Cheating sheet

2300011505 钟明衡

基本运算

python不需要提前声明变量类型，但是变量操作要注意类型。变量类型也可以自己声明，不过暂时用不到。

单个的变量有字符串、整数、浮点数、布尔类型等，可以转换（注意字符串不能随意转换成整数、浮点数，以及浮点精度问题）：

```
s1 = '100'
a = int(s1)    # 此时a=100
s2 = '3.14'
b = float(s2)  # 此时b=3.14
```

多个的变量有元组、列表、字典、集合，元组只能整个赋值不能修改，列表和字典可以修改，集合可以加减元素：

```
t = (1, 2, 3)                 # 元组tuple（t[2]为3）
l = [1, 2, 3]                 # 列表list（l[2]为3）
d = {1: 'a', 2: 'b', 3: 'c'}  # 字典dict（d[2]为'b'）
s=set()                       # 集合set
```

基本运算：

```
a, b = 5, 2       # 赋值
print(a+b)        # 加法，7
print(a-b)        # 减法，3
print(a*b)        # 乘法，10
print(a/b)        # 除法，2.5
print(a**b)       # 乘方，25
print(a % b)      # 取余，1
print(a//b)       # 整除，2
print(abs(b-a))   # 绝对值，3
a, b = b, a       # 交换
print(min(a, b)) # 最小值，2
print(max(a, b)) # 最大值，5
```

条件语句和布尔运算

if elif else没什么好说的，写的时候思路放清晰些，别搞错了关系。

有些东西可以直接转换成布尔值：

```
if False or '' or 0 or 0.00 or [] or () or {}:
    print('Hello World!')
# 什么都没有输出，上面的这些值全为False
```

常用的布尔运算：

```
if False and True:  # and：全为True则True，否则False
    print(1)
if False or True:   # or：有True则True，否则False
    print(2)
if not False:       # not：否
    print(3)
# 输出2 3
```

字符串操作

字符串加法、乘法是直接拼接：

```
s1, s2 = 'abc', 'def'
s3 = s1+s2  # s3为'abcdef'
s4 = s1*3   # s4为'abcabcabc'
```

遍历字符串，是遍历每个字符：

```
s = 'abcdef'
for a in s:
    print(a)
# 输出一列abcdef
```

单个字符可以和ASCII码互相转换（'0'：48，'A'：65，'a'：97）：

```
print(ord('A'))  # 输出65
print(chr(97))   # 输出a
```

字符串大小写和其他变化：

```
a = 'Hello World, Hello Python!'  # 下面的操作都不会改变a，但是会有返回值
print(a.capitalize())  # 完全按照英文大写规则：Hello world, hello python!
print(a.title())       # 所有单词首字母大写：Hello World, Hello Python!
print(a.lower())       # 全部小写：hello world, hello python!
print(a.upper())       # 全部大写：HELLO WORLD, HELLO PYTHON!
print(a.split(','))    # 从','拆分：['Hello World', ' Hello Python!']
```

一般来讲，字符串可以看作一个列表，元素为每一个字符，区别在于字符串不能任意修改某个字符。

列表

列表加法、乘法也是直接拼接（注意时间复杂度为$O(n_1+n_2)$）：

```
l1, l2 = [1, 2, 3], [4, 5, 6]
l3 = l1+l2  # l3为[1,2,3,4,5,6]
l4 = l1*3   # l4为[1,2,3,1,2,3,1,2,3]
```

列表的常见操作如下：

```
l = [1, 2, 3]
# 创建列表，l=[1,2,3]，列表的元素可以是任何东西，列表也可以
a, b, c, d = len(l), sum(l), min(l), max(l)
# 分别为列表长度、和、最小值、最大值，时间复杂度都是O(1)
l.append(4)
# 在列表末尾添加一个元素，时间复杂度为O(1)，l=[1,2,3,4]
l.pop()
# 弹出列表的最后一个元素，时间复杂度为O(1)，l=[1,2,3]
a = l.pop(0)
# 弹出列表的指定位置的元素，时间复杂度为O(n)，弹出操作是有返回值的，l[2,3]，a=1
l.sort()
# 排序，时间复杂度为O(nlogn)，注意tuple是不让排序的，l=[2,3]
l = sorted(l, reverse=True)
# 也是排序，时间复杂度为O(nlogn)，返回值为排序好的列表，l=[3,2]
l.insert(1, 7)
# 在指定索引处插入一个元素，时间复杂度为O(n)，插完以后那个元素的索引就是指定的那个，l=[3,7,2]
a = l.index(7)
# 返回首个值为指定元素的索引，时间复杂度为O(n)，a=1
l.remove(3)
# 删除首个值为指定值的元素，时间复杂度为O(n)，l=[7,2]
del l[0]
# 删除指定位置的元素，和pop差不多，但是没返回值
a = l[-1]
# 负数索引代表倒数第几个元素
l = [1, 2, 3, 4, 5, 6, 7]
# 切片操作，时间复杂度为O(n)，起始位置（默认0）会包括，终止位置（默认len(l)）不包括
l = l[1:]   # 切片，l=[2,3,4,5,6,7]
l = l[:-1]  # 切片，l=[2,3,4,5,6]
l = l[1:4]  # 切片，l=[3,4,5]
```

创建多维列表，注意避免浅拷贝：

```
l = [0]*5                       # 创建一个全为0的列表[0,0,0,0,0]
l = [i for i in range(5)]       # 创建[0,1,2,3,4]
M = [[0]*5 for i in range(4)]   # 创建一个4行5列的二维列表
```

完全复制一个列表，要用深拷贝，否则会修改同一个列表：

```
from copy import deepcopy
l1 = [1, 2, 3, 4, 5]
l2 = deepcopy(l1)
```

列表可以压缩：

```python
a = [1, 2, 3]
b = ['c', 'a', 'b']
zipped = list(zip(a, b))
print(zipped)  # 输出[(1, 'c'), (2, 'a'), (3, 'b')]
c, d = zip(*sorted(zipped, key=lambda x: x[1]))
print(c)  # 输出(2, 3, 1)
print(d)  # 输出('a', 'b', 'c')
# 注意解压缩后得到的是元组
# 如果压缩的列表不一样长，长的那个后面的部分会被截掉
```

字典

遍历字典有以下方法：

```python
d = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
print(d.items())   # [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
print(d.keys())    # [1, 2, 3, 4]
print(d.values())  # ['a', 'b', 'c', 'd']
```

除了传统字典，collections库中还有以下字典：

空字典defaultdict，如果key不存在就会报错。下面生成的字典，如果key不存在，默认值为0：

```python
from collections import defaultdict
dic = defaultdict(int)
```

计数器Counter，记录列表中每个元素出现了几次，默认按出现次数从大到小排序：

```python
from collections import Counter
l = [1, 2, 3, 2, 1, 2, 3, 4, 5, 6]
print(Counter(l))
# 输出Counter({2: 3, 1: 2, 3: 2, 4: 1, 5: 1, 6: 1})
```

有序字典Ordereddict，按插入顺序排序的字典（就相当于items是一个列表）：

```python
from collections import OrderedDict
d = OrderedDict([(1, 'a'), (2, 'b'), (3, 'c')])
d.move_to_end(1)  # 把指定元素移到末尾
d.popitem()       # 弹出末尾元素
# 此时d={2:'b',3:'c'}
```

集合

集合中的元素不重复，无序，操作如下：

```python
l = set([1, 2, 3, 4])  # 可从列表转换
l.add(6)               # 加入元素
l.add(6)
l.remove(1)            # 删除元素
print(l)               # 输出{2, 3, 4, 6}
print(3 in l)          # 3在，输出True
```

集合的in操作时间复杂度为$O(1)$，而列表中则为$O(n)$

堆

堆可以以$O(\log n)$的时间复杂度插入数据，并且以$O(1)$的时间复杂度弹出最小值（加负号入堆就是最大值了）。

堆一开始可以是一个空列表，或者将列表变为堆，例子如下：

```python
from heapq import heapify, heappop, heappush
l = [5, 4, 3, 1]
heapify(l)          # 列表可以转化为堆
print(heappop(l))   # 弹出最小值1，有返回值，也可以单独用
heappush(l, 2)      # 推入2
print(l[0])         # 输出最小值2
```

数组

数组array是只能有一种数据类型的列表，更省内存，但不够灵活：

```python
import array
l = array('i', [1, 2, 3, 4, 5])
```

类型代码如下：

- `'b'`: 有符号字节（signed byte）
- `'B'`: 无符号字节（unsigned byte）
- `'h'`: 有符号短整型（signed short）
- `'H'`: 无符号短整型（unsigned short）
- `'i'`: 有符号整型（signed int）
- `'I'`: 无符号整型（unsigned int）
- `'l'`: 有符号长整型（signed long）
- `'L'`: 无符号长整型（unsigned long）
- `'f'`: 单精度浮点数（float）
- `'d'`: 双精度浮点数（double）

数组和列表的其他操作完全一致。

双向队列

双向队列deque可以以$O(1)$复杂度操作头尾的元素，但是查询中间的元素复杂度为$O(n)$：

```
from collections import deque
l = deque([2, 3, 4, 5])
l.append(6)          # 右插入6，此时l=[2,3,4,5,6]
l.appendleft(1)      # 左插入1，此时l=[1,2,3,4,5,6]
l.pop()              # 右弹出，可以有返回值
print(l.popleft())   # 左弹出，输出1
```

接收数据

python的数据是按行接收的，默认读取一行为一个字符串，可以转换为其他类型：

```
s = input()          # 输入一行一个字符串
n = int(input())     # 输入一行一个整数
a = float(input())   # 输入一行一个浮点数
```

可以按照某个字符串来拆分，拆分的结果为一个列表，其中每个元素为字符串：

```
l1 = input().split()      # 默认按空格拆分
l2 = input().split(',')   # 按照指定的','拆分
```

使用map来对每个元素进行操作：

```
a, b = map(int, input().split())      # 输入一行两个整数
l = list(map(int, input().split()))   # 输入一行任意多个整数
```

读取一个m行n列的矩阵：

```
m, n = map(int, input().split())
M = []
for i in range(m):
    M.append(list(map(int, input().split())))
```

有些问题加保护圈可以降低读取难度，下面是一个加一圈0的例子：

```
m, n = map(int, input().split())
M = [[0]*(n+2)]
for i in range(m):
    M.append([0]+list(map(int, input().split()))+[0])
M.append([0]*(n+2))
```

输入t组数据的模板：

```
for _ in range(int(input())):
    solve()
```

可以用列表把答案存起来一起输出，会快些：

```
t = int(input())
ans = [0]*t
for _ in range(t):
    solve()
for a in ans:
    print(a)
```

以某种方式结束输入（以输入0为例）：

```
while True:
    n = int(input())
    if n == 0:
        break
    solve()
```

自动结束输入：

```
while True:
    try:
        n = int(input())
        solve()
    except EOFError:
        break
```

输出

直接用print，会输出原本的格式：

```
a, s, l = 1, 'aaa', [1, 2, 3]
print(a)  # 输出1
print(s)  # 输出aaa
print(l)  # 输出[1, 2, 3]
```

end控制输出的末尾，默认为一个换行符：

```
print(1, end=' ')
print(2)
'''输出结果为：
1 2

'''
```

join会把字符串或者全是字符串的列表用指定的字符串隔开，如果要用空格隔开整数，记得换成字符串，或者用for循环：

```
s, l = '12345', ['1', '23', '4', '5']
print(' '.join(s))            # 输出1 2 3 4 5
print(','.join(l))            # 输出1,23,4,5
l = [1, 2, 3, 4, 5]
print(' '.join(map(str, l)))  # 输出1 2 3 4 5
for i in range(len(l)-1):
    print(l[i], end=' ')
print(l[-1])                  # 输出1 2 3 4 5
```

用%来代表特定格式：

```
a, b, s = 100, 3.14159, 'abcd'
print('a=%d,s=%s' % (a, s))  # 输出a=100,s=abcd
print('%5d' % a)             # 输出   100，这里%5d表示五位整数，不足的位数在前面用空格补齐
print('%.2f' % b)            # 输出3.14，这里%.2f表示保留两位小数，四舍五入
```

特别地，保留小数位数的方法如下：

```
from math import ceil, floor
a, b = 6.65, 6.75
print('%d %d' % (int(a), int(b)))              # 直接变整数输出，会抹掉小数部分：6 6
print('%.1f %.1f' % (a, b))                    # %.nf保留n位小数，四舍五入：6.7 6.8
print(str(round(a, 1))+' '+str(round(b, 1)))  # round很迷惑，少用（WA了可以试试）：6.7 7.8
print('%d %d' % (ceil(a), floor(a)))           # 分别是向上、向下取整：7 6
```

循环

range是一个按照起始、终止、步长生成的等差数列，第一个元是起始值，终止值一定不包括在内：

```
l=[i for i in range(5)]          # 0,1,2,3,4
l=[i for i in range(1, 11, 2)]   # 1,3,5,7,9
l=[i for i in range(12, 0, -3)]  # 12,9,6,3
```

for循环遍历列表中的每一个元素：

```
l = [1, 2, 3, 4, 5]
for i in l:
    print(i)
# 输出一列1 2 3 4 5
```

配合enumerate，可以加上索引（不过感觉有点多余）：

```python
l = ['a', 'b', 'c', 'd', 'e']
for i, a in enumerate(l):
    print('l['+str(i)+'] = '+a)
# 输出一列l[0]=a l[1]=b……
```

while循环首先检查条件是否满足，如果满足那就执行后面的语句：

```python
i = 0
while i < 5:
    i += 1
# 退出后i=5
```

continue直接跳到下一个循环开始，break直接跳出当前循环（只跳一层，跳多层可以用flag），它们后面的语句都不会执行。

进制转换和二进制运算

直接用内置函数就可以实现进制转换：

```python
n = 2024
print(bin(n))  # 转为二进制，输出0b11111101000
print(oct(n))  # 转为八进制，输出0o3750
print(hex(n))  # 转为十六进制，输出0x7e8（字母默认小写）
# 输出结果为字符串，前两位代表了进制
s1, s2 = '10101', 'aBc'
print(int(s1, 2))   # 从二进制转为十进制，输出21
print(int(s2, 16))  # 从十六进制转为十进制，输出2748（字母大小写不影响）
```

下面是各种二进制运算：

```python
a = 0b1101
b = 0b1010
print(bin(a & b))   # 与AND，都为1则为1，否则为0，输出0b1000
print(bin(a | b))   # 或OR，都为0则为0，否则为1，输出0b1111
print(bin(a ^ b))   # 异或XOR，相同为0，不同为1，输出0b111
print(bin(~a))      # 取反NOT，变符号，每一位取反，输出-0b1110
print(bin(a << 2))  # 左移，相当于乘以2的n次方，右边空位根据正负用0或1补齐，输出0b110100
print(bin(a >> 1))  # 右移，相当于除以2的n次方，左边空位根据正负用0或1补齐，输出0b110
# 其他进制的数也可以这样计算，会变成二进制计算完以后变回去
```

库

用import调用库：

```python
import math
print(math.sqrt(100))  # 输出10.0
```

使用from，则不需要打点，且省操作：

```python
from math import sqrt
print(sqrt(100))  # 输出10.0
```

下面是math库中常用的函数（优先用内置函数，更快）：

```python
import math
print(math.pow(10,2))       # 幂，输出100.0
print(math.sqrt(100))       # 开方，输出10.0
print(math.factorial(5))    # 阶乘，输出120
print(math.floor(10.2))     # 向下取整，输出10
print(math.ceil(10.2))      # 向上取整，输出11
print(math.gcd(60, 15))     # 最大公约数，输出15
print(math.log(1000, 10))   # 对数，输出3
print(math.cos(math.pi))    # 三角函数、常数派，输出-1.0
print(math.e)               # 自然常数，输出2.718281828459045
print(math.comb(5, 2))      # C(5,2)，输出10
```

自定义函数

使用lambda可以定义匿名函数，在key中很常用：

```python
l = [(1, 4), (3, 6), (5, 2)]
print(max(l, key=lambda x: x[0]+x[1]))  # 输出和最大的一组
```

使用def来自定义函数，函数可以返回值，也可以不返回值。返回后，下面的语句不会执行。

下面是一个不返回值的函数：

```python
def solve():
    a, b = map(int, input().split())
    print(a+b)
    return
```

下面是一个返回值的函数：

```python
def add(a, b):
    return a+b
```

注意局部变量在返回以后不会变，而全局变量会变：

```python
def change(a):
    global b
    a += 1
    b += 1
    return


a, b = 1, 1
change(a)
print('%d %d' % (a, b))  # 输出1 2，a没变但b变了
```

函数可以调用自己，也可以返回自己，这在之后的递归中会用到。有内置函数就尽量用，会更快。

质数筛

下面是一个最快的质数筛（欧拉筛）：

```python
n = int(input())
is_prime = [True] * (n + 1)
primes = []
for i in range(2, n + 1):
    if is_prime[i]:
        primes.append(i)
        for j in range(i * i, n + 1, i):
            is_prime[j] = False
```

排序

直接用内置函数就可以排序，默认从小到大，如果加reverse那就是从大到小：

```python
l = [4, 3, 1, 2, 5]
l.sort()                          # 此时l=[1,2,3,4,5]
print(sorted(l, reverse=True))  # l=[5,4,3,2,1]
```

sort、min、max都可以加key，即判断大小的标准：

```python
d = {2: 'two', 4: 'four', 1: 'one', 3: 'three'}
d = dict(sorted(d.items(), key=lambda x: x[0]))
print(d)  # 输出{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

手写的快速排序：

```python
def qsort(l):
    if len(l) <= 1:
        return l
    m = l[len(l)//2]
    left, middle, right = [], [], []
    for i in l:
        if i < m:
            left.append(i)
        elif i > m:
            right.append(i)
        else:
            middle.append(i)
    return qsort(left)+middle+qsort(right)
```

暴力枚举

直接把每一种情况都试一次，就是枚举。基本要求是不要漏掉情况。

枚举很容易超时，如果不是万不得已尽量别用。另外，有些方法可以剪枝，比如排除掉绝对不可能的情况，或者利用各变量之间的不独立性来减少循环。

贪心

贪心是纯数学问题，找最优策略主要有两种思路：

1. 最优（直接找到最优的方法）
2. 不减（按规则进行任意的操作，收益永不减少）

递归

递归就是要找通项公式，然后用循环或者函数都可以实现。找通项公式很难，一般是一个贪心，一旦找到了问题就会很简单。

递归层数太多可能会RE，下面是修改（解除）最大层数限制的方法（改完以后很容易TLE）：

```python
from sys import setrecursionlimit
setrecursionlimit(None)  # 接触最大层数限制
```

二分查找

有二分查找的内置库（注意，必须是严格升序列表才能用）：

```python
from bisect import bisect
l = [1, 3, 3, 3, 5, 7, 9]
print(bisect(l, 3))   # 输出4
print(bisect(l, 4))   # 输出4
print(bisect(l, 0))   # 输出0
print(bisect(l, 10))  # 输出7
# 可以发现，bisect会找到尽量右的位置，在该位置插入数后仍保持升序
```

左右是可以换的：

```
from bisect import bisect_right, bisect_left
l = [1, 3, 3, 3, 5, 7, 9]
print(bisect_left(l, 3))   # 输出1
print(bisect_right(l, 3))  # 输出4
# left尽量左，right尽量右
```

升序列表，可以用下面的方法检查一个元素是否存在，复杂度$O(\log n)$：

```
from bisect import bisect


def check(l, a):
    if not l:  # 记得排除空列表情况
        return False
    return l[bisect(l, a)-1] == a


l = [1, 2, 4, 5, 6]
print(check(l, 4))  # 输出True
print(check(l, 3))  # 输出No
```

手写的二分查找：

```
def find(l, a):
    if not l:
        return -1
    left, right = -1, len(l)
    while right - left >= 2:
        middle = (left+right)//2
        if l[middle] <= a:
            left = middle
        else:
            right = middle
    if l[left] == a:
        return left
    else:
        return -1


l = [1, 2, 3, 3, 3, 3, 4, 5]
print(find(l, 3))  # 输出5
# 这个程序会找到最右的位置，如果不存在就输出-1，交换left和right的更新顺序即可找最左
```

深度优先搜索

下面是一个走迷宫的例子，计算有多少条路径：

```
# 路0，终点1，墙2
dx, dy = [1, 0, -1, 0], [0, 1, 0, -1]  # 向各个方向走
ans = 0


def dfs(x, y, M, used):
    global ans
    if M[y][x] == 1:  # 走到终点，ans+1
        ans += 1
        return
    for i in range(4):  # 四个方向走
        newx = x+dx[i]
        newy = y+dy[i]
        if not used[newy][newx] and M[newy][newx] != 2:  # 可以走
            used[newy][newx] = True  # 上标记
            dfs(newx, newy, M, used)  # 走下一步
            used[newy][newx] = False  # 取消标记


m, n = map(int, input().split())
M = [[2]*(n+2)]  # 加一圈2的保护圈
for i in range(m):
    M.append([2]+list(map(int, input().split()))+[2])
M.append([2]*(n+2))
used = [[False]*(n+2) for i in range(m+2)]  # 标记走过的位置
used[1][1] = True
dfs(1, 1, M, used)
print(ans)
```

深搜其实就是暴力枚举，数据不是很小就别用。注意标记走过的路径。

广度优先搜索

下面是一个走迷宫的例子，计算最短路径：

```
# 路0，终点1，墙2
from sys import exit

m, n = map(int, input().split())
M = [[2]*(n+2)]  # 加一圈2的保护圈
for i in range(m):
    M.append([2]+list(map(int, input().split()))+[2])
M.append([2]*(n+2))
if M[1][1] == 1:  # 初始就是终点，直接结束
    print(0)
    exit()
step, start, end = 0, 0, 0
```

```
x, y = [1], [1]  # 存储要走的位置
used = [[False]*(n+2) for i in range(m+2)]  # 标记走过的位置
used[1][1] = True
dx, dy = [1, 0, -1, 0], [0, 1, 0, -1]  # 向各个方向走
while end != len(x):  # 双指针，分别指向当前步数开始和结束位置
    step += 1  # 走一步
    start, end = end, len(x)  # 更新指针
    for i in range(start, end):
        for j in range(4):  # 四个方向走
            newx = x[i]+dx[j]
            newy = y[i]+dy[j]
            if M[newy][newx] == 1:  # 走到终点，直接结束
                print(step)
                exit()
            elif not used[newy][newx] and M[newy][newx] == 0:  # 不是终点但是可以走，加入到下一步要走的位置
                x.append(newx)
                y.append(newy)
                used[newy][newx] = True
print('NO')  # 走不到终点
```

广搜有很多地方可以修改，比如“广度”的含义、一个位置什么时候才上标记，等等。

kmp算法

下面是一个标准的kmp算法程序：

```
def buildnext(pat):
    next, l, r = [0], 0, 0
    while r < len(pat):
        if pat[l] == pat[r]:
            l += 1
            r += 1
            next.append(l)
        elif l != 0:
            l = next[l-1]
        else:
            r += 1
            next.append(0)
    return next


def search(txt, pat, next):
    ans, i, j = [], 0, 0
    while i < len(txt):
        if txt[i] == pat[j]:
            i += 1
            j += 1
        elif j != 0:
            j = next[j-1]
        else:
```

```
            i += 1
        if j == len(pat):
            ans.append(i-j)
            j = next[j-1]
    return ans


txt, pat = input().split()  # 在txt中寻找pat出现的位置
ans = search(txt, pat, buildnext(pat))
if ans:
    print(' '.join(map(str, ans)))
else:
    print('NO')
```

其他技巧

加个负号，就可以把最大问题转换成最小问题

可以通过数据量来反推允许的的最大时间复杂度：

$10^9 - O(n)$

$10^5 - O(n \log n)$

$10^3 - O(n^2)$

如果一个问题正着很难，可以试着反过来

变量名不要取重了

未通过的应对方法

CE：看报错的内容，针对性地修改

WA：改细节，注意特殊情况的判别，有可能要改算法

MLE/TLE：加强剪枝，不过大概率要换算法

RE：指针越界，或者递归层数太多，前者很好改，后者可能要改递归层数或者改算法