计概复习：语言、算法与优化

写在开头

马上要期末考试了，照我一贯的习惯要把学过的东西整理一遍（就当做复习吧）。正好不知道考试的时候cheat sheet带什么，于是干脆边整理边敲下来，到时候带进考场或许会有点用，也可以和同学们分享一下我对某些知识的理解。当然这篇文档可能不太适合作为cheat sheet，原因大概有如下几点：

1.我一直很不理解cheat sheet存在的意义。我觉得考试的时候应该用自己完全掌握的东西，而不是知道某些东西然后照抄模版。。。。。。不过确实有些东西自己不是很熟或者再写一遍比较耗时间，既然老师允许，我就也得准备一点。。。。。。所以这篇文档主要目的是用于整理和复习，而不是记录模版代码

2.我对课程内容和算法确实很感兴趣，但是我更关注的是算法背后的思考方式和逻辑而不是具体的实现过程，所以我实现出来的代码常常不太美观，可能没有太大的直接借鉴的价值（注：这篇文档里所有的代码都是我自己现写的，可以保证能实现目的以及复杂度数量级正确，具体细节和美观度不做任何保证）

3.我比较习惯于完全想明白一种方法再去接受它，更多是接受那些自己能想出来的方法，所以做的题其实很有限，了解的方法也很有限。还有一些方法我可能明白它的原理，但是由于它不太符合我的审美而没有欲望去仔细理解它的实现（这确实是我的问题，但是有的东西我真的觉得很恶心。。。）

也就是这篇文档可能以对方法的理解和我对方法背后思维的剖析为主，逻辑味和数学味可能比较重。。。。。。

4.写得太长了。。。。。。我也不知道我怎么就写了这么久这么多，写到一半都想放弃了。。。。。。

尽量涵盖了我所能想到的考试范围内的知识点，但是不排除忘记了一些东西，也可能漏掉了很多小技巧（有些是我觉得很无聊，更多是我根本就不会。。。。。。）

一、语言（python）

这一版块主要写和python语言特性以及具体实现相关的一些内容，一些库会在这里提出而在“优化”版块中介绍具体应用场景。其实这是我最不擅长的一块，无数的函数和语法糖我都不知道。。。。。。这里只总结一些我比较了解和常用的东西。

1.输入输出与字符串处理

输入处理

str.split(sep=None) 最常用，几乎无处不在。注意有时候需要加上sep参数以在指定的地方分隔，以及要确认分隔出的段数是否统一，形如a,b = input().split()在input不是分出恰好两段的时候会导致RE。如果不知道具体分出几段可以直接使用input().split()，会返回一个列表

split也常和map配合使用，例如将每个输入都转为整数。还是同样的问题，确认会被分成多少段；另外注意map返回的是一个map对象，要想索引或切片之类的得转成list

例如info = list(map(int,input().split()))；也有info = [int(x) for x in input().split()]的写法，都行

字符串处理

str.title()首字母大写（每个单词） str.lower()/upper()每个字母小/大写 str.strip()去除空格，有相应的rstrip/lstrip去掉尾部/头部的空格 在格式化字符串时可能有用（查到strip也可以带参数，去掉指定的某些字符，但是没用过）

空格处理有时候很麻烦（OJ04030 统计单词数）

字符串是不可变对象！可以取索引、切片、连接（“+”），但是不能修改。如果需要修改用列表会更好（不然就得每次弄一个新的字符串，浪费内存）

ord() chr() 可以完成字符与ASCII码的转化，有些时候（比如要把字母和其在字母表内的顺序对应时）有用（但是我很少用）

str.find()查找指定字符，注意如果有的话会返回第一个找到的，如果没有会返回-1而不是报错！（这些特性有时候很好用但是另一些时候可能导致错误）

str.zfill()自动在前面补0补到所需位数

格式化输出

python格式化输出有三种方式：C风格的print("%d,%.1f" %(ans,res))，str.format方法print('{},{:.1f}'.format(ans,res))以及比较新的特性f-string print(f'{ans},{res:.1f}')。其中format方法和f-string可以带各种参数，自由度非常高（但是我基本上只会用第一种。。。后面两种用的很少而且很不熟）

另外在输出时常用的str.join(list)方法，可以简洁输出一个列表中的元素，但是注意这里的list里面的元素必须是str类型，否则会报错

print是可以带sep和end参数的，设置输出之间的间隔和输出之后的结束字符（默认sep=' '也就是print两个数之间会自带空格；end=' \n'也就是输出之后会自动换行）有时候用这些参数可以方便格式化输出

2.数学运算

Python支持大整数运算（这点比C好太多了），但是注意数很大的时候运算可能会很慢，所以应当用提前取模的办法尽量避免超大数的运算（OJ02706 麦森数）。

float('inf')表示无穷大，在初始化某些边界值时可能有用（当然你也可以写一个很大的数，但是万一脑子一抽数的大小没写够。。。。。。就尴尬了）

Python的float自然也有精度问题，所以很多公式数学上没问题但是会WA。。。尽量用int运算（OJ01064 网线主管），有除法的时候如果确保整除，用//代替/（OJ18155 组合乘积）。除法还要注意是否可能出现除以0的情况（OJ18223 24点）

舍入时注意round不是严格意义上的四舍五入，遇到恰好.5会向偶数舍入。floor和ceil是安全的（当然如果刚好是整数也可能会有精度方面的问题）

float的等于判断不能用“==”，要用绝对值的差小于某个极小量（或者用math库中的isclose）

绝对精确的小数运算可以用decimal库，这是另外一个类，不是用浮点存储的所以不会损失精度（但是我没用过，好像大部分时候也没必要用）

Python也支持复数运算，但是由于精度问题很多数学公式用上去会WA，所以似乎没什么用

进制转化：bin,oct,hex 注意得到的是带前缀的字符串

math库：各种常用数学运算都有。最常用的sqrt(当然似乎可以用**0.5代替),对数、三角、反三角也都有；还有e,pi等常数，inf表示无穷大；前面提到的floor,ceil,isclose；一般的取幂pow,阶乘factorial,组合数comb。这些工具有时候有用。

3.列表与字典

Python中最常用的两个内置类型，用好了相当强大

列表（有序）

list由于实现的原因，方便索引和增加append以及弹出尾端元素pop（都是O(1)的）。但是注意del remove pop(0) insert index等都是O(n)的！反复remove很有可能导致超时，这里的办法一般是开一个真值表先打标记（参见“懒删除”）

切片操作关于所切长度是线性复杂度，反复切片也很可能超时；切片list[k:l]当k>=l时不会报错而是返回空列表

判断in list也是线性复杂度！尽量避免in list的判断，必要时最好用dict或set代替。

查找值list.index()慎用，不仅是线性复杂度，而且在找不到的时候会抛出IndexError

Python特性：允许负数下标，正数越界才会报错。大部分时候会带来方便，但是有时候会让事情变麻烦（try……except可能无效）

注意函数有无返回值：list.sort(),list.reverse()都是原地修改而不返回，如要用返回值需用sorted()和reversed()（注意reversed()返回的不是列表而是reversed对象，如需用列表要用list转换类型，但是for循环则不需要转换，参见“可迭代对象”）

sum([])会返回0，但是min/max([])会报错！一定要小心

列表解析式：[f(x) for x in list (if P(x))] 创建列表时很不错的简便方式；注意外面要用中括号，用小括号的话得到的是一个生成器

列表可以相加，相当于连接；对应list.extend()方法（效果一样）（我更习惯用append，但是据说一次加上很多元素的话extend会更快）

遍历列表的时候通常用for循环；但是尽量避免一边循环一边删除/添加元素。如果必须的话建议改用while循环。

浅拷贝问题：说实话这块底层原理我也不太清晰。。。但是如何避免出错大概是知道的

基本原则：大于1维很可能出问题，尽量不要用*或者copy拷贝一个列表，不要在不同的地方引用同一个列表变量

例如[[0] * m]不会有问题，但是[[[0] * m] * n]会出问题（每行会一起变）

这种时候可以写成[[0] * m for _ in range(n)] (_ 表示无用变量)

拷贝一维列表的时候有时需要用a = b[:]复制一份，因为a=b的话a和b会指向同一个列表（参见“DFS回溯”）

但是二维列表的话这样写似乎仍然会出问题。一个选择是使用copy模块中的deepcopy深拷贝（但是好像比较耗资源，我一般是用别的办法绕开，没怎么用过deepcopy）

字典（无序键值对）

这玩意居然是内置的基本数据类型，实在是太强大了。

key和value都可以是任意类型的东西，所以自由度非常高（不过key一定要是可哈希的，粗略理解可以认为是不可变的，比如list不能作为key；但用list作为value是非常常见的）。而且字典用哈希实现，访问、删除、修改、查找（in dict)的平均复杂度都是O(1)，有非常好的性能

把有些信息用字典提前存下来，在需要的时候直接访问，有时是降低复杂度的好办法（参见“打表（桶）”）

dict有时候可以代替多维列表（用元组作为key），可以避免下标-1脑子抽风的情况。而且这个时候几个维度是完全对称的，不像list会有一个烦人的嵌套顺序。但是坏处在于字典占用的空间会比列表大不少，有MLE的风险

字典一般认为是无序的，最新特性好像能够维持顺序（但是我一般还是当无序的来用）。如果要按顺序遍历通常用for x in sorted(dict.keys()/values())

dict.keys()/values()/items()分别返回键/值/键值对列表

dict.get(key,default=None)查找指定key的value，如果key不存在不会报错而是返回给定的默认值（有时候可能有用但是这个方法我没用过）

dict.pop(key,default=None)弹出指定键值对（我也没用过）

注意dict的key要求互不相同；如果key已经存在，再赋值时会直接覆盖。对于某些需要保留重数的题要小心。

字典也有和高维列表相同的浅拷贝问题（但是反正我没copy过字典。。。）

一般来讲字典比列表省时间、方便（特别是处理离散化的值），但废空间，不要乱用。

4.函数

传参的时候可以对某些参数设置默认值，有时候能简化程序

注意你想要的参数的类型和你在函数里对该参数的用法要一致，Python不会帮你检查，但是不一致会报错

任何时候return都会终止函数运行而回到函数调用行。在复杂的递归写法中，对return的理解很重要

在函数体里用yield得到一个生成器（应该用不上，我也很少用）

注意全局变量与局部变量（特别是在DFS中，有时候需要在函数体内加上一行global语句），以及函数是否更改了变量的值（这个说实话我也不太清晰）

匿名函数lambda x,y:f(x,y) 有时候比单独定义一个函数方便得多（尤其在作为key参数的时候）

函数主要用于封装一段程序以便重复调用或者让程序逻辑更清晰

另一种封装的方式是类class，功能更齐全（但是大部分算法题可以不用）

5.其他数据结构

包括内置结构和一些模块里的结构；有些数据结构会在算法或优化部分着重介绍

元组tuple

不可变对象，一般用于记录信息，例如作为字典中的key，或者表示点的坐标。注意使用元组的时候可以写a,b = info[i]，省略小括号。

枚举enumerate

一般用法是for i,x in enumerate(list),遍历list中的（下标，值）对，有时可以简化程序（我没怎么用过）

集合set

可以看成只有key没有value的字典，无重复值，无序，在需要去重的时候可能有用。集合中各种操作也是平均O(1)的，包括in set判断。不过集合也比较费空间。

array模块

紧凑数组，只能存同类型的元素，但是省空间。我只用过一次，确实成功避免了MLE

queue模块

包含Queue类和PriorityQueue类，但是可以分别用deque和heapq代替，似乎会更快

heapq模块

提供堆工具，是分摊查找最小值和插入复杂度的最好工具之一。详见“优化”部分。

collections模块

有序字典Ordereddict：保持插入顺序的字典（我没用过）

计数器Counter：自动数数，有时非常方便。两个计数器之间可以进行运算，有时有一定用处

默认值字典defaultdict：避免Keyerror，简化程序（纯粹用来简化）

双端队列deque：头尾的删除和添加操作都是O(1)的，兼具queue（参见“BFS”）和stack（参见“递归”）的功能，在某些算法中是必需的。但是注意deque访问中间元素是O(n)复杂度，它只方便对头尾进行操作

可迭代对象（iterable），可哈希对象（hushiable），可索引对象（subscriptable）

对可迭代对象都可以用for循环遍历；很多方法的参数也不一定要是list而只需是可迭代对象，如reversed对象、range对象、生成器、map对象等

只有可哈希对象可以作为字典的key；大部分可哈希对象和不可变对象是一致的

只有可索引对象可以做取下标操作；这个有时候在报错中能够见到，譬如set是不能取下标的

6.杂项

函数工具functools

可以理解为函数的函数

最重要的lru-cache：自动打表机，帮忙缓存（参见“记忆化搜索”）

cmp_to_key：用于将比较函数转化为key函数提供给sort之类作参数

还有一些reduce之类的东西，我不熟，也不会用

迭代器工具itertools

可以对迭代器操作，某些时候还挺有用的

排列permutations,组合combinations,连接chain,笛卡尔积product,累积accumulate(自动前缀和),分组groupby等（我都不熟，有需要可以自己查）

二分查找工具bisect

详见“优化”部分

正则表达式工具re

据说处理字符串匹配的时候很好用，但是我不会用

其他的小东西比如list(zip(*matrix))作转置，isinstance判断类型之类的，这种实现小技巧我是真不太会，也不太关心（不过确实好用，方便，**希望大家不要学我，多掌握一些这种小技巧没坏处**）

二、算法

这里所说的算法，更多的是解决问题的思考方式而非一些具体算法（当然也有一些经典算法）。很多题目有多种不同的思考方式，可能导向相同或不同的方法；同一种解决方式从不同的角度看也可能可以被划分为不同的算法。这里只是对课程范围内的几种常见的思考方式做了一个初步的整理。

1.数学方法math

在我看来，所有算法问题本质都是数学问题；只是数学方法局限性比较大，需要约束条件很特殊或者研究对象有很好的性质。所以在最开始分析问题的时候，我通常是把题目当成数学问题来分析的；这么做偶尔能利用数学知识直接得到结果，大部分时候会得到一个入手的思路。

当然也有一些和数学问题有关的算法，例如处理质数问题的“筛法”

埃氏筛（基本上够用了）

```
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(1,N):
    if is_prime[i]:
        primes.append(i)
        for k in range(2*i,N,i): #用素数去筛掉它的倍数
            is_prime[k] = False
```

欧拉筛（线性筛）

```
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(2,N):
    if is_prime[i]:
        primes.append(i)
    for p in primes: #筛掉每个数的素数倍
        if p*i >= N:
            break
        is_prime[p*i] = False
        if i % p == 0: #这样能保证每个数都被它的最小素因数筛掉！
            break
```

由于欧拉筛保证每次用最小素因子筛掉数，故可用来计算某些数论函数的值：只要数论函数f满足**f(n)和f(n/p)有比较好的递推关系**即可

2.暴力bruteforce

这是典型的计算机思维，利用计算机的运算能力模拟现实情况。通常只在数据量比较小的时候行得通，但也是需要考虑的。首先需要明白题目在说什么，其次要能知道如何暴力解决、如何有序地实现出题目要求的状态、如何尽量减少所要考虑的情况，这样才有进一步考虑算法的可能。

例如 OJ02811 熄灯问题，OJ02692 假币问题

3.贪心greedy

greedy的核心在于充分减少需要考虑的情况数。这需要对象在数学上满足某些好的性质，使得按某种顺序考虑时，不需要考虑完所有情况就可以立刻知道大部分情况不可能是最优的，从而直接将他们舍弃；或者说，某种"直观看上去比较好的办法"一定是最好的，局部最优解一定能导致全局最优解。

贪心方法往往简洁、快速，时间复杂度低，但由于它需要忽略掉一些状态，需要所研究的问题有一些比较好的性质，因此解决问题的范围相对受限。

这里想说一下贪心的证明。在实际做题时通常是靠直觉猜出贪心策略，"这么选肯定是最好的"，然后直接写程序验证；但为了完全解决一个贪心问题，有时候证明才是更难的一步。贪心策略的证明有时直接用某种放缩能够得到，更常用的方法是利用归纳法或者取出"最小反例"，并利用贪心算法的局部最优性质进行局部调整。证明通过若干次局部调整能保持所求值的单调性且总能得到贪心策略中的情况也是一种办法。

贪心方法通常依赖于某种"排序"，策略通常是每次取出某种"极端"的元素。根据题目的不同，贪心策略千变万化，有时最明显的贪心策略不能得到最优解，有时需要做某种思维上的转化才能得到最优解。寻找贪心策略时，考查特例和分析最优解需要满足的性质是常用的思考方式，直觉也起到非常重要的作用。以下试举数例：

装箱问题，乌鸦坐飞机：从大往小塞，尽量塞满。比较简单的贪心

OJ25566 CPU调度：容易猜到贪心策略是把w[i]长的尽量往前放，这样"不怎么挤占后面的空间"，但证明并不容易。对于这个题来说，**直觉**很重要。

OJ04137 最小新整数：转换思路，考虑留下l=n-k位。注意到留下的数要尽可能小，首先是首位要尽可能小（**分析最优解性质**），于是留下的第一个数一定是前k+1位中最小的（因为前k+1位中至少要留下一位），设为第t位。如果t=k+1则已经做完，否则前t位已经不用考虑（删t-1位，留1位），接下来要删k-t+1位，那么留下的第一个数一定是剩下前k-t+2位中最小的。依此类推。（这里其实有一点递归的思想，决定第一个要删掉的数之后化归为一个更小的同构问题）

OJ12559 最大最小整数：这里的麻烦在于有一个整数是另一个的前段的情形，不容易判断。需要想到把每个整数自循环足够多次后再排序。另一种我觉得更自然的想法是考虑何时最优，最优的时候任两个相邻数互换一定会变差（**局部调整**），这样可以得到任两个数之间的序关系；继而证明这是一个全序关系，依此排序。

OJ02181 jumping cows：直观看应该奇数时间喝最高的，偶数时间喝最低的，但问题在于必须按顺序喝，也就是必须找出原序列的一个子列。这里的key point在于**把原序列分成若干个单调段**（这在处理序列问题时常常有用，例如OJ26976 摆动序列），可以利用局部调整证明奇数时间喝"高峰"，偶数时间喝"低谷"一定是最优的。OJ26971 分发糖果也是类似的思路，也是需要把序列的单调段拆出来，但是计算时更麻烦

具体实现的时候需要一些技巧，记录并判断目前的单调方向，例如像下面这么写：

```
sign = 1
cnt = 0
ans = 0
for i in range(n):
    if (info[i+1]-info[i])*sign<0:
        cnt += 1
        ans += info[i]
        sign = -sign
```

OJ16067 电影节，OJ01328 Radar Installation，OJ27104世界杯只因，OJ04144畜栏保留问题：

这几个题是一个系列的，都属于**区间问题**，基本涵盖了区间问题的所有常见类型，贪心策略和证明方法也都有类似之处（详见https://zhuanlan.zhihu.com/p/446371757）其中畜栏保留问题由于要求输出一种合题意的构造方案且数据比较大，需要应用单调队列优化。

这类区间问题普遍的办法是将区间排序（注意按左端点和右端点排序是有区别的！通常在**从左往右扫的时候按右端点排序**），然后按顺序考虑每一个区间，每次将**第一个满足要求**的区间加进来，同时**更新**剩下区间所需满足的要求（通常是区间右端点），完成一轮操作。大概像下面这么写：

```
now_right = -1
cnt = 0
for l,r in sorted(info):
    if l>now_right:
        cnt += 1
        now_right = r
```

OJ02287 Tian Ji--The Horse Racing：从问题的两端考虑，利用**局部调整**可以发现如果田忌最小马能胜齐王最小马，则应用最小打最小（以最小代价收下齐王的最小）；若必败，则应用最小打最大（以最小代价浪费齐王的最大）。麻烦在于最小马相等的情形。这个时候需要**对称地考虑最大马**，并在最大马也相等的情形结合起来做局部调整。这是这个题最难的地方。

分析最优解的思路在本题能做出一些结论，但似乎无法写出好的算法（平局很讨厌，复杂度降不下来）

OJ02431 expedition：这个题就需要一些**思维上的转换**了。直观上看应该每次加尽量多的油，但是在哪个范围里选最多呢？这里有一种看法：每次路过一个加油站就把油全拿上，但是不加；**每次没油的时候再挑一桶加**，这个时候挑的一定是目前手上最多的，于是问题迎刃而解。

这种**“延迟操作”**的思考方式在某些时候是很有用的。

OJ18164 剪绳子：这题首先要**转换思维角度**，把“剪绳子”看成“接绳子”（**逆向思维**），进而能够猜出每次接最小的两端绳子是最好的（但是证明并不容易）。具体实现要用heap结构。

这题实质上是经典的Huffman编码

4.递归recursion

递归的思想在数学和计算机领域都是极其重要的。和greedy不同，它并不以忽略某些状态为目标，而是致力于**将目前状态化为更小的同构情形**，以此逐步分解问题从而解决问题。在无法直接解决问题时，这往往是有力的方法，

由于递归过程中常常会重复处理某些子问题，递归方法是一种有效的方法，却往往不是一种高效的方法。因此，DP（动态规划）方法应运而生。实际上大部分DP方法解决的问题中，都或多或少有递归的思想。

实现递归的关键在于找到原问题和**更小的同构问题**之间的联系。常见的方式是去掉最后一步/某个最小（最大）的元素，考查问题的变化；如果此时新问题与原问题同构，往往就找到了递归关系。注意递归的时候不一定要把n变到n-1,只需要把n化为一个比n小的数即可；甚至可能没有明显的n，但是能将原问题转化为一个与之同构且某个特征量量严格更小的子问题，也能应用递归。另外，有时候在递归时需要对原问题稍加推广或者加上一些参数，这在DP方法中有典型的体现。

数据结构：栈stack

递归的底层实现实际上是用栈，也就是说原则上可以用栈来手动实现递归。栈在python中可以直接用列表list来模拟，“后进先出”。有时候递归时会出现递归次数超出限制的情况，常用的操作是sys.setrecursionlimit()，简单处理后通常就没有问题；实在不行就手动用栈来实现（我好像还没用过）

应用栈/递归的典型问题：OJ03704 扩号匹配问题，OJ02775 文件结构图，OJ02694 波兰表达式（应用递归是前序、中序、后序表达式转换和求值的标准方法）注意想清楚何时入栈/出栈，递归时何时调用子问题，何时返回。

5.动态规划dynamic programming（DP）

动态规划几乎是最重要的算法。有人说“万物皆可DP”，绝非妄言。对于无法用贪心解决的问题，DP往往是首先考虑的办法，也往往是最高效的办法；很多贪心问题也同样能用DP方法来解决（只是可能效率比贪心稍差）。

在我看来DP分为狭义和广义的两类。狭义的DP可以看做是递归的优化版本，利用按序的循环遍历状态空间，并在每次求出一个状态的值后记录以避免递归算法中子问题的重复计算。广义的DP思想则是一种比较通用的优化想法，即“尽量用上已经得到的信息”，从而显著降低时间复杂度。由于需要记录信息，必然要“以空间换时间”。

所谓“最优子结构性质”通常可以帮忙初步判断DP思路的可行性：如果一个问题的很多子问题都和原问题同构，而且原问题最优时这些子问题必然最优，则很可能能用DP方法解决。这其实也就是一种递归性质。

尝试用DP方法解决问题的思考方式通常与递归方法类似，寻找合适的、便于转移的状态以及方便的状态转移方式往往是DP解题的关键。

下面给出一些常见的DP问题类型和例子：

递推计数

这类问题严格来说不算DP，就是用递推方法解决问题。最简单的譬如斐波那契型数列（OJ02786 Pell数列，OJ23556 小青蛙跳荷叶），分拆数（OJ04119 复杂的整数划分问题）以及某些不定方程的解数之类的。

稍微难一点的例子譬如OJ09267 核电站（不太容易建立递推），OJ04150上机（首先要分析出题目需要满足的条件，转化为一个纯数列问题；其次需要按末位分类进行递推），OJ25573红蓝玫瑰（需要加上一个数列，记录翻转过的序列所需的操作次数）

这类问题关键往往在于建立递推关系。无法直接建立递推关系时可以考虑**引入若干辅助问题**一并考虑：如果A问题能转移到A的同构问题和B的同构问题之一，而B问题也是如此，则同样可以递推解决。

最长公共子序列

经典问题。考虑序列A的前i项和序列B的前j项的最长公共子序列，按最后一位是否相等转移。（这种**截取序列前若干项作为子状态**的手法在DP问题中几乎成为了一种标准方法）

```
for i in range(len(A)):
    for j in range(len(B)):
        if A[i] == B[j]:
            dp[i][j] = dp[i-1][j-1]+1
        else:
            dp[i][j] = max(dp[i-1][j],dp[i][j-1])
```

注：最长回文子序列问题可化归为最长公共子序列问题：可以证明，A的最长回文子序列长度等于A与reversed(A)的最长公共子序列长度（并不显然）。例如OJ01159 Palindrome 就可如此解决。

最长单调子序列

经典问题。以递增为例，考虑以第i项结尾的最长单调子序列，枚举该子序列中倒数第二项完成转移。（这里有一个**分类**的想法，按末位分类，对于末位确定的序列方便转移

```
dp = [1]*n
for i in range(1,n):
    for j in range(i):
        if A[j]<A[i]:
            dp[i] = max(dp[i],dp[j]+1)
ans = sum(dp)
```

还有一个nlogn的算法，非常巧妙（我自己肯定想不到）。维护数组d,d[i-1]表示**长为i的递增子序列中末位最小值**。可以非常巧妙地维护d数组，需要应用二分查找工具。

```
import bisect
d = [A[0]]
ans = 1
for i in range(1,n):
    if A[i]>d[-1]:
        d.append(A[i])
        ans += 1
    else:
        index = bisect.bisect_left(d,A[i])
        d[index] = A[i]
```

背包问题

这是最经典的DP问题，会衍生出各种不同的变式。详见https://oi-wiki.org/dp/knapsack/

0-1背包

考虑取前i个物品用t时间所能得到的最大值，枚举第i个物品是否取完成转移。注意这里**加上时间参数t**，因为转移过程中t的限定可能会变。“加参数”是DP问题中最重要的技巧之一。

```
dp = [0]*T
for i in range(n):
    for t in range(T,time[i]-1,-1):
        dp[t] = max(dp[t],dp[t-time[i]]+value[i])
ans = dp[T]
```

这里采用“**滚动数组**”的方法将二维数组压缩成一维，是DP问题中常用的技巧。这基于选前i个物品的状态仅依赖于选前i-1个物品的状态。注意**内层循环要倒着遍历**！

完全背包

将0-1背包中内层循环改为正着遍历即可（这里其实就利用了**先前已经得到的信息**来简化转移：在先前的转移中物品i可能已经用过若干次了）

多重背包

最简单的思路是将多个同样的物品看成多个不同的物品，从而化为0-1背包。稍作优化：可以改善拆分方式，譬如将m个1拆成x_1,x_2,……,x_t个1，只需要这些x_i中取若干个的和能组合出1至m即可。最高效的拆分方式是尽可能拆成2的幂，也就是所谓“二进制优化”

```
dp = [0]*T
for i in range(n):
    all_num = nums[i]
    k = 1
    while all_num>0:
        use_num = min(k,all_num) #处理最后剩不足2的幂的情形
        for t in range(T,use_num*time[i]-1,-1):
            dp[t] = max(dp[t-use_num*time[i]]+use_num*value[i],dp[t])
        k *= 2
        all_num -= use_nume
```

变式

如果需要求出所有可能达到的值，需要用集合更新，状态设计方式不变（OJ01742 coins）

有时候可能需要考虑在给定所选物品的数量的情况下最优化，这时dp数组要多带一个数量参数（忘记哪个题了）

有时候背包问题的限制可能更多，需要加更多的参数（OJ04102 宠物小精灵之收服；这个题还有一个要点是**更换参数的选取**。设计DP状态时不一定把要求的作为最终结果，有时把**要求的东西作为参数**会更方便）

注：背包问题的DP解法需要时间T不太大，因为要遍历每个可能的T。如果T很大而物品数量很少，采用DFS枚举物品的选法有时是更好的选择。

采用合适的顺序遍历

某些问题看起来不方便用DP解决，因为状态似乎是无序的；但是只要能够找到状态的某个特征量有序，就可以以此为顺序完成DP（当然这类问题最省脑子的办法参见“记忆化搜索”）

OJ01088滑雪：要求最长的递降通路长度，状态转移是容易的，但是顺序并不易确定。注意到每步**总是往更低的地方滑**，按**高度**从小到大遍历即可。

OJ01191棋盘分割：每次分割后棋盘会变小，按**棋盘的大小**从小到大遍历即可（当然这样的话状态比较复杂；这题其实更适合记忆化搜索）

OJ01661帮助Jimmy：每次会往下跳，按**高度**从下往上遍历即可；为了减少状态数，可以只考虑**横坐标为某个平台端点**的点（DP相比记忆化搜索最大的缺陷就在于要设计出能够遍历的状态，而且要尽量减少状态数量）

目前就只能想到这么多了。。。。。。DP无处不在，肯定还有很多技巧方法没有写到，啥时候想起来再补吧。

6.广度优先搜索Breadth First Search（BFS）

作为图搜索的基本算法之一（另一个是DFS），BFS在某些场合有重要的应用。需要注意的是，BFS本质上是一种搜索算法，但在解题时通常用来处理最短路问题（连通分支、走迷宫等问题理论上也能用BFS，但一般用DFS解决；对于最短路问题，由于用DFS可能需要遍历所有可能路径，BFS的时间复杂度常常会小得多）

基本想法：每次搜当前节点的邻点，实质上就是"按距离搜索"。经典写法是维护一个队列，每次取出队头的点，并将它的所有邻点放入队列。

```
from collections import defaultdict,deque
vis = defaultdict(bool)
vis[start] = True
points = deque()
points.append((0,start))
while points:
    step,now = points.popleft()
    if now == end:
        ans = step
        break
    for follow in neighbour[now]:
        if not vis[follow]:
            points.append((step+1,follow))
            vis[follow] = True
```

这里的vis也可以改用集合，但是真值表可能更好一些。注意这里将距离和点绑定成为元组，顺便记录了最短距离。另外这里是每次在入队时更新vis，这样可以减少一些不必要的入队，但在某些变形的问题中会导致WA（后面会详细解释）

我更喜欢的写法是开一个dist数组记录距离，并直接用dist数组判断点是否已经入队：

```
from collections import deque
dist = {}
for x in graph:
    dist[x] = float('inf')
dist[start] = 0
points = deque()
points.append(start)
while points:
    now = points.popleft()
    d = dist[now]
    if now == end:
        ans = d
        break
    for follow in neighbour[now]:
        if dist[follow] > d+1:
            dist[follow] = d+1
            points.append(follow)
```

另外一种写法是直接用list或set记录每一层，一层一层地更新（但是用的人好像比较少）

```
from collections import defaultdict
vis = defaultdict(bool)
vis[start] = True
sheaf = {start}
step = 0
```

```
while True:
    if not sheaf:
        break
    if end in sheaf:
        ans = step
        break
    step += 1
    new_sheaf = set()
    for point in sheaf:
        for follow in neighbour[point]:
            if not vis[follow]:
                new_sheaf.add(follow)
    sheaf = new_sheaf
```

以上是BFS的基本写法，下面给出BFS在解决问题中的几种变通应用方式

合理选择建图方式

由于BFS本质是在图上做搜索，选择合适的建图方式有时候能直接把问题转化为经典形式。一定要记住，搜索的时候“邻点”不一定只能是题中的“邻点”！

另外BFS对有向图也适用，这意味着有时候边不一定要是“对称”的。

例如OJ02802 小游戏：题目要求线段数最小，只需对所有同行或同列且连线不穿过卡片的点连边，即化为经典情形。这里有一个细节要处理：遇到卡片之后不能再拐弯，所以除了开始的卡片，其余卡片应当是不引出边的

又如有的题两点之间有传送门可以瞬移，意思就是这两个点应当被看成同一个点

拆点技巧

这是BFS应用中极其重要的方法。对于有约束条件的一些问题，可以通过将一个点拆为多个点，“分层处理”，从而完美刻画约束条件。以下试举数例：

OJ04116 拯救行动：麻烦在于打骑士要额外多花1时间。可以将每个骑士拆成两个点x和@,其中任意邻点向x引入边，x仅向@引一条边，@向邻点引出边，便可用经典BFS方法解决

OJ04129 变换的迷宫：可以将每个点拆为K个点，也就是K层，每次从第i层的点走到第i+1层的邻点（模K意义下）。没有石头的点对应K层都没有石头；有石头的点对应第0层没有石头，而另外K-1层有石头。于是可用经典BFS解决，注意走到任一层的终点都可看作完成任务。

OJ04115 鸣人和佐助：将每个点拆为T+1个点，第i层表示还剩余i个查克拉。手下的点向下一层引出边（手下的第0层的点不引出边，因为此时没法打手下了），其余点向同一层引出边，即化为经典问题。

OJ04130 Saving Tang Monk：这题几乎是顶配的BFS题。为了处理钥匙，可以将每个点拆成M个点，只有在第M-1层而且捡到了钥匙M才能上一层，只有在第M层才能算完成任务。为了处理蛇，可以仿照04116拯救行动的方法，将每层每个有蛇的点再拆成两个点。这样在理论上即化归为经典问题。（但是太恶心了，我写半天AC不了）

处理带权图

严格来说带权图的最短路算法并不是BFS，但这些方法也可以看成BFS的扩展。其本质其实还是上面的拆点方法，只是做了一些简化和优化。

这里的关键在于：并不按层数扩展，而是按距离扩展。每次取出**当前队列中最近的点**（要用到heap），可以证明此时它的距离不可能再变短（这里有一点**贪心思想**），于是可以放心标记。（事实上这就是Dijkstra算法）

```
import heapq
```

```
dist = {}
for x in graph:
    dist[x] = float('inf')
dist[start] = 0
points = []
heapq.heappush(points,(0,start))
while points:
    step,now = heapq.heappop(points)
    if now == end:
        ans = step
        break
    for follow in neighbour[now]:
        if dist[follow] > d+weight[(now,follow)]:
            dist[follow] = d+weight[(now,follow)]
            heapq.heappush(points,(dist[follow],follow))
```

这里注意：在边带权的情形，如果要用vis数组标记，必须**出队标记**而不是入队标记，因为只有出队时最短距离才稳定。但这可能带来额外的时间复杂度。对于**点带权**的情形，入队标记没有问题。

另外还有一种处理方法：仍用deque，但**允许反复入队**。每次搜索邻点时更新距离，如果还不在队里则入队并标记；每次出队时**将标记改为False**以便再次入队。这种方法在最坏情况下复杂度可能较高，但平均复杂度相当好，且适用于更广泛的情形（spfa算法）

7.深度优先搜索Depth First Search（DFS）

DFS在某种意义上很“暴力”，常常带有“枚举遍历”的意味，时间复杂度也通常比较高。但由于实现简便，能够处理绝大多数实际情况（很多实际问题除了DFS遍历没有更好的办法），应用仍然相当广泛。这里注意：狭义的图搜索算法DFS和一般所说的DFS实际上有一些区别。图搜索DFS是一种搜索算法，每个点只会搜到一次；但实际问题中更常需要的是利用DFS的搜索思想来遍历所有可能路径。

由于遍历方法通常很耗时间，为了降低时间复杂度，往往还要进行必要的剪枝（参见“优化”）

迷宫问题

这是最经典的DFS问题，问能否走到出口、输出可行路径、输出连通分支数、输出连通块大小等。这种问题应用经典的图搜索DFS即可解决，不需要回溯，**每个点只需要搜到一次**，所以**不需要撤销标记。**

```
directions = [(-1,0),(1,0),(0,-1),(0,1)]
cnt = 0
ans = []
vis = [[False]*n for _ in range(m)]
def dfs(r,c):
    global area
    if vis[r][c]:
        return
    vis[r][c] = True
    area += 1
    for dr,dc in directions:
        if 0<=r+dr<m and 0<=c+dc<n and info[r+dr][c+dc]!='#':
            dfs(r+dr,c+dc)
for i in range(m):
    for j in range(n):
        if not vis[i][j]:
            cnt += 1
            area = 0
```

```
            dfs(i,j)
            ans.append(area)
```

注意这里要对下标越界与否做判断；另一种写法是在外面加一圈"#"作为保护圈，从而可以省去判断。

稍微麻烦一点的题目：OJ23558 有界的深度优先搜索，需要带step参数，在step=L时return

DFS回溯（枚举）

这是DFS题目中最常出现的类型。由于要遍历所有可能，在每次做出选择之后需要**撤销选择标记**，以便不影响其它的路径（也就是说一个点**会被搜到很多次**）

这其实是一种**枚举想法**。理论上这些题目可以用一系列的for循环加条件判断解决，譬如八皇后问题，可以写7个for循环。但一方面这样写很麻烦，另一方面如果需要的循环数量n是变量，就没法这么写了。这时就要借助于DFS的思想。实质上**DFS遍历的顺序和多重for循环的顺序是一致的**，每次调用dfs函数的过程可以看成是**下一层的for循环**。

这种问题的实现细节其实有点麻烦。不同的写法可能都能成功，但一点微小的改动也可能对程序造成极大的影响，甚至return语句的不同顺序都可能导致不同的结果。关键在于何时做标记、撤销标记，何时返回，以及如何处理统计量。

下面以输出所有可能的最短路径为例展示回溯问题的写法。

```
vis = [[False]*n for _ in range(m)]
directions = [(-1,0),(1,0),(0,1),(0,-1)]
ans = []
dist = float('inf')
def dfs(r,c,pre=[],step=0):
    if step>dist:
        return
    if vis[r][c] or info[r][c]=='#':
        return
    vis[r][c] = True
    pre.append((r,c))
    if r==end_r and c==end_c:
        if step==dist:
            ans.append(pre[:]) #要加入的是pre的copy，因为pre会变
            return
        elif step < dist:
            ans = [pre[:]]
            dist = step
            return
    for dr,dc in directions:
        if 0<=r+dr<m and 0<=c+dc<n:
            dfs(r+dr,c+dc,pre,step+1) #也有在这里做选择和撤销选择的写法，也可以
    vis[r][c] = False
    pre.pop() #回溯，"撤销选择"
```

用栈实现DFS

由于递归的底层实现是栈，DFS算法基于递归，故用栈也能实现DFS（当然这样写的人比较少）

用栈实现DFS时**和BFS的写法很像**（每次取出一个点，将它的邻点中还未搜索过的点入栈），只是将BFS中的队列换成栈而已。从这个角度看，BFS和DFS之间似乎有一种"对偶"的关系。

如何用栈实现DFS回溯我还没想清楚，是否**允许多次入栈**即可？（入栈时打标记，出栈时撤销？这样的话就和spfa的写法很像了，但是没试过，不知道对不对）

三、优化

这里所说的“优化”，有时候也常常被归为算法。但是这里提到的方法通常更加通用，在各种题目各种算法的局部都可能应用这里的方法来达到时间复杂度上的优化（这里的优化主要针对时间；针对空间的优化方法本来就比较少，我也很少碰到需要优化空间的情形）

打表（桶）

有人非常推崇“桶”，但我对“万物皆桶”的说法其实不太认同。尽管如此，打表记录确实是最基本也最常用的优化方法，体现了优化的根本思想“以空间换时间”。用合适的数据结构记录合适的信息，以便充分利用已得到的结果，是打表的精髓。可以说，DP方法就是将递归方法利用“打表”进行优化。

数据预处理

预处理是“打表”的一种最简单也最常用的方式，预先记录某些信息以便处理问题时直接调用，避免重复计算。常见方法如**记录“前缀和”**以快速得到任意连续段的和（这种方法还可推广到高维的前缀和），利用**差分数组**以方便对连续段的同时更新（一段同时加x反映在差分数组上就是头加x,尾减x;再用前缀和还原即可），打**下标和值的对应表**以方便查找值在列表中的位置，打**真值表**以方便查找值是否出现过（用集合也可以，但效率稍差），打**计数桶**以方便查找值出现的次数等。

维护适当的信息

这里所说的信息和预处理中的不同，它们在任务执行过程中**可能改变**。如果维护太多的信息，在更新信息时可能很麻烦；如果维护太少的信息，求解问题时又可能需要复杂的运算。寻找合适的信息来维护，既便于更新，又便于求解（或者说**分摊更新与求解的复杂度**），常常是这类问题的关键。

通常会考虑维护一些**特征量**，例如最大值、最小值、端点/边界点、某个特征值出现次数、某个极端位置的量等；这些特征量足够提供求解问题的信息，而又不像全状态那样难以维护。

（这样的题目很多，在很多算法中其实也都有这样的思想；例子暂时想不起来了，大家可以自行补充）

记忆化搜索

可以看做介于递归和DP之间的形式。由于某些问题的状态**很难按顺序遍历**，记忆化搜索是解决这类问题的最好方法。（这里要插一句，我特别讨厌这种方法，觉得它不太优美。。。但是有的时候确实好用）

基本想法是用一个表记录下已求解过的状态，递归**调用时先查表**，如果表里已经有该状态则不用重复求解

```
memo = {0:0}
def solve(n):
    if n in memo:
        return memo[n]
    memo[n] = #这里是普通的递归，将n化为更小的问题做递归调用，得到n的结果并记录
    return memo[n]
```

@lru-cache(maxsize=None)

这是python内置库functools中的工具，可以自动实现打表缓存的功能。注意函数的参数必须是**可哈希对象**，如果是列表之类的可变对象需要**转化成元组**再传入函数。

这个工具用得好可以简便地降低递归程序的时间复杂度，但会显著浪费空间。有时候要手动设定maxsize来达到时间和空间的平衡。

剪枝

这也是我很讨厌的一个东西，主要是它肯定能够降低时间复杂度，但能降低多少。。。。。。不清楚，和数据有关

剪枝的核心思想和贪心类似，扔掉一些不可能为最优解的情况，从而削减需要考虑的状态数。但它通常不像贪心算法能够得到一个很好的策略；它更像“走一步看一步”，如果意识到当前一定不是最优的路就“及时回头”。为了更早发现当前路径的“非最优性”，通常需要维护一些已有解法的信息；信息越全面，能削减掉越多的情况，但这样需要记录的东西就越多，剪枝时的判断也就越复杂。需要在两者之间做一个权衡；通常先做初步的剪枝，如果仍然超时再考虑更细致的剪枝。

在DFS算法中剪枝通常是必不可少的；在其它算法中如果能合理剪枝，也能提升算法效率。

位运算

由于计算机的底层实现，位运算的速度比一般运算要快上不少。合理应用位运算，有时能得到数倍的效率提升。

位运算通常用来**操作状态。**例如0表示不选，1表示选，于是可以用一个二进制数来代替集合或真值表，用**按位与、按位或**代替集合的交与并。同时当状态较为复杂时，用数来表示状态往往能使DP方法写得更为简洁，转移方式更为清晰（所谓的“状态压缩DP”，例如OJ04124 海贼王之伟大航路）

另外一个用位运算得到显著优化的例子是OJ01742 coins，但位运算的应用例子相对较少，它只能带来常数上的优化，而难以做到量级上的提升。

二分查找

二分在更多的地方被归为一种算法，但在我看来它更像是一种无处不在的优化技巧。只要在**有序序列**中查找所要的元素，就能用二分的方法把线性复杂度降低到对数复杂度。

bisect库

python内置二分查找工具，能够实现基本的二分查找功能而避免手写（手写二分查找时边界条件容易出错）

注意bisect库只适用于**升序序列**，对于降序序列需要将列中每个数取相反数再使用；如果要按指定的key排序，可以将每个元素变为元组，把key放到元组的第一个形成元组序列（这种方法在接下来的单调队列优化中也有应用）

bisect_left和bisect_right分别表示插入可能位置中最靠左/右的位置；注意返回的是下标。

insort函数实现插入功能，原地修改列表而不返回值；但是**它是O(n)的**！往往并不能带来量级上的优化

手写二分查找

有不同的维护边界条件的方式，重点是保持开、闭区间性质的统一，以及检查是否会陷入死循环：可以考虑刚刚没能退出循环的极端情况，想想l和r会如何变化。这样也有助于判断最后的答案到底是哪个变量。

```
l = 0;r = N
while l<r:
    mid = (l+r)//2
    if is_valid(mid):
        l = mid+1 #这样保证闭区间[l,r]内每个数都是未知是否可行的
    else:
        r = mid
ans = l (=r) #由于l=r，都是答案
```

二分查找的应用

二分查找有两种典型的应用：

一是方程求解问题，这种问题相对比较无聊。注意这个时候while循环的退出条件应当是l和r的差小于所需精度。

二是一类最优化问题，特别是“最值的最值”问题。这类问题所求的最优值通常具有”单调性质“（例如OJ01064 网线主管，OJ08210 河中跳房子），即小于某个数的都可以但大于它的都不行。对于这种问题，可以考虑**直接去枚举**最优值的可能取值。利用单调性质采用二分算法，这一步只有logn的复杂度；接下来的问题转化为**判断该最优值**是否满足要求。这样就把最优化问题转化为了判定问题，而判定问题有时有比较好的办法解决。所谓“0-1分数规划问题”（OJ01045 放弃考试）也能用类似方法解决。

单调队列（二叉堆）

python有内置库heapq可以实现单调队列结构，在很多问题中非常有用。

heapq最重要的功能是“分摊”**插入**与**查找最小值**的复杂度（同样分摊复杂度的数据结构还有树状数组和线段树，思想有点类似，但是超纲了而且得手写，很麻烦）。heappush和heappop的复杂度都是O(logn)，对于需要**不断更新**且**不断查找**的问题，heap常常是值得考虑的数据结构（OJ27256 当前队列中位数，OJ27384 候选人追踪）

对于需要按指定key排序的问题，处理方法与“二分查找”一样。

懒删除

注意使用heap时**不能使用通常列表的方法**，这样会破坏堆结构；只能使用heap库中的操作。无法简便删除heap中任何一个数，是这个数据结构最大的问题（如果要删除，就得重新建堆；只有弹出最小数是方便的）

这里通常的解决方法是所谓“懒删除”，也即“逻辑删除”：需要删除一个元素时先对其打上标记，等到操作到该元素时再将其弹出去（由于一般list中删除的复杂度是线性的，故在一般列表中也常采用这种方法，操作时跳过打了标记的元素；但这种方法在heap中的运用更为典型）。打上标记后元素仍在序列中，但我们将其**视作已经不存在**进行操作；等到需要真正操作该元素（该元素已到堆顶）时，再做实质上的删除，这样实质上没有改变总的复杂度，但避免了从heap中直接删除元素。

```
#每次要删除x时out[x]+=1
from heapq import heappop,heappush
while ls:
    x = heappop(ls)
    if not out[x]:
        new_min = x
        heappush(ls,x) #不需要弹出的，记得压回去
        break
    out[x]-=1
```

四、思考与调试

这里整理一些思考问题的入手方式以及调试时要注意的细节（肯定不全，欢迎补充）

思考

首先判断题目的大致类型。有些题目只需要按要求实现即可，不需要考虑算法，这个时候就只用注意实现细节。一定要把题目要求读清楚，不要想当然！注意题目的坑点。

对于算法题目，题中给的数据范围有时有比较大的提示价值。如果数据比较小，可能考虑枚举或者DFS遍历；如果数据很大，大概率只能用O(n)的方法，那么肯定首先考虑贪心。另外相对较小的数据可能是入手点，考虑遍历方式的时候要避开范围很大的数据。

如果枚举明显会超时而贪心策略难以想到，可以考虑递归的思路以期得到DP方法。这个时候就要考虑如何把问题化归为更小的同构子问题（参见“递归”）

BFS和DFS题目特征相对明显，注意约束条件的变化和相适应的调整。

写代码之前尽量估测算法的大致复杂度。但是如果想到的方法有些勉强（不一定超时）又暂时想不到更好的方法，也应该先写出来试一试。

没有思路的时候，换个角度看问题通常是不错的办法。逆向思考，延迟操作，转换视角，有时能帮助找到问题的突破口，比如猜到贪心策略

如果实在没有思路就先跳过。有时间的时候可以从各种算法的角度思考，也许会有新的收获

调试

最常见的：TLE,RE,WA

TLE一般意味着算法有问题，当然也有可能是具体实现的问题。首先检查程序中有没有特别耗时的操作，例如列表很多操作都是线性，反复做很容易超时；如果没有，基本意味着算法不够好。这个时候就要重新思考算法了、

RE相对好调，在程序中逐行检查哪里可能出错。注意一些细节问题，比如列表为空之类的边界情况（在“语言”一章基本上都有写）。如果有递归，考虑爆栈的可能。

WA是最麻烦的。首先确认实现的正确性，确认没有弱智的多print或者缩进错误之类的，同时确认输出格式是否有问题。其次确认算法逻辑的正确性，同样是关注边界极端情况和初始化的值是否正确。如果是猜测的贪心方法，可能意味着贪心策略不对需要另想算法。如果样例就出错则相对容易，可以在程序中print一些变量的值来分析算法哪里有问题。如果仍然不知道哪里错了，就多读读题，想想有没有一些自己默认了但是其实不一定的条件。还不会的话就跳过吧。。。。。。