{sum}) / 2$

此时问题就转化为，装满容量为 x 的背包，有几种方法。

这里的 x ，就是`bagSize`，也就是我们后面要求的背包容量。

大家看到 $(\text{target} + \text{sum}) / 2$ 应该担心计算的过程中向下取整有没有影响。

这么担心就对了，例如`sum` 是5，`S`是2的话其实就是无解的。同时如果 `S`的绝对值已经大于`sum`，那么也是没有方案的。

先进行数学推导，再解题

这里只是求组合数，如果要求列出组合结果就要用回溯了

五部曲:

1. 确定dp数组以及下标的含义

dp[j] 表示: **填满j (包括j) 这么大容积的包, 有dp[j]种方法**

1. 确定递推公式

有哪些来源可以推出dp[j]呢?

只要搞到nums[i], 凑成dp[j]就有dp[j - nums[i]] 种方法。

例如: dp[j], j 为5,

- 已经有一个1 (nums[i]) 的话, 有 dp[4]种方法 凑成 容量为5的背包。
- 已经有一个2 (nums[i]) 的话, 有 dp[3]种方法 凑成 容量为5的背包。
- 已经有一个3 (nums[i]) 的话, 有 dp[2]中方法 凑成 容量为5的背包
- 已经有一个4 (nums[i]) 的话, 有 dp[1]中方法 凑成 容量为5的背包
- 已经有一个5 (nums[i]) 的话, 有 dp[0]中方法 凑成 容量为5的背包

那么凑整dp[5]有多少方法呢, 也就是把 所有的 dp[j - nums[i]] 累加起来。

3.初始化

dp[0]=1

4.遍历顺序

nums放在外循环, target在内循环, 且内循环倒序。

代码如下:

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum=0;
        for(int i=0;i<nums.size();i++){
            sum+=nums[i];
        }
        if(abs(target)>sum)return 0;
        if((sum+target)&2!=0)return 0;
        int n=(sum+target)/2;
        vector<int>dp(n+1,0);
        dp[0]=1;
        for(int i=0;i<nums.size();i++){
            for(int j=n;j>=nums[i];j--){
                dp[j]+=dp[j-nums[i]];
            }
        }
        return dp[n];
    }
};
```

记得有符号串的时候判定的时候一定要加上引号啊啊啊啊啊啊啊啊

示例:

```
vector<string>a;
//如果a="123"
if(a[0]=='1')
//=====ATTENTION=====
//并不是
if(a[0]==1)
```

完全背包

完全背包即同一物品可以多次选择

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。

在C++中，INT_MAX是一个整数常量，表示int类型的最大值，但它只适用于非负整数

有时候需要初始化为最大值//初始化为最大值 `vector<int>dp(amount+1,INT_MAX);`

139.单词拆分：

给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

示例 1：

输入：s = "leetcode", wordDict = ["leet", "code"]

输出：true

解释：返回 true 因为 "leetcode" 可以由 "leet" 和 "code" 拼接成。

代码：

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        vector<bool>dp(s.size(),false);
        dp[0]=true;
        unordered_set<string>wordset(wordDict.begin(),wordDict.end());
        //多次+排序
        for(int j=1;j<=s.size();j++){
            for(int i=0;i<j;i++){
                string str=s.substr(i,j-i);
                if(dp[i]==true&&(wordset.find(str)!=wordset.end()))
                    dp[j]=true;
            }
        }
        return dp[s.size()];
    }
};
```

做题分辨：

- 组合or排列
- 一次or多次

多重背包

有N种物品和一个容量为V 的背包。第i种物品最多有Mi件可用，每件耗费的空间是Ci，价值是Wi。求解将哪些物品装入背包可使这些物品的耗费的空间 总和不超过背包容量，且价值总和最大。

每件物品最多有Mi件可用，把Mi件摊开，**其实就是一个01背包问题了**

注意：

```
for (int i = 0; i < n; i++) {  
    while (nums[i] > 1) { // 物品数量不是一的，都展开  
        weight.push_back(weight[i]);  
        value.push_back(value[i]);  
        nums[i]--;  
    }  
}
```

如果物品数量很多的话，C++中，这种操作十分费时，**主要消耗在vector的动态底层扩容上**。（其实这里也可以优化，先把 所有物品数量都计算好，一起申请vector的空间。

背包结束，继续动态规划了

打家劫舍

打家劫舍II

有环了

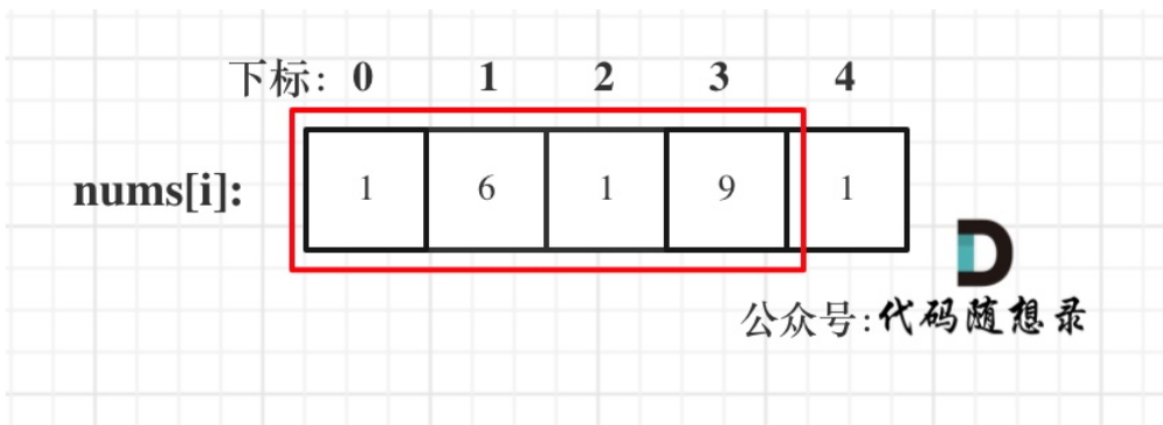
你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着**第一个房屋和最后一个房屋是紧挨着的**。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下，能够偷窃到的最高金额。

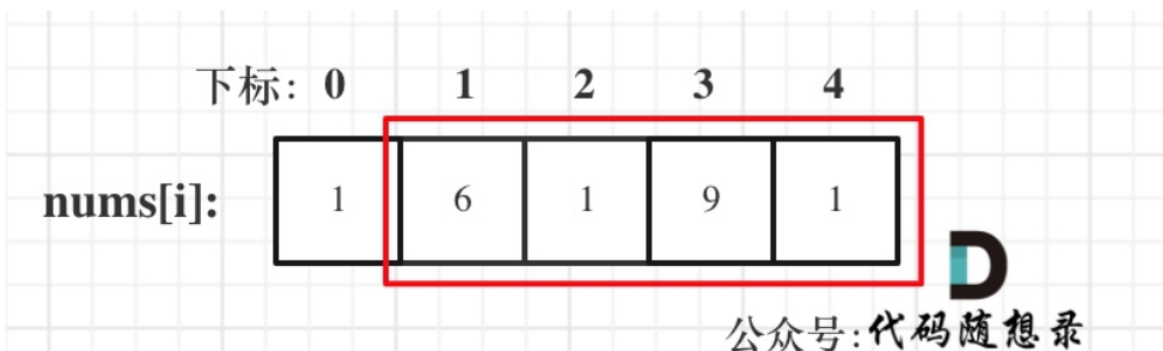
示例 1：

- 输入：nums = [2,3,2]
- 输出：3
- 解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。

- 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



解题时考虑这两个情况就可以了

又是一周4.1-4.7

打家劫舍III

树形结构，直接考虑递归

必须是后序遍历

这道题是树形DP的入门题目，通过这道题目大家应该也了解了，所谓树形DP就是在树上进行递归公式的推导。

股票系列

子序列系列

300.最长递增子序列

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。
例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

示例 1：

- 输入：`nums = [10,9,2,5,3,7,101,18]`
- 输出：4

- 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4。

1. dp[i]的定义

本题中，正确定义dp数组的含义十分重要。

dp[i]表示i之前包括i的以nums[i]结尾的最长递增子序列的长度

位置i的最长升序子序列等于**j从0到i-1**各个位置的最长升序子序列 + 1 的最大值。

所以：if (nums[i] > nums[j]) dp[i] = max(dp[i], dp[j] + 1);

动态规划有时候需要j来第二层循环的计算

674.最长连续递增序列

这个是连续，上面的题没有要求连续

不同之处在于不需要j了，nums[i]和nums[i-1]对比就行

```
for (int i = 1; i < nums.size(); i++) {  
    if (nums[i] > nums[i - 1]) { // 连续记录  
        dp[i] = dp[i - 1] + 1;  
    }  
}
```

718.最长重复子数组

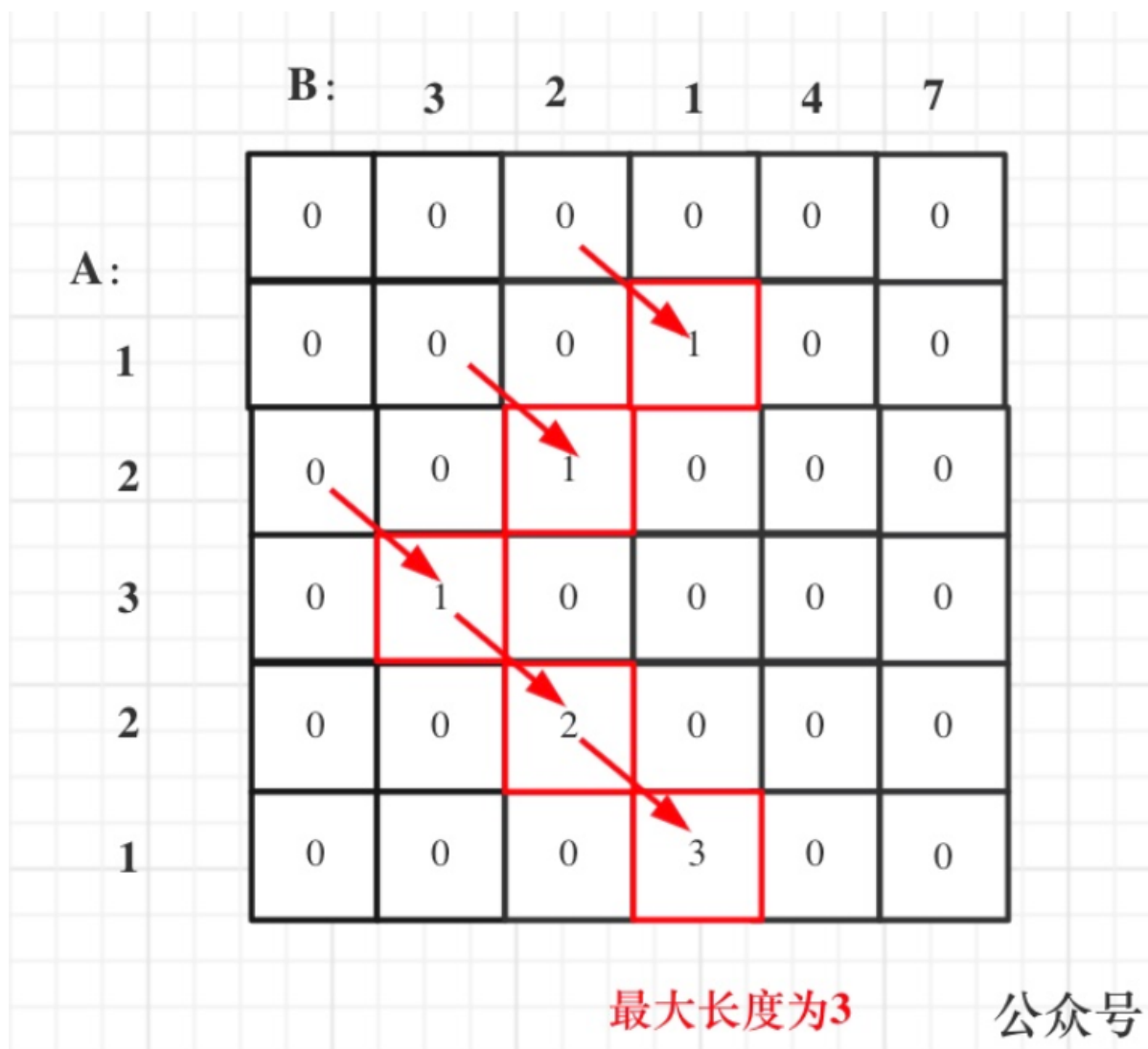
给两个整数数组 `nums1` 和 `nums2`，返回 两个数组中 **公共的**、长度最长的子数组的长度。

示例 1：

输入：nums1 = [1,2,3,2,1]，nums2 = [3,2,1,4,7]

输出：3

解释：长度最长的公共子数组是 [3,2,1]



关键代码

```
for (int i = 1; i <= nums1.size(); i++) {
    for (int j = 1; j <= nums2.size(); j++) {
        if (nums1[i - 1] == nums2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1] + 1;
        }
    }
}
```

最后从数组中找结果时要从dp数组的每一个元素里遍历来找，并不是最后一行数组，因为最长重复子数组可能在任一地方结束，那后面的元素的那一行又回归到0

子数组代表要求连续

44.最长公共子序列

没有要求连续了，但是相对顺序不能变

dp[i][j]的含义：长度为[0, i - 1]的字符串text1与长度为[0, j - 1]的字符串text2的最长公共子序列长度

递推图：

输入: text1 = "abcde", text2 = "ace"

		a	c	e
dp[i][j]		0	0	0
a		0	1	1
b		0	1	1
c		0	1	2
d		0	1	2
e		0	1	2

递推公式如下:

```
for(int i=1;i<=size1;i++){
    for(int j=1;j<=size2;j++){
        //=====注意这里是减1=====
        if(text1[i-1]==text2[j-1])
            dp[i][j]=dp[i-1][j-1]+1;
        else
            dp[i][j]=max(dp[i-1][j],dp[i][j-1]);

        if(res<dp[i][j])res=dp[i][j];
    }
}
```

最大子序和

dp[i]是包含dp[i]在内的子序, 要么延续前面的加上nums[i], 要么就是从i重新开始

$dp[i] = \max(nums[i], dp[i-1] + nums[i])$

判断子序列

给定字符串 **s** 和 **t**, 判断 **s** 是否为 **t** 的子序列。

字符串的一个子序列是原始字符串删除一些 (也可以不删除) 字符而不改变剩余字符相对位置形成的新字符串。(例如, "ace" 是 "abcde" 的一个子序列, 而 "aec" 不是)。

示例 1:


```
输入: s = "abc", t = "ahbgdc"
输出: true
```

这是编辑距离的入门题目

if ($s[i-1] \neq t[j-1]$), 此时**相当于t要删除元素**, **t如果把当前元素 $t[j-1]$ 删除**, 那么 $dp[i][j]$ 的数值就是看 $s[i-1]$ 与 $t[j-2]$ 的比较结果了, 即: $dp[i][j] = dp[i][j-1]$;

前面的最长公共子序列是s和t数组都可以删, 所以else的时候选择max的 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$;

不同的子序列

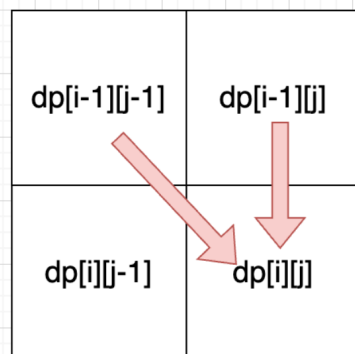
给你两个字符串 **s** 和 **t**, 统计并返回在 **s** 的 **子序列** 中 **t** 出现的个数, 结果需要对 $10^9 + 7$ 取模。

示例 1:

```
输入: s = "rabbbit", t = "rabbit"
输出: 3
解释:
如下所示, 有 3 种可以从 s 中得到 "rabbit" 的方案。
```

讲下个人理解: 这题可以看作一个01背包问题 s串是物品, t串是背包。sl是s串的长度, tl是t串的长度。那么题目可以转成从s串中任选tl个字符可以拼成t串的组合个数。我们定义 $dp[i][j]$ 为从s串中的前i个字符中任选j个字符可以拼成t串前j个字符表示的子串的组合数。那么**按照背包问题的步骤**对s串的第i个字符我们有两种选择:

1. **不选择第i个字符**, 那么 $dp[i][j] = dp[i-1][j]$
 2. **选择第i个字符**, 那么当 $s[i] == t[j]$ 的情况下 $dp[i][j] = dp[i-1][j-1]$
- 综合两种情况可以得到 $dp[i][j] = dp[i-1][j] + dp[i-1][j-1] (s[i] == t[j])$



4月5号

一天差不多4-5题吧, 进度还是挺慢