一、遍历与基本语法。

状态：Accepted

源代码

```
result=[]
while True:
    n,m=map(int,input().split())
    if n==0 and m==0:
        break
    queue=[i for i in range(1,n+1)]
    start=0

    while len(queue)>1:
        removed=(start+m-1)%len(queue) #取余确定要移除的元素，不想超出就取模
        queue.pop(removed) #删除
        start=removed % len(queue)#更新起点位置，取模不超过队伍

    result.append(queue[0])
for num in result:
    print(num)
```

```
from collections import deque

def josephus(n, k):
    # 初始化队列，将所有人的编号按顺序加入队列
    queue = deque(range(1, n + 1))

    result = []
    while queue:
        # 将前 k-1 个人移到队列的末尾
        for _ in range(k - 1):
            queue.append(queue.popleft())
        # 移出第 k 个人，并输出其编号
        result.append(queue.popleft())

    # 输出结果，最后一个元素后面没有空格
    print(" ".join(map(str, result)))

# 读取输入
n, k = map(int, input().split())
josephus(n, k)
```

• 取模防止超出边界 从列表中删除元素　　也可以使用队列

小数的输入和小数点后两位的保留：n=float(input())　　n=f'{n:.2f}'　　print(n)

数学模块的使用：四舍五入为 round(),

在同一行隔空格输入多个 print(' '.join(map(str,result))) result 是一个列表，想把列表里面的东西逐一输出

纯字母字符串可以直接比较 aaaa 和 abba 可以直接比较字典序

判断数列的单调性，可以一开始就把整个列表定为 TRUE，遇到不对的就 False 掉然后马上 break

罗马数字与整数之间的转换（整数与罗马字典）

```
def romanToInt(self, s: str) -> int:
    ROMAN = {'I': 1,'V': 5,'X': 10,'L': 50,'C': 100,'D': 500,'M': 1000,}
    ans = 0
    for x, y in pairwise(s):  # 遍历相邻的罗马数字
        x, y = ROMAN[x], ROMAN[y]
        ans += x if x >= y else -x
    return ans + ROMAN[s[-1]]  # 加上最后一个
```

```
    # 使用哈希表，按照从大到小顺序排列
    hashmap = {1000:'M', 900:'CM', 500:'D', 400:'CD', 100:'C', 90:'XC', 50:'L', 40:'XL', 10:'X',
9:'IX', 5:'V', 4:'IV', 1:'I'}
    res = ''
    for key in hashmap:
        if num // key != 0:
            count = num // key  # 比如输入4000，count 为 4
            res += hashmap[key] * count
            num %= key
    return res
```

```
def is_monotonic_increasing(n, a):
    isAsc = True  # 假设序列是单调递增的
    for i in range(n - 1):
        if a[i] > a[i + 1]:  # 如果发现某一对相邻元素不满足递增条件
            isAsc = False  # 标记为非单调递增
            break  # 跳出循环
    return "YES" if isAsc else "NO"  # 根据标志位输出结果
```

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        for i in range(len(nums)):
            for j in range(i+1,len(nums)):
                if nums[i]+nums[j]==target:
                    return [i,j]
```

两数之和输出两者下标，直接使用暴力遍历，两层循环就可以了。

字典 多函数

```
day1=input()
day2=input()
result=[]
for i in range(4):
    if int(day1[i]) < int(day2[i]):
        result.append('YES')
        break
else:

    for i in range(2):
        if int(day1[5+i]) < int(day2[5+i]):
            result.append('YES')
            break
    else:

        for i in range(2):
            if int(day1[8+i]) < int(day2[8+i]):
                result.append('YES')
                break

if len(result)==0:
    result.append('NO')

for i in result:
    print(i)
```

这个想告诉我们一个什么事呢，就是比较完以后储存结果，要怎么才能跳到下一步。这个是在比较时间

```
l1,l2=l1[::-1],l2[::-1]
#这里不能用reverse
print(l1)
print(l2)
num1=int(''.join(str(x) for x in l1))
num2=int(''.join(str(y) for y in l2))
print(num1)
print(num2)
num=str(num1+num2)
result=[]
for i in range(0,len(num)):
    result.append(num[i])
print(result)
```

涉及列表逆转还有提取字符串

```
ame=[]
max_l=0
for char in s:
    if char in ame:
        max_l=max(max_l,len(ame))
        ame=ame[ame.index(char)+1:]
    ame.append(char)
max_l=max(max_l,len(ame))

return max_l
```

处理最长非重复字符串，也是两层循环，然后不断更新已经见过的，思路也挺好玩的。

```
def longestCommonPrefix(self, strs: List[str]) -> str:
    same=strs[0]
    for j,c in enumerate(same):
        for str in strs:
            if j ==len(str) or str[j]!=c:
                return same[:j]
    return same
```

寻找最长公共前缀，把他想成一个矩阵，通过对每一列的字母进行遍历来更新

```
lst_words=list(input().split())
lst_words=lst_words[::-1]
result=[]
for word in lst_words:
    word=word.lower()
    word=word.capitalize()
    result.append(word)
print(' '.join(str(m) for m in result))
```

```
s=input()
start_char=s[0]
start_count=1
result=[]
for i in range(1,len(s)):
    if s[i]==s[i-1]:
        start_count+=1
    else:
        result.append([start_char,start_count])
        start_char=s[i]
        start_count=1
    if i==len(s)-1:
        result.append([start_char,start_count])
for num in result:
    print(' '.join(str(m) for m in num))
```

首字母大写，逆序，大小写，对了字母序是可以直接比较的，你还记得吗。|这个是用来处理索引问题的，越界了要时刻注意索引的大小

```
a,b=map(int,input().split())
count=0
for num in range(a,b+1):
    num=str(num)
    if '1' in num:
        count+=num.count('1')
print(count)
```

```
s=input()
for _ in range(8):
    m,n=map(int,input().split())
    s=s[:m]+s[m:n][::-1]+s[n:]
print(s)
```

这个可以统计字符串中某个字符出现的次数|　　拼接字符串以及局部翻转

```
    n, ans = len(points), 1
    for i, x in enumerate(points):
        for j in range(i + 1, n):
            y = points[j]
            # 枚举点对 (i,j) 并统计有多少点在该线上，起始 cnt = 2 代表只有 i 和 j 两个点在此线上
            cnt = 2
            for k in range(j + 1, n):
                p = points[k]
                s1 = (y[1] - x[1]) * (p[0] - y[0])
                s2 = (p[1] - y[1]) * (y[0] - x[0])
                if s1 == s2: cnt += 1
            ans = max(ans, cnt)
    return ans
```

遍历查找最多多少个点共线

```
new_s=''.join(char.lower() for char in s if char.isalnum())
new_s=new_s[::-1]
return new_s
```

快速提取字符串中的所有字母用 isalnum，提取所有数字用 isdigit

```
n=int(input())
matrix=[[0 for _ in range(n)] for _ in range(n)]
start=1
top,bottom,left,right=0,n-1,0,n-1
#上下左右边界的初始值
#思路基本上没错，就是通过右下左上的顺序不断循环，主要是要不断
while start <=n*n:#start是用来表示当前处理到的位置，也可
    for i in range(left,right+1):#先往右走
        matrix[top][i]=start
        start+=1
    top+=1#处理完最上行

    for i in range(top,bottom+1):
        matrix[i][right]=start
        start+=1
    right-=1#处理完最右列
    if top<=bottom:#需要检查边界有空间可以填充
        for i in range(right,left-1,-1):
            matrix[bottom][i]=start
            start+=1
        bottom-=1#处理完最下行
    if left<=right:
        for i in range(bottom,top-1,-1):
            matrix[i][left]=start
            start+=1
        left+=1#处理完最左列
```

```
result=[]
pigs=[]#主栈
min_pigs=[]
while True:
    try:
        command=input().strip()
        if command=='':
            break


        if command=='pop':
            if pigs: #主栈非空，自动排除了没有猪的情况
                removed=pigs.pop()
                if removed==min_pigs[-1]:
                    min_pigs.pop() #有可能是最小的猪一起被移
        elif command.startswith('push'):
            _,weight=command.split()
            weight=int(weight)
            pigs.append(weight)
            if not min_pigs or weight<=min_pigs[-1]:
                min_pigs.append(weight)#且要保证最小的在顶
        elif command=='min':
            if min_pigs:
                result.append(min_pigs[-1])
            else:
                pass

    except EOFError:
        break
for i in result:
    print(i)
```

矩阵处理中的螺旋矩阵，专门处理这种环绕式的走法　辅助栈的使用以及处理空行为止的多行输入。

```
for _ in range(n):
    a.append(tuple(input().split()))
a=sorted(a,key=lambda x: (-int(x[1]),x[0]))
for c,b in a:
```

多元素排序写法

```
    def convert(self, s: str, numRows: int) -> str:
        if numRows < 2: return s
        res = ["" for _ in range(numRows)]
        i, flag = 0, -1
        for c in s:
            res[i] += c
            if i == 0 or i == numRows - 1: flag = -flag
            i += flag
        return "".join(res)
```

z 型变换，使用 flag 来控制方向，我懂了。

二、dp

```python
n=int(input())
tower=[]
for _ in range(n):
    tower.append(list(map(int,input().split())))
dp=tower[-1].copy()
for i in range(n-2,-1,-1):
    for j in range(len(tower[i])):
        dp[j]=max(dp[j],dp[j+1])+tower[i][j]
print(dp[0])
```

数塔 dp，倒着来，最大值。

```python
n=int(input())
nums=list(map(int,input().split()))
dp=nums.copy()
start=[0]*n
end=[0]*n
start[0]=0
for i in range(1,n):
    dp[i]=max(dp[i-1]+nums[i],nums[i])
    if dp[i]==dp[i-1]+nums[i]:
        start[i]=start[i-1]
        end[i]=i
    else:
        start[i]=i
        end[i]=i
k=dp.index(max(dp))
print(max(dp),start[k]+1,end[k]+1)
```

|最大值查找问题，以及标记最大的区间

```python
def maxProfit(self, prices: List[int]) -> int:
    cost, profit = float('+inf'), 0
    for price in prices:
        cost = min(cost, price)
        profit = max(profit, price - cost)
    return profit
```

```python
n=int(input())
nums=list(map(int,input().split()))
dp=[1]*n
for i in range(1,n):
    for j in range(i):
        if nums[i]>=nums[j]:
            dp[i]=max(dp[i],dp[j]+1)
print(max(dp))
```

最长上升子序列

```python
n=int(input())
nums=list(map(int,input().split()))
dp=[1]*n
pre=[-1]*n #记录前一个元素下标
result=[]
for i in range(1,n):
    for j in range(i):
        if nums[i]>=nums[j]:
            if dp[i]<dp[j]+1:
                dp[i]=dp[j]+1
                pre[i]=j
k=dp.index(max(dp)) #最后一个元素的下标

while k !=-1:
    result.append(nums[k])
    k=pre[k]
result.reverse()
print(max(dp))
print(' '.join(str(x) for x in result))
```

要记录元素的子序列问题，这个时候一般有一个辅助数组

```python
tex_1=input().strip()
tex_2=input().strip()
m=len(tex_1)
n=len(tex_2)
dp=[[0]*(n+1) for _ in range(m+1)] #表示1的前i个字母和2的前j个字母的最长公共序列长度
for i in range(1,m+1):
    for j in range(1,n+1):
        if tex_1[i-1]==tex_2[j-1]:
            dp[i][j]=dp[i-1][j-1]+1
        else:
            dp[i][j]=max(dp[i-1][j],dp[i][j-1])#那就分别忽略两个序列的第i/j个字符进行判断
print(dp[m][n])
```

最长公共字符串问题，我去看看如果要我输出字符串的话要怎么办。

```python
lcs = []
i, j = m, n
while i > 0 and j > 0:
    if text1[i - 1] == text2[j - 1]:
        lcs.append(text1[i - 1])
        i -= 1
        j -= 1
    elif dp[i - 1][j] > dp[i][j - 1]:
        i -= 1
    else:
        j -= 1
return lcs[::-1]
```

```python
n,v=map(int,input().split())
weig=list(map(int,input().split()))
pric=list(map(int,input().split()))
dp=[[0]*(v+1) for _ in range(n+1)] #用背包重量遍历
for i in range(1,n+1):
    for j in range(v+1):#可能的背包重量
        if j>=weig[i-1]:#j表示的是当前背包重量
            dp[i][j]=max(dp[i-1][j],dp[i-1][j-weig[i-1]]+pric[i-1])
        else:
            dp[i][j]=dp[i-1][j]


print(dp[n][v])
```

最基本背包，都是二维

```
n,v=map(int,input().split())
weight=list(map(int,input().split()))
values=list(map(int,input().split()))
dp=[0]*(v+1) #初始化为一维动态规划问题
for i in range(n):
    for j in range(weight[i],v+1):
        dp[j]=max(dp[j],dp[j-weight[i]]+values[i]) #装进去以后背包剩余容量会减少但是价值会增加
print(dp[v])
```

完全背包问题，也就是一个物品可以选择多次。

```
while True:
    try:
        n=int(input())
        dp=[[0]*(n+1) for _ in range(n+1)] #dp[m][n]表示将m分成最大数为n有多少种分法
        for i in range(n+1):
            dp[0][i]=1 #零怎么分都只有一种
        for j in range(1,n+1):
            dp[j][1]=1

        for i in range(1,n+1):
            for j in range(1,n+1):
                if j>i: #分的最大数不能超过当前数字
                    dp[i][j]=dp[i][i]
                else:
                    dp[i][j]=dp[i-j][j]+dp[i][j-1]
                    #表示不包括j有多少种方法以及至少有一个j，剩下的数字就是i-j
        print(dp[n][n])
    except EOFError:
        break
```

整数拆解问题，将当前最大数保留或者不保留，保留就往下找剩下的数字怎么分，不保留就看看下一个最大数怎么分，记得初始化好就行。

```
while True:
    try:
        str1,str2=map(str,input().split())
        n,m=len(str1),len(str2)
        dp=[[0]*(m+1) for _ in range(n+1)]
        dp[0][0]=0
        for i in range(1,m+1):
            dp[0][i]=0
        for i in range(1,n+1):
            dp[i][0]=0
        for i in range(1,n+1):
            for j in range(1,m+1):
                if str1[i-1]==str2[j-1]:
                    dp[i][j]=1+dp[i-1][j-1]
                else:
                    dp[i][j]=max(dp[i-1][j],dp[i][j-1])
        print(dp[n][m])
```

字符串的公共子序列

```
def longest_palindromic_substring_length(s: str) -> int:
    n = len(s)
    if n == 0:
        return 0
    # dp[i][j] 表示 s[i:j+1] 是否为回文串
    dp = [[False] * n for _ in range(n)]
    max_length = 1  # 初始化最长回文子串的长度为 1
    # 单个字符一定是回文串
    for i in range(n):
        dp[i][i] = True
    # 处理长度为 2 的子串
    for i in range(n - 1):
        if s[i] == s[i + 1]:
            dp[i][i + 1] = True
            max_length = 2
    # 处理长度大于 2 的子串
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j] and dp[i + 1][j - 1]:
                dp[i][j] = True
                max_length = length
    return max_length
# 示例输入
s = input().strip()
print(longest_palindromic_substring_length(s))
```

最长回文字符串

```python
def min_energy_to_reach_end(n, heights):
    # 初始化 dp 数组，dp[i] 表示到达第 i 个高台的最小能量消耗
    dp = [0] *n
    # 初始条件：到达第一个高台不需要能量
    dp[0] = 0
    # 从第二个高台开始计算
    for i in range(1, n):
        # 从第 i-1 个高台跳过来的能量消耗
        dp[i] = dp[i - 1] + abs(heights[i] - heights[i - 1])

        # 如果可以从第 i-2 个高台跳过来，则取最小值
        if i - 2 >= 0:
            dp[i] = min(dp[i], dp[i - 2] + abs(heights[i] - heights[i - 2]))
    # 返回到达最后一个高台的最小能量消耗
    return dp[n - 1]
n = int(input())
heights = list(map(int, input().split()))
print(min_energy_to_reach_end(n, heights))
```

最小消耗能量问题

```python
def count_sequences(n):
    mod = 10007
    dp = [1, 9]  # dp[0] 表示前一位选 0，dp[1] 表示前一位不选 0
    for i in range(2, n + 1):
        tmp = [0, 0]
        tmp[0] = dp[1]  # 当前位是 0，前一位不能是 0
        tmp[1] = (dp[0] + dp[1]) * 9 % mod  # 当前位不是 0，前一位可以是 0 或不是 0
        dp =tmp
    return (dp[0] + dp[1])%mod
n = int(input())
print(count_sequences(n))
```

寻找长度为 n 的序列不能有两个连续的 0 的序列数量

```python
def min_cost_to_paint(n, costs):
    # 初始化dp数组
    dp = [[0] * 3 for _ in range(n)]

    # 初始条件
    dp[0][0] = costs[0][0]
    dp[0][1] = costs[0][1]
    dp[0][2] = costs[0][2]

    # 状态转移
    for i in range(1, n):
        dp[i][0] = min(dp[i-1][1], dp[i-1][2]) + costs[i][0]  # 涂红色
        dp[i][1] = min(dp[i-1][0], dp[i-1][2]) + costs[i][1]  # 涂黄色
        dp[i][2] = min(dp[i-1][0], dp[i-1][1]) + costs[i][2]  # 涂蓝色

    # 最终结果
    return min(dp[n-1])

# 输入处理
n = int(input())
costs = [list(map(int, input().split())) for _ in range(n)]

# 输出结果
print(min_cost_to_paint(n, costs))
```

最小涂色问题，就是连着两个不能涂成相同的颜色，且要成本最小，二位数组做下来

```python
def min_edit_distance(s: str, t: str) -> int:
    len_s, len_t = len(s), len(t)
    # 初始化dp数组，大小为 (len_s + 1) x (len_t + 1)
    # dp[i][j] 表示将 s 的前 i 个字符转换为 t 的前 j 个字符所需的最小编辑距离
    dp = [[0] * (len_t + 1) for _ in range(len_s + 1)]
    # 将空字符串转换为 t 的前 j 个字符需要 j 次插入操作
    for j in range(len_t + 1):
        dp[0][j] = j
    # 将 s 的前 i 个字符转换为空字符串需要 i 次删除操作
    for i in range(len_s + 1):
        dp[i][0] = i
    # 状态转移方程：填充dp数组
    for i in range(1, len_s + 1):
        for j in range(1, len_t + 1):
            if s[i - 1] == t[j - 1]:
                # 如果当前字符相同，则不需要额外操作，继承左上角的值
                dp[i][j] = dp[i - 1][j - 1]
            else:
                # 如果当前字符不同，则取三种操作中最小的那个并加1
                dp[i][j] = min(dp[i - 1][j] + 1,dp[i][j - 1] + 1,dp[i - 1][j - 1] + 1)#分别对应删除插入和替换
    return dp[len_s][len_t]
```

动态规划解决编辑距离问题

三，贪心。

```python
h=int(input())
m=int(input())
courses=[]
for _ in range(m):
    s,c=map(float,input().split())
    courses.append([s,c])

for course in courses:
    course.append(course[0] * course[1])
courses.sort(key=lambda x: -x[2]) #按照性价比逆序排序
total_time=2*h-0.5*m
total_increase=0
for course in courses:
    if total_time<=0:
        break
    time=min(total_time,5/course[0])
    total_time-=time
    total_increase+=course[0]*time*course[1]

print(f'{total_increase:.1f}')
```

```python
n,m=map(int,input().split())
r=list(map(int,input().split()))
r.sort()
rs=[]
for i in range(len(r)-1):
    rs.append(r[i+1]-r[i])
rs.sort()
print(sum(rs[:n-m]))
```

贪心处理 GPA，lambda 排序，矩阵扩展维度，小数输出以及浮点数处理　　贪心处理差值

一个贪心，安装雷达，就是不断更新右端点　　涉及字符串拼接和倒序考虑贪心

```python
import math
def findings(n,r,points):  1 个用法
    if r< 0:
        return -1
    ranges=[]
    for x,y in points:
        if y>r:
            return -1
        delta=math.sqrt(r**2-y**2)
        ranges.append((x-delta,x+delta))#这里是把这个邻域存起来，主要是我们已经简化成了x轴上的讨论
    if not ranges:#如果没有符合要求的邻域，也找不到
        return -1
    ranges.sort(key=lambda x:x[1])#将得到的邻域进行排序，但是我不太搞得懂为什么是按照邻域之后进行排序
    number=1
    k=ranges[0][1]
    for start,end in ranges[1:]:
        if k<start:
            k=end
            number+=1
    return number
```

```python
for i in range(len(num)-1,-1,-1):
    if int(num[i])%2==0:
        a=num[i]
        num=num[:i]+num[i+1:]+[a]
        k=1
        break
```

```python
import math
def id(A):
    if math.sqrt(A).is_integer():
        return "Yes"
    for i in range(1,len(str(A))):
        left=int(str(A)[:i])
        right=int(str(A)[i:])
        if left>0 and math.sqrt(left).is_integer() and id(right)=="Yes" and not str(right).startswith('0'):
            return "Yes"
    return "No"
A=int(input())
print(id(A))
```

受到祝福，体现一个分割还有递归

贪心处理搭桥问题

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        ans = 0  # 记录跳跃次数
        cur_right = 0  # 当前跳跃范围内能到达的最远位置（桥的右端点）
        next_right = 0  # 下一步能到达的最远位置的最大值（下一座桥的右端点最大值）
        for i in range(len(nums) - 1):
            # 更新下一步能到达的最远位置。每个方格能让我们跳到的最远距离，并记住其中最远的一个。
            next_right = max(next_right, i + nums[i])
            # 如果我们已经走到了当前跳跃范围的最右边（即到达了`cur_right`），
            # 这意味着我们需要建造一座新桥来继续前进。
            if i == cur_right:
                # 更新当前跳跃范围内的最远可达位置为下一步能到达的最远位置。
                cur_right = next_right
                # 因为我们用完了之前的跳跃范围，所以需要再跳一次。
                ans += 1
        return ans  # 返回总跳跃次数
```

加油站，环形贪心

```python
# ans：最终返回的起始索引
# min_s：遍历过程中遇到的最小油量（用于确定新的潜在起点）
# s：当前剩余油量（初始为0）
ans = min_s = s = 0
# 遍历所有加油站及其对应的消耗量
for i, (g, c) in enumerate(zip(gas, cost)):
    # 更新当前油量：在第i个加油站加油g单位后，减去从i到i+1所需的c单位油
    s += g - c
    # 如果当前油量小于之前记录的最小油量，则更新最小油量
    if s < min_s:
        # 更新最小油量min_s，并设置ans为下一个位置（i+1）作为新的潜在起点。
        # 注意：此时汽车已经移动到了i+1的位置，所以将ans设为i+1。
        min_s = s
        ans = i + 1
# 循环结束后，s即为所有加油站的加油量总和减去消耗量总和。
# 如果总的净油量s小于0，说明无论从哪个点开始都无法完成一圈，返回-1。
# 否则返回找到的起始索引ans。
return -1 if s < 0 else ans
```

四、查找和矩阵。

```python
import heapq
n,m,p=map(int,input().split())
directions=[(-1,0),(1,0),(0,1),(0,-1)]
matrix=[]
for _ in range(n):
    matrix.append(list(input().split()))
result=[]
for _ in range(p):
    ans='NO' #初始化为走不到
    s_x,s_y,e_x,e_y=map(int,input().split())

    if matrix[s_x][s_y]!='#' and matrix[e_x][e_y]!='#': #如果初始化可以通过
        dict={(s_x,s_y):0} #使用字典记录到达每一个节点的最小消耗
        queue=[(0,s_x,s_y)] #开始记录

        while queue:
            k,i,j=heapq.heappop(queue) #从队列中取出点

            if i==e_x and j==e_y:
                ans=k
                break

            for w,z in directions:
                new_x,new_y=i+w,j+z
                if 0<=new_x<n and 0<=new_y<m and matrix[new_x][new_y]!='#':
                    cost=k+abs(int(matrix[new_x][new_y])-int(matrix[i][j]))

                    if (new_x,new_y) not in dict or cost<dict[(new_x,new_y)]:
                        dict[(new_x,new_y)]=cost
                        heapq.heappush(queue, __item: (cost,new_x,new_y))
    result.append(ans)
```

bfs 模板

```python
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        n = len(matrix)
        # 深拷贝 matrix -> tmp
        tmp = copy.deepcopy(matrix)
        # 根据元素旋转公式，遍历修改原矩阵 matrix 的各元素
        for i in range(n):
            for j in range(n):
                matrix[j][n - 1 - i] = tmp[i][j]
```

深拷贝解决矩阵旋转 90 度问题

```python
from collections import deque
# 定义四个方向，右、下、左、上
dire = [(0, 1), (1, 0), (0, -1), (-1, 0)]
def bfs(a, x1, y1, x2, y2):  1 个用法
    visit = set()  # 使用集合来避免重复访问
    queue = deque([(x1, y1, x2, y2)])
    visit.add((x1, y1, x2, y2))  # 初始点加入访问集合

    while queue:
        xa, ya, xb, yb = queue.popleft()
        # 遍历四个方向
        for xi, yi in dire:
            # 计算新位置
            nx1, ny1 = xa + xi, ya + yi
            nx2, ny2 = xb + xi, yb + yi

            # 判断新位置是否合法
            if 0 <= nx1 < a and 0 <= ny1 < a and 0 <= nx2 < a and 0 <= ny2 < a:
                if (nx1, ny1, nx2, ny2) not in visit and Matrix[nx1][ny1] != 1 and Matrix[nx2][ny2] != 1:
                    # 加入队列并标记访问
                    queue.append((nx1, ny1, nx2, ny2))
                    visit.add((nx1, ny1, nx2, ny2))
                    # 检查是否到达目标
                    if Matrix[nx1][ny1] == 9 or Matrix[nx2][ny2] == 9:
                        return True
    return False

# 读取输入
a = int(input())
Matrix = [list(map(int, input().split())) for _ in range(a)]
```

然后是一个 dfs 模板

```python
a = int(input())
Matrix = [list(map(int, input().split())) for _ in range(a)]

# 找到第一个和第二个 '5' 的位置
x1, y1, x2, y2 = -1, -1, -1, -1
found_first = False

for i in range(a):
    for j in range(a):
        if Matrix[i][j] == 5:
            if not found_first:
                x1, y1 = i, j
                Matrix[i][j] = 0  # 标记为已访问
                found_first = True
            else:
                x2, y2 = i, j
                Matrix[i][j] = 0  # 标记为已访问
                break
    if x2 != -1:  # 如果第二个 5 已经找到
        break

# 运行 BFS 检查是否可以从 (x1, y1) 到 (x2, y2)
check = bfs(a, x1, y1, x2, y2)
print('yes' if check else 'no')
```

```python
    m = n * 2  # 括号长度，每对括号包含一个左括号和一个右括号
    ans = []  # 存储结果的列表
    path = [''] * m  # 初始化路径数组，用于存储当前构造的括号字符串

    # 定义递归函数 dfs，参数 i 表示当前处理的位置，open 表示已使用的左括号数量
    def dfs(i: int, open: int) -> None:
        if i == m:  # 如果已经填满了 m 个括号
            ans.append(''.join(path))  # 将当前路径转换为字符串并加入答案列表
            return

        if open < n:  # 如果左括号的数量小于 n，可以添加左括号
            path[i] = '('  # 在当前位置放置左括号
            dfs(i + 1, open + 1)  # 递归调用，处理下一个位置，并增加左括号计数

        if i - open < open:  # 如果右括号的数量小于左括号的数量，可以添加右括号
            path[i] = ')'  # 在当前位置放置右括号
            dfs(i + 1, open)  # 递归调用，处理下一个位置，左括号计数不变

    dfs(0, 0)  # 从第 0 个位置开始，初始时没有使用任何左括号
    return ans  # 返回结果列表
```

生成合法括号组合的 dfs

```python
def exist(self, board: List[List[str]], word: str) -> bool:
    def dfs(i, j, k):
        if not 0 <= i < len(board) or not 0 <= j < len(board[0]) or board[i][j] != word[k]:
                return False
        if k == len(word) - 1: return True
        board[i][j] = ''
        res = dfs(i + 1, j, k + 1) or dfs(i - 1, j, k + 1) or dfs(i, j + 1, k + 1) or dfs(i, j - 1, k

        board[i][j] = word[k]
        return res

    for i in range(len(board)):
        for j in range(len(board[0])):
            if dfs(i, j, 0): return True
    return False
    #在矩阵中dfs找单词，就是很标准的dfs思路
```

更加简单的一个 dfs 模板，还比较模板化

```python
n,m=map(int,input().split())
matrix=[[0]*(m+2) for _ in range(n+2)]
for i in range(1,n+1):
    matrix[i]=[0]+list(map(int,input().split()))+[0]
#初始化
new_matrix=[[0]*(m+2) for _ in range(n+2)]
for i in range(1,n+1):
    for j in range(1,m+1):
        k=matrix[i-1][j]+matrix[i-1][j-1]+matrix[i-1][j+1]+matrix[i][j
        if matrix[i][j]==1:
            if k<2 or k>3:
                new_matrix[i][j]=0
            else:
                new_matrix[i][j]=1
        else:
            if k==3:
                new_matrix[i][j]=1
            else:
                new_matrix[i][j]=0
for i in range(1,n+1):
    print(' '.join(map(str,new_matrix[i][1:-1])))
```

加保护圈，邻居问题，注意不要越界

```python
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        row = [[0] * 9 for _ in range(9)]   # 记录每一行中每个数字是否出现过
        col = [[0] * 9 for _ in range(9)]   # 记录每一列中每个数字是否出现过
        block = [[0] * 9 for _ in range(9)] # 记录每个 3x3 子网格中每个数字是否出现过
        for i in range(9):  # i 是行索引
            for j in range(9):  # j 是列索引
                if board[i][j] != '.':  # 如果当前格子不是空格，则需要进一步处理
                    num = int(board[i][j]) - 1  # 将字符形式的数字转换为整数，并减去 1 以
适应从 0 开始的索引
                    b = (i // 3) * 3 + j // 3  # 计算当前格子所属的 3x3 子网格的唯一索引
                    # 检查该数字是否已经在对应的行、列或块中出现过
                    # 如果任何一个条件成立，则说明数独无效，立即返回 False
                    if row[i][num] or col[j][num] or block[b][num]:
                        return False
                    # 如果当前数字是第一次出现在对应的行、列和块中，则将其对应的标记位置为 1
                    # 这样可以确保后续遇到相同数字时能够正确检测到重复
                    row[i][num] = col[j][num] = block[b][num] = 1
        # 如果遍历完整个棋盘后没有发现任何重复的数字，则返回 True，表示数独有效
        return True
```

检查数独的有效，涉及矩阵处理

```python
def partition(self, s: str) -> List[List[str]]:
    n = len(s)# 获取字符串长度
    ans = []# 用于存储所有满足条件的分割方案
    path = []# 当前正在构建的分割方案（路径）
    def dfs(i: int) -> None:
        if i == n:# 如果已经遍历到了字符串末尾，说明找到了一个完整的分割方案
            ans.append(path.copy())# 注意，这里需要复制 path，因为后续会继续使用 path 进行回溯
            return
        # 枚举子串结束位置 j，从当前索引 i 到字符串末尾
        for j in range(i, n):
            # 提取子串 t = s[i:j+1]，并检查是否为回文
            t = s[i:j + 1]
            if t == t[::-1]:  # Python 中判断字符串是否为回文的方法
                # 如果是回文，则将其加入当前路径，并递归处理剩余部分
                path.append(t)
                # 递归调用，开始考虑下一个子串的起始位置 j + 1
                dfs(j + 1)
                # 回溯，移除最后加入的子串，尝试其他可能性
                path.pop()
    # 从索引 0 开始启动深度优先搜索
    dfs(0)
    # 返回所有有效的分割方案
    return ans
```

Dfs 寻找回文串，里面有一个回溯

```python
n = len(s)  # 获取字符串长度
ans = []  # 存储所有有效的IP地址
path = []  # 当前构建中的IP地址片段
def dfs(i):
    if i == n and len(path) == 4:
        # 如果已经遍历到了字符串末尾，并且已经得到了4个IP段，则找到了一个有效IP地址
        # 使用 "." 连接各个IP段，并将其添加到结果列表中
        ans.append(".".join(map(str, path.copy())))
        return

    for j in range(i, n):
        # 枚举子串结束位置 j，从当前索引 i 到字符串末尾
        t = s[i:j+1]  # 提取子串 t = s[i:j+1]

        # 如果子串包含前导零但不等于 "0"，则跳过（例如 "01", "001" 等）
        if str(int(t)) != t:
            break

        # 只有当 path 中的片段数量小于4 并且 当前子串是一个有效的IP段时，才继续递归
        if len(path) < 4 and 0 <= int(t) <= 255:
            path.append(int(t))  # 将当前子串加入路径
            dfs(j + 1)  # 递归处理剩余部分
            path.pop()  # 回溯：移除最后加入的子串，尝试其他可能性
dfs(0)  # 从索引 0 开始启动深度优先搜索
return ans  # 返回所有有效的IP地址
```

Dfs 查找一个字符串是否可以组成一个 ip 地址，实质上是进行切分看有多少种组合

五、二分

```python
    def maxArea(self, height: List[int]) -> int:
        left=0
        right=len(height)-1
        max_v=0
        while left<right:
            if height[left]<height[right]:
                max_v=max(max_v,height[left]*(right-left))
                left+=1
            else:
                max_v=max(max_v,height[right]*(right-left))
                right-=1
        return max_v
```

二分解决短板问题，主要是看怎么移动上下限。

```python
nums.sort()
n = len(nums)
ans = list()
for i in range(0,n):
    if(i>0 and nums[i]==nums[i-1]):
        continue
    j = i+1
    k = n-1
    while j<k:
        if nums[i]+nums[j]+nums[k]<0:
            j+=1
        elif nums[i]+nums[j]+nums[k]>0:
            k-=1
        else:
            ans.append([nums[i],nums[j],nums[k]])
            j+=1
            while nums[j]==nums[j-1] and j<k:
                j+=1
            k -= 1
            while j < k and nums[k] == nums[k + 1]:
                k -= 1
return ans
```

```python
for a in range(n - 3):  # 枚举第一个数
    x = nums[a]
    if a and x == nums[a - 1]:  # 跳过重复数字
        continue
    if x + nums[a + 1] + nums[a + 2] + nums[a + 3] > target:  # 优化一
        break
    if x + nums[-3] + nums[-2] + nums[-1] < target:  # 优化二
        continue
    for b in range(a + 1, n - 2):  # 枚举第二个数
        y = nums[b]
        if b > a + 1 and y == nums[b - 1]:  # 跳过重复数字
            continue
        if x + y + nums[b + 1] + nums[b + 2] > target:  # 优化一
            break
        if x + y + nums[-2] + nums[-1] < target:  # 优化二
            continue
```

二分解决三数和为 0 且不重复,四数之和只需要优化一下

二分动态规划接雨水,前后遍历更新最大最小

```python
n = len(height)  # 获取高度列表的长度
# 初始化 pre_max 数组，用于存储从左到右遍历时，每个位置及其左侧的最大柱子高度。
# pre_max[i] 表示在第 i 个位置时，它自己以及它左边所有柱子中的最大高度。
pre_max = [0] * n
if n > 0:  # 确保数组非空，避免索引错误
    pre_max[0] = height[0]  # 第一个柱子的最大高度就是它自己的高度
    for i in range(1, n):
        # 对于每个后续的位置 i，我们更新 pre_max[i] 为当前位置的高度和前一个位置最大高度中的较大者。
        # 这样做是为了确保如果我们想在这个位置储水，左边必须有足够高的柱子支撑。
        pre_max[i] = max(pre_max[i - 1], height[i])
# 初始化 suf_max 数组，用于存储从右到左遍历时，每个位置及其右侧的最大柱子高度。
# suf_max[i] 表示在第 i 个位置时，它自己以及它右边所有柱子中的最大高度。
suf_max = [0] * n
if n > 0:  # 确保数组非空，避免索引错误
    suf_max[-1] = height[-1]  # 最后一个柱子的最大高度就是它自己的高度
    for i in range(n - 2, -1, -1):
        # 对于每个往前的位置 i，我们比较当前位置的高度和下一个位置的最大高度，
        # 选择较大的那个作为当前位置的最大高度。这是为了确保在该位置储水时，右边有足够的高度支撑。
        suf_max[i] = max(suf_max[i + 1], height[i])

ans = 0  # 初始化总储水量为 0
for i, h in enumerate(height):
    # 对于每个柱子，我们可以储水的数量取决于它左右两边最大高度中的较小者，
    # 因为水位不能超过这个高度。然后我们减去柱子本身的高度 h，得到的就是该位置可以储水的数量。
    # 如果 min(pre_max[i], suf_max[i]) 小于等于当前柱子高度 h，则该位置不能储存水。
    if min(pre_max[i], suf_max[i]) > h:
        ans += min(pre_max[i], suf_max[i]) - h  # 累加当前位置的储水量
return ans  # 返回总的储水量
```

六、工具

```python
from itertools import product

class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:
            return []
        # 电话键盘上的字母映射
        apl = {'2': 'abc', '3': 'def',
            '4': 'ghi', '5': 'jkl', '6': 'mno',
            '7': 'pqrs', '8': 'tuv', '9': 'wxyz'}
        # 获取输入数字对应的字符集列表
        char_sets = [apl[digit] for digit in digits if digit in apl]
        # 如果没有有效的字符集，则返回空列表
        if not char_sets:
            return []
        # 使用 product 计算笛卡尔积，并将结果转换为字符串列表
        combinations = [''.join(combination) for combination in product(*char_sets)]
        return combinations
```

直接使用 product 生成组合

```python
class Solution:
    def isValid(self, s: str) -> bool:
        if len(s)%2==1:
            return False
        pairs={')':'(','}':'{',"]":'['}
        stack=list()
        for cha in s:
            if cha in pairs:
                if not stack or stack[-1]!=pairs[cha]:
                    return False
                stack.pop()
            else:
                stack.append(cha)
        return not stack
```

```python
import bisect
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        # 从列表的末尾开始向前查找，寻找第一个降序的位置
        index = len(nums) - 1  # 初始化 index 为最后一个元素的索引
        # 寻找第一个满足 nums[index] > nums[index-1] 的位置
        while index > 0:
            if nums[index] > nums[index - 1]:  # 找到一个降序对
                break
            index -= 1  # 如果没有找到，继续向前检查
        # 如果找到了这样的降序对 (index > 0)
        if index > 0:
            # 对降序对之后的所有元素进行排序，使之成为最小排列
            nums[index:] = sorted(nums[index:])  # 排序子列表
            # 使用 bisect.bisect 查找比 nums[index-1] 稍大的最小元素的位置
            # 注意，bisect.bisect 实际上调用的是 bisect_right，它返回的是右侧插入点
            swap_index = bisect.bisect(nums, nums[index - 1], lo=index)
            # 交换这两个元素，以形成下一个排列
            nums[swap_index], nums[index - 1] = nums[index - 1], nums[swap_index]
        else:
            # 如果整个数组都是非递增的（即没有找到任何降序对），则直接反转得到最小排列
            nums.reverse()
```

括号匹配，比较简单栈　　使用 bisect 求下一排列

```python
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        ans = defaultdict(list)
        for s in strs:
            key = "".join(sorted(s))
            ans[key].append(s)
        return list(ans.values())
```

异位词分组,defaultdict 的使用

```python
n=int(input())
trees=[[int(x) for x in input().split()]for _ in range(n)]
count=2
if n==1:
    print(1)
else:
    for i in range(1,n-1):
        if trees[i][0]-trees[i-1][0]>trees[i][1]:
            count+=1
        elif trees[i+1][0]-trees[i][0]>trees[i][1]:
            count+=1
            trees[i][0]+=trees[i][1]
    print(count)
```

砍树问题，涉及相邻数据的处理

```python
n = 0
while True:
    a, b, c, d = map(int, input().split())
    n += 1
    if a + b + c + d >= 0:
        found = False
        for i in range(1, 21253):
            if (i - a) % 23 == 0 and (i - b) % 28 == 0 and (i - c) % 33 == 0:
                if i - d >= 0:
                    print(f"Case {n}: the next triple peak occurs in {i - d} days.")
                else:
                    print(f"Case {n}: the next triple peak occurs in {i - d + 21252} days.")
                found = True
                break
        if not found:
            print(f"Case {n}: No valid solution found within the range.")
    else:
        break
```

计算生理周期，解决下一次几数相聚的问题

```python
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        ans = 0
        nums = set(nums)
        for x in nums:
            if x - 1 in nums: continue # 如果num-1在nums中，那么以num-1为起点的序列一定比以num为1的序列长
            nxt = x + 1
            while nxt in nums:
                nxt += 1

            ans = max(ans, nxt-x) # [num, nxt-1] 这里-1是因为nxt上来先加了1
        return ans
```

寻找最长连续字符串的集合写法，以及通过-1 判断是否重复的思路

```
while True:
    try:
        s = input()
    except EOFError:
        break
    if s.count('@') != 1:
        print("NO")
        continue
    if (s[0]=='@' or s[-1]=='@' or s[0]=='.' or s[-1]=='.'):
        print("NO")
        continue
    if (s.find("@.") != -1 or s.find(".@") != -1):
        print("NO")
        continue
    p = s.find("@")
    q = s.find( __sub: ".", p + 1)

    print('NO' if q == -1 else 'YES')
```

邮箱验证里面有 find 的使用

```
result=[]
pigs=[]#主栈
min_pigs=[]
while True:
    try:
        command=input().strip()
        if command=='':
            break
        if command=='pop':
            if pigs: #主栈非空，自动排除了没有猪的情况
                removed=pigs.pop()
                if removed==min_pigs[-1]:
                    min_pigs.pop() #有可能是最小的猪一起被移走
        elif command.startswith('push'):
            _,weight=command.split()
            weight=int(weight)
            pigs.append(weight)
            if not min_pigs or weight<=min_pigs[-1]:
                min_pigs.append(weight)#且要保证最小的在顶部，这样一拿就可以拿出来
        elif command=='min':
            if min_pigs:
                result.append(min_pigs[-1])
            else:
                pass
    except EOFError:
        break
for i in result:
    print(i)
```

快速堆猪使用辅助栈

```
from collections import deque

def main():
    n = int(input())  # 读取整数个数
    q = deque(map(int, input().split()))  # 将输入的整数入队

    while len(q) > 1:  # 当队列中元素数量大于1时，继续操作
        front1 = q.popleft()  # 取出队列的第一个元素
        front2 = q.popleft()  # 取出队列的第二个元素
        q.append(front1 + front2)  # 将两个元素的和重新入队

    print(q[0])  # 输出队列中剩下的唯一元素
```

```
def main():
    n = int(input())  # 读取队列长度
    q1 = deque(map(int, input().split()))  # 初始化队列 q1
    q2 = deque(map(int, input().split()))  # 初始化队列 q2

    counter = 0  # 初始化操作轮数计数器

    while q1:  # 当 q1 不为空时继续操作
        if q1[0] == q2[0]:  # 如果队首元素相同
            q1.popleft()  # 出队 q1 的队首元素
            q2.popleft()  # 出队 q2 的队首元素
        else:
            q2.append(q2.popleft())  # 将 q2 的队首元素移至队尾

        counter += 1  # 增加操作轮数计数器

    print(counter)  # 输出操作的总轮数
```

求和队列问题以及两队列的匹配问题