python习题经验总结

oj 02792 集合加法

```
for x in compa:
        cnt+=b.count(x)
```

实际上对于更低复杂度的统计，可以通过下列方式：

```
from collections import Counter
    b_count = Counter(b)
     for x in compa:
        cnt += b_count.get(x, 0)
```

这里相当于利用字典进行统计，counter的复杂度在On量级；get作为字典的一个函数，dict.get(key,default=None)

这样代码从原先的5000ms缩短到了50ms（好家伙

oj 02981 大整数加法

```
a="0"+a
b="0"+b
if l1>l2:
    b="0"*(l1-l2)+b
else:
    a="0"*(l2-l1)+a
```

本题实质上是字符串处理问题，核心在于对列表进行处理使得在for循环过程中不会报错。

同时，应当注意到0+0=0这个坑点。（这是不可避免地，它要求输出结果的第一位不能是0，显然代码实现过程中自然地会让0+0这个情况报错，需要发现这一点.

```
for i in range(len(s1)):
    sum[i+1] += (sum[i]+s1[i]+s2[i])//10
    sum[i] = (s1[i]+s2[i]+sum[i])%10
```

题解对加法的处理显然要更简便，运用了reverse处理，相比于：

```
for i in range(-1,-max(l1,l2)-2,-1):
    s=c[i]+int(a[i])+int(b[i])
    if s>=10:
        c[i]=str(s-10)
        c[i-1]=int(c[i-1]+1)
    else:
        c[i]=str(s)
```

cf 545D queue

```
for i in range(n):
    if tsum<=queue[i]:
        sat+=1
        tsum+=queue[i]
    else:
        continue
```

这个题目非常之有趣，上午花了四十分钟把原代码改的越来越复杂臃肿，但总是有很多情况考虑不到。但中午吃完饭后灵机一动想到了上面的方法，在短短五分钟内就迅速完成了此题。

oj 04138 质数的和与积

本题的思路非常之简单，时间卡的也不严，可以说是有手就行，但可以借此机会回忆一下求筛选质数的一种快速筛选方法：

埃氏筛，复杂度O(nloglogn)：

```
def underp(t):
    ints=[True]*(t+1)
    ints[0]=ints[1]=False
    for x in range(2,int(t**0.5)+1):
        if ints[x]:
            for m in range(x**2,int(t)+1,x):
                ints[m]=False
    return [num for num,prime in enumerate(ints) if prime]
```

enumerate()函数相当于遍历一个集合的过程中，同时给出索引值和内容。

欧拉筛，复杂度O(n)：

```
def euler_sieve(limit):
    is_prime = [True] * (limit + 1)
    primes = []
    min_prime_factor = [0] * (limit + 1)
    for num in range(2, limit + 1):
        if is_prime[num]:
            primes.append(num)
            min_prime_factor[num] = num
        for prime in primes:
            if prime * num > limit:
                break
            is_prime[prime * num] = False
            min_prime_factor[prime * num] = prime
            if num % prime == 0:
                break
    return primes
```

oj 03253 约瑟夫问题2

```
while any(people):
        for index,peo in enumerate(people,1):
            if peo:
                if num==1:
                    people[index-1]=False
                    num=m
                else:
                    num-=1
```

个人认为，这是模拟算法中最简单易懂的方式了。其中利用了any()，这就是说，如果其中存在任何一个True都会输出True。

这里放一下之前版本的代码：

```
while z!=n:
        num+=1
        if num>n-1:
            num-=n
        if num+1 not in posints:
            continue
        s-=1
        if s==1:
            if z==n-1:
                print(num+1)
            else:
                print(num+1,end=",")
            posints.remove(num+1)
            z+=1
            s=p+1
```

可以看到，这里我们利用的是remove的方式，（之前甚至判断条件里用的是index），无论是pop(i)还是remove，对列表而言时间复杂度都是O(n)级别的，但如果我们修改列表元素的值，那么这只是一个O(1)复杂度的操作。

sy176 序列合并

双指针问题：本质上是构造两个指针同时对一/两个数组进行操作。

```
while p1<n or p2<m:
    if p1==n:
        lres.append(lb[p2])
        p2+=1
    elif p2==m:
        lres.append(la[p1])
        p1+=1
    else:
        if la[p1]<=lb[p2]:
            lres.append(la[p1])
            p1+=1
        else:
            lres.append(lb[p2])
            p2+=1
print(" ".join(map(str,lres)))
```

这里是让两个指针在不同数组中交替移动。

值得一提的是，最后一行实际上提供了一种对于一个整数列表输出一行元素的方式。

oj 18211 军备竞赛

```
design.sort()
while p1<=p2:
    while p1<=p2:
        if p-design[p1]>=0:
            p-=design[p1]
            p1+=1
            weap+=1
        else:
            break
    if weap>=eneweap+1 and p2-p1>1:
        p+=design[p2]
        p2-=1
        eneweap+=1
    else:
        break
```

对于贪心算法，有一点在于，可以通过排序的方式使得思路更清晰一些，有的时候可以使得策略选择大大优化。

本题依然是对撞指针问题，但关键点在于，最后的临界特殊情况（只剩最后一份图纸的时候不用卖）这一点需要单独讨论。同时另一个特殊情况是当图纸只有一份，这时需要单独考虑。

cf 706B intersting drink

```
for _ in range(m):
    coins=int(input())
    index=bisect.bisect_right(shops,coins)
    choice.append(index)
```

第一种方法利用python中二分查找的库,bisect_right(list,element)，返回值是比element大的最小的数，而bisect_left(list,element)的返回值是比element小的最大的且最靠左的数。当然，如果这两个函数无法满足条件，可以利用双指针自行设计二分查找的程序。一般而言，二分查找的时间复杂度是O(logn)。

```
for price in prices:
    dp[price] += 1
for i in range(1, len(dp)):
    dp[i] += dp[i-1]
```

第二种方法是利用dp思想，构造一个记忆数组，从而把循环过程改为查找过程，这样的时间复杂度是O(n)。

oj 19757 saruman's army\oj 04100 进程检测

```
import bisect
    troops.sort()
    while p<n-1:
        p1=bisect.bisect_left(troops,troops[p]+r)
        if p1<n-1:
            p=bisect.bisect_right(troops,troops[p1]+r)
```

本题其实与进程检测完全类似，只不过这里相当于进程检测中开始和结束的时间长度固定，但要求检测位置必须按特定位置给出。

原则上，本题完全可以利用双指针，复杂度为O(n^2)，这里利用二分查找，不光省去了写双指针的过程，同时将复杂度降到了

O(nlogn)，但代价是bisect函数并不是很灵活，所以需要疯狂debug（bushi。

具体贪心的思路是，从起点出发向前走r，找到最大的在r范围内的位置放置第一个水晶球，然后再向右走r，找到最小的不在r范围内的位置作为新的起点。

```
start.sort()
while i<n:
    pro+=1
    endtime=start[i][1]
    while i<n and start[i][0]<=endtime:
        endtime=min(start[i][1],endtime)
```

进程检测问题其实提示我们，可以先将需要处理的数据进行排序，可能直接就有贪心的思路了。此题的思路其实是，找到一个起始进程，所有进程起始时间在其结束时间之内的进程都可以同时完成，然后逐层向前推进。

此题还让本人掌握了一种同时赋值的小技巧：

```
a,b=1,2
if a==b==-1:
    break
```

赋值时最好不要写成连等符号。

oj 02783 holiday hotel

```
for x in data:
    if x[0]<=q[0] and x[1]<=q[1]:
        s=0
        break
    if x[0]>=q[0] and x[1]>=q[1]:
        data.remove(x)
if s==1:
    data.append(q)
```

这段代码直接TLE了，这也是第一次正式遇到复杂的算法问题。时间复杂度为O(n^2)。

```
s1=sorted(data,key=lambda x:x[0])
for i in range(0,n):
    if s1[i][1]<distmax:
        if s1[i][0]!=valmin:
            cnt+=1
        distmax=s1[i][1]
        valmin=s1[i][0]
```

此题的核心就在于需要进行一次排序，同时也是一个贪心算法，这种两个量需要比大小的问题一般都可以考虑进行一次排序。

此外，关于lambda函数，其实有一种双重排序的语句：

```
s1=sorted(data,key=lambda x:(x[0],x[1])
```

btw，多插一嘴，如果想要一正一反的二级排序，在第二项后面添加负号即可。

oj 04133 垃圾炸弹/oj 02659 bomb game

本题想法非常自然，将总和提前存入数组中计算整个二维列表的最大值即可，是一道非常基础的矩阵练习题。

```
for j1 in range(max(0,xi-d),min(1024,xi+d)+1):#col
    for j2 in range(max(0,yi-d),min(1024,yi+d)+1):
        trash[j2][j1]+=tr
```

而bomb game稍微复杂一些，关键是要理清命中与未命中的逻辑：如果是0意味着内部的格子一定不可能有，如果是1意味着外部的格子一定不可能有。同时要注意边界条件的问题。（如上）

```
if t==0:
    if indx-peff-1<=rx<=indx+peff-1 and indy-peff-1<=ry<=indy+peff-1:
        board[rx][ry]=0
else:
    if not (indx-peff-1<=rx<=indx+peff-1 and indy-peff-1<=ry<=indy+peff-1):
        board[rx][ry]=0
```

这是gpt给出的代码，从中我们还有两处简化的语法可以学习：if条件可以采用连续不等式判断，以及可以用not对整个命题进行否定。

不过两题涉及到一个有趣的问题，如何简便地给出一个矩阵的最大值，如何统计矩阵中一个元素出现次数，如何计算矩阵中元素的总和。

```
mx=max(max(row) for row in trash)
mx=max(map(max,trash))
cnt=sum(row.count(mx) for row in trash)
cnt=sum(sum(row) for row in board)
cnt=sum(map(sum,trash))
```

这相当于进行了一次flatten运算，虽然，这个操作并没有降低时间复杂度，但使得代码更为简洁明了。

1443C delivery dillema

题解上看了一下想到的确实是最优解法，复杂度即为排序复杂度O(nlogn)。思路其实是先把送餐的时间进行排序，如果前面所有自取的总时间和比这个小，且加上这个也比他小，那这个也自取；注意当取完前面一个后，总时间有可能比下一个送餐的时间要久，所以需要提前判断。注意一下边界值n=1时候的退出条件。

```
var=sorted(zip(diliver,own),reverse=True)
mx,su=var[0][0],0
for i in range(t):
    su+=var[i][1]
    if su>=var[i][0]:
        mx=var[i][0]
        break
    elif i<t-1:
        mx=var[i+1][0]
```

从语法角度，遇到多个条件的时候尽量把思路捋清楚再写，不然很容易乱掉。另外，elif后面可以不用加else。

这里运用了zip()函数，一般而言其实没有什么用，只是能将两个列表合并为元组，而zip(*)的作用是将这样一个元组列表解压缩成两个独立的列表。一种简单的运用是：

```
a=[1,2,3]
b=["a","b","c"]
for num,let in zip(a,b):
    print(num,let,dict(a,b))
```

cf 1364A XXXXX

此题曾一度浪费了本人大量的时间。算法大致分为两步，第一个核心步骤在于计算前缀和，再进行对r取模，这样就可以得到一个从值为0到r-1的个序列。

```
def mod(lst,num):
    return [i%num for i in lst]
for i in range(n):
    su[i+1]=su[i]+elem[i]
lst0=mod(su,x)
```

后续的部分其实可以简化为，找到最小的两个不相同的数，最大的两个不相同的数，分三种情况即可得到最大值。现在的问题是，如何找到一个序列中最小（大）的两个不相同的数，能否实现不用进行对n=1情况的分类讨论？

```
p1=next((i for i in range(1,n+1) if lst0[i]!=0),n+1)
p2=next((i for i in range(n,-1,-1) if lst0[i]!=last_value),-1)
```

python中的next函数给出了一个完美的答案，为了更好地理解next函数，先来了解一下iter函数以及迭代器。

所谓迭代器可以理解为遍历的指标，与可迭代对象不同的是，他迭代一次过后将变为空。iter函数则是将可迭代对象转化为一个迭代器。而next(iter(a),default)的含义是，将使迭代器迭代入下一个指标，当遍历结束后，将会变为default的值。

我们惊人的发现，这个函数满足所有前面我们需要的要求。

第六次课 recursion：

对于优化时间复杂度，可以采用sys库中的sys.stdin.read以及sys.stdout.write来整体处理。最好遵循下面的格式：

```
import sys
from math import gcd
input,output=sys.stdin.read,sys.stdout.write
data,ans=input().strip().split("\n"),[]
for lines in data:
    a,b=map(int,lines.split())
    ans.append(gcd(a,b))
output("\n".join(map(str,ans))+"\n")
```

这里输入过程仍然套用原来的格式即可，但注意，这里data是一个类似很多行的数据，通过strip删去空元素（防止输出中断）；在实际调用过程中，利用for lines in data语句代替原先的for循环，再对lines进行处理，注意这里每行lines包含的数据就和原先的逐行输入形式完全一致；利用这串代码还可以很好地代替try except语句。

递归问题解决 oj28717 字符串比较：

```
def cmpstr(a,b):
    if b=="":
        return "NO"
    else:
        if a=="":
            return "YES"
        else:
            if abs(ord(a[0])-ord("k"))<abs(ord(b[0])-ord("k")):
                return "YES"
            elif abs(ord(a[0])-ord("k"))>abs(ord(b[0])-ord("k")):
                return "NO"
            else:
                return cmpstr(a[1:],b[1:])
p=int(input())
for _ in range(p):
    a,b=input().split()
    print(cmpstr(a,b))
```

核心思路其实就是按位进行比较，但值得注意的是，第一次尝试本题时在调用cmpstr(a[1:],b[1:])时忘记写了return。从思路上，这样即便执行了这部分代码，相当于else这部分的值是YES或者NO，但注意，这个情况下函数并没有返回任何值！这带给我们的做题经验是，要搞清楚函数是如何返回的，返回了什么，是否返回到了主函数中。

oj05585 晶矿的个数

```
dir=[[1,0],[0,-1],[0,1],[-1,0]]
def dfs(x,y,ores,color):
```

```
    ores[x][y]="#"
    for direction in dir:
        x+=direction[0]
        y+=direction[1]
        if ores[x][y]==color:
            ores[x][y]="#"
            dfs(x,y,ores,color)
        x-=direction[0]
        y-=direction[1]
    return ores
n=int(input())
for _ in range(n):
    l=int(input())
    ore=[["#" for _ in range(l+2)]for _ in range(l+2)]
    cntr,cntb=0,0
    for i in range(1,l+1):
        ore[i][1:l+1]=list(input())
    for prow in range(1,l+1):
        for pcol in range(1,l+1):
            if ore[prow][pcol]=="r":
                ore=dfs(prow,pcol,ore,"r")
                cntr+=1
            elif ore[prow][pcol]=="b":
                ore=dfs(prow,pcol,ore,"b")
                cntb+=1
    print(cntr,cntb)
```

非常类似先前做到的小游戏一题，但更简单一些，涉及到走迷宫问题的策略，也就是深搜算法的实现，其本质上也是一个递归的过程；注意思考本题的函数是如何退出的，以及从哪个x，y的位置退出的（实际上是由于四个方向均无法前行，接着一遍一遍返回ores的值并退回到最开始的位置，最后退回到原位）。

oj18182 打怪兽

```
import collections
import heapq
nCases=int(input())
for _ in range(nCases):
    n,m,b=map(int,input().split())
    d=collections.defaultdict(list)#如果d不是键，那么创建一个空列表
    for _ in range(n):
        ti,xi=map(int,input().split())
        if len(d[ti])<m:
            heapq.heappush(d[ti],xi)#添加最小值
        else:
            heapq.heappushpop(d[ti],xi)#直接插入的同时弹出最小值
    for t in d:
        d[t]=sum(d[t])
    dp=sorted(d.items())
    for time,damage in dp:
        b-=damage
        if b<=0:
            print(time)
```

```
            break
    else:
        print("alive")
```

语法层面，最后部分告诉我们，if else语句是可以跨越for循环写的；对于元组列表，可以直接利用双变量接取元组两个元素的值。

数据结构：堆。在python中，堆可以利用列表直接实现。堆可以理解为一个小根堆，也就是说，根节点的值一定大于等于子节点，虽然他的算法比较复杂，但是他能做到，把将第一个值变为列表的最小值，同时修改时可以自动进行排序，同时也很方便弹出最小值。heapq的函数比较常用的有heappush，heaappushpop(先插入元素，再弹出堆头)，nlargest(m,n)（返回一个列表，为n的前m个最大值），时间复杂度比直接找最大值要快。

字典技巧：d=defaultdict(default)。d被定义为一个字典，在调用字典的键时如果没有该键，那么就按default类型创造一个空的该类型（空列表或是整形0）。dict.items()：类似enumerate，可以将字典的键和值压到一个元组列表中。

oj 02386 find lakes

```
import sys
n,m=map(int,input().split())
sys.setrecursionlimit(20000)
pond=[["."for _ in range(m+2)] for _ in range(n+2)]
tags=[[True for _ in range(m+2)] for _ in range(n+2)]
for i in range(1,n+1):
    lines=list(str(input()))
    pond[i][1:m+1]=lines
nums=0
def findlakes(x,y):
    tags[x][y]=False
    for dirx in [-1,0,1]:
        for diry in [-1,0,1]:
            if tags[x+dirx][y+diry] and pond[x+dirx][y+diry]=="W":
                findlakes(x+dirx,y+diry)
    return
for i in range(1,n+1):
    for j in range(1,m+1):
        if tags[i][j] and pond[i][j]=="W":
            nums+=1
            findlakes(i,j)
print(nums)
```

dfs的模板题，走迷宫类问题的经典套路。可以改进的地方在于，首先标签可以去掉，直接在原数据上进行修改即可；方向可以直接通过一个遍历完成（手敲一个元组列表）。这里可以看到，setrecursionlimit是一个非常必要的操作，递归深度调小了很容易导致RE错误。

oj 23927 走出迷宫

```
n=int(input())
dir=[(1,0),(0,1)]
maze=[[1 for _ in range(n+2)]for _ in range(n+2)]
```

```
for i in range(1,n+1):
    maze[i][1:n+1]=list(map(int,input().split()))
stack=[(1,1)]
while stack:
    x,y=map(int,stack.pop())
    for xmove,ymove in dir:
        if maze[x+xmove][y+ymove]==1:
            continue
        maze[x+xmove][y+ymove]=1
        stack.append((x+xmove,y+ymove))
    if x==y==n:
        print("Yes")
        break
else:
    print("No")
```

个人认为这串代码已经足够精炼而且清晰了，与所有走迷宫类题目完全类似，只不过此题用到了列表的栈的性质进行了优化，其实本质上是一种思想。注意，这里其实并没有进行回溯，只是将所有岔路口可能的选项都存起来了。

线段树与位运算专题：

```
from collections import OrderedDict
od=OrderedDict()
od["a"]=1
od["b"]=2
od["c"]=3
od["d"]=21
del od["a"]
print(od.popitem())
#删除并返回最后一个键值对，如果加上last=True则返回第一个键值对，模拟堆栈
print(od.pop("b"))#删除键并返回相应的值



p=5
print(p>>1,p<<1,(p+1)>>1,(p-1)>>1,p^1)
#位运算：>>表示右移1位，等价于//2；<<表示左移一位，等价于*2
# ^表示异或算符，表示相同取1，不相同取0，可以用于局部反转，反转奇偶性
# &表示与算符，均为1取1，否则为0，用于判断奇偶性

n=8
arr=[1,5,4,3,5,7,5,7]
segtree=[0]*2*n
#构建叶节点的值
for i in range(n):
    segtree[n+i]=arr[i]
#构建父节点的值，对于i的父节点，两个孩子位置在2*i以及2*i+1
for i in range(n-1,0,-1):
    segtree[i]=segtree[2*i]+segtree[2*i+1]
def updatetreenode(p,value):
    segtree[p+n]=value
    p=p+n
```

```
    while p>1:
        segtree[p>>1]=segtree[p]+segtree[p^1] #实际上p^1就是通过树的一个孩子找到另一个
孩子
        p>>=1 #这个语句完全等价于从一个树的子节点寻找到根节点位置
updatetreenode(2,8)
#注意线段树求和用的是左闭右开区间
def query(l,r):
    res,l,r=0,l+n,r+n
    while l<r:
#注意子节点的左孩子标号会是从偶数开始的
        if (l&1):
            res+=segtree[l]
            l+=1
        if (r&1):#判断奇偶性，对于多出来的元素直接加到求和里，不断返回上一层
            r-=1
            res+=segtree[r]
        l>>=1
        r>>=1
    return res
print(segtree,query(2,5))
```

oj 01958 strange towers of Hanoi

```
dp=[float("inf")]*14
dp[1]=1
print(dp[1])
for i in range(2,13):
    for j in range(1,i):
        dp[i]=min(2*dp[j]+pow(2,i-j)-1,dp[i])
    print(dp[i])
```

核心思想非常高级：仍然是考虑子问题，将其拆分为三个步骤。将若干个盘子经由两个辅助杆放到一个指定的塔上；将剩余的盘子经由一个辅助杆放到一个指定塔上；将若干个盘子经由两个辅助杆放到一个指定的塔上。第一步与第三步均可使用dp，第二步可以直接使用三柱汉诺塔的结论。

几个有借鉴意义的排序算法：

```
def QuickSort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0] # Choose the first element as the pivot
        left = [x for x in arr[1:] if x < pivot]
        right = [x for x in arr[1:] if x >= pivot]
    return QuickSort(left) + [pivot] + QuickSort(right)
if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
20,
    17, 10]
    print(arr_in)
    arr_out = QuickSort(arr_in)
```

```
    print(arr_out)
```

快速排序，最常用，利用二分法的思想，找到一个基准不断将数据分为左边和右边两部分。

```
def MergeSort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = MergeSort(arr[:mid])
    right = MergeSort(arr[mid:])
    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
    j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
20,
    17, 10]
    print(arr_in)
    arr_out = MergeSort(arr_in)
    print(arr_out)
```

归并排序，采用分治的思想，也是采用一分为二的方法。

```
def BubbleSort(arr):
    for i in range(len(arr) - 1):
        for j in range(len(arr) - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
20,
    17, 10]
    print(arr_in)
    arr_out = BubbleSort(arr_in)
    print(arr_out)
```

冒泡排序，最经典的排序算法，只需要相邻项比较即可实现。

oj 04137 最小新整数

```
t=int(input())
for _ in range(t):
    n,k=map(int,input().split())
    n=list(str(n))
    m,stack,cnt=len(n)-k,[n[0]],0
    for i in range(1,len(n)):
        while cnt<k and stack and n[i]<stack[-1]:
            stack.pop()
            cnt+=1
        stack.append(n[i])
    while len(stack)>m:
        stack.pop()
    print(*stack,sep="")
```

单调栈来了：此题的单调栈模型并不困难，关键在于，此题结合了贪心算法，更恶心的地方在于如何讨论一串相等的数字，他们究竟是否应该被弹走或是保留。实际上，这取决于他们后面的数字，因此结论是要保留，但这就要求了，如果后面的数字比他们小，那就需要把前面几个连续的全部弹走，而非一个一个弹走。

oj 21458 健身房

```
T,n=map(int,input().split())
exercise,dp=[],[[-float("inf") for _ in range(T+1)] for _ in range(n+1)]
for _ in range(n):
    ti,wi=map(int,input().split())
    exercise.append((ti,wi))
exercise.sort()
for i in range(1,n+1):
    for j in range(1,T+1):
        if dp[i-1][j]!=-float("inf"):
            dp[i][j]=max(dp[i-1][j],dp[i][j])
            if j+exercise[i-1][0]<=T:
                dp[i][j+exercise[i-1][0]]=max(dp[i][j+exercise[i-1][0]],dp[i-1]
[j]+exercise[i-1][1])
        if exercise[i-1][0]==j:
            dp[i][j]=max(dp[i][j],exercise[i-1][1])
print(dp[n][T] if dp[n][T]!=-float("inf") else -1)
```

写的可以说是史中之史，对于“恰好”类型背包还是太过于生疏了，使用二维dp，险些超时。

```
t,n=map(int,input().split())
dp=[0]+[-1]*(t+1)
for i in range(n):
    k,w=map(int,input().split())
    for j in range(t,k-1,-1):
        if dp[j-k]!=-1:
            dp[j]=max(dp[j-k]+w,dp[j])
print(dp[t])
```

大佬的代码，其中有两个非常巧妙的地方：一个是类似硬币一题的处理技巧，将第零项设为0，其余设为负无穷，这样能有效处理“5块钱可以由1个5块硬币的方式给出”这个条件；另一个是对“01”型背包的处理，直接在输入的过程中进行dp处理，节省时间，同时巧妙解决了“每类只有一个”的限制。

LC 42 接雨水

```
class Solution(object):
    def trap(self, height):
        l,water=len(height),0
        hleft,hright,stackleft,stackright=[],[],[],[]
        for j in range(l):
            if stackleft==[] or stackleft[-1]<=height[j]:
                stackleft.append(height[j])
            hleft.append(stackleft[-1])
            if stackright==[] or stackright[-1]<=height[l-j-1]:
                stackright.append(height[l-j-1])
            hright.append(stackright[-1])
        for i in range(l):
            water+=min(hleft[i],hright[l-i-1])-height[i]
        return water
```

最后采用的方法其实并不是严格的单调栈，但基本类似，转化成为找到每一个位置左右两边最高的元素，按行扫描；另一种思路是利用双指针，按列进行扫描。

oj 01384 piggy-bank

```
t=int(input())
for _ in range(t):
    e,f=map(int,input().split())
    coins,we=[],f-e
    n=int(input())
    for _ in range(n):
        p,w=map(int,input().split())
        coins.append((p,w))
    dp=[0]+[float("inf")]*we
    for i in range(1,we+1):
        mn=float("inf")
        for j in coins:
            if j[1]<=i:
                mn=min(mn,dp[i-j[1]]+j[0])
        dp[i]=mn
    if dp[we]!=float("inf"):
        print(f"The minimum amount of money in the piggy-bank is {dp[we]}.")
    else:
        print("This is impossible.")
```

典中典之，非要用二维列表把问题搞复杂，并不是说有两个参量的问题都需要二维列表。本题和硬币非常非常像，包括外加一个0其余全部赋值inf的处理。

sy 321 迷宫最短路径

```
import collections
direction=[(1,0),(-1,0),(0,1),(0,-1)]
def bfs(x1,y1):
```

```
    pathset={(x1,y1)}
    queue=collections.deque()
    queue.append((x1,y1,[(x1+1,y1+1)]))
    while queue:
        x,y,path=queue.popleft()
        if x==n-1 and y==m-1:
            break
        for dir in direction:
            xnew=x+dir[0]
            ynew=y+dir[1]
            if 0<=xnew<=n-1 and 0<=ynew<=m-1 and maze[xnew][ynew]==0 and
not((xnew,ynew) in pathset):
                p=(path+[(xnew+1,ynew+1)]).copy()
                queue.append((xnew,ynew,p))
                pathset.add((xnew,ynew))
    return path
n,m=map(int,input().split())
maze=[[1 for _ in range(m)] for _ in range(n)]
for i in range(n):
    maze[i]=list(map(int,input().split()))
pathend=bfs(0,0)
for x,y in pathend:
    print(x,y)
```

bfs模板题，核心思想是使用双向队列，从右边不断加入下一层新的元素，并不断弹出最左边的元素，由于最后要输出路径结果，从而需要存储先前的路径。

oj 27310 积木

```
tag=[True]*4
def search(word):
    global cube,success,tag
    if word==[]:
        success=1
        return
    x=word[0]
    for i in range(4):
        if tag[i] and x in cube[i]:
            tag[i]=False
            search(word[1:len(word)])
            tag[i]=True
    return
n=int(input())
cube=[]
for _ in range(4):
    cube.append(list(str(input())))
for _ in range(n):
    success=0
    search(list(str(input())))
    print("YES" if success==1 else "NO")
```

披着暴力的dfs，实现方式很多，但要想到此题相当于一个走迷宫问题，不同字母可以对应不同的色子，为了保证同一个色子只能用一次，要加一个标记数组，同时要注意回溯的问题。

oj 28389 跳高

```
from collections import deque
import bisect
n=int(input())
lst=list(map(int,input().split()))
highest=deque([lst[0]])
for i in range(1,n):
    if highest[0]>lst[i]:
        highest.appendleft(lst[i])
    else:
        p=bisect.bisect_right(highest,lst[i])
        highest[p-1]=lst[i]
print(len(highest))
```

很简约的代码，本质其实是构造一个单调队列，再使用二分查找。如果目前高度小于前面最小值，则在左端入队，反之则在满足单调的情况下将特定位置增高。

leetcode 135 分发糖果

```
class Solution(object):
    def candy(self,ratings):
        n=len(ratings)
        left_to_right=[1]*n
        right_to_left=[1]*n
        for i in range(1,n):
            if ratings[i]>ratings[i-1]:
                left_to_right[i]=left_to_right[i-1]+1
        for i in range(n-2,-1,-1):
            if ratings[i]>ratings[i+1]:
                right_to_left[i]=right_to_left[i+1]+1
        total=0
        for i in range(n):
            total+=max(left_to_right[i],right_to_left[i])
        return total
```

和接雨水放在一起这个题不是没有原因的，同样是利用了左右单调栈，数学模型实际上是，找到一个数左边单调序列长度与右边单调序列长度的最大值，再求和。

oj 09267 核电站

```
def nuclear(x,y,memo):
    if x==1:
        return 1 if y==1 else 2
    if (x,y) in memo:
        return memo[(x,y)]
    if y!=1:
        memo[(x,y)]=nuclear(x-1,y-1,memo)+nuclear(x-1,m,memo)
    else:
        memo[(x,y)]=nuclear(x-1,m,memo)
    return memo[(x,y)]
n,m=map(int,input().split())
memo={}
print(nuclear(n,m,memo))
```

递归解法，直接做会导致超时，这是因为很多重复的项会计算很多遍，因此建立一个字典存放先前计算过的值。此外，此题递归的思路是用(x,y)表示当前状态，x表示目前还剩下多少个位置，y表示目前还能连续放几个核材料。

系统总结背包问题：

01背包

```
n,v=map(int,input().split())
weight=list(map(int,input().split()))
prices=list(map(int,input().split()))
dp=[([0]+[-float("inf") for _ in range(v)]) for _ in range(n+1)]
for i in range(1,n+1):
    for j in range(1,v+1):
        if j>=weight[i-1]:
            dp[i][j]=max(dp[i-1][j],dp[i-1][j-weight[i-1]]+prices[i-1])
        else:
            dp[i][j]=dp[i-1][j]
print(max(dp[n]))
```

二维dp，不推荐，但一定要掌握，会涉及到反转求和的问题，解决方案之一是直接对上一层进行求和即可。因为如果不这样处理会使得同一个物体可以被取用很多遍。

求出取样方案：

```
n,v=map(int,input().split())
weight=list(map(int,input().split()))
prices=list(map(int,input().split()))
dp,choice=[0]*(v+1),[[] for _ in range(v+1)]
for i in range(n):
    for j in range(v,weight[i]-1,-1):
        cpy=choice[j-weight[i]]+[i+1]
        if dp[j-weight[i]]+prices[i]>dp[j]:
            dp[j]=dp[j-weight[i]]+prices[i]
            choice[j]=cpy
        elif dp[j-weight[i]]+prices[i]==dp[j]:
            if cpy<choice[j]:
                choice[j]=cpy
print(dp[-1])
```

```
print(*choice[-1])
```

一个小的知识点：列表和字符串是可以直接进行字典序比大小的。这里改进为了一维dp，涉及到倒序求和的技巧，这里需要注意的是，一维dp是可以便输入边处理的，但代价是需要用倒序的方式防止重复次数的出现。

实战：

恰好01背包

```
n,v=map(int,input().split())
weight=list(map(int,input().split()))
dp=[1]+[0]*v
for i in range(n):
    for j in range(v,weight[i]-1,-1):
        dp[j]=(dp[j]+dp[j-weight[i]])%10007
print(dp[-1])
```

只是把取最大值改为了求和，同时注意到01背包的逆序问题。

完全背包

oj 04117 简单的整数划分

```
while True:
    try:
        n=int(input())
        dp=[([1]+[0 for _ in range(n)]) for _ in range(n+1)]
        for i in range(1,n+1):
            for j in range(1,n+1):
                if i>j:
                    dp[i][j]=dp[i-1][j]
                else:
                    dp[i][j]=dp[i-1][j]+dp[i][j-i]
        print(dp[n][n])
    except EOFError:
        break
```

完全背包对应的是顺序遍历，但这里涉及到无序的问题，建立二维dp可以有效解决该问题。如果利用一维结果，那么会出现1+2+1和1+1+2计算两遍的情形。

```
pow2=[i for i in range(1,51)]
while True:
    try:
        num=int(input())
        dp=[1]+[0]*(num)
        for i in pow2:
            if i>num:
                break
            for j in range(i,num+1):
                dp[j]+=dp[j-i]
        print(dp[-1])
    except EOFError:
        break
```

完全背包是用无限数量的物品填充，与下题的区别在于，这里要用递增的数字去填充这个dp列表。

cf 431C k-tree

```
n,k,d=map(int,input().split())
dp,dpp=[0]*(n+1),[0]*(n+1)
dp[0],dpp[0]=1,1
for i in range(1,n+1):
    for j in range(1,k+1):
        if j>i:
            break
        dp[i]=(dp[i-j]+dp[i])
for i in range(1,n+1):
    for j in range(1,d):
        if j>i:
            break
        dpp[i]=(dpp[i-j]+dpp[i])
print((dp[-1]-dpp[-1])%1000000007)
```

本题实际上对应的是完全背包的有序问题，因此直接建立一维dp列表即可实现，因为这里1+1+2和1+2+1会被记为两种情况。

确定数量背包问题：

oj 01664 放苹果

```
t=int(input())
for _ in range(t):
    n,m=map(int,input().split())
    dp=[[1]*(n+1)]*2+[([1]+[0 for _ in range(n)]) for _ in range(m-1)]
    for i in range(2,m+1):
        for j in range(1,n+1):
            if j>=i:
                dp[i][j]=dp[i-1][j]+dp[i][j-i]
            else:
                dp[i][j]=dp[j][j]
    print(dp[-1][-1])
```

本题的想法是，实际上我们擅长做的是考虑可以空的盘子，这样实际上我们把不可空的盘子数量统一扣除一个即变成了可以空的盘子数量。顺着思路往下，如果可空的盘子数大于苹果数，那多出的那部分盘子一定空。

多限制背包：

```
import sys
n,m,k=map(int,input().split())
dp=[[0]*m]+[[float("inf") for _ in range(m)] for _ in range(k)]
for i in range(1,k+1):
    cost,attack=map(int,input().split())
    for j in range(i,0,-1):
        for s in range(attack,m):
            if dp[j-1][s-attack]+cost<=n:
                dp[j][s]=min(dp[j-1][s-attack]+cost,dp[j][s])
for q in range(k,0,-1):
    for s in range(m):
```

```
        if dp[q][s]!=float("inf"):
            print(q,m-s)
            sys.exit()
print(0,m)
```

这里有三个参数，精灵球数，血量以及能捉到的个数，实际上可以将范围最大的参数作为dp列表中的元素，从而降低时间复杂度；注意在遍历小精灵个数时应当采用倒序的0-1背包方法。

多重背包：

```
while True:
    m,n=map(int,input().split())
    if n==m==0:
        break
    lst=list(map(int,input().split()))
    prices,coins=lst[:m],lst[m:]
    dic=dict(zip(prices,coins))
    dp=[1]+[0]*n
    for i in range(m):
        cost=prices[i]
        nums=dic[cost]
        k=1
        while nums>=k:
            nums-=k
            costeff=cost*k
            numseff=k
            for i in range(n,costeff-1,-1):
                if dp[i-costeff]:
                    dp[i]=1
            k*=2
        if nums>0:
            costeff=cost*nums
            numseff=nums
            for i in range(n,costeff-1,-1):
                if dp[i-costeff]:
                    dp[i]=1
    print(sum(dp)-1)
```

对于多重背包，可以利用一系列二进制数对其进行“打包”处理，在判断可否/求解极值的问题中可以直接利用除法，但如果要求解方案数需要进行严格的二进制分解操作。

```
from typing import List
class Solution:
    def waysToReachTarget(self, target: int, types: List[List[int]]) -> int:
        dp=[1]+[0]*target
        for count,money in types:
            for j in range(target,0,-1):
                for k in range(1,min(count,j//money)+1):
                    dp[j]+=dp[j-money*k]
                dp[j]=dp[j]%(pow(10,9)+7)
        return dp[-1]
```

对于统计数量的多重背包，似乎不能通过二进制的方式优化，这时采用1~n枚举的方式，同时倒叙枚举dp列表实现遍历。

lc 72 编辑距离

```
class Solution(object):
    def minDistance(self, word1, word2):
        lst1,lst2=list(word1),list(word2)
        l1,l2=len(lst1),len(lst2)
        dp=[[i for i in range(l1+1)]]+[([j+1]+[0 for _ in range(l1)]) for j in
range(l2)]
        for j in range(1,l1+1):
            for i in range(1,l2+1):
                if lst1[j-1]==lst2[i-1]:
                    dp[i][j]=dp[i-1][j-1]
                else:
                    dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1
        return dp[-1][-1] if word1!=word2 else 0
```

一道精彩绝伦的题目，三种不同操作竟然恰好能够对应到三类转移方程，同时，空字符串的情形也是坑点满满。

oj 27237 体育游戏跳房子

```
from collections import deque
def bfs(a,b):
    queue=deque([(a,"")])
    pastpath=set()
    pastpath.add(a)
    while queue:
        pos,ways=queue.pop()
        if pos==b:
            return ways
        p1=3*pos
        p2=int(pos/2)
        for p,toss in [(p1,"H"),(p2,"O")]:
            if p not in pastpath:
                pastpath.add(p)
                queue.appendleft((p,ways+toss))
while True:
    n,m=map(int,input().split())
    if n==m==0:
        break
    m1=bfs(n,m)
    print(len(m1))
    print(m1)
```

一道非常全面的bfs题目，常见错误是MLE，关键在于要想到使用字符串可以直接记录路径而应当避免使用列表；同时，字典序的要求可以说是一个超级大坑，实际上根本不需要额外关注字典序的问题，只要求H先于O执行即可，坑!

lc 3 无重复子序列

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        from collections import defaultdict
        nearest,st,mx=defaultdict(lambda:-1),0,0
        for i,char in enumerate(s):
            if nearest[char]>=st:
                st=nearest[char]+1
            nearest[char]=i
            mx=max(mx,i-st+1)
        return mx
```

滑动窗口问题，建立一个字典并不断更新每个元素最新出现的位置，在遍历整个字符串的过程中不断更新最大值。这里使用了defaultdict函数，并更改了默认值。

sy 386 最短距离

```
import heapq
def dijkstra(st,ed):
    global graph,n
    pq=[(0,st)]
    distance=[float("inf")]*n
    distance[st]=0
    while pq:
        dis,pos=heapq.heappop(pq)
        if pos==ed:
            return dis
        for nexpos,nexdis in graph[pos]:
            newdis=nexdis+dis
            if newdis<distance[nexpos]:
                distance[nexpos]=newdis
                heapq.heappush(pq,(newdis,nexpos))
    return -1 if distance[ed]==float("inf") else distance[ed]
n,m,s,t=map(int,input().split())
graph=[[] for _ in range(n)]
for _ in range(m):
    u,v,w=map(int,input().split())
    graph[u].append((v,w))
    graph[v].append((u,w))
print(dijkstra(s,t))
```

本题利用了dijkstra算法，类似一个广搜，建立一个小根堆使得每次先遍历目前为止距离最近的节点，但注意堆内元组的顺序！

lc 630 课程表III

```
import heapq
class Solution(object):
    def scheduleCourse(self, courses):
        courses.sort(key=lambda x:x[1])
        l,su,gocourse=len(courses),0,[]
        for i in range(l):
            heapq.heappush(gocourse,-courses[i][0])
            su+=courses[i][0]
            if su>courses[i][1]:
                su-=-heapq.heappop(gocourse)
        return len(gocourse)
```

大顶堆最nb的一集，和potions是完全一样的思路，不确定每一项能不能取到，那就建立一个堆存起来可能会弹出的最大值。

lc 1889 装包裹的最小浪费空间

```
import bisect
class Solution(object):
    def minWastedSpace(self, packages, boxes):
        packages.sort()
        boxes.sort(key=lambda x:min(x))
        l=len(packages)
        suffix,mn=[packages[0]]+[0]*(l-1),float("inf")
        for i in range(1,l):
            suffix[i]=suffix[i-1]+packages[i]
        for x in boxes:
            x.sort()
            remains,pre=0,0
            if x[-1]<packages[-1]:
                continue
            for k in x:
                p=bisect.bisect_right(packages,k)
                if p>=1:
                    remains+=(p-pre)*k-suffix[p-1]+(suffix[pre-1] if pre!=0 else
0)
                    pre=p
            mn=min(mn,remains)
        return mn%1000000007 if mn!=float("inf") else -1
```

几个关键节点，将包裹排序然后利用二分查找的方法，遍历每一个box再对每个盒子大小进行排序，贪心过程的本质仍然是一个排序算法。

lc 2136 全部开花的最早一天

```
from functools import cmp_to_key
def swap(a,b):
    if max(a[0]+a[1],a[0]+b[1]+b[0])>max(b[0]+b[1],b[0]+a[1]+a[0]):
        return 1
    else:
        return -1
```

```
class Solution(object):
    def earliestFullBloom(self, plantTime, growTime):
        l,st,ed=len(growTime),-1,0
        totalgrow=[(plantTime[i],growTime[i]) for i in range(l)]
        totalgrow.sort(key=cmp_to_key(swap))
        for i in range(l):
            st+=totalgrow[i][0]
            ed=max(ed,st+totalgrow[i][1]+1)
        return ed
```

交换类贪心问题，之前做到过的最大数最小数就是这一类问题，关键是找到交换比较的依据，实际上国王游戏也可以用该方法进行排序。

lc 2931 购买物品最大开销

```
import heapq
class Solution(object):
    def maxSpending(self, values):
        stack,n,m,price=[],len(values),len(values[0]),0
        for i in range(n):
            stack.append((values[i][-1],i))
            values[i].pop()
        heapq.heapify(stack)
        for j in range(n*m):
            cost,indx=heapq.heappop(stack)
            price+=cost*(j+1)
            if values[indx]:
                elem=values[indx].pop()
                heapq.heappush(stack,(elem,indx))
        return price
```

利用排序不等式的贪心问题，即小的对小的，大的对大的。实现过程中利用了堆数据结构进行了优化。

oj 26978 滑动窗口最大值

```
from collections import deque
n,k=map(int,input().split())
lst=list(map(int,input().split()))
queue=deque()
for i in range(n):
    while queue and queue[0]<=i-k:
        queue.popleft()
    while queue and lst[queue[-1]]<lst[i]:
        queue.pop()
    queue.append(i)
    if i>=k-1:
        print(lst[queue[0]],end=" ")
```

建立一个元素递减的双端队列，但这个队列实际上是一个指针队列，存放的是数组的指标而非值，这一步的思路非常巧妙，解决无法准确弹出一定角标范围内的值的问题。

lc 416 分割等和子集/lc 698 划分为k个等和子集

```
class Solution(object):
    def canPartition(self, nums):
        l=len(nums)
        if sum(nums)%2==1:
            return "false"
        else:
            target=sum(nums)//2
            dp=[True]+[False]*(target)
            for i in range(l):
                for j in range(target,nums[i]-1,-1):
                    if dp[j-nums[i]]:
                        dp[j]=True
            if dp[-1]:
                return "true"
            else:
                return "false"
```

对于k=2的情形，可以变为一个dp问题：即是否村存在和为target的子序列。这是一个判断型的0-1背包，倒序遍历即可。此题很容易脑抽，需要想到从总和的角度入手，不要暴力比较任意两个子序列。

```
from functools import lru_cache
def canPartitionKSubsets(nums,k):
    lru_cache(maxsize=None)
    def recursion(nums,su,k):
        nums=list(nums)
        su=list(su)
        if nums:
            for i in range(k):
                if nums[0]+su[i]<=target:
                    su[i]+=nums[0]
                    if recursion(tuple(nums[1:]),tuple(su),k):
                        return True
                    su[i]-=nums[0]
                else:
                    continue
            return False
        else:
            for i in range(k):
                if su[i]!=target:
                    return False
            return True
    if sum(nums)%k!=0:
        return False
    else:
        target=sum(nums)//k
        nums.sort(reverse=True)
        su=[nums[0]]+[0 for _ in range(k-1)]
        return recursion(tuple(nums[1:]),tuple(su),k)
```

一道非常有启发性的递归题目，涵盖了多方面技巧：关于lru_cache的调用，maxsize可以设置为None，但注意，如果函数中是可变类型，则要将其转换为不可变类型；递归终点何时返回值要考虑清楚。

yhf's help:

```
from functools import lru_cache
from typing import List

class Solution:
    def canPartitionKSubsets(self, nums: List[int], k: int) -> bool:
        total = sum(nums)
        if total % k != 0:
            return False

        target = total // k
        nums.sort(reverse=True)

        # Early exit if the largest number is greater than the target
        if nums[0] > target:
            return False

        # The 'su' array keeps track of the sum of each subset
        su = [0] * k

        @lru_cache(maxsize=None)
        def recursion(nums_tuple, su_tuple):
            nums = list(nums_tuple)
            su = list(su_tuple)

            if not nums:
                return all(s == target for s in su)

            current_num = nums[0]
            for i in range(k):
                if su[i] + current_num <= target:
                    su[i] += current_num
                    if recursion(tuple(nums[1:]), tuple(su)):
                        return True
                    su[i] -= current_num

                # If the current subset is empty and adding the number doesn't
work, break
                if su[i] == 0:
                    break

            return False

        return recursion(tuple(nums), tuple(su))
```

这里是考虑到，如果这个篮子是0，那么后面所有篮子的和仍旧也将为0，因此就不需要再遍历了，是非常有效的剪枝。

oj 26646 建筑修建

```
n,m=map(int,(input().split()))
st,ed,cnt=0,m-1,1
```

```
for _ in range(n):
    x,y=map(int,input().split())
    if x>ed:
        stpre=max(ed+1,x-y+1)
        if stpre+y-1<=m-1:
            cnt+=1
            ed=stpre+y-1
            st=stpre
    else:
        newed=max(st,x-y+1)+y-1
        if ed>=newed:
            ed=newed
print(cnt)
```

和很久之前的woodcutter非常的像，仍旧是尽量靠左放置，但这里是一个运动的区间，因此更新st以及ed的方式又有点像进程检测；当然，这个题更难一点可以先考察一下排序（一开始想了很久为什么要按x排序）

sy 361 学校的班级人数

```
def find(x):
    if parent[x]!=x:
        parent[x]=find(parent[x])
    return parent[x]
def union(a,b):
    parent[find(a)]=find(b)
n,m=map(int,input().split())
parent,stat=[i for i in range(n+1)],[0]*(n+1)
for _ in range(m):
    a,b=map(int,input().split())
    union(a,b)
classes=list(set(find(x) for x in range(1,n+1)))
print(len(classes))
for i in range(1,n+1):
    stat[find(i)]+=1
stat.sort(reverse=True)
for x in range(len(classes)-1):
    print(stat[x],end=" ")
print(stat[len(classes)-1])
```

并查集的核心思想：如何寻找根节点；如何建立联系；进行操作。这里想进一步让链表关系更为清晰，可以改写为：

```
def union(a,b):
    roota=find(a)
    rootb=find(b)
    if roota!=rootb:
        parent[roota]=rootb
```

在这种情况下，本题想要进行的统计操作也就非常简单了。

sy 134 全排列

```
from itertools import permutations
n=int(input())
a=sorted(list(set(list(permutations(list(map(int,input().split()))))))
for x in a:
    print(*x)
```

当我用出permutation，阁下又该如何应对？permutation(numlist,num)，第一个参数是所有元素的个数，第二项是有几个变量参与排序，注意本身他就是按升序输出排序的。

oj 18164 剪绳子

```
import heapq
n=int(input())
lst=list(map(int,input().split()))
heapq.heapify(lst)
cnt=0
while len(lst)>1:
    mn1=heapq.heappop(lst)
    mn2=heapq.heappop(lst)
    heapq.heappush(lst,mn1+mn2)
    cnt+=(mn1+mn2)
print(cnt)
```

nbdhuffman算法，关键在于它神奇的思路，每次找最小的两个进行合并。这里甚至都没有用到树本身的结构，只是用到了它权值和最小的性质。其实构建树的过程已经给出了证明：离根越远的地方加的次数越多，当然希望它越小越好，而堆的数据结构又非常好的可以实现这一点。

oj 01694 波兰表达式

```
sign=["+","-","*","/"]
lst=list(input().split())
stack,auxstack=[],[]
for i in range(len(lst)-1,-1,-1):
    if lst[i] not in sign:
        stack.append(float(lst[i]))
    else:
        a,b=stack.pop()
        stack.append(eval(str(a)+lst[i]+str(b)))
print(f"{float(stack[0]):.6f}")
```

本质上其实是树的思想，从后向前遍历。

oj 18155 组合乘积

```
def dfs(lst,aim,target):
    if aim==target:
        return True
```

```
    if lst==[]:
        return False
    for i in range(len(lst)):
        if abs(aim*lst[i])<=abs(target):
            aim=aim*lst[i]
            if dfs(lst[i+1:],aim,target):
                return True
            aim=aim//lst[i]
    return False
target=int(input())
numlst=list(map(int,input().split()))
numlst.sort(reverse=True)
l=len(numlst)
if (1 not in numlst and numlst.count(-1)<2) and target==1:
    print("NO")
else:
    print("YES" if dfs(numlst,1,target) else "NO")
```

题目本身不难，是非常常规的递归dfs问题，但需要注意aim在运算过程中是浮动的！类似国王游戏，因此需要用//进行运算。

lc 123 买卖股票的最佳时机

```
from typing import List
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        l=len(prices)
        dpmn1=[prices[0]]+[0]*(l-1)
        mxproleft,mxproright=[0]*l,[0]*l
        dpmn2=[0]*(l-1)+[prices[-1]]
        for i in range(1,l):
            dpmn1[i]=min(dpmn1[i-1],prices[i])
            mxproleft[i]=prices[i]-dpmn1[i]
            dpmn2[l-i-1]=max(dpmn2[l-i],prices[l-i-1])
            mxproright[l-i-1]=max(mxproright[l-i],dpmn2[l-i-1]-prices[l-i-1])
        mxpro=[0]+[mxproleft[i]+mxproright[i+1] for i in range(l-1)]+
[mxproleft[-1]]
        return max(mxpro)
```

仔细思考，会发现此题同机智股民牢章的关系就和拦截导弹之于登山的关系一样，把同一个过程从左右重复两遍。

oj 4115 鸣人和佐助

```
from collections import deque
direction=[(1,0),(-1,0),(0,1),(0,-1)]
def bfs(chac,stx,sty,edx,edy):
    queue=deque([(stx,sty,chac,0)])
    pathset=set([(stx,sty,chac)])
    while queue:
        xnow,ynow,chacnow,t=queue.popleft()
        for dx,dy in direction:
```

```
            xnew=xnow+dx
            ynew=ynow+dy
            if xnew==edx and ynew==edy:
                return t+1
            if 0<=xnew<m and 0<=ynew<n:
                if (xnew,ynew,chacnow) not in pathset:
                    if trace[xnew][ynew]=="#":
                        if chacnow>0:
                            queue.append((xnew,ynew,chacnow-1,t+1))
                            pathset.add((xnew,ynew,chacnow-1))
                    else:
                        queue.append((xnew,ynew,chacnow,t+1))
                        pathset.add((xnew,ynew,chacnow))
    return -1
m,n,t=map(int,input().split())
trace=[""]*m
for i in range(m):
    trace[i]=input()
    if "@" in trace[i]:
        stx=i
        sty=trace[i].index("@")
    if "+" in trace[i]:
        edx=i
        edy=trace[i].index("+")
print(bfs(t,stx,sty,edx,edy))
```

几个优化的点：地图不一定要建二维列表，直接用字符串表示也可以；对于特殊类型的bfs，可以考虑更改入队条件，比如变换的迷宫增添了时间指标，此题增加了查克拉指标。此题不需要对查克拉做额外的优化，这是因为题目中本身查克拉就比较少。

oj 7622 求排列的逆序数

```
def mergesort(x,n):
    if n==1:
        return x,0
    fr=n//2
    lst1,re1=mergesort(x[:fr],fr)
    lst2,re2=mergesort(x[fr:],n-fr)
    sre,re,p1,p2,t=[],0,0,0,re1+re2
    while re<n:
        if p1==fr:
            while p2<n-fr:
                sre.append(lst2[p2])
                p2+=1
            break
        if p2==n-fr:
            while p1<fr:
                sre.append(lst1[p1])
                p1+=1
            break
        if lst2[p2]<lst1[p1]:
            t+=(fr-p1)
            sre.append(lst2[p2])
```

```
            p2+=1
        else:
            sre.append(lst1[p1])
            p1+=1
        re+=1
    return sre,t
n=int(input())
lst=list(map(int,input().split()))
x,p=mergesort(lst,n)
print(p)
```

本题考察的是归并排序，操作起来是递归的思想，注意排序过程的写法以及逆序数的计算方法。

lc 871 最低加油次数

```
from typing import List
class Solution:
    def minRefuelStops(self, target: int, startFuel: int, stations:
List[List[int]]) -> int:
        l=len(stations)
        dp=[startFuel]+[0]*l
        for i,(pos,oil) in enumerate(stations):
            for j in range(i,-1,-1):
                if dp[j]>=pos:
                    dp[j+1]=max(dp[j+1],dp[j]+oil)
        return next((i for i,v in enumerate(dp) if v>=target),-1)
```

dp法，思路非常难想，数组的含义是，加k次油最多可以走到的距离。

```
from typing import List
import heapq
class Solution:
    def minRefuelStops(self, target: int, startFuel: int, stations:
List[List[int]]) -> int:
        stations.sort()
        l=len(stations)
        stations=[(0,0)]+stations
        oil=[(-stations[i][1],stations[i][0]-stations[i-1][0]) for i in
range(1,l+1)]+[(0,target-stations[-1][0])]
        move,rem,cnt=[],startFuel,0
        for i in range(l+1):
            rem-=oil[i][1]
            while rem<0:
                if move:
                    energy,pos=heapq.heappop(move)
                    rem+=-energy
                    cnt+=1
                else:
                    return -1
            heapq.heappush(move,oil[i])
        return cnt
```

最大堆法，稍微好想一些，利用了反悔贪心的最大堆法。

lc 32 有效括号匹配

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        l=len(s)
        dp=[0]*(l+1)
        for i in range(2,l+1):
            if s[i-1]==")":
                if s[i-2]=="(":
                    dp[i]=dp[i-2]+2
                else:
                    if i-dp[i-1]-2>=0 and s[i-dp[i-1]-2]=="(":
                        dp[i]=dp[i-dp[i-1]-2]+dp[i-1]+2
        return max(dp)
```

孩子你无敌了（（（和编辑距离一样，一个让人无法理解如何想到的dp转移方程，唯一找一点套路的话，可能是以这个点结尾的最长有效括号，之后分类讨论上一位。

lc 76 最小覆盖子串

```
from collections import deque,Counter
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        sldwindow=deque([])
        l,cnt,mnlen=len(s),len(t),(float("inf"),"")
        needdic=dict(Counter(t))
        keyset=set(needdic.keys())
        for j in range(l):
            sldwindow.append(j)
            if s[j] in keyset:
                needdic[s[j]]-=1
                if needdic[s[j]]>=0:
                    cnt-=1
            if cnt==0:
                for i in range(sldwindow[0],j+1):
                    sldwindow.popleft()
                    if s[i] in keyset:
                        if needdic[s[i]]<0:
                            needdic[s[i]]+=1
                        else:
                            if j-i+1<mnlen[0]:
                                mnlen=(j-i+1,s[i:j+1])
                            if needdic[s[i]]>=0:
                                cnt+=1
                            needdic[s[i]]+=1
                            break
        return mnlen[1]
```

孩子你又无敌了，nnd期末要是你敢考我就跟你爆了！双端队列可以替换同向双指针，利用了counter统计目标的元素个数。

区间问题

区间分组问题

oj 4144 畜栏保留问题

```
import heapq
n=int(input())
cows=[]
for k in range(n):
    a,b=map(int,input().split())
    cows.append((a,b,k))
cows.sort(key=lambda x:(x[0],x[1]))
cnt,feed=1,[0]*n
feed[cows[0][2]]=1
mnqueue=[(cows[0][1],1)]
heapq.heapify(mnqueue)
for j in range(1,n):
    last,idx=heapq.heappop(mnqueue)
    if cows[j][0]>last:
        feed[cows[j][2]]=idx
        heapq.heappush(mnqueue,(cows[j][1],idx))
    else:
        cnt+=1
        heapq.heappush(mnqueue,(last,idx))
        heapq.heappush(mnqueue,(cows[j][1],cnt))
        feed[cows[j][2]]=cnt
print(cnt)
print(*feed,sep="\n")
```

这是一个区间分组的模板题，时间复杂度为O(nlogn)，先对开始时间排序，然后贪心，让下一头牛叠在上一头牛的后面，为了实现这个操作，可以建一个最小堆存放每个畜栏结束的时间，然后每次弹最小值。

区间覆盖问题

lc 1024 视频剪接

```
from typing import List
class Solution:
    def videoStitching(self, clips: List[List[int]], time: int) -> int:
        clips.sort(key=lambda x:(x[0],-x[1]))
        l=len(clips)
        st,ed,i,cnt=0,time,0,0
        while st<ed and i<l:
            maxrange=0
            while i<l and clips[i][0]<=st:
                maxrange=(max(maxrange,clips[i][1]))
                i+=1
            if maxrange<=st:
                return -1
            st=maxrange
            cnt+=1
            if st>=time:
```

```
                break
        if st<time:
            return -1
        return cnt
```

按照起点排序，每次寻找小于等于起点的片段所能更新的最大值。

oj 1661 help jimmy

```
#help jimmy 347ms
import sys
from functools import lru_cache
sys.setrecursionlimit(200000)
t=int(input())
for _ in range(t):
    @lru_cache(maxsize=None)
    def dfs(stx,sty,n,hm):
        time,q=float("inf"),0
        for i in range(n):
            if plains[i][0]<=stx<=plains[i][1] and 0<sty-plains[i][2]<=hm:
                #分别考虑左边右边
                timer=min(dfs(plains[i][0],plains[i][2],n,hm)+stx-plains[i][0],\
                          dfs(plains[i][1],plains[i][2],n,hm)+plains[i][1]-
stx)+sty-plains[i][2]
                time=min(timer,time)
                q=1
                #上了一个平台就不能上第二个
                break
        #落地条件
        if sty<=hm and q==0:
            time=sty
        return time
    n,stx,sty,hmax=map(int,input().split())
    plains=[]
    for _ in range(n):
        plains.append(tuple(map(int,input().split())))
    #平台按照高度排序
    plains.sort(key=lambda x:-x[2])
    print(dfs(stx,sty,n,hmax))
```

kbd今明老师，深搜+lru_cache，分别考虑左右然后遍历。

oj 1724 roads

```
from collections import defaultdict
import heapq
def dijkstra(stpoint,citynum,money):
    queue=[(0,stpoint,0)]
    heapq.heapify(queue)
    while queue:
        dis,st,cost=heapq.heappop(queue)
        visited[st-1]=cost
```

```
        if st==citynum:
            return dis
        for next,path,expense in graph[st]:
            if expense+cost<=money:
                if visited[next-1]>=expense+cost:
                    heapq.heappush(queue,(dis+path,next,expense+cost))
    return -1
money=int(input())
citynum=int(input())
roads=int(input())
graph=defaultdict(list)
for _ in range(roads):
    s,d,l,t=map(int,input().split())
    graph[s].append((d,l,t))
visited=[0]+[float("inf")]*(citynum-1)
print(dijkstra(1,citynum,money))
```

dijkstra算法，利用堆的特性。原则上讲有几种处理方法：不加距离判断，但记录遍历过的位置inque；加距离判断visited，但这里其实只是为了判断是不是经过过这里，因此可以换成别的参量，记录距离参量可以原地更新，但其他参量需要后置更新。

lc 394 字符串解码

```
nums=set([str(i) for i in range(10)])
class Solution:
    def decodeString(self, s: str) -> str:
        l=len(s)
        stack,sget,numget=[],[],""
        for i in range(l):
            if s[i]=="]":
                while stack[-1]!="[":
                    sget.append(stack.pop())
                stack.pop()
                sget=sget[-1::-1]
                while stack and stack[-1] in nums:
                    numget+=stack.pop()
                numget=numget[-1::-1]
                sget*=int(numget)
                stack+=sget
                numget,sget="",[]
            else:
                stack.append(s[i])
        target="".join(stack)
        return target
```

本题是波兰表达式的升级版本，最大的差异在于此题是正序遍历，判断的依据是左括号的出现，困难！

oj 4030 统计单词数

```
s=input().lower()
text=" "+input().lower()+" "
indx=text.find(" "+s+" ")
cnt=text.split().count(s)
if indx==-1:
    print(-1)
else:
    print(cnt,indx)
```

一道很无聊的语法题，但又必须要掌握。含空格的问题，一定要考虑到可能有多个空格的情况；find函数可以用来寻找给定字符串中特定的子串。

oj 25815 回文字符串

```
s=input()
l=len(s)
dp=[[0]*i+[float("inf") for _ in range(l-i)] for i in range(1,l+1)]
for y in range(1,l):
    for pl in range(0,l-y):
        ynew=y+pl
        xnew=pl
        if s[xnew]==s[ynew]:
            dp[xnew][ynew]=dp[xnew+1][ynew-1]#
        else:
            dp[xnew][ynew]=min(dp[xnew+1][ynew],dp[xnew][ynew-1],dp[xnew+1][ynew-
1])+1
print(dp[0][-1])
```

编辑距离+回文字符串的判断，关键考虑到#行转移方程的含义——如果i位和j位相同，那最小编辑距离就是i-1位到j-1位的编辑距离。

oj 4135 月度开销

```
import sys,math
sys.setrecursionlimit(200000)
def check(put):
    global m
    cnt,bucket=0,1
    for i in range(n):
        if cost[i]>put:
            return False
        if cnt+cost[i]<=put:
            cnt+=cost[i]
        else:
            cnt=cost[i]
            bucket+=1
    if bucket<=m:
        return True
    else:
        return False
n,m=map(int,input().split())
```

```
cost=[]
for _ in range(n):
    cost.append(int(input()))
su=sum(cost)
lim=math.ceil(su/m)
ans=cost[0]
bucket=[0]*m
while lim<=su:
    mid=(lim+su)//2
    ju=check(mid)
    if ju:
        ans=mid
        su=mid-1
    else:
        lim=mid+1
print(ans)
```

与过河一样的思路，从平均值的角度入手。

cf 2033D kousuke's assignment

```
t=int(input())
for _ in range(t):
    n=int(input())
    numset,cnt,lstsum=set(),0,[0]*(n+1)
    lst=[0]+list(map(int,input().split()))
    for i in range(1,n+1):
        lstsum[i]=lstsum[i-1]+lst[i]
    for i in range(n+1):
        if lstsum[i] not in numset:
            numset.add(lstsum[i])
        else:
            numset.clear()
            numset.add(lstsum[i])
            cnt+=1
    print(cnt)
```

先进行前缀和的处理，很像愉悦的旋律，要寻找一个序列中两个相同的数的个数，那就找不同的数最多有多长。

lc 15 三数之和

```
for i in range(l):
    if nums[i]>0:
        break
    if i>0 and nums[i]==nums[i-1]:
        continue
    p1,p2=i+1,l-1
    while p1<p2:
        while nums[p1]+nums[p2]+nums[i]<0:
            p1+=1
        while nums[p1]+nums[p2]+nums[i]>0:
```

```
            p2-=1
        if p1<p2 and nums[p1]+nums[p2]+nums[i]==0:
            threenum.append([nums[p1],nums[p2],nums[i]])
            p1+=1
            p2-=1
            while p1<p2 and nums[p1]==nums[p1-1]:
                p1+=1
            while p2>p1 and nums[p2]==nums[p2+1]:
                p2-=1
```

左端固定指针，右端为两个移动指针向中心靠拢。