**算法训练笔记**

一维的数据结构要加速，往往要升维，因为多了一个信息

以空间复杂度换时间复杂度

自顶向下编程，往往具体实现的通用函数写在父类里面

一切复杂算法：**都是找最近重复性**

改进算法的思路：发现并去掉重复的步骤

1、从覃超老师布置的题目中，至少完成并提交 2 道代码作业

2、每周完成一篇学习总结你的作业初始满分为 100 分，最后作业总分，将占据毕业总成绩的 30%具体计分标准：。

• 每周没有按时（周日 23:59 前）提交作业，-2分

• 截至毕业，每缺一周学习总结，-4分

• 截至毕业，每缺一周代码作业，-6分

你的日常作业总分，将占据毕业总成绩的30%，大家一定要坚持提交作业，否则等于有装备没输出，技能得不到提升，看了再多也只是纸上谈兵。（完成作业只是提升过程中的一个可以量化的动作）。

班班时刻陪在你身边，请克制你的拖延。

# 【Git和GitHub使用】

参考《玩转Git三剑客》视频课程

## 【什么是版本控制】

Git是分布式版本控制软件，作者是Linus（Linux系统作者）

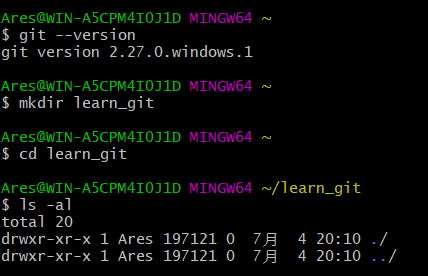
例如石墨文档，显示了不同的人修改记录，并且还提供了还原功能，让当前的作品可以还原当之前的任何一个版本上；

项目的代码总是在不断的迭代和优化；

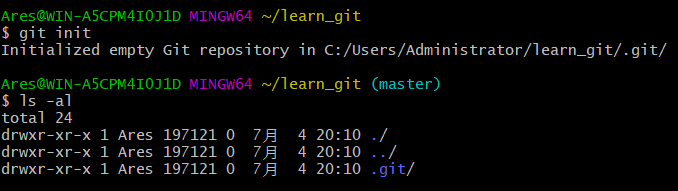
## 【配置git】

默认安装完毕后，windows和Linux下的是命令一模一样的；

1、查看安装信息与创建新目录（以后也要创建新目录的）

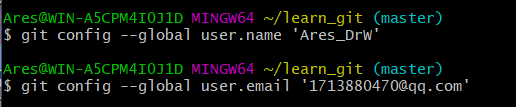


2、创建自定义的Git仓库（多了一个.git/文件夹）





3、配置仓库的个人信息（日后多人协作的时候用到）

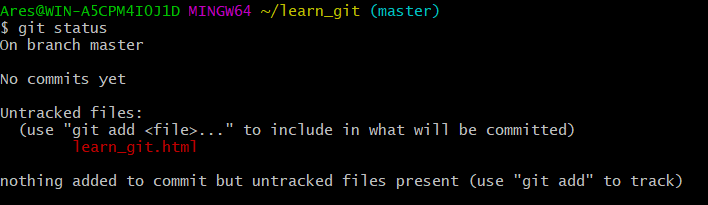




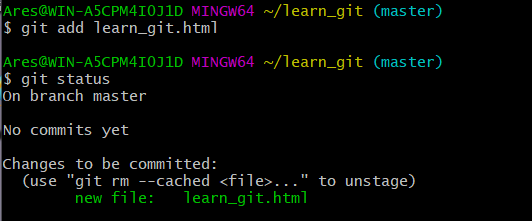
## 【Git操作流程：工作区、暂存区和仓库】



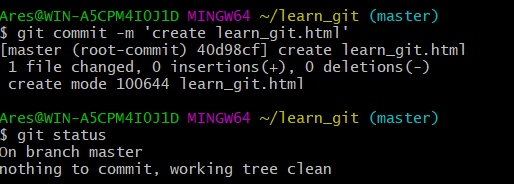
从无到有的创建文件加入到git目录下，git可以追踪到工作区存已修改文件



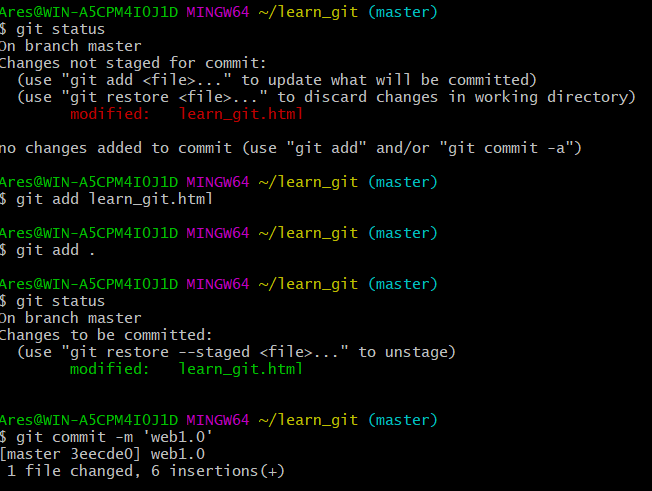
增加了git.add之后就是已经追踪的状态，这个时候就是已追踪：



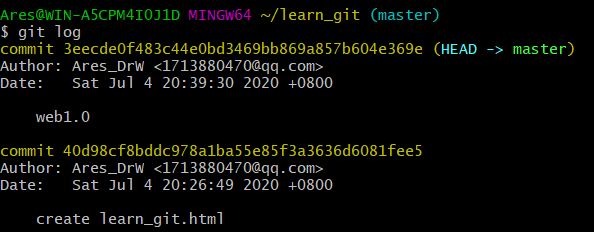
git commit [“update information”]->str之后就是已提交（工作树清零）



修改文件，重复上述过程，可以提交代码：



git log查看所有人员的提交目录：



实际工程应用中，应当把这个-m信息写的更加详细才行

## 【本地的仓库同步到GitHub上】

GitHub就是一个开源项目的聚集平台；

你还可以把修改后的代码提交给开源项目的管理者

总之，在GitHub你可以看到无数优秀的开源代码，如果可以的话，参与进去

新建GitHub仓库有两种模式，第一种是从头新建仓库，第二种是fork别人的代码。

在https协议下，你可以输入git global –config credential.helper store

首先就是建立公钥和私钥的链接：

**在本地的仓库目录下输入：ssh-keygen -t rsa -C '1713880470@qq.com'**

**到指定的目录里面去查看公钥和私钥 cd /c/Users/Administrator/.ssh**

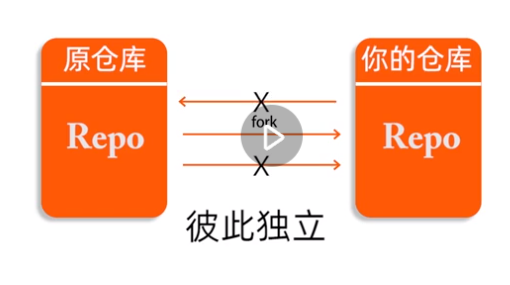
**拿出公钥：cat id\_rsa.pub**

**粘贴到个人账户（右上角的大setting中）新建一个ssh key**

**查看是否已经链接成功：ssh –T** [**git@github.com**](mailto:git@github.com)

## 【训练营作业提交流程】

首先要做的一件事情就是把老师的仓库fork过来：



fork就是在自己的账号下新建一个完全一样的git仓库



从命令行去push一个我到线上的仓库

git remote add origin [git@github.com:Ares-debugger/learn.git](mailto:git@github.com:Ares-debugger/learn.git) # origin名称！

git push -u origin master # push 到这个上面去

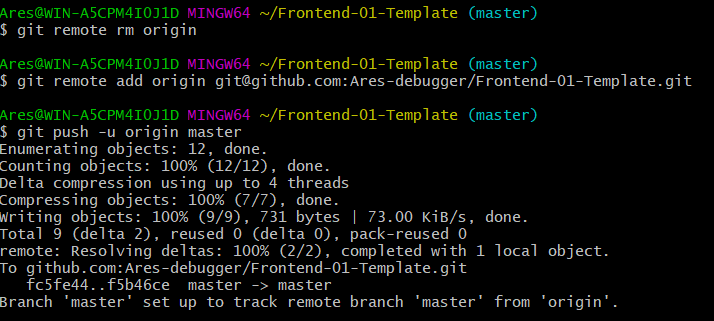
origin 相当于一个远程仓库的代号，我们可以起名

运行这个命令的时候，一定要在本地仓库的路径下

实际的话可能不太行，需要强推：git push origin +master

总结一下：更新文件后，add、commit，然后git强推。例如：

最后一次成功是因为，删除了全部的现有库，重新建立和在线库的连接。

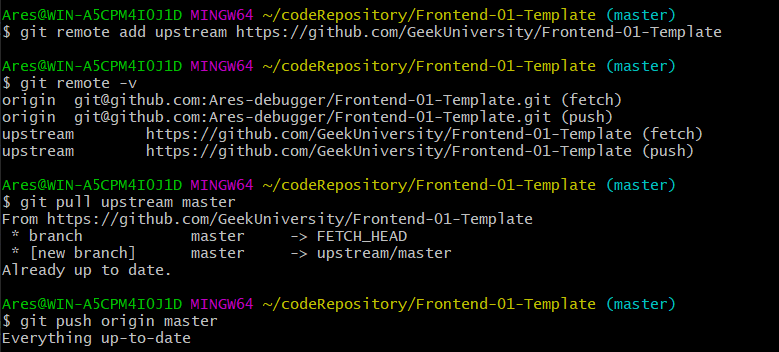


实际上，fork下来的仓库，origin是已经设置好的，直接推送就可以了。

[从新建仓库到同步仓库的完整流程](算法训练/同步仓库的流程.txt)

## 【保持fork的仓库和本账户仓库同步】

需要指定fork过来的仓库的upstream源：然后拉取到本地仓库即可



# 【如何学习算法—各种职业化训练的方式】

**三步走的方式：1、切碎知识点；2、刻意练习；3、反馈**

知识点：切碎成一个脑图——马斯克(SpaceX)认为，所有的知识都是一个树形结构，每一个知识都应该和你熟悉的知识最后挂在一起生成一个树形结构；要达到职业的话，一定要进行每个领域的职业化练习；

刻意练习：练到条件反射的地步！！练到精熟！基本功是区别业余和职业选手的根本。

过遍数！！**5遍刷题法**，必须这里

练习缺陷的地方，不要一直停留在舒适区；

反馈：国际站的第一视角；提交自己的作业

1、三分看视频理解：7分练习：

最佳看视频的方法：1.5-2倍速看，**难点反复看，反复加深记忆**

最差办法是原速看完只看一遍

2、摒弃旧习惯：

（1）不要死磕

（2）五毒神掌：敢于放手，**敢于**死记硬背代码

（3）不懒于看高手的代码，职业选手的第一视角（国际站高票代码）

3、**最佳方式：5分钟想不出来，直接用五毒神掌把别人的变成自己的**

死磕的话，没有借势而起的感觉，结果没有精力看高票答案

死磕的话，90%的学习那么多次为什么都放弃了

4、5字：过遍数，而不是每次花很长时间，过遍数！！！

5、不要AC了事，要看高票代码和高质量题解

6、建立已做代码仓库，每个题目要做5遍

# 【如何刻意的做题】

切题四件套：

（1）确保题目的理解是正确的，审题不清！！

（2）想所有可能的解法！！比较时空复杂度；

（3）写代码；

（4）测试样例多写几个

五遍刷题法：

（1）第一遍：

5-10分钟读题+思考

立刻看解法，比较解法的优劣（形成文档）

背诵并默写好的解法

（2）第二遍：

从无到有自己写->Leetcode上通过

每种解法比较、体会，想一想优化的方法

（3）第三遍：

24小时之后，对前一天做的题目进行训练->不熟练的地方专项练习

没时间的话就背下来，背下来总会吧！！

（4）第四遍：过了一周之后

重做，并且对不熟练的题目进行专项练习

（5）面试前一周进行恢复性训练

**最终要达到的效果：遇到这个问题，马上一二三解法是什么，模板是什么**

# 【LeetCode使用】

中文站：leetcode-cn.com

国际站：leetcode.com

任何一个题目，你只要把-cn去掉就可以来到国际站，最主要的就是这个discuss很不一样，上面有不少外国的巨佬贡献的代码。

# 【编程指法和快捷键的使用】

Home 和 end 去到行头和行尾

# 【刻意练习：自顶向下的编程风格】

关注最高的逻辑，逐层封装底层的代码

# 【相关数据结构导论】

## 【数据结构和算法密不可分】

数据结构和数据类型有区别。数据结构=数据元素（抽象）+数据之间的关系，数据类型=数据结构+数据操作。因此数据类型高于数据结构。算法来源于对数据结构的深刻理解。

## 【数据结构的分类和定义】

简单的来说，分为线性结构和非线性结构。

线性结构三大特点：

（1）有且仅有一个**开始节点**和一个**终端节点**；

（2）线性结构所有节点最多只有一个**直接前驱节点**和一个**直接后继节点**；

举例：线性表->栈->队列->串都属于线性结构

非线性结构就有对应的特点了：一个节点可能**多个直接前驱节点**和**多个直接后继节点**。

举例：数组，广义表、树、图、堆

## 【常用数据结构分析】

（1）**数组-矩阵-张量**，典型的非线性结构；

（2）**栈（弹夹）**：特殊的线性表，后进先出，常用于重要数据的现场保护；只能在***一个表的一个固定端进行数据结点的插入和删除操作***

（3）**队列**：先进先出：***只能在表的一端进行插入、在另一端进行删除***。

（4）**链表**：指针域和数据域

（5）**树**：在树结构中，有且仅有一个根结点，该结点没有前驱结点。在树结构中的***其他结点都有且仅有一个前驱结点***，而且***可以有两个后继结点***，m≥0。

（6）**堆**：是一种特殊的树，**根结点的值是所有结点中最小的或者最大的**，并且根结点的两个子树也是一个堆结构（区分为最小堆和最大堆）

（7）**图**：数据结点一般称为**顶点**，而边是顶点的**有序偶对**。

（8）**散列表**：字典思想。是最容易实现的。

## 【数据结构的运算（及其组合）】

（1）**检索**：filter满足一定条件的节点；

（2）**插入**：在数据结构中增加新节点；

（3）**删除**：把指定的节点从数据结构中去掉；

（4）**更新**：改变指定节点**一个或多个字段的值**；

（5）**排序**：按照某种顺序重新排列；

## 【设计数据结构要点】

（1）设计数据结构的“属性”；（2）设计类别

# 【面试考点】

（1）链表、数组、跳表的优缺点和增删改查时间复杂度（要非常清楚）

# 【七种时间复杂度分析】

Big O notation

**常数<对数<线性<平方<立方<指数<阶乘**



不考虑前面的系数，只看最高复杂度。

（1）优化时间复杂度会带来天壤之别的收益；

（2）数学方法通常可以非常好的做到这一点；

我们**只看执行的次数是否和n有关（执行了多少次）**。x重循环时平方与立方的复杂度；

O(logn)是怎么来的呢？

for(int I = 1;i<n;i=i\*2){print(x)} // x =2,4,6,….=>执行log2(n)次

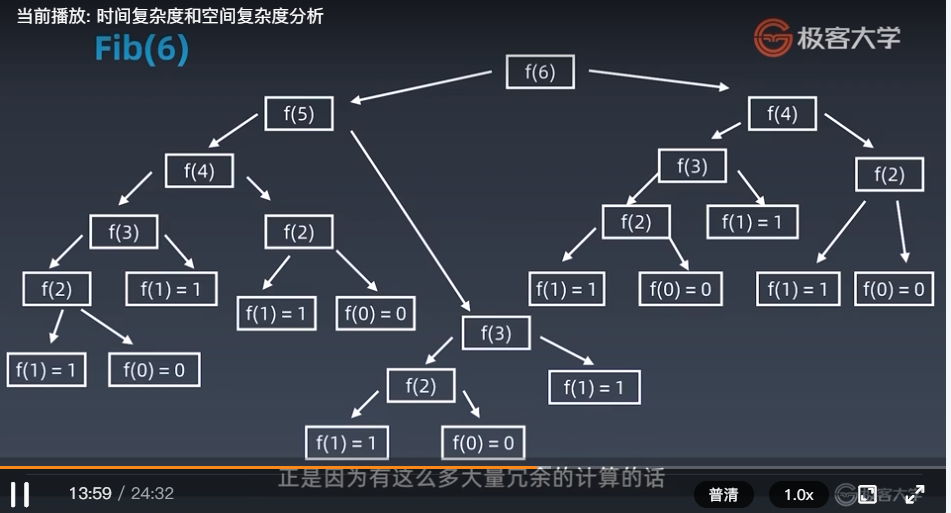
O(k^n)是怎么来的呢？其实是递归

求斐波那契数列的第n项

循环很好解决复杂度的计算，但是递归怎么做呢？**——画递归状态树**！！！

int fib(int n){if (n<2) return n; return fib(n-1) + fib(n-2)}

要计算F(6)->要计算F(5)和F(4)



发现两个问题：

（1）***每展开一层，节点数增加2倍***——这就是指数级的由来（其实一个循环就可以）

（2）***有重复的节点***——这就是改进的策略——增加缓存，存储中间变量

**利用“主定理”可以轻松的计算出任何一个分治或者递归算法的时间复杂度：**

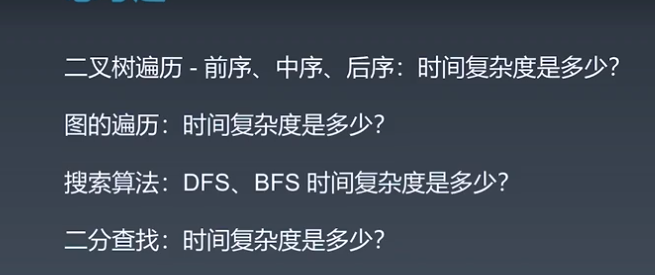
有4种情形，在平时是可以用上的：

1、二分查找——O(logn)

2、二叉树遍历——O(n) 虽然是分治，但是每一个是相同的（**每个节点仅访问一次**）

3、有序二维矩阵二分查找——O(n)

4、归并排序（最优排序算法）——O(nlogn)



前面三个都是：只要是每个节点有且仅有1次访问，那么时间复杂度就是O(n)

# ——————————————————

# 第一大部分：一维数据结构

双指针往两边扩散、双指针往里面收敛、双指针循环枚举、快慢双指针

# 【空间复杂度分析】

原则1：如果你开了数组，那么基本上就是O(n) 多维数组以此类推

原则2：如果有递归的话，递归最深的深度就是空间复杂度。这个深度不是展开节点的次数，而是从第一个节点到最深节点的步长。

例如计算Fib数列的话，是n层（因为一直到1）——空间复杂度是O(n)

加了缓存的话（记忆化搜索），重复计算过的就保存起来，不需要再次计算—时间O(n)

这种方法还是O(n)(时间和空间)

动态规划法：一样是开n的数组

再次优化，只要存n-1和n-2就可以了，这样把时间降低到O(n)，空间降低到O(1)

# 【数组—为了快速查询】

开辟一段数组，就是向内存管理器开辟一段**连续空间**。访问下标的时间复杂度是O(1)

数组的问题：增加删除数组元素的时候：插入操作的时候，要改变插入元素后面所有的内存地址。同样，删除操作也是，这时候是O(n)的时间复杂度（平均是O(n/2)不考虑常数）

在源代码中是如何实现的呢？源代码有一个copy操作，如果我要添加，就要把index后面的部分**（以index开头的部分）复制到index+1开头**。最后让data[index] = elem。如果你要对array或者ArrayList进行大量的增加和删除操作的话，会导致大量的copy操作。

# 【链表—为了快速修改】

**在添加、删除操作特别频繁的情况下**：应该使用链表。元素定义好之后，有value和next两个属性，一般来说都要定义class Node。通常情况下头指针叫Head，尾指针叫tail，链接方式不同分为单向链表、双向链表和循环链表

class LinkedList{int data; **Node next**; Node(int d){data=d;}}}

链表增删都是常数时间复杂度。**但是检索的时候就很麻烦了，就是O(n)的时间复杂度**。

# 【跳表—有序链表的改进】

针对链表在lookup操作上是O(n)的复杂度，因此又有提出来了改进措施：

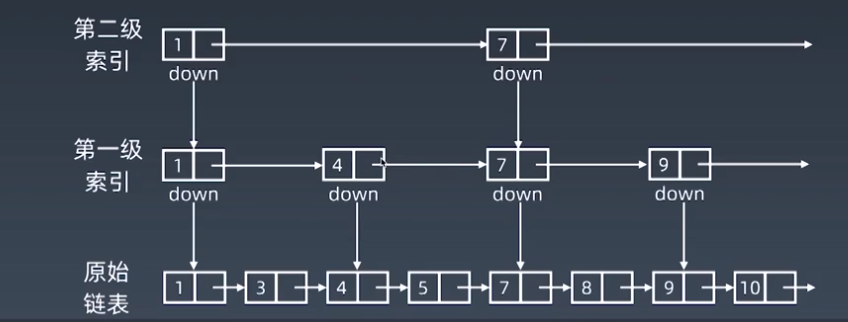
首先要明确一点，O(logn)<O(n)的，那么，能用二分查找而坚决不用遍历。好，什么情况下能用二分查找？就是**“有序”**的时候并且是数组。

有序链表如何加速呢？跳表对应的是平衡树，二叉搜索树中的平衡树和二分查找。由于出现的晚，一些老项目里面都是平衡树。新项目用的都是跳表来取代。

一维的数据结构要是想加速，肯定要升维，因为升维之后，可以用附加的信息来取代“挨个走完”才能获取到的信息。这样一来，就可以**使用增加索引的方法**。

采样构造索引，以空间换时间。例如，我们要查8，先比较1和7，比第一级别索引都大，那么直接往下找，发现在7和9之间，再往下找，到原始链表里面。链表长度越多，那么构建的索引也就越多。

跳表其实就是stride扩大了。最后的时间复杂度就是O(logn) 缺点在于，跳表维护成本很高。增删元素的话，要更新所有的索引。LRU缓存算法就是，高效的空间换时间



空间复杂度是O(n)（不考虑前面的系数项）

# 【数组&链表&跳表算法实战】

做题五毒神掌：

（1）思考5-10分钟，思考不出来马上看题解不纠结（你想不出算法的）；

（2）背诵、默写直到熟练；

（3）自己写：闭卷考试

## 针对数组的算法1：快慢双指针算法

**有一种叫快慢双指针，一种叫左右双指针，左右双指针要用的话，一般来说都需要是有序的，一般都是判断条件，if fun(left,right)满足一定条件的话left+1 or right-1**

**要点：慢指针始终指向我关心的部分，快指针每次移动1，慢指针只有在接收到快指针的检索信号的时候才会移动。**

### 案例1：移动0：[0,1,0,3,12]->[1,3,12,0,0]

解法1：开辟新数组，统计0的数量，遍历后加上，时间O(n),空间O(n)

zero\_count = 0

b = []

for I in range(len(nums)):

if nums[i] == 0: zero\_count +=1

if nums[i] != 0: b.append(nums[i])

for I in range(zero\_count):

b.append(0)

解法2：inplace操作，**加上一个j，始终指向非零元素要填充的地方**

j = 0 # 非零元素要填充的下标

for I in range(len(nums)):

if nums[i] ！= 0:

nums[j] = nums[I] # 执行填充操作

if i!=j：

nums[i] = 0 # 只有发生了交换，才可以给0

j +=1 # 这时才可以放心的让j指向下一个位置

时间O(n),空间O(1)

解法3：交换法，实际上有一点冒泡排序的意思了，但是要注意，只记录第一个0的坐标！！只记录第一个0的坐标！！！实际上这个问题里面，f\_z\_idx应该加1，而不是赋值为i

first\_zero\_index = 0

for i in range(len(nums)):

if nums[i] != 0:

nums[first\_zero\_index],nums[i] = nums[i],nums[first\_zero\_index]

first\_zero\_index += 1 # 这就是不应该，连续两个0的时候，就会出错

### 案例2：找出不重复的集合长度：[1,1,2,2]->[1,2] 2

按照这个思路，慢指针的覆盖范围，就是最后输出“没有重复数字的元素集合”

每次循环，**如果快指针指向的值不等于慢指针指向的值，则表示出现了新值**

首先应该让慢指针移动到新的位置，再进行赋值。

current = 0

for I in range(1,len(nums)):

if nums[index] != nums[current]: # 出现了新值

current += 1

nums[current] = nums[index] # 后面的往前移动

### 案例3：11.盛水最多的容器(双指针&优化更新法)：

left和right双指针，移动较小值的指针，达到最好效果。

首先，高肯定是max(heights[left],heights[right]),宽是多少呢？

每次比较heights[left]和heights[right]的大小，向内移动那个较小的

喜大普奔，一次跑通：时间复杂度O(n)，空间复杂度O(1)

def maxArea(self, heights):

"""

:type heights: List[int]

:rtype: int

"""

left=0

right = len(heights)-1

maxAre = min(heights[left],heights[right])\*(right-left)

**while(left < right):** # 用for I in range(len(heights)) 也是可以的

if heights[left] < heights[right]:

# 移动较小的

left += 1

else:

right -= 1

**maxAre = max(maxAre,min(heights[left],heights[right])\*(right-left))**

**# 一个max函数完成了所谓的“动态更新”：**

return maxAre

注意双指针循环体的写法：开始时并不知道循环n次，因此可以使用while进行，条件是left和right是分治的。实际上每次移动一个指针，所有指针遍历所有的节点，那么就是

### 案例4：两数之和&三数之和

先用暴力破解的办法把两数之和解决了，hash表的方法日后再说：

res = []

for i in range(len(nums)-1):

for j in range(i+1,len(nums)):

if nums[i] + nums[j] == target:

res.append([i,j])

return res

三数之和，其实就是a+b = -c 要求更高，要进行去重了。但是list嵌套的情况下，是不可以进行set去重的。只好新开一个列表去重；

暴力破解法：

res = []

for i in range(len(nums)**-2**):

for j in range(**i+1**,len(nums)**-1)**:

for k in range**(j+1**,len(nums)):

if nums[i] + nums[j] + nums[k] == 0:

res.append(sorted([nums[i],nums[j],nums[k]]))

res\_2 = []

for item in res:

**if item not in res\_2:**

res\_2.append(item)

return res\_2

双指针方法：先排序，省去一重循环，一言不和左右双指针夹逼：

res = []

sorted\_nums = sorted(nums)

for i in range(len(sorted\_nums)-2):

left = i + 1

right = len(sorted\_nums) -1

while(left < right):

if sorted\_nums[i] + sorted\_nums[left] + sorted\_nums[right] < 0: # 移动左边

left += 1

elif sorted\_nums[i] + sorted\_nums[left] + sorted\_nums[right] == 0:

res.append(sorted([sorted\_nums[i],sorted\_nums[left],sorted\_nums[right]]))

left += 1 # 否则写成死循环

else:

right -= 1

# 去重代码

### 案例5：反转链表

### 案例6：环向链表

是否存在节点指向前面？

方法1：Hash表，访问过的节点都放在里面；方法2：快慢双指针

# 【最近相关性：栈stack：pop是最重要的操作】

添加删除操作都是O(1)，要查询的话都是遍历一遍；

**操作：进栈出栈取栈顶：**

**stack：直接搜python stack：**

**pop：弹出栈顶元素并返回；**

**push：往栈里面加**

**stack[-1]：仅仅取栈顶**

**length()：取出长度**

**什么样的问题可以用栈来解决？——具有“最近相关性”，什么叫最近相关性呢？就是像洋葱一样一层层剥开的结构，例如有效的括号问题：**

## 针对栈的算法1：栈本身

### 案例1：有效的括号：

思路：如果是xxx就压到栈里面去，如果xxx就和栈顶元素匹配；

这里面，**如果是左括号，就压到栈里面去，如果是右括号，就和栈顶元素匹配，匹配上的话，弹出栈顶元素**。如果不匹配的话直接返回False，最后如果这个栈回到空的话，就说明是True。

栈在python中的实现就是列表，列表可以append和pop

dic = {'(':')','[':']','{':'}','?':"?"}

stack = ['?']

for c in s:

if c in dic:

stack.append(c)

elif dic[stack.pop()]!=c:return False # 如果列表是空的话，弹出操作会导致失败

return len(stack)==1

## 针对栈的算法2：双栈操作

### 案例2：设计一个栈并且O(1)时间内查到最小元素

辅助栈始终保持等量的最小元素

class MinStack(object):

def \_\_init\_\_(self):

"""

initialize your data structure here.

"""

self.data = []

self.min\_aux = [] # 始终保持等量的最小值

def push(self, x):

"""

:type x: int

:rtype: None

"""

self.data.append(x)

if len(self.min\_aux)==0 or x < self.min\_aux[-1]:

self.min\_aux.append(x)

else:

self.min\_aux.append(self.min\_aux[-1]) # 防止pop出去的太多了影响最小值

def pop(self):

"""

:rtype: None

"""

if self.data:

self.data.pop()

self.min\_aux.pop()

def top(self):

"""

:rtype: int

"""

if self.data:

return self.data[-1]

def getMin(self):

"""

:rtype: int

"""

if self.min\_aux:

return self.min\_aux[-1]

### 案例3：只用栈实现队列，只用队列实现栈

## 针对栈的算法3：

### 案例4：柱状图的最大面积

**暴力法1**：固定2找1

两层循环，找(I,j)之后，找到I,j之间最矮的柱子，然后就可以了；

这样的话其实是O(n\*\*3)的时间复杂度

**暴力法2**：固定1找2

加速：当前的左右边界不断遍历

for I -> 0,n-1:

找到left bound 和right 边界，例如5，左边第一个比他小的是2，右边第一个比他小的是2，这样的话，这个i能达到的范围就是2个—得到当前的计算结果

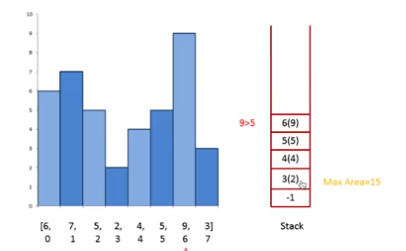
栈法：为什么用栈来解决呢？我们开始暴力办法的时候，找左边界和右边界能不能更快一点？其实我们每次到一根棒子上都要进行枚举，那么实际上很费时间。我们希望第二次遍历左边的时候，**可以O(1)求解而不是O(n)->需要缓存一下**。

——**只要有一个有序的栈，按照高度排列出来的话，最小值应该马上就可以知道的**。

——右边的话其实也合理，我不探的话永远都不知道。

——这题的思路就是维护一个有序的栈，从栈底到栈顶是从小到大排列的->在入栈的时候要进行一些处理：

核心逻辑：如果一个新元素加入栈的话，发现比栈顶元素大，表示右边界没法确定，**一直累加起来，如果比栈顶元素小的话，说明当前栈顶元素的左右边界都是可以确定的，左边界是栈顶元素的下一个（因为比他小）**，右边界是新加入的元素，从而需要把栈顶元素弹出，进行栈顶元素的Area计算，把当前元素加入到里面来。



# 【先开后到：队列queue】

先入先出都是O(1)的。

qsize()->队列大小

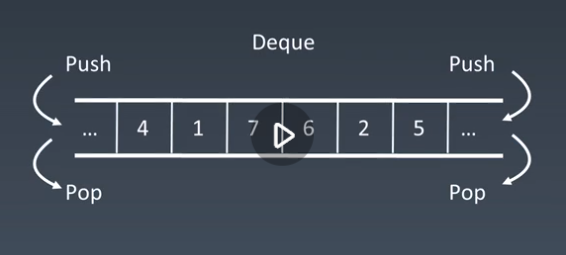
empty()和full()队列为空或者为满

put(item)

# 【双端队列deque】

工程代码中直接使用collections.deque

实战中，都是双端队列：其实就是stack和queue的结合体。可以往前面和后面添加或者删除数据。deque（double ended queue）



deque：直接搜python deque。

enqueue(item)->return self.queue.append(item)

dequeue(item)->return self.queue.pop(0)

size->len(self.queue)



## 针对双端队列的算法1: 滑动窗口

### 案例1：滑动窗口中的最大值

暴力：for I-> 0,n-1

for k -> i+1,i+k:

temp = nums[i+1]

if …

直接用双端队列来求解

# 【元素优先级(条件)队列】

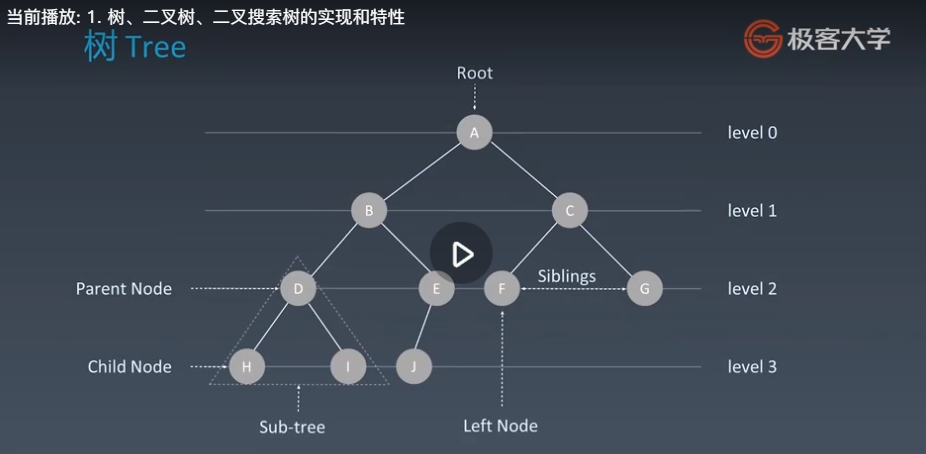
取出操作是O(logn);顺序不再是先入先出或者先入后出了，而是按照**元素的优先级**取出；其实是保持了一定的有序性：底层实现的较为复杂的，可以是heap、二叉搜索树或者是平衡二叉搜索树；

优先级如何定义呢，在java当中，你每次放进去的函数必须实现这样一个comparater（）这个函数

# 【树】

树就是一种典型的二维数据结构。

这样想，如果链表的next指针不是指向一个节点，而是指向多个节点。例如next->(next1,next2,next….) 那么就变成二维的数据结构了。（也就是多个后继结点）



树和图最关键的节点是什么？最关键的区别是“有没有环”。也就是有节点指回去。链表是特殊化的树，树就是特殊化的图。

Python中树的定义：

class TreeNode:

def \_\_init\_\_(self,val):

self.val = val

self.left, self.right = None,None

C++ 中树的定义：

struct TreeNode:

int val;

TreeNode \*left;

TreeNode \*rignt;

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

出现树形结构的原因：

（1）***递归***：Fibo数列的递归，就是一个树状的节点

（2）***随机状态转移***：状态向外扩散选择不同的状态

注定了搜索算法和树的姻缘

如果树的节点都是无序的，那么就需要遍历。

三句话递归着就可以找，三句话的顺序就是不同的**递归方式。**

前序：根-左-右；中序：左-根-右，后序：左-右-根

算法原型

def preorder(self,root):

if root:

self.traverse\_path\_append(root.val)

self.preorder(root.left)

self.preorder(root.right)

def inorder(self,root):

if root:

self.inorder(root.left)

self.traverse\_path\_append(root.val)

self.inorder(root.right)

def postorder(self,root):

if root:

self.postorder(root.left)

self.postorder(root.right)

self.traverse\_path\_append(root.val)

——————————————————————————————

class Solution:

def inorderTraversal(self,root):

"""

:type root: TreeNode

:rtype: List[int]

# **需要定义一个辅助的递归函数**

"""

res = []

def helper\_inorder(root):

if root:

helper\_inorder(root.left)

res.append(root.val)

helper\_inorder(root.right)

helper\_inorder(root)

return res 为什么基于递归呢？因为树的定义，本身没法有效的进行循环。

***最好的方式是，直接对每个根节点调用相同的遍历函数***

## 【二叉搜索树】

遍历一个树就是O(n)的复杂度，那么和一个链表没有太大的区别。二叉搜索树就是排序：要求**左子树所有节点**都小于根节点，**右子树所有节点**都大于根节点，且以此类推。中序遍历是一个升序的遍历：

插入、查询操作都是O(logn)的，这是因为，每次查询节点直接和根节点比较，可以砍掉一半的节点，类似于实现了二分查找——O(log2n)

插入的话也是一样，最后没找到，停下来的地方就是应该插入的地方。

优点在于插入的操作完全不需要干扰其他任何节点。

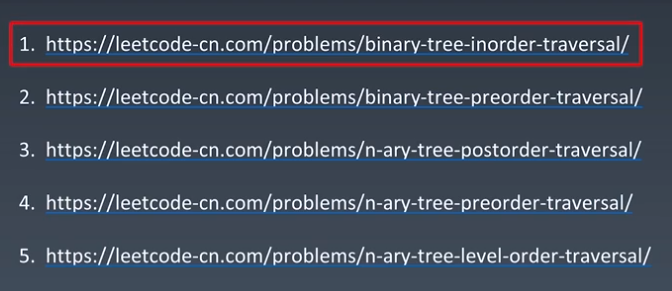
### 【创建一个二叉搜索树】

把**所有节点以此插入到这个搜索树**里面去即可。

### 【删除节点】

叶子节点直接删掉就可以了；

根节点的话，不能说把它的子树全部删掉了啊！！应该让一个新节点充当原来的位置！！应该是找一个**紧邻根节点但是小于根节点**的点&刚刚大于根节点的（右子树里面最小的节点）。题目主要是：二叉树和N叉树的中序遍历



简单讲讲递归和栈的关系：每次递归都是在新开的栈里面压入函数和参数，因为栈秉承后进先出的原则，最里面的肯定是最开始的要解决的函数，因此递归就是栈。

递归也不见得就比循环差，**这个锅不应该递归来背，而是你自己没用缓存把递归写残了**。现在的编译器对于递归的优化，都是很高效的。递归的主要是一因为要开一些栈，而不是算法本身上的问题。

如果要推广到N叉树的话，也可以进行处理：N叉树就没有中了，只有前后和层序遍历：时间复杂度和空间复杂度都是O(n)

class Node(object):

def \_\_init\_\_(self, val=None, children=None):

self.val = val #

**self.children = children # 这就是一个列表了**

class Solution(object):

def postorder(self, root):

res = []

def helper\_postorder(root):

if root:

for child in root.children:

helper\_postorder(child) # 只需要

res.append(child.val)

helper\_postorder(root)

if root:

res.append(root.val)

return res

# 【递归】

递归适用于子结构也具备“相似性”也就是整体具备“自相似性”的情况。

递归本质上就是循环，只不过是使用**循环体不断的调用自己**完成循环。计算机底层其实是不支持循环而是支持递归的，这是因为他只会不断的跳转到函数体去实现。

递归Recursion 从前有座山……《盗梦空间》

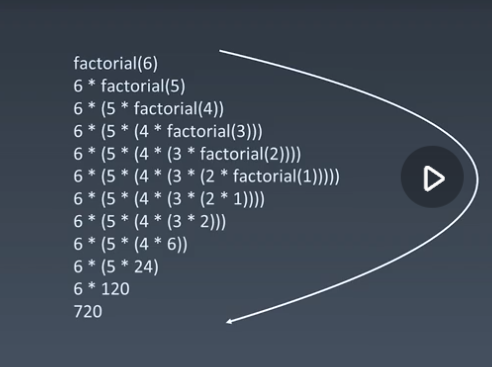
## 【递归的特点】

（1）一层一层的下，一层一层的回来；

（2）**通过函数的参数完成不同层之间的状态传递**；

（3）每一层的环境和其他的都是一份拷贝，互不影响

递归的运算，一层的话是递归栈（剥洋葱），两层的话是递归树：



**在调用递归的时候，系统会自动给我们做出来一个调用栈**。

## 【递归代码模板——啪啪啪写上去】

def recursion(level,param1,param2……):

# resursion terminator

if level > MAX\_LEVEL:

return process\_result

# process logic in current level

process(level,data……)

# drill down

self.recursion(level+1,p1….)

# reverse the current level status if needed

第一步：先把递归终止条件写出来。

第二步：处理当前层的逻辑（实际上是参数之间的运算）

第三步：下到下一层，把level+1，把参数放到下一层（其实是前面改变的）

第四步：如果有清理当前层的话，就清理，环境和全局变量可能需要清理

## 【递归思维要点】

（1）熟练后抵制进行人肉递归与人肉尝试算法（因为本身很简单）

（2）找到最近最简的方法，将其拆解成可重复解决的问题（**最近重复子问题**）

（3）数学归纳法思维，最简单的条件是成立

## 【递归实战解析】

### 爬楼梯问题：假设有n级台阶，你每次可以上1或者2级，问一共有多少种解法

这里面，一定要找到“重复性”，**重复性往往存在与相邻两次的结果中**：

为什么说递归难？**因为递归是基于结果而不是基于已知的**。

这里比较厉害了，递归的终止条件是1级或者2级台阶。

再往上我们就不要穷举了：**而是把n-1的结果当成已知**：

例如，n=3的时候，有两种情况，第一种是n=1的解法往上走1+1，第二种是n=2的解法往上走一步。第三种是n=1的解法往上走2

抽象出来，最大的重复性就出现了：

假设结果为f(n)，f(n)=f(n-1)+1 **& f(n-2)+2 & f(n-2)+1+1（这是不对的）**

**f(n-2)+1+1 ⬄ f(n-1)+1**

并且这里也没有漏算的情况，也没有重复的情况

mutual exclusive complete exclusive （分项都是互斥的）

但是这样的话也不够，不是一个通项公式啊,题目的要求并不是打印出所有的“方法”，而是打印出“有多少种”

f(1)=1 f(2)=2 f(3)=f(1)+f(2)=3

这是因为：f(n-1)表示我已经站上台阶了**的方法**，对于每一个f(n-1)方法，我是不是只需要+1就可以获取到站在f(n-1)上往f(n)走的数量；对于每一个已经站在f(n-2)上的方法，那么我只有一步跨两个台阶的方法。（**有一种是重复的**）

因此自然可以写出：

f(n)=f（n-1）+f（n-2）

斐波那契数列有几种方法：傻递归：

def climbStairs(self, n):

"""

:type n: int

:rtype: int

"""

if n==1:return 1

if n==2:return 2

return self.climbStairs(n-1) + self.climbStairs(n-2)

很明显，超出时间限制

第二种：**动态规划更新状态**的方式，实际上整体只有三个数在动：

f(n),f(n-1)和f(n-2)

我**只需要一个列表循环滚动过去即可**,且不需要其他中间变量

if n < 3: return n

f1,f2,f3 = 1,2,3

for I in range(n):

f3 = f1 + f2

f1 = f2 # 滚动

f2 = f3 # 滚动

return f3

一下子把时间复杂度从O(2^n)降低到了O(n)，空间复杂度从O(n)降到了O(1)

左边括号随时可以加，但是不要超过n，右括号不能出现的情况，必须之前有左括号，且左括号个数大于右括号个数